S POETRY

Patrick J. Burns

Contents

List of Tables

Chapter 1

Essentials

Five great strengths of S are its variety of objects, its vector orientation, the power of subscripting, object-oriented programming and graphics. Each of these is explored before turning to other matters.

A powerful and fairly unique part of S that will be addressed in later chapters is the possibility of computing on the language. Essentially everything in S—for instance, a call to a function—is an S object. One viewpoint is that S has self-knowledge. This self-awareness makes a lot of things possible in S that are not in other languages.

Of fundamental importance is that S is a *language*. This makes S much more useful than if it were merely a "package" for statistics or graphics or mathematics. Imagine if English were not a language, but merely a collection of words that could only be used individually—a package. Then what is expressed in this sentence is far more complex than any meaning that could be expressed with the English package.

1.1 Objects

I briefly outline the important types of objects that exist in S. All objects possess a *length* and a *mode*. While the length of an object is generally straightforward, the mode is more arbitrary. The concept of mode should become clear as examples unfold.

The most basic object is an *atomic vector*—often merely called a vector. The atomic modes to speak of are—numeric, logical, character and complex. If an object is atomic, then each *element* has the same mode. There is no such thing as an atomic vector that contains both numbers and logicals.

The length of a vector is the number of elements it contains. For example, the length of a numeric or complex vector says how many numbers are contained in the object.

Although S does distinguish between double-precision floating-point numbers, single-precision floating-point numbers and integers, S considers this a detail that should not concern the user. Only when you are interfacing to C or Fortran is it an issue. In S there is no difference between writing 1 and 1.0.

In addition to the usual numbers, numeric objects may contain `NA`s and `Inf`s. There are two flavors of `NA`, the usual type means "missing value". The other `NA` is `NaN`, meaning "Not-a-Number"—these occur through mathematical operations such as zero divided by zero, and infinity minus infinity. You can distinguish `NaN`s from ordinary `NA`s with the `is.nan` function. An `Inf` is an infinite value and, for numeric data, may be either positive or negative. You may initially think that these *special values* are burdensome, but in fact they eliminate a great deal of bother.

Objects of mode `complex` contain complex numbers. The imaginary part of a complex number is written with an "`i`" at the end. An example of a complex vector of length 2 is:

```
c(1+3i, 2.4-8.7i)
```

Complex vectors have special values analogous to those for numeric objects.

There are three logical values: `TRUE`, `FALSE` and `NA`. `TRUE` is also written `T`, and similarly, `FALSE` can be written `F`.

Each element of a vector of mode `character` is a character string. Each string contains an arbitrary number of characters. There is not a missing value for mode character—often the empty string is used where a missing value is fitting. The backslash is a special character—rather than meaning "backslash", it is the escape. For example, `"\\"` means "backslash" and `"\n"` means "newline". Character strings are surrounded by either double quotes or single quotes. If double quotes are used, then a double quote within the string needs a backslash in front of it. Likewise, a single quote needs to be escaped if the string is delimited by single quotes. Double quotes are always used when character vectors are printed.

```
> c("single ' quote", 'double " quote',
+ "double \" quote", 'single \' quote')
[1] "single ' quote"  "double \" quote"
[3] "double \" quote" "single ' quote"
```

There are a few other characters that are formed by following a backslash with another character. A backslash followed by a triple of octal digits gives the corresponding ASCII code.

There is one more object that is classified as atomic—`NULL`. Just as zero is

a very important number, an object that stands for "nothing" is a powerful instrument.

Since everything in an atomic vector must be of one type, they are pleasantly simple, but limited in applicability. To put various types into a single object, you need a *list*. Lists are objects of mode `list` and the length of a list is the number of *components* it contains. Each component of a list is another S object. For example, the first component may be a numeric vector of length 20, the second component a character vector of length 3, and the third component a list of length 24. (Note that Becker, Chambers and Wilks (1988, p112) use the word "component" only when it is named, but I use it no matter what.)

The availability of lists in S is a very powerful feature—it sets it apart from much other software. One more thing allows an even richer set of objects in S.

Each object may have a set of *attributes*. The attributes of an object is a list in which each component has a name. You can imagine each S object having a hook where the attributes list hangs—some objects have nothing there, others have quite involved lists there.

S understands some attributes internally—the *names* attribute is an example. The names of an object is a vector of character strings that has the same length as the object. Suppose we issue the command:

```
jjcw <- c(stevie=4, munchkin=2)
```

The object named `jjcw` is a numeric vector of length 2. Its attributes are a list with 1 component named `"names"`, which is a character vector of length 2.

Arrays provide a good example of attributes. A *matrix*—a rectangular arrangement of elements—is a two-dimensional array. The S notion of *array* generalizes matrices from two dimensions to an arbitrary number of dimensions. An S array is merely a vector that has a `dim` attribute, and optionally a `dimnames` attribute. The `dim` is a vector of integers; the length of the `dim` is the dimension of the array, and the elements tell how many rows, columns, etc. are in the array. The product of the elements in the `dim` must equal the length of the object. Thus a matrix has a `dim` attribute of length 2. The `dimnames` provide names along each dimension.

By adding just the `dim` attribute, we get an object that has a quite different nature. You have the ability to give objects any attributes, so that they can take on whatever nature you choose. The `class` attribute is the granddaddy attribute of them all. This is what drives the object-oriented programming, which we will come to shortly. *The shapes a bright container can contain!* [1]

There are additional types of objects in S, many of which will be discussed later. But we already have all the basics—mode, length and attributes define an object.

Many objects are given names. Although there are ways of using almost any name, generally objects are given "valid" S names. A valid name contains only alphanumeric characters and periods. The first digit, if any, in the name must be preceded by an alphabetic character. Capital letters are distinct from lower-case characters. Assignment functions, like `dim<-` do not have "valid" names so special measures need to be used at times when working with them. See `valid.s.name` on page 279.

One particular object is `.Last.value` which is essentially the last value that was not given a name. For instance, if you decide that you should have assigned the last command to an object name, then you can do something like:

```
> jj <- .Last.value
```

1.2 Vectorization

Many S functions are *vectorized*, meaning that they operate on each of the elements of a vector. For example, the command `log(x)` returns a vector that is the same length as `x`; each element of the result is the natural logarithm of the corresponding element in `x`.

Vectorization tends to be frightening to new users, and taken for granted by experienced users. That users become unconscious of the fact that `log(x)` is shorthand for a whole series of operations is testimony to the power of vectorization. *Drinking my juices up* [2]

The arithmetic operators in S are vectorized. If `x` and `y` are the same length, then `x+y` returns a vector in which each element is the sum of the corresponding elements in `x` and `y`. Additionally the command `x+2` returns a vector as long as `x` containing two plus the corresponding element of `x`.

The main source of confusion is when the two vectors are not the same length and neither has length 1. The rule is that the shorter vector is replicated to be the length of the longer vector. Replication (as done by the `rep` function and implicitly done by the arithmetic operators) means to keep re-using the elements in order until the result is the proper length. The `x+2` example is really a case of this—the length one vector containing `2` is replicated to the length of `x`.

Consider the command:

```
x ^ (2:3)
```

If `x` is 1 long, then the result is length 2 with the first element being the square of the element in `x` and the second being its cube. If `x` has length 2, then the result is again length 2 with the first element being the square of the first element of `x` and the second element being the cube of the second element of `x`. More generally, suppose `x` has length $2n$. Then the result is the same length as `x` and element $2i$ of the result will be the cube of element $2i$ of `x`, and element

$2i-1$ of the result will be the square of element $2i-1$ of `x`. The same is true when `x` has length $2n+1$, but in this case you will also get a warning that the longer length is not an even multiple of the shorter length. Here is such a warning with a different command:

```
> 1:4 + 5:1
[1] 6 6 6 6 2
Warning messages:
  Length of longer object is not a multiple of the
        length of the shorter object in: 1:4 + 5:1
```

It is often the case that something is wrong when such a warning appears.

1.3 Subscripting

The term *subscripting* refers to the extraction or replacement of parts of objects. A first step in mastering S is to thoroughly understand subscripting.

There are a number of ways of subscripting. Each possibility is described below. If the expression that contains the subscripting is on the left-hand side of an assignment, then the values are replaced, otherwise they are merely extracted.

```
value <- x[sub]  # extraction
x[sub] <- value  # replacement
```

In what follows, I often speak in terms of extraction, but replacement is analogous (except where noted).

[with positive numbers

A single square bracket with positive numbers extracts the elements with indices equal to the numbers. For example, the command

```
state.name[1:5]
```

extracts the first five state names. The result is usually the length of the numeric vector that is inside the brackets. There is no constraint on the length of the subscripting vector or the number of times any particular index appears.

The subscripting numbers are coerced to be integer before subscripting is attempted. The actual rule on the length of the result is that it is the length of the integer subscripting vector after zeros have been removed. Elements that are `NA` or larger than the length of the vector being subscripted create `NA`s (or empty strings) in the result.

If you perform a replacement and a subscript is larger than the length of the subscripted vector, then the vector will be as long as the largest subscript and any elements not given a value will be NA. For example:

```
> jj <- 1:5
> jj[10] <- 34.4
> jj
 [1]  1.0  2.0  3.0  4.0  5.0   NA   NA   NA   NA 34.4
```

.

[with negative numbers

When the vector inside the square brackets contains negative numbers, then all but the specified elements are selected. The command:

```
state.name[-c(1, 50)]
```

returns the names of all of the states except the first and the $50th$.

Elements of the subscripting vector that are zero, repeated, or beyond the length of the vector are ignored. A missing value in the subscripting vector creates an error. If there is a mix of positive and negative subscripts, you will get an error.

[with characters

The subscripting vector can be character, in which case it corresponds to the names of the vector. You do not need the names to match completely, the strings in the subscripting vector only need enough of the first part of each name so that it is uniquely identified. It is a general feature of S that if there is a finite population of character strings from which to pick, then you can abbreviate:

```
> jjcw
 stevie munchkin
      4        2
> jjcw["stevie"]
 stevie
      4
> jjcw["st"]
 stevie
      4
```

DANGER. Be careful when performing a replacement. If you do not use the full name, then you will end up with both the original value and the new value with the shortened name:

```
> jjcw["st"] <- 6
> jjcw
 stevie munchkin st
      4        2  6
```

[with logicals

The command

```
x[x>0]
```

is an example of subscripting with a logical vector. When the subscripting vector is logical, it is implicitly replicated to be the same length as the vector being subscripted. The result of the subscripting operation will be as long as the number of `TRUE` and `NA` elements in the subscripting vector. An `NA` in the subscripting vector implies an `NA` in the result.

The naturalness and simplicity of this form of subscripting belie its great utility. Here are a few tragically inadequate examples:

```
> jjseq <- 1:100
> jjeven <- rep(c(F, T), length=100)
> jjthree <- rep(c(F, F, T), length=100)
> jjseq[jjeven]
 [1]   2   4   6   8  10  12  14  16  18  20  22  24
[13]  26  28  30  32  34  36  38  40  42  44  46  48
[25]  50  52  54  56  58  60  62  64  66  68  70  72
[37]  74  76  78  80  82  84  86  88  90  92  94  96
[49]  98 100
> jjseq[jjeven & jjthree] # divisible by 6
 [1]  6 12 18 24 30 36 42 48 54 60 66 72 78 84 90 96
> jjseq[jjeven | jjthree]
 [1]   2   3   4   6   8   9  10  12  14  15  16  18
[13]  20  21  22  24  26  27  28  30  32  33  34  36
[25]  38  39  40  42  44  45  46  48  50  51  52  54
[37]  56  57  58  60  62  63  64  66  68  69  70  72
[49]  74  75  76  78  80  81  82  84  86  87  88  90
[61]  92  93  94  96  98  99 100
```

DANGER. Here is a puzzle that you may run into at some point.

```
> jjwt2
 dorothy harold munchkin stevie
```

```
      14     15        2      4
> jjwt2[jjsubw] <- 21:22
Warning messages:
  Replacement length not a multiple of number of
        elements to replace in: jjwt2[jjsubw] <- 21:22
> length(jjwt2[jjsubw])
[1] 2
> length(21:22)
[1] 2
```

Both sides of the assignment are the same length, so we think that S must surely be going crazy. However, once we look at the subscripting vector, the situation starts to become clear.

```
> jjsubw
[1]  F  T NA  F
> jjwt2
 dorothy harold munchkin stevie
      14     21        2      4
```

The missing value in the subscripting vector counts in extraction, but not in replacement.

The `p.replace` function on page 73 contains examples that are perhaps more enlightening. The `match` function can be used to perform similar operations with subscripting—see page 71.

The $ operator

The `$` operator is the most common way to extract a component from a list. On the left is the name of the list, and on the right is the name of the component of interest. As in subscripting with character vectors, the component name only needs enough of the start to be uniquely identified.

```
> .Machine$double.eps
[1] 2.220446e-16
> .Machine$double.ep
[1] 2.220446e-16
> .Machine$double.e
NULL
```

You may be surprised about the result of the last command. There is more than one component of `.Machine` that partially matches `"double.e"`:

```
> names(.Machine)[grep("double.e", names(.Machine))]
[1] "double.eps"      "double.exponent"
```

In such a case, the match has to be exact or `NULL` is returned. It is of significance that an error does not occur.

DANGER. The caveat about abbreviating when doing a replacement applies here also. So don't do it.

```
> jj
$abc:
[1] 1 2 3 4 5 6 7 8 9

> jj$a
[1] 1 2 3 4 5 6 7 8 9
> jj$a <- 3456
> jj
$abc:
[1] 1 2 3 4 5 6 7 8 9

$a:
[1] 3456
```

[[

The `$` operator could be removed from the S language without reducing functionality. The `[[` operator does everything that `$` does and more. The statement `.Machine$double.eps` is precisely the same as `.Machine[["double.eps"]]`. The former is 5 characters shorter and is prettier—that's the entire difference.

If the component has no name, then `$` can not be used—you must use `[[` with a numeric index.

Also use `[[` if the subscripting object is the name of an object that contains what you want to subscript. The statement `x$compname` which is equivalent to `x[["compname"]]` is distinct from `x[[compname]]`. When the latter is the correct formulation, `compname` will be a length 1 vector that is either a positive integer or a character string of the (start of the) name of one of the components of `x`.

DANGER. Although I've presented the `[` operator as if it worked just on atomic vectors, it in fact works on lists (and lots of other objects) also. Beware the difference between `x[[1]]` and `x[1]`. The first gives you the first component of `x`, and the second gives you a list of length 1 whose component is the first component of `x`. Look at the difference:

```
> .Machine[[1]]
```

```
[1] 2.220446e-16
> .Machine[1]
$double.eps:
[1] 2.220446e-16
```

The `[[` operator always gets a single component (while `[` may get any number). A specialized use of `[[` allows the subscripting vector to have length more than 1, this is discussed on page 203.

Subscripted arrays

Arrays and data frames have special subscripting capabilities—each dimension may be subscripted independently. The subscripting vector for each dimension can be any of the forms given for `[` above. Note, though, that character subscripting vectors now refer to dimnames rather than names. An example is:

```
state.x77[1:5, c("Area", "Pop")]
```

If you subscript an array and one or more dimensions are only 1 long, then those dimensions are dropped. For example

```
state.x77[1:5, "Area"]
```

will be an ordinary vector and not a matrix. The array `iris` is three-dimensional, and

```
iris[1:2, 1:2, 4]
```

will be a matrix while

```
iris[1:2, 2, 4]
```

will just be a vector. If this is not the behavior you want, then use `drop=F` in the subscript. For example

```
iris[1:2, 2, 4, drop=F]
```

remains a three-dimensional array.

DANGER. Dimension dropping is a common source of intermittent bugs. Without the `drop=F` the code works fine as long as the array keeps the shape that the programmer expected, but there is trouble as soon as one of the subscripted dimensions has length 1. When subscripting an array in a function definition, it is good practice to explicitly specify whether `drop` should be `TRUE`

or `FALSE`.

When subscripting an array, you can always use just a single subscripting vector which treats the object as if it did not have dimensions. An array is a vector with the elements wrapped into the dimensions. A matrix has consecutive elements running down the columns. So if `x` is a matrix with 5 rows, then the expression `x[5:7]` picks out the 5th row of the first column and the first two rows of the second column . The first dimension of an array moves fastest and the last dimension moves slowest. You can see the pattern by issuing a command like:

```
array(1:24, 4:2)
```

This flexibility with subscripting can lead to bugs if commas are forgotten.

Data frames are really lists in which each component is the contents of a column, so subscripting a data frame with a single subscripting vector is thinking of the object as this list.

DANGER. Unlike ordinary vectors, arrays may not be assigned an out of bounds value. Compare:

```
> jj <- 1:3; jj[5] <- 9; jj
> > [1]  1  2  3 NA  9
> jjm <- as.matrix(1:3); jjm[5,1] <- 9; jjm
> Error in jjm[5, 1] <- 9: Array subscript (5) out of
        bounds, should be at most 3
Dumped
>      [,1]
[1,]    1
[2,]    2
[3,]    3
```

The first works while the second creates an error. A logical reason for this discrepancy is that increasing a dimension of an array is much more involved than increasing the length of a vector. If you want to make sure that assignments will be in bounds, you can coerce to an array.

[with a matrix

The methods of subscripting arrays that I have described have the constraint that the result also looks like an array. But we may want to select elements that

appear "at random". It is possible to think of the problem in terms of a single subscripting vector, but there may be information from the array that is useful.

Here's an example. Suppose that we want to replace the smallest value in each row of a matrix. So we know that we want to subscript once from each row, but there is no one column to pick.

The solution is to subscript with a matrix. A matrix can be used only as the subscripter of an array. The number of columns of the subscripting matrix must equal the number of dimensions in the array being subscripted, and the number of rows of the subscripting matrix is the number of elements being subscripted. The numbers in the subscripting matrix should be positive.

Here is a function that solves the smallest value replacement problem.

```
"fjjrowmin"<-
function(xmat, new=-Inf)
{
     rmin <- apply(xmat, 1, function(x) order(x)[1])
     xmat[cbind(1:nrow(xmat), rmin)] <- new
     xmat
}
```

In this function `rmin` has an element corresponding to each row of the input `xmat`. Each element of `rmin` is the position (column) within the row that contains the smallest value in that row of `xmat`. A two-column matrix is then created with `cbind`.

The `incidmat.mathgraph` function on page 312 provides another example of this form of subscripting.

DANGER. Many versions of S do not allow a subscripting matrix to be character. This is a bug (in my opinion) since character subscripts are equivalent to positive numeric subscripts.

Note that a logical matrix as a subscript is treated the same as if it were an ordinary vector. Otherwise

```
xmat[xmat < 0] <- NA
```

would not work in the useful way it does.

Empty subscripts

Subscripting vectors may be missing—meaning (perhaps ironically) everything is included as is. The most common use of this is when subscripting arrays. The command

```
state.x77[1:5, ]
```

returns the first 5 rows and all of the columns of `state.x77`.

An empty subscript can be useful for ordinary vectors also. The command

```
x[] <- 0
```

replaces each element of `x` with 0, but leaves its attributes (such as names) alone. Thus it is often different than

```
x <- rep(0, length(x))
```

DANGER. A missing subscripting vector is decidedly different than a zero length vector such as `NULL`. If the subscripting vector is `NULL` (actually atomic and zero length), then the result has zero length.

1.4 Object-Oriented Programming

The idea of object-oriented programming is simple, but carries a lot of weight. Here's the whole thing: if you told a group of people "dress for work", then you would expect each to put on clothes appropriate for that individual's job. Likewise it is possible for S objects to get dressed appropriately depending on what class of object they are.

When a data frame is printed, it looks like a matrix. The `print` function is using object-orientation to make that happen. Though the actual structure of a data frame and a matrix is very different, the concepts of data frame and matrix are almost identical. Thus it is appropriate that they look the same when printed.

Object-orientation simplifies. If you want to print an object, you don't need to find out what type of object it is, then try to remember the proper function to use on that type of object, then do it. You merely use `print` and the right thing happens.

It can also simplify programming. Programming is simplified for the reason above, plus it also suggests a proper ensemble of functions to write for a particular application. Furthermore, *inheritance* allows a very productive way to leverage existing code.

Version 4 of S has changed some of the mechanics of how object-orientation is achieved, but the version 3 approach works in version 4. What follows is how version 3 does it.

Some functions are *generic*—`print` is an example—meaning that the action depends on the type of object given as an argument. Generic functions have

one or more *methods*. A method is the function that is actually used when a generic function is called. It is the `"class"` attribute of an argument (usually the first argument) that determines which of the possible methods will be used.

If the object does not have a `"class"` attribute, then the default method is indicated. The class of an object is a character vector. Each element of the class is examined in turn until one matches a method. If no element of the class matches a method, then the default method is used. (Whenever the default method is called for and it does not exist, an error occurs.) In version 3 the name of a method is the name of the generic function followed by a period followed by the name of the class. So the print method for data frames (class `"data.frame"`) is `print.data.frame` and the default print method is `print.default`. Version 4 allows more flexible naming.

Inheritance results from a `class` attribute longer than 1. If an object's class has length 2, then the first class inherits from the second. Suppose that you want to create a class of object that looks like a data frame, but contains a specific type of data. Remember that the class vector goes from specific to general, so the class could be something like:

```
c("my.dframe", "data.frame")
```

You could write methods for some generic functions, perhaps `print` and `summary`, but not others that you plan to use, like `[` (subscripting), so that the data frame method is used. The `loan` function presented on page 226 is almost like this.

Inheritance should be based on similarity of the structure of the objects, not on conceptual similarity. For example, objects of class `"terms"` are digested formulas (class `"formula"`). It would be advantageous in spots if terms inherited from formulas since they are conceptually very similar. However, they are not structurally very similar at all, so the decision was that there be no inheritance.

A generic `print` threatens ill. Since what you see is not what you get, it means that OOP can stand for "Obfuscation-Oriented Programming" as well as "Object-Oriented Programming". The onus is on the programmer of the `print` method to eliminate as much confusion as possible. I think there is room for improvement in many of the print methods that now exist (including ones that I have written).

This leads to the issue of private versus public views. Data frames are a good example to take up. A data frame is implemented as a list with each component containing the contents of a column. Since it is a list, I can use `lapply` on it to apply a function to each column. The idea of a data frame is that it be a rectangular arrangement of elements in which the type of element in one column need not be the same as the elements in other columns. This is the public view—that it looks like a matrix. The private view is that it is implemented as a list. Data frames could be changed so that the public view remains the same, but their implementation is different so that my `lapply` trick would not work. Ideally the user of a class of objects (as opposed to the creator of the class) should be able to use only generic functions, and hence need no

information about the precise structure of the object, that is, of the private view.

S does not enforce object-orientation in any way. Objects that you create need not have a class attribute; and you can use a method function (with few exceptions) as an ordinary function on any objects you choose.

More on object-orientation starts on page 223.

1.5 Graphics

Graphics are an integral part of S. The topic deserves a book of its own, and is only skimmed in this one. Here I describe the basics of how graphics work in S.

Two principles that S graphics follow is that they are device independent, and that each graph can be built up with numerous commands. Device independence means that the same commands are used to create a graph on a PostScript printer as on a monitor running Motif—the only difference is what type of graphics device the user has set to listen for graphics commands. The ability to add to plots means that complex graphics can be created easily.

A *graphics device* is an S function that arranges for graphics to be rendered. For example, the `postscript` function makes it so that a file of PostScript commands will result when a graphics command is given. Graphics devices are put into one of three categories. Hard-copy devices are for creating a physical picture; the most common is the `postscript` device. Window devices produce graphics when you are running a window system on the network where S is running; the S-PLUS `motif` device is an example. Finally there are terminal devices that are used when S is run remotely. For instance, I can slouch in front of my Macintosh at home, telnet to a Unix machine where I run S, and get graphics on my Macintosh screen using `tek4105`, which performs Tektronics emulation. There should be a "Devices" help file that informs you of graphics devices available to you.

A graphics device has a state that is queried and modified by the `par` function. The state is described by a reasonably large number of *graphics parameters*. Examples of graphics parameters include `cex` (character expansion) which gives the size of the text relative to the standard font, and `mfcol` which controls the number of plots per page of output.

Graphics functions—functions that say what to draw, as opposed to graphics devices—are divided into high-level and low-level functions. High-level functions, like `plot` and `barplot`, create an entire new figure. Low-level functions (`points` and `lines`, for example) merely add to the existing figure. Some functions have an `add` argument so that they can function in either capacity. You can create your own high-level graphics function from scratch by starting with a call to `frame` and then using whatever low-level functions you like.

The surface of a device (to be concrete, think of a page of hard-copy output) is called a *graphics frame*, and contains a number of conceptual regions. Within

Operator	Associativity	Task
`$`	left to right	component subscript
`[ [[`	left to right	subscript
`^`	right to left	exponentiation
`-`	right to left	unary minus
`:`	none	sequence
`%% %`*whatever*`%`	left to right	in-built and user-defined
`* /`	left to right	multiply and divide
`+ -`	left to right	addition and subtraction
`== < > >= <= !=`	left to right	comparison
`!`	right to left	not
`& | && ||`	left to right	and, or
`~`	none	formula
`<- _ <<-`	right to left	assignment
`;`	left to right	statement end

Table 1.1: Precedence table for the S language, version 3.

the frame there are one or more *figures*. Surrounding the set of figures is the *outer margin*; this often contains zero area. Within each figure is a *plot area*. Surrounding the plot area within the figure is the *margin* for that figure. The plot area is where the points in a scatter plot go. The margin contains titles, tick labels and axis labels.

Some of the graphics functions are listed on page 113, and the `par` function is discussed briefly on page 48.

1.6 Precedence

As in many languages, operators in S obey precedence. The command

```
6 + 8 * 9
```

will perform the multiplication before the addition. If you want the addition done first, use parentheses:

```
(6 + 8) * 9
```

For the most part, the precedence in S is intuitive, the one I have the hardest time with is the `:` operator. See table 1.1.

When operators have equal precedence, almost all of them associate from left to right—that is, the leftmost operation is performed first. The exceptions are the assignment operators `<-` and `<<-` plus exponentiation with `^` which each associate from right to left.

Somewhat related to precedence is the brace. A pair of braces ({ }) bundles a number of statements into a single statement. For example, you need to use braces when you write a function that contains more than one statement. The same is true in `if` statements and loops.

1.7 Coercion Happens

The order of the atomic modes is: logical, numeric, complex, character. This is ordered by the amount of information possible. There are only three possible logical values, any logical value can be represented as a numeric value, any numeric value can be represented as a complex value, and any of the above plus the contents of the Library of Congress can be represented by a character value.

Whenever an atomic vector is created with elements of more than one mode, then the result will have the mode of the most general element. For example, the mode of `c(T, "cat")` will be character and the mode of `c(3, 4i)` will be complex.

The expression `c(T, NA)` implies that the mode of a solitary `NA` must have mode `logical`. This, by the way, implies that if a single `NA` is used as a subscripting vector, then that is a logical subscript, not a numeric one. Compare:

```
> c(T, NA)
[1]  T NA
> c(T, as.numeric(NA))
[1]  1 NA
```

Although most coercions are obvious, the ones involving logicals are not. When coercing logical to numeric or complex, a `TRUE` becomes 1, and a `FALSE` becomes 0. A number coerced to logical is `TRUE` unless it is zero (or `NA`).

In many operations coercion occurs automatically. This is the grease that allows S to run smoothly (and sometimes a slimy trick if the result is not what you expected). A common use of coercion is in an expression like

```
if(length(x)) ...
```

where `if` is expecting a logical value but it gets a numeric one. The expression is equivalent to

```
if(length(x) != 0) ...
```

but more compact and almost as intuitive.

A very useful coercion occurs in `sum(x > 0)`; the result of `x > 0` is a logical vector the same length as `x`, but `sum` expects numeric or complex arguments so it coerces the logical vector to numeric (the least general mode that it needs). The result is the number of positive values in `x`.

DANGER. Notice:

```
> 50 < "7"
[1] T
```

This shows that the coercion must be to character, and that you want to think carefully about comparisons on characters.

1.8 Looping

The only places where actual work gets done in S functions are inside calls to `.C`, `.Fortran` or `.Internal`. That is, in calls to C routines, Fortran routines, or the special routines (written in C) that understand S objects. The more directly you get to these endpoints, the more efficient your code will be.

It is well-known to users of S that using `for` loops and its cousins can be quite slow. Fundamentally, this is not the fault of the looping mechanism itself, but that a lot of calls to S functions are being made. There is a fair amount of overhead for each call to an S function; if a loop does hundreds or thousands of iterations, the overhead adds up to a noticeable amount. So the general rule is that the less that is in loops, the faster your code.

Here are three situations: no loop is used; `apply`, `lapply` or a relative is used; a loop is used.

No loop is required if the same operation is done on each element of an object. If you want to replace each negative element of a matrix with zero, then you could loop over the rows, loop over the columns and change each element individually. A faster, more direct way would be:

```
xmat[xmat < 0] <- 0
```

If the same operation is done on sections of an object, then a call to `apply` or `lapply` or a similar function is probably a good choice. The `apply` function operates on arrays, and `lapply` operates on lists. In older versions of S, the apply family of functions was implemented as `for` loops, but version 3.2 of S-PLUS changed them so that they are more efficient than a `for` loop. When these internal apply functions are available to you, they can speed execution substantially compared to the equivalent `for` loop. A section on the apply family begins on page 75, in particular, see the sorting example on page 76.

When one iteration requires results from previous iterations, there is usually no alternative to using a loop. Loops can be eliminated in a few special cases of this through the use of functions like `cumsum` and the S-PLUS functions `cumprod`, `cummax`, `filter`. When you do use a loop, try to put as little in the loop as

possible. Also try to loop as few times as possible—sometimes there is a choice of how to loop.

1.9 Function Arguments

Arguments to functions can be divided into required arguments and optional arguments. The ability to have optional arguments is yet another powerful feature of S. You can look at it either as getting more flexibility without any complication, or as getting simplicity with no reduction of functionality.

Since not all arguments need to be given in a call to a function, there needs to be a mechanism to match the function arguments to the arguments in the call. Arguments may be specified by name (and since there is a limited set of arguments for a function, the usual convention of abbreviation may be used). Arguments may also be specified by their position in the call.

Here is the general rule for matching the call arguments to the function arguments: First, names that match exactly are paired up. Second, names that partially match are paired. Finally, remaining arguments in the call are matched to the remaining arguments in the function in order. *Come live with me and be my love* [3]

There is a complication. The `...` construct (called "ellipsis" but more commonly "three-dots") allows a function to have an arbitrary number of arguments. There is a distinction between arguments that appear in the function before the three-dots and those that appear after it. Arguments before the three-dots are matched as usual—by partial name matching and/or by position. Arguments that appear after the three-dots are matched only by full name. All unmatched arguments go into the three-dots. Carefully written help files indicate arguments that may only be matched by full name by placing an equal sign after the argument name when it is described (the `paste` help is an example). It does not matter where these arguments appear in the call—there is no requirement that they be last.

1.10 A Model of Computation

It is advantageous to be able to visualize what is happening when an S function (or any program) runs. The description I'm about to give is not accurate in every detail, but should serve the purpose of helping you to picture what is happening between the time you hit the return key and you get back an S prompt. *That maps are of time, not place* [4]

As in business, the three most important things in computing are location, location, location. There are three types of space that are important for S objects: disk-space, RAM and swap-space. Disk-space is where permanent S objects live. RAM (random access memory) is where S objects that are being

used in a calculation must be. Swap-space is an extension of RAM for when things get rough.

If we multiply the permanent dataset `prim9` by `2`, S first finds `prim9` in disk-space and makes a copy of it in RAM. Then the multiplication of each element is performed. The multiplications can take place because the computer knows the *address* in RAM of each element of the copy of `prim9`. More particularly, the situation is probably that it knows that all of the elements are lined up side by side, it knows how many there are, it knows how much space each one takes, and it knows the address of the first one.

We can envision RAM as a long line of boxes, where each has an address. At any one time some of these boxes are in use, and probably some are empty. When S needs to bring another object into RAM, the computer finds a place for the object and remembers the address where it was put. If there is not enough room in RAM for the new object, then the operating system guesses about what in RAM won't be needed for a while, and puts that in the swap-space (storage area). At a point when something that is in swap-space is needed, then something else needs to be put in swap-space and what is needed is "swapped" or "paged" into that place in RAM.

When computations are small, then everything fits in RAM and there are no problems. If RAM fills up, then paging starts; this dramatically slows the progress of computations, as the computer spends most of its time trying to solve the puzzle of how to get all of the necessary ingredients into RAM at the same time. If the computation needs more memory than RAM plus the swap-space can hold, then the computer has to give up—sometimes the exit from this situation is less than graceful. Here is an example of an error message from running out of memory:

```
Error in 1:ll: Unable to obtain requested dynamic memory
Dumped
```

DANGER. Do not confuse this with trying to create an object that is larger than the current limit:

```
Error in 1:ll: Trying to allocate a vector with too many
        elements (40000000)
Dumped
```

This latter error can be fixed by increasing the `object.size` option. The former error can only be solved by changing the computations or using a machine that has more memory.

Humans tend to make a distinction between data like `2` and `prim9`, versus

operations like multiplication. But at the RAM level, functions are also just a collection of addresses.

We are now in a position to examine why S is so slow, or more positively, why C is so fast. Each object in a C program is declared to be of a certain type, so the computer knows how much space each object (or at least each element of the object) takes. C is compiled so that it has a clear map of which addresses it needs to visit in which order before it starts. In comparison with this, S is on a treasure hunt—S goes to one spot, then gets a clue to where it has to go next, where it gets a new clue, and so on.

So why doesn't S do what C does? Flexibility. An input to an S function is not restricted to a particular type. S functions do not care when functions that they call are changed; the definition of functions at the time of use are used, not the definition at the time that the calling function was created. And so on, etcetera.

S and C should not be thought of as competitors, with C winning because it is faster or with S winning because it is more flexible. They should be thought of as associates, each with its own strengths, that should each be employed as appropriate.

1.11 Things To Do

Write a function that uses each form of subscripting.

What happens when you subscript with complex numbers? Is this the most useful behavior?

Write a command that uses at least one operator from each line of the precedence table.

Read each S function available to you, and try to understand how it works.

1.12 Further Reading

Becker, Chambers and Wilks (1988) *The New S Language* is the fundamental document for the S language. Chambers and Hastie (1992) *Statistical Models in S* discusses object-orientation as it first appeared in S—Appendix A may be of particular interest.

Spector (1994) *An Introduction to S and S-Plus* provides a gentler introduction to S than this book as well as covering graphics more thoroughly. Venables and Ripley (1994, 1997) *Modern Applied Statistics with S-Plus* also discuss the S language in a few chapters and then go on to provide examples of statistical analyses in S.

If you are a historian or you want to see how far S has come, then you can look at Becker and Chambers (1984) *S: An Interactive Environment for Data Analysis and Graphics*; and Becker and Chambers (1985) *Extending the S System*. These two books are obsolete for the purposes of computing.

1.13 Quotations

[1]Theodore Roethke "I Knew a Woman"

[2]Henry David Thoreau "I Am a Parcel of Vain Strivings Tied"

[3]Christopher Marlowe "The Passionate Shepherd to His Love"

[4]Henry Reed "Lessons of the War: Judging Distances"

Chapter 2

Poetics

In which right thinking is advocated.

2.1 Abstraction

Abstraction is the heart of programming—all else is mere detail.

A major symptom of the need to abstract is the occurrence of redundant code. You may immediately realize that you would rather not type the same statements over and over again, so abstracting those lines into a function saves your fingers. But there is a more important reason. Suppose that you decide something needs to change in those lines—you found a bug or the scope of the functionality changed—then the single function holds a tremendous advantage over the repetitious code. In the repetitious case, you need to change every occurrence (yet more typing), and there will be a bug when you miss one—and that bug will be hard to find.

Capricious Rule 1 *Whenever a group of at least three similar lines of code occurs more than once, a function shall be written to perform the task.*

The simplest form of abstraction in S is to take a task that you use one or more command lines to do, and make that into a function. In the example below we assume the user wants to look at the histogram and get the summary and variance for a number of datasets.

```
> fjjexplore
function(x)
{
        hist(x)
        c(summary(x), var = var(x))
```

```
}
> fjjexplore(freeny.y) # a plot appears
  Min. 1st Qu. Median  Mean 3rd Qu.  Max.        var
 8.791   9.045  9.314 9.306   9.591 9.794 0.09961393
```

Every function embodies an abstraction, some abstractions are more useful than others. Subscripting in S is an example of powerful abstraction. Several ways of approaching subscripting are all handled similarly—there is both generality and simplicity.

One of the benefits of object-oriented programming is that it forces the programmer to abstract the problem. The abstraction chosen may be good or poor of course, but at least there is hope of avoiding an amorphous mass.

DANGER. If your code reads like a phone book, something is wrong. Change it. This generally can be fixed by bundling like objects together and possibly vectorizing.

A particular type of abstraction is indirection. An example is the `lib.loc` object that states the directories where S libraries may be found. The `library` function is not limited to looking for libraries in one set place, instead there is a two-step process which is flexible and convenient.

The `poet.dyn.load` function is another example of indirection.

```
"poet.dyn.load"<-
function(fname)
{
        if(!exists("Poet.Location"))
                stop("set Poet.Location to poetry directory")
        real.loc <- paste(Poet.Location, fname, sep = "/")
        if(exists("dyn.load"))
                ans <- dyn.load(real.loc)
        else ans <- dyn.load2(real.loc)
        invisible(ans)
}
```

There are object files that need to be loaded to use several of the functions described in this book. In order to be loaded, the directory where the files live needs to be known, which is going to be different for each installation. The `poet.dyn.load` function provides indirection for the location of the object files. It also allows an easy way of changing between `dyn.load` and `dyn.load2` depending on the availability of the functions in specific versions—another instance of indirection.

2.2 Simplicity

Programming Perl lists three virtues of programmers: laziness, impatience and hubris.

Being a lazy person, I think of laziness as the most important of the three. *Unix Power Tools* credits *Programming Perl* as describing laziness as the "quality that makes you go to great effort to reduce overall energy expenditure." Jon Bentley in *Programming Pearls* takes a slightly different tack, "Good programmers are a little bit lazy: they sit back and wait for an insight rather than rushing forward with their first idea."

Another good way to be lazy is to prove the feasibility of a large project before you spend a lot of time on programming details. Look for critical points that might not be doable within the given constraints—show-stoppers—and solve those first. A likely example is that speed may not be adequate—moving strategic pieces of the code from S to C can save the day. When looking for show-stoppers, you are concentrating on the weakest portions of your plan. Even if the original plan appears feasible, you may find a more streamlined attack on the problem.

Impatience is useful to make computers do more work than we do (and to make them do all the boring stuff). For instance, it is a virtue to write a small function to do something that you are repeatedly doing on the command line.

Hubris forces the programmer to produce quality solutions, and to attempt the impossible.

I would like to add another virtue to the stack: simple-mindedness. Simpletons avoid whatever is complex. Functions that are simple are easily used and easily adapted. Most importantly, simple functions are less often broken.

Capricious Rule 1 *No function shall contain more than 37 lines of code.*

I must confess that I have written a couple 500-line functions of which I am reasonably proud, but these are exceptional cases in which a lot must be done.

The absolutely, positively most splendiferous mistake of beginning programmers is to try to do too much in a single function. Think in terms of building blocks. Consider all of the things that need to be done to achieve your goal, and look for ways to break it up into subtasks. *And it grew both day and night, Till it bore an apple bright.* [5]

But why? Because small functions are easier to understand. Because the subfunctions that you create are going to be useful again. Because a long, involved function is hard to adapt to a slightly different purpose.

Each function should have a contract—it accepts certain inputs, and agrees to produce a specific output. This is the core idea of "modular programming"—that the code can be broken into pieces that can each be verified to do what they are supposed to. If all the pieces work, then the whole system should work.

When changes are made, then testing can be focused near the changes.

A very important thing not to do is write functions with side effects. Specifically, do not write functions that create or modify objects. Instead collect all of the newly created information (probably in a list), and return it from the function. There has been a proposal to ban the `<<-` operator to make it a little less convenient to change objects behind the scenes; I second the motion. Side effects destroy the transparency of a set of functions, making it more likely that the user will make assumptions that are not true.

On the other end of the same stick, avoid using global variables as much as possible. It is much more transparent when all necessary variables are passed in as arguments. It is also likely that the function will be much more versatile.

Data hiding is another technique to achieve simplicity. The most obvious example of data hiding is some of the `print` methods in S. For example, an `lm` object represents a linear regression. The object contains a number of components that are used in prediction and so on, yet its printed form is merely a terse summary of what the object represents. Similarly details can be hidden from functions.

Although the `missing` function is a useful tool, it should be used only when necessary because it can complicate programming. Let's look at two functions that do the same thing, but in different ways.

```
"fjjmiss1" <-
function(x, y)
{
     if(missing(y)) {
          ... }
     ...
}


"fjjmiss2" <-
function(x, y=NULL)
{
     if(is.null(y)) {
          ... }
     ...
}
```

Assuming that the ellipses in the two functions are the same, the results will be the same. Actually this is not quite true—if `NULL` is a valid input for `y`, then the two functions are different. Looking at these functions in isolation, `fjjmiss1` has the advantage since it is more general.

When we consider these functions being called by other functions, the picture changes. A function that calls `fjjmiss1` needs an `if` statement that depends on whether or not `y` is to be missing. A call to `fjjmiss2` has no such complication. You not only want each function to be as simple as possible, you want the interaction between functions to be smooth as well. An appropriate use of `missing` is in the `[.stack` function on page 228.

Unlike poetry in a natural language where there is rhyme, meter and so on, there are few constraints when writing in S. An exception is the naming of function arguments and list components. We desire simplicity of use as well as simplicity of construction, so it is nice to have argument and component names that can be abbreviated effectively. All of the names should be distinguishable by the first few letters—a sort of anti-alliteration—while maintaining clear identification of the meaning.

Capricious Rule 1 *The first two letters of an argument name shall distinguish it from all of the other argument names.*

While we are on the subject of virtues, there is yet a fifth to be had—distrust. Distrust everyone's programming, especially your own. As people gain experience with computing this tends to be a natural occurrence. However, it is better to cultivate distrust actively than to suffer bouts of depression due to disappointment.

2.3 Consistency

If you believe that consistency is the hobgoblin of small minds, then be small-minded when programming. Consistency makes it fast for users to learn the system, keeps some bugs from getting into the code, and makes it easier to fix bugs that do appear. As Robert Frost said in "The Figure a Poem Makes", "No surprise for the writer, no surprise for the reader."

Think of consistency as abstraction on a more diffuse scale. If we always use the same argument name for the same thing, we really have abstracted the concept of that argument.

Follow naming conventions to maintain consistency. Argument names in S are traditionally in lower case. However, an exception is when the function takes a function as an argument along with an arbitrary number of arguments for that function. The arguments for the function accepting a function should have its arguments all in capitals. For example, the arguments to `lapply` are:

```
X, FUN, ...
```

The reason these arguments are capitalized is so their names will not conflict with argument names for the function passed in as `FUN`. You can make a call like:

```
lapply(as.list(seq(0,.1, by=.01)), mean, x=prim9)
```

The `X` of `lapply` and the `x` of `mean` (in this case set to be `prim9`) can peacefully coexist. Unfortunately there are several functions that take functions as arguments that do not adhere to this convention. An example of using this convention is the `line.integral` function (page 320). An alternative method of handling the problem is given in the `genopt` function (page 331).

Version 3 of S forces a naming convention on methods for generic functions. Version 4 does not enforce the convention, but it is probably wise not to stray from it without reason. Object names tend to be in lower case, you don't need to follow this but you should decide what you are saying when you capitalize.

Objects that live in database 0 usually start with a dot and have the next letter capitalized.

Give similar arguments in different functions the same name. For example `iter.max` is used in several functions to specify the maximum number of iterations allowed. Some other functions use `niter` for the same purpose. I think `iter.max` is a better choice because it more clearly states its meaning, and variables that start with "n" are more common than those starting with "i". Note that S is not confused when the same name is used in nested functions.

Variables within functions should have clearly meaningful names. It is bad practice to use the same name for more than one meaning within the function (an exception can be made to save space when dealing with large objects).

Avoid names that can be easily confused. For example, `nnn` and `nnnn` would be better named `n3` and `n4`.

Capricious Rule 1 *No variable shall be named* `tmp`.

All variables are temporary so `tmp` is not descriptive. Thus it is tempting to have it stand for more than one thing in a single function.

On a Horse Who Bit a Clergyman

The steed bit his master;
How came this to pass?
He heard the good pastor
Say, "All flesh is grass."

anonymous
(Eighteenth Century)

Object-oriented programming is not enforced by S, and if object-orientation is used, there are few constraints on its form. We get more flexibility at the possible expense of consistency. It is the prgrammer's responsibility to maintain consistency.

2.4 Beauty

Once you have a simple, effectively abstracted function that conforms to all known conventions, you are probably approaching the beautiful.

when the world is mud-
luscious [6]

One of the most beautiful functions in S is `rep`. This function does a lot with three straightforward arguments. The `times` argument is what makes it transcend mere utility. In its most common use, `times` says how many replications of the object we want. But turning that on its side in the light of vectorization, we can also give a vector for `times` that tells how many times to replicate each element in turn. Thus we can give commands like:

```
> rep(1:5, rep(2, 5))
 [1] 1 1 2 2 3 3 4 4 5 5
> rep(1:5, 4:0)
 [1] 1 1 1 1 2 2 2 3 3 4
```

Since computers are less amused by ambiguity than we, it should be the case that these two meanings for `times` meet up nicely for length one objects, and they do.

The `length` argument states the length that the answer is to be.

```
> rep(1:5, len=4)
[1] 1 2 3 4
> rep(1:5, len=7)
[1] 1 2 3 4 5 1 2
```

DANGER. The most common bug when using `rep` is to forget to name the `length` argument in the call, so that it is interpreted to be the `times` argument.

```
> rep(1:5, 2)
 [1] 1 2 3 4 5 1 2 3 4 5
> rep(1:5, leng=2)
[1] 1 2
```

`rep` could have been written so that it did less, or it could have had a bunch more arguments that made the whole thing complicated. Instead it holds to a middle ground of both power and simplicity. This brings to mind the following rule.

Capricious Rule 1 *All functions shall have three to seven arguments.*

Too few arguments means that the abstraction embodied by the function is not general enough. Too many arguments means that too much is done and hence it is too complicated.

Arbitrary limits are ugly and should be avoided. S does a good job of not having them. There are only three places that I can think of where they exist. The number of arguments that can be given to a C or Fortran routine through `.C` or `.Fortran` is limited to some arbitrary number (about 50 in most versions of S-PLUS, I think). The number of explanatory variables that can be given to the `leaps` function is limited to something like 31. The maximum order of the polynomial solved by `polyroot` is 48. If the solution you have to a problem contains an arbitrary limit, your pride (remember hubris?) should get in the way of taking it seriously. *Disturbed some rythm, old and of vast importance* [7]

The final flourish of creating a beautiful function is to give it a good name. The name should clearly indicate the functionality. The number of characters in the name should be inversely proportional to the number of times it will be typed—a function that is typed only a few times a month can afford to have a longer name than one typed several times a day. A descriptive name for an infrequently used function also makes it easier to find. Avoid using good, general names (like maybe `fly` or `run`) for inferior functions.

2.5 Doing It

Much has been made of "top-down" programming. It is sometimes professed that the way to approach a programming task is to specify the main structure and how everything relates to its neighbors in advance, only details are left to be worked out. I suspect that there are few worlds born whole as out of Zeus' head, but if you run across one, by all means take advantage of it.

In my experience it is more likely that the problem is only partially known (or knowable) in advance. Experimentation provides insight into what the real question is, and how wide of a net can be cast. As more is learned, the proper structure starts to show itself. If "top-down" is taken to mean think before you type, then I like it; but if taken to the extreme, you may paint yourself into some corners. An apt slogan is one seen on bumperstickers: "Think Globally, Act Locally."

A good image for code generation is one of punctuated evolution. There are short intervals of massive changes interspersed with long periods of almost no

change at all. The periods of change can be spurred either by new needs or by a better structure becoming evident.

In many of the examples in this book two or more versions of a function are given, with a later version being (hopefully) an improvement of the preceeding version. Obviously the aim is not to produce examples of poor coding for you to emulate. The main reason is to drive home that revising code should be standard practice. Write an initial version, play with it a little, then fix the limitations that you've found—always trying to move toward simpler and more general.

Unlike living creatures, computer programs unfortunately often need to exhibit backward-compatibility. This is a dark cloud over beautiful programming—either consistency, non-redundancy or backward-compatibility has to give way. If the program is not wide-spread, then breaking the compatibility by forcing the use of the new structure is best. If fame has spread though, the choices tend to be reduced to retaining the same interaction (though internals might change), or having two systems that do basically the same thing.

This is a caution to get it right the first time. If you have a significant piece of code, give it to a few people to test and comment on. Do not sell it. Do not propagate it over the internet (do as I say, not as I do). Think about it, hone it until it is right, then prosyletize.

There is an adage (never followed) that software should be written once, then thrown completely away and rewritten. Nobody ever does it because it seems extraordinarily wasteful, but I'm sure it is closer to optimal than how software actually is written.

In the midst of all of this pondering on programming etiquette, truth may break in. It is seldom the case that programming is performed with no constraints. As a deadline worth millions looms, there may not be time to create perfect functions. Take the gold and tidy your code later. *So was I once myself a swinger of birches.* [8]

Think safety

Many building construction sites have the prominent sign "THINK SAFETY". There should be such a sign whereever software is under construction.

Keep in mind that code evolves and adapts. It is not enough to get it right for this one use—think also of how minor changes to the code might introduce bugs. *With trembling care, knowing that most things break;* [9]

For example, the length of a vector may be 5 in a function, so you can use 5 as the length of the vector—hard-code the value—but someone who changes the code and uses a different length vector may not find all of the locations where 5 means the length of this vector. It is safer to assign a name to the length of the vector.

If you are adding arguments to a function that is already in use, then put the new arguments at the end. If you do this, then calls based on the previous definition of the function will continue to work properly.

If the use of a function critically depends on the value of a default, then explicitly give the default for that argument. A common fragment of code is

```
if(any(is.na(match(x, y))))
```

which depends on the default value of the `nomatch` argument to `match`. If someone decides to create their own version of `match` with the default value of `nomatch` changed to `0`, say, then the above construction will not work. Safer would be:

```
if(any(is.na(match(x, y, nomatch=NA))))
```

DANGER. Do not change the `match` function's default value of `nomatch` because there are numerous places that do not use the safe approach.

Here is an example of where thinking of safety would have helped.

```
> jjpbx
  a          b
1 1 -0.7710749
2 2  1.3230554
3 3 -1.0032299
> class(jjpbx)
[1] "matrix"
> jjpbx[1,]
 a          b
 1 -0.7710749
> dim(.Last.value)
NULL
```

It seems perfectly natural to me that we may want an object of class `matrix` to remain a matrix when subscripted—that is, for the default value of the `drop` argument in subscripting to be `FALSE`. Here is a method to do that.

```
"[.matrix"<-
function(x, i, j, drop = F)
{
        cl <- class(x)
        x <- unclass(x)
        ans <- x[i, j, drop = drop]
```

```
        # avoid NULL matrices
        if(length(dim(ans)))
                class(ans) <- cl
        ans
}
```

Subscripting indeed works, but now the `drop` function doesn't work on these objects.

```
> jjone.d <- jjpbx[1,]
> dim(jjone.d)
[1] 1 2
> dim(drop(jjone.d))
[1] 1 2
```

Adding the seven characters `,drop=T` to the definition of `drop` fixes the problem:

```
"drop"<-
function(x)
{
        n <- length(dim(x))
        if(n == 0)
                return(x)
        eval(parse(text = paste("x[", paste(rep(",", n -
                1), collapse = ""), ",drop=T]")))
}
> dim(drop(jjone.d))
NULL
```

This will cause `drop` to fail on objects that do not have a `drop` argument in subscripting but do have a dim—that seems quite acceptable.

Put error messages where appropriate (using the `stop` function). This is harder than may be naively thought. You do not want to unduly clutter or weigh down functions with excessive error checking. On the other hand, you want errors when they do occur to produce a message that will make the cause of the problem clear. In general, low-level functions that aren't intended to be called directly by the user should have few error checks, functions called a lot by inexperienced users should be heavily checked.

Most important is that a function should never be allowed to give a wrong answer. Better a totally undecipherable error or the computer crash than two times two equal five.

Here is how I decide where to put error checks:

• If the problem will result in a wrong answer, put in a check (and look around for more of the same).

• If the problem will result in an error farther on and the error condition will be vague or misleading, then put in an error check.

• If the problem will create a reasonable error elsewhere but this is likely to be used by novices, then put in an error check.

Implicit in the list above is that thought has been put into where errors come from. Such thought is useful not only for placing error messages, but also to eliminate bugs.

Once you know where you want an error message, you need to decide what it is to say. I believe in short messages (so the user will actually read it). However, clarity and accuracy should take precedence over brevity.

Documentation

There are three forms of documentation that you can write for your S code. Help files, and comments in the code are two types. The other (which I term prose documentation) is an explanation that might appear as a user's manual. Prose documentation would tend to show how to use a group of functions in a coordinated manner as well as motivating why one would want to use the functionality.

I think writing a help file for a function is an integral part of writing the function. (In version 4 the documentation is a part of the S object rather than a separate file, the principles remain the same.) Writing the documentation clarifies how the function fits with others, and suggests tests for bugs as well as being a useful product in its own right. You can not possibly know that your function is written as well as possible until you have written a thorough help file for it.

When writing a help file, it is a good goal to write it so that a person without a computer and with just the help files can confidently write correct programs. (This is a dreadful way to program, but it is a good image to have when writing help files.)

The `prompt` function produces a file that is a template of the help file you want. It fills in many of the items of interest and marks the spots where you need to put in information.

Here are some things to consider when writing a help file:

• Describe each argument fully. There are a few possibilities:

  • The description is clear. Fine. You need a measure for what "clear" means—think of a reasonably intelligent person who is completely uninformed about the subject. The description should include the type of object expected: "integer giving the number of ..." is preferable to "number of ..."; "matrix of explanatory data", not "explanatory data". This

gives more opportunity to ponder what happens if the input is different from what is expected, as well as being more explicit for the user.

• The description is clear, but does not match what the function actually does. Change the function or the description, as appropriate. For example, you might have a statement like "numerical array. When three-dimensional, then ..." I often find that I don't know what the "then" clause is, that I know what it should be but I'm not sure the function does that, or that the function does something more sensible than my description. This is a very expedient way to force me to think through what the function should be doing, to debug the function, and to document it all in one step.

• The argument is hard to explain. Rethink. If the argument can't be explained, then surely users will get it wrong, and more likely the whole thing is ill-defined. An ill-conceived argument is a potent source of bugs.

• Are the arguments in an appropriate order? Required arguments should be first. The order should match that of functions with similar arguments. In general, the most used arguments should be near the front. Do all of them have good names that are consistent with other function arguments?

• Explain the output fully. While you do this, wonder whether everything relevant is returned, and nothing but.

• If there are side effects, spell them out.

• Provide examples that demonstrate the use and usefulness of the function. If possible, give examples that the user can copy directly to get output—the in-built datasets are expedient for this. This is the section that is likely to be the most informative if done properly. No matter how clear and concise you think your explanations are, a good example will better answer the user's questions.

• State all else that would be useful to the user. This might include references to literature, details of the computing algorithm, and explanation of how to use the function.

• A most important part of a help file is the BUGS section. I realize that it's fashionable to pretend that bugs do not exist, but those who claim to have no bugs in their code are either fools or liars. Many functions as they are written contain known bugs, even if these are just limitations on what they might be expected to do. Write these down while they are fresh in your mind so that users are warned, and you have a clear map of what remains to be done to the function.

My feeling on comments in code is that there should be some, but they should tend to be sparse. Put in comments when you are doing something clever—it is very easy to forget how clever you are. Also add comments if you

are working around a bug. Explain the bug fully so that it will be clear when the workaround can be removed. This is important—I have several comments scattered through my code that inform of a workaround, but do not contain enough information to know when the workaround can be abandoned.

If your code needs a lot of commenting to make it understandable, then you should consider rewriting the code so it is simpler rather than adding comments.

Minimize expense

A programmer's job is to minimize expense. Often this optimization is too narrowly defined as minimizing the memory and time usage of the code as it is run. Part of the reason for focusing on these is that they are the easiest to measure.

The actual problem also includes the time to program the solution, the time to train users, the time to maintain the code, the time to adapt the code to new uses, and so on. It is hard to overestimate the time required to maintain code, it is an important component of the expense. Remember that humans are expensive (and easily frustrated or bored), while computers get cheaper all the time.

2.6 Critique of S

I give S high marks for having effective abstractions and providing the ability to create further abstractions. It allows and encourages a great deal of simplicity. On the whole it is a very poetic language.

S does have its faults, though. One is that the name-space is easily cluttered. Although you can specify that a particular name should come from a specific place, it is not a very convenient operation. I don't see any solution to this, you merely need to be careful about naming objects.

Another weakness of S is that it has a lot of inconsistencies and redundancies. Almost all of this is because S has changed over the years, and there has been the desire to have as much backward-compatibility as possible. So some things that should logically be object-oriented are not, some functionality is duplicated, some functions use formulas while similar functions do not, and so on. There are also some function arguments that have different names in different places. All of this can frustrate a new user. However, all of the inconsistency in S is on the surface—the underlying language is incredibly consistent and simple.

2.7 Things to Do

Take a complicated function (preferably one of your own) and abstract the portions of what it does into several subfunctions. Repeat this on the same function as many times as you can, performing the abstraction differently each time.

Put comments into your functions so that they are completely explained. Then revise the functions so that as much commenting as possible can be eliminated.

Write help for each of your important objects. It can be very useful to document non-functions as well as functions.

Find a beautiful S function (`rep` is not an allowable answer). What is good about it?

What would be good functionality for a function named `fly`?

For each S function available to you, decide how to make it better.

2.8 Further Reading

The books *Programming Pearls* and *More Programming Pearls* by Jon Bentley are very good at encouraging good programming.

The Elements of Programming Style by Kernighan and Plauger has much wisdom that is still applicable, even though it was written in the late Paleolithic. Here is a sample: "The fact that a four line comment is needed to explain what is going on should be enough to rewrite the code. An even better reason is that the comment is wrong." They often present some published code and then show how it can be improved. A wonderful illustration of their spirit is that some of the examples of poor programming in the second edition are their code from the first edition.

Structure and Interpretation of Computer Programs by Abelson, Sussman and Sussman has abstraction as its main theme. It uses a form of Lisp as its language of choice—most of the examples could be translated into S without undue contortion.

When real conceptual trouble strikes, I occasionally pull out *How to Solve It* by George Polya. This book talks about how to solve mathematical problems, but the techniques are suitable for any problem solving. An example of his wisdom is that it is often useful to picture what a solution would look like, and then work backwards from there. The most memorable part of the book for me is his rules of style, which I paraphrase:

Rule of Style 1 *If you have nothing to say, then don't say it.*

Rule of Style 2 *If by some miracle you have two things to say, say one thing and then the other.*

2.9 Quotations

[5]William Blake "A Poison Tree"

[6]e. e. cummings "In Just-"

[7]Theodore Roethke "Moss-Gathering"

[8]Robert Frost "Birches"

[9]Edwin Arlington Robinson "Mr. Flood's Party"

Chapter 3

Ecology

In which peaceful relations are encouraged.

This chapter starts with topics concerned with the use of S in general—databases and their organization, naming objects. Then it turns to subjects more directly related to programming.

3.1 Databases

S has a search list where it looks for all of the objects that it needs. (A better name would be "search path" since it is not an S list, but the usage is established.) You can see what databases are currently on the search list with the command:

```
search()
```

The first database on the list is called the *working database*, and this is where assignments are written (it is possible to make assignments elsewhere). You need to have write permission for the working database in order to have S happy with you.

When S needs to find an object, then it looks in each database in the search list in turn. Once it finds an object with the correct name (and possibly the correct mode), it uses that object. So an object named `x` that lives in database 2 will *mask* an object named `x` that lives in database 3. This is a simplified version of how S searches for objects, for a more truthful explanation, see page 210.

You can add to the search list with the `attach` function. Typically you give it a character string which names a directory containing S objects:

```
attach("../other_directory/.Data")
```

This will place the new database in position 2, but an optional argument allows you to select where in the search list the new one is to fall.

You can attach lists and data frames to serve as databases. The columns of data frames and the named components of lists are the objects within the database.

```
> x1
Error: Object "x1" not found
> jjx <- list(x1=1:6, x2=letters[1:9])
> attach(jjx)
> x1
[1] 1 2 3 4 5 6
```

This can be an excellent way to organize computation in some situations. I don't recommend using a data frame or a list as the working database, however.

To remove databases from the search list, use `detach`. Usually you give `detach` the number of the database that you want to disappear.

Libraries are mildly special directories of S objects. To see what libraries are available to you, do:

```
library()
```

To get some more information about a library, you can do a command like:

```
library(help="examples")
```

and do

```
library(examples)
```

to attach the library. In version 3 of S the only real difference between a library and an ordinary directory of S objects is that the `.First.lib` function will perform some startup actions when the library is attached. Also the user does not need to know the path of the directory of functions. If you want to add libraries of your own, then S-PLUS allows you to create a `lib.loc` object that is a character vector of the pathnames of the directories where libraries may be found.

DANGER. In (most) versions of S, there are some superficial bugs when `lib.loc` is set. I don't know of any that are harmful.

3.2 Object Names

When I first came to S, I was thrilled that the objects I created remained permanently—even after I quit S. This was in contrast to systems where I needed

to explicitly state what I wanted to save before quitting. By the time my directory contained a few hundred objects, I began to learn the price to be paid for this convenience. Wouldn't it be nice to be able to discover the name of any particular object you want, or to be able to quickly decide what to throw away when the system administrator yells at you for hogging all of the disk-space? *Look on my works, ye Mighty, and despair!* [10]

The types of objects that you create can be divided into five categories: permanent functions, permanent data, modified system functions, temporary functions and temporary data. Some people might want to add a category of derived data, to represent data that can be reproduced, but at some cost. Permanent functions are functions that you have written that you will continue to want to use. Likewise, permanent data are objects other than functions that you want to keep long-term. What I call modified system functions are those that are similar to an in-built function, but does something slightly different. For example, I have a function called `p.unix.time` which is basically the same as `unix.time`, except that it returns 3 numbers: the total computer time, the clock time and the memory size. Temporary objects are those that will not affect you adversely should they disappear.

With proper naming conventions for your objects, you can ease the burden of object proliferation. A convention for temporary objects is most important. I start the name of a temporary function with `fjj` and start the name of other temporary objects with `jj`. Here's my reasoning: `jj` is easy to type, it doesn't appear in the words of any natural language I'm likely to learn soon, it seldom appears in abbreviations, and it can be thought of as deriving from "junk".

Other naming conventions that I've seen include starting the object name with `foo`, `tmp` or `junk`, or using triples of letters. The problem with the first three is that it is conceivable that the name of an object you want to keep may sneak into the convention. The problem with the last is that it is not convenient to list them.

I find it very useful to differentiate between temporary functions and other temporary objects. Functions are small so throwing them away buys you very little disk-space. Functions also tend to have a longer useful life than data—I can read a function and know what it does, but a matrix of numbers even with distinguishing dimnames soon becomes a mystery.

If I want to list all of my temporary objects, I can say:

```
objects(pat="jj")
```

To exclude the temporary functions, I say:

```
objects(pat="^jj")
```

(The "hat" symbol indicates the beginning of a word.) I can remove all of these with

```
remove(objects(pat="^jj"))
```

This brings us to an advertisement for some form of command line editing. If you typed the last statement but missed typing the second "j", then you might well be sorry. A safer approach is to look at the result of the `objects` command, and then wrap the call to `remove` around the same call to `objects`. Command line editing is the ability to recall past commands, and to re-execute them, possibly after moodifying the command. This means that text editing is involved. On Unix there are essentially two choices for an editor—`emacs` or `vi`. If you know neither, then learn `emacs`. For command line editing in S, the best method is to use `S-mode` in `emacs`. Admirers of `S-mode` are many and fervent. If you already know `vi` well and you have S-PLUS, then an alternative is to use the `e` flag when you start S-PLUS:

```
% Splus -e
```

You can then hit the "escape" key, go back into past commands and edit them.

A more complicated removal might be handled like:

```
jj <- objects(pat="^jj")
jj <- jj[-c(4, 9, 138)]
remove(jj)
```

Note that `jj` may be one of the objects removed. That's okay—suicide is allowed.

In S-PLUS, `objects.summary` reports various information on objects:

```
> objects.summary(pat="jjm", what=c("data.class",
+     "storage.mode", "extent") )
        data.class storage.mode extent
fjjmem1   function     function      3
fjjmem2   function     function      3
fjjmem3   function     function      3
fjjmem4   function     function      3
    jjm     matrix     integer   3 x 1
  jjmat     matrix     double    5 x 5
```

In the example above we ask for objects in the working database (the default) that have names containing "jjm", and we do not want the dataset date to be part of the output. Below we want to get all of the functions in the working database (and nothing else), and sort them by date:

```
objects.summary(mode="function", order="dataset.date")
```

It is good to have a naming convention for modified system functions. A method I've seen is to append `f.` to the system name, `f.matrix` for example.

This seems fine if you are an isolated user. However, if you are in a group of S users and functions are likely to be shared, elaboration is politic. Consider the situation of two or three `f.matrix` functions in a group. Person x uses function `fy` written by person y, function `fy` uses the `f.matrix` written by y, but `fy` sees the `f.matrix` written by x when x uses `fy` (got that?). The worst type of error might occur—everything looks fine, but the answer is wrong.

A solution is for each individual to have their own prefix to use. We would have `x.matrix` and `y.matrix` instead of two `f.matrix` functions. This also makes it easier to direct questions and comments to the right person.

The `diffmask` function will tell you if and how two objects of the same name differ:

```
"diffmask"<-
function(x, old = loc[2], new = loc[1])
{
        if(!is.character(x) || length(x) > 1)
                x <- deparse(substitute(x))
        loc <- find(x, num = T)
        if(any(is.na(c(old, new))))
                stop("need two locations for object")
        if(is.numeric(old))
                oldlab <- old
        else oldlab <- "old"
        if(is.numeric(new))
                newlab <- new
        else newlab <- "new"
        filenames <- tempfile(paste("database", c(
                oldlab, newlab), "", sep = "."))
        on.exit(unlink(filenames))
        dput(get(x, where = old), filenames[1])
        dput(get(x, where = new), filenames[2])
        unix(paste("diff -c ", filenames[1], filenames[
                2]), out = F)
}
```

CODE NOTE. If both `old` and `new` are given, then there is no need to do the `find` command. Taking that into account, the line could be changed to:

```
        if(nargs() < 3)
                loc <- find(x, num = T)
```

This uses `find` unless all three arguments are passed into the call. In this case, the probability that all three arguments will be given is infinitesimally small and the `find` call is not expensive, so I opted not to do this. However, there are circumstances in which it is decidedly valuable to avoid computations.

I find `diffmask` quite useful—I wrote it for myself, not the book. Often I have a local version of a function that is also somewhere else on the search list because I am modifying or debugging the function. I use `diffmask` to see the current modifications.

3.3 Divide and Conquer

No matter how organized your naming convention, there comes a time of too much. The most common solution is to create separate S data directories for the various projects you work on. Here is a Unix script to set up an S directory in the current directory.

```
mkdir .Data
mkdir .Data/.Help
# you may need to change the next line
Splus < $HOME/pS/dotFirst.q
```

Assuming that the script is called `Shere` and is executable, then you merely need to go to the directory where you want to run S, and give the command `Shere` to Unix. After which you can use the new `.Data` by starting S from the directory.

The `dotFirst.q` file contains a solution to the main problem with dividing your S objects into separate directories. Almost everyone has a set of general functions that they always want to have available. You don't want to copy the same functions into each directory (bad, bad, bad), but you can have one directory where your general functions live, and always attach that directory as S starts.

You may be wondering why I dislike the idea of copying functions around. This is the redundancy problem once again, but on a different scale. Suppose that you do copy a function into several different directories. Now suppose that you discover a bug in the function as you are using it in one of the directories. You will fix the bug where you are working, but—barring extraordinary circumstances—you will not change it in all of the other directories. It is likely that you will need to rediscover the bug in one or two more directories before the bug is extinct. Even without bugs, extra features may be added to some versions of the function so that you end up with several similar functions that are non-trivial to get back into sync with each other.

A `.First` function is used to perform tasks as an S session starts. Just before S gives you the first prompt of the session, it checks to see if there is a `.First` function in the working database, and if so, evaluates a call to it. (There is also a `.Last`, but I haven't yet discovered a good use for it.) Here is what that `dotFirst.q` referred to in `Shere` might look like:

```
.First <-
```

```
function()
{
        options(error = dump.frames, object.size =
                20000000)
        attach("../pS/dotFunctions")
        cat("attaching dotFunctions\n")
        if(exists(".privateFirst", where = 1))
                .privateFirst()
        cat("the answer to 2 * 2 is", 2 * 2, "\n")
}
```

The statement involving `.privateFirst` allows each individual directory to have its own start up actions. Under this scheme you use `.privateFirst` like you would normally use `.First`.

Note that the `attach` statement presumes that all of the places where S is to be used are on the same level of the directory tree. If this is not to be the case, then that line needs to be changed—in S-PLUS a statement like `getenv("HOME")` could be part of the solution.

What is important is the effect of dividing your objects rationally while maintaining convenience. The exact mechanism is not very important. You may have noticed that abstraction is still the operable concept. It is just that now we are attempting proper abstraction of groups of S objects.

In S-PLUS there is also a `.First.local` that can be used at a site so that a set of startup actions is performed for all of the users. For example, objects may be assigned, and libraries attached.

3.4 Environment

Your S session is controlled by a number of options that can be inspected and changed by the `options` function. There are quite a number of options each of which tends to be recognized by one or more functions, and some of the options are directed at the S language itself.

You can add your own options—there is no distinction between the options that come with S and those that you create. However, with freedom comes responsibility. When you intend to change an existing option but misspell it, then you get a new option and the old option still retains its old value. The function `soptions` (as in "safe options"), listed below, gives a warning when you are adding an option so that you will realize when you've made a mistake. You can always use `soptions` in lieu of `options`.

```
"soptions"<-
function(...)
{
        in.names <- names(list(...))
```

```
        oldnames <- names(options())
        newnames <- in.names[!match(in.names, oldnames,
                nomatch = 0)]
        if(length(newnames)) {
                warning(paste("new options added: ",
                        paste(newnames, collapse =
                        ", ")))
        }
        ans <- options(...)
        if(.Auto.print)
                ans
        else invisible(ans)
}
```

An example of when this is useful is:

```
> soptions(warning=2)
Warning messages:
  new options added:  warning in: soptions(warning = 2)
```

The actual option is named `warn` and what should have been typed was:

```
> soptions(warn=2)
```

If the `options` function were used instead of `soptions`, then no warning would have appeared and the user would probably not realize for a while that the option was not changed. This example also points out a weakness of `soptions`—there will be no warning message if the `warn` option is not set properly.

Some of the more important options are now discussed. See the `options` help file for more information on these options and to learn about other options.

The `object.size` option limits the size in bytes of an object when it is created. This is a safety feature so that blunders along the lines of

```
1:Inf
```

are not taken seriously. In many versions of S the default value is rather small, so this is a good candidate to be changed in your `.First`. There is also an `object.size` function that returns the size in bytes of its argument.

The `digits` option controls the number of significant digits that are printed.

```
> cat(pi, "\n")
3.14159265358979
> print(pi)
[1] 3.141593
> cat(format(pi), "\n")
```

```
3.141593
> options(digits=14)
> print(pi)
[1] 3.1415926535898
> cat(pi, "\n")
3.14159265358979
```

From this example you can see that `print` obeys the `digits` option, as does `format`, but `cat` does not.

The `width` and `length` options also concern printing. `width` gives the maximum number of characters to put on a line. For example, it should typically be 80 when printing to a terminal, and it is set at 55 for listings in this book. `length` specifies the number of lines per page, and is used, for instance, to decide how often to print column headings when printing a matrix. If you only want the column labels printed once at the start, then set the `length` option to some large number.

The `error` option gives the action to be performed when an error during evaluation occurs. This can be either a function (with no arguments) or an expression of a call to a function. So you can say

```
options(error=dump.frames)
```

or

```
options(error=expression(dump.frames()))
```

but not

```
options(error=dump.frames()) # WRONG
```

Your two main choices for `error` are `dump.calls` and `dump.frames`. The former returns the stack of calls at the time of the error, while the latter returns the stack of calls plus the state of each variable in each call at the time of the error. See the debugging chapter to learn why you care. The `ignore.error` function on page 215 is an instance where it is natural to set the `error` option to `NULL`.

The `warn` option controls what happens when a warning is encountered. The two extremes are `-1` to suppress warnings, and `2` to convert warnings to errors.

You can change the prompts that S gives by using the `prompt` and `continue` options. You may, for example, want the prompt to include the name of the current directory.

The `keep` option states what should be kept in a certain hash table. After an S session starts up, each object that is used is tested to see if it should be put in the hash table. An object that is in the hash table is found quicker than objects that are not in it. Usually `keep` is set to `"function"` so that functions

and only functions are hashed. You may have noticed that a function runs slower the first time it is called in a session than subsequently. You can set `keep` to `"any"` which will put all objects in the hash table. This is generally a bad idea because you will soon use lots of RAM, but it can be a useful strategy to temporarily use this setting to hash some objects that will be accessed numerous times in the session. Going the other way, `keep` can be set to `NULL` to disable hashing entirely. This could be used if you are going to access a large number of functions only once. See the discussion of `synchronize` on page 103 for more on this hash table.

Some other more esoteric options that can be useful are `expressions`, `echo` and `interrupt`. See page 219 for a discussion of how options are implemented.

The `par` function is similar to `options` in many respects, but it controls the behavior of S graphics. The graphics parameters mostly have names consisting of three characters, so the meaning of a graphics parameter is not completely discernible from its name—take this as a lesson of how not to name objects.

DANGER. Although generally analogous, the `options` and `par` functions do have different conventions for the return value.

```
> par("mar")
[1] 5.1 4.1 4.1 2.1
> par(c("mar", "cex"))
$mar:
[1] 5.1 4.1 4.1 2.1

$cex:
[1] 1
```

When querying the value of a single graphics parameter, `par` returns the value of the parameter, and only returns a list of values when two or more parameters are queried. In contrast, `options` always returns a list. However, when a graphics parameter is being set, then `par` does return a list (invisibly) so that the following idiom will work:

```
on.exit(par(old.par))
old.par <- par(mar=c(8,4,1,1)+.1)
```

3.5 Source Control

It is wise (and more relaxing) to have your code under a control system. I will talk about SCCS, but there are others, such as RCS. SCCS is an acronym for "Source Code Control System". The idea is that you have a history of versions of the code (or whatever) so that you can recover any particular version. The main reasons to use such a system are: the new version that you just wrote may not work right so you want to go back to what did work; you need to reproduce the state that existed at some specific time in the past; you are protected from accidentally destroying what exists. Believe me, the last point is no small bonus.

The two S functions called `sccs` and `diffsccs` make it trivial to put S objects under SCCS control. The only preparation is to make sure that you have SCCS available on your machine and to create a subdirectory called `SCCS` in the directory where you want the control files to live. Start S in this same directory. An object named `my.fun` is put under control with the command:

```
sccs(my.fun)
```

This creates a file in the current directory called `my.fun.q` which is a dump of the function. Furthermore there is a related file inside the SCCS directory to keep track of the versions (done in a memory efficient manner).

My usual method when I am about to edit a function is to do:

```
diffsccs(my.fun)
```

to see if what is there is different than the last SCCS version. If it is, then I use `sccs` on it, and then edit the function. I have found it best to try out my changes to a function rather than immediately creating a new version, hence the reason that SCCS is often not up-to-date. If the S object has a name that is not nice for a filename, then using the `xname` argument to `sccs` and `diffsccs` is essential. Here is an example:

```
diffsccs("/.rationalnum", "Div.rationalnum")
```

If you want to replace the current definition of a function with the last version (or if you accidentally removed the function), then use `source` or `restore` on the `.q` file. To get a version farther back, you will need to use an SCCS command before using `source` or its relatives.

In addition to putting S objects under source control, it is good to control help file, C and Fortran sources. For these you need to learn more about SCCS. *Unix Power Tools* by Peek, O'Reilly and Loukides (1993) has a brief introduction to SCCS (despite what they say, nothing needs to be added to files put under SCCS). Details of the use of SCCS are available in the man page. You may find it useful to write some simple scripts or aliases to perform the tasks with names that make sense to you.

Here are the listings for the two functions:

```
"sccs"<-
function(x, xname = x, new = !length(unix(paste("ls",
        xfile), out = T)), already.out = F)
{
        ok.dump <- function(x, xfile, do.dd)
        {
                if(do.dd) {
                        data.dump(x, xfile)
                }
                else {
                        dump(x, xfile)
                }
        }
        # start of main function
        if(is.character(x)) {
                if(length(x) > 1)
                        stop("only one at a time")
        }
        else x <- deparse(substitute(x))
        do.dd <- !is.function(get(x))
        if(do.dd) {
                xfile <- paste(xname, "Q", sep = ".")
        }
        else {
                xfile <- paste(xname, "q", sep = ".")
        }
        if(new) {
                ok.dump(x, xfile, do.dd)
                unix(paste("sccs create", xfile), out
                         = F)
                unix(paste("rm ,", xfile, sep = ""),
                        out = F)
        }
        else {
                if(!already.out)
                        unix(paste("sccs edit", xfile),
                                out = F)
                ok.dump(x, xfile, do.dd)
                unix(paste("sccs delget", xfile), out
                         = F)
        }
        hcom <- paste("test -f ", xname, ".d", sep =
                "")
        if(unix(hcom, out = F))
                warning(paste("file ", xname,
                        ".d not found, do you have a help file?",
```

```
                        sep = ""))
        invisible(xfile)
}
```

CODE NOTE. This looks messy, but it is mainly a bunch of decisions. Most of the trickier parts will be covered in the next chapter. The main substance of the function is knowing the SCCS commands, but switching between `dump` and `data.dump` is an important feature.

CODE NOTE. The function has two different ways of testing for the existence of a file. This is done for the purpose of demonstrating options. I advise you to pick one way of doing a task, especially in a single function.

The ".q" in the file names comes from QPE (S's *nom de guerre*). The ending ".s" is sometimes used for assembly code.

```
"diffsccs"<-
function(x, xname = x)
{
        if(is.character(x)) {
                if(length(x) > 1)
                        stop("only one at a time")
        }
        else x <- deparse(substitute(x))
        do.dd <- !is.function(get(x))
        xfile <- tempfile("dumpsccs")
        on.exit(unlink(xfile))
        if(do.dd) {
                data.dump(x, xfile)
                suffix <- ".Q"
        }
        else {
                dump(x, xfile)
                suffix <- ".q"
        }
        cmd <- paste("diff -c ", xname, suffix,
                " ", xfile, sep = "")
        invisible(unix(cmd, out = F))
}
```

CODE NOTE. Actually this is misnamed since it doesn't depend on SCCS at all. It checks the difference between the current value of the object in S and a representation of it in a file in the current directory—it presumes that the file is not being changed behind the back of `sccs`.

3.6 Test Suites

Correct answers are nice. A step in that direction is knowing that today's answers are the same as last week's. The `verify` function—given below—does this. Here are some uses of `verify`. You have made some efficiency improvements to your functions, and you want to ensure that you get the same answers. You have moved to a different version of S, or a different version of the operating system, or a different architecture. You are making your code publicly available and it is desirable that the user has some assurance that they are getting good answers on their setup. A particularly vivid example is that the person may want to know if the code works under both S and R.

Tests for software are usually given less than their due. Testing ranges from formal test suites (like those that `verify` runs) to the completely informal. Although I think it is very important to have formal test suites for functions that you depend upon, probably the informal testing of functions as you write them is the most important testing that is performed. The informal testing should be fodder for the formal tests. I'll now discuss three types of tests: black box, white box and regression. I think it is good to know these distinctions, though it is not necessarily clear into which category a particular test falls.

The name "black-box test" comes from the idea that the person writing the test does not know anything about how the code is written—code need not even exist when the test is drafted. The tester merely knows what the code is supposed to do. If there is a specification for the code, then this is obviously a starting place for the tests. Documentation of the code—especially if written by some one other than the programmer—is a good source. The test should definitely include specific examples in the documentation (and hence it becomes a test of the documentation also). Implications that the documentation makes should be tested. Test the most common uses. Once that is done, try to get the test to hit as much functionality as feasible.

White-box testing means that the tester has knowledge of the code. Given knowledge of which S functions are used, tests may be based on weaknesses or common misuse of some of those functions. The structure of the code may imply areas where it is most vulnerable.

Regression tests are tests of bugs that have been fixed. It seems ludicrous, but these are actually very valuable. First off, reincarnation definitely exists for bugs—it is not at all uncommon for a bug that had been fixed to reappear. Also the bug may be an indication of a general weakness that the test might guard against. When you write a regression test, it is important that you test that the code is right, not that you don't get a particular wrong answer. That is, you want the test to be:

```
2 * 2 == 4  # good test
```

not

```
2 * 2 != 5  # poor test
```

Although there is no truly effective way to test software, there are some general principles that are good to follow.

Test along "seams". Just what this means depends on the situation, but it can mean values of 1 or 0 when positive integers are expected, zero length inputs, singular matrices, arrays that have one or more dimensions only 1 long, etc. For example, it was important to make sure that `soptions` (page 45) worked when there were no argument names.

Give the function garbage. This should not just be random garbage, it should be inputs that a naive user might feed the function. For instance, we may well have written the `sccs` function so that it only accepted a character vector containing the names of objects. If we had written it that way, then a "garbage" input would have been the unquoted name of a function.

A class of bugs to look for—with either black or white box tests—is "one-off" bugs. For instance it is easy to get the index wrong when translating between S and C, or to get the size of something wrong by not including both ends.

Another class of bug is "first-only". Sometimes a routine will work only the first time it is used in a session. Alternatively, it might not work the first time, but work subsequently. Bugs like this are the result of some global variable not operating properly.

Testing should be continuous. Expert programmers are always skeptical, they forever question if everything is making sense. *'tis our turn now To hear such tunes as killed the cow.* [11]

Let's think about what sort of informal testing should be done for the `sccs` function listed above. The first thing, of course, is to make sure that it works at all—so test it on a function that hasn't been put under control, then change that same function and use `sccs` again to make sure that it updates okay. One potentially flaky spot is that we allow as input both a character string containing the name of the object and the name itself. We want to make sure that the same behavior occurs when the input changes between these two forms. Still along this line we should look at what happens when the object that we want to put under SCCS control is character, and specifically that a character vector of length 1 can be properly put under control. Another potential trouble spot is when the name of the object is weird and we need to use the `xname` argument.

We may consider testing `sccs` thoroughly. To do this, we would want to test it on objects with and without troublesome names that were functions and non-functions, possibly with character data as a special case. We would want to test it with `new` being both TRUE and FALSE, and with `already.out` both TRUE and FALSE. So a simple, little function that doesn't do very much is going to require on the order of 10 test cases to cover all possibilities. This should give you some appreciation for the difficulty of testing a large piece of software.

`verify` is a generic function with two methods. The default method expects a vector of character strings, where each string is an S command. The default method returns an object of class `"verify"`. The other method of `verify` is for objects of class `verify` and it compares the current answers to the original answers to see if they have changed (significantly).

To start, give `verify` a character vector containing the commands that you want to test. In the test below, we give `verify` two arguments—the first is a character vector of commands, and the second is a list of the objects to be used in the commands. Both objects given as arguments have names. Names are optional for the commands—but convenient when there are a large number of commands in the test. It is mandatory to have names for the second argument.

```
> jjverif <- verify(c(sin="sin(x)", ran="runif(9)"),
+       list(x=1:4))
> jjverif
Passed:
 sin ran
  NA  NA
Commands:
      sin        ran
 "sin(x)" "runif(9)"
Names of data: x
Specifics:
version: Version 3.1 Release 1 for Sun SPARC, SunOS 4.x : 1992
Machine: revery
Date: Fri Dec 10 12:07:37 EST 1830
```

The answers from each test are available as input to subsequent tests by the name of `Test.` plus the name of the test or its index number:

```
> jjverif2 <- verify(c(sin="sin(x)", ran="runif(9)",
+      more="outer(Test.sin, Test.ran)"), list(x=1:4))
> jjverif2
Passed:
 sin ran more
  NA  NA   NA
Commands:
      sin        ran                  more
 "sin(x)" "runif(9)" "outer(Test.sin, Test.ran)"
Names of data: x
Specifics:
version: Version 3.1 Release 1 for Sun SPARC, SunOS 4.x : 1992
Machine: julia
Date: Sat Aug 24 12:08:37 GMT 1591
```

Once you have this original test object, you can use it as an argument to `verify` in whatever new environment you like. The result of the new call to `verify` will

tell you if any of the answers are different, and if so, then how. At some point, it may be that you decide that the old tests were wrong and that the new one is giving the correct answers.

```
> verify(jjverif)
Passed:
 sin ran
   T   T
Commands:
      sin        ran
 "sin(x)" "runif(9)"
Names of data: x
Specifics:
version: Version 3.3 Release 1 for Silicon Graphics Iris,
 IRIX 5.2 : 1995
Machine: azcan
Date: Thu Oct 2 12:14:09 EST 1879
```

The commands that are in your test suite should each cover as much functionality as possible—just as a line of good poetry packs the maximum amount of meaning. For example, testing `transcribe` is also a test of `perl` because `transcribe` calls `perl`. The suite as a whole should test all of the significant functionality, and anything that seems prone to being wrong.

There is the task of creating the initial vector of character strings of commands. I'll give the method I've used to do this. Create a clean directory with nothing in the `.Data`. Write a file containing the commands that you want to test, with each one assigned to `Test.`*something*. Objects that are to be passed into `verify`—like `variousnum` in my example—are assigned in this clean directory.

```
> !head poet.verif.q
Test.transcribe _ transcribe("state.name",".","_")
Test.loan _ loan(2000, .08, 2, 1997)
Test.update.loan _ update(Test.loan, rep(100,5))
Test.numbase _ numberbase(-9:9,2,10)
Test.ratnum _ rationalnum(variousnum, rep(variousnum, rep(16, 16)))
Test.ratnum.op _ Test.ratnum - 2 * Test.ratnum
Test.pg1 _ polygamma(1:10, "trig")
> source("poet.verif.q")
```

As you write this file, you can use `source` to evaluate the file of statements. At this stage you have the opportunity to see if the results are as they should be.

Once you are pleased, you need to make the character vector to feed to `verify`. Make a copy of the file so that you retain the original (to make additions), and edit the copy by taking out the assignments. Using an underscore

makes it easy to do the editing as long as any assignments within the test proper are not underscores. You can use another copy of the file to get the names to give to the commands.

```
> !cp poet.verif.q poet.verif.grab
> # edit poet.verif.grab
> q()
```

Now, go to the home of the code so that the verification object will live in the same directory.

```
> jjpv <- scan("../spo.test/poet.verif.grab", what="",
+     sep="\n")
> names(jjpv) <- jjvn
> poet.verif <- verify(jjpv, list(variousnum=get("variousnum",
+    where="../spo.test/.Data")))
```

Scan the commands into S so that each command is a separate element of the resulting character vector. You can then try this out on `verify` to see how it goes. It will, of course, work seamlessly.

```
> print(poet.verif, short=T)
Passed:
 transcribe loan update.loan numbase ratnum ratnum.op
         NA   NA          NA       NA     NA         NA
 pg1 pg2 pg3 digam1 digam2 digam3 digam4 digam5
  NA  NA  NA     NA     NA     NA     NA     NA
 expint1 expint2 mathgra getpath line.int lagrange
      NA       NA       NA       NA        NA        NA
 global.var sym.addr symsqrt stack.init stack.pop
         NA        NA       NA          NA         NA
 que.init que.pop quad.form twice.2
       NA      NA         NA       NA
Names of data: variousnum
Specifics:
version: Version 3.1 Release 1 for Sun SPARC, SunOS 4.x : 1992
Machine: plum
Date: Mon Sep 17 08:37:51 EST 1883
```

So now `poet.verif` is the verification object that users of the code can use at their whim with a command like:

```
> print(verify(poet.verif), short=T)
```

DANGER. This really was done under S-PLUS version 3.1, but the test will system terminate using the usual `verify.default`. I added a `cat` statement in

the loop, and it ran fine. This bug was fixed (I believe) in 3.2. Symptoms are that strange things happen in a loop, and changing something like putting in calls to `cat` make it disappear or change.

There is also subscripting of verify objects so that you can run only part of a test. For example, if you do not have an ANSI C compiler, you can do:

```
verify(poet.verif[-28])
```

to remove the one test that demands this.

Below is the listing for `verify.verify`, the `verify` method for objects of class `"verify"`. The default method is similar, though it has an extra argument for data.

```
"verify.verify"<-
function(x)
{
        commands <- attr(x, "commands")
        random.seed <- attr(x, "random.seed")
        if(length(random.seed)) {
                old.seed <- .Random.seed
                on.exit(.Random.seed <<- old.seed)
                .Random.seed <<- random.seed
        }
        n <- length(x)
        data <- attr(x, "data")
        passed <- ans <- vector("list", n)
        tnam <- xnam <- names(x)
        if(!length(tnam))
                tnam <- 1:n
        for(i in 1:n) {
                if(length(data)) {
                        ans[[i]] <- eval(parse(text =
                                commands[i]), data)
                }
                else {
                        ans[[i]] <- eval(parse(text =
                                commands[i]))
                }
                passed[[i]] <- all.equal(x[[i]], ans[[
                        i]])
                data[[paste("Test", tnam[i], sep = "."
                        )]] <- ans[[i]]
        }
```

```
        if(all(unlist(lapply(passed, mode)) ==
                "logical"))
                passed <- unlist(passed)
        names(passed) <- xnam
        length(data) <- length(data) - n
        specifics <- list(version = if(exists(
                "version")) version else NULL, machine
                 = unix("hostname"), date = date())
        attributes(ans) <- list(names = xnam, data =
                data, commands = commands, passed =
                passed, random.seed = random.seed,
                specifics = specifics, class =
                "verify")
        ans
}
```

Mainly this function loops over the commands in the test suite, evaluates them, adds each result to the list of available variables, and compares the current results to the old results. An important feature is that it takes care of getting the random seed right if the test involves any functions that use the S random generator.

3.7 Time Monitoring

There is one operation in S that is essentially instantaneous—assignment. The object is already there, so assignment merely adds the pointer for the name. So, in general, if an object is used twice or more in a function, it will be faster (and probably more memory efficient) to name the object.

A simple tool in S to monitor the time used by function calls is `unix.time`. Just give the command you want to time as the argument to `unix.time`. `unix.time` returns five numbers—the first two are different kinds of computer time and the last two are computer times for subprocesses; the third number is the elapsed (wall) time rounded to the nearest second. The last two numbers are zero for almost all commands. I have a personal function that reports the total computer time and the elapsed time—I find that more useful. *When I do count the clock* [12]

```
> unix.time(rationalnum(1:10, 3))
[1] 0.04000092 0.00000000 0.00000000 0.00000000 0.00000000
> unix.time(for(i in 1:100) rationalnum(1:10, 3))
[1] 3.7300014 0.1600001 9.0000000 0.0000000 0.0000000
> unix.time(for(i in 1:100) rationalnum(1:10, 3))
[1] 3.7200012 0.1499999 8.0000000 0.0000000 0.0000000
> unix.time(for(i in 1:100) rationalnum(1:10, 3))
```

```
[1] 3.7000008 0.1800001 8.0000000 0.0000000 0.0000000
```

In the first call the total time is just a fraction of a second. In order to get timing that is at all accurate, you need times that are several seconds. The usual trick is to use a `for` loop to call the command some number of times. The last three commands are all the same, and have similar timing, so we seem to be getting accurate timing (though 4 seconds is a little slim still). You might notice that the elapsed time is a lot more than the computer time. In this case it is because there are other users on the machine. Sometimes the elapsed time is greater even when the machine is doing nothing else. This implies that your command uses a significant amount of time that is not being accounted for, such as disk access time.

The `interlude` function monitors the number of calls to and the execution time of selected functions. This is useful to see where the time goes in long calculations. This is often called "profiling", but that name is already taken in S. The function is rather involved, so a discussion of the code for it is deferred until page 220.

This can be used either to see how much time particular functions take, or how many times they are called. The purpose in the first case is often to see where the computation time is going with the view of speeding things up. Sometimes you have very little idea of how often various functions get called. It is feasible to think that there are cases where knowing the number of calls to some function can suggest a more efficient algorithm.

I used `interlude` to striking advantage in one case. A function given a 2-dimensional problem ran in a few minutes, but it took eighteen hours to complete on a 30-dimensional problem when my calculations based on the 2-dimensional case showed that it should only take a couple hours. I immediately thought of paging as the reason, but an examination of the process showed that memory use was not excessive and paging was not the cause. Then I put `interlude` on the case, and it showed that `rank` was the source of the problem (there were lots of calls to it). I wrote a `rank.fast` function that ignored missing values (there were none in my application) and ties (which were rare and unimportant). The 30-dimensional problem then ran in about 30 minutes.

There are poor uses of this function as well as good ones—the "Code Tuning" chapter of Bentley (1986) discusses the issue from the point of view of a language like C. He points out that it is important to get the big picture right before mucking around in the details.

To use `interlude`, give it a vector of character strings naming the functions that you want to monitor. For instance:

```
> interlude(c("qr", "c", "rep", "seq"))
Warning messages:
1: assigning "qr" on database 0 masks an object of the
```

```
        same name on database 5
2: assigning "c" on database 0 masks an object of the
        same name on database 7
3: assigning "rep" on database 0 masks an object of
        the same name on database 5
4: assigning "seq" on database 0 masks an object of
        the same name on database 7
> jj <- lm(formula = freeny.y ~ freeny.x)
```

Execute as many commands as you like, and when you want to see the results, type:

```
> summary.interlude()
    total.time number.calls  mean.time
 qr 0.00000000            0         NA
  c 0.01666665            9 0.00185185
rep 0.00000000            0         NA
seq 0.08333325            1 0.08333325
```

Use `summary.interlude` whenever you like. You can remove some or all of the functions from being monitored with `uninterlude`. The default is to remove all the functions.

DANGER. `interlude` creates temporary versions of the functions that will last no longer than the present session. The `trace` function uses this same mechanism—it will not work to use both `trace` and `interlude` on a function at the same time.

For accurate timing, you should avoid monitoring functions that are used by other functions being monitored.

DANGER. You do not want to edit functions that are under `interlude` since you will get the `interlude` changes mixed into your copy of the function. It makes a mess.

3.8 Memory Monitoring

The amount of memory that an S function takes to run is generally the most important measure of its efficiency. It is also hard to understand. The amount of memory a function uses depends—sometimes heavily—on the state of memory at the time that the function is called. Seemingly trivial changes to a function can sometimes have dramatic effects on memory usage.

If you have a specific task that you are only doing once and you are running out of memory, then there are a few things that you can do. The first thing to try is to get out of S and start a fresh session—the command may work since the new S session is using less memory. If this doesn't work, then see if you can break the task into pieces that can be performed individually. For example, suppose that you are doing a linear regression

```
> ans <- lm(huge.y ~ huge.x1 + huge.x2)
```

and you are running out of memory. The `lm` function does quite a lot of data manipulation before it gets to the actual computation of the regression. We can do the data manipulation by hand, and then call the computational function directly:

```
> huge.xmat <- cbind("(Intercept)"=1, huge.x1, huge.x2)
> ans.alt <- lm.fit.qr(huge.xmat, huge.y)
```

The `ans.alt` object is without a few of the flourishes that `ans` has, but probably has all you need. For large objects the alternative method will use substantially less memory.

The remainder of this section presumes that you have a programming task that will be done repeatedly, and you want it to be memory efficient. It is the maximum memory that a function uses during the call that is important—it doesn't matter if that usage is throughout the entire function or just at one line of it.

DANGER. There is one idiom that is guaranteed to waste memory. Here is an example:

```
# AVOID THIS
ans <- NULL
for(i in 1:n) {
    ans <- c(ans, sum(n:i))
}
```

If you are worrying that this won't work because on the first iteration we are concatenating `NULL` with something, that is not a problem—it works fine. The problem is that this code fragments memory. Before the `for` loop, `ans` takes up a certain number of bytes (the size of the S header). On each pass through the loop, `ans` grows in size so it generally needs to move from where it has been living to a new location. This frees up the space it was using, but many objects will be too large for the hole so there will be numerous small holes in the memory which means that more total memory has to be used.

The example above can be written much more efficiently:

```
# better
ans <- numeric(n)
for(i in 1:n) {
    ans[i] <- sum(n:i)
}
```

Of course, in this example the truly best way would be to figure out the formula, and avoid the loop altogether.

When you don't know what the final length of the answer will be, then the second version will not work. If this is the case and the loop is not long, then I just use the adding-on technique. But if the loop is long and you are working with large objects, you will want to find an alternative.

There are less callous ways of fragmenting memory. Function `fjjloop` provides a simple example of a type of operation that is common in some of my work.

```
> fjjloop
function(x, backwards = F)
{
        nobs <- dim(x)[1]
        ans <- numeric(nobs)
        iseq <- 2:nobs
        if(backwards)
                iseq <- rev(iseq)
        for(i in iseq) {
                this.x <- x[1:i,  ]
                this.v <- var(this.x)
                ans[i] <- sum(this.v)
        }
        fjjcomma(memory.size())
}
```

The `fjjcomma` function is just a call to the S-PLUS `format` function so that commas are placed in numbers. (It needs S-PLUS version 3.2 or later.)

The key characteristic is that the data changes size throughout the course of the loop. It is also of importance that the order in which the iterations are performed can be changed. When the `backwards` argument is `FALSE`, then the data grows in size throughout the loop, and they shrink in size when `backwards` is `TRUE`. To test the memory use in these two cases, we issue each command as the first in a session of S.

```
> # in fresh session
> fjjloop(jjbig)
[1] "4,209,384"
```

```
> # in fresh session
> fjjloop(jjbig, back=T)
[1] "1,997,544"
```

Clearly the backwards order uses substantially less memory. (`jjbig` is 100 by 100 in this example, by the way.) The reason that backwards is superior is that on each iteration the new `this.x` can fit where the old one was, while fresh memory needs to be used when `this.x` is growing in size.

There are two main reasons for memory growth in S—garbage and copies. Garbage is created in loops as objects are continually created and destroyed. S has a garbage compactor that recovers most of the memory, but some garbage remains. Often the main expenditure of memory is because there are numerous copies of objects. S functions guarantee that they will not change the objects that were passed in as arguments (unless the programmer specifically programs that to happen), so each function tends to make its own copy of its arguments. One of the main things to do in managing memory is to try to reduce the number of copies of objects that are likely to be large.

The primary diagnostic for memory use is statements like

```
cat("at location 2, memory is", memory.size(), "\n")
```

that are scattered through the code. To use these reliably, each time you call the function you need to have just started the S session (possibly followed by a specific set of commands). You also need to use objects that are large enough (maybe on the order of a megabyte) so that you can see how many copies are being created.

Suppose that we have two matrices A and B where A is symmetric and we want to get the larger matrix that is partitioned as:

$$\begin{matrix} A & AB' \\ BA & BAB' \end{matrix}$$

If these matrices are large, then memory may be of concern. Let's look at some possible definitions of functions to do this and how they use memory. The first attempt is compact and simple to understand:

```
fjjaugsym1
function(a, b)
{
        part <- rbind(a, b %*% a)
        cbind(part, part %*% t(b))
}
```

Here are the objects to test the function with.

```
> dim(jja)
[1] 200 200
> dim(jjb)
[1] 100 200
> fjjcomma(object.size(jja))
[1] "320,124"
> fjjcomma(object.size(jjb))
[1] "160,124"
```

When testing memory use, it is useful to create objects at least this large so that significant changes in memory will be easily visible.

```
# New S Session
> jj <- fjjaugsym1(jja, jjb)
> fjjcomma(memory.size(T))
[1] "4,961,400"
> fjjcomma(object.size(jj))
[1] "720,124"
```

Let's do some accounting now. Inputs total about half a megabyte and there was about a half of a megabyte used before we got the first S prompt. So about 2 megabytes are accounted for, leaving about 3 megabytes as "wasted". If your function does anything at all, then there is going to be memory used in addition to the inputs and output—the aim is to keep the amount small.

The second attempt is significantly longer, but the logic is still simple. The idea is to create a matrix that is the size of the result, and then do replacements.

```
"fjjaugsym2"<-
function(a, b)
{
        da <- dim(a)
        db <- dim(b)
        newsize <- sum(db)
        ans <- array(NA, c(newsize, newsize))
        oseq <- 1:da[1]
        nseq <- 1:db[1] + da[1]
        ans[oseq, oseq] <- a
        part <- b %*% a
        ans[nseq, oseq] <- part
        part <- t(part)
        ans[oseq, nseq] <- part
        ans[nseq, nseq] <- b %*% part
        dna <- dimnames(a)[[1]]
        if(!length(dna))
                dna <- rep("", da[1])
        dnb <- dimnames(b)[[1]]
```

```
        if(!length(dnb))
                dnb <- rep("", db[1])
        newdn <- c(dna, dnb)
        if(sum(nchar(newdn)))
                dimnames(ans) <- list(newdn, newdn)
        ans
}
```

The first thing to do is check to make sure that it does the same thing as the previous version:

```
> all.equal(fjjaugsym2(var(freeny.x), matrix(1:8,2)),
+         fjjaugsym1(var(freeny.x), matrix(1:8,2)))
[1] T
```

Now test the memory use with our large objects:

```
# New S Session
> jj <- fjjaugsym2(jja, jjb)
> fjjcomma(memory.size(T))
[1] "3,470,456"
```

So this saves 1.5 megabytes over `fjjaugsym1`. A logical improvement to this latest version is to change the `NA` in the call to `array` to `as.double(NA)`. This way `ans` will be initialized to be its final size instead of half as big. However in this test, it actually uses more memory—I don't understand why.

Some of the lines in the function can be permuted which may have consequences for memory use. Obviously the same amount is going to be written to memory—the issue is how efficiently memory is recycled. I didn't try very hard, but I didn't find a better order than the one given.

While memory isn't cleaned up well within a function, all of the memory is released (except that used by the result) when the function exits. This can be used to advantage at times in managing memory by putting some calculations into a subfunction. There's a catch—the memory you save must more than compensate for any extra copies that you create when you increase the depth of the function calls. Encapsulating most of the contents of a loop into a subfunction often helps relieve memory growth.

Loops in S return a value, though this value is seldom used. At times memory use can be reduced by making the return value of a loop be something small (like `NULL`) rather than the large object that happens to be the value. Version 4.0 of S-PLUS eliminates the return value of loops.

Below I show the results of an attempt (slightly unsuccessful) at reducing memory use of a test function. The subfunction principle and the `NULL` value for the loop were tried separately and together. Here are the function definitions:

```
> fjjmem1
function(iter = 10, n = 100)
{
        ans <- vector("list", iter)
        for(i in 1:iter) {
                cat("i is", i, "memory.size is",
                        memory.size(), "\n")
                x <- outer(rep(1, n), rep(5, n))
                ans[[i]] <- x %*% x
        }
        NULL
}
> fjjmem2
function(iter = 10, n = 100)
{
        ans <- vector("list", iter)
        for(i in 1:iter) {
                cat("i is", i, "memory.size is",
                        memory.size(), "\n")
                x <- outer(rep(1, n), rep(5, n))
                ans[[i]] <- x %*% x
                NULL
        }
        NULL
}
> fjjmem3
function(iter = 10, n = 100)
{
        subfun <- function(n)
        {
                x <- outer(rep(1, n), rep(5, n))
                x %*% x
        }
        ans <- vector("list", iter)
        for(i in 1:iter) {
                cat("i is", i, "memory.size is",
                        memory.size(), "\n")
                ans[[i]] <- subfun(n)
        }
        NULL
}
> fjjmem4
function(iter = 10, n = 100)
{
        subfun <- function(n)
        {
```

```
                x <- outer(rep(1, n), rep(5, n))
                x %*% x
        }
        ans <- vector("list", iter)
        for(i in 1:iter) {
                cat("i is", i, "memory.size is",
                        memory.size(), "\n")
                ans[[i]] <- subfun(n)
                NULL
        }
        NULL
}
```

Here are the results using S-PLUS version 3.3 on a Silicon Graphics. Function `fjjmem1` performed the same as `fjjmem2`:

```
# new S session
> fjjmem2(n=300)
i is 1 memory.size is 544584
i is 2 memory.size is 6381384
i is 3 memory.size is 7823176
i is 4 memory.size is 7823176
i is 5 memory.size is 9264968
i is 6 memory.size is 10706760
i is 7 memory.size is 12148552
i is 8 memory.size is 12148552
i is 9 memory.size is 13590344
i is 10 memory.size is 13590344

# new S session
> fjjmem3(n=300) # subfun
i is 1 memory.size is 544584
i is 2 memory.size is 6401864
i is 3 memory.size is 7843656
i is 4 memory.size is 7843656
i is 5 memory.size is 8564552
i is 6 memory.size is 9285448
i is 7 memory.size is 10006344
i is 8 memory.size is 10727240
i is 9 memory.size is 11448136
i is 10 memory.size is 12169032

# new S session
> fjjmem4(n=300) # subfun and NULL return
i is 1 memory.size is 544584
i is 2 memory.size is 6401864
```

```
i is 3 memory.size is 7843656
i is 4 memory.size is 8564552
i is 5 memory.size is 9285448
i is 6 memory.size is 10006344
i is 7 memory.size is 10727240
i is 8 memory.size is 11448136
i is 9 memory.size is 12169032
i is 10 memory.size is 12889928
```

The subfunction saved a minor amount of memory. Notice that putting the `NULL` at the end of the `for` loop, which is supposed to reduce memory use, actually increased it when the subfunction is used. This is part of the frustration. There were no differences in memory among these four functions using S-PLUS version 3.1 on a Sparc (the memory use there was slightly higher than here).

3.9 Things to Do

Decide what naming convention for S objects makes the most sense for you, and rename your objects to fit it.

Decide how your S objects should be divided, and do it.

Put your important files and S objects under source code control, and create `verify` objects for your important S functions.

Write a version of `soptions` that guarantees there will be a warning message printed when a new option is added. Which functionality is better?

Test the memory and time use of some of your functions.

3.10 Further Reading

The literature on software testing is wide and varied, and contains no magic bullets that I know of. One interesting little book is Davis (1995) *201 Principles of Software Development*.

3.11 Quotations

[10]Percy Bysshe Shelley "Ozymandias"

[11]A. E. Housman "Terrence, This is Stupid Stuff ..."

[12]William Shakespeare (Sonnet 12)

Chapter 4

Vocabulary

This chapter introduces some of the more important functions to know. Most important is that you can find information on functions for yourself. If you are running S-PLUS on a window system, then you should use the `help.start` function before requesting help. There are two commands for getting help on a function or other object—the following are equivalent:

```
help(match)
?match
```

You do not need to surround the name with quotes, although you may. Neither do you need quotes when you ask for the name of a help file that is not an S object:

```
help(Syntax)
```

In addition to the "Syntax" and "Devices" help files, the two most useful help files of this sort in S-PLUS are "Release.Notes" and "Command.edit".

You do need quotes around "illegal" names:

```
> ?break
? only meaningful if followed by name or call
Dumped
> ??
? only meaningful if followed by name or call
Dumped
> ?%%
Syntax error: sp.op ("%%") used illegally at this point:
?%%
> ?"?"
> ?  # same as last command
```

For the most part, `help` and `?` are synonyms, but `?` takes the special form "methods" which gives you a list of possible methods for a generic function. (It just looks at the names, it doesn't check if they really are methods or not.)

```
> ?methods(summary)
The following are possible methods for summary
        Select any for which you want to see documentation:

1: summary.aov
2: summary.aovlist
3: summary.data.frame
4: summary.default
5: summary.factor
6: summary.gam
7: summary.glm
8: summary.interlude
9: summary.lm
10: summary.loess
11: summary.mlm
12: summary.model.matrix
13: summary.ms
14: summary.nls
15: summary.ordered
16: summary.tree
17: summary.varcomp
Selection:
```

The result is a menu. Give the number of any object you want to get help for, type 0 to exit. See more on the `menu` function on page 106.

S-PLUS has a `methods` function which mimics the "?methods" form, but you get a character vector instead of a menu of options. The `methods` function also has a `class` argument so that you can discover which generic functions have methods for a specific class:

```
> methods(class="numberbase")
                    .Data                .Data
 "numberbase.numberbase" "print.numberbase"
```

Functions are categorized in the S-PLUS Reference Manual, in the S-PLUS help window, and in Appendix 3 of Becker, Chambers and Wilks (1988). Use one of these when you know what type of functionality you want but do not know the name of the function that performs it. The categorization could use some improvement, but is still useful.

If your system is properly configured, you can get hard-copy versions of help files (including your own) via the `help` function.

4.1 Efficiency

Improving the efficiency of your S functions can be well worth some effort. Here are a few functions that can often be used to make your functions more direct. But remember that large efficiency gains can be made by using a better algorithm, not just by coding the same algorithm better. *when the cities lie at the monster's feet there are left the mountains.* [13]

Whenever you have a number of elements that are to be selected from a group of elements, `match` should come to mind. It can be used to avoid some loops.

The `match` function has three arguments `x`, `table` and `nomatch` (the S-PLUS version also has an `incomparables` argument). It returns a numeric vector as long as `x`. The ith element of the result is the index of `table` that matches the ith element of `x`. `nomatch` states what should be used when the element of `x` doesn't match anything in `table`.

Although straightforward, `match` can be employed in a variety of ways—I always use the fact that the result has the same length as `x` as a hint about how to use it in a given situation.

Suppose that we have two character vectors `many.strings` and `good.strings` and we want the elements of `many.strings` that are in `good.strings`. This is done with:

```
many.strings[match(good.strings, many.strings, nomatch=0)]
```

A specific example is:

```
> letters[match(c("e","D","ff","z"), letters, nomatch=0)]
[1] "e" "z"
```

Let's think about what is happening. `match` gives indices for its second argument—which is `many.strings` which is the vector being subscripted—so we have hope of getting the right subscripts. We know that the correct answer can't be longer than `good.strings`, and since that is the first argument to `match`, that must be true. So it is certainly a feasible solution, and in fact it isn't hard to see that it really is doing what we want.

Now suppose that we have a named vector `the.data` and we want to remove any elements that have names in `bad.ones` (see the `soptions` function on page 45 for a similar situation). Here is one way of getting this:

```
the.data[!match(names(the.data), bad.ones, nomatch=0)]
```

`match` returns a numeric vector that contains zeros for elements that do not match and a non-zero for those that do match a `bad.ones` element. The "bang" operator coerces this to logical so the subscripting vector is logical and what we want. Here is an example:

```
> jjwt
 dorothy harold munchkin stevie
      14     15        2      4
> jjwt[!match(names(jjwt), c("stevie","esther"), nomatch=0)]
 dorothy harold munchkin
      14     15        2
```

DANGER. An almost equivalent version is:

```
the.data[ -match(bad.ones, names(the.data), nomatch=0)]
```

This works only as long as at least one match exists. First, let's look at what happens when it is okay. `match` gives indices for the names of `the.data` that are in `bad.ones` plus possibly some zeros; these are made negative and used in subscripting. The subscripting ignores the zeros and removes elements corresponding to the negative subscripts. When there are no matches, then the subscripting vector is all negative zeros, which are thought to be positive zeros, so the result of the whole operation is a vector of length zero, rather than the whole of `the.data` as we want.

DANGER. If the elements of the `table` argument are not unique, only the first occurrence is used.

```
> match(c("here","aa","b","here"), c("here",letters,"here"))
[1]  1 NA  3  1
```

An example of when this might come into play is selecting all of the rows of a matrix that have a column matching some values. As here where we want the rows of the matrix where the first column contains either a 1 or a 2.

```
> jjmatr <- cbind(rep(1:3,4), rep(1:4,rep(3,4)))
> jjmatr[match(1:2, jjmatr[,1]),] # wrong
     [,1] [,2]
[1,]    1    1
[2,]    2    1
> jjmatr[!is.na(match(jjmatr[,1], 1:2, no=NA)),] # right
     [,1] [,2]
[1,]    1    1
[2,]    2    1
[3,]    1    2
[4,]    2    2
[5,]    1    3
[6,]    2    3
```

```
[7,]    1    4
[8,]    2    4
```

Here is a function that uses `match`.

```
"p.replace"<-
function(x, old, new)
{
        if(length(old) != length(new))
                stop("old and new must be same length")
        x.ind <- match(x, old, nomatch = 0)
        x[as.logical(x.ind)] <- new[x.ind[x.ind != 0]]
        x
}
```

This substitutes values from the `new` argument for values in the `old` argument. Notice that the name begins with "p." which is my personal prefix for modified in-built functions (see page 41). There is an in-built `replace`, but even if there hadn't been, I would have wanted to put my prefix on since "replace" seems such a common name.

An example of its use is:

```
> p.replace(c("The", "corgi", "pranced", "by", "the", "corgi"),
+   old=c("corgi", "pranced"), new=c("kitty", "strolled"))
[1] "The"      "kitty"    "strolled" "by"       "the"      "kitty"
```

For substituting characters within strings, see the `transcribe` function on page 276.

The `match` function is used in many spots, including the `to.base10` function (page 236).

Functionality related to that of `match` exists in `pmatch`, `charmatch` (both discussed on page 94) and `grep` (page 83).

I find the `outer` function one of the most interesting in S. The statement

```
outer(x, y)
```

produces the equivalent of the mathematical statement xy' (resulting in a matrix) when x and y are vectors. `outer` is much more general than this though. First off, it can use any function that is vectorized for two arguments, not just the default multiplication. An example that might be surprising is

```
jj2let <- outer(letters, letters, paste, sep="")
```

This produces a 26 by 26 matrix containing each two letter combination. (Additional arguments to the function can be given—the `sep=""` is passed to the `paste` call.)

The first two arguments to `outer` may be arrays as well as plain vectors, the result is an array with dimension equal to

```
c(dim(as.array(x)), dim(as.array(y)))
```

So the command

```
outer(jj2let, letters, paste, sep="")
```

produces a 26 by 26 by 26 array of all the combinations of three letters.

There are times when using `outer` cuts down on execution time, but increases memory use—one of those unfortunate times when you must decide between evils. The `interpolator.lagrange` function (page 325) and `exp.integral` (page 272) provide examples of `outer`.

`sweep` also involves arrays. If you want each element of a vector to be subtracted from the corresponding row of a matrix, then you can just rely on vectorization and automatic replication to do that:

```
> jjmat
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    4    7   10   13
[2,]    2    5    8   11   14
[3,]    3    6    9   12   15
> jjmat - 1:3
     [,1] [,2] [,3] [,4] [,5]
[1,]    0    3    6    9   12
[2,]    0    3    6    9   12
[3,]    0    3    6    9   12
```

However, if you want the operation to be relative to columns, then `sweep` comes to the rescue:

```
> sweep(jjmat, 2, 1:5, "-")
     [,1] [,2] [,3] [,4] [,5]
[1,]    0    2    4    6    8
[2,]    1    3    5    7    9
[3,]    2    4    6    8   10
```

The "2" gives the dimension to work on—like the `MARGIN` argument to `apply` (see the next section). The `1:5` is the vector to use as the second argument to the function, and subtraction is the function being used.

`sweep` can handle higher dimensional arrays, and any function that is vectorized on two arguments.

When you have an atomic vector and you want to make it into a list with each element of the vector in some component of the list, then use `split`. An example is

```
> split(state.name[1:9], state.region[1:9])
$Northeast:
[1] "Connecticut"

$South:
[1] "Alabama"  "Arkansas" "Delaware" "Florida"

$"North Central":
character(0)

$West:
[1] "Alaska"     "Arizona"    "California" "Colorado"
```

`state.region` is a category that corresponds to the `state.name` vector. This produces a list that has one component for each category in `state.region`, and each of the components is a vector of the state names within the region.

4.2 The Apply Family

The functions `apply`, `lapply`, `sapply` and `tapply` all apply a function to sections of some object. These are used in lieu of a `for` loop. In general, the appropriate apply function is faster—often dramatically so—than the equivalent loop.

Each of the four apply functions have an argument named `FUN` which can be either a function or the name of a function. All three of the following are essentially equivalent:

```
lapply(jjl, "mean")
lapply(jjl, mean)
lapply(jjl, function(x) sum(x)/length(x))
```

The first two are the same in the sense that the same function is applied. The second two are the same in the sense that a function (rather than the name of a function) is being passed into `lapply` as the `FUN` argument.

Each apply function also accepts an arbitrary number of additional arguments to be given to `FUN`. There's an example of this on page 28.

The `apply` function is for arrays. When you use `apply`, the `MARGIN` argument specifies the dimensions that you want to keep. So a statement like

```
apply(xmat, 2, myfun)
```

saves the second dimension while the first dimension (and any others) are "collapsed" into the function call. For higher dimensional arrays, the `MARGIN` can have length greater than one; these again are the dimensions to be saved.

DANGER. When the function used in `apply` returns a vector, then that becomes the first dimension of the array that is returned. So a command like:

```
apply(freeny.x, 1, sort)
```

will return the transpose of what you would naively expect. The `stable.apply` function on page 287 can be used to keep the dimensions of the result in the same order as those of the input.

In recent versions of S-PLUS, the apply functions offer a speed advantage over the equivalent `for` loop. Here is a function to sort the columns of a matrix using a loop and an equivalent function using `apply`.

```
"fjjcsort1"<-
function(x)
{
        for(i in 1:ncol(x)) {
                x[, i] <- sort(x[, i])
        }
        x
}

fjjcsort2
function(x)
{
        apply(x, 2, sort)
}
```

Note that it seems unlikely that you would really want to do such an operation. As always, we want to test that they do the same thing—no matter how obvious.

```
> all.equal(fjjcsort1(freeny.x), fjjcsort2(freeny.x))
[1] T
> dim(jjm)
[1]   20 1000
> dim(jjm2)
[1]    20 10000
```

Using two different sizes of matrix we time these two versions.

```
> p.unix.time(for(i in 1:10) fjjcsort1(jjm))
 comp time clock time  memory
     42.24         43 3088936
> p.unix.time(for(i in 1:10) fjjcsort2(jjm))
 comp time clock time  memory
     34.02         35 3695144
> p.unix.time(fjjcsort1(jjm2))
 comp time clock time  memory
     45.49         46 7459368
> p.unix.time(fjjcsort2(jjm2))
 comp time clock time   memory
     39.12         40 21688872
```

We can see that `apply` does indeed improve the speed. There is an indication, though, that memory use could be a problem with the `apply` version.

Always it is going to be more efficient if you can vectorize the problem. This problem seems unamenable to doing it all at once, but there is a trick that works (that I wish I'd thought of myself).

```
"fjjcsort3"<-
function(x)
{
        xatt <- attributes(x)
        ans <- x[order(col(x), x)]
        attributes(ans) <- xatt
        ans
}
```

After making sure that it does what we want, we time it like the others:

```
> p.unix.time(for(i in 1:10) fjjcsort3(jjm))
 comp time clock time  memory
  9.130001          9 2875944
> p.unix.time(fjjcsort3(jjm2))
 comp time clock time   memory
     13.25         14 14856744
```

Now we have some substantial speed improvement.

`lapply` applies a function to each component of a list:

```
> lapply(list(a=jjm, b=t(jjm)), dim)
$a:
[1] 5 3

$b:
[1] 3 5
```

`sapply` is used in the same situations as `lapply`, but generally returns a simplified object. When the function being applied returns a single number in each case, then the result of `sapply` is the same as `unlist` of the same call to `lapply`. If the function returns vectors that are all one length, then these two are different:

```
> unlist(lapply(list(a=jjm, b=t(jjm)), dim))
 a1 a2 b1 b2
  5  3  3  5
> sapply(list(a=jjm, b=t(jjm)), dim)
     a b
[1,] 5 3
[2,] 3 5
```

`sapply` produces a matrix when all of the results are the same length. If the results vary in length, then `sapply` is the same as `lapply`.

DANGER. Internal generic functions (see page 225) do not work correctly in `lapply` or `sapply`.

```
> lapply(list(jjm, as.data.frame(jjm)), dim)
[[1]]:
[1] 5 3

[[2]]:
NULL
```

The workaround is to write a wrapper function:

```
> lapply(list(jjm, as.data.frame(jjm)),
+       function(x) dim(x))
[[1]]:
[1] 5 3

[[2]]:
[1] 5 3
```

This bug has sometimes been attributed to the fact that `lapply` is internal in recent versions of S-PLUS, but the output above was from Version 3.1 where `lapply` uses a `for` loop.

`lapply` and `sapply` can take a vector as their first argument. One use of this is to apply functions where you want more than one of the arguments to change. The following example assumes that we have two lists of matrices and we want the result to be a list whose components are the products of the corresponding components of the original lists.

```
> lapply(seq(along=jjalist), function(i, x, y)
+         x[[i]] %*% y[[i]], x=jjalist, y=jjblist)
```

The trick is to write a function whose first argument is used to subscript the other arguments.

The following three vectors are used in the discussion of `tapply`.

```
> jjwt
 dorothy harold munchkin stevie
      14     15        2      4
> jjgender
[1] "female" "male"   "female" "female"
> jjtype
[1] "corgi" "corgi" "cat"   "cat"
```

`tapply` is used in situations where you have one or more categories, but there is not necessarily an equal number of points in each combination of categories. This is sometimes called a ragged array. Here we look at the mean of `jjwt` within each combination:

```
> tapply(jjwt, list(jjgender, jjtype), mean)
       cat corgi
female   3    14
  male  NA    15
> mode(.Last.value)
[1] "numeric"
```

This is a typical numeric matrix, but with a missing value because there are no male cats. However, the result of the following command gives us a matrix of the same shape again, but this time the mode of the matrix is `list`:

```
> jjlistmat <- tapply(jjwt, list(jjgender, jjtype),
+         function(x) x)
> jjlistmat
             cat corgi
female numeric, 2 14
  male NULL, 0    15
> jjlistmat["female", "cat"]
[[1]]:
 munchkin stevie
        2      4

> jjlistmat["female", "cat"][[1]]
 munchkin stevie
        2      4
```

```
> jjlistmat["female", "corgi"]
[[1]]:
 dorothy
      14
```

Version 3.4 of S-PLUS added a `simplify` argument to `tapply` which you can set to `FALSE` to ensure that you always get an object of mode `list`.

The `FUN` argument is optional in `tapply`. When `FUN` is not given, then the result is a vector of indices into the array that is created when `FUN` is given. (Logically, the first argument could be optional when `FUN` is missing since its values are not used, but it is used for its length, and hence still required.) Here we use `tapply` without `FUN` to produce a vector to be used by `split`:

```
> split(jjwt, tapply(jjwt, list(jjgender, jjtype)))
$"1":
 munchkin stevie
        2      4

$"3":
 dorothy
      14

$"4":
 harold
     15
```

The names of this list show that `tapply` did return the indices for `jjlistmat`. The last two results are essentially equivalent.

We can make the list that `split` creates look more inviting by changing the category we give it:

```
> split(jjwt, paste(jjgender, jjtype))
$"female cat":
 munchkin stevie
        2      4

$"female corgi":
 dorothy
      14

$"male corgi":
 harold
     15
```

An alternative to `tapply` that is in S-PLUS is `by`. This is object-oriented and can do some things that `tapply` can't.

4.3 Pedestrian

First is a list of the truly pedestrian, few of these functions will be discussed more than briefly. *The apparition of these faces in the crowd;* [14]

```
abs sqrt exp log log10 sum prod cumsum cumprod min max
pmin pmax cummin cummax range gamma lgamma

round signif trunc ceiling floor

sin cos tan asin acos atan
sinh cosh tanh asinh acosh atanh

Re Im Mod Arg Conj

length c list unlist names attributes attr array
dim dimnames unique duplicated

matrix nrow ncol row col rbind cbind t crossprod
solve backsolve eigen svd qr chol kronecker

scale interp approx fft
```

Most of the mathematical functions accept complex as well as numeric inputs. Functions that are S-PLUS additions that may not be in other versions are `cumprod`, `cummax`, `cummin` and `kronecker`.

The `rep` function—one of the most useful in S—was discussed earlier. If you missed it, go back to page 29.

The `order` function produces the vector of indices that will sort a vector. That is,

```
x[order(x)]
```

is the same as

```
sort(x)
```

`order` is useful when you want to sort several vectors in parallel:

```
xord <- order(x)
xsort <- x[xord]
ysort <- y[xord]
```

`xsort` is actually sorted; `ysort` probably isn't sorted but it is in the order that made `x` sorted. A more common example is to sort the rows of a matrix relative to one of the columns:

```
xmat[order(xmat[,1]), ]
```

If the order to use depends on more than one vector—with secondary vectors breaking ties—then `order` is also used:

```
xyzord <- order(x, y, z)
```

An example of this use is in the `fjjcsort3` function on page 77.

To reverse the order of a vector or list, use the `rev` function. So you might have

```
xord.bigfirst <- rev(order(x))
```

`paste` is very useful for dealing with character data. Any number of arguments can be given to it plus the two special arguments `sep` and `collapse` (which must be given by their full names). A character vector is created which is as long as the longest argument, shorter arguments are replicated. The `sep` argument is a character string that is placed at each spot that is pasted together. The `collapse` argument turns the result into a single string. The command

```
paste(1:9, "000", sep=",", collapse=" ")
```

results in the single string

```
"1,000 2,000 3,000 4,000 5,000 6,000 7,000 8,000 9,000"
```

`nchar` returns a vector which is the count of the number of characters in each element of its argument. For example

```
nchar(state.name)
```

is a numeric vector of length 50 containing the number of letters (and spaces) in each state name.

DANGER. Note that backslash-something counts as just one character, not the number of keystrokes that you need to type to get it.

`substring` takes a character vector, a numeric vector of the first character to be used, and a numeric vector of the last character to be used (which defaults to some large number). These arguments are replicated to each be as long as the longest, and the result is a vector of substrings. To break a string into individual characters do:

```
substring(x, 1:nchar(x), 1:nchar(x))
```

If for a particular element `last` is less than `first`, then the result for that element will be the empty string. In particular, `last` can be zero.

`grep` returns the indices of a character vector (second argument) that match the pattern (first argument). When you use the `grep` function, it is tempting to think of Unix wildcarding as with Unix `ls`. This is especially true when using the `pattern` argument to `objects` which creates a call to `grep`. But instead you need to think of regular expressions.

DANGER. The behavior of `grep` may be somewhat machine dependent.

```
> jjgreptest
[1] "abc"     "a.bc"    "a\\bc"   "a\tbc" "aabbcc"

> grep(".", jjgreptest) # finds everyone
[1] 1 2 3 4 5
> grep("\.", jjgreptest) # still finds everyone
[1] 1 2 3 4 5
> grep("\\.", jjgreptest) # finds the dot
[1] 2
```

Now I search for backslashes:

```
> grep("\\\\\\\\", jjgreptest)
[1] 3
```

There are 8 backslashes in the command above—escape characters grow in powers of two.

```
> grep("\\\t", jjgreptest)
[1] 4
```

DANGER. If there are any newlines in your strings, you are in trouble:

```
> jjgreptest2
[1] "abc"     "a.bc"    "a\\bc"   "a\nbc" "aabbcc"
> length(jjgreptest2)
[1] 5
> grep("bc", jjgreptest2)
[1] 1 2 3 5 6
```

It is a little unusual to see something in the $6th$ element of a 5-element vector.

A couple of specialized S functions that deal with character strings are `abbreviate` and `make.names`.

Access Unix functionality with the `unix` function. The first argument is a character string that gives the Unix command. The `output` argument is an important control:

```
> jjd1 <- unix("date")
> jjd1
[1] "Wed Mar 26 15:22:23 PST 1874"
> jjd2 <- unix("date", out=F)
Mon Nov 28 15:22:45 GMT 1757
> jjd2
[1] 0
```

When `output` is TRUE (the default), then the output of the Unix command is captured in an S character vector (one element for each line of the output) and returned. If `output` is FALSE, then the output of the command goes to standard-out and the return value in S is the status of the Unix command. The Unix convention is that 0 means no problem and a non-zero value means some type of error. So S code might look something like:

```
if(status <- unix(cmd, out=F))
     stop(paste("Unix trouble, code", status))
```

If you have several commands, you need to put them all into one string with the commands separated by semicolons or newlines:

```
> cmds <- c("date", "echo $SHOME")
> unix(paste(cmds, collapse=";"), out=F)
Sun Oct 14 20:51:20 EST 1894
/usr/lang/s
[1] 0
```

There is also an `input` argument to `unix`. This is seldom used since usually inputs are contained in the command argument. The `perl` function (page 275) provides a good example of the use of the `input` argument.

In S-PLUS there is also the `unix.shell` function which allows you to select the shell that will be used rather than being forced to use the Bourne shell:

```
> unix("echo $shell")
[1] ""
> unix.shell("echo $shell", shell="/bin/sh")
```

```
[1] ""
> unix.shell("echo $shell", shell="/bin/csh")
[1] "/bin/csh"
```

Arrays are generally created with the `array` function. You may create matrices either with `array` or with `matrix`. The advantage of `array` is that it is slightly faster (`matrix` calls `array`). The advantages of `matrix` are that you only need to know one of the dimensions and you can choose to fill along either columns or rows.

The `diag` function also creates matrices. The behavior of this function is highly dependent on its argument. if you give it a single number, it returns an identity matrix of that size:

```
> diag(3)
     [,1] [,2] [,3]
[1,]    1    0    0
[2,]    0    1    0
[3,]    0    0    1
```

If you give it a vector, it returns a square matrix with that vector along its diagonal:

```
> diag(c(4.3, 5.7, -2.9))
     [,1] [,2] [,3]
[1,]  4.3  0.0  0.0
[2,]  0.0  5.7  0.0
[3,]  0.0  0.0 -2.9
```

If you give it a matrix, it returns the diagonal of the matrix:

```
> diag(freeny.x)
[1]  8.79636  4.70217  5.83112 12.98060
> diag(as.matrix(3))
[1] 3
```

There are arguments to control the shape of the result, and there is an assignment form to change the diagonal of a matrix.

DANGER. When an array with three or more dimensions is given to `diag`, then it is treated as a simple vector (creates a large diagonal matrix) rather than extracting some definition of the diagonal from the array.

`lower.tri` returns a logical matrix the same size as its argument. Its value is TRUE for all elements below the diagonal. The S-PLUS (version 3.2 and later) version of the function has a `diag` argument that allows the diagonal elements to be TRUE also. Similar functionality can be achieved with `row` and `col`; for example, to get the super-diagonal do:

```
> row(jjmat) - col(jjmat) == -1
     [,1] [,2] [,3] [,4] [,5]
[1,]    F    T    F    F    F
[2,]    F    F    T    F    F
[3,]    F    F    F    T    F
[4,]    F    F    F    F    T
[5,]    F    F    F    F    F
```

A command like

```
solve(A) %*% x # avoid this
```

is an inefficient use of `solve`. The reason that `solve` is not called something like `invert` is precisely this case. We don't really care about the inverse of `A`, we only want the final answer. The more efficient command is:

```
solve(A, x)
```

`backsolve` is available when you have a triangular set of equations to solve.

Given one or more vectors that can be thought of as categories, the `table` function counts up the number of occurrences in each combination. It is called "table" because of the statistical concept of a contingency table, but non-statisticians may think of it as a binning function.

```
> table(jjgender)
 female male
      3    1
> table(jjtype)
 cat corgi
   2     2
> table(jjgender, jjtype)
       cat corgi
female   2     1
  male   0     1
```

Somewhat related is the `cut` function. While `table` counts the number of occurrences of categories, `cut` breaks numerical data into categories. This is another sense of binning.

DANGER. Do not abbreviate the `breaks` argument of `cut` to `break`:

```
> cut(jjwt, break=seq(0,25,by=5))
Syntax error: "=" used illegally at this point:
cut(jjwt, break=
```

`break` is a reserved word in S, so the parser is ever on the lookout for it. Any of the following forms are acceptable though:

```
> cut(jjwt, brea=seq(0,25,by=5))
[1] 3 3 1 1
attr(, "levels"):
[1] " 0+ thru  5" " 5+ thru 10" "10+ thru 15"
[4] "15+ thru 20" "20+ thru 25"
> cut(jjwt, breaks=seq(0,25,by=5))
[1] 3 3 1 1
attr(, "levels"):
[1] " 0+ thru  5" " 5+ thru 10" "10+ thru 15"
[4] "15+ thru 20" "20+ thru 25"
> cut(jjwt, "break"=seq(0,25,by=5))
[1] 3 3 1 1
attr(, "levels"):
[1] " 0+ thru  5" " 5+ thru 10" "10+ thru 15"
[4] "15+ thru 20" "20+ thru 25"
```

Order dependent counting is done with the S-PLUS function `rle` (run-length encoding).

The `diff` function performs differencing of successive elements. When `x` is a vector, then

```
diff(x)
```

is equivalent to

```
x[-1] - x[-length(x)]
```

however, `diff` has arguments to control the number of differences, and the lag. Also `diff` does each column of a matrix separately.

`aperm` stands for array dimension permutation. Consider the three-dimensional array:

```
> jja

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

, , 2
     [,1] [,2] [,3]
[1,]    7    9   11
[2,]    8   10   12

, , 3
     [,1] [,2] [,3]
[1,]   13   15   17
[2,]   14   16   18

, , 4
     [,1] [,2] [,3]
[1,]   19   21   23
[2,]   20   22   24
> dim(jja)
[1] 2 3 4
```

`aperm` is usually used with two arguments—the array to be permuted and the permutation. The ith element of the permutation vector is the dimension in the current array that will be the ith dimension in the resulting array.

```
> aperm(jja, c(2,3,1))

, , 1
     [,1] [,2] [,3] [,4]
[1,]    1    7   13   19
[2,]    3    9   15   21
[3,]    5   11   17   23

, , 2
     [,1] [,2] [,3] [,4]
[1,]    2    8   14   20
[2,]    4   10   16   22
[3,]    6   12   18   24
```

If the ith element of the vector that you have contains the dimension in the result that is the ith current dimension, then use `order` on the vector:

```
> aperm(jja, order(c(2,3,1)))
```

```
, , 1
     [,1] [,2]
[1,]    1    2
[2,]    7    8
[3,]   13   14
[4,]   19   20

, , 2
     [,1] [,2]
[1,]    3    4
[2,]    9   10
[3,]   15   16
[4,]   21   22

, , 3
     [,1] [,2]
[1,]    5    6
[2,]   11   12
[3,]   17   18
[4,]   23   24
```

Let's review. If the values are for the input and the indices are for the result, then the vector is appropriate to use as the `perm` argument. If the values are for the result and the indices are for the input, then you need to use `order`.

Be careful when testing with three-dimensional arrays since $(2, 3, 1)$ and $(3, 1, 2)$ are the only permutations that do not map to themselves.

```
> jjperm3
     [,1] [,2] [,3] [,4] [,5] [,6]
[1,]    1    1    2    2    3    3
[2,]    2    3    1    3    1    2
[3,]    3    2    3    1    2    1
> jjperm3o <- apply(jjperm3, 2, order)
> jjperm3o
     [,1] [,2] [,3] [,4] [,5] [,6]
[1,]    1    1    2    3    2    3
[2,]    2    3    1    1    3    2
[3,]    3    2    3    2    1    1
> apply(jjperm3 != jjperm3o, 2, all)
[1] F F F T T F
> jjperm3[,.Last.value]
     [,1] [,2]
[1,]    2    3
[2,]    3    1
[3,]    1    2
```

The `.Machine` object is a list of the machine constants, like largest integer and epsilon. It is often useful to pass pieces of this into C or Fortran code that you have written to eliminate machine dependency of your S functions. `pi` is another number that is a part of the S language.

4.4 Input and Output

Commonly a more frustrating part of computing is getting data from one program to another. S is no exception. Essentially all data that have not already been made into S objects must come into S via `scan`, which reads ASCII data. The use of `scan` ranges from very simple, to exceptionally complicated. A strange, but at times useful, way to use `scan` is to read from standard input rather than from a file—cutting numbers from some window and pasting to the S session is one example. In the remainder of the discussion of `scan`, the input will be assumed to be a file.

If a simple vector is to be read in, then you merely need to give the name of the file and possibly an indication of what mode to give the data (numeric is the default). Otherwise, the file is assumed to be in the form of records and fields. Typically there is one record per line in the file, but this is not mandatory. By default, fields are assumed to be separated by white-space, that is, by spaces, tabs or newlines. The `sep` argument to `scan` controls the separation character, and this default is denoted by `sep=""`. (Since a separator of no characters doesn't make sense, the empty string was available for a compound choice.) If the file contains strings containing spaces, then the default separator will not work. Tabs and semicolons are popular choices for separators. Only the first character of the `sep` argument is used.

Once past the problem of the separator, the key argument to `scan` is `what`. The `what` argument provides a template for the resulting object. Here are three examples:

```
jjnum <- scan("numbers.txt")
jjchar <- scan("text.txt", "", sep="\t")
jjrecords <- scan("records.txt", list(name="", age=0,
      NULL, volume=0), sep=";")
```

The first example is with a file that contains a bunch of numbers separated by white-space; the result, `jjnum`, is a numeric vector. The second example yields a character vector and the separator is a tab, presumably to allow spaces within strings. The final example will result in a list with four components. The first component will be named `"name"`, will be character, and will be read from the first field of each record. The second component is numeric and corresponds to the second field in the file. The third field is not read because of the `NULL` in the third component of the `what` argument, and this component will be `NULL` in `jjrecords`. The fourth component is once again numeric. If the file contained

more than four fields per record, or if not all records were on a single line, then the statement above would be wrong. The `flush` argument allows you to ignore unused fields, and `multi.line` allows records to span more than one line in the file.

If your file is in fixed format (meaning each field takes a certain number of characters), then there are two ways of getting the data read in. The easiest if you have S-PLUS is to use the `widths` argument to `scan`, which takes a vector of the field widths. The other way is to use `make.fields` which creates a new file that contains the fields interspersed with a separator.

If there are some parts of the file that you don't want to read in, then use the `skip` and `n` arguments. `skip` says how many lines at the top of the file not to read. The main purpose of this argument is when there is header information that does not look like the actual data. The `n` argument says how many items in all (records times fields) are to be read.

Some files don't conform to the records and fields format, and it seems impossible to get what you want. There are two strategies you can use, possibly even in conjunction. You can read everything into S—probably all as character—and straighten it out with S commands. You can use tools like Perl, `awk` and `sed` to convert the file to a more hospitable format.

If you are reading in a large object and you know how big it is, then you can save some memory by creating the `what` argument to be the same size as the final result. As `scan` reads the file, it keeps increasing the size of where it is putting the data. This wastes memory in two ways. It grows the objects incrementally so it fragments memory some, and the last time that it grows, it almost surely will overshoot the size that it needs.

DANGER. You will probably have problems if you try to read in numbers that were double-precision Fortran. Fortran uses a "d" rather than an "e" to indicate the exponent. `scan` does not understand the "d" and assumes that the value is not a valid number. You need to convert (carefully) the "d"s to "e"s before using `scan`. (Or possibly read them into S as character, and then do the conversion.)

If your file is very nicely arranged in records and fields, then you can use `read.table` to read the file and return a data frame. It does accept fixed format files.

DANGER. There is no free lunch. The convenience of `read.table` is diminished by the fact that it will decide on its own to use a field as the row names of the result. The field will then not appear in the data frame. I suggest that you always give a value for the `row.names` argument to avoid surprises.

For transferring S objects around, see the discussion starting on page 96 regarding `data.dump`, `dump`, `data.restore`, `source` and `restore`.

S-PLUS (and statlib) has a `sas.get` function to import data from SAS. There are also several commercial programs that allow easy transfer between a number of packages including S.

`print` and `cat` both create printed output, but in most other respects are quite different. `print` produces representations of S objects, while `cat` is a low-level function that interprets character strings. If you `print` a character string with a backslash-n in it, then the backslash-n will be visible within the string (and the string will generally be surrounded by quotes). But if you `cat` that same string, then the text following the backslash-n will be on the next line of output—`cat` interprets the backslash-n rather than just reproducing it. `print` is a generic function, and many `print` methods use `cat` to do part of the printing.

```
> print("\\blah\nblah")
[1] "\\blah\nblah"
> cat("\\blah\nblah")
\blah
blah>
```

Notice that the S prompt is directly after the string that was catted. This is because there was not a backslash-n at the end of the string.

When you are writing a `print` method, the last line of the function should be:

```
invisible(x)
```

This makes the return value `x` while avoiding an infinite loop of printing. The name of the argument should definitely be `x` (see the discussion of arguments to generic functions on page 128). The reason to have the return value be the input is so that

```
jj <- print(myfun())
```

will work the same as

```
jj <- myfun()
print(jj)
```

You also want to have the `...` form in the argument list of your `print` methods. Often your `print` method will call `print` and you can pass `...` to those calls. Even if you don't have `print` calls, you want `...` because your method may be called by another `print` call that uses arguments that your method does not have. An example would be if an object of your class is in a list that is printed.

While `print` always prints to standard-out, you can tell `cat` to send its characters to a file. `cat` takes an arbitrary number of arguments—each element of the first argument is printed, then each element of the second argument and so on. There are numerous additional arguments to `cat` to control its output. Note that `cat` does not obey the `digits` option, as `print` and `format` do. Thus numbers printed by `cat` have their full precision.

The `format` function is mainly used to turn numbers into suitably aligned character strings. It returns strings that all have the same number of characters. In S-PLUS version 3.2 and later, you can control whether or not numbers are written in scientific notation among other features. Notice the blank spaces so that the decimal points line up:

```
> format(c(222, pi))
[1] "222.000000" "  3.141593"
> as.matrix(format(c(222, pi)))
             [,1]
[1,] "222.000000"
[2,] "  3.141593"
```

See the `justify` function on page 278 for similar functionality.

Although `write` is likely to fool you more than help you, the S-PLUS function `write.table` is useful. `write.table` writes a matrix or a data frame to an ASCII file as you would expect it to be written.

The `sink` function allows you to send the output from S commands to a file. Unfortunately in version 3 of S there is no convenient way to send the output both to the screen and to the file. Here is an example of how to put both the S commands and their results into a file:

```
> options(echo=T)
> sink("jj.out")
sink("jj.out")
> freeny.x
> sink()
> 1:3
1:3
[1] 1 2 3
> options(echo=F)
options(echo=F)
```

The `echo` option, if TRUE, means that the command is repeated, and hence it ends up in the sink file. We can see what happens after the second `sink` command (with no arguments) which changes the output back to normal. The `echo` option doesn't take effect until after the `option` call.

DANGER. If you are using a version of S prior to S-PLUS version 3.4, you can not nest `sink` commands. In the older versions, each call to `sink` rescinds the previous one. In particular, `dump` uses `sink` so if you use `dump` while a sink is in effect, the `dump` command will work okay but your sink will be wiped out—output will go to standard-out as usual.

4.5 Synonyms

This section discusses functions that are similar to another. Understanding the differences can speed programming and eliminate bugs. *We can find no scar, But internal difference, Where the meanings, are* – [15]

When you have a vector of strings to match and they are possibly abbreviated, then `match` won't do. What does work is `pmatch`, and `charmatch` in S-PLUS. (These two functions were produced at about the same time in two different locations.) They are almost identical in effect, the main difference occurs when there is an ambiguous match.

The following commands have ambiguous matches for `"Ala"` and `"New"` since both `"Alaska"` and `"Alabama"` partially match `"Ala"`, and several state names begin with `"New"`.

```
> pmatch(c("Ala", "Mont", "New"), state.name)
[1] NA 26 NA
> charmatch(c("Ala", "Mont", "New"), state.name)
[1]  0 26  0
> pmatch(c("Ala", "Mont", "New"), state.name, nomatch=Inf)
[1] Inf  26 Inf
```

`pmatch` considers an ambiguous match to be `nomatch` while `charmatch` always returns 0 for an ambiguous match.

There are two additional differences between them. The first is that `pmatch` does not match empty strings:

```
> charmatch("", c(letters, ""))
[1] 27
> pmatch("", c(letters, ""), nomatch=0)
[1] 0
```

The second is that `pmatch` has a `duplicates.ok` argument.

```
> pmatch(c("a", "b", "a"), letters)
[1]  1  2 NA
```

```
> pmatch(c("a", "b", "a"), letters, dup=T)
[1] 1 2 1
> charmatch(c("a", "b", "a"), letters)
[1] 1 2 1
```

`charmatch` forces duplicates to be okay.

The `unabbrev.value` function (page 218) uses `pmatch`.

The `rm` function is a short cut for `remove`. `remove` takes a character vector of the names of the objects to be deleted. `rm` just takes the unquoted names of the objects. The two following commands would do the same thing:

```
remove(c("jja", "jj45"))
rm(jja, jj45)
```

`rm` removes objects only from the working database; `remove` has arguments `where` and `frame` so that you can select the location of the deletions.

The `seq` function is a generalization of the `:` operator, which it calls. The `:` operator is usually given integers, and always steps 1 or -1. `seq` allows arbitrary step sizes (the default is 1), starting point (the default is 1), ending point (the default is 1), and length.

A handy argument is `along`. The statement

```
seq(along=x)
```

returns a sequence starting at 1 that is as long as `x`. The important part is that if `x` has length zero, then the sequence has zero length also—this is very useful in `for` loops.

The `if` construct, the `ifelse` function, and the operators `&&`, `&`, `||` and `|` are all related. `if` takes a logical value that is meant to be just one long; if the length is greater than one, then the first value is used and a warning is issued:

```
> if(c(F, T)) "true" else "false"
[1] "false"
Warning messages:
  Condition has 2 elements: only the first used in:
        if(c(F, T)) "true" else "false"
```

Do not ignore such warnings—it means you have at least done something silly and more likely that there is a bug. You may have forgotten to use `any` or `all`.

The `&&` (and control operator) and `||` (or control operator) operators go with `if` and length one logicals. These do not evaluate their second argument

if they already know the answer; for instance `||` does not need to look at the second argument if the first one is `TRUE`.

On the other hand, `&` and `|` are vectorized *"and"* and *"or"* operators, so it is not appropriate that they should reserve judgment on evaluating their second argument. The `ifelse` function performs a vectorized decision, so it needs to use `&` and `|`. By the way, there is the `xor` function which performs a vectorized exclusive-or operation.

`ifelse` is very S in being vectorized, but it is less useful than you might think. For instance, the statement:

```
x <- ifelse(is.na(x) | x < 0, 0, x)
```

which changes missing values and negatives in `x` to `0`, can be done more directly with

```
x[is.na(x) | x < 0] <- 0
```

Let's summarize. Suppose we have two (length one) logical values called `logical.val1` and `logical.val2`, and we have two logical vectors with length greater than one called `logical.vec1` and `logical.vec2`. Then here are several statements and their acceptability:

```
if(logical.val1 && logical.val2) 0 else 1 # correct
ifelse(logical.vec1 & logical.vec2, 0, 1) # correct
if(logical.val1 & logical.val2) 0 else 1 # slightly inefficient
if(logical.vec1 && logical.vec2) 0 else 1 # wrong
if(logical.vec1 & logical.vec2) 0 else 1 # wrong
ifelse(logical.vec1 && logical.vec2, 0, 1) # wrong
ifelse(logical.val1 & logical.val2, 0, 1) # slightly inefficient
```

There are two ways of putting ASCII representations of S objects into files. (If you want just the values without retaining the characteristics of the S object, then use `write.table` (page 93) or `cat`.) `dump` creates representations that are (relatively) easy for humans to comprehend—the format is S commands. `data.dump` also produces ASCII files, but the format is decidedly tailored towards machines rather than humans. The reason that there are two choices is that files created by `dump` can take an enormous amount of memory (and hence time) when they are read back into S. The larger the size of the objects, the more problem there is with restoring a dump file. Since functions are generally small and commonly read by humans, `dump` is usually used for them. Most often `data.dump` should be used for other types of objects.

If a file was produced by `data.dump`, then it must be read back into S with `data.restore`. There are several ways to read files created by `dump`. The most common is to use `source`. `restore` is almost like `source`, but there is a

difference that can be important. Essentially, `source` acts as if the contents of the file were typed to S, but `restore` starts up a new session of S and reads the file. So `restore` always puts the objects into the working database for the current directory, while `source` puts the objects into the current working database. Because `restore` starts up a new session, there are going to be implications for memory usage. Yet another way to read a dump file into S is to redirect the file while you are in Unix. For example:

```
% S < myfile.q  # Unix command
```

This also puts the objects into the working database for the current directory.

`storage.mode` and `mode` always give the same answer, unless the mode of the object is numeric. In that case `storage.mode` is one of `"double"`, `"single"` or `"integer"`. Storage mode is generally only of interest when using `.C` or `.Fortran`.

The `class` function returns the value of the `"class"` attribute of an object. `data.class` does the same if the object has a `"class"` attribute, but can return something other than `NULL` for objects that do not have a class. This looks to see if it is one of the types of object that was in common use before object-orientation came along. The possibilities are `"matrix"`; `"array"` for arrays that are not matrices; `"ts"` for time series; plus categories yield the class they would have if they were factors.

In S-PLUS there are functions called `which.na`, `which.inf` and `which.nan`. These are like `is.na`, etcetera, except that they return a vector of numeric indices rather than a logical vector. The result of `which.na` is the indices of the result of `is.na` that are `TRUE`. If you expect an object to be large and to have few missing values, then using `which.na` instead of `is.na` can save a substantial amount of memory. If `x` has length one million and contains ten missing values, then the length of the result of `is.na` is one million while the result of `which.na` is only ten long.

The `max` and `min` functions each return a single number that is the maximum or minimum of all of the numbers in all of the arguments. There are times when you want a vectorized form that gives the maximum or minimum for each location among a number of vectors—the `pmax` and `pmin` functions do this. Compare:

```
> pmin(1:10, 10:1)
 [1] 1 2 3 4 5 5 4 3 2 1
> min(1:10, 10:1)
[1] 1
```

DANGER. It is not so uncommon to have a bug because `max` is used where `pmax` is needed. These are especially hard to discover and track down. A little extra thought whenever you use `max` or `min` can be a good thing.

The `deparse` and `as.character` functions both turn non-character objects into something of mode character. `deparse`, as its name implies, returns a character string that can be parsed by S, the length of the result of `deparse` is usually 1 (but not guaranteed to be). `as.character` thinks in terms of vectors and always returns an object that has the same length as the input. Using `as.character` where `deparse` is needed can cause some subtle bugs.

4.6 Linguistics

Use `stop` to create an error condition in a function. If the problem is not serious enough to warrant an error but is still worth mentioning, then use `warning`. A few pointers are given on placing calls to `stop` and `warning` on page 33.

Do not confuse `warning` with `warnings`. `warning` is used within functions to create a warning. `warnings` is used to look at the warnings that were created if there were a lot of them.

The `exists` function takes a character string containing the name of an object and returns a logical value that states whether or not the object was found. Arguments can be used to direct the search. This is useful to avoid an error when the object does not exist. An example is the `poet.dyn.load` function on page 24.

The `get` function goes further than `exists` and returns the object if it is found (an error results if it isn't found, hence one use for `exists`). Most of the time `get` is unnecessary because just giving the name of the object gets it. But `get` has several uses. It is used when what you have is a character string containing the object name, rather than the name itself. The following fragment should give an indication of when this might occur:

```
for(i in 1:n) get(paste("data", i, sep=""))
```

It is used when the name does not parse to a valid object name:

```
get("dim<-")
get("%*%")
get("valid_name_in_other_language")
"[<-.mathgraph" <- emacs(get("[<-.mathgraph"))
```

It is used when there is more than one instance of an object name on the search list, and you want one of the ones that is not first:

```
get("x", where=4)
```

It is used when the object is in a database that is not currently on the search list, or that at least is not guaranteed to be on it.

DANGER. The first argument needs to be a string that is the name of the object as S knows it. Do not include any of the path for the database there:

```
get("/usr/me/good.data/x") # probably not what you want
get("x", where="/usr/me/good.data") # more likely right
```

You can sometimes save some memory with a statement like:

```
get("x", immediate=T)
```

This says that you want `x` but you only care about it for this one instance so it should not be cached for future reference. The cache that we are avoiding here starts when a statement is evaluated and is destroyed again when we get back to the S prompt—it is not the same cache that the `keep` option controls.

The `assign` function is in a sense the opposite of `get`, and is used for many of the same reasons. You can use it whenever an assignment is made (that is, it is a synonym for `<-`), but there are times when you must use it. Use it when what you have is a character string of the name rather than the name itself.

```
for(i in 1:n) assign(paste("data", i, sep="."),
    scan(paste("file", i, sep="")))
```

You can use quotes around a name that is being assigned to:

```
"%myop%" <- fjj
```

So `assign` is not necessary here, even if the name is not a valid S name.

DANGER. You can not have a function on the left of an assignment operator that returns a character string of the names:

```
paste("data", i, sep="") <- 1:i  # THIS WON'T WORK
```

The `assign` function is necessary in this case.

```
assign(paste("data", i, sep=""), 1:i)
```

A common use of `assign` is to assign an object to a location other than the usual location. From the prompt, this is generally a database later in the search list:

```
assign("jj", jj, where=4)
```

Within a function, this is usually assignments to frame 1 or database 0:

```
assign("fjJ.X", x, frame=1)
```

Within a function you can also force the assignment to be written to disk immediately rather than only being written as you get the S prompt back. An example is:

```
assign("jj", jj, where=1, immediate=T)
```

This form of statement can be very useful, but it is a definite violation of the no side effects rule—use this sparingly and with good reason.

DANGER. A statement like

```
assign("x$new", 3)
```

does not do anything to a component of an object named `x`. Rather it creates an object with the unfriendly name `x$new`:

```
> x
$a:
[1] 1

> assign("x$new", 3, where=1)
> x
$a:
[1] 1

> get("x$new")
[1] 3
```

You need to assign the entire object in one go:

```
> assign("x", c(x, new=3), where=1)
> x
$a:
[1] 1

$new:
[1] 3
```

Often you want to create a local version of the object and then use `assign`.

An unusual (but very useful) function is `on.exit`. This allows you to perform actions as the function exits—often some cleanup—without getting in the way of the return value. *Oh build your ship of death, your little ark* [16]

Here is a typical example:

```
on.exit(par(old.par))
old.par <- par(cex=2)
```

You, of course, think that I must have these commands in reverse order. But we do want the `on.exit` command first, and it's not going to break—I'll get to that. The purpose of these two lines is to set a graphics parameter to a certain value, and then make sure that the parameter is returned to its previous value when the function quits. In this case, we could almost get by without `on.exit` by resetting the graphics parameter after we do all of the graphics, and then return whatever it is that we want (in other cases there is no way to do the necessary cleanup and get the proper return value). However, without `on.exit` the graphics parameter will be permanently changed if an error (or interrupt) occurs between the time that the parameter is changed and it is changed back. `on.exit` makes sure that the action is performed whether the function exits normally or exits due to an error.

Let's analyze what happens with our `on.exit` example. If the function exits normally, then the parameter is changed, changed back, and the function returns its value (and performs its side effects). If an error occurs after the call to `par`, then the `on.exit` expression is used to change it back. If an error occurs during the call to `par`, then `old.par` doesn't exist so the `on.exit` expression will fail (you'll get an "Error during wrapup" message), but no matter since the graphics parameter wasn't changed anyway. If we had the `on.exit` call second, then it would be possible (though highly unlikely) that the parameter get permanently changed.

The `interlude` function listed on page 220 creates functions that use `on.exit`. Fancier use of `on.exit` includes adding to the actions to be performed, and deleting all of the `on.exit` statements.

A common function to be used in `on.exit` is `unlink`. This removes a file—typically a file that was created within the function calling `unlink`. A characteristic use is:

```
the.filename <- tempfile("Spfun")
on.exit(unlink(the.filename))
# the file is created by some command
```

`inherits` in its most common use returns a number (that can be interpreted as a logical) indicating if a certain string is one of the elements of the object's class. It might be used like:

```
if(inherits(x, "factor")) ...
```

More involved use is also possible.

Use the idiom `deparse(substitute(x))` to create a character string of the name that was used for `x` in the call to a function. Labeling plots is a common example. The `sccs` function on page 49 uses this. A trivial example is:

```
> fjjcopy
function(x)
list(name = deparse(substitute(x)), sum = sum(x))
> fjjcopy(freeny.x)
$name:
[1] "freeny.x"

$sum:
[1] 1282.411

> fjjcopy(1:8)
$name:
[1] "1:8"

$sum:
[1] 36
```

We will learn more about `substitute` later (page 324), but at this point you can just think of this as the magic potion that performs this operation.

Many objects have a component or an attribute named `call` which is an image of the command that created the object. This is very useful for telling you what the object represents. The `call` component or attribute is generally created with the `match.call` function.

```
> fjjcopy
function(x)
list(name = deparse(substitute(x)), sum = sum(x), call
         = match.call())
> fjjcopy(freeny.x)
$name:
[1] "freeny.x"

$sum:
```

```
[1] 1282.411

$call:
fjjcopy(x = freeny.x)
```

Let me lobby you to use this convention in your functions. Note that the result of `match.call` is not a character string, but rather a language object. *And I plucked a hollow reed* [17]

Sometimes you have a list of the arguments to give to a function, or that is the easiest thing to get. In such a case, use `do.call`:

```
> jj <- as.list(sample(10, 3))
> do.call("paste", c(jj, sep=", "))
[1] "1, 10, 7"
```

The `genopt` function on page 331 shows `do.call` in a more realistic setting, as does `c.mathgraph` on page 308.

If you know how to perform an operation but there doesn't seem to be a way to directly do the command with what you have, then there is always the eval-parse-text idiom. A dumb example is

```
ans <- eval(parse(text=paste("fjjmain.", method,"(x)",sep="")))
```

The `fjjcheckratop` function on page 254 and `bind.array` on page 289 provide further examples.

The `synchronize` function is used to ensure that hash tables associated with a database are up-to-date. This is only a concern if more than one S session is using the same database and at least one of them is writing to it. There are two hash tables involved.

The first is the hash table of names of objects within the database. As assignments are made in an S session, this hash table is revised, but the session doesn't know about changes that are made behind its back. Below we have two S sessions running and both using the same working database. Session 1 creates a new object:

```
S1> jjnew <- 1:10
S1> jjnew
 [1]  1  2  3  4  5  6  7  8  9 10
```

Session 2 doesn't see the new object until after the call to `synchronize`:

```
S2> jjnew
Error: Object "jjnew" not found
```

```
S2> synchronize(1)
NULL
S2> jjnew
 [1]  1  2  3  4  5  6  7  8  9 10
```

The other hash table that can get out of date is the hash table of objects (rather than object names) that is controlled by the `keep` option. Generally only functions are put into this hash table (but see page 47). In this example both sessions start with `jj` being 1 through 10.

```
S1> jj
 [1]  1  2  3  4  5  6  7  8  9 10
S1> jj[5] <- 55555
S1> jj
 [1]     1     2     3     4 55555     6     7     8
 [9]     9    10

S2> jj
 [1]     1     2     3     4 55555     6     7     8
 [9]     9    10
```

The second session sees the change in `jj`, there is no need for `synchronize`. But it is different if both sessions have used function `fjj` and the first session changes the definition:

```
S1> fjj
function(x)
1/x
S1> fjj <- function(x) 2/x^2
S1> fjj
function(x)
2/x^2
```

Because session 2 has `fjj` in its object hash table, it doesn't see the revised version that is in the database until it calls `synchronize`:

```
S2> fjj
function(x)
1/x
S2> synchronize(1)
NULL
S2> fjj
function(x)
2/x^2
```

`synchronize` can also be used without an argument. Suppose that you have a `for` loop that takes a long time to perform each iteration like:

```
for(i in seq(along = lcf.ans) {
     lcf.ans[[i]] <- long.complicated.function(x[i], y[[i]])
}
```

As this stands, `lcf.ans` (which we are assuming resides in the working database) will not be changed (written to disk) until the entire loop is finished. You may be concerned that the machine will crash before the end of the loop, or you may want to have the results as they get computed. You can force `lcf.ans` to be updated at each iteration with:

```
for(i in seq(along = lcf.ans) {
     lcf.ans[[i]] <- long.complicated.function(x[i], y[[i]])
     synchronize()
}
```

In this case, it would have been better to use `assign` with `immediate=T`, but if lots of things change in the loop, `synchronize` becomes an easier solution. For more on this topic, see the chapter on large computations starting on page 355.

The `readline` function captures a line of text. This is generally used in functions that interact with the user. Here is an example:

```
"fjjartshow"<-
function()
{
        cat("type 'y' to get picture 'n' to quit\n")
        repeat {
                cat("Do picture? ")
                ans <- substring(readline(), 1, 1)
                switch(ans,
                        y = ,
                        Y = {
                                plot(1, type = "n",
                                  xlim = c(0, 10),
                                  ylim = c(0, 10),
                                  axes = F, ylab = "",
                                  xlab = "")
                                polygon(sample(10, 50,
                                  replace = T), sample(
                                  10, 50, replace = T))
                        }
                        ,
                        n = ,
                        N = ,
                        q = ,
                        Q = break,
```

```
                        cat("Say what?"))
        }
}
```

CODE NOTE. In the instructions for using the function there appear to be only two valid responses, yet the function actually allows more. That is, there are some undocumented features. Undocumented features are fine as long as you are not going to want to change them. There should be a balance between the users' current convenience and the programmer's future options.

If the text that is to be read is S language, then it is better to use `parse` to capture the text than to use `readline`. But in the case of `fjjartshow` the input is neither S language nor arbitrary text, but merely a simple choice. The `menu` function is the proper tool for this case. Here is a revised artshow:

```
"fjjartshow2"<-
function(corners = 50, sieve = 10)
{
        repeat {
                switch(menu(c("(another) plot",
                                "help on polygon"),
                                title =
                                "type 0 to exit") + 1,
                        break,
                        {
                                plot(1, type = "n",
                                  xlim = c(0, sieve),
                                  ylim = c(0, sieve),
                                  axes = F, ylab = "",
                                  xlab = "")
                                polygon(sample(sieve,
                                  corners, replace = T
                                  ), sample(sieve,
                                  corners, replace = T))
                        }
                        ,
                        help("polygon"))
        }
        invisible()
}
```

`menu` takes a character vector of choices and optionally a title to be printed above each menu. The user will be shown a numbered menu of the choices and prompted to select one. It is conventional to have 0 mean "exit" which is what this function does. Here is a session where just one plot is created, and the help file is not chosen at all:

```
> fjjartshow2()
```

```
type 0 to exit
1: (another) plot
2: help on polygon
Selection: 1
type 0 to exit
1: (another) plot
2: help on polygon
Selection: 0
>
```

Periodically you want the return value of a function not to be printed automatically. The `invisible` function does this. Graphics functions, for example, often return something invisibly. Note, though, that invisibility is a trickier concept than you might think—see how the `soptions` function on page 45 does it, and see page 219 for an explanation of why that works.

4.7 Numerics

There are several functions for matrix decompositions. `qr` performs a QR decomposition; this is the decomposition on which the least squares regressions are based. Singular value decompositions are done with `svd`, and `eigen` does eigen decompositions. Choleski decompositions are computed with either `chol` or `choleski`. `chol` merely returns the triangular matrix, while `choleski` returns a list that includes the condition number and the pivoting (reordering for numerical accuracy).

Many (but not all) of the functions described in the rest of this section are S-PLUS functions.

You can numerically integrate an S function with `integrate`. Infinite limits are allowed. (The `line.integral` function on page 320 is an alternative with advantages and disadvantages.)

DANGER. The function that you give to `integrate` must be vectorized. If it isn't, you can get wrong answers.

```
> integrate(function(x) min(x, 2), 1, 4)$integral
[1] 0.5736171
Warning messages:
  subdivision limit reached in: integrate.f(f, lower,
        upper, subdivisions, rel.tol,  ....
> integrate(function(x) pmin(x, 2), 1, 4)$integral
[1] 5.499997
```

Don't count on there being a warning message when you get it wrong.

DANGER. The `integrate` function doesn't follow the capitalization convention on its arguments that is discussed on page 27 so it is possible that the extra arguments to your function can collide with arguments of `integrate`. Here is an example where we are integrating with respect to `x`, and `s` is an additional argument to the function.

```
> integrate(function(x, s) x*s, 1, 3, s=20)
Error in qf15(f, lower, upper, aux = ..: singularity
        encountered
Dumped
> traceback()
Message: singularity encountered
5: stop("singularity encountered")
4: qf15(f, lower, upper, aux = optargs(list(...), f, 1))
3: integrate.f(f, lower, upper, subdivisions, rel.tol,
2: integrate(function(x, s)
1:
```

The call fails, and the traceback doesn't enlighten us at all. What is happening is that the `s` that we pass in is being used as the value of the `subdivisions` argument to `integrate`.

Symbolic derivatives of some simple functions are given by `deriv` or `D`. `D` has the more human face:

```
> D(expression(sin(x)/x), "x")
cos(x)/x - sin(x)/x^2
> D(expression(sin(x) * y/x), "x")
(cos(x) * y)/x - (sin(x) * y)/x^2
```

But you can get an actual function from `deriv`:

```
> deriv(expression(sin(x) * y/x), "x", function(x, y) NULL)
function(x, y)
{
        .expr2 <- (sin(x)) * y
        .value <- .expr2/x
        .grad <- array(0, c(length(.value), 1), list(
                NULL, "x"))
        .grad[, "x"] <- (((cos(x)) * y)/x) - (.expr2/(
```

```
                x^2))
        attr(.value, "gradient") <- .grad
        .value
}
```

This returns a vector of the expression which has a gradient attribute containing the derivative.

The `approx` function performs linear interpolation given two vectors.

More commonly useful is the `interp` function that performs interpolation to a regular two-dimensional grid given three vectors. In particular, this is often used to create output that is suitable as input to the graphics functions `persp`, `contour` and `image`.

Use `polyroot` to find the roots of a polynomial. The coefficients may be complex as well as numeric, and the degree can be as high as 48.

To find some root of a non-polynomial univariate function, use `uniroot`.

`optimize` finds a local minimum of a univariate function.

Both `nlminb` and `nlmin` minimize general nonlinear functions. They each have their little quirks, but the only essential difference between them is that `nlminb` allows box constraints to be put on the parameters while `nlmin` does not. They are both based on PORT routines. The `portoptgen` function (see starting at page 344) provides an alternative.

The `ms` function provides optimization from a different perspective. It takes a formula and a data frame. The formula is applied to each row of the data frame, and the result is summed across rows. It seeks a minimum of that sum.

The `fft` function performs a fast Fourier transform. In addition to the statistical operation of spectral analysis, this can be used for convolutions. `fft` accepts arrays as well as vectors, so it performs higher dimensional transforms.

Other numerical functionality is performed by `vecnorm`, `kronecker`, `peaks`, `chull` (for two-dimensional convex hulls), `spline`, `ivp.ab` (for differential equations), `napsack` and `stepfun`.

The `primes` function may be found in the `progexam` library to S-PLUS. This finds prime numbers. The S-PLUS Programmer's Manual shows a `factors` function, but it doesn't appear in the library.

4.8 Randomness

Pseudo-random functions within S use a single random number generator. The seed for the generator is stored in `.Random.seed`, which is a vector of 12 small integers. As far as I know, this is robust, but it is not a wise idea to assign anything to `.Random.seed` that was not previously a random seed.

If you want to reproduce results of random functions, then you need to save the seed and restore it just before the call that is to reproduce the results:

```
> jjsave.seed1 <- .Random.seed
> runif(4)
[1] 0.5701077 0.5981965 0.2946985 0.4174643
> runif(4)
[1] 0.93036326 0.93932162 0.08411177 0.94985548
> .Random.seed <- jjsave.seed1
> runif(4)
[1] 0.5701077 0.5981965 0.2946985 0.4174643
```

DANGER. Do not assign `.Random.seed` within a function, or anywhere else that is not a database on the search list.

```
> fjjran1
function(n = 3)
{
# DO NOT DO THIS
        .Random.seed <- .Random.seed
        for(i in 1:n) {
                print(runif(3))
        }
        invisible()
}
> fjjran1()
[1] 0.5157684 0.5730458 0.7764676
[1] 0.5157684 0.5730458 0.7764676
[1] 0.5157684 0.5730458 0.7764676
> fjjran1()
[1] 0.9350564 0.7920594 0.6019266
[1] 0.9350564 0.7920594 0.6019266
[1] 0.9350564 0.7920594 0.6019266
```

Here's the explanation for what is happening. The random functions look for the first value of `.Random.seed` that they see, which in this case is the value set in `fjjran1`. When a random function is done, it updates `.Random.seed` on the working database. So while we stay inside `fjjran1` we only see one version of `.Random.seed` even though new versions are being written to the

working database. When the second call to `fjjran1` starts, it sees the value of `.Random.seed` that was updated on the last loop of the previous call to `fjjran1`.

Another way of ensuring reproducibility is to use `set.seed`. This takes an integer that should be between 1 and 1024, and sets the random seed to a certain value based on the integer given.

```
> set.seed(1023); print(.Random.seed)
>  [1] 21 14 49 16 62  1 32 22 36 23 28  3
> set.seed(-1); print(.Random.seed)
>  [1] 21 14 49 16 62  1 32 22 36 23 28  3
```

The example above shows that `set.seed` returns the same answer for arguments that are the same modulo 1024.

When a function that uses randomness is stopped on an error or interrupted, the random seed will not be updated. The technical reason for this is that assignments are not committed until the execution of the function is finished. This is the proper behavior—you wouldn't want the seed to be changed when an error occurs. If the seed did change, then you couldn't be sure that your fix to the function eliminated the bug—it could be that the new set of random numbers doesn't exhibit the bug.

S contains functions for a number of probability distributions. Usually there are four functions for each distribution. The function names begin with one of the four letters 'r', 'd', 'p', 'q', and end with a code for the distribution. Examples are `runif`, `dunif`, `punif`, `qunif`, and `rnorm`, `dnorm`, `pnorm`, `qnorm`. The 'r' stands for random, 'd' is for density, 'p' is for probability, and 'q' is for quantile. The random functions generate numbers from the distribution, the probability functions give the probability of being less than or equal to the value given (probability distribution function), and the quantile functions return the value where the distribution function achieves the probability given. These functions are vectorized.

DANGER. You need to be careful when you get to the boundary of parameter spaces.

```
> jjsd <- c(1.4, 0, 3.4)
> rnorm(3, mean=1, sd=jjsd) # bad news
[1] 0.7728073        NA 0.5524101
Warning messages:
  One or more nonpositive parameters in: rnorm(3, mean
        = 1, sd = jjsd)
```

```
> rnorm(3, mean=1) * jjsd # wrong
[1] 2.000239 0.000000 6.574893
> rnorm(3) * jjsd + 1 # right
[1] -0.02761199  1.00000000  3.91233952
```

DANGER. Discrete distributions can be tricky, rounding is often a good idea.

```
> jj2 <- c(2, 2 - .Machine$double.eps)
> jj2
[1] 2 2
> ppois(jj2, 1.5)
[1] 0.8088468 0.5578254
> ppois(round(jj2), 1.5)
[1] 0.8088468 0.8088468
```

The `sample` function is a random function of a different sort. In its simplest use, it returns a permutation of its argument. (A single integer stands for the integers up to it from 1.)

```
> names(jjwt)
[1] "dorothy"  "harold"   "munchkin" "stevie"
> sample(names(jjwt))
[1] "harold"   "munchkin" "dorothy"  "stevie"
> sample(10)
 [1]  7  2  5 10  3  9  1  6  8  4
```

An important argument is `replace` which tells whether or not items are to be replaced after they have been selected. (I remember that the default value of `replace` is FALSE since a permutation results when `replace` is missing.)

```
> sample(names(jjwt), 10, rep=T)
 [1] "harold"   "stevie"   "harold"   "harold"
 [5] "dorothy"  "stevie"   "dorothy"  "munchkin"
 [9] "munchkin" "munchkin"
```

There is also a `probability` argument that takes the relative probabilities for each item in the population. This need not sum to 1, `sample` will normalize it.

```
> sample(names(jjwt), 10, rep=T, prob=1:4)
```

```
 [1] "stevie"   "stevie"   "stevie"   "dorothy"
 [5] "munchkin" "stevie"   "munchkin" "munchkin"
 [9] "munchkin" "stevie"
```

4.9 Graphics

Here is another part of my cursory look at graphics in S.

The trellis package is a relatively recent addition to the graphical capabilities in S. Primarily trellis is concerned with ways to plot data of several variables in a comprehensible manner.

`plot` is the most conspicuous graphics function. It is generic and there are numerous methods for it. The default method does what I expect you would think, plus a little more.

Plots of three variables are done with `persp`, `contour` and the S-PLUS function `image`. The trellis function `wireframe` is similar to `persp`.

General purpose graphics functions include `barplot`, `dotchart`, `matplot`, `symbols` and `pie`. Pie charts are out of favor with statistical graphics experts—they essentially require us to estimate angles, which humans are not very good at doing. Use dot charts or bar plots instead.

Some functions of a more statistical nature are `boxplot`, `hist`, `pairs`, `qqnorm`, `qqplot`, `tsplot`, and the S-PLUS function `biplot`.

Among the most useful low-level graphics functions are `title`, `text`, `mtext`, `legend`, `points`, `lines`, `abline`, `arrows`, `segments`, `polygon`, `axis`, `rug`, `box` and `frame`.

Two functions that are available on many graphics devices for interacting with plots are `identify` and `locator`. `identify` writes something on the plot about the nearest datapoint to the place that you specify. `locator` will give the coordinates to locations that you select.

4.10 Statistics

Functions to do simple summary statistics are `mean`, `median`, `var`, `cor`, `quantile`. The S-PLUS functions `location.m`, `cov.mve`, `scale.tau` and `scale.a` perform robust estimation. S-PLUS also contains `cov.wt` for covariance estimation with observation weights.

Functions in S-PLUS that perform hypothesis tests are `t.test`, `wilcox.test`, `cor.test`, `var.test`, `prop.test`, `chisq.test`, `fisher.test`, `kruskal.test`, `mantelhaen.test`, `mcnemar.test`, and `friedman.test`. Goodness of fit tests are performed with `chisq.gof` and `ks.gof`. `crosstabs` is also an S-PLUS function.

There are two ways to perform ordinary least squares regression—either via `lm` which takes a formula, or with `lsfit` which is an older function. If you want to use a function that is quick at performing regressions on data that is known to be good (no missing values and so on), then `lm.fit.qr` would be a good choice. A situation that I have in mind is if the function you are writing does iterative regressions. If you want to fit a least squares model with the coefficients constrained to be positive, then use the `nnls.fit` function of S-PLUS.

Robust regression may be done via `l1fit` which performs least absolute deviation regression, or with `rreg` which does M-estimates of regression. High-breakdown regression is done with the `ltsreg` (least trimmed squares regression) S-PLUS function.

Analysis of variance (ANOVA) is accomplished with `aov`. (There is an `anova` function—which does mean analysis of variance—but this is not the function you want to use to fit an ANOVA model.) Variance components can be estimated with the S-PLUS function `varcomp`. The `lme` and `nlme` functions (linear mixed effects and nonlinear mixed effects) provide related functionality in recent versions of S-PLUS. S-PLUS also has functions for making quality control charts.

Yet another area where S-PLUS has added a lot of functionality is in time series. Univariate ARIMA models are fit with `arima.mle` and its friends. The `arima.fracdiff` function fits (long-memory) fractionally differenced ARIMA models. `ar` fits univariate and multivariate AR models, and `spectrum` performs spectral analysis on univariate and multivariate series. In addition, multivariate autocorrelation estimates (including cross-correlations) are available via `acf` and univariate robust seasonal decompositions are created with `stl` (vanilla S function).

Functions for multivariate analysis include `cmdscale` for classical multidimensional scaling, and `cancor` for canonical correlation. Multivariate analysis within S-PLUS also includes principal components with `princomp`, factor analysis with `factanal` and multivariate analysis of variance with `manova`.

Survival analysis functions are available in S-PLUS and from statlib.

The `glm` function fits generalized linear models. Notice that one of the functions to fit regression was `lm` as in "linear model". One description of regression is that the expected value of the response is some linear combination of the explanatory variables; furthermore, the errors from this are distributed as a Gaussian, or at least continuously distributed and symmetric about zero. Generalized linear models are generalized in two ways. First, that the expected response is some specified function of a linear combination of the explanatory variables; that function is called the *link* function. Second that the errors can come from a variety of distributions, such as binomial, Poisson or gamma. Logistic regression fits within the scheme of generalized linear models.

Smoothing is an important element of modern statistics. Explicit functions for smoothing include `smooth`, `loess.smooth`, `smooth.spline` and the S-PLUS functions `ksmooth` and `supsmu`.

Much of the rest of the statistical functionality that I'll talk about are some combination of the idea of generalized linear models and smoothing.

Directly along this line are generalized additive models which are fit with `gam`. A generalized additive model is just like a generalized linear model except that we find a linear combination of smooth functions of each explanatory variable, rather than a linear combination of the variables themselves. Each smooth function is a function of one variable only.

The `ace` and `avas` S-PLUS functions fit models that are similar to generalized additive models, except that they do not have a link function, but they allow the response to be transformed via a smooth as well as the explanatory variables.

The S-PLUS function `ppreg` fits projection pursuit regression models. These take a linear combination of the explanatory variables first and then does a smooth. Several of these terms may be added together to approximate the response.

`loess` does local (robust) regressions so that the response is approximated locally in each region of the space of the explanatory variables.

The `nls` function performs nonlinear least squares. This is different from ordinary regression only because the parameters do not enter linearly in the model. For example a linear equation would be

$$y = ax_1 + bx_2 + c \tag{4.1}$$

where the parameters are a, b and c. Here is a nonlinear equation

$$z = e^{ax_1} e^{bx_2} e^{c} \tag{4.2}$$

that can be transformed to a linear form by taking logarithms. The following formula is inherently nonlinear:

$$y = ax^b + c \tag{4.3}$$

Nonlinear regression with box constraints on the parameters may be done in S-PLUS with `nlregb`.

Finally, we get far afield with `tree` which fits classification and regression trees. A tree is a regression tree if the response takes on continuous values, and it is a classification tree if the response is categorical. Trees are decidedly non-smooth. The basic operation in growing a tree is splitting a node. The object is to have the response within each of the two groups as alike as possible. At first there is only one node—the whole dataset. You split that in two, which means to find the best combination of the explanatory variable and the location in the explanatory variable to break the data into two. Then split

each of the two nodes you just created. Continue splitting nodes until you don't want to anymore. (The most common current practice is to create more than enough nodes, and then to decide which splits to throw away—inefficient computationally, but more efficient statistically.)

Trees have become popular because in some respects they are very easy. When you have your tree and you want to know how to classify a new case, you only need to consider one variable at a time: Number of feet greater than 3? Yes. Weight less than 10 kg? No. Then must be a corgi.

4.11 Under-Documented

This section presents functions that don't have help files, or didn't have help for some portion of their life, or for other reasons are less visible than I think they should be.

`zapsmall` is a little-known and deviously clever little function. It changes numbers that are very close to zero relative to the largest number (in absolute value) in the vector. It was created to be used for statistics that are printed out, but it can also be used for informal testing as in:

```
zapsmall(real.answer - fjj(input))
```

Here we think that the result of `fjj(input)` should be the same as `real.answer` with perhaps some noise thrown in. If you see numbers that are on the order of rounding error, then you know that there are no numbers much bigger than that. On the other hand, you might see a bunch of zeros and a few non-zero numbers.

S-PLUS contains a group of functions that do set operations on atomic vectors. Their names are `union`, `intersect`, and `setdiff`.

Another secretive S-PLUS function is `find.objects` which is like `find` except that it finds object names that contain a certain string rather than matching the string exactly. The other difference is that `find.objects` only takes a string—you may not pass the unquoted name to it. Using `find.objects` is like using `objects` with the `pattern` argument except that `find.objects` looks in all of the databases on the search list, not just one.

The `slice.index` S-PLUS function is a relatively recent addition. It is a generalization of the `row` and `col` functions to higher dimensional arrays.

The `merge.levels` function allows you to combine levels in a factor. For example, if you had a factor representing species, you could use `merge.levels` to make a new factor representing genera or families.

If you are poor and use S-PLUS via modem without a windowing system, then you will not have the help window system available to you. The `topic` function searches for phrases within the titles of help files, so that you may be able to find some functionality even if you don't know the function name.

```
> topic("mark")
invisible       Mark Function as Non-Printing
Question.mark   On-line Information on Functions, Objects, ...
util    Earnings and Market/Book Ratio for Utilities
util.earn       Earnings and Market/Book Ratio for Utilities
util.mktbook    Earnings and Market/Book Ratio for Utilities
```

An S-PLUS function that can come in handy for messing with character data is `AsciiToInt`. This doesn't seem to be intended for public use, but it probably won't go away soon. For each character in a character vector, `AsciiToInt` returns the corresponding ASCII code. The result that `AsciiToInt` returns is one numeric vector.

```
> match(AsciiToInt("/tmp/myomy/Sfun.45"),
+       AsciiToInt("/"), nomatch=0)
 [1] 1 0 0 0 1 0 0 0 0 0 1 0 0 0 0 0 0 0
```

4.12 Deprecated

The `category` function and its relatives do the same thing that `factor`, `ordered` and their relatives do. `category` was written before object-orientation came along. It holds no benefit and some deficits relative to the object-oriented versions, though factors are no angels as can be seen starting from page 130. *And your exasperating pit-pat* [18]

`sort.list` does the same thing as, but a little less than `order`, and the name can be a source of confusion. Use `order` instead.

4.13 Things to Do

Create a vectorized function that divides each string in a vector into its individual characters.

The S-PLUS functions `union` and `intersect` use atomic vectors to represent sets. Set theory implies that lists are the proper objects for sets. Create a suite of functions that perform set operations on lists. Include a symmetric difference. Can you make them reasonably efficient? Is object-orientation useful? Is a `complement` function feasible?

Make `zapsmall` work for complex numbers. What do you want it to do?

Figure out why the following happens:

```
> jjm
     [,1] [,2] [,3]
[1,]    1    6   11
[2,]    2    7   12
[3,]    3    8   13
[4,]    4    9   14
[5,]    5   10   15
> apply(jjm, 1, "+")
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    2    3    4    5
[2,]    6    7    8    9   10
[3,]   11   12   13   14   15
```

Learn the format that `data.dump` produces. Hack at it until you understand it.

Create a function that will read in double-precision numbers that are written in scientific notation with a "d" instead of an "e".

Make it so that the statement

```
paste("data", i, sep="") <- 1:i  # THIS WON'T WORK
```

will work. Is this a good thing to do?

Learn what each function in your version of S does.

4.14 Further Reading

For more on statistical functionality in S and S-PLUS, see Venables and Ripley (1997) *Modern Applied Statistics with S-Plus*, which comes with high praise. Spector (1994) *An Introduction to S and S-Plus* also surveys the statistical capabilities of S-PLUS. Chambers and Hastie (1992) *Statistical Models in S* describes the modeling functions that were produced at Bell Labs.

Books that describe some of the more esoteric statistical methods include McCullagh and Nelder (1989) *Generalized Linear Models*; Hastie and Tibshirani (1990) *Generalized Additive Models*; Bates and Watts (1988) *Nonlinear Regression Analysis and Its Applications*; Seber and Wild (1989) *Nonlinear Regression*; Breiman, Friedman, Olshen and Stone (1984) *Classification and Regression Trees*; Ripley (1996) *Pattern Recognition and Neural Networks*; Beran (1994) *Statistics for Long-Memory Processes*.

Many of the graphical ideas embodied in the trellis package are discussed in Cleveland (1993) *Visualizing Data*.

Reading help files and snooping through directories for undocumented objects are the best ways to increase your vocabulary. Of course for the undocumented (and under-documented) functions, you will need to use the ultimate documentation—the code itself.

Further functionality may be found in the S section of statlib. I'll not survey its contents because some uncooperative person would go and add something good after I made the list. Additions to the section are often announced on S-news.

4.15 Quotations

[13]Robinson Jeffers "Shine, Perishing Republic"

[14]Ezra Pound "In a Station of the Metro"

[15]Emily Dickinson 258 (There's a certain Slant of light)

[16]D. H. Lawrence "The Ship of Death"

[17]William Blake "Piping Down the Valleys Wild"

[18]Stevie Smith "No Categories!"

Chapter 5

Choppy Water

In which we learn to forecast the seas.

This covers only programming—problems with statistics and with graphics are not given.

5.1 Identity Crisis

Benjamin Wharf, a linguist, had the theory that our view of the world is shaped by the words of our language. He presented the example that lots of fires are started by "empty" gasoline cans. (He worked for an insurance company.) An "empty" gasoline container is actually full of vapors, which is the explosive part.

DANGER. Two of the most confusing functions are `is.vector` and `as.vector`. The common notion of "vector" is an atomic vector. However, these functions think of a vector as an object without attributes. A better name would have been `is.simple`.

DANGER. But wait, you also get inconsistency about names:

```
> jjvec <- jjvecn <- 1:26
> names(jjvecn) <- letters
> is.vector(jjvec)
[1] T
> is.vector(jjvecn)
[1] F
> jjlist <- jjlistn <- as.list(1:26)
> names(jjlistn) <- letters
```

```
> is.vector(jjlist)
[1] T
> is.vector(jjlistn)
[1] T
```

So to paraphrase, it is easier for a list to be a vector than for a vector to be a vector. You'll be glad to hear that `as.vector` and `is.vector` always agree with each other. Whenever you are tempted to use one of these functions, you should definitely ponder if this is the right thing.

DANGER. Related to this are functions like `as.double` and `as.character`. These, like `as.vector`, strip attributes off of the object as well as coercing the contents. To preserve the attributes, you want to assign to the mode or storage mode:

```
storage.mode(xmat) <- "double" # saves attributes
```

DANGER. However, you should note that an object may have attributes and still pass the `is.double` test. It is generally the case that `as.`*xxx*`(x)` is not guaranteed to be the same as `x` when `is.`*xxx*`(x)` is `TRUE`.

DANGER. `is.matrix` can also be a source of confusion. The definition of `is.matrix` is an object with a dim of length 2. Data frames fit this definition. The following nonsense

```
if(is.matrix(x)) x <- as.matrix(x)
```

actually is a useful idiom—it coerces a data frame to a matrix. `is.array` has similar behavior.

DANGER. I've said it elsewhere, but it is worth saying again that using `drop=F` when subscripting matrices and arrays in function definitions often prevents bugs.

DANGER.

```
> jjm6 <- matrix(1:6,3)
> jjm4 <- 10 + matrix(1:4,2)
> jjm6
     [,1] [,2]
[1,]    1    4
[2,]    2    5
[3,]    3    6
> jjm4
     [,1] [,2]
[1,]   11   13
[2,]   12   14
> jjm6[c(T,F,T),] <- jjm4[1,]
```

Here is our plan with the command above: that each of the rows marked TRUE in `jjm6` will end up being the same as the first row of `jjm4`. It's disappointment time, because S has a different plan.

```
> jjm6
     [,1] [,2]
[1,]   11   11
[2,]    2    5
[3,]   13   13
```

S thinks of this as a certain number of positions being filled in `jjm6` (arranged in the usual column-major order), and replicates the values on the right-hand side of the assignment to be the right length.

DANGER. You can use the `c` function to combine lists as well as atomic vectors. If one of the arguments is a list, then it is as if all of the other arguments are coerced to be lists with `as.list`.

```
> c(list("fjd",3:4), b=5:7)
[[1]]:
[1] "fjd"

[[2]]:
[1] 3 4

$b1:
[1] 5

$b2:
[1] 6
```

```
$b3:
[1] 7

> c(list("fjd",3:4), b=list(5:7))
[[1]]:
[1] "fjd"

[[2]]:
[1] 3 4

$b:
[1] 5 6 7
```

In most cases, it is the second of these that will be the proper formulation. If the vector is only of length one, then it doesn't matter which is used.

DANGER. There are times when S creates 1-dimensional arrays—arrays where the `dim` has length 1. One way to get such a critter is to use `t` to transpose an object that is not an array. It is rare that these appear, and even rarer that they are useful. The only problem with them is our tendency to think that they should be exactly the same as their non-array counterparts. They almost are.

DANGER. `tempfile` is another Wharfian word—it should be called *tempfilename* to show that it merely returns the name of a temporary file, rather than creating the file.

DANGER. When going from a floating point number to an integer, truncating is much faster than rounding. In the slow, old days that was an important consideration, and S still retains that feature. Whether the coercion is through `is.integer`, changing the storage mode to integer, or an internal coercion, numbers are essentially truncated. There is a little leeway—numbers that are very close to an integer are rounded up to the integer. S does an amazingly good job of it, but sometimes fails. The following example may be machine-dependent.

```
> (2 - .1)/.1
[1] 19
> as.integer(.Last.value)
[1] 18
```

Whenever you do computations to get numbers that are going to be coerced to integer, it is good practice to round them directly.

DANGER. S hides the fact that numbers can be represented as double precision floating point, integer, and so on. Most of the numbers that S creates are double precision. On very rare occasions a number stored as an integer will act differently than if it had been stored as a double. The most common way to get integers is with the `:` operator.

```
> storage.mode(1:10)
[1] "integer"
```

DANGER. Keep in mind that `is.integer` is a test of the storage mode, not of the values in the object. The vector `c(0, 4, 6)` contains values that are logically integer, but the storage mode is double.

DANGER. A complex vector does not test TRUE in `is.numeric`.

DANGER. An object need not be "null" just because it has length zero.

```
> is.null(numeric(0))
[1] F
```

Often where `is.null` appears, it is better to test if the length of the object is zero.

DANGER. When you are changing an option, then you have to get the name of the option precisely right. For instance if you want to turn warning messages into errors, and you type

```
options(warning=2) # WRONG
```

then you won't get what you want because the name of the option is `warn`, you need to say

```
options(warn=2)
```

The `soptions` function, given on page 45, guards against this problem.

DANGER. The `all.equal` function does not return a logical value when the objects are not equivalent. So code like:

```
if(!all.equal(x, y)) ...
```

is not going to work properly.

DANGER. This is quite picky, but if you are in a picky mood ...

```
> all.equal(as.numeric(NA), 0/0)
[1] T
> is.nan(as.numeric(NA))
[1] F
> is.nan(0/0)
[1] T
```

NaN's and NA's are not distinguished by `all.equal`.

DANGER. The "not" operator can not be the first character in a command because it will be interpreted as an escape to the operating system.

```
> !is.na(longley.y)
Badly placed ()'s
> (!is.na(longley.y))
 1947 1948 1949 1950 1951 1952 1953 1954 1955 1956
    T    T    T    T    T    T    T    T    T    T
 1957 1958 1959 1960 1961 1962
    T    T    T    T    T    T
```

The "Badly placed" message is a syntax error from Unix.

DANGER. The `lag` function does not change the actualy data, it only changes the time points that the observations represent. Furthermore, in a sense it is "lead" rather than "lag".

5.2 Off the Wall

DANGER. As far as I know there is only one spot in S where a space makes a difference. If you type `x<-2` and expect a logical vector containing the values of `x` that are less than `-2`, you will be saddened to learn that you don't get your logical vector, and you have just destroyed `x` by assigning it the value 2.

DANGER. S has a number of reserved words that you may not use as the name of an object. Probably the most common reserved words that you might want to use are `F`, `T`, `return`, `break`, `next`. You can see an example of this problem on page 87.

DANGER. A natural thing to try when testing for missing values is:

```
> longley.y == NA
 1947 1948 1949 1950 1951 1952 1953 1954 1955 1956
   NA   NA   NA   NA   NA   NA   NA   NA   NA   NA
 1957 1958 1959 1960 1961 1962
   NA   NA   NA   NA   NA   NA
```

I've tried this myself a few times years after I knew better. The problem is that the `NA` is replicated to the length of `x` and then the answer is that number of missing values. The proper way to get this done is to use `is.na`:

```
> is.na(longley.y)
 1947 1948 1949 1950 1951 1952 1953 1954 1955 1956
    F    F    F    F    F    F    F    F    F    F
 1957 1958 1959 1960 1961 1962
    F    F    F    F    F    F
```

DANGER. Do not modify the looping variable in a `for` loop. You will confuse at least yourself and possibly S also.

DANGER. At least for version 3, you want corresponding arguments in the generic function and its methods to have the same name. Suppose that the first argument to the generic function is `x` and the corresponding argument in one of the methods is `object`. When we call the generic function, the argument is evaluated and called `x`. The class says to use our method, so that function becomes the operable one. The method needs `object` so it evaluates the first argument, again. In the best circumstances this is merely inefficient. At times, though, the double evaluation can be a bug—for instance, if randomness is involved.

Let's make it even worse. If the method has `object` as the argument corresponding to `x` in the generic function, and the method also has `x` as an additional argument, real trouble develops. Now when we get to the method, it re-evaluates the first argument to get `object`, but it doesn't evaluate `x` because it already knows what `x` is (though it is the wrong `x` as far as the method is concerned). *And what a congress of stinks!* [19]

DANGER. To create a vector of doubles that is 10 long, you say `double(10)` and you say `integer(10)` to create a vector of 10 integers. But `list(10)` does not create a list that is 10 long—it creates a list that has one component (that is the number 10). To initialize a list with length 10, you need to use the circuitous statement:

```
vector("list", 10)
```

This is another reminder that the word "vector" has more than one meaning in S.

DANGER. A common code fragment goes like

```
for(i in 1:length(x)) ...
```

This is fine, except when the length of `x` is zero, in which case `i` is first 1 and then 0. Seldom what you want. If there is a chance that the length of `x` will be zero, then use the following instead:

```
for(i in seq(along = x)) ...
```

DANGER. While we are on the subject, another popular way to mess this up is to say:

```
for(i in length(x)) ...
```

This only does the last of the intended iterations.

DANGER. If you dump (using `dump`) an object that contains a call component or attribute, then the restoration is not going to work. The call component will be evaluated instead of remaining merely an image of what to evaluate. This is another reason to use `data.dump` when making ASCII representations of objects that are not functions.

DANGER. In version 3 of S-PLUS a common idiom to arrange for object code to be loaded automatically is something like:

```
if(!is.loaded(symbol.C("myfun_Sp")))
    dyn.load("myfun.o")
```

If you do this, make sure that you get the symbol right for `is.loaded`. In particular, you want to spell the routine name correctly and you do not want to forget the `symbol.C` or `symbol.For`. If you get it wrong, then the loading will occur each time that the function is called instead of once per session. This isn't so bad if it is only called a few times in a session. But when it is called hundreds of times, then pieces of memory get eaten by the loaded code, your memory gets fragmented, and you can start paging or run out of memory all together.

DANGER. Consider the following function.

```
"switch.chartest"<-
function(this.op)
{
        ans <- switch(this.op,
                a = {
                        cat("doing a\n")
                        1
                }
                ,
                b = {
                        cat("doing b\n")
                        2
                }
```

```
                ,
                {
                        cat("doing other\n")
                        3
                }
                )
        ans
}
> switch.chartest
```

Now give this a few values:

```
> switch.chartest("a")
doing a
[1] 1
> switch.chartest("b")
doing b
[1] 2
> switch.chartest(" ")
doing other
[1] 3
> switch.chartest("")
```

It is fine except when we give it an empty string. When `switch` dispatches on an empty string, then it does nothing instead of doing the default code.

```
> jj <- switch.chartest("")
> jj
> mode(jj)
[1] "missing"
```

And the nothing that it does is a little bizarre. This is a bug, and it exists at least in S-PLUS version 3.4 and earlier.

5.3 Factors

There are a number of sticky bits with objects of class `factor`. Unfortunately, this section is not likely to cover all of the problems. Examples will use the object `jjfac`.

```
> jjtypen
  Harold Dorothy Munchkin Stevie
 "corgi" "corgi" "cat"    "cat"
> jjfac <- factor(jjtypen)
```

DANGER. The first thing that is strange is that the names are not printed even though they are there.

```
> jjfac
[1] corgi corgi cat   cat
> unclass(jjfac)
 Harold Dorothy Munchkin Stevie
      2       2        1      1
attr(, "levels"):
[1] "cat"   "corgi"
> names(jjfac)
[1] "Harold"   "Dorothy"  "Munchkin" "Stevie"
```

This is a design decision that is sensible once you consider the alternatives. Factors are printed as strings with no quotes around them, so are names. So if the names were printed, it would be rather ambiguous at times which corresponded to names and which to the data. If factors used quotes, then they would be hard to distinguish from character data.

DANGER. Here is reasonable behavior from a version of S-PLUS:

```
> factor(factor(jjtypen, exclude="cat"))
 Harold Dorothy Munchkin Stevie
 corgi  corgi   NA       NA
> version
Version 3.1 Release 1 for Sun SPARC, SunOS 4.x : 1992
```

Though note that the names are printed here, unlike in other cases. Now with a later version:

```
> factor(factor(jjtypen, exclude="cat"))
 Harold Dorothy Munchkin Stevie
 NA     NA      NA       NA
Levels:
[1] "1"
> version
Version 3.4 Release 1 for Silicon Graphics Iris, IRIX 5.3 : 1996
```

Obviiously, someone fixed something. (This is stolen from a message to S-News by John Maindonald.)

DANGER.

```
> labels(jjfac)
[1] "Harold"   "Dorothy"  "Munchkin" "Stevie"
```

There is a `labels` argument to `factor`, but this is not the same as the result of the `labels` function on the factor. The `labels` argument tells how to make the `levels` attribute of the factor. The `labels` function is generic and is meant to give a name to each datum in the object, in this case it is the same as the names.

DANGER. Coercion from factor is not necessarily smooth and enlightened. (Z. Todd Taylor pointed this out to S-news.)

```
> paste(names(jjfac), "is a", jjfac)
[1] "Harold is a 2"   "Dorothy is a 2"
[3] "Munchkin is a 1" "Stevie is a 1"
```

Though this answer may arguably be true, it is not what some of us would have in mind. We need to explicitly coerce to character.

```
> as.character(jjfac)
[1] "corgi" "corgi" "cat"   "cat"
> paste(names(jjfac), "is a", as.character(jjfac))
[1] "Harold is a corgi"  "Dorothy is a corgi"
[3] "Munchkin is a cat"  "Stevie is a cat"
```

Those involved are happier with this.

DANGER. There is not a `factor` method for `c` (at the time of writing at least).

```
> jjpet <- c(Sybil="cat", Shera="cat", Fred="chinchilla")
> jjpetfac <- factor(jjpet)
> jjpetfac
[1] cat        cat        chinchilla
> c(jjfac, jjpetfac)
 Harold Dorothy Munchkin Stevie Sybil Shera Fred
      2       2        1      1     1     1    2
> unclass(.Last.value)
 Harold Dorothy Munchkin Stevie Sybil Shera Fred
      2       2        1      1     1     1    2
```

We really want to take this more scenic route.

```
> factor(c(as.character(jjfac), as.character(jjpetfac)))
[1] corgi      corgi      cat        cat
[5] cat        cat        chinchilla
```

DANGER. Be careful when you use the `codes` function. The levels may be permuted because `codes` sorts them.

```
> jjfac2 <- factor(jjtypen, levels=c("corgi", "cat"))
> unclass(jjfac2)
 Harold Dorothy Munchkin Stevie
      1       1        2      2
attr(, "levels"):
[1] "corgi" "cat"
> codes(jjfac2)
[1] 2 2 1 1
```

Here `codes` gives a different code than what is actually in the object since the levels are not in alphabetic order.

DANGER. You want to pay a little attention when you are creating ordered factors.

```
> jjo
[1]  90  90 100  90 110 100 110
> ordered(jjo)
[1] 90  90  100 90  110 100 110

 90 < 100 < 110
```

`jjo` is a numeric vector. We make an ordered factor with it, and everything is fine. However, the same character vector doesn't give the same answer.

```
> ordered(as.character(jjo))
[1] 90  90  100 90  110 100 110

 100 < 110 < 90
```

Now we get 90 being greater than 110. The sorting is being done in dictionary order rather than numeric order.

DANGER. There is a special idiom for coercing an ordered factor that represents numbers into those numbers.

```
> jjof <- ordered(jjo)
> jjof
[1] 90  90  100 90  110 100 110

 90 < 100 < 110
> as.numeric(jjof)
[1] 1 1 2 1 3 2 3
> as.numeric(as.character(jjof))
[1]  90  90 100  90 110 100 110
```

Just using `as.numeric` is tempting, but not right. All sorts of trouble might ensue if it were changed so that it did work since the object underlying factors is a numeric vector.

DANGER. A reason that many of the problems with factors come up so often is that data frames are excessively fond of factors—they favor factors over character data. When a data frame is created, often a character vector is coerced to be a factor.

5.4 Things to Do

Create a new class of objects for categorical data that has none of the problems that factors do.

Look through your functions for these trouble spots.

5.5 Further Reading

The S-news list.

5.6 Quotations

[19]Theodore Roethke "Root Cellar"

Chapter 6

Debugging

In which we learn to bail the boat.

When S reads a line of your input (that is, *parses* it), there are three possible results:

- There is a syntax error.
- The command is incomplete.
- The command is complete.

Once a comand is complete, S will go on to evaluate the command.

If you get a syntax error, there is really no alternative to puzzling over the line in question with the hint that S gives you. *When a crop is so thin* [20]

When you write code in an editor, then you can save yourself a few occurrences of errors due to missing braces by typing the braces before filling them in. So at one point you have:

```
if(length(x)) {
} else {
}
```

then you go back and fill in the empty braced expressions. You might think that this is slow because you have to back up and fill in, but you save a lot of time by not having as many syntax errors, and by not needing to count braces as you type. It can also save you from getting the logic wrong.

6.1 Masking

Consider the two commands:

```
> c <- matrix(1:9, 3)
> t(c)
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    5    6
[3,]    7    8    9
Warning messages:
  Looking for object "c" of mode "function", ignored
        one of mode "integer" in: print.structure(x, a,
        quote = quote, prefix = prefix)
```

Everything seems fine at the first command, but on the second we get a mysterious warning message. It can seem particularly mysterious when such warnings start appearing a long time after the assignment is made.

Here is what is going on. There is an in-built function named `c` that is used in the process of printing the transposed matrix, but we have created another object named `c` that S sees first. Since the `c` that we created is not a function, it continues searching until it finds a function named `c`, and uses that. S is just being friendly when it issues the warning. We can avoid such warnings by renaming our object. In addition to `c`, other likely names that are good to avoid are `t`, `C` and `s`. (The names `T` and `F` are prohibited.)

In contrast, consider the following command:

```
# DO NOT DO THIS
> c <- function(...) cat(list(...)[[1]])
Warning messages:
  assigning "c" masks an object of the same name on
        database 5
```

The result of this is a different warning message.

Now the `c` that we just created will be used instead of the in-built `c` function (and trouble will ensue). Unless you really mean to replace the in-built function, rename your function immediately.

The S-PLUS function `masked` can be used to see what objects in the current directory have the same names as objects in the rest of the databases on the search list.

```
> masked()
[1] ".Random.seed" "c"
```

We see our `c` function that is masking the real one (it would also appear if it weren't a function). `.Random.seed` is also masked—this is fine, it merely means that we have used a random function with this working database. A danger of masking functions becomes clear when we try to get rid of our version of the `c` function.

```
> rm(c)
Error in remove(list): need names of objects to remove,
        not the objects themselves
Dumped
> remove("c")
> masked()
[1] ".Random.seed"
```

When the bizarre occurs, consider masking as a likely source of the problem.

6.2 Dumping

So what can go wrong? You can get a syntax error, you can get "object not found" errors, and other sorts of errors. Almost all errors except syntax errors and system terminating result in the error action. When a dumpable error happens, then S performs the action given by the `error` option. In version 3 there are basically three choices for this option. It can be `NULL`, which is seldom useful; it can be `dump.calls`, the default; or `dump.frames`.

Each of `dump.calls` and `dump.frames` creates an object named `last.dump` in the working database. When `dump.calls` is used, then `last.dump` contains the string of function calls that was in effect at the time of the error. When `dump.frames` is used, then `last.dump` contains not only the function calls, but also all of the objects in the frame for each function call.

Both `traceback` and `debugger` use `last.dump` (by default). `traceback` is essentially just a printing method to show the function calls. Although very simple, this is often enough information to pinpoint the problem. I think of `traceback` as the first thing to do after an error dump.

If `traceback` is not enough to comprehend the error, then `debugger` can be used. This allows you to look at objects within specific frames as they were at the time of the error. The `debugger` function has the same sort of interface as `browser`. In order to use `debugger` except as a synonym for `traceback`, you need to set the `error` option to `dump.frames` like:

```
options(error=dump.frames)
```

or

```
options(error=expression(dump.frames()))
```

If this was not done prior to the time of the error, then you will need to recreate the error in order to look at objects with `debugger`.

There are times when you get a warning that you know is an indication of trouble, but you don't understand why it is happening. This is when you should turn a warning into an error. Do this with the command:

```
options(warn=2)
```

You can then recreate the warning, which will now be an error, and use the same tools as with any other error.

6.3 Exploring

When you don't have an error condition, but either something is wrong or you are not sure that everything is right, then you need some means of exploring the computations.

A very good function for this is `browser`. This function allows you to look at the objects in the frame of a function as they exist at the time of the call to `browser`. You can use `traceback` when you are at a `browser` prompt to see who called whom. *This is a boy that pats the floor to see if the world is there, is flat* [21]

I have considered making a version of `fix` that automatically puts a call to `browser` in the definition of a new function. I use `browser` a lot to test functions as I write them. Write part of the function, make sure that the result so far is as expected; write some more, check that. I find this more productive (and fun) than trying to decipher errors from a function that I don't check as I go.

(It is reputed that Charlie Chaplin didn't care if he won a volley of tennis—the important bit was looking graceful while he played. I suspect he was a formidable tennis opponent.) It's not important that every line you write works right off, what is important is that you not assume code works unless you have evidence.

DANGER. The `browser.default` function has a test in it to see if S is being used interactively. If not, then an error occurs. Thus, a BATCH job can be ruined if you leave a call to `browser` in one of the functions it uses. The top of `browser.default` can be changed to something like the following:

```
if(!interactive()) {
        warning("for interactive use")
        return(NULL)
}
```

This makes it so that `browser` is ignored when it is called in a non-interactive setting except that a warning is issued. This seems a punishment more fitting of the crime. *they're taking him to prison for the colour of his hair* [22]

When you are in `browser`, you can assign objects to the working database with the `<<-` operator so that you can use those objects for testing. It is often natural at this point to interrupt (hit control-backslash) the browsing instead

of going through the whole computation. If you do, then your variables won't be there. The problem is that the assignments are not committed. You can interrupt the browsing, but you need to synchronize the database before you do. See talk of `synchronize` on page 103.

Another exploration technique is the old standby of putting `print` and `cat` statements in function definitions. This works in any language, so it is quite popular. An advantage over `browser` is that the computations are not interrupted, so you can see a lot of information with minimal effort.

The `trace` function allows you to get information each time a traced function is called. An example of a good use for `trace` is that you can trace all of the methods of a generic function, so that you can be sure that the functions being called are the ones that you expect.

DANGER. `trace` creates modified versions of the functions being traced. You want to be careful when editing functions that you do not edit a traced version because you will end up with a bunch of garbage in the definition that you will need to get rid of later.

Version 3.2 of S-PLUS introduced the `inspect` function. This is reminiscent of debuggers (like `dbx`) for C and Fortran code. You give `inspect` a command to evaluate, like:

```
inspect(myfun(myarg1, myarg2))
```

You can then step through the calculations and look at objects as you go. `inspect`, like `trace`, creates modified versions of functions that are being inspected, but unlike `trace` the functions that `inspect` creates disappear right away so there is no chance of inadvertently getting the modified version.

You generally need to use the techniques of this section to discover where a "system terminating" happens. System termination occurs when S detects an inconsistency, and it kills itself rather than risking the possibility of corrupting something permanently. A system termination means that there is a bug somewhere.

Almost always a system termination is the result of a problem with a call to `.C` or `.Fortran` that has been added locally. Usually (but not necessarily) the termination occurs during the call to C or Fortran. Once you believe that you know the call that is causing the problem, there are a few things to check. This is discussed on page 179.

6.4 Examples

The best way to learn to debug is to see it done. Here are some examples of actual debugging in S. As you might expect, they have been streamlined to avoid some of the dead ends.

the dimnames bug

The presentation of this example definitely starts toward the end of the process. I got a "system terminating" while using a fairly involved function; it took several iterations to pare away pieces of the function in order to narrow down the location of the problem. The obvious first step was to look for local functions that contained `.C` or `.Fortran`, but there were none.

Here is a simple example where everything works.

```
> jjmat <- matrix(1:6, 2)
> dimnames(jjmat) <- list(1:2, c("a","b","c"))
> jjmat
  a b c
1 1 3 5
2 2 4 6
> rbind(jjmat, jjmat)
  a b c
1 1 3 5
2 2 4 6
1 1 3 5
2 2 4 6
```

Now we do something that should change nothing at all.

```
> dimnames(jjmat)[[1]] <- 1:2
> jjmat
  a b c
1 1 3 5
2 2 4 6
> rbind(jjmat, jjmat)
System terminating: bad address
```

But in fact the behavior is significantly different as you can see. The sharp-eyed will recognize what the problem is in the output of the next command.

```
> dimnames(jjmat)
[[1]]:
[1] 1 2

[[2]]:
[1] "a" "b" "c"
```

A broader hint is provided by this command:

```
> mode(dimnames(jjmat)[[1]])
[1] "numeric"
```

The bug is that when a component of the dimnames of an array is assigned a numeric vector, the coercion to character does not take place. The bug is insidious since not all operations are affected by it.

This example shows off none of the debugging tools in S. There aren't any that are especially useful. You can try using `trace` or inserting `cat` statements to identify just where the "system terminating" occurs, but you are on your own after that.

the print.matrix bug

Here we have a matrix, and when we try to print it out with only a few digits, we see that the `digits` argument is not respected.

```
> jjpbx <- as.matrix(data.frame(a=1:3, b=rnorm(3)))
> jjpbx
  a          b
1 1 -0.7710749
2 2  1.3230554
3 3 -1.0032299
> print(jjpbx, digits=2)
  a          b
1 1 -0.7710749
2 2  1.3230554
3 3 -1.0032299
```

There is no error, and not even a warning to force into an error. So we need some other means to ferret out where the problem lies. One idea is to trace all of the functions that contain the word "print" in their name.

```
> trace(find.objects("print"))
> print(jjpbx, digits=2)
On entry:         print.matrix(jjpbx, digits = 2)
  a          b
1 1 -0.7710749
2 2  1.3230554
3 3 -1.0032299
Warning messages:
  Looking for object "print" of mode "function",
      ignored one of mode "{"
```

(The warning message is indicating that there is a bug in `trace` that will give us another opportunity to practice debugging.) What we've learned is that the

`print.matrix` function is used, and it is the only function used that is currently being traced.

Below we see that the object has class `"matrix"`, and that the bug disappears when we unclass the matrix. This is the key step in the debugging, but was arrived at by lucky accident more than by the inevitable discovery of our debugging strategy.

```
> class(jjpbx)
[1] "matrix"
Warning messages:
  Looking for object "print" of mode "function",
      ignored one of mode "{"
> print(unclass(jjpbx), digits=2)
  a     b
1 1 -0.77
2 2  1.32
3 3 -1.00
Warning messages:
  Looking for object "print" of mode "function",
      ignored one of mode "{"
> untrace()
```

This is substantial progress—the bug has been isolated to the `print.matrix` function. The task is now to understand what is happening inside `print.matrix`. Below is the fragment of `print.matrix` that seems to be pertinent.

```
        if(length(clab)) {
                nd <- switch(mode(x),
                        numeric = if(missing(digits))
                                  .Options$digits
                                 else digits,
                        character = max(nchar(x)) + 2,
                        complex = 2 * (if(missing(
                                digits)) .Options$
                                  digits else digits),
                        2)
                if(abbreviate.labels && max(nchar(clab
                        )) > nd + 3)
                        clab <- abbreviate(clab, c(nd,
                                nd + 3))
        }
        prmatrix(x, rowlab = dn[[1]], collab = clab,
                quote = quote, right = right)
```

We place a call to `browser` just before the call to `prmatrix`, and then try printing to see what happens.

```
> print(jjpbx, digits=2)
Called from: print.matrix(jjpbx, digits = 2 . . .
b(2)> digits
[1] 2
b(2)> traceback()
5: eval(i, eval.frame, parent) from 4
4: browser.default(nframe, message = paste("Called from:", from 3
3: browser() from 2
2: print.matrix(jjpbx, digits = 2) from 1
1:  from 1
b(2)> 0
  a          b
1 1 -0.7710749
2 2  1.3230554
3 3 -1.0032299
```

The `digits` object is as we expect, but still the function doesn't work. Now we try the same thing using `print.default`.

```
> print.default(jjpbx, digits=2)
Called from: print.matrix(x, quote = quote)
b(4)> digits
NULL
b(4)> traceback()
7: eval(i, eval.frame, parent) from 6
6: browser.default(nframe, message = paste("Called from:", from 5
5: browser() from 4
4: print.matrix(x, quote = quote) from 3
3: print.structure(x, a, quote = quote, prefix = prefix) from 2
2: print.default(jjpbx, digits = 2) from 1
1:  from 1
b(4)> 0
  a     b
1 1 -0.77
2 2  1.32
3 3 -1.00
attr(, "class"):
[1] "matrix"
```

This time `digits` is `NULL`, which seems wrong, but the function works.

A little thought and investigation shows that the problem is that `digits` needs to be fed to `options`, which is not happening when the computations go directly to `print.matrix`. We solve the bug by adding the following `if` statement to the top of `print.matrix`.

```
if(!is.null(digits)) {
```

```
                if((length(digits) != 1 || digits < 1) ||
                        digits > 20)
                        warning("Bad value for digits")
                else {
                        on.exit(options(d))
                        d <- options(digits = digits)
                }
        }
```

Now we get what we want, and we check that the `digits` option really is put back to the default value.

```
> print(jjpbx, digits=2)
  a     b
1 1 -0.77
2 2  1.32
3 3 -1.00
> options("digits")
$digits:
[1] 7
```

the trace bug

Here we investigate the problem of tracing `print` that we spotted in the last example. My first hypothesis was that there's a problem tracing functions whose body is not wrapped in braces. Tracing a function like that worked fine so let's try tracing the `print` function, but by a different name.

```
> fjj <- print
> trace("fjj")
> fjj(fjj)
function(x, ...)
{
        if(.Traceon) {
                .Internal(assign(".Traceon", F, where = 0),
              "S_put")
                cat("On entry: ")
                std.trace()
                .Internal(assign(".Traceon", T, where = 0),
                    "S_put")
        }
        UseMethod("print")
}
```

This works fine, but tracing the `print` function does not.

```
> trace("print")
> print
{
        if(.Traceon) {
                .Internal(assign(".Traceon", F, where = 0),
                    "S_put")
                cat("On entry: ")
                std.trace()
                .Internal(assign(".Traceon", T, where = 0),
                    "S_put")
        }
}
Warning messages:
  Looking for object "print" of mode "function",
       ignored one of mode "{"
> untrace()
```

Now it is obviously the case that the name "print" is the culprit. The `trace` function has an argument named `print`, which appears to be the only possible source of confusion.

A solution is to add a dot to the end of the `print` argument in `trace` so that it becomes the `print.` argument. This has no effect on the use of the function.

```
> trace("print")
> print
On entry:         print(print)
function(x, ...)
{
        if(.Traceon) {
                .Internal(assign(".Traceon", F, where = 0),
                    "S_put")
                cat("On entry: ")
                std.trace()
                .Internal(assign(".Traceon", T, where = 0),
                    "S_put")
        }
        UseMethod("print")
}
> untrace()
```

It is now possible to trace `print` without the extraneous warning.

So we are in the highly unlikely situation of confidently producing a fix for the problem while having only a superficial understanding of how the problem arises. But our curiosity gets the better of us and we continue the investigation in order to understand how the `print` argument confuses the production of a `print` function.

We use `inspect` to examine the computation (using the original definition of `trace`).

```
> inspect(trace("print"))
entering function trace
stopped in trace (frame 3), at:
        exprs <- expression({
                NULL
                NULL
        ...
d> step
```

Here I omit a number of `step` commands in order to get to the interesting part.

```
d> eval name
[1] "print"
d> eval print
[1] T
d> eval what
[1] "print"
d> step
stopped in trace (frame 3), at:
        fun <- get(name, mode = "function")
d> step
stopped in trace (frame 3), at:
        get(name, mode = "function")
d> step
stopped in trace (frame 3), at:
        n <- length(fun)
d> eval fun
[1] T
```

Here is the trouble. The `fun` object is getting the `print` argument rather than the `print` function even though we are trying to restrict the `get` to functions. We continue to explore.

```
d> eval mode((print)
Syntax error: end.of.file ("\n") used illegally at this point:
mode((print)

Calls at time of error:

7: error = function() from 6
6: parse(text = s) from 5
5: eval.it(str.args[1], local = loc.frame) from 4
4: debug.tracer(what = TR.GENERIC, index = c(6, 14, 3, 4)) from 3
3: trace("print") from 1
```

```
2: inspect(trace("print")) from 1
1:  from 1

Dumping frames ... Dumped

local frame (frame of error) is parse (frame 6)
d> quit
d> eval mode(print)
[1] "logical"
d> eval get("print", mode="function")
[1] T
```

The first attempt to look at the mode of `print` resulted in an error due to a typo, so `inspect` essentially calls itself on the error. We don't care about this error, so we tell `inspect` to quit in order to get back to the function that we do care about. Finally, we confirm that `get` is doing the wrong thing.

We have now solved the mystery of how `trace` is going wrong—there is nothing wrong with `trace`, the problem is that there is a bug in `get`. Now we have to regard our fix for `trace` as a workaround for the `get` bug.

the get bug

We now try to zero in on the real problem with `get`. The first step is to create a simple function that reproduces the behavior.

```
"fjjget"<-
function(print = T, mode = "function")
{
        print   # this line to evaluate it
        ans <- get("print", mode = mode)
        ans
}
```

The `print` argument needs to be used before the call to `get` in order for it not to be of mode `argument`. The function does indeed exhibit the problem.

```
> fjjget()
[1] T
> fjjget(mode="function")
[1] T
> fjjget(mode="logical")
[1] T
> fjjget(mode="numeric")
[1] T
> fjjget(print=print, mode="numeric")
function(x, ...)
```

```
UseMethod("print")
```

It appears to be the case that the `mode` argument is ignored entirely when the object exists in the current frame. Let's look at `get.default` to see what we are dealing with.

```
> get.default
function(name, where = NULL, frame, mode = "any",
        inherit = F, immediate = F)
{
        code <- if(immediate) 20 else 0
        if(missing(where)) {
                if(missing(frame))
                        .Internal(get(name, mode,
                                inherit), "S_get", T,
                                code)
                else .Internal(get(name, mode, frame),
                                "S_get", T, code + 2)
        }
        else .Internal(get(name, mode, where), "S_get",
                        T, code + 1)
}
```

We try a new function that restricts the search to a frame to see what part of the code is broken.

```
> fjjget2
function(print = T, mode = "function")
{
        print   # this line to evaluate it
        ans <- get.default("print", mode = mode, frame
                 = 2)
        ans
}
```

The following example shows that this works okay. It creates an error, so we go into the debugger to verify the value of `print`.

```
> fjjget2(print=print, mode="numeric")
Error in get.default("print", mode = ..: Object "print"
        of mode numeric not found
Dumped
> debugger()
Message: Object "print" of mode numeric not found

1:
2: fjjget2(print = print, mode = "numeric")
```

```
3: get.default("print", mode = mode, frame = 2)
Selection: 2
Frame of fjjget2(print = print, mode = "numeric")
d(2)> print
function(x, ...)
UseMethod("print")
d(2)> 0

1:
2: fjjget2(print = print, mode = "numeric")
3: get.default("print", mode = mode, frame = 2)
Selection: 0
NULL
>
```

Our conclusion is that the `mode` argument to `get.default` is ignored if the search is unrestricted and the object is found in the current frame. We could continue to see just what goes wrong, but since the function is internal, we can't do anything about it anyway.

6.5 Strategy

Remember to breathe.

The primary thing that you want to do when debugging is to continually simplify the problem. This narrows down where the problem can be. For example, when we thought there was a bug in `get` that was causing trouble in `trace`, we wrote a simple function that imitated the pertinent part of the situation. When you do this, there can be one of two outcomes; either the problem still occurs in which case you have a simpler problem to solve, or the bug does not occur in which case you have proved there are more necessary conditions than are in the simplification. *For nothing can be sole or whole That has not been rent.* [23]

Bentley (1986) offers these words of advise: "understand the code at all times, and resist those foul urges to 'just change it until it works' ". The urges are foul because this tactic is likely to cause other bugs down the line, while the real bug is here where it looks right.

There are times of frustration when everything is right except the answer. When this happens to you, remember the following rule.

Capricious Rule 1 *The harder a bug is to find, the sillier the error.*

For example, I once spent three full days figuring out a bug in some Fortran. It turned out that the problem was merely that a number that had been scaled

at the start was not scaled back to the original units. Once I realized that, I felt like a fool because several of the symptoms pointed directly there—but it was too simple, and I couldn't see it. You might try to give me an out by saying that the code wasn't mine. It would have been four days if it were my own because I would be even more sure that everything was right.

When it is taking a long time to find a bug, switch to searching for the stupidest mistakes possible. Often it helps to make a hardcopy of the code—it can be easier (for some psychological reason beyond my understanding, and unfortunately for the trees) to spot mistakes on paper than it is on a computer screen.

6.6 Bug Reports

Most of the time when programming, the problems that arise are self-imposed. Ocassionally, however, you may find a bug in code that comes from somewhere else. You should then send a bug report to the responsible party. The easier you make it to fix the bug, the more effective your report. Here are the ingredients of a good bug report:

• Identify the software and hardware involved. Always include the general type of machine, e.g., Sparc, Pentium, and it is good practice to be more specific. State what operating system is in use, e.g., SunOS 4.1.3. State the version of S, e.g., S-PLUS 3.3. Give this information even if you think it is superfluous—it may not be. If the problem involves graphics, then whatever software and hardware is involved with the graphics should also be identified.

• Give a simple, direct way to reproduce the problem. The example should either be self-contained, or only use objects that are easily obtained by the readers of the report. Random numbers are suitable if you give the seed so that they are reproducible.

• If the problem results in an error condition, give a traceback of the error.

• State what is wrong. Even if it seems obvious to you, it may not be obvious to the reader.

• State what good behavior would be.

• State as best you know the limits of the problem. For example, you might say, "The problem seems to only occur when there is at least one missing value in argument blah of function blahblah."

• If you think you know how to fix the problem, give your fix.

• Include any workarounds. A workaround can be a kludgy fix to the problem function, or a way to get the intended result while avoiding the problem.

Focus on the part about getting a simple example. Many times this exercise will show you that the bug is in your way of thinking rather than in the software. Other times it will result that there is a bug, but it is different than you had thought. It is always educational—I don't think there is a better way to learn a language than tracking down bugs.

Here is an example bug report:

Dear Pat,

When I try to use "genopt", it breaks.

This clearly needs work. Assuming the best of circumstances that Pat is highly motivated to have "genopt" as clean as possible, and that he has ample time to work on it, there is virtually no chance of Pat divining what the reporter witnessed. If Pat is short on either pride or time, then reports like this will make no dent in his day.

It bears repeating that the object is to allow the receiver to identify, reproduce and fix the bug as easily as possible.

I have (of course) presented the easy case. Some bugs are intermittent, and you may not be able to find a way to consistently reproduce the problem. Reporting such behavior can be useful, but you should not raise your hopes too high about the bug being found. Sometimes the data that you are using when the bug occurs is confidential. Try using different data. If that doesn't work, you can consider disguising the data so that confidentiality is not broken. If all else fails, you can send the report without data that reproduces the problem.

I make it a policy to never submit a bug report the same day that I write it. At worst I can merely check over it with a fresh mind to ensure that others will understand it. At best I will realize that there was some twist that I missed, and the whole report was off the mark.

This is a bit of a digression, but the same sort of principles apply when asking questions on S-news or similar venues. Here is a minimal list of things to include in a question to S-news:

• The version of S or S-PLUS that you are using, including the machine type.

• What you want to do. Include enough detail so that it will be clear to the reader.

• Why you want to do it. It is often the case that there is a simpler way to achieve the goal that may have nothing to do with your specific question.

6.7 Things to Do

Write your own error action function like `dump.frames`. What do you want it to do? What happens when there are bugs in it that cause an error condition?

Send bug reports for any problems that you know about.

6.8 Further Reading

Programming Pearls by Jon Bentley includes some debugging advise.

If you are overly frustrated, consider reading something humorous, like a story by P. G. Wodehouse.

6.9 Quotations

[20]Cecil Day-Lewis "A Failure"

[21]Elizabeth Bishop "Visits to St. Elizabeths"

[22]A. E. Housman "Oh Who is That Young Sinner"

[23]William Butler Yeats "Crazy Jane Talks with the Bishop"

Chapter 7

Unix for S Programmers

You generally need to know very little about your operating system in order to use S. But there are a few things that can be helpful. If you are not using some variant of Unix, then you may wish to skip this chapter, though toolkits do exist so that you can get much of the functionality discussed here.

7.1 Environment Variables

The two fundamental entities in Unix are *processes* and files. Files are pretty intuitive except that they are abstracted to include more than you might expect. Simple Unix commands result in one process being created, other commands (like starting S) result in a number of processes. Each process has a parent, and may spawn children of its own.

The program that gives us a Unix prompt is a process. When we start S (for example), the process started for S is the child of the process that gave us the Unix prompt. The child process inherits the *environment* of the parent. The child can then change its environment. The environment is the collection of environment variables and the values that they have.

By convention environment variables are all in upper-case. To access the value of a variable, put a `$` in front of the variable name. So you could issue a Unix command like

```
ls $HOME
```

to list the files in your home directory.

S uses a number of environment variables. Since an escape to Unix creates a subprocess of S, you can see S's environment with the S command:

```
> !env
```

(or possibly `!printenv`). Here is an edited version of the result with only the most pertinent variables shown. This is using version 3.3 of S-PLUS on an SGI machine.

```
> !env
DISPLAY=pburns.seanet.com:0
HOME=/byron/users/burns
LD_LIBRARY_PATH=/usr/lang/SC1.0
LESS=-PPaging with 'less'
MANPATH=/usr/local/man:/usr/share/catman:/usr/man:/usr/catman
PATH=/usr/lang/gnu/bin:/usr/lang/gnu/lib:/usr/sbin:/byron/users/bur
ns/bin:/usr/local/bin:/bin:/usr/bin/X11:/usr/bsd:.:/usr/etc:/etc:/u
sr/bin
PWD=/user/users/burns/spo
REMOTEHOST=pburns.seanet.com
SHELL=/bin/csh
SHOME=/byron/splus/3.3/sgi
SLIC_RSHCMD=rsh %s env SHOME=$SHOME Splus LICENSE server start
S_CC_FLAGS=
S_CLHISTFILE=/byron/users/burns/.Splus_history
S_DEFAULT=NEW
S_DL2_EXTRAS=
S_FRONT_END_RUNNING=1
S_LASERJET_DPI=300
S_LASERJET_PRINT_COMMAND=lp -Ppoe
S_LD_FLAGS=
S_LIB_DYNLOAD2= -lF77 -lI77 -lU77 -lisam -lelf -lmld -lm -lbsd -lsun -lc
S_LIB_STATIC= -lF77 -lI77 -lU77 -lisam -lelf -lmld -lm -lbsd -lsun -lc
S_PAGER=less
S_PATH=/byron/splus/3.3/sgi/splus/.Functions:/byron/splus/3.3/sgi/s
tat/.Functions:/byron/splus/3.3/sgi/s/.Functions:/byron/splus/3.3/s
gi/s/.Datasets:/byron/splus/3.3/sgi/stat/.Datasets:/byron/splus/3.3
/sgi/splus/.Datasets
S_POSTSCRIPT_PRINT_COMMAND=lp
S_PRINTGRAPH_METHOD=postscript
S_PRINT_ORIENTATION=landscape
S_SCRIPT_SCCS_MAJOR=1
S_SCRIPT_SCCS_MINOR=73
S_SHELL=/bin/csh
S_WORK=.Data:/byron/users/burns/.Data
TERM=vt100
USER=burns
XAPPLRESDIR=/usr/lib/X11/app-defaults
XFILESEARCHPATH=/byron/splus/3.3/sgi/splus/lib/X11/app-defaults/%N:/
usr/lib/X11/%T/%N
```

Most of the S specific variables start with `S_`. They include variables to control printing, loading object code and so on. The `S_WORK` and `S_PATH` variables provide what is to be on the search list as S starts up. The first thing in `S_WORK` that exists is used as the working database, and everything in `S_PATH` is then

put on the search list.

A variable that doesn't appear here, but which can be set is `S_FIRST`. This should contain S commands; the commands are executed when S starts, and the `.First` function (if it exists) is not executed.

```
% setenv S_FIRST 'options(error=NULL); cat("error off\n")'
% Splus
S-PLUS : Copyright (c) 1988, 1995 MathSoft, Inc.
S : Copyright AT&T.
Version 3.3 Release 1 for Silicon Graphics Iris, IRIX 5.2 : 1995
Working data will be in .Data
error off
>
```

A variable that can be useful in special situations is `S_SILENT_STARTUP`. This suppresses the copyright printing.

```
% setenv S_SILENT_STARTUP 1
% Splus
error off
>
```

`setenv` is the C shell command to set an environment variable. The Bourne Shell has a different mechanism. There are "shell" variables as well as environment variables. The distinction is that the children of a process inherit the environment variables but not the shell variables.

7.2 Using Unix

Generally the first thing you do each time you meet up with Unix is to type a password. Always maintain a secure password. Here are some rules for picking a password:

• Do not use a name or a word as a password. Hackers can automatically try each item in a list as a guess for your password—a dictionary or a list of users is a likely choice. In particular, never have the password the same as the login name.

• Do not use an identification number or birthday as a password—a hacker could discover these and try them.

• Passwords should be easy to type (so it isn't easy to observe the password as you type it in), but should contain some non-alphabetic characters.

• They should also not be too short. There are only about half a million "words" that contain 4 lower-case letters, but on the order of 10^{10} that contain 6

characters from both lower- and upper-case letters, the digits and a few symbols. Half a million is not too many for some hackers to try, 10^{10} is.

Combining two words in some way, or using an acronym are two common ways of making a good password. I (you'll be shocked to learn) generally create an acronym out of a line of poetry. For example, *the slow smokeless burning of decay* [24] becomes "ts2bod". Once you have a good password, it is important to not let it be compromised, and to change it if you suspect it could have been. Never put your password in an email message or similar place—a hacker can search for occurrences of "password".

When you are at a Unix prompt, you are really conversing with a *shell*. A shell is the outer clothing of Unix and may be changed. The most common shell for interactive use is the C shell, and the most common shell for programming is the Bourne shell. Alternatives include the Korn shell and the Bourne-again shell. Most things are the same in all of the shells, but there are differences in such things as history mechanisms and flow control. The reason that you can't use an S command like

```
> attach("~/.Data") # this won't work
```

is that the tilde construct is a C shell device, but `attach` uses the Bourne shell to look for the file.

Two universal abbreviations concerning path names is the double dot (`..`) which means the directory directly above the current directory, and the dot (`.`) which means the current directory. For example to copy a bunch of files to the current directory, you could give a Unix command like:

```
% cp ../other_dir/*.c .
```

The C shell has a history mechanism that can be very handy. Some amazing things can be done with it, but I'll just give the most basic commands. `!!` repeats the last command. `!`*number* repeats the command with that number, for example, `!21` repeats command number 21. `!$` gets the last "word" of the last command, and `!*` gets all but the first word of the last command. So you can do something like:

```
% ls jj*
% rm !$
```

to first make sure that there is nothing untoward in the expansion, and then make sure that you get the exact same thing in the second command (rule out typos).

I stated earlier that Unix files are abstracted to include more than our usual sense of files. Three important "files" are standard-in, standard-out

and standard-error. Standard-in is where most Unix programs expect their input to come from. Standard-out is where most programs put their output, and standard-error is where error messages usually appear. These can be changed or *redirected* to actual files, for example

```
% S < commands.q
```

makes S look in `commands.q` for its input, so standard-in is redirected. The command

```
% S < commands.q > s.out
```

is almost like using the S `BATCH` utility except that `BATCH` takes care of some details.

A Unix *pipe* is related to redirection—the standard-out from one process is used as the standard-in of another. Here is an example:

```
% setenv S_FIRST 'invisible()'
% setenv S_SILENT_STARTUP 1
% set jj=7
% echo $jj "+" $jj "/3" | Splus
[1] 9.333333
%
```

A more practical instance of this is shown on page 191 in the discussion of `CHAPTER` and `data.dump`.

Commands in Unix may be put into the *background*, meaning that the process continues to compute but you get the prompt back so that you can perform other tasks. The command

```
% S < c.in > c.out
```

is in the foreground so you need to wait until S has done all of the commands in `c.in` before the prompt will come back. In contrast if you type:

```
% Splus < c.in > c.out &
[1] 21806
% jobs
[1]  + Running               Splus < c.in > c.out
```

then you will get the prompt back right away. The `&` character puts processes into the background.

Unix recognizes three types of quotes. Both double quote and single quote are used to protect characters from being interpreted—read some Unix documentation to get the difference between `"` and `'`. The backquote is used to substitute in the result of a command. For example

```
% Splus SHOME
/byron/splus/3.3/sgi
```

returns the directory that is the home of S, so the command:

```
% ls ‘Splus SHOME‘
Version            local/
adm/               module/
bin/               newfun/
cmd/               s/
doc/               splus/
include/           stat/
library/
```

lists the contents of that directory.

Unix allows you to give an *alias* to a command. A trivial example is that someone comfortable with DOS can alias `dir` to be `ls`. A less trivial example is:

```
% alias pgb 'ps -aux | grep burns'
```

This creates the command `pgb` that gives a list of all of the processes that belong to my login name (more correctly all the processes whose listing from `ps` contains "burns"). Another non-trivial alias is the following:

```
% alias pd 'pushd \!*'
```

This (in the C shell) is as if you typed "pushd" when you type "pd", including whatever arguments there are, if any.

When Unix looks for a command, it uses the environment variable `PATH` to decide where to look for the command. This is precisely analogous to the search list in S. The `PATH` is a sequence of directories separated by colons. If there are no characters between two colons, that spot represents the current directory.

You can create your own Unix commands (called *scripts*) merely by putting the command(s) into a file and making that file executable. Another way of getting my `pgb` would be:

```
% echo "ps -aux | grep burns" > pgb
% chmod +x pgb
```

It is customary to put your scripts into a directory called `bin` directly under your home directory.

7.3 Vocabulary

In each case below, I give only a hint of the full functionality. You can look at the "man page" for the command with the `man` command. For example

```
% man cat
% man man
% man csh
```

will give you help on the `cat` command, the `man` command and the C shell (including its commands), respectively.

The `cat` command has several uses, but the most common is to print out the entire contents of a file. If you want to look at only the top or the bottom of the file, then use `head` or `tail`. An especially powerful feature of `tail` is the `f` flag which continuously prints lines as they are added to the end of the file until you interrupt it (control-c). My most common use of this is:

```
% S BATCH b.in b.out
% tail -f b.out
```

The `grep` command is used to search for particular strings in lines of files. It uses regular expressions to do the matching.

`ps` lists the processes that are current on the machine. One common set of flags to use is `ps -aux` with one flavor of Unix, which is basically equivalent to `ps -ef` in the other major flavor.

One reason to list the processes is to kill off one of them. The `kill` command takes process id numbers and attempts to kill that process (and its children). I often want to kill an S batch job. To do this, I use my `pgb` command to list the processes that I own, I then look for the main process of the S job and note its id number. So a typical sequence of commands might be:

```
% S BATCH b.in b.out
% tail -f b.out

# (output from what S is doing, control-c to get prompt)

% ps -ef | grep burns
   burns 21741 21740  0 19:15:40 pts/1    0:00 -csh
   burns 21806 21741 80 19:37:13 pts/1    2:39 /byron/splus/3.
   burns 21812 21741  7 19:40:57 pts/1    0:00 ps -ef
   burns 21810 21806  0 19:37:13 pts/1    0:00 /byron/splus/3.
% kill -9 21806
% !ps
ps -ef | grep burns
```

```
burns 21741 21740  0 19:15:40 pts/1    0:00 -csh
burns 21817 21741  0 19:42:18 pts/1    0:00 grep burns
burns 21816 21741  4 19:42:18 pts/1    0:00 ps -ef
```

The `9` flag on `kill` is needed because S batch jobs don't die easily, so extra force is necessary.

Each file in Unix has a set of permissions attached to it. There can be permission to read, to write, and to execute. Such permissions can be given to the owner of the file, the owner's group, and to all users. The `chmod` command is used to change permissions. `chmod` understands the letters `r`, `w` and `x` with either a plus or minus in front. For example, if you want to block the S auditing process, you can make the `.Audit` file in the working database unwriteable with a command like:

```
% chmod -w .Data/.Audit
```

To make it writeable again, you would do:

```
% chmod +w .Data/.Audit
```

To see what the permissions are on a file, you can use the `l` flag of `ls`:

```
% ls -l .Data/.Audit
```

`chmod` also will take a three digit octal representing the permissions. The digit positions refer to the user, the group and everyone; the numbers are the sum of the permissions where 4 is read, 2 is write and 1 is execute.

The permissions of files as they are created is controlled by `umask`.

The `diff` command is used to see what is different between two files. The `diffsccs` function (page 51) is an example of its use.

Links are a way of saving disk-space (and a means of abstraction). A link essentially just gives more than one name to a file. On several platforms S-PLUS links help files that are for more than one function, rather than having a separate help file for each. The `ln` command creates links.

The `at` command allows you to start processes at an arbitrary time. You might for instance want to start an S batch job two hours after you leave when you presume that everyone will be off the system. The `crontab` command is used to start jobs on a regular schedule, for example every weekday at five o'clock.

The `make` command is extraordinary software. Here's the idea: you have blocks which depend on other blocks, which may depend on yet more blocks, etc.

If one or more of the blocks change, you want everything updated that needs to be (but unnecessary work should be avoided). `make` provides such functionality. The magic to be performed is stated in a file named `Makefile`.

DANGER. It is important to know that tabs are an important character in Makefiles. If you are using `vi`, you can see the actual characters in a file with the command `:se list`, and `:se nolist` turns it off.

One example of when `make` is useful is if you have some C or Fortran code that depends on a bunch of libraries and you want to dyn.load it into S. `make` can be used to give the commands that creates the object file that can be loaded into S. Thus making it trivial to recreate the object file after you have edited your code. This is a traditional use of `make`, but its general principle is expansive enough that it can be used for a variety of purposes. *the owls were bearing the farm away* [25]

The `find` command does more than you can imagine, but a simple use of it is to find files that you don't know the location of. The command

```
% find . -name sccs.q -print
```

looks for files named "sccs.q" in the current directory and all its subdirectories and then prints the location of any that it finds. A common mistake is to forget to give the directory from which to start. If you want to look for files that match a certain pattern, then you can give `find` a wildcard pattern (as opposed to a regular expression) but you must put quotes around it:

```
find . -name 'sccs*' -print
```

Functionality for finding out about different versions of a Unix command in the current environment is performed with `which` and `whereis`.

```
% which ps
/bin/ps
% whereis ps
ps: /bin/ps /usr/bin/ps /sbin/ps
/usr/share/catman/u_man/cat1/ps.z
```

The `which` command states which version of the command will be used, while `whereis` gives the locations where the command (or its documentation) is found on the current path.

The `stty` command is used to control terminal commands. To see the more important (in someone's eyes) settings, just type the command:

```
% stty
speed 9600 baud; -parity hupcl clocal
line = 1; intr = ^C; old-swtch = ^@; dsusp = ^@;
brkint -inpck icrnl onlcr tab3
echo echoe echok echoke
```

Add a `-a` flag to see all of the settings. A common setting to change is the key that is used to erase characters. The command is

```
% stty erase
```

where after the "erase" there is a space, and then you hit the key that you want to use.

Some versions of `stty` allow you to change the number of rows and columns for the display so that programs like editors will work properly.

The `test` command is quite useful in Unix programming. The following S function uses `test` to give information on file names.

```
"filetest"<-
function(filename, directory = F, read = F, write = F,
        execute = F, size = F)
{
        fbe <- unix(paste("test -f", filename, "-o -d",
                filename), out = F) == 0
        extras <- c(directory, read, write, execute,
                size)
        ans <- c(exists = fbe, directory = NA, read =
                NA, write = NA, execute = NA, size =
                NA)
        if(!fbe || !any(extras))
                return(ans)
        cmds <- paste(c("test -d", "test -r",
                "test -w", "test -x", "test -s"),
                filename)
        for(i in seq(along = extras)) {
                if(!extras[i])
                        next
                ans[i + 1] <- unix(cmds[i], out = F) ==
                        0
        }
        ans
}
```

Here are some examples of its use:

```
> filetest("polygamma.q")
 exists directory read write execute size
      T        NA   NA    NA      NA   NA
> filetest("SCCS", dir=T, write=T)
 exists directory read write execute size
      T         T   NA     T      NA   NA
> filetest("not_there", size=T, dir=T)
 exists directory read write execute size
      F        NA   NA    NA      NA   NA
```

It is used in portoptgen on page 348.

Although the use of window systems reduces the need, the pushd and popd commands in the C shell are still handy. These two commands implement the notion of a directory stack. pushd adds to the stack (among other things), and popd removes the top directory from the stack. Suppose we start in our home directory and give the command:

```
% pushd proj1
~/proj1 ~
```

We are now in the proj1 directory and home is the second directory in the stack. We can add to the stack by giving a new directory to pushd:

```
% pushd ~/proj2
~/proj2 ~/proj1 ~
% pushd
~/proj1 ~/proj2 ~
```

pushd with no arguments switches the top two directories in the stack. If we do:

```
% pushd +2
~ ~/proj1 ~/proj2
```

then the third directory is brought to the top (and the stack is treated circularly). The "2" can be generalized to higher numbers, by the way.

```
% pushd
~/proj1 ~ ~/proj2
% popd
~ ~/proj2
% dirs
~ ~/proj2
```

popd removes the top directory, and dirs displays the directory stack.

The `shar` command is a way of bundling a number of files into one file. Much of the S software on statlib is in shar files. A shar file contains the files to be created and Unix (Bourne shell) instructions on what to name the files and where the boundaries of the files are. The top of a shar file generally states how to unpack it. You should be in the directory where you want the files to live, and give a command like:

```
% /bin/sh sharfile
```

to unpack the file `sharfile`.

`awk` and `sed` are used to manipulate text, such as making substitutions. These are both good programs, but if you don't know them, I suggest that you learn Perl instead. Perl is a language that encompasses `awk`, `sed` and Unix scripts. If you know both Perl and S, you can rule the world.

Your last wish should always be for more wishes. The `apropos` command is used to try to find documentation on commands that relate to a certain subject. You might type:

```
% apropos tree
ftw, nftw (3C)          - walk a file tree
tlink (1)               - clone a file tree using symbolic links
tsearch, tfind, tdelete, twalk (3C) - manage binary search trees
```

and you will get back a brief description of a number of commands, most of which are likely to have nothing to do with your intention; or you might get nothing back.

7.4 Things to Do

Change your password if it is poor or known to others.

Check the permissions on your files and see if they are how you want them.

Read the man page of interesting commands.

7.5 Further Reading

Unix Power Tools by Peek, O'Reilly and Loukides is a very good book for learning Unix. It presumes a little competence with Unix but goes a long way toward teaching the whole of it. Unless you are a Unix expert, the answer to any Unix question that you have is likely to be in this book. I'll admit that it isn't always easy to go directly to that answer, but the journey that you take is often as useful as getting the answer to your original question.

7.6 Quotations

[24]Robert Frost "The Wood-Pile"
[25]Dylan Thomas "Fern Hill"

Chapter 8

C for S Programmers

C is a natural partner for S. If you know C but not S, then the next section can help you to learn S.

8.1 Comparison of S and C

The big difference between C and S is that C is compiled while S is interpreted. Thus C is much faster at calculations than S is—changing strategic portions of S code into C can often cause execution time to drop from minutes to seconds. *This science is then like gathering flowers of the weed* [26]

S code and C code look very similar (this is not accidental). However, there are some striking differences.

DANGER. The most bothersome discrepancy is that indexing in C is zero-based while indexing in S is one-based. So saying `x[1]` gets the first element in S but the second element in C. Although this difference is a fertile source of bugs, I think each language has done the right thing—zero-base is good for machines where C has its strength, and one-base is good for humans.

C requires that everything has its type declared. C also requires the user to take care of allocating and freeing memory, while this is automatic in S. Nothing is vectorized in C. C demands that each command end with a semicolon (there is no penalty for a C programmer who puts such semicolons in S code).

C has a number of operators that S does not. The most important of these are the increment operators like `i++` and `i += n`. Another example is the ability of C to operate on bits.

Names in S and C have similar forms except for one important difference. In S, periods are the only non-alphanumeric character allowed in names. In C,

it is underscores that are the only non-alphanumeric character. Furthermore, the period is an operator in C, and the underscore is an operator in S. Like S, C distinguishes lower-case from upper-case letters. There is a limit to the length of a name in C, but the ANSI standard requires that that length be at least 31 characters.

There are two flavors of C, the original version is known as K&R C after Kernighan and Ritchie who wrote it and the book describing it. The other is ANSI C. The ANSI standard mainly extended and clarified the language, but the most visible change is in how arguments to functions are given. The old-style (K&R) contains just the argument names within the argument list, and the arguments are declared outside the list, as in:

```
double my_fun(x, n)
long n;
double *x;
```

The new-style (ANSI) contains the declarations within the argument list:

```
double my_fun( double *x, long n)
```

Being old-fashioned, most of my example C functions follow the old-style.

A feature of C that is entirely novel relative to S is *pointers*. A pointer is an object that contains the address of some value. In S you can only grab hold of the values, never the address. Pointers are what tend to cause the most confusion to novice C programmers. There really isn't all that much to them though, and there's no need to panic. If you have a pointer (which is the address of something), then you can "dereference" it (get the value at that address) with the `*` operator. (`*` also means multiplication as it does in S, but which is meant is distinguished by context.) A code fragment is:

```
double *xpointer, old, new;

old = *xpointer;
*xpointer = new;
```

Here both `old` and `new` are double-precision numbers and `xpointer` is a pointer to a double. The original value at the `xpointer` address is given to `old`, then the value of `new` is placed at that address.

You can also get the address of an object with the `&` operator (this is sort of the inverse of the dereferencing operator). An example that is subtly different from the last is:

```
double *xpointer, old, new;

old = *xpointer;
xpointer = &new;
```

C Operator	Task	S Equivalent
`[`	array subscript	`[`
`.`	structure member	`$`
`->`	structure pointer member	`$`
`&`	address of	*none*
`*`	dereference	*none*
`++`	increment	*none*
`--`	decrement	*none*
`*`	multiply	`*`
`/`	floating or integer divide	`/ %/%`
`%`	modulo	`%%`
`<< >>`	bit shift	*none*

Table 8.1: Some C operators and approximate S equivalents, if any.

Operator	C Meaning	S Meaning
`.`	structure member	character in names
`->`	structure pointer member	assignment to right
`_`	character in names	assignment
`&`	address of *or* bitwise and	vectorized and
`^`	bitwise exclusive or	exponentiation
`|`	bitwise or	vectorized or
`?`	ifelse trinary operator	help
`:`	ifelse trinary operator	sequence
`=`	assignment	argument matching
`#`	introduce preprocessor command	introduce comment

Table 8.2: Operators with different meanings in C than in S

This still has the same value for `old` and `xpointer` still points to the same value. However, in the first example the value at one address is changed, while in the second it is addresses that change. The second example would be unusual—the first example would more often be correct. The `&` operator is most commonly used in function calls where the function is expecting a pointer, but you have a value.

Table 8.1 lists some of the operators in C, and table 8.2 lists some operators that have the potential for confusion between C and S. Exponentiation in C is done with the `pow` function in `math.h`. The modulo operator `%` in C works only on integers, but `%%` in S works with floating point numbers also.

Flow control is almost identical in the two languages. In particular, braces are used precisely the same to group multiple statements into a single statement. A notable exception is that the control mechanisms for `for` loops are distinctly different. The format and to some extent the functionality of `switch` differs in C and S. C has a do-while loop that is absent in S (`repeat` can be used to do

the same thing). C does not have a `repeat`, but this is easily replaced with `while(1)`. One more difference is that `next` in S is `continue` in C (I favor S in this regard).

An example of a `for` loop in S is:

```
for(i in 1:10) { x[i] <- i }
```

while a similar `for` loop in C would be:

```
for(i=0; i < 10; i++) { x[i] = i; }
```

So there are three parts inside the parentheses of a C `for`. The first part is the initialization. The second part is a test, where the loop is exited the first time that the test is false (so it is possible that the body of the loop is never entered). The third part states what to do after each iteration. (Notice that the resulting vector in C will be different than the one in S.)

If you want more than one statement in either the first or third part, you can put commas between the statements. Here's an example:

```
for(i=0, ij=0; i < 10; i++, ij += n)
```

Another differeence is that the iteration variable in an S `for` is local to the loop. In C it is not local to the loop, and—as we've seen—there need not even be a unique variable.

An example of the format of a `switch` statement in C is:

```
switch(x) {
   case 'b': y += 3;  z = 6; break;
   case 'd':
   case 'e': y += 2; return(y);
   case 'h': z += 8;  /* fall through */
   case 'n': y += 9; break;
   default: y -= 8; break;
}
```

This is a bizarre example, but shows the rules. The switch variable can be either a character or integer—in this case it is character. The body of the `switch` is one expression which is divided by a number of case statements. Case b shows that one difference with S is that there can be any number of statements for a C case, while S demands a single expression for a case (multiple statements need to be surrounded by braces). Case d is empty which means that it is identical to the next non-empty case—this is like S except that S doesn't have this feature for numeric variables. Case h shows how the C `switch` is fundamentally different than that of S. Since there is no `break` or `return` in case h, it goes into case n also. Hence the order of the cases in a C `switch` can make a difference. It is typical that each case end with `break`.

C has a unique way of specifying comments—they begin with `/*` and end with `*/`. You can have anything in a comment except `*/` (that is, comments don't nest). A definite advantage to this form of comment delimiting is that it is easy to comment out a section of code. There is no equivalent in S to do this. However, there is a trick—you can put the code inside an `if` that is always FALSE.

The `#` symbol is used in preprocessor commands. The C preprocessor is a powerful feature of C that inserts include files, makes platform dependent versions, and many more things—there is no equivalent feature in S. The preprocessor performs its commands to change the source file before the compiler sees it. For example, the first line of a C source file might be:

```
#include <S.h>
```

This inserts the contents of the `S.h` file into the source file. The angle brackets mean that the preprocessor is free to look in a number of locations for the file. The `S.h` file is the header file that comes with S—it contains a number of definitions that can be used in C code. Although code using `S.h` is often called from S, it need not be.

The next step is to compile the source code. You can do this directly, but if you have S-PLUS, then you can use the utility created for the purpose. An example is:

```
Splus COMPILE myfun.c
```

The `COMPILE` utility is meant to perform an appropriate compilation on whatever platform you are using. And often does.

8.2 Loading Code into S

The C functions that can be called from S are of a restricted class. All of the arguments must be pointers (which makes sense if you consider that S vectors translate to C arrays). The function is typically declared as void since return values are thrown away. The arguments may be `*long` (integer or logical in S), `*double` (double in S), `*float` (single in S), or `**char` (character in S). There is an exception—see the discussion of the `pointers` argument to `.C` on page 177.

Note that structures are not allowed as arguments. If you learn C especially for S, there is the tendency to avoid the use of structures since they may not be passed in. Fight this impulse—structures in C are the equivalent of lists in S, and are very useful.

If you have existing C code that you want to use in S, it should not be difficult to write a wrapper function in C so that it obeys the requirements that S has. There is a limit on the number of arguments allowed, but this is large enough that it should seldom be of concern.

The S executable (the file that contains the stuff that makes S actually go) contains a *symbol table*. First off, a *symbol* is a translation of a routine name based on platform specific rules. Generally a symbol is the routine name with underscores before and/or after it. C symbols are different than Fortran symbols. In S-PLUS you can do:

```
symbol.C("fjj")
symbol.For("fjj")
```

to see how your system does it. A symbol table associates each symbol that it knows to an address.

When you load code into S, you are extending or changing the symbol table. The two main divisions of loading code are static loading and dynamic loading. Static loading means that you combine the executable as it stands with additional code and get a new (larger) executable. Dynamic loading is when the executable is already executing and you add the code to this instance of execution—the executable is not changed, merely the S session.

Static loading is done with the `LOAD` utility. One good use of static loading is if everyone at a site needs access to a specific set of routines that are thoroughly debugged. These routines can be statically loaded into S, and the script with which users start S can point to this new executable. The other good use for static loading is when dynamic loading is unavailable or doesn't work for the code at hand. The disadvantage of static loading is that a new executable is created—the size of which is measured in megabytes. An explanation of how `LOAD` works is on page 191.

In S-PLUS there are three functions that do dynamic loading: `dyn.load`, `dyn.load2` and `dyn.load.shared`. Not all of these will be available on a particular platform.

`dyn.load` is the simplest of the three. It takes a character string which is the name of an object file; all of the symbols in the file need to be defined either in the file or in the S session. (Actually `dyn.load` will take a vector of file names—the order of the files can matter because it is like a series of individual calls to `dyn.load`.)

`dyn.load2` and `dyn.load.shared` are different ways to solve the same problem. These functions allow symbols that are defined in neither the source file nor the S session to be pulled from libraries. Unlike `dyn.load`, you can not make two different calls to `dyn.load2` to load related symbols—all the symbols must be put into the S symbol table with a single call to `dyn.load2`.

Troubles with dynamic loading are fairly common, and often hard to solve. The more complex the situation with symbols, the more likely problems will arise. It is not likely that dynamically loading a self-contained file will fail. If using `dyn.load2` or `dyn.load.shared` doesn't work, you can try using the Unix command `ld` to create a file that contains all of the necessary symbols so that using `dyn.load` is a possibility. Picturing what is happening with the symbols

object	S	C	Fortran	Perl
single value	vector	scalar	scalar	scalar
ordered, 1 type	vector	array	vector	array, list
rectangle of 1 type	matrix		matrix	
hyperrectangle of 1 type	array		array	
collection, several types	list	structure		

Table 8.3: Terminology from various languages.

can be a substantial help in solving a loading problem. A tool that can help is the `NM` utility discussed on page 192.

One loading problem that can arise is when there are duplicate symbols. This generally happens when the same function is in two different places. However, there is the possibility that there are two different functions with the same name. C provides a very good way of keeping this from happening. All C functions except those that are used in a `.C` call in S can be declared static. This means that you can have a C function called `sort` and it will not conflict with any other "sort" as long as you declare it static in your C source file. If you declare as many functions as possible to be static, then there will only be a few symbols that can possibly conflict. To reduce the possibility that these remaining names will conflict with something, you can put a personal prefix or suffix on the name. Many of the in-built C routines in S begin with a capital S and an underscore. For instance, there could well be a C function called "S_sort". My personal convention for names that are going to be used in S is to put "_Sp" at the end; so my sort routine—if it were not static—would be called "sort_Sp".

8.3 Arrays

It is common for C programmers to think of matrices as pointers to pointers so that they can be subscripted like:

```
cmat[i][j]
```

However, an S matrix passed into C is merely a C array (i.e., a pointer). If you have code that uses pointers to pointers for matrices, then you will need a wrapper function in C to convert back and forth between the two conventions. Table 8.3 lists terminology from some languages for a few simple data structures. There is plenty of room to trip over the nomenclature between S and C. For example, a C programmer may well think that a "character vector" is a character string, while S programmers know that a "character vector" is a vector containing character strings.

If you are writing your own numerical code, then I advise treating matrices in C the same way that S does. You need to do some arithmetic to get a specific row and column, but it is easy to do. This is the way that Fortran handles

matrices also, so Fortran matrix code can be called from C when using this convention. It is simple to generalize to higher dimensional arrays.

Here is an example that computes quadratic forms $x'Qx$ where Q is an n by n matrix and x is a vector. The S function generalizes this to allow a number of quadratic forms to be evaluated in one call by passing in a matrix `x`, each of whose columns is an x in the quadratic form.

```
static double
quad_form(double *Q, double *x, long n)
{
        long i, j, ij;
        double ans = 0.0;

        for(i=0; i < n; i++) {
                for(j=0, ij = i * n; j < n; j++, ij++) {
                        ans = ans + x[i] * Q[ij] * x[j];
                }
        }
        return(ans);
}


void
quad_form_Sp(double *Q, double *x, long *xdim, double *ans)
{
        long i, ii, n;
        double quad_form(double*, double*, long);

        n = xdim[0];

        for(i=0, ii=0; i < xdim[1]; i++, ii += n) {
                ans[i] = quad_form(Q, x + ii, n);
        }
}
```

Stepping through the matrices is made efficient by incrementing the subscripts rather than always performing a multiplication. Here is the S function that calls this code.

```
"quad.form"<-
function(qmat, x)
{
        dq <- dim(qmat)
        dx <- dim(as.matrix(x))
        if(length(dq) != 2 || dq[1] != dq[2])
                stop("qmat must be a square matrix")
```

```
        if(dx[1] != dq[2])
                stop("qmat and x do not match")
        if(!is.loaded(symbol.C("quad_form_Sp")))
                poet.dyn.load("quad_form.o")
        .C("quad_form_Sp",
                as.double(qmat),
                as.double(x),
                as.integer(dx),
                double(dx[2]))[[4]]
}
```

This function saves a little time compared to the usual S operation by using one operation instead of two matrix multiplications. If you really want to evaluate a number of quadractic forms with the same kernel, then this will save a lot of time. If the kernel matrix is symmetric and large, then some more efficiency can be squeezed out by modifying the looping in `quad_form`.

8.4 C Code for S

One type of functionality that the S header provides is for handling missing values and special values. The C functions include `is_na`, `is_inf`, `na_set`, `na_set3` and `inf_set`. (Actually these are generally macros rather than functions, so you don't want to have side effects in the arguments.) The first two arguments to each of these is a pointer to the value of interest, and the storage mode of the object (back in S), the names to use for the modes are listed in table 8.4. `is_na` and `na_set3` use the symbols `Is_NA` and `Is_NaN` to distinguish between missing values and not-a-numbers. So to set the ith element ($i + 1$th in S) of `x` to be a not-a-number, you would say:

```
na_set3(x + i, DOUBLE, Is_NaN);
```

You can see some of the C functions that handle special values in the code for the `digamma` function on page 262. If you want to allow missing values in the arguments to your C function (or Fortran subroutine) that is called from S, then you need to use the `NAOK` argument to `.C`. The `specialsok` argument in S-PLUS allows infinite values. Neither of these arguments may be abbreviated.

C code that will be called by S and needs to allocate memory can use the `S_alloc` C function. This is similar to `malloc`, but does not have a corresponding `free` statement. The memory that is allocated by `S_alloc` is automatically freed when the S memory frame containing the C call is exited. It is freed even if there is an abnormal exit from the C code, so you are guaranteed that the space will be freed.

If your routine uses workspace, then it is better to allocate the workspace in C with `S_alloc` rather than passing it in through the call to `.C`. S will copy the

S storage mode	Name in C	C declaration
logical	LGL	long
integer	INT	long
single	REAL	float
double	DOUBLE	double
complex	COMPLEX	complex

Table 8.4: Modes within C Code.

object in the latter case. Use of S_alloc might look like:

```
long *iwork;
double *work;
work = (double *)S_alloc(n, sizeof(double));
iwork = (long *)S_alloc(2 * n, sizeof(long));
```

You can issue the equivalent of stop and warning statements in your C code called by S. These start with PROBLEM followed by a string, followed by arguments to be placed in the string (if any), followed by another magic phrase indicating if it is an error or a warning. An example of a warning is:

```
PROBLEM "bad, bad, i is %d, x is %g", i, x
     WARNING(NULL_ENTRY);
```

An example of an error statement is:

```
PROBLEM "really bad" RECOVER(NULL_ENTRY);
```

Note that there are no parentheses (they are in the definition of the PROBLEM macro and its mates), and you need not have a newline at the end of the string.

S-PLUS makes available routines so that you can use the S random number generator in your C code. Before you can generate any random numbers, you need the line:

```
seed_in((long *) NULL);
```

This makes the random seed available. After all random numbers have been generated, say:

```
seed_out((long *) NULL);
```

This updates the stored random seed, so that the same "random" numbers will not be generated repeatedly. The two functions that generate random numbers are unif_rand and norm_rand, both of which return a double.

The example below shows how to use the random functions. It produces a sequence of random Gaussian deviates that terminates the first time that one of the random numbers is greater than a specified value. It also introduces a couple more tricks for C programs in S—the `S_realloc` function and the `pointers` argument to `.C`.

```
#include <S.h>

void
rand_seq_Sp(pars, ipars, ans, alen)
long *ipars, *alen;
double *pars, **ans;
{
        long count=0, curlen, incr=ipars[0], safety=ipars[1];
        double threshold=pars[0], rmn=pars[1], rsd=pars[2];
        double this_rand;

        *ans = (double *)S_alloc(incr, sizeof(double));
        curlen = incr;

        seed_in((long *) NULL);

        while(1) {
                this_rand = norm_rand() * rsd + rmn;
                if(count >= curlen) {
                        S_realloc(*ans, curlen + incr, curlen,
                                sizeof(double));
                        curlen += incr;
                }
                (*ans)[count++] = this_rand;
                if(this_rand > threshold) break;
                if(count >= safety) {
                        PROBLEM "reached maximum length, exiting"
                                WARNING(NULL_ENTRY);
                        break;
                }
        }
        seed_out((long *) NULL);
        *alen = count;
}
```

The random part of this function is easy. Reasonably easy is the use of `S_realloc`—when the space allocated by `S_alloc` is no longer large enough, `S_realloc` allocates a new, larger space and takes care of transferring the old values to the new location. Its arguments are the pointer to the space, the new length, the old length, and the size of the elements.

Not so transparent is what is happening with `ans`. Generally `ans` would be a pointer to double, but here it is a pointer to a pointer to double. The reason is that the length of `ans` is not known in advance, so it uses the `pointers`

argument to the `.C` S function. Here is the S function that calls this code:

```
"fjjrand.seq"<-
function(threshold, mean = 0, sd = 1, increment = 1000,
        safety = 10000)
{
        if(!is.loaded(symbol.C("rand_seq_Sp")))
                poet.dyn.load("rand_seq.o")
        ans <- .C("rand_seq_Sp",
                parameters = as.double(c(threshold[1],
                        mean[1], sd[1])),
                as.integer(c(increment[1], safety[1])),
                sequence = double(0),
                pointers = c(F, F, T))[c("sequence",
                "parameters")]
        names(ans$parameters) <- c("threshold", "mean",
                "stand dev")
        ans$call <- match.call()
        ans
}
```

Notice that there is one less argument to `.C` (excluding the `pointers` argument) than there is in the C function. Each argument to `.C` that is a pointer corresponds to two arguments in C—the first C argument is a pointer to a pointer, and the second argument is a pointer to long that gives the length of the object.

DANGER. When using the `pointers` argument, don't forget to set the length of the variable-length objects in C.

In the call to `.C` the first element of `threshold` and several other variables are extracted when combining variables together for arguments to the C function. This ensures that a misspecified S argument will not completely mess up the C call. An alternative strategy is to check the length of each variable.

S-PLUS gives you routines that help you when calling Fortran from C. See the PORT optimization example on page 340.

A C function that is included with S is `call_S` which allows C code used by S to go back to S and evaluate an S function. I'm not going to give an example of its use. Although it does have its place, it is less useful than might be thought. (You still have the overhead of the S function call, so you can't gain speed by looping in C instead of S.) Examples of how to use `call_S` are in Becker, Chambers and Wilks (1988) and Spector (1994).

8.5 Debugging

If you are unlucky, one of the first things you will see when you use an S function that calls some of your C or Fortran code is something like:

```
System terminating: bad address
```

and you are thrown out of S and back into the operating system. The system terminating happens when S detects that something bad has happened, and it kills itself to prevent permanent damage in case of corruption. The system terminating gives an indication of what went wrong. The most common is "bad address" which is often an indication of running off the end of an array, or an argument mismatch. The second most common is "bus error" which is an indication of serious confusion. *I never heard that sound by day.* [27]

If you see a system terminating in an S session in which no local code is used and you are not masking any in-built functions, then there is a bug. You should report this to your provider of S. A system terminating can be caused from a corrupted version of an in-built function—and it need not be apparent what the problem is. Likewise, locally written code that is loaded into S can cause a problem that only later results in a system termination. Usually, though, the termination occurs immediately.

The most common problem with `.C` or `.Fortran` calls is that there is the wrong number of arguments or they are not declared properly. Each argument in a `.C` or `.Fortran` call needs to be either coerced to the correct storage mode or to have had its storage mode coerced directly before the call. If not, it is a bug.

Capricious Rule 1 *Your skin crawls when you see an argument to* `.C` *or* `.Fortran` *whose storage mode is not coerced.*

DANGER. Storage mode integer in S corresponds to `long` in C. On many machines a `long` is identical to an `int` so it is possible to declare S integers as `int` in C and have the program run. Don't do it. Some day when you try to make this go on some other machine where the `long-int` identity doesn't hold, there are likely to be a number of frustrating hours wasted trying to track down the problem. Remember that the more subtle and silly the mistake, the harder it is to find.

DANGER. A common way to have the number of arguments different in the `.C` call and the C function is to edit the C function and recompile, but forget to edit the S function.

On Unix the `lint` command looks for mistakes in C code, like mismatches of type and variable numbers of arguments. If you have S-PLUS and you are using `S.h`, then you can use a command like:

```
lint -I‘Splus SHOME‘/include myfun.c
```

This tells `lint` the appropriate place in which to look for the S header file.

A tried and true debugging technique is to insert print statements into the code. The C function `printf` is the most likely candidate to use to print messages. The first argument to `printf` is a character string which may contain excaped characters such as backslash-n, and it also typically contains locations for any subsequent arguments. The locations for the additional arguments are marked with a symbol that starts with `%` and is followed by a letter that indicates the type of object to be printed. Here's an example:

```
printf("i is %d  x sub i is %f\n", i, x[i]);
```

which might result in the output:

```
i is 0  x sub i is 4.9
i is 1  x sub i is 9.3
```

If you are inserting print statements because the function is hanging and you want to know how far it is getting, then you will need to flush the output. To save time, C puts output into a buffer and only really writes once the buffer is full. So if the buffer never fills, you won't see your output. The usual command to flush `printf` output is:

```
fflush(stdout);
```

(`stdout` is defined in `S.h`.)

When I'm in a disciplined mood, I make an initial debugging statement in the C code that prints a version number. The routine is: modify the C code, compile the C code, load the code into S, try out the code. Once I've done this several times and frustration is setting in, I sometimes forget either to compile the code or load the code so it appears that my last change had no effect when in fact the last change is not being used. Of course I need to remember to change the version number when I edit the code in order for this trick to be effective.

There are times when optimization during compilation—especially a high level of optimization—causes bugs to appear. My strategy is to use the standard level of optimization except on files that are primarily arithmetic; and I generally time the computations to see if the optimization really works.

8.6 Common C Mistakes

DANGER. The most common mistake is to leave out semicolons. This is seldom a serious mistake since the compiler will catch most instances. A more serious mistake can be putting a semicolon where it doesn't belong. In:

```
if(i); j = 0;
```

the code will compile without a problem, but will not do what was probably intended. The code as it stands will test the value of `i`, then do nothing, and finally always set `j` to 0. If the intention is to set `j` to 0 if `i` is not, then that first semicolon has got to go.

DANGER. It is easy to get the indexing wrong on an array. It is natural to translate the S expression:

```
for(i in 1:n)
```

into C as:

```
for(i=1; i <= n; i++)  /* WRONG */
```

Once you are past that, you might try the more subtle:

```
for(i=0; i <= n; i++)  /* STILL WRONG */
```

You really want:

```
for(i=0; i < n; i++)  /* RIGHT */
```

DANGER. Be careful when performing division with integers—the result is an integer, not a floating point number. Integer division doesn't commute like floating point division does. Consider the code fragment:

```
long n, ans1, ans2;
ans1 = n * (n + 1) / 2L;
ans2 = n / 2L * (n + 1);
```

`ans2` is not the same as `ans1` unless `n` is even. `ans1` is probably what was intended. An extra pair of parentheses to guarantee the computation order would

be a good idea.

DANGER. If you have a statement containing:

```
xmat[i,j]  /* WRONG */
```

then it is wrong (but it will compile). The comma operator evaluates each statement but has the value of the right-most statement. So the above is equivalent to `xmat[j]`.

DANGER. Another common mistake is to use `=` where `==` is desired. The statement:

```
if(j=4) j++;  /* WRONG */
```

will once again compile, but not do the right thing. After this statement `j` will always be 5 no matter what it is before the statement. C, like S, returns a value from assignment. Such a mistake is not possible in version 3 of S, since it is a syntax error. (You will have the opportunity for such bugs in version 4, however.)

DANGER. Do not use `print` instead of `printf`. The code will compile, but have severe problems in the execution.

8.7 Fortran

I give Fortran less emphasis partly from personal taste (though I admit that newer versions of Fortran are definite improvements on older ones), and partly because there are more complications when interfacing Fortran to S than when interfacing C. However Fortran has the one big advantage that lots of algorithms are programmed in it. It is usually faster and safer to have a little hassle interfacing existing Fortran to S than to write new C code.

A squabble between Fortran and S is that S doesn't handle Fortran input/output. If you have the source code for the Fortran, then the easiest thing to do is to comment out all of the input/output statements. This step may be as simple as just commenting out lines, or it may involve extensive changes to

have data passed in through arguments rather than read in. If you don't have the source code, then you may need to ensure that S has a number of symbols loaded that it generally doesn't have by writing C functions and loading them. For example, one function might look like:

```
#include <S.h>

s_wsFe()
{
    fprintf(stderr, "Trying to use Fortran I/O\n");
}
```

S does come with some Fortran subroutines that produce printed output in S format. These are often useful for debugging purposes. The names of the routines are `dblepr` for double-precision data, `intpr` for integers (and logicals), and `realpr` for single-precision numbers. Each of these allows a message also. An example call is:

```
 call dblepr('x is', -1, x, nx)
```

The first argument is a message. The second argument is the length of the message, though most compilers are nice enough to do the counting for you if you give it a negative 1. The third argument is the name of the variable that you want to print, and the fourth argument is the number of elements to be printed. You do not need to print all of the values in the variable—it will print the first few values if you choose.

DANGER. At least in some versions of S, the name of the subroutine that you are calling in the `.Fortran` call needs to be in all lower-case. Fortran doesn't care about the difference, but S needs to have it that way.

It is a lot easier to get name conflicts in Fortran names because the name space is so much more limited (you only have 6 characters to work with), and all of the names are visible. You merely have to do the best that you can to try to avoid collisions. In particular, do not use "sort", S already has three or four.

I often put:

```
implicit none
```

statements into Fortran routines (when the compiler accepts it) and then declare all objects explicitly. This can uncover problems that are not otherwise apparent.

S-PLUS has facilities for returning error messages from your Fortran code called from S. You can learn about the S functions in the `xerror.summary` help. You have a choice of two Fortran subroutines—`xerror` and `xerrwv`.

Here is exciting code that does important stuff.

```
 subroutine xerrsp(x, n)
     implicit none
     integer n
     double precision x(n)
     integer i, m99l, m2l
     character*100 m99, m2
     real xe

     call dblepr('x values are', -1, x, n)

     m99 = "x value is negative"
     m99l = len(m99)
     m2 = "r1 is more than 2"
     m2l = len(m2)

     do i=1, n
        if(x(i) .lt. 0.0) then
           call xerror(m99, m99l, 99, 1)
        endif
        if(x(i) .gt. 2.0) then
           xe = x(i)
           call xerrwv(m2, m2l, 2, 1, 0, 0, 0, 1, xe, 0.0)
        endif
     end do
     return
     end
```

The first four arguments to `xerror` and `xerrwv` are the same: a message, the length of the message, an error number, and an error severity. The `xerror` subroutine only has these arguments, but `xerrwv` has six more to allow you to print up to two intergers and two real values. The fifth argument is the number of integers to print—this must be 0,1 or 2. The next two arguments are the integers. The eighth argument is the number of reals to print, again this must be 0, 1 or 2. The remaining arguments are the reals to print. You will notice in the code above that there is a variable to hold the real value to be printed by `xerrwv` so that a `real` is passed in rather than a `double precision`. Below is the S function that calls this code.

```
"fjjxerrsp"<-
function(x)
{
        if(!is.loaded(symbol.For("xerrsp")))
                poet.dyn.load("xerrsp.o")
```

```
        xerror.clear()
        ans <- .Fortran("xerrsp",
                as.double(x),
                as.integer(length(x)))[[1]]
        if(options()$warn >= 0)
                xerror.summary()
        ans
}
```

An example of what happens is:

```
> fjjxerrsp(-1:3)
x values are
[1] -1  0  1  2  3

recoverable error in...
x value is negative

 error number =     99

recoverable error in...
r1 is more than 2

      in message above, r1 = 3
 error number =      2
           error message summary
message start                                  nerr     level     count
x value is negative                              99         1         1
r1 is more than 2                                 2         1         1
 other errors not individually tabulated =          0

[1] -1  0  1  2  3
```

If you are using Fortran code, then there is going to be an interface between Fortran and C somewhere since S is written in C. You can push the interface down into C code that you write. You can call a Fortran routine in C by using the symbol name rather than just the routine name—C really only cares about the address of the function. The `S.h` include file contains macros to get this right. The other thing to note is that all of the arguments to the Fortran routine must be pointers. (Fortran only uses pointers, thus it doesn't need the concept of "pointer".) The PORT optimization on page 340 is an example of how to call Fortran from C.

8.8 Things to Do

Find portions of S functions that would benefit by being coded in C, and do it. What is the difference in execution speed? How much flexibility are you

sacrificing? Can you change it so that you lose less flexibility while retaining most of the speed improvement?

8.9 Further Reading

The fountainhead of C is Kernighan and Ritchie (1988) *The C Programming Language, 2nd Edition*. This is a terse book—perhaps overly so—but carefully written. A quite interesting book is Libes (1993) *Obfuscated C and Other Mysteries*.

There is a raft of other books on C. One that I happen to have is Darnell and Margolis (1991) *C: A Software Engineering Approach*. I make no claims that this is the best book on C, but it is reasonably good, and it definitely is not the worst (I think I have that one also).

8.10 Quotations

[26]Louis Zukofsky "It's Hard to See but Think of a Sea"

[27]Robert Francis "By Night"

Chapter 9

S for S Programmers

This chapter covers topics about S that are not explained in the previous chapters. Parts are of less than general interest.

9.1 Databases

When you start S, it looks for a `.Data` directory to use as the working database. In general you can think of this as merely holding files that each contain one of your S objects, where the name of the file and the name of the S object are the same. However, there are other entities living in `.Data` directories that may be of interest. Many of these are alluded to in the section below on utilities.

The help files live in the `.Help` subdirectory of the `.Data`. Each object with help has a corresponding file in `.Help`. There is also the possibility of a `.Cat.Help` subdirectory—see page 193.

The `.Audit` file is the location where the audit trail of S commands is kept. It is the `.Audit` in the working database as a session starts that is used. Once started, the same audit file is used throughout a session, no matter what the working database is.

For names that are troublesome for the operating system, the actual file will be called something like `__3` and there is a file called `___nonfile` that is a dictionary to translate between the S object name and the file name.

DANGER. At least in some versions of S-PLUS, there are occasions when a file is written with a name like `__34352` that is not indexed in the `___nonfile` file. I'm not sure of the circumstances under which these get created, I'm guessing that it is perhaps when there is some sort of interruption. Sometimes these objects can be quite large. If you see a suspicious file like this, you can check to see that it is not listed in `___nonfile` and remove it.

In recent S-PLUS versions the databases of in-built objects contain one file that holds all of the objects (called `__BIG`) and a file to index the objects (called `__BIGIN`). This reduces the amount of memory that the database uses.

9.2 Utilities

When you give the command to start S, you are really starting a Unix script that does a number of things, one of which is to start the executable (the actual language), which is called `Sqpe`. One of the things that this script allows is an extra argument which is the name of a *utility*. A utility is an executable that lives in the `cmd` subdirectory of `SHOME`—the utility is executed instead of the S language.

The `BATCH` utility is probably the most commonly used. It is invoked like

```
% S BATCH b.in b.out
```

where `b.in` is a file of S commands and `b.out` is where the results go. If `b.out` already exists, the previous contents will be deleted. Note that what I am writing as `S` here should be replaced with the command you use to start S.

`BATCH` jobs are good for running long computations. If you often run several `BATCH` jobs on a variety of machines, then starting your input files with

```
!hostname
date()
```

can help you keep all of the jobs straight. For instance, if you want to kill one of the jobs, you can tell which process to kill even if you have more than one batch job running on the machine in question. I like to put `date()` at the end of the file also.

It is frustrating to check a batch job after a weekend and find that it didn't do anything because of a typo or other minor error. A partial guard against this is to perform an S command like:

```
parse(file="b.in")
```

where `b.in` is the name of the input file for the batch job. The result will either be an expression or an error. It will be syntax errors that are found, such as missing parentheses. If there is no error, you still don't know for sure that it will work—you may need to attach a directory, or may have misspelled a name—but I often find this check worth the effort. See also my fix for `browser.default` on page 138.

The command

```
% S SHOME
```

simply returns the path of where S lives. Though simple, it is very expedient at times. You can get a sense of all of the utilities available to you by doing:

```
% ls `S SHOME`/cmd
```

Not all of the files there are utilities, but many are.

The `REPORT` utility is probably underutilized. This takes a file that is generally in a typesetting language like TEX or troff that contains some specially marked sections of S commands. The result is a different file with the output resulting from the S commands replacing those commands.

Here is a trivial example. Our (naive) input file (called `rep.in`) looks like:

```
% cat rep.in
Harold weighs { jjwt["harold"] } kilograms, but
Dorothy weighs {jjwt["dor"]}.
```

The S commands are surrouned by braces. The utility is fired up and then we view the resulting file:

```
% Splus REPORT rep.in rep.out1
REPORT will run in batch: input from file rep.in,
output on file rep.out1.
% cat rep.out1
S-PLUS : Copyright (c) 1988, 1995 MathSoft, Inc.
S : Copyright AT&T.
Version 3.3 Release 1 for Silicon Graphics Iris,
IRIX 5.2 : 1995
Working data will be in .Data
attaching dotFunctions
Harold weighs harold
15 kilograms, but
Dorothy weighs dorothy
14.
```

We get two problems in one go. The first problem is that all of the printing when S-PLUS starts gets put into the output file. The second problem is that we are getting the names of the quantities that we want as well as the numbers themselves. What gets put into the output file is everything that S says in response to the commands, so we need to be a little more careful about our commands. The `cat` function comes to the rescue:

```
% cat rep.in2
Harold weighs { cat(jjwt["harold"], "\n", sep="") } kilograms, but
Dorothy weighs {cat(jjwt["dor"], "\n", sep="")}.
```

Note that the backslash-n in the `cat` calls is of importance.

The solution to the startup printing is solved with:

```
% setenv S_SILENT_STARTUP 1
% setenv S_FIRST ''
```

The second command is optional, but in my case my `.First` prints something, and this command avoids the call to `.First`. If the commands will need your `.First` and it prints something, then you will need to change the `.First` or have the `S_FIRST` environment variable do the necessary actions.

```
% Splus REPORT rep.in2 rep.out2
REPORT will run in batch: input from file rep.in2,
output on file rep.out2.
% cat rep.out2
Harold weighs 15 kilograms, but
Dorothy weighs 14.
```

Now our output file looks fine.

You may be worrying that TEX files are going to be full of braces, so S is going to want to evaluate just about the whole file. The solution to this is to use the `-e` flag to `REPORT`. A bare `-e` means that the introductory brace for S commands is preceded by a backslash.

```
% cat rep.in3
Harold weighs \{ cat(jjwt["harold"], "\n", sep="") } kilograms,
but {\it Dorothy weighs \{cat(jjwt["dor"], "\n", sep="")}.}

% Splus REPORT -e rep.in3 rep.out3
REPORT will run in batch: input from file rep.in3,
output on file rep.out3.
% cat rep.out3
Harold weighs 15 kilograms,
but {\it Dorothy weighs 14.}
```

DANGER. There are occurrences of backslash-{ in TEX files, so this is only a partial solution. The help file for `REPORT` states that you can follow the `-e` with a character to use in place of the backslash, but that functionality seems to be broken (at least as of S-PLUS version 3.3 under some Unix systems). A workaround would be to change the backslash-{ combinations in the file that are not intended for S to something else, then change them back after S has

processed the file.

The `LOAD` utility performs static loading into S. A command with it might look like:

```
% S LOAD mycode.o my_other_code.o
```

If all is well, this will create a file named `local.Sqpe` in the current directory that will contain the usual S executable along with the additional code given in the command.

The script that starts S looks in the current directory for a file named `local.Sqpe`. If it exists, then this is used as the S executable. If it doesn't exist, then the usual executable is used. The `local.Sqpe` file is not going to be small, so you probably don't want many of them around.

There is more discussion of loading code into S starting on page 171.

A utility added to S-PLUS is `COMPILE` which is used to compile source files in a way that makes the resulting object files suitable to be loaded into S. This is really a call to `make`. One thing that it does is make sure that the `S.h` include file is found.

If `COMPILE` doesn't work properly, it is probably because of a problem with your local installation.

The purpose of the `CHAPTER` utility is to make it easy to create a library of S objects. The result of `CHAPTER` is to create a `Makefile` that allows easy updating. There are locations in the `Makefile` where you put the S dump files and the help files. Once the `Makefile` is edited, then you merely need to do:

```
% make install
```

and the S objects and help files are all put in the right place. You can redo the `make install` command any time that you have added files to the `Makefile`, or that you have changed existing dump or help files.

DANGER. The `Makefile` that `CHAPTER` creates does not handle objects created by `data.dump`. In the `Makefile` for the code that goes with this book that was made by `CHAPTER`, I have added the target:

```
install.objs: install.funs
        echo "poet.data.restore(unix('ls $(DDOBJS)'))" | Splus
```

which is made part of the `install` target. The `DDOBJS` variable in my case is declared to be:

```
DDOBJS=poet.verif.Q portoptgen.ctemplate.Q
```

These are the objects that the `sccs` function used `data.dump` on.

The situation is complicated when C or Fortran code is also involved, but `CHAPTER` can ease the pain in this case also.

The S-PLUS utility `NM` is a convenient way to use the Unix `nm` command to examine the symbols in a file of object code. The advantage of `Splus NM` over `nm` is that the former has the same format on all platforms. Here is an example:

```
% Splus NM digamma.o
digamma.o:          U .stret8
digamma.o:00000230 T _digamma_complex_Sp
digamma.o:00000000 T _digamma_real_Sp
digamma.o:          U _log
digamma.o:          U _set_inf
digamma.o:          U _set_na
digamma.o:          U _test_inf
digamma.o:          U _test_na
```

The C functions that are defined as static within the file do not appear, but the other functions do appear and are marked with a "T", meaning that they are defined. Symbols marked "U" are not part of the object file, so if this file is to be loaded into S, then those symbols have to be known to S.

Another S-PLUS utility is `HINSTALL` (it is undocumented). This installs help files, and is used like:

```
% Splus HINSTALL .Data/.Help perl.d transcribe.d
```

The first argument after `HINSTALL` is the help directory where the files are to go. After that an arbitrary number of files can be given that contain help. An advantage of using `HINSTALL` is that it automatically takes care of a file that is for more than one object. Near the top of a help file you will find a line like:

```
.FN stack
```

This line says what object the help file is for. To add the `print` method for `stack` to the help file, make it look like:

```
.FN stack
.FN print.stack
```

Mostly this `FN` command is ignored, but `HINSTALL` knows about it and does the right thing. That is, just putting `FN` lines in the file does nothing to make help for the additional objects unless you use `HINSTALL`.

Because version 4 of S changes the way help works, I imagine that `HINSTALL` will go extinct.

The category part of the S-PLUS help window system looks for the special file in each database on the search list that contains the information from the help files based on the function names, the titles and the keywords. When you add help files to the `.Help` subdirectory of a database, then the keywords in the new help files won't be recognized until you update the file. Update it with the command:

```
% Splus help.findsum .Data
```

(This utility fell outside of the capitalization convention, you'll notice.) The keywords in the help file can not be arbitrary—only a recognized set of keywords are used. See the list in the help file for `prompt`. There is also a `help.findsum` S-PLUS function that performs the same task as the utility.

The `ADDKEYWORD` utility is meant to facilitate adding keywords to the list of acceptable ones. My one attempt at using this utility met with failure. However, it did usefully point to the file that needs to be changed. It is reasonably obvious what to do once you find the right file.

The help files in version 3 of S are formatted by the Unix `nroff` command. Some machines do not have `nroff` so S will look in a `.Cat.Help` directory for preformatted help files before looking in the `.Help` directory. You can also use this if the extra added speed of getting help is worth the extra space taken up. The `CATHELP` utility will put the formatted files into the `.Cat.Help` directory. It is used like:

```
% mkdir .Data/.Cat.Help
% Splus CATHELP .Data
```

S-PLUS has added the `MASKED` utility that shows objects in the working database that mask other objects.

```
% Splus MASKED
Searching user directory .Data ...
```

S-PLUS also has a `masked` function, but if things are really messed up, it is possible that the function won't do. See page 135 for a discussion of masking.

One of the things that S does behind the scenes is to keep an audit file of the commands given to S. This is used by the `history` function, can be examined with the `AUDIT` utility, and has potential to be used for other purposes also. (The magic to turn auditing off is given on page 160.) If the audit file grows to more than a certain size, you will get a message like:

```
> q()
Warning: Audit file is 519851 characters long.
Run TRUNC_AUDIT to truncate it. See help file for complete
        instructions.
```

This is telling you to run a Unix command like:

```
% Splus TRUNC_AUDIT
Truncating audit file (.Data/.Audit) to most recent
100000 characters
Audit file truncated to 99913 characters
```

This saved the most recent part of the file and threw away the older part. You can give it a number which tells the approximate number of characters to save. If you want none of the file saved, you would say:

```
% Splus TRUNC_AUDIT 0
```

The `AUDIT` utility provides a means of viewing what an audit file contains. The default file is the one that would be written to when you start S in the current directory—that is, the `.Audit` file that is in the `.Data` that S will use as the working directory—but you can specify the file of your choice.

```
% S AUDIT
Reading audit file ".Data/.Audit"
3652 statements, 352 identifiers
audit: G jjn
.Data/jjn
3463: jjn
3462: jjn[jjn < 0] <- NA
3461: jjn
```

Above we start up `AUDIT` and look at the commands that "get" an object named `jjn`. Below we view the names that were both put and got (there are a lot, so the output is abridged):

```
audit: N GP
rw .Data/jjsy
rw .Data/jjmat
rw .Data/jjv
  ...
```

Perhaps one of the more useful commands is the ability to backtrack a command number:

```
audit: B 3463
jjn
~~~~~~~~~
3463: jjn
3462: jjn[jjn < 0] <- NA
3268: jjn <- 1:5

audit: q
```

Type "q" to exit the AUDIT program.

DANGER. I haven't ever gotten the E command of AUDIT to work. Perhaps it does work on some machine, but I haven't found it.

The LICENSE utility of S-PLUS is primarily for system administration, but there is one function of general interest. The command

```
% Splus LICENSE users
```

produces a list of those currently using S-PLUS licenses. So you can decide who to harass when you can't get a license.

Finally two more utilities that may be of interest are EXEC and CSH. The EXEC utility executes a Unix command in the environment of S, and CSH gives you a C-shell with the S environment.

```
% Splus EXEC env
```

9.3 Zero Length Objects

Objects of zero length are not uncommon in S. They can be unnerving for the uninitiated, but their existence adds a great deal of versatility to the language. *Should we believe in nothing?* [28]

Here is a common situation where zero length objects occur.

```
> jjn <- 1:5
> jjn[jjn < 0]
numeric(0)
```

Even when doing a replacement here, there is no problem.

```
> jjn[jjn < 0] <- NA
> jjn
[1] 1 2 3 4 5
```

A source of confusion is the difference between a zero length character vector and the empty string.

```
> character()
character(0)
> length(character())
[1] 0
> length("")
[1] 1
> nchar(character())
numeric(0)
> nchar("")
[1] 0
```

The `character()` object contains no strings, while `""` is one string that has no characters in it.

Recursive objects can have zero length also:

```
> list()
list()
> length(list())
[1] 0
> parse(text="")
expression()
```

An object printed as `NULL` (mode `null`) is a special zero length object. An object that is `numeric(0)` contains nothing, but if it contained anything, the elements would be numbers. A `NULL` object contains nothing, and it is not specified what it would contain if it did have something. *Its vacancy glitters round us everywhere.* [29]

A common occurrence of `NULL` is as the result of the extraction from a list of a component that doesn't exist.

```
> jjlist
$a:
[1] "aaa" "aa"
```

```
$b:
[1] "bb"  "bbb"

$c:
[1] "ccc" "ccc"

> jjlist$d
NULL
```

Though it is often irrelevant, there is a difference between a component that does not exist and a component whose value is `NULL`.

```
> jjlist[2] <- NULL # no change
> jjlist
$a:
[1] "aaa" "aa"

$b:
[1] "bb"  "bbb"

$c:
[1] "ccc" "ccc"

> jjlist[2] <- list(NULL) # change component
> jjlist
$a:
[1] "aaa" "aa"

$b:
NULL

$c:
[1] "ccc" "ccc"

> jjlist[[2]] <- NULL # delete component
> jjlist
$a:
[1] "aaa" "aa"

$c:
[1] "ccc" "ccc"
```

So to remove a component, use double brackets and assign `NULL`. To change the value of a component to `NULL`, use single brackets and a list whose component is `NULL`.

The `fjjcylinder` function returns the radius, height and volume of a cylinder. Give the function any two of these, and it computes the third.

```
"fjjcylinder"<-
function(r = ((2 * V)/pi/h)^0.5, h = (2 * V)/pi/r^2, V
         = 0.5 * pi * r^2 * h)
{
        list(r = r, h = h, V = V)
}
> fjjcylinder(r=1:3, h=3.5)
$r:
[1] 1 2 3

$h:
[1] 3.5

$V:
[1]  5.497787 21.991149 49.480084

> fjjcylinder(h=3.5, V=21:23)
$r:
[1] 1.954410 2.000402 2.045361

$h:
[1] 3.5

$V:
[1] 21 22 23
```

Obviously if only one argument is given, there has to be some sort of trouble.

```
> fjjcylinder(r=4)
Error in fjjcylinder(r = 4): Recursive occurrence of
        default argument "h"
Dumped
```

Where we're caught is when S notices that it is being asked to resolve a circular reference. Now we look at the state of the objects at the time of the error.

```
> debugger()
Message: Recursive occurrence of default argument "h"

1:
2: fjjcylinder(r = 4)
Selection: 2
Frame of fjjcylinder(r = 4)
d(2)> ?
```

```
1: V
2: h
3: r
d(2)> mode(r)
[1] "numeric"
d(2)> mode(h)
[1] "unknown"
d(2)> mode(V)
[1] "unknown"
d(2)> missing(r)
[1] F
d(2)> missing(h)
[1] T
d(2)> missing(V)
[1] T
```

Note that not all versions of S (or S-PLUS) will give you the same answers in this example.

There are really two senses of what a missing object is. The first and most proper is that it is an object that corresponds to an argument of a function that was not given in the call. For example, the call `fjjcylinder(r=4)` means that `h` and `V` will be missing.

The second sense is that "missing" means an object of mode `missing`. Here is a little test case:

```
> fjjtestmiss1
function(x, y, z)
{
        ans <- logical(3)
        names(ans) <- c("x", "y", "z")
        ans2 <- ans
        ans["x"] <- missing(x)
        ans["y"] <- missing(y)
        ans["z"] <- missing(z)
        mode(y) <- "missing"
        ans2["x"] <- missing(x)
        ans2["y"] <- missing(y)
        ans2["z"] <- missing(z)
        list(before = ans, after = ans2, call =
                match.call())
}
> fjjtestmiss1(1,2)
$before:
 x y z
 F F T
```

```
$after:
 x y z
 F T T

$call:
fjjtestmiss1(x = 1, y = 2)

> fjjtestmiss1()
Error in fjjtestmiss1: Argument "y" is missing, with no
        default: fjjtestmiss1()
Dumped
```

So you can not make a missing argument missing. The proper approach is:

```
> fjjtestmiss2
function(x, y, z)
{
        ans <- logical(3)
        names(ans) <- c("x", "y", "z")
        ans2 <- ans
        ans["x"] <- missing(x)
        ans["y"] <- missing(y)
        ans["z"] <- missing(z)
        if(!missing(y))
                mode(y) <- "missing"
        ans2["x"] <- missing(x)
        ans2["y"] <- missing(y)
        ans2["z"] <- missing(z)
        list(before = ans, after = ans2, call =
                match.call())
}
```

I'm not sure why you would ever want to do such a thing, but it appeals to my sense of irony.

A mode of `unknown` means that S is left confused about the situation, as in the `fjjcylinder(r=4)` call where `V` can't be computed. *Nobody knew where I was and now I am no longer there.* [30]

There is a way to make a constraint-based function like `fjjcylinder` behave better.

```
"cylinder"<-
function(r = ((2 * V)/pi/h)^0.5, h = (2 * V)/pi/r^2, V
         = 0.5 * pi * r^2 * h)
{
        if(nargs() != 2)
```

```
                stop("must give exactly two arguments"
                     )
        list(r = r, h = h, V = V)
}
```

The `nargs` function returns the number of arguments that are actually passed into the call for the function it is in. We know in this case that we need exactly two arguments. We could allow all three arguments to be given, and perform some sort of adjustment to make the three quantities follow the constraint.

```
> cylinder(r=4)
Error in cylinder(r = 4): must give exactly two arguments
Dumped
```

This is a more rational error message than that from `fjjcylinder`.

9.4 Modes

There are numerous modes, each producing a different type of object. The modes are divided into three categories: atomic, recursive and language. Atomic (which includes mode `null` as well as those given in chapter 1) does not overlap with the other two categories. Recursive objects may contain other objects of the same mode. Some language objects are recursive.

Although not a mode, "structures" are a way that objects are described. The `structure` function has a `.Data` argument and takes an arbitrary number of additional arguments. These additional arguments are the attributes of the object. Here are some examples:

```
> dput(1:4)
c(1, 2, 3, 4)
> jjstruc <- 1:4
> class(jjstruc) <- "whim"
> dput(jjstruc)
structure(.Data = c(1, 2, 3, 4), class = "whim")
> jjmat <- matrix(1:4, 2)
> dput(jjmat)
structure(.Data = c(1, 2, 3, 4), .Dim = c(2, 2))
> attributes(jjmat)
$dim:
[1] 2 2

> jjmat2 <- 1:4
> attr(jjmat2, ".Dim") _ c(2,2)
> jjmat2
```

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

So `.Dim` is another name for the `dim` attribute (due I'm sure to some sort of historical aberration). By the way, the `dput` function is what does the real work in the `dump` function.

S functions are objects of mode `function`. S thinks the length of a function is the number of arguments plus one. The extra "component" is the body of the function. The names of a function are the names of the arguments, the body is not named. Since the default of an argument to a function can be a function, functions are recursive. In common usage, as on page 25, the length of a function is the number of lines contained in the body of the function.

A special type of function is an *assignment function*. These are functions that appear on the "wrong" side of an assignment, for example:

```
dim(x) <- c(4, 6)
```

The assignment function in this case has the (invalid) name of `dim<-`. Assignment functions have their own protocol to follow—see the explanation of `names<-.rationalnum` on page 248.

Using an assignment function may cause a local version of the object to be created. When you are at the S prompt, then the local version will be in the working database.

```
> find("freeny.x")
[1] "/usr/lang/s3.1/s/.Datasets"
> dim(freeny.x) <- c(13, 12)
Warning messages:
  Invalid dimnames deleted in: dim(freeny.x) <- c(13, 12)
> find("freeny.x")
[1] ".Data"                       "/usr/lang/s3.1/s/.Datasets"
```

When in a function, then the new copy will be in the frame of the function. Examples of this are in `fjjlog2` on page 220, and `loan` on page 226.

Objects of mode `expression` are (the equivalent of) parsed commands. This is a recursive mode since expressions can contain expressions.

Mode `call` represents objects that are calls to a function. The first component of a call is the name (mode `name`) of the function being called. The rest of the call is the arguments given.

```
> length(expression(sin(1:4)))
```

```
[1] 1
> mode(expression(sin(1:4))[[1]])
[1] "call"
> expression(sin(1:4))[[1]][[1]]
sin
> mode(expression(sin(1:4))[[1]][[1]])
[1] "name"
> expression(sin(1:4))[[1]][[2]]
1:4
> mode(expression(sin(1:4))[[1]][[2]])
[1] "call"
> mode(expression(sin(1:4))[[c(1,2,2)]])
[1] "numeric"
```

Here we see an example of a type of subscripting that was promised in chapter 1. The command

```
x[[c(1,2,2)]]
```

is equivalent to

```
x[[1]][[2]][[2]]
```

Working with language constructs is one of the more likely spots where this form of subscripting is useful. The `portoptgen` function on page 348 uses it extensively to build an S function. Another equivalent form is:

```
x[[list(1,2,2)]]
```

This allows a mixture of numeric and character subscripts.

Formulas use the tilde operator. This operator is particularly lazy—it does nothing but return a call to itself. This seems like either a contradiction or an infinite loop, but it works nonetheless. It preserves what was typed while being in a form that can be taken apart and manipulated.

```
> jjform <- y ~ x1 + x2
> mode(jjform)
[1] "call"
> is.recursive(jjform)
[1] T
> is.language(jjform)
[1] T
> jjform[[1]]
 ~
> jjform[[2]]
y
```

```
> jjform[[3]][[2]]
x1
> jjform[[c(3,2)]]
x1
```

See the chapter on formulas for more.

Something of mode `name` is in the crack between the character string that names an object, and the object itself. *To stand in the doorway is to stand in more brief worlds than this one* [31]

```
> jjform[[2]] <- as.name("z")
> jjform
z ~ x1 + x2

> jjln <- list(a=0, b=as.name("freeny.y"))
> jjln
$a:
[1] 0

$b:
freeny.y

> jjln$b
freeny.y
> eval(jjln$b)
           1Q      2Q      3Q      4Q
1962:         8.79236 8.79137 8.81486
1963: 8.81301 8.90751 8.93673 8.96161
1964: 8.96044 9.00868 9.03049 9.06906
1965: 9.05871 9.10698 9.12685 9.17096
1966: 9.18665 9.23823 9.26487 9.28436
1967: 9.31378 9.35025 9.35835 9.39767
1968: 9.42150 9.44223 9.48721 9.52374
1969: 9.53980 9.58123 9.60048 9.64496
1970: 9.64390 9.69405 9.69958 9.68683
1971: 9.71774 9.74924 9.77536 9.79424
```

Names are elusive because they are almost always evaluated.

There are several modes that control flow within the language. These modes include: `if`, `for`, `while`, `repeat`, `return`, `break`, and `next`.

```
> jjif <- expression(if(x>0) y else z)[[1]]
> jjif
if(x > 0) y else z
> mode(jjif)
```

```
[1] "if"
> length(jjif)
[1] 3
> is.recursive(jjif)
[1] T
> is.language(jjif)
[1] T
> jjif[[1]]
x > 0
> mode(jjif[[1]])
[1] "call"
```

So an `if` object has length 3 (2 when the "else" is missing) and each piece is usually some other language object. Here we see how an `if` without an `else` works:

```
> rm(jj, jjx)
> jj <- 1
> jjx <- if(jj == 0) 3
> jjx
NULL
> rm(jjx)
> if(jj == 0) jjx <- 3
NULL
> jjx
Error: Object "jjx" not found
```

The value of the `else` clause when it is missing is `NULL`. In the second statement above the assignment to `jjx` is not made so it doesn't exist rather than having the value `NULL` as in the first statement.

```
> jjbrk <- vector("break")
> jjbrk
break
> is.recursive(jjbrk)
[1] F
> is.atomic(jjbrk)
[1] F
> is.language(jjbrk)
[1] T
> length(jjbrk)
[1] 0
```

So an object of mode `break` is neither recursive nor atomic.

See the `find.assign` function on page 211 for a couple additional modes, and to see language modes in action.

9.5 Evaluation

The first step S takes in order to perform a command is to parse it. This is the process of breaking the command into units of known type. Consider the pseudo-English sentence "The wugbun was quentled in the nebrad pebfin." We parse this by realizing that "wugbun" is a noun and the subject of the sentence; "quentled" is a verb in the past tense of the passive voice; "nebrad" is an adjective modifying "pebfin" which is a noun and the object of the preposition "in." We don't know what these words mean, but that doesn't hinder us from understanding the form of the sentence. Learning what the words mean and then understanding the sentence would be analogous to evaluation in S. *I can't blab such blither blubber.* [32]

When S parses a statement, it looks at each character in turn and tries to make sense of it. If it sees `!` as the first character, then it interprets this as an escape to the operating system, but if it sees it only later in a statement, then it tries to fit it into one of the operators `!=` or `!` (meaning "not"). S decides what are numbers and what are object names. If at some point, the parser can't fit a character in sensibly, then a syntax error results. Here are two statements:

```
f   (a, b)
f + (a, b)
```

They are exactly alike except for the third character. The first statement will parse fine. The second statement will parse okay through the "a", but when it hits the comma, it realizes there is a problem—the parenthesis can't be thought of as the start of an argument list because of the `+` operator, and argument separation is the only use in S that a comma has.

```
> parse(text="f + (a, b)")
Syntax error: "," used illegally at this point:
f + (a,
Dumped
```

There is no problem making objects with new names, but the parser would not understand it if you tried to create new operators at whim. You can not add, say, `++` as a new operator because that would change the syntax of the language.

```
> ++2
[1] 2
> +-++2
[1] -2
> 3//5
Syntax error: */ ("/") used illegally at this point:
3//
```

Using ++ in S does parse—it is interpreted as two unary plus signs in a row. There is no interpretation for something like // so a syntax error results.

However, S provides a way to add new operators, but the name of the operator must begin and end in %. Here is a list of all such operators from S-PLUS version 3.3.

```
> find.objects("%")
 $SHOME/s/.Functions $SHOME/splus/.Functions
 "%%"                "%*%"
 $SHOME/splus/.Functions $SHOME/s/.Functions
 "%*%.default"           "%/%"
 $SHOME/s/.Functions
 "%o%"
```

The %*% function is generic with only a default method appearing here.

The characters that appear inside the % symbols are quite arbitrary. The following operator even contains a pound sign that elsewhere would signal the start of a comment. (Quotes are not allowed.)

```
> "%@#&*#%" <- function(e1, e2) rpois(e1, e2)
> 3 %@#&*#% 5
[1] 3 5 2
```

Below is the definition of an "element of" operator that might be of use at times.

```
> "%e%" <- function(x,y) match(x, y, nomatch=0) > 0
> c(-1, 34, 2) %e% 1:10
[1] F F T
> 1:10 %e% c(-1, 34, 2)
 [1] F T F F F F F F F F
```

Statisticians looking at the list of operators may be wondering why %in% that is used in analysis of variance formulas is not listed. The functionality is there:

```
> raov(Moisture ~ Batch + Sample %in% Batch, pigment)
Call:
   raov(formula = Moisture ~ Batch + Sample %in% Batch,
        data = pigment)

Terms:
                   Batch Sample %in% Batch Residuals
 Sum of Squares 1210.933           869.750    27.500
Deg. of Freedom       14                15        30
```

```
Residual standard error: 0.9574271
Estimated effects are balanced
```

The `%in%` is not really a function, likewise the `+` in the formula is not really addition. Formulas use S parsing (so the precedence in formulas is the same as in S commands), but the formulas are not evaluated in S.

In the evaluation process, S creates a *memory frame* (also called an evaluation frame) for each function call as it is evaluated. Actually, since this is the chapter where I'm promising to tell the truth, I have to state that there are "quick calls" (which are `.Internal` functions satisfying certain conditions) that do not get their own frame. *We Thin gin. We Jazz June.* [33]

As a function is evaluated, functions called within that function need to be evaluated, and functions within those functions need to be evaluated, and so on. Hence there is a stack of memory frames that grows and shrinks throughout the evaluation. Each of the frames has a unique number which gives the depth it is in the stack.

Below are some examples of frames shown by `traceback` from a call to `browser`. The function called on the command line will be:

```
> fjj4
function(x, add.new = F, exp.use = F)
{
        char.logic <- paste(add.new, exp.use, sep =
                ".")
        switch(char.logic,
                FALSE.FALSE = log(fjjb(x) + 3),
                FALSE.TRUE = exp(fjjb(x) + 3),
                TRUE.FALSE = log(fjjb(x) %+% 3),
                TRUE.TRUE = exp(fjjb(x) %+% 3))
}
```

CODE NOTE. The `char.logic` variable provides a mechanism for flattening out a messy nest of "if-else"s. In this case where it is only two deep, it is perhaps a little gratuitous, but with more nesting, it can be very aesthetic, especially if a portion of code corresponds to more than one case.

Functions used by `fjj4` include:

```
> fjjb
function(x, y = 2)
{
        browser()
        x + y
}
> get("%+%")
```

```
function(e1, e2)
{
        .Internal(e1 + e2, "do_op", T, 5)
}
```

The %+% operator is the same as the + operator except that the former has braces around the call to .Internal.

```
> fjj4(5)
Called from: fjjb(x)
b(4)> traceback()
7: eval(i, eval.frame, parent) from 6
6: browser.default(nframe, message = paste("Called from:", from 5
5: browser() from 4
4: fjjb(x) from 2
3: log(fjjb(x) + 3) from 2
2: fjj4(5) from 1
1:  from 1
b(4)> 0
[1] 2.302585
> fjj4(5, add=T)
Called from: fjjb(x)
b(5)> traceback()
8: eval(i, eval.frame, parent) from 7
7: browser.default(nframe, message = paste("Called from:", from 6
6: browser() from 5
5: fjjb(x) from 2
4: fjjb(x) %+% 3 from 2
3: log(fjjb(x) %+% 3) from 2
2: fjj4(5, add = T) from 1
1:  from 1
b(5)> 0
[1] 2.302585
> fjj4(5, T, T)
Called from: fjjb(x)
b(4)> traceback()
7: eval(i, eval.frame, parent) from 6
6: browser.default(nframe, message = paste("Called from:", from 5
5: browser() from 4
4: fjjb(x) from 2
3: fjjb(x) %+% 3 from 2
2: fjj4(5, T, T) from 1
1:  from 1
b(4)> 0
[1] 22026.47
```

From these tracebacks we can infer that `exp` and `+` are quick calls while `log` and `%+%` are not. When `log` is used, there is a frame corresponding to it, but there is never a frame for `exp`.

The number after the "from" gives the frame number of the parent of each frame. Frame 1 has the curious property of being its own parent.

When S looks for a an object, it looks in a number of places until it finds an object that satisfies the requirements. Often S knows that it needs a function so it will ignore objects with the same name that are not functions (it gives a warning if this happens). The places searched are:

- The first place S looks is the current frame.
- The second place where it looks is a funny critter called frame 1. Frame 1 is best explained as lasting from the start of a command being evaluated until the prompt returns. Note that not all frames are searched.
- The third place to look is database 0, also referred to as frame 0, the session database or the session frame—so many names for one place. Database 0 starts when an S session starts and dies when S is exited.
- Finally each database on the search list is searched in turn.
- If the object is not found in any of these places, then an error occurs.

Frame 1 is generally underused. When a variable needs to be global, this is almost always the location where it should be put. You want the variable to live only as long as required. If it will only be used during a function call, then it should go into frame 1. If it needs to outlive a single function call, but can be logically or conveniently recreated each session, then it should go into database 0. Only when the variable needs to live beyond a single session, should it go into the working database.

The `whence` function returns the location where an object is found. It first looks for the appropriate object in the memory frame of interest, then in frame 1, then in database 0, then on the search list. A negative return value means that the object is found in a frame.

```
"whence"<-
function(x, mode. = "any", offset = 0)
{
        if(!is.character(x) || length(x) != 1)
                stop("need single character string for x")
        parent <- sys.parent() - offset
        if(exists(x, mode = mode., frame = parent))
                return( - parent)
        if(exists(x, mode = mode., frame = 1))
```

```
                return(-1)
        if(exists(x, mode = mode., frame = 0))
                return(0)
        find(x, mode = mode., numeric = T)[1]
}
```

The `whence` function is used on page 228 in the `[.stack` function.

The `find.assign` function takes an expression and attempts to return all of the variables that are assigned to in the expression.

```
"find.assign"<-
function(line)
{
        switch(mode(line),
                "<-" = ,
                "<<-" = {
                        ans <- line[[1]]
                        if(mode(ans) == "call")
                                return(character(0))
                        else if(mode(line[[2]]) == "function") {
                                this.fun <- line[[2]]
                                return(c(ans, names(this.fun)[ - length(
                                  this.fun)], Recall(this.fun[[length(this.fun)
                                  ]])))
                        }
                        else if(mode(line[[2]]) == "<-" || mode(line[[2]]) ==
                                "<<-") {
                                return(c(ans, Recall(line[[2]])))
                        }
                        else return(as.character(ans))
                }
                ,
                comment.expression = return(Recall(line[[1]])),
                "{" = {
                        ans <- character(0)
                        for(i in 1:length(line)) {
                                ans <- c(ans, Recall(line[[i]]))
                        }
                        return(ans)
                }
                ,
                "if" = return(c(Recall(line[[1]]), Recall(line[[2]]), Recall(
                        line[[3]]))),
                "while" = {
                        return(c(Recall(line[[1]]), Recall(line[[2]])))
                }
                ,
                "for" = {
```

```
                        loopv <- substring(deparse(line)[1], 5)
                        loopv <- substring(loopv, 1, match(32, AsciiToInt(loopv
                                )) - 1)
                        return(c(loopv, Recall(line[[3]])))
                }
                ,
                "repeat" = return(Recall(line[[1]])),
                call = {
                        cnam <- as.character(line[[1]])
                        switch(cnam,
                                assign = return(line[[2]]),
                                switch = {
                                  ans <- character(0)
                                  for(i in 2:length(line)) {
                                    ans <- c(ans, Recall(line[[i]]))
                                  }
                                  return(ans)
                                }
                                ,
                                return(character(0)))
                }
                )
}
```

The basic idea of this function is dead simple—switch on the mode of the output, then recurse on pieces of the input using `Recall` (see page 239 for further explanation of `Recall` and recursion). The details can get a little messy, though.

Although writing `find.assign` is a fine exercise to understand S, its real purpose is as a subfunction to `global.vars` which attempts to return the variables that are global to a function. A mistake that is easy to make is to use a different name in the body of a function than in the argument list for the same variable. For example:

```
> global.vars(function(xmat) symsqrt(x))
[1] "x"
```

This makes the misspelled name in the body a global variable. The situation is particularly dangerous when there is an object of that name on the search list—possibly created for debugging purposes. The `global.vars` function is an attempt to have a diagnostic for this.

```
"global.vars"<-
function(fun)
{
        if(is.character(fun))
                fun <- get(fun)
        fnam <- names(fun)
        fnam <- fnam[ - length(fnam)]
```

```
        allv <- all.vars(fun, uniq = T)
        allv <- allv[!match(allv, c("NA", "T", "F",
                "Inf", "NULL", ".Internal", fnam),
                nomatch = 0)]
        body <- fun[[length(fun)]]
        if(mode(body) == "{") {
                anam <- character(0)
                for(i in 1:length(body)) {
                        anam <- c(anam, find.assign(
                                body[[i]]))
                }
        }
        else anam <- character(0)
        if(length(anam))
                allv <- allv[ - match(anam, allv,
                        nomatch = 0)]
        allv
}
```

The `all.vars` function returns a vector of all of the variables in an expression, in this case we want just the unique names rather than each occurence of a name. Next we get rid of the common names we know are wrong and the argument names. Finally we get rid of the names that are assigned to in the body of the function.

If you've been paying attention you should be a little concerned about this last move—on page 72 I said there was a problem with subscripting with negative numbers from `match`. But given the protection for the length of `anam`, there will be at least one match if that line of code is reached, so it should be fine.

S uses *lazy evaluation*. This means that arguments to functions are evaluated only when they are actually needed. Being used by another function doesn't count—the object is passed into the call unevaluated. Once the value is needed, it is evaluated in the appropriate frame and the value is effectively passed to all of the intermediate frames. An advantage of lazy evaluation is that unnecessary evaluation is avoided. Another is that the default to an argument can involve variables created inside the function as long as the variables exist by the time that the argument is needed. The down-side is that in certain situations there are problems with knowing where the evaluation should take place. The most common way to get this problem is when using formulas inside of functions—see page 291. The `delay.eval` function on page 217 slightly illuminates lazy evaluation.

S is essentially a functional language, which means that variables in one function are not changed by functions called by the function. That is, side effects are minimized, and generally only occur in a limited set of circumstances.

Actually, it is possible with the `assign` function to change objects in other frames (one reason why S is not strictly a functional language).

When objects are assigned to databases, they are not actually written to the disk until just before S gives another prompt. The assignments are said not to be *committed* until the prompt is given. For the most part, this is of no concern, but there are times when it matters. Instances where it does matter include large, repetitive computations (page 355), and `browser` calls (page 138). The `immediate` argument to `assign` insures that the commitment is made at the time of evaluation rather than held until later. See also the discussion of `synchronize` on page 103.

There is a number of functions that provide information on the stack of memory frames at a given moment. Here is a list of them:

```
> objects(5, pat="^sys")
 [1] "sys.call"     "sys.calls"    "sys.frame"
 [4] "sys.frames"   "sys.function" "sys.nframe"
 [7] "sys.on.exit"  "sys.parent"   "sys.parents"
[10] "sys.status"   "sys.trace"
```

We can use the `fjj4` function again to explore what some of these functions do.

```
> fjj4(6)
Called from: fjjb(x)
b(4)> traceback()
7: eval(i, eval.frame, parent) from 6
6: browser.default(nframe, message = paste("Called from:", from 5
5: browser() from 4
4: fjjb(x) from 2
3: log(fjjb(x) + 3) from 2
2: fjj4(6) from 1
1:  from 1
b(4)> sys.nframe()
[1] 4
b(4)> sys.parent()
[1] 2
```

Although it is a little confusing with the `browser` complicating things, we get the answer that we are in frame number 4 and the parent to frame 4 is frame 2.

```
b(4)> sys.call()
fjjb(x)
b(4)> sys.frame()
```

```
$.Auto.print:
[1] T

$x:
.Argument(x, x = )

$y:
.Argument(, y = 2)

b(4)> sys.function()
function(x, y = 2)
{
        browser()
        x + y
}
```

The `.Argument` (which is a mode) objects are unevaluated arguments to the function.

```
b(4)> sys.parents()
[1] 1 1 2 2 4 5 6 4
```

`sys.call` is a close synonym of `match.call`. Either of them can be used to produce the `call` component or attribute of the results of functions. The difference is that `match.call` always gives the argument name:

```
> fjj5
function(x)
{
        match.call()
}
> fjj6
function(x)
{
        sys.call()
}
> fjj5(y ~ x+z)
fjj5(x = y ~ x + z)
> fjj6(y ~ x+z)
fjj6(y ~ x + z)
```

The `ignore.error` function allows a value to be given to a computation that produces an error, and the computation continues. This function provides an example of how to control evaluation as well as introducing the `restart` function.

```
"ignore.error"<-
function(call, value = NULL)
{
        nframe <- sys.nframe()
        flag <- paste("flag.ignorE.error", nframe, sep = ".")
        if(exists(flag, frame = 1) && get(flag, frame = 1)) {
                assign(flag, F, frame = 1)
                cat("(ignored)\n", file = "|stderr")
                return(value)
        }
        else assign(flag, T, frame = 1)
        restart(T)
        ans <- eval(call, nframe - 1)
        assign(flag, F, frame = 1)
        ans
}
```

Let's go through the execution of this function. We start by finding where we are in the stack of frames. Next the variable name for the flag is created—the funny capitalization helps to avoid name conflicts, but the important part is the frame number in case `ignore.error` is called in more than one level. If the flag exists and is TRUE, then the flag is changed to FALSE, the error message is appended, and the function exits with the return value. Otherwise the flag is assigned to be TRUE, and `restart` is called so that evaluation will resume after an error. Now finally, the actual statement of interest is evaluated in the proper frame. If no error occurs, then evaluation within `ignore.error` continues—the flag is turned to FALSE and the result is returned. If an error does occur, then the `restart` causes evaluation to resume at the beginning of the `ignore.error` call which means that the condition in the `if` will be TRUE. Note that you may be in for a wild ride if you use a function containing `restart` that is not properly debugged.

Below is a function that uses `ignore.error` and an example of its use. When `ignore.error` is used, it is reasonable to set the `error` option to `NULL` to save time.

```
"fjjigerr"<-
function(input)
{
        this.mat <- base.mat <- matrix(c(2, 1, 1, 1),
                2)
        lin <- length(input)
        ans <- array(NA, c(lin, 2), list(input, NULL))
        for(i in 1:lin) {
                this.mat[4] <- base.mat[4]/input[i]
                ans[i,  ] <- ignore.error(eigen(
                        this.mat)$val, NA)
```

```
        }
        ans
}
> fjjigerr(-2:1)
Error in eigen.default(this.mat): missing or infinite
        values in x
Dumped
(ignored)
       [,1]       [,2]
-2 2.350781 -0.8507811
-1 2.302776 -1.3027756
 0       NA         NA
 1 2.618034  0.3819660
```

The `for` loop in `fjjigerr` uses `1:lin` where `lin` is `length(input)`. There is no assurance that `input` will have a length, so the loop might fail—see page 128. In this case it isn't tragic since it is just a temporary function, and it will fail on `array` anyway. However, it is good to keep these things in mind for when they do count.

The `delay.eval` function plays on lazy evaluation to allow evaluation to be performed in a different memory frame than usual.

```
"delay.eval"<-
function(expr, frames = 1)
{
        eval(substitute(expr), local = sys.parent() + frames)
}
```

For example, the default value for `ncol.arg` in `diag.default` is `n`. The `n` is local to `diag.default` and not part of the environment of the frame calling `diag.default`. We can use the `n` inside a call to `delay.eval` though.

```
> args(diag.default)
function(x = 1, nrow.arg, ncol.arg = n)
NULL

> diag(1, nrow=3, ncol=2*n)
Error: Object "n" not found
Dumped

> diag(1, nrow=3, ncol=delay.eval(2*n))
     [,1] [,2] [,3] [,4] [,5] [,6]
[1,]    1    0    0    0    0    0
[2,]    0    1    0    0    0    0
[3,]    0    0    1    0    0    0
```

The `unabbrev.value` function is meant to allow an abbreviated function argument. That is, the argument can be one of a few strings that would need to be given in full without some mechanism like this.

```
"unabbrev.value"<-
function(x, choices)
{
        err <- F
        if(!is.character(x) || length(x) != 1) {
                err <- T
                emsg <- paste("need single character string for",
                        deparse(substitute(x)))
        }
        else {
                xnum <- pmatch(x, choices, nomatch = 0
                        )
                if(xnum == 0) {
                        err <- T
                        emsg <- paste(
                                "unknown or ambiguous choice for ",
                                deparse(substitute(x)),
                                ": ", x, sep = "")
                }
        }
        if(err) {
                eval(call("stop", emsg), local =
                        sys.parent())
        }
        else choices[xnum]
}
```

A particular feature of this function is that the error message contained in `emsg`—if `err` is TRUE—will come from the function with the argument, not from `unabbrev.value`.

An example of the use of this function is in `line.integral` on page 320.

The `.Internal` function calls C code that is specially written to understand S. You can not write your own `.Internal` functions, and you can not have access to the code that they use. However, a brief explanation of a `.Internal` call might help you sometime. Here is an example:

```
> get("*")
function(e1, e2)
.Internal(e1 * e2, "do_op", T, 4)
```

The `.Internal` function has four arguments. The first argument is an image of how the call must have looked, with appropriate substitutions made. S uses

this image internally—it is more than cosmetic. The second argument names the section of internal S code that is to be used, and the fourth argument is the specific part of that section that is to be used. For example, the * function uses case 4 of the do_op code. The second argument is of significance relative to the groups of the object-oriented programming. The third argument is a logical value which states whether the arguments are to be evaluated or not. One might think that arguments should always be evaluated, but then one would be wrong.

Options are implemented by using the .Options object that lives in database 0. This is a logical place for it since it must have persistence, but it is easily created and modified appropriately as S starts up. The .Options object is a named list where the name is the option and the component is the value of the option. For almost all purposes, you should only change options through the options function (or a modification like soptions given on page 45). If this were always the case, then there would be no reason to care how options are implemented. However there are a few occasions where finesse is desirable.

Here is a revision of the log function so that a warning is not given when negative numbers are given to it.

```
> fjjlog1
function(...)
{
        on.exit(options(old.opt))
        old.opt <- options(warn = -1)
        log(...)
}
```

Now we test that it works and time it relative to the original.

```
> fjjlog1(-4:4)
[1]        NA        NA        NA        NA      -Inf
[6] 0.0000000 0.6931472 1.0986123 1.3862944
> unix.time(for(i in 1:1000) fjjlog1(-4:4))
[1] 101.42   2.39 106.00   0.00   0.00
> unix.time(for(i in 1:1000) log(-4:4))
[1] 0.94000244 0.01999998 1.00000000 0.00000000
[5] 0.00000000
There were 50 warnings (use warnings() to see them)
```

If we had to put up with this much loss of speed, then we might well be happier with the warnings. (Note there were really 1000 warnings, but S saved only the first 50 of them.) But a different revision performs much better:

```
> unix.time(for(i in 1:1000) fjjlog2(-4:4))
[1] 3.22000122 0.02999973 3.00000000 0.00000000
[5] 0.00000000
```

The revised function is:

```
> fjjlog2
function(...)
{
        .Options$warn <- -1
        log(...)
}
```

This creates a local version of `.Options` with the change to `warn`. The new version of the `.Options` object is created due to the action of the `$<-` function. There is no need of `on.exit` because the local version will die when `fjjlog2` exits.

When you type the name of an object at an S prompt, the object is printed. A simple and logical idea, but it is a little bit complicated to implement because not everything is printed. There is a special object named `.Auto.print` that controls automatic printing. Suppose that we want to time a statement:

```
> unix.time(mean(rnorm(100)))
[1] 0.00999999 0.00999999 0.00000000 0.00000000 0.00000000
```

The statement above gives us the timing, but we don't get to see the result of the computation. We need a call to `print` around the statement being timed in order to see its result:

```
> unix.time(print(mean(rnorm(100))))
[1] 0.02105694
```

Now we see the result of the statement, but not the timing. The reason is that the last statement of `print` methods is `invisible(x)` so that the object being printed is returned but there can't be an infinite loop of printing. That call to `invisible` has set `.Auto.print` in frame 1 to FALSE so at the point that `unix.time` exits, automatic printing is not in effect. To see both results, we need two calls to `print`:

```
> print(unix.time(print(mean(rnorm(100)))))
[1] 0.2123258
[1] 0.00999999 0.00000000 0.00000000 0.00000000 0.00000000
```

Here is the definition of the `interlude` function whose use was explained on page 59.

```
"interlude"<-
function(x)
{
```

```
        if(!is.character(x))
                x <- deparse(substitute(x))
        if(exists(".Interlude", where = 0))
                ilist <- get(".Interlude", where = 0)
        else {
                ilist <- list(0)
                class(ilist) <- "interlude"
        }
        onelist <- list(total.time = rep(0, 5), ncalls = 0)
        iexpr <- expression(NULL, on.exit({
                .interludE.final <- proc.time()
                if(length(.interludE.init) == 3)
                        .interludE.init <- c(.interludE.init, 0, 0)
                if(length(.interludE.final) == 3)
                        .interludE.final <- c(.interludE.final, 0, 0)
                .interludE.list <- get(".Interlude", where = 0)
                .interludE.thisl <- .interludE.list[[.interludE.name]]
                .interludE.thisl$total.time <- .interludE.thisl$total.time + (
                        .interludE.final - .interludE.init)
                .interludE.thisl$ncalls <- .interludE.thisl$ncalls + 1
                .interludE.list[[.interludE.name]] <- .interludE.thisl
                assign(".Interlude", .interludE.list, where = 0)
        }
        ), .interludE.init <- proc.time(), NULL)
        mode(iexpr) <- "{"
        lenex <- length(iexpr)
        makefun <- rep(T, length(x))
        names(makefun) <- x
        if(length(old <- intersect(names(ilist), x)))
                makefun[old] <- F
        for(i in x) {
                ilist[[i]] <- onelist
                if(makefun[i]) {
                        thisfun <- get(i)
                        thisn <- length(thisfun)
                        thisbody <- iexpr
                        thisbody[[1]] <- substitute(.interludE.name <- iname,
                                list(iname = i))
                        thisbody[[lenex]] <- thisfun[[thisn]]
                        thisfun[[thisn]] <- thisbody
                        assign(i, thisfun, where = 0)
                }
        }
        assign(".Interlude", ilist, where = 0)
}
```

Let's start at the bottom and work towards the top. The last thing that `interlude` does is to assign `.Interlude` to the session database. This object is a list in which each component holds the time and number of calls for each of the functions being profiled.

Now look at the `for` loop—this loops over each function to be profiled. The first thing is to initialize the proper component of the list that will become `.Interlude`. The next thing to do is to modify the function to be profiled if necessary and assign the modified function to the session database.

Modifying the existing function is done by creating an object of mode "brace", then filling this object in with the name of the function being modified, and with the body of the real function (remember that the last component of a function is its body—the previous components are its arguments).

Here is an example of a function after `interlude` has gotten hold of it.

```
> sqrt
function(x)
x^0.5
> interlude(sqrt)
Warning messages:
  assigning "sqrt" on database 0 masks an object of
        the same name on database 5
> sqrt
function(x)
{
        .interludE.name <- "sqrt"
        on.exit({
                .interludE.final <- proc.time()
                if(length(.interludE.init) == 3)
                        .interludE.init <- c(
                                .interludE.init, 0, 0)
                if(length(.interludE.final) == 3)
                        .interludE.final <- c(
                                .interludE.final, 0, 0)
                .interludE.list <- get(".Interlude",
                        where = 0)
                .interludE.thisl <- .interludE.list[[
                        .interludE.name]]
                .interludE.thisl$total.time <-
                        .interludE.thisl$total.time + (
                        .interludE.final -
                        .interludE.init)
                .interludE.thisl$ncalls <-
                        .interludE.thisl$ncalls + 1
                .interludE.list[[.interludE.name]] <-
                        .interludE.thisl
                assign(".Interlude", .interludE.list,
                        where = 0)
        }
        )
```

```
        .interludE.init <- proc.time()
        x^0.5
}
```

So the usual function has just the last line of the modified function. There are three new commands in the function: the name of the function is captured, an `on.exit` call with a bunch of commands in it, and the current time is captured. (The `proc.time` function contains the current amount of time that the S session has consumed; so we get how much time the function uses by subtracting the result of `proc.time` at the beginning from the result at the end.)

More of this sort of function modification is exhibited in the section on Lagrange interpolation starting on page 322.

The `.Program` object controls what S does with the expressions that it gets. The in-built `.Program` essentially parses then evaluates the expressions (plus some other details like deciding what to print). You can use your own `.Program` if you like. This is a bit like putting a new foundation under S, and is not something that everyone will want to do.

Signals are a way to get a program to change its behavior while it is running. Although it is possible to change the keystrokes for signals initiated by the user, I will refer to them by their common mappings. The signals are sent by holding down the "control" key and hitting another character. The most common signals in Unix are control-c (interrupt), control-backslash (a stronger interrupt), control-d (a quit signal), and control-z (background).

Use control-c to stop an S command and get back to the prompt. Control-backslash is used in S to immediately exit from a call to `browser` (or `debugger`) and get back to the S prompt. Use control-backslash when you are at a `browser` prompt and you want to get back to an S prompt rather than have the function continue. You probably won't want to hit control-backslash twice since the second signal will carry you from the S prompt to the Unix prompt. In contrast control-c will leave you at an S prompt if you are at one already, and will leave you at a `browser` prompt if that's where you are. It is a Unix convention that control-d will exit you from a program—S follows that convention. Control-d is equivalent to the S command `q()`.

There are other types of signals that are initiated by programs; examples are floating point exceptions and segmentation violations.

9.6 Object-Oriented Programming

The `UseMethod` function is the key ingredient of a generic function. Often a call to `UseMethod` is the entire body of the function. Typically it is the class of the first argument which controls the method that will be dispatched. Almost all generic functions look like:

```
> cavort
function(x, ...)
UseMethod("cavort")
```

The first argument to `UseMethod` is the name of the generic function. Although this is optional, it is good form to include it. It is especially important to give the generic name when the name includes a period—there could be ambiguity of the method to use otherwise. Another use of this argument is to allow nicknames for the generic function. An example is:

```
> residuals
function(object, ...)
UseMethod("residuals")
> resid
function(object, ...)
UseMethod("residuals")
```

If you want an argument other than the first to be the one whose class controls the generic function, then the name of the argument needs to be given as the second argument to `UseMethod`.

```
> gambol
function(x, y, z, ...)
UseMethod("gambol", z)
```

Generic functions include `UseMethod`, however they need not be a bare call to `UseMethod`. Perhaps the simplest logical case where it isn't is:

```
"%myop%" <-
function(x, y)
if(length(class(x)))
     UseMethod("%myop%")
else
     UseMethod("%myop%", y)
```

This defines a generic operator where it first checks to see if it can dispatch using `x` and if not, it dispatches using `y`. Version 4 of S will change what you want to do here.

The `%myop%` function violates the rule that a generic function should always have the three-dots as an argument, but there is no point as long as it is always used as an operator.

A look with `browser` in the frame of a generic function reveals some new objects.

```
> gambol(2,4,jjz)
Called from: gambol.foo(2, 4, jjz)
```

```
b(2)> ?
1: .Class
2: .Method
3: .Group
4: z
5: .Generic
6: x
7: y
8: z
b(2)> .Class
[1] "foo"
b(2)> .Method
[1] "gambol.foo"
b(2)> .Group
[1] ""
b(2)> .Generic
[1] "gambol"
```

The objects `.Class`, `.Method`, `.Group` and `.Generic` keep track of all of the information that may be necessary during evaluation. The `.Generic` and `.Method` objects are self-explanatory. The `.Class` object gives the classes, including the present one, that may be of use. In the case below, the `"bar"` class of the input is not used because there is no method corresponding to that class.

```
> gambol(2, 5, jjzb)
Called from: gambol.foo(2, 5, jjzb)
b(2)> .Class
[1] "foo" "zax"
b(2)> class(z)
[1] "bar" "foo" "zax"
```

Groups provide a labor saving mechanism by allowing the programmer to write a single method for a number of generic functions. The useful groups are "Math", "Summary" and "Ops". The Math functions include the trigonometric functions, `gamma` and so on. The Summary functions are `all`, `any`, `max`, `min`, `prod`, `range` and `sum`. The Ops are the operators like addition and subtraction. See the Math example with rational numbers on page 252 for a look at using group methods.

I've been lying to you again. Although all of the generic functions that you write will use `UseMethod`, functions that are a call to `.Internal` are also generic. (Each group consists of the functions that call the same routine in their `.Internal`.) So you can write a `dim.foo` function which will be used as a method of `dim` for class `foo` even though `dim` is not obviously generic. These functions are called internal generic functions.

For the most part, methods have no distinguishing features—they look like ordinary functions. An exception is that `NextMethod` may be used in a method. As it's name implies, `Nextmethod` is synonymous with the next method available for the generic function. I have seldom found `NextMethod` useful, though there is an example of its use on page 252. Do note that the names of arguments in methods should be the same as those in the generic function, as described on page 128.

Inheritance is the part of object-orientation that allows you to build on top of existing classes. In some situations this is extremely powerful. I'm sure that some would argue that inheritance is the main point of object-orientation.

As an example, let's produce a "loan" object that inherits from data frames. The `loan` function creates an object that contains the details of the loan as it is first made.

```
"loan"<-
function(amount, rate, month, year)
{
        names(month.name) <- month.name
        ans <- data.frame(month = month.name[month],
                year = year, principal = amount,
                interest = as.double(NA), payment =
                as.double(NA))
        attr(ans, "rate") <- rate
        if(is.character(month))
                month <- pmatch(month, month.name)
        attr(ans, "last.date") <- c(month = month,
                year = year)
        class(ans) <- c("loan", "data.frame")
        ans
}
```

All that `loan` does is return a data frame that is a little bit specialized for what we want. In terms of inheritance, the key line is assigning the class to have length greater than one. Next comes a method of the `update` function for `loan` objects. The input in addition to the loan object is a vector of the next payments to be made.

```
"update.loan"<-
function(object, payment)
{
        last.date <- attr(object, "last.date")
        rate <- attr(object, "rate")
        npay <- length(payment)
        new.month <- last.date["month"] + 1:npay
        new.year <- (new.month - 1) %/% 12 + last.date[
```

```
                "year"]
        new.month <- (new.month - 1) %% 12 + 1
        last.prin <- object[nrow(object), "principal"]
        new.prin <- new.inter <- numeric(npay)
        for(i in 1:npay) {
                new.inter[i] <- this.inter <- round((
                        last.prin * rate)/12, 2)
                new.prin[i] <- last.prin <- last.prin +
                        this.inter - payment[i]
        }
        ans <- rbind(object, data.frame(month =
                month.name[new.month], year = new.year,
                principal = new.prin, interest =
                new.inter, payment = payment))
        attr(ans, "last.date") <- c(month = new.month[
                npay], year = new.year[npay])
        attr(ans, "rate") <- rate
        ans
}
```

With just these two functions we have everything that we really need.

```
> jjl <- loan(45000, .07, 2, 1902)
> jjl
     month year principal interest payment
1 February 1902     45000       NA      NA
> update(jjl, rep(275, 5))
     month year principal interest payment
1 February 1902  45000.00       NA      NA
2    March 1902  44987.50   262.50     275
3    April 1902  44974.93   262.43     275
4      May 1902  44962.28   262.35     275
5     June 1902  44949.56   262.28     275
6     July 1902  44936.77   262.21     275
```

That inheritance comes into this, is not entirely obvious—that's part of the power of it. We have used subscripting, `rbind` and `print` methods inherited from data frames without really noticing it. Each of those three functions is non-trivial, but we didn't have to worry about them in our application. It would be handy to write a `summary` method for the `loan` class—the inherited method is decidedly not what we want.

The first line of `loan` that puts names onto `month.name` allows us to use either the number of the month or (an abbreviation of) the month name.

```
> loan(45000, .07, "Feb", 1902)
     month year principal interest payment
1 February 1902     45000       NA      NA
```

A new version of `month.name` is created in the frame of `loan`, but after the function exits, the same `month.name` as usual will be the one seen.

9.7 Data Structures

An important topic in computer science is data structures. Examples of useful data structures are hash tables, binary trees, and linked lists. In large measure S programmers do not need to worry about these because S already provides tools equivalent to these structures. For example, named lists or vectors can perform similar functionality to a hash table.

Stacks are a common data structure in computer science. Below I present an implementation of stacks in S. I do this with great trepidation for two reasons. Stacks are often used because they are fundamental structures in the language and hence efficient—this is not the case here. More importantly this implementation breaks the expectation about side effects in S—subscripting a stack causes it to change, which is decidedly non-standard in S. My hope is that more good will result from this example than abuse. I remain skeptical, though.

```
"stack"<-
function(length. = 64, initial = NULL, update = T)
{
        ans <- vector("list", length.)
        if(size <- length(initial)) {
                if(size > length. && is.logical(update
                        ) && !update)
                        stop("stack overflow")
                ans[1:size] <- initial
        }
        atl <- list(class = "stack", size = size,
                update = update)
        attributes(ans) <- atl
        ans
}
```

The `stack` function returns an object of class `stack`.

CODE NOTE. Notice that there is a dot at the end of the `length.` argument. Without the dot, the argument name would conflict with the `length` function and we would get annoying warning messages. Because of partial argument name matching, we can think of the argument name being `length` without the dot.

The following function is the "pop".

```
"[.stack"<-
```

```
function(x, i)
{
        if(!missing(i))
                stop("only empty subscripting allowed")
        size <- attr(x, "size")
        if(!size)
                return(NULL)
        ans <- unclass(x)[[size]]
        xname <- deparse(substitute(x))
        xloc <- whence(xname, offset = 1)
        attr(x, "size") <- size - 1
        if(xloc < 0)
                assign(xname, x, frame = abs(xloc))
        else assign(xname, x, where = min(xloc, 1))
        ans
}
```

This performs the unpleasant deed of assigning the changed object behind the scenes, possibly in a new location. The last item in the stack is put into `ans`, and the size of the stack is changed. The item is not erased from the list, though that may be a good idea for applications in which the items on the stack are large.

The next function pushes an item onto the stack. This is a less troublesome function since a side effect is expected.

```
"[<-.stack"<-
function(x, i, value)
{
        if(!missing(i))
                stop("only empty subscripting allowed")
        size <- attr(x, "size")
        if(size == length(x)) {
                update <- attr(x, "update")
                xat <- attributes(x)
                if(is.logical(update)) {
                        if(update) {
                                length(x) <- 2 *
                                  length(x)
                        }
                        else stop("stack overflow")
                }
                else {
                        length(x) <- length(x) +
                                update
                }
                attributes(x) <- xat
```

```
        # length change destroys attributes
        }
        x[[size + 1]] <- value
        attr(x, "size") <- size + 1
        x
}
```

The main complication here is that the list underlying the stack may need to be lengthened. The `update` argument can be either logical or numeric. If it is numeric, then this is the number of components to be added to the list each time it must be lengthened. When `update` is logical, then either the length is doubled, or growing the stack is disallowed.

Of course we need a `print` method.

```
"print.stack"<-
function(x, ...)
{
        size <- attr(x, "size")
        if(size)
                print(unclass(x)[1:size], ...)
        else cat("(empty stack)\n")
        cat("Class:", class(x), "\n")
        invisible(x)
}
```

Here is a simple example of how a stack works.

```
> jjstk <- stack(3)
> print.default(jjstk)
[[1]]:
NULL

[[2]]:
NULL

[[3]]:
NULL

attr(, "class"):
[1] "stack"
attr(, "size"):
[1] 0
attr(, "update"):
[1] T
> jjstk
(empty stack)
```

```
Class: stack
> jjstk[] <- 1:4
> jjstk[] <- 1:3
> jjstk
[[1]]:
[1] 1 2 3 4

[[2]]:
[1] 1 2 3

Class: stack
> jjstk[]
[1] 1 2 3
> jjstk
[[1]]:
[1] 1 2 3 4

Class: stack
> jjstk[]
[1] 1 2 3 4
> jjstk[]
NULL
```

Some would argue that an error should result from popping an empty stack. I think it is better in general to not create an error when the result is not demonstrably wrong—specific applications can make it an error if need be. This is to some degree a matter of taste.

At this point the garden-variety use of the functions seems to be okay. One might be concerned if adding a `NULL` item onto the stack works properly (it does), and there are all sorts of things to be tested concerning the side effects.

Queues can be defined analogously.

9.8 The R Language

To a great extent, code that works in S will do the same thing in R. However, there are some significant differences. One of the most drastic is that R inherits a different idea of how to search for objects due to its lineage to Lisp.

Here is an example using R.

```
R> x <- -1:4
R> fjj.s.r
function ()
{
subfun <- function() {
```

```
x * x
}
x <- 23:25
subfun()
}
R> fjj.s.r()
[1] 529 576 625
```

And the same example in S.

```
S> x <- -1:4
S> fjj.s.r
function()
{
        subfun <- function()
        {
                x * x
        }
        x <- 23:25
        subfun()
}
S> fjj.s.r()
[1]  1  0  1  4  9 16
```

The formatting of the function is different, but the significant difference is the answer. In S it is the global `x` that is used (see page 210 if you need to review). While in R the `x` that is defined in the function is used. That `x` is a mate to `subfun` because they are defined in the same environment.

9.9 Things to Do

Decide which pieces of code that you have written would be improved by making it object-oriented. Do it.

Create a function that fully checks the input file for `BATCH`.

Discover the modes that I haven't talked about, and learn what they do.

Find and fix the bugs in `find.assign` and `global.vars`.

Find some `.Internal` calls that do not evaluate their arguments. Why don't they?

The `loan` function uses a quite general name for a rather specific case. Generalize `loan` so that it is worthy of its name.

Why didn't I use functions named `pop` and `push` to implement stacks? Should I have? Is there a better rule for the location of assignment for a stack that has been popped?

Implement a class for deques.

9.10 Further Reading

Becker, Chambers and Wilks (1988) *The New S Language* contains a thorough discussion of the S language. Chambers and Hastie (1992) *Statistical Models in S* has some further discussion of S; in particular, Appendix A explains the object-orientation in S.

A discussion of basic data structures is in Knuth (1973).

You can find out about R via statlib.

9.11 Quotations

[28]David Wagoner "Looking for Mountain Beavers"

[29]Wallace Stevens "Evening Without Angels"

[30]Gwendolyn Brooks "Boy Breaking Glass"

[31]Philip Mead "The Soldier"

[32]Theodor Geisel "Fox in Sox"

[33]Gwendolyn Brooks "We Real Cool"

Chapter 10

Numbers

Here are some examples, mostly half-baked, of numerical computations.

10.1 Changing Base

It is not unusual to want to view numbers in a base other than base 10—or in more technical terms—to change radix. Binary, octal and hexadecimal are particularly common bases to use.

We'll look at an implementation of this called `numberbase`. Note that this function is limited to integers. Here it is in action:

```
> jj <- numberbase(1:20)
> jj
 [1] 1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17
[18] 18 19 20
(base 10)
> numberbase(jj, 2)
 [1] 1     10    11    100   101   110   111   1000
 [9] 1001  1010  1011  1100  1101  1110  1111  10000
[17] 10001 10010 10011 10100
(base 2)
> numberbase(jj, 8)
 [1] 1  2  3  4  5  6  7  10 11 12 13 14 15 16 17 20 21
[18] 22 23 24
(base 8)
> numberbase(jj, 16)
 [1] 1  2  3  4  5  6  7  8  9  a  b  c  d  e  f  10 11
[18] 12 13 14
(base 16)
> numberbase(c("10011101", "1101010001"), 10, old=2)
```

```
[1] 157 849
(base 10)
```

The `numberbase` function itself is a typical generic function:

```
"numberbase"<-
function(x, ...)
UseMethod("numberbase")
```

It has two methods written for it—a default method and the rather perverse idea of a method for objects of the class that it creates:

```
"numberbase.default"<-
function(x, newbase = 10, oldbase = 10)
{
        ans <- to.base10(x, oldbase)
        if(newbase != 10)
                ans <- numberbase(ans, newbase)
        ans
}

"numberbase.numberbase"<-
function(x, newbase = 10)
{
        from.base10(attr(x, "value"), newbase)
}
```

These are nice and simple, but only because they have a couple of slaves that do all of the work. The default method doubly cheats by calling the generic function again. Dividing the computations like this clarifies the scheme slightly—the function names crystalize the abstractions.

Once we look into these lower-level functions, we see how the system works.

```
"to.base10"<-
function(raw, oldbase)
{
        to.base10.sub <- function(raw, mult, digested)
        {
                ncr <- nchar(raw)
                this.raw <- substring(raw, ncr, ncr)
                this.raw <- match(this.raw, c(0:9,
                        letters, "-")) - 1
                digested <- ifelse(this.raw == 36,  -
                        digested, digested + this.raw *
                        mult)
```

```
                list(raw = substring(raw, 1, ncr - 1),
                        digested = digested)
        }
        # start of main function
        if(is.numeric(raw)) {
                if(any(abs(round(raw) - raw) >
                        .Machine$double.eps)) {
                        warning("rounding non-integers")
                }
                raw <- as.character(round(raw))
        }
        else if(!is.character(raw))
                stop(paste("can not handle data of mode",
                        mode(raw)))
        raw <- transcribe(raw, "A-Z", "a-z")
        multiple <- 1
        todo <- nchar(raw) > 0
        digested <- rep(0, length(raw))
        while(any(todo)) {
                current <- to.base10.sub(raw = raw[
                        todo], mult = multiple,
                        digested = digested[todo])
                multiple <- multiple * oldbase
                digested[todo] <- current$digested
                raw[todo] <- current$raw
                todo <- nchar(raw) > 0
        }
        ans <- as.character(digested)
        attr(ans, "value") <- digested
        attr(ans, "base") <- 10
        class(ans) <- "numberbase"
        ans
}
```

The first step is to convert numbers to character strings. Then the characters are converted to their numeric value starting with the one's place and moving to the higher value places. Elements that have the largest number of characters in their representation control the length of the `while` loop. Finally, `ans` is made into an object of class `"numberbase"`. The subfunction (called `to.base10.sub`) takes care of doing the messy details of the work within the loop. (We should worry for an instant what happens in the call to `substring` when `ncr` is 1, but it works okay.)

The structure of a numberbase object is that it is a vector of the character representation in the base of interest, and it has an attribute called `"base"` that states the base of the representation, and it has a `"value"` attribute that contains the numeric value.

`from.base10` could have been written with precisely the same structure as `to.base10`, but it is written in a different style to display features of the language. Again, let me emphasize that you should use one style throughout a project—we can expect three times as much debugging of `to.base10` and `from.base10` than if we had stuck to one style.

The subfunction in `from.base10` uses recursion, rather than being called from a loop as is the subfunction in `to.base10`.

```
"from.base10"<-
function(raw, newbase)
{
        from.base10.sub <- function(raw, newbase)
        {
                zero <- raw == 0
                if(all(zero))
                        return(character(0))
                ans <- character(length(raw))
                this.digit <- raw[!zero] %% newbase
                if(newbase > 10)
                        this.digit <- c(0:9, letters)[
                                this.digit + 1]
                ans[!zero] <- paste(Recall(raw[!zero] %/%
                        newbase, newbase), this.digit,
                        sep = "")
                ans
        }
        # start of main function
        if(length(newbase) != 1 || newbase > 36.5 ||
                newbase < 1.5 || abs(round(newbase) -
                newbase) > .Machine$double.eps)
                stop("need single integer between 2 and 36 as base"
                        )
        raw <- as.numeric(raw)
        wna <- which.na(raw)
        if(length(wna)) {
                realraw <- raw
                raw <- raw[ - wna]
        }
        if(any(abs(round(raw) - raw) > .Machine$
                double.eps))
                warning("rounding non-integers")
        raw <- round(raw)
        ans <- from.base10.sub(abs(raw), round(newbase))
        ans[nchar(ans) == 0] <- "0"
        ans[raw < 0] <- paste("-", ans[raw < 0], sep
                 = "")
```

```
        if(length(wna)) {
                realans <- character(length(realraw))
                realans[wna] <- "NA"
                realans[ - wna] <- ans
                ans <- realans
                raw <- realraw
        }
        attr(ans, "value") <- raw
        attr(ans, "base") <- newbase
        class(ans) <- "numberbase"
        ans
}
```

Except for some messing about with missing values, the main actions are the same in this function as in `to.base10` except there is no loop.

`from.base10.sub` calls itself recursively through the `Recall` function. As in all recursive functions, the first thing done in `from.base10.sub` is to return if the condition is met that means it is at the bottom of the recursion. It then does some work, and calls itself on the portion that still needs further work. If you are confused about how this works, you can put a call to `browser` just before the call to `paste` in order to investigate.

S functions may call themselves recursively by using their own name or by using `Recall`. There are two reasons to use `Recall`. If you change the function name, then you do not need to change the definition of the function if you use `Recall`—you do if you hard-code the name in the definition. The second reason is that `Recall` ensures that the function will be found. Since `from.base10.sub` is defined within another function, it wouldn't work if it called itself recursively by name since the new call wouldn't search the frame where the function is defined. (Page 210 tells the rules for object searching.)

Many problems have a very natural recursive solution that is easy for us to comprehend. Unfortunately, recursion does not tend to be natural in computer languages. If the recursion is guaranteed not to go too deep, then using recursion is good—the code is likely to be simple and easy to understand, and resources won't be strained. If the recursion is deep, then the code will be inefficient, and may break on larger problems. A recursive algorithm can always be converted to an iterative one.

The second example below illustrates what happens when recursion goes too deep in S:

```
> numberbase(.Machine$integer.max, 16)
[1] 7fffffff
(base 16)
> numberbase(.Machine$integer.max, 2)
Error: Expressions nested beyond limit (256) --
        increase limit with options(expressions=...)
```

```
        only 27 of 83 frames dumped
Dumped
> options(expressions=350)
> numberbase(.Machine$integer.max, 2)
[1] 11111111111111111111111111111111
(base 2)
```

The `expressions` option is a safety valve to keep unintentional recursion from creating an infinite loop. Since `integer.max` is the worst case, there is no reason to give up the recursive algorithm, though we may want to change the `expressions` option within the function. However, if we upgrade the functionality to include floating point numbers, we should worry about the recursion.

Here is the `print` method for numberbase objects.

```
"print.numberbase"<-
function(x, ...)
{
        xv <- as.vector(x)
        names(xv) <- names(x)
        print(xv, quote = F)
        cat(paste("(base ", attr(x, "base"), ")\n",
                sep = ""))
        invisible(x)
}
```

The most important aspects are that it follows the convention that it returns its argument invisibly, and that that argument is named `x` (see page 92). Quoting of the character strings is turned off so that the result will look like numbers and not character strings. Of course, indicating the base is a requirement. Additional arguments are allowed to come in even though they are not used.

The above functions seem to work reasonably well. However, informal testing uncovered the following behavior:

```
> numberbase(c(NA,"10011101", "1101010001"), 10, old=2)
[1] 56  157 849
(base 10)
```

This is an example of a test of "garbage" inputs (page 53)—the input is not precisely what was expected, but it is well within the realm of reason. It's decidedly impolite for missing values to transform themselves into normal looking data, so we need to fix this. A new definition for `to.base10.sub` makes it better:

```
        to.base10.sub <- function(raw, mult, digested,
```

```
                oldbase)
        {
                ncr <- nchar(raw)
                this.raw <- substring(raw, ncr, ncr)
                this.raw <- match(this.raw, c(0:9,
                        letters, "-")) - 1
                digested <- ifelse(this.raw == 36,  -
                        digested, digested + this.raw *
                        mult)
                bad <- this.raw >= oldbase & this.raw <
                        36
                digested[bad] <- NA
                list(raw = substring(raw, 1, ncr - 1),
                        digested = digested)
        }
```

One check on whether `numberbase` works or not, is to go to a base and then use the characters for that base to go back to the original base and see if anything is different.

```
"fjjchecknumberbase"<-
function(x, ...)
{
        xnb <- numberbase(x, ...)
        xbase <- attr(xnb, "base")
        xnb <- as.vector(xnb)
        for(i in 2:36) {
                this.nb <- numberbase(xnb, new = i,
                        old = xbase)
                this.back <- numberbase(as.vector(
                        this.nb), new = xbase, old = i)
                if(any(this.back != xnb))
                        stop(paste(
                                "bad conversion between bases",
                                xbase, "and", i))
                else cat("bases", xbase, "and", i,
                                "okay\n")
        }
}
```

We can test every combination of bases with a loop like:

```
> for(i in 2:36) fjjchecknumberbase(-1000:1000, new=i)
bases 2 and 2 okay
bases 2 and 3 okay
bases 2 and 4 okay
```

```
    ...
```

We don't know if it always works, but we at least know that it works on the numbers that we gave it. Given that it works for these, it is only extreme cases where trouble is likely to be.

As long as the input is numeric, we are (rather accidentally) protected from special values causing problems because an error results if we include any:

```
> numberbase(c(-10:10, NA, Inf))
Error in to.base10: Missing value where logical needed:
if(any(abs(round(raw) - raw) > .Machine$double.eps))
        . . .
Dumped
```

In summary, we can have reasonable faith that `numberbase` performs correctly, though it would be a nicety to have special values work.

10.2 Rational Numbers

I was disillusioned once my first computer program (in Fortran) gave me an answer. It was a program to do the quadratic equation, and it produced answers like 1.999998 and 5.999999. I knew very well that the correct answers were 2 and 6, so I couldn't see why a big, fancy computer couldn't know it also. The reason, of course, is that computations are typically done with floating point numbers of finite length so that small numerical errors usually occur with each operation. S hides this better than that Fortran program did because S typically computes in double precision and prints in single precision, but the problem persists.

If you want precise computations with non-integer numbers (and your problem is suitable), then you can use rational numbers where the numerator and denominator are each integers. Here we create a class of objects for rational numbers. *Do you dance, Minnaloushe, do you dance?* [34]

```
> jjr <- rationalnum(num=1, den=1:9)
> jjr
[1] 1   1/2 1/3 1/4 1/5 1/6 1/7 1/8 1/9
> jjr * 2 - 1 / jjr
[1] 1     -1    -7/3  -7/2  -23/5 -17/3 -47/7 -31/4
[9] -79/9
```

The primary function that creates rational numbers is:

```
"rationalnum"<-
function(numerator, denominator)
{
```

```
        n <- max(ln <- length(numerator), ld <- length(
                denominator))
        if(ln != n)
                numerator <- rep(numerator, length = n
                        )
        if(ld != n)
                denominator <- rep(denominator, length
                         = n)
        anam <- names(numerator)
        if(!length(anam))
                anam <- names(denominator)
        ans <- list(numerator = numerator, denominator
                 = denominator, names = anam)
        class(ans) <- "rationalnum"
        reduce.rationalnum(ans)
}
```

There are assignments of the arguments to `max`. You can put assignments virtually anywhere, but there is a catch—you need to make sure that lazy evaluation isn't going to get in the way. If the argument is not used in the call, then it will not be evaluated and hence the assignment not made.

After the function takes care of some bureaucracy, we see that the `"rationalnum"` class consists of a list of three components—the vector of numerators, the vector of denominators, and a vector of names (which may be `NULL`). But the soul of the function lies elsewhere in `reduce.rationalnum`:

```
"reduce.rationalnum"<-
function(x)
{
        signnz <- function(x)
        {
                x <- sign(x)
                x[x == 0] <- 1
                x
        }
        # start of main function
        nas <- is.na(x$numerator) | is.na(x$
                denominator)
        # take care of Inf before coercing to integer
        if(any(out <- is.infinite(x$denominator))) {
                x$denominator[out] <- 1
                x$denominator[out & is.infinite(x$
                        numerator)] <- 0
                x$numerator[out] <- 0
        }
        if(any(out <- is.infinite(x$numerator))) {
```

```
                x$numerator[out] <- sign(x$numerator[
                        out]) * signnz(x$denominator[
                        out])
                x$denominator[out] <- 0
        }
        x$numerator <- as.integer(x$numerator)
        x$denominator <- as.integer(x$denominator)
        fact <- great.common.div(x$numerator,
                x$denominator)
        x$numerator <- as.integer(x$numerator/fact *
                signnz(x$denominator))
        x$denominator <- as.integer(abs(x$denominator/
                fact))
        x$numerator[nas] <- NA
        x
}
```

The purpose of this function is to express each number in the form in which the numerator and denominator have no common factor. This again is mainly taken up with details like handling missing values and infinity. The interesting part is in `great.common.div` which uses Euclid's algorithm to compute the greatest common divisor for each numerator-denominator pair.

```
"great.common.div"<-
function(x, y)
{
        if(length(x) != (n <- length(y)))
                stop("x and y different lengths")
        out <- !is.finite(x) | !is.finite(y)
        x <- round(x)
        y <- round(y)
        okay <- y != 0
        okay[out] <- F
        ans <- z <- x
        while(any(okay)) {
                z[okay] <- x[okay] %% y[okay]
                zz <- z == 0 | is.na(z)
                ans[zz] <- y[zz]
                okay <- okay & !zz
                x[okay] <- y[okay]
                y[okay] <- z[okay]
        }
        ans <- abs(as.integer(ans))
        ans[out] <- NA
        ans
}
```

This function, like the `to.base10` function discussed earlier, uses vectorized operations and has a loop that stops when enough computing has been performed on the worst-case element.

The idea of the algorithm is that the greatest common divisor of u and v is the same as the greatest common divisor of v and u mod v. For instance, if we start with the pair $(8, 12)$, then 8 mod 12 is 8, so we have the pair $(12, 8)$. Next, 12 mod $8 = 4$ to get the pair $(8, 4)$. This leads to the pair $(4, 0)$. Because the v is zero, we know we are done and the answer is 4.

Knuth (1981, p 316) wants the greatest common denominator of 0 with 0 to be 0, and the greatest common denominator of u with 0 to be the absolute value of u.

```
> great.common.div(c(-4:4), rep(0,9))
[1] 4 3 2 1 0 1 2 3 4
```

Here is a more typical problem:

```
> great.common.div(1:100, rep(12, 100))
  [1]  1  2  3  4  1  6  1  4  3  2  1 12  1  2  3  4
 [17]  1  6  1  4  3  2  1 12  1  2  3  4  1  6  1  4
 [33]  3  2  1 12  1  2  3  4  1  6  1  4  3  2  1 12
 [49]  1  2  3  4  1  6  1  4  3  2  1 12  1  2  3  4
 [65]  1  6  1  4  3  2  1 12  1  2  3  4  1  6  1  4
 [81]  3  2  1 12  1  2  3  4  1  6  1  4  3  2  1 12
 [97]  1  2  3  4
```

The print method for the rational number class is defined as:

```
> print.rationalnum
function(x, ...)
{
        out <- paste(x$numerator, "/", x$denominator,
                sep = "")
        names(out) <- x$names
        xna <- is.na(x$numerator) | is.na(x$
                denominator)
        d1 <- x$denominator == 1
        out[d1] <- x$numerator[d1]
        out[xna] <- "NA"
        dz <- x$denominator == 0
        nz <- x$numerator == 0
        out[!xna & nz & !dz] <- "0"
        out[!xna & nz & dz] <- "NA"
        out[!xna & x$numerator < 0 & dz] <- "-Inf"
        out[!xna & x$numerator > 0 & dz] <- "Inf"
        print(out, quote = F, ...)
```

```
        invisible(x)
}
```

The first line and the last two lines are really what the function is about—paste together a character string of the right thing, print it without quotes and return the original invisibly. All of the rest is details for missing values and such.

Several functions use `as.rationalnum` to make sure that an object is of class `"rationalnum"` when appropriate. Here is the default method for this function:

```
"as.rationalnum.default"<-
function(x)
{
        if(inherits(x, "rationalnum"))
                return(x)
        switch(mode(x),
                numeric = {
                        ans <- x
                        ans[] <- NA
                        good <- !is.na(x)
                        x <- x[good]
                        ints <- x == ceiling(x)
                        if(count <- sum(!ints))
                                warning(paste(count,
                                  "NA(s) created coercing to rationalnum"
                                  ))
                        ans[good][ints] <- x[ints]
                        rationalnum(ans, 1)
                }
                ,
                stop(paste("can not coerce to rationalnum from mode",
                        mode(x))))
}
```

Note that there are no other methods for this function, so we could have made it an ordinary function. However, it is believable that other methods could be desired. We are opening the door to unforeseen uses at the expense of only a couple dozen extra keystrokes. Since we are making it generic, we can eke out a little more efficiency by deleting the first two lines, and creating a method of `as.rationalnum` for class `"rationalnum"`.

Coercion to rational numbers is not easy, and in consequence the function doesn't really do much. If we already have rational numbers, great. Otherwise, we can convert integers, but nothing else.

Now we come to the arithmetic operators. Here is addition:

```
"+.rationalnum"<-
function(e1, e2)
{
```

```
        if(missing(e2))
                return(e1)
        e1 <- as.rationalnum(e1)
        e2 <- as.rationalnum(e2)
        # propagate names, let default ops do the details
        num1 <- e1$numerator
        names(num1) <- names(e1)
        den2 <- e2$denominator
        names(den2) <- names(e2)
        rationalnum(num1 * den2 + e2$numerator * e1$
                denominator, e1$denominator * den2)
}
```

The first thing is to see if this is a unary operator, in which case we don't need to do anything.

Next, do what seems to be unnecessary—coerce vectors to be rational numbers. Why would we be inside a method for rational numbers if we don't have rational numbers to begin with? Well, there is the perverse way—the method is called directly:

```
"+.rationalnum"(1, 4)
```

but if a user does that, then they can get what they deserve. The real reason we care is that we will end up in this method if only one of the two arguments has class `"rationalnum"`. We need to make sure that both are of this class before we proceed. To make something more than a toy, we would want the coercion to tend to go the other way—coerce rational to numeric and then proceed (but if the non-rational numbers can be coerced to rational, then we would want to do that). It can be a deep problem of what to do when the two arguments to a binary operator are of different types.

Then we spend a few lines arranging for the names of the rational numbers to end up in the result. As the comment says, we let the usual operators do all of the work. Finally, we plug what we need into `rationalnum` which will take care of reducing all of the fractions.

The minus operator is decidedly unlike the addition operator. This time the unary operator does something, and the binary operator cheats its way out.

```
"-.rationalnum"<-
function(e1, e2)
{
        e1 <- as.rationalnum(e1)
        if(missing(e2)) {
                e1$numerator <-  - e1$numerator
                return(e1)
        }
        e1 + ( - e2)
```

```
}
```

The multiplication and division operators are very similar to addition, except that they do not allow the possibility of a unary operation.

Now we delve into the housekeeping functions:

```
"names.rationalnum"<-
function(x)
x$names
```

From this we can see that "names" is confounded. We have the names of the numbers, which is what is contained in the `names` component that the above function returns; and there are the names of the list that is the implementation of the rational number class.

```
> names(jjr)
 [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l"
[13] "m" "n" "o" "p" "q" "r" "s" "t" "u" "v" "w" "x"
[25] "y" "z"
> names(unclass(jjr))
[1] "numerator"   "denominator" "names"
```

We need to keep these two straight.

The `names` function has an assignment form, and we can write a method for this also:

```
"names<-.rationalnum"<-
function(x, value)
{
        value <- as.character(value)
        if(length(value) != length(x))
                stop("bad length for names")
        x$names <- value
        x
}
```

This is the typical form of an assignment function. One argument—often named `x`—that is the object being changed, and a second argument named `value` which is the value given to the appropriate part of `x` (the object inside the parentheses to the left of the assignment operator). The return value of the function is `x`. From what you have seen, you should be upset by this definition—it presupposes that there is a `length.rationalnum` (which is given below).

```
> jjr <- rationalnum(num=1, den=1:26)
> jjr
```

```
 [1] 1    1/2  1/3  1/4  1/5  1/6  1/7  1/8  1/9  1/10
[11] 1/11 1/12 1/13 1/14 1/15 1/16 1/17 1/18 1/19 1/20
[21] 1/21 1/22 1/23 1/24 1/25 1/26
> jjr2 <- jjr
> names(jjr) <- names(jjr2) <- letters
> jjr
 a   b   c   d   e   f   g   h   i    j    k    l
 1 1/2 1/3 1/4 1/5 1/6 1/7 1/8 1/9 1/10 1/11 1/12
    m    n    o    p    q    r    s    t    u    v
 1/13 1/14 1/15 1/16 1/17 1/18 1/19 1/20 1/21 1/22
    w    x    y    z
 1/23 1/24 1/25 1/26
```

As this example shows, S is smart about making the right thing happen. From the definition of `names<-.rationalnum` it seems that we would be trying to assign the whole of `jjr2` to the names of `jjr`, but that is not the case.

We would like the length of a rational number object to be the number of numbers represented by the object, as with floating point objects.

```
"length.rationalnum"<-
function(x)
length(x$numerator)
```

An assignment form is also possible:

```
"length<-.rationalnum"<-
function(x, value)
{
        length(x$numerator) <- value
        length(x$denominator) <- value
        if(length(x$names))
                length(x$names) <- value
        x
}
```

DANGER. Changing the meaning of `length`—and to a lesser extent `names`—should be undertaken only with sound reason and with some trepidation. If all of the holes aren't filled, there is room for disaster.

We need to know which values are missing, so we need a method for the `is.na` function:

```
"is.na.rationalnum"<-
```

```
function(x)
{
        is.na(x$numerator) | is.na(x$denominator) | is.nan(x)
}
```

This in turn needs to know about NaN's:

```
"is.nan.rationalnum"<-
function(x)
{
        ans <- x$numerator == 0 & x$denominator == 0
        # both infinity should not occur
        ans[is.na(ans)] <- F
        names(ans) <- x$names
        ans
}
```

Subscripting is extremely important functionality—here is a method for that:

```
"[.rationalnum"<-
function(x, i)
{
        if(is.character(i))
                i <- pmatch(i, x$names, dup = T)
        x$numerator <- x$numerator[i]
        x$denominator <- x$denominator[i]
        x$names <- x$names[i]
        x
}
```

Notice that we have to handle character subscripts on our own since the names are not as S expects. The assignment form uses a safer approach by putting the names onto the numerator and denominator, and then depending entirely on the default subscripting behavior:

```
"[<-.rationalnum"<-
function(x, i, value)
{
        value <- as.rationalnum(value)
        den <- x$denominator
        num <- x$numerator
        names(den) <- names(num) <- x$names
        den[i] <- value$denominator
        num[i] <- value$numerator
        x$denominator <- den
        x$numerator <- num
```

```
        x$names <- names(den)
        x
}
```

We, of course, want to be able to change from rational numbers to numeric vectors. Here's how:

```
"as.numeric.rationalnum"<-
function(x)
{
        ans <- as.vector(x$numerator/x$denominator)
        names(ans) <- names(x)
        ans
}
```

A method for `c` is often quite useful. It presents the problem that it should allow an arbitrary number of arguments.

```
"c.rationalnum"<-
function(...)
{
        dots <- list(...)
        num <- den <- nam <- NULL
        has.names <- F
        for(i in seq(along = dots)) {
                this.rat <- as.rationalnum(dots[[i]])
                num <- c(num, this.rat$numerator)
                den <- c(den, this.rat$denominator)
                this.nam <- this.rat$names
                if(length(this.nam))
                        has.names <- T
                else this.nam <- rep("", length(
                                this.rat))
                nam <- c(nam, this.nam)
        }
        ans <- list(numerator = num, denominator = den
                )
        if(has.names)
                ans$names <- nam
        class(ans) <- "rationalnum"
        ans
}
```

This uses the idiom of concatenating within a loop that I claimed earlier you should not use. With some work we could avoid it, but I can't think of an alternative that is very appetizing. Furthermore it seems extremely rare that

the loop would be very long. The main complication in this function is that some but not all of the arguments may have names. The bother with `has.names` is to keep the names lined up properly.

The following statement shows that we can stand to improve `as.rationalnum`:

```
> c(jj3, NA, jj3)
Error in switch(mode(x),: can not coerce to rationalnum
        from mode logical
Dumped
```

The provinces of his body revolted [35]

A Math group method for rational numbers is useful:

```
"Math.rationalnum"<-
function(x, ...)
{
        warning("coercing rational numbers to numeric")
        x <- as.numeric(x)
        NextMethod(.Generic)
}
```

The Math group consists of a number of functions like `sin` and `exp`. Most of these are not closed under rational numbers, that is, if a rational number is given as an argument, then the exact answer need not be a rational number. So a logical thing to do for these functions is to coerce the rational numbers to numeric and then use the usual function.

The `.Generic` object is a character string that gives the name of the generic function that was called. So `Math.rationalnum` does three simple things: warn that coercion is taking place, do the coercion, call the next (default) method for the function. Here is an example:

```
 jjr2 <- rationalnum(-5:6, 12)
> exp(jjr2)
 [1] 0.6592406 0.7165313 0.7788008 0.8464817 0.9200444
 [6] 1.0000000 1.0869040 1.1813604 1.2840254 1.3956124
[11] 1.5168968 1.6487213
Warning messages:
  coercing rational numbers to numeric in: Math.rati\
        onalnum(jjr2)
> abs(jjr2)
 [1] 0.41666667 0.33333333 0.25000000 0.16666667
 [5] 0.08333333 0.00000000 0.08333333 0.16666667
 [9] 0.25000000 0.33333333 0.41666667 0.50000000
Warning messages:
  coercing rational numbers to numeric in: Math.rati\
        onalnum(jjr2)
```

We have gone too far—there is no need to coerce to numeric to get the absolute value. The solution is to write a method for `abs`—this specific method will take precedence over the group method.

```
"abs.rationalnum"<-
function(x)
{
        x$numerator <- abs(x$numerator)
        x$denominator <- abs(x$denominator)
        x
}
```

After this is defined, we get:

```
> abs(jjr2)
 [1] 5/12 1/3  1/4  1/6  1/12 0    1/12 1/6  1/4  1/3
[11] 5/12 1/2
```

The default method for `unique` is implemented as selecting those elements that are not duplicated. In writing a method of `unique` for rational numbers, our trick of doing the same operation on numerator and denominator is not going to work. The following is a cheap way out; to match the default method, we would need to treat incomparables also.

```
"unique.rationalnum"<-
function(x)
{
        xchar <- paste(x$numerator, "/", x$denominator
                )
        x$names <- NULL
        x[!duplicated(xchar)]
}
```

With rational numbers we have the opportunity to get very precise answers that are wrong. Checking the veracity of the computations is an important part of the programming. Here is a function that helps with that job.

```
"fjjcheckrationalnum"<-
function(num = c( - Inf, -10, -7, -4, -2, -1, 0, 1, 2,
        4, 7, 10, Inf, NA, 0/0), den = num, test = T)
{
        anum <- expand.grid(num, den)
        ratnum <- rationalnum(anum[, 1], anum[, 2])
        if(test) {
                all.equal(anum[, 1]/anum[, 2], ratnum$
                        num/ratnum$den)
```

```
        }
        else ratnum
}
```

The `expand.grid` call gives us a data frame containing every combination of numbers in `num` with numbers in `den`. The function then either tests the value of the rational numbers versus their true value, or returns the rational numbers. The default value for `num` gives all of the special values plus 0, 1, 2, a prime number and some composite numbers.

```
> fjjcheckrationalnum()
[1] T
> jjrt <- fjjcheckrationalnum(test=F)
> length(jjrt)
[1] 225
> class(jjrt)
[1] "rationalnum"
```

The test passes, and then we capture the vector of rational numbers for further testing.

This is a function to test operations on rational numbers:

```
"fjjcheckratop"<-
function(op, r1, r2, test = T)
{
        unary <- missing(r2)
        n1 <- as.numeric(r1)
        if(unary) {
                ra <- eval(parse(text = paste(op,
                        deparse(substitute(r1)))))
                nn <- get(op)(n1)
        }
        else {
                n2 <- as.numeric(r2)
                ra <- eval(parse(text = paste(deparse(
                        substitute(r1)), op, deparse(
                        substitute(r2)))))
                nn <- get(op)(n1, n2)
        }
        if(test)
                all.equal(nn, as.numeric(ra))
        else cbind(nn, as.numeric(ra))
}
```

This has the same sort of form: do the computation with both rational numbers and numeric vectors, and either compare them to see if they are all the same, or return the numbers.

```
> fjjcheckratop("+", jjrt, jjrt)
[1] "Missing value mismatches: 93 in current, 61  in target"
> jjrplus <- fjjcheckratop("+", jjrt, jjrt, F)
```

Now we see trouble. The second call gives us the numbers so that we can investigate. The trouble boils down to:

```
> jjri <- rationalnum(Inf, 1)
> jjri
[1] Inf
> jjri + jjri
[1] NA
```

The problem is that infinity is represented as 1/0 but it would be better if it were $\infty/1$. This led to a rewrite of `reduce.rationalnum`. The revised version is:

```
"reduce.rationalnum"<-
function(x)
{
        signnz <- function(x)
        {
                x <- sign(x)
                x[x == 0] <- 1
                x
        }
# start of main function
        num <- round(x$numerator)
        den <- round(x$denominator)
        nans <- ((num == 0 & den == 0) | (is.inf(num) &
                is.inf(den)))
        nans[is.na(nans)] <- F
        nans[is.nan(num) | is.nan(den)] <- T
        nas <- (is.na(num) | is.na(den)) & !nans
        infs <- ((is.inf(num) & !is.inf(den)) | (num !=
                0 & den == 0)) & !nas & !nans
        zeros <- is.inf(den) & !nans & !nas
        out <- nas | nans | infs | zeros
        fact <- great.common.div(num[!out], den[!out])
        x$numerator[!out] <- as.integer(num[!out]/fact *
                signnz(den[!out]))
        x$denominator[!out] <- as.integer(abs(den[!out
                ]/fact))
        x$numerator[nans] <- x$denominator[nans] <-
                as.integer(0)
        x$numerator[infs] <- (Inf * signnz(den[infs]) *
```

```
                signnz(num[infs]))
        x$denominator[infs | zeros] <- 1
        x$numerator[zeros] <- 0
        x$numerator[nas] <- NA
        x
}
```

I like this version much better. That it works correctly is of course a plus, but it is more firmly rooted in the problem. The first version tried to get by with minimal attention to the weird cases—as a result it had code that was hard to understand (as well as being wrong). This version accepts that there are special cases to be addressed, and attacks them head on. Thankfully the bug has saved us from staying with the original version.

You will notice that I'm fairly liberal in my use of parentheses in logical statements—it's easier this way to be sure that S and I are both thinking the same.

10.3 Polygamma

The digamma function, often denoted $\psi(x)$, is defined as

$$\psi(x) = \frac{d}{dx} log_e \Gamma(x) \tag{10.1}$$

That is, this is the first derivative of the `lgamma` function.

DANGER. Some people use $\Gamma(x+1)$ instead of $\Gamma(x)$. There is an easy translation between the two forms. However, this difference is subtle enough to give you headaches if you are checking your answers, and wrong answers if you are not.

Each polygamma function is a derivative of $\psi(x)$ of some order. The trigamma function is the first derivative, the tetragamma function is the second derivative, and so on.

Our `polygamma` S function specifies the order of the derivative of ψ via either a number or a character string.

```
> polygamma(1:5, 1)
[1] 1.6449341 0.6449341 0.3949341 0.2838230 0.2213230
> polygamma(1:5, "trigam")
[1] 1.6449341 0.6449341 0.3949341 0.2838230 0.2213230
> polygamma(1, 1:4)
[1]   1.644934  -2.404114   6.493939 -24.886266
```

The first two commands are the same—they merely use different conventions for indicating which polygamma function to compute. The last call is for a single x value at four different orders of the derivative.

Here is the S code:

```
"polygamma"<-
function(x, n, low = 0.0001, high = 100, terms = 5)
{
        if(!length(x))
                return(x)
        if(is.character(n)) {
                n <- pmatch(n, c("trigamma",
                        "tetragamma", "pentagamma",
                        "hexagamma"), dup=T)
                if(any(is.na(n)))
                        stop("unknown or ambiguous name")
        }
        else {
                n <- round(n)
                if(any(n < 1))
                        stop("n must be a positive integer")
        }
        terms[terms > 30] <- 30
        xatt <- attributes(x)
        x <- as.numeric(x)
        if(any(x <= 0, na.rm = T))
                stop("only positive values allowed")
        alldat <- cbind(x = x, n = n, low = low, high
                 = high, terms = terms)
        xna <- is.na(alldat[, "x"])
        alldat[xna, "x"] <- 0.5 * alldat[xna, "low"]
        if(!is.loaded(symbol.C("polygamma_Sp")))
                poet.dyn.load("polygamma.o")
        ans <- .C("polygamma_Sp",
                as.double(alldat[, "x"]),
                as.integer(dim(alldat)[1]),
                as.integer(alldat[, "n"]),
                as.double(alldat[, "low"]),
                as.double(alldat[, "high"]),
                as.integer(alldat[, "terms"]),
                as.double(gamma(alldat[, "n"] + 1)))[[
                1]]
        ans[xna] <- NA
        if(length(ans) == length(x))
                attributes(ans) <- xatt
        ans
```

```
}
```

Here are the steps that the function performs. If the length of the input is zero, then just return it—it is often much cleaner when functions work with zero length inputs. If `n` is given as character, then convert to the numeric order of the derivative. Save the attributes of the input `x` so that they can be given to the output. Test the validity of the input, and scream if it isn't right. The call to `cbind` is a kludge of significance. We want the function to be vectorized in all of the arguments, so we use `cbind` to do all of the necessary replications and issuing of warnings in a single line of our code. We're not interested in the matrix that results—we only want everyone to be the same length. Another kludge follows in which the locations of missing values are noted, and then replaced by an arbitrary number—we'll put the missing values back in the end. Finally we call C where the real work is done, and then clean up the result before returning.

Below is the C code that computes the values. This is listed in a logical order, but note that some compilers insist that static objects be defined in a file before they are used. Such compilers would be upset with the order of these functions.

```
#include <math.h>

void
polygamma_Sp(x, len, n, low, high, terms, nfact)
long *len, *n, *terms;
double *x, *low, *high, *nfact;
{
        long i;
        double polygamma();

        for(i=0; i < *len; i++) {
                x[i] = polygamma(x[i], n[i], low[i], high[i],
                        terms[i], nfact[i]);
        }
}


/* Bernoulli numbers of even order from 2 to 60 */
static double
bernou[30] = {1.0/6.0, -1.0/30.0, 1.0/42.0, -1.0/30.0, 5.0/66.0,
        -691.0/2730.0, 7.0/6.0, -3617.0/510.0, 43867.0/798.0,
        -174611.0/330.0, 854513.0/138.0, -236364091.0/2730.0,
        8553103.0/6.0, -23749461029.0/870.0, 8615841276005.0/14322.0,
        -7709321041217.0/510.0, 2577687858367.0/6.0,
        -1.371165521e13, 4.883323190e14, -1.929657934e16,
        8.416930476e17, -4.033807185e19, 2.115074864e21,
        -1.208662652e23, 7.500866746e24, -5.038778101e26,
        3.652877648e28, -2.849876930e30, 2.386542750e32,
        -2.139994926e34};
```

```
static double
polygamma(x, n, low, high, terms, nfact)
long n, terms;
double x, low, high, nfact;
{
        /*
         * polygamma function of a real positive x
         * no checks are made here on the suitability
         * of arguments
         */

        long i;
        double asign, ans = 0.0, nd = (double) n, nexp, ser = 0.0;
        double t0, x2_inv;

        asign = (n % 2) ? 1.0 : -1.0;

        if(x < low) {
                return(asign * nfact / nd * pow(x, - nd) *
                        (1.0 + nd * .5 / x));
        }

        nexp = - nd - 1.0;
        while(x < high) {
                ans = ans + asign * nfact * pow(x, nexp);
                x = x + 1.0;
        }

        t0 = nfact / nd * pow(x, - nd);
        ser = t0 * ( 1.0 + nd * .5 / x);
        x2_inv = pow(x, -2.0);
        for(i=0; i < terms; i++) {
                if(n ==1) {
                        t0 = t0 * x2_inv;
                } else {
                        t0 = (2.0 * i + nd + 3.0) / (2.0 * i + 4.0) *
                                (2.0 * i + nd + 2.0) / (2.0 * i + 3.0) *
                                t0 * x2_inv;
                }
                ser = ser + bernou[i] * t0;
        }

        ans = ans + asign * ser;
        return(ans);
}
```

`polygamma_Sp` is called from S—it merely vectorizes the call to the real computing function. Here's the idea of the algorithm that the `polygamma` C

function uses. If x is close to zero, then we have a good approximation. If x is large, then we have a good approximation. We have a formula for the function at x in terms of the function at $x + 1$. If x is neither big nor small, then use the recurrence formula n times where $x + n$ is larger than the lower bound for "big".

Checking that we are getting correct answers is a little problematic. We can achieve some measure of comfort by seeing that our answers match some of those listed in Abramowitz and Stegun (1964). All of the results of the following function should be close to zero.

```
> fjjcheckpolygam
function()
{
        x <- c(1.095, 1.92, 1.11, 1.11, 1.98, 1.98)
        n <- c(1, 1, 2, 3, 2, 3)
        pg <- c(1.4426631755, 0.6789231293,
                -1.8170975731, 4.3602088083,
                -0.4141726631, 0.5120891127)
        polygamma(x, n) - pg
}
> fjjcheckpolygam()
[1]  1.090934e-10  3.076084e-11  2.168104e-09
[4] -1.319442e-10  2.126171e-09 -9.041279e-11
```

In general this looks good, but the values for the tetragamma (third and fifth) are a little worrisome. Let's revise the function so that we can control the computations and investigate further.

```
> fjjcheckpolygam
function(...)
{
        x <- c(1.095, 1.92, 1.11, 1.11, 1.98, 1.98)
        n <- c(1, 1, 2, 3, 2, 3)
        pg <- c(1.4426631755, 0.6789231293,
                -1.8170975731, 4.3602088083,
                -0.4141726631, 0.5120891127)
        polygamma(x, n, ...) - pg
}
> fjjcheckpolygam(terms=9)
[1]  1.090934e-10  3.076084e-11  2.168104e-09
[4] -1.319442e-10  2.126171e-09 -9.041279e-11
> fjjcheckpolygam(high=200)
[1]  1.090941e-10  3.076095e-11  9.430390e-11
[4] -1.956035e-11  1.252483e-10  1.713751e-11
```

Increasing the number of terms has little effect, but changing `high` does give us the accuracy contained in the table.

Another check is to see if the multiplication formula works. The formula is:

$$\psi^{(n)}(mx) = \frac{1}{m^{n+1}} \sum_{k=0}^{m-1} \psi^{(n)}\left(x + \frac{k}{m}\right) \tag{10.2}$$

The following function vectorizes this with the help of the `outer` function.

```
> fjjpolygammult
function(x, n, mult = 2, ...)
{
        if(length(n) != 1 || length(mult) != 1)
                stop("bad input")
        partials <- polygamma(outer(x, (0:(mult - 1))/
                mult, "+"), n)
        partials <- partials %*% rep(mult^( - (n + 1)),
                mult)
        drop((polygamma(x * mult, n, ...) - partials)/
                abs(partials))
}
```

One side of the equation is subtracted from the other so that the result should be near zero, and the difference is normalized so we get the relative difference. The "bad input" error message would be too terse if this were a function that would be used much. But this is a private function that is just for verifying, so just making sure that we use it properly is good enough.

Here is how we do:

```
> fjjpolygammult(c(.5, 5, 10, 100, 1000, 1e6), 1)
[1] -4.049608e-16 -6.598018e-16  5.413523e-16
[4] -1.730390e-16  0.000000e+00  2.117582e-16
> fjjpolygammult(c(.5, 5, 10, 100, 1000, 1e6), 2)
[1]  6.955094e-10  1.513223e-07  6.362287e-07
[4] -1.636471e-05 -1.663615e-07 -1.668093e-13
> fjjpolygammult(c(.5, 5, 10, 100, 1000, 1e6), 3)
[1] -1.574678e-11 -4.407812e-08 -3.796090e-07
[4]  4.270955e-05  4.364444e-07  4.377635e-13
> fjjpolygammult(c(.5, 5, 10, 100, 1000, 1e6), 20)
[1]  2.104482e-16  0.000000e+00  1.124138e-16
[4] -1.860500e-03 -2.072273e-05 -2.097891e-11
```

We can do a transformation to see about how many significant digits we get:

```
> round(-log10(abs(fjjpolygammult(c(0.5, 5, 10, 100,
+ 1000, 1e6), 2, high=900))), 1)
[1]  9.6  7.3  6.7  4.7  6.8 12.8
```

Values near 100 seem to be the most troublesome. The following examines the effect of changes in the `high` argument for tetragamma of 100:

```
> print(as.matrix(polygamma(100, 2, high=c(50,100, 200,
+       400, 1000, 5000, 1e5))), digits=15)
                     [,1]
[1,] -0.000101002777700007
[2,] -0.000101002777700007
[3,] -0.000101004860945850
[4,] -0.000101004991152816
[5,] -0.000101004999611128
[6,] -0.000101004999832995
[7,] -0.000101004999833349
```

The differences in these values shows that it is probably too much to hope for to have one single choice for the parameters that control the algorithm. This does, however, show the value of S to investigate the quality of the computations.

10.4 Digamma

The digamma function is more useful and simpler than the polygamma, but it comes afterwards because complex as well as numeric arguments are allowed in the S implementation.

```
> digamma(-2:5)
[1]         NA         NA         NA -0.5772157
[5]  0.4227843  0.9227843  1.2561177  1.5061177
> digamma(complex(re=0:4, im=4:0))
[1] 1.3915363+1.6957963i 1.1079807+1.4041297i
[3] 0.9145915+0.9208073i 0.9946503+0.3766740i
[5] 1.2561177+0.0000000i
```

```
"digamma"<-
function(x)
{
        if(!is.loaded(symbol.C("digamma_real_Sp")))
                poet.dyn.load("digamma.o")
        if(is.complex(x)) {
                n <- length(x)
                if(!n)
                        return(x)
                ans <- .C("digamma_complex_Sp",
                        NAOK = T,
                        specialsok = T,
```

```
                        z = as.complex(x),
                        as.integer(n),
                        ans.almost = complex(n))
                ans <- ans$ans.almost + log(ans$z)
                attributes(ans) <- attributes(x)
                ans
        }
        else {
                storage.mode(x) <- "double"
                .C("digamma_real_Sp",
                        NAOK = T,
                        specialsok = T,
                        x,
                        as.integer(length(x)))[[1]]
        }
}
```

This function contains only a test of the mode of the input to decide which of two C functions to use.

Here's the C code:

```
#include <math.h>
#include <S.h>

void
digamma_real_Sp(x, n)
long *n;
double *x;
{
        long i;
        double digamma_real_pos();

        for(i=0; i < *n; i++) {
                if(is_na(x + i, DOUBLE)) ;
                else if(x[i] <= 0.0) na_set3(x + i, DOUBLE, Is_NaN);
                else if(is_inf(x + i, DOUBLE)) inf_set(x + i, DOUBLE, 1);
                else x[i] = digamma_real_pos(x[i]);
        }
}


static double stirling[] = {
                -8.333333333333333e-02,  8.333333333333333e-03,
                -3.968253968253968e-03,  4.166666666666667e-03,
                -7.575757575757576e-03,  2.109279609279609e-02,
                -8.333333333333334e-02,  4.432598039215686e-01,
                -3.053954330270120e+00,  2.645621212121212e+01,
                -2.814601449275362e+02,  3.607510546398047e+03
```

```
};


static double
digamma_real_pos(x)
double x;
{
        long i;
        double lower, upper, euler_one, ans, x_inv, x_pow;

        lower = 1.0e-8;
        upper = 19.5;
        /* euler = -.57721566490153286060606512; */
        euler_one = .42278433509846713939393488;

        /* expects x to be positive and finite - NO CHECKS HERE */
        if(x < lower) {
                ans = - 1.0 / x - 1.0 / (1.0 + x) + euler_one;
                return(ans);
        }

        ans = 0.0;
        while(x < upper) {
                ans = ans - 1.0 / x;
                x = x + 1.0;
        }

        x_inv = 1.0 / x;
        ans = ans + log(x) - .5 * x_inv;

        x_inv = x_inv * x_inv;
        x_pow = x_inv;

        for(i=0; i < 12; i++) {
                ans = ans + stirling[i] * x_pow;
                x_pow = x_pow * x_inv;
        }
        return(ans);
}



void
digamma_complex_Sp(z, n, ans)
long *n;
complex z[], ans[];
/* the usual shortcut of *z for z[] is not a good idea here */
{
        long j, i;
        double upper;
```

```
        complex z_inv, z_pow;
        complex mult_complex(), inverse_complex();

        upper = 19.5;

        for(j=0; j < *n; j++) {
                if(is_na(z + j, COMPLEX)) {
                        continue;
                }
                if(is_inf(z + j, COMPLEX)) {
                        na_set3(ans + j, COMPLEX, Is_NaN);
                        continue;
                }
                ans[j].re = 0.0; ans[j].im = 0.0;
                while(z[j].re < upper) {
                        z_inv = inverse_complex(z[j]);
                        ans[j].re = ans[j].re - z_inv.re;
                        ans[j].im = ans[j].im - z_inv.im;
                        z[j].re = z[j].re + 1.0;
                }

                z_inv = inverse_complex(z[j]);
                ans[j].re = ans[j].re - .5 * z_inv.re;
                ans[j].im = ans[j].im - .5 * z_inv.im;

                z_inv = mult_complex(z_inv, z_inv);
                z_pow = z_inv;

                for(i=0; i < 12; i++) {
                        ans[j].re = ans[j].re + stirling[i] * z_pow.re;
                        ans[j].im = ans[j].im + stirling[i] * z_pow.im;
                        z_pow = mult_complex(z_pow, z_inv);
                }
                /* ans still needs log(z) added to it */
        }
}


static complex
mult_complex(x, y)
complex x, y;
{
        complex ans;

        ans.re = x.re * y.re - x.im * y.im;
        ans.im = x.im * y.re + x.re * y.im;
        return(ans);
}
```

```
static complex
inverse_complex(x)
complex x;
{
        complex ans;
        double den;

        den = x.im * x.im + x.re * x.re;

        ans.re = x.re / den;
        ans.im = - x.im / den;
        return(ans);
}
```

The computing algorithms are analogous to the polygamma algorithm. In this case, though, the parameters are hard-coded in C. Another difference is that the complex version has no lower cutoff—it iterates until the real part is large enough. Hence it will be slow to compute values for numbers with large negative real parts.

The C routines called by S take care of special values with the macros that live in `S.h`. Complex arithmetic operators are not available to us, so we need to make our own. To perform this algorithm, all we need is complex multiplication and inversion. We need a complex logarithm also, but that can be done at the end in S.

This code provides an example of how to test and set special values in C code written for S. Note that the `.C` call needs additional arguments so that special values can be passed into C. An introduction to dealing with special values in C is on page 175.

The comment in `digamma_real_pos` about checks on appropriate inputs is to help ensure that good code will be written when this routine is stolen for another purpose.

Here is a test of garbage input:

```
> digamma(letters)
 [1] NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA
[18] NA NA NA NA NA NA NA NA NA
Warning messages:
  26 missing values generated coercing from character
        to numeric in: storage.mode(x) <- "double"
```

This is not such a bad thing to happen—it doesn't seem worth creating an error for this.

As with the polygamma functions, it is a non-trivial task to verify that our computations are correct. One thing we know for all real numbers is that the complex version should match the real version:

```
> Mod(digamma(10:20) - digamma(as.complex(10:20)))
 [1] 0.000000e+00 0.000000e+00 0.000000e+00
 [4] 0.000000e+00 0.000000e+00 0.000000e+00
 [7] 4.440892e-16 4.440892e-16 0.000000e+00
[10] 0.000000e+00 0.000000e+00
```

This is not an especially good test to see if we get the right answers (since both algorithms could be wrong), but it is a good test to make sure that we haven't made a blunder in one of the C functions—the complex arithmetic would have been a likely spot.

Here is a function that returns the difference between the result of `digamma` and values from a table in Abramowitz and Stegun (1964).

```
> fjjcheckdigam
function()
{
        x <- c(1, 1.055, 1.965, 2, 56, 100)
        digx <- c(-0.57721566490153, -0.4902094448,
                0.3999605371, 0.4227843351,
                4.016396547, 4.6001618527)
        digamma(x) - digx
}
> fjjcheckdigam()
[1] -3.330669e-15 -1.574552e-11  2.545408e-12
[4] -1.533662e-12  2.455547e-11  3.808776e-11
```

We only have 10 digits from Abramowitz and Stegun except for the first value which is Euler's constant, so the test passes. Here is a similar test with complex numbers:

```
> fjjcheckdigamcom
function()
{
        z <- c(2+10i, 1.4+5.9i, 1+8.4i)
        digamz <- c(2.31332+1.42179i, 1.78533+1.41907i,
                2.1294144191+1.51127i)
        digamma(z) - digamz
}
> fjjcheckdigamcom()
[1]  2.537704e-07-3.574195e-06i
[2]  2.306153e-06-1.216597e-06i
[3] -2.293676e-11+2.517271e-06i
```

This also gives as good of results as we can expect.

There is a duplication formula for the digamma. We can use it to reassure ourselves. The equation is

$$\psi(2z) = \frac{1}{2}\psi(z) + \frac{1}{2}\psi\left(z + \frac{1}{2}\right) + \log(2) \tag{10.3}$$

Once again in the test function, we subtract one side of the equation from the other so that the result should be near zero:

```
> fjjdigamdup
function(z)
{
        Mod(0.5 * digamma(z) + 0.5 * digamma(z + 0.5) +
                log(2) - digamma(2 * z))
}
> fjjdigamdup(complex(re=1:10, im=2:11))
 [1] 3.140185e-16 2.220446e-16 4.440892e-16
 [4] 8.950904e-16 0.000000e+00 1.110223e-16
 [7] 1.110223e-16 1.110223e-16 0.000000e+00
[10] 4.440892e-16
```

We can also use the reflection formula as a test. The equation is

$$\psi(1 - z) = \psi(z) + \pi \cot(\pi z) \tag{10.4}$$

and the test function is:

```
> fjjdigamreflect
function(z)
{
        Mod(digamma(z) + pi/tan(pi * z) - digamma(1 -
                z))
}
> fjjdigamreflect(jj <- complex(re=rnorm(10, sd=1000),
+       im=rnorm(10, sd=1000)))
 [1] 8.881784e-16 1.776357e-15 9.930137e-16
 [4] 2.220446e-16 2.220446e-16 1.110223e-16
 [7] 9.155134e-16 2.081668e-16 9.930137e-16
[10] 1.110223e-15
```

So far we haven't found a reason to disbelieve `digamma`.

10.5 Continued Fractions

Continued fractions are a way of approximating a function. They often represent an alternative to a series approximation to the function. When both types of

approximation exist for a function, it can be the case that the continued fraction converges fast in one part of the domain and the series converges fast in the other.

A continued fraction can be written as

$$\cfrac{a_1}{b_1 + \cfrac{a_2}{b_2 + \frac{a_3}{b_3}}} \tag{10.5}$$

but for typographical convenience is usually written

$$\frac{a_1}{b_1+}\ \frac{a_2}{b_2+}\ \frac{a_3}{b_3+} \tag{10.6}$$

which gives the opportunity to add an ellipsis since most continued fractions of interest are infinite.

The first thing that we do is write a simple function to compute a continued fraction:

```
"fjjcontinue.fraction"<-
function(a, b)
{
        n <- length(a)
        if(length(b) != n)
                stop("a and b do not match")
        if(n < 2)
                stop("not enough terms")
        bottom <- b[n]
        for(i in n:2) {
                bottom <- b[i - 1] + a[i]/bottom
        }
        a[1]/bottom
}
```

This has the distinct disadvantage that it is not vectorized, so it is of little value. Below is one way of vectorizing the computations—I'm not convinced that it is the best way.

```
"continue.fraction"<-
function(num, den)
{
        num <- as.matrix(num)
        den <- as.matrix(den)
        if(any(dim(num) != dim(den)))
                stop("num and den do not match")
        n <- nrow(num)
        if(n < 2)
```

```
                stop("not enough terms")
        bottom <- den[n,  ]
        for(i in n:2) {
                bottom <- den[i - 1,  ] + num[i,  ]/
                        bottom
        }
        num[1,  ]/bottom
}
```

This function takes two matrices of the same size, the number of columns is the length of the result, and the number of rows is the number of terms in the continued fractions. Although this function could be called directly, it is more likely that another function that returns a mathematical function will call it.

One function that can be computed well with a continued fraction is the exponential integral. It is defined as

$$E_n(z) = \int_1^\infty \frac{e^{-zt}}{t^n} dt \tag{10.7}$$

where n is a non-negative integer. Real values of z must be positive.

Press *et al* (1992) give the continued fraction in which we are interested:

$$E_n(z) = e^{-z}\left(\frac{1}{z+n-}\ \frac{n}{z+n+2-}\ \frac{2(n+1)}{z+n+4-}\ \cdots\ \frac{i(n+i-1)}{z+n+2i-}\ \cdots\right) \tag{10.8}$$

This is relatively easy to translate into S:

```
"fjjexp.integral"<-
function(z, n, nterms = 9)
{
        outlen <- max(length(z), length(n))
        z <- rep(z, length = outlen)
        n <- rep(n, length = outlen)
        tseq <- 0:(nterms - 1)
        num <-  - outer(tseq - 1, n, "+") * tseq
        num[1,  ] <- 1
        den <- outer(2 * tseq, z + n, "+")
        exp( - z) * continue.fraction(num, den)
}
```

The `outer` function is instrumental in creating the matrices that are required by `continue.fraction`.

As always, we turn to the tables in Abramowitz and Stegun (1964) to see if we are doing it right:

```
> fjjcheckexpint
function(nterms = 9)
{
        x <- c(1.95, 0.99, 0.5, 0.01, 0.01, 0.01, 0.01,
                1.01, 1.01, 1.01, 1.01, 1.01)
        n <- c(1, 1, 1, 3, 4, 10, 20, 2, 3, 4, 10, 20)
        enx <- c(0.05241438, 0.223099826, 0.559773595,
                0.4902766, 0.3283824, 0.1098682,
                0.052079, 0.1463199, 0.1082179,
                0.084973, 0.0359929, 0.0181539)
        fjjexp.integral(x, n, nterms = nterms) - enx
}
> fjjcheckexpint()
 [1] -1.756406e-07 -2.682575e-05 -9.489412e-04
 [4] -7.990076e-03 -1.397079e-03 -2.188219e-06
 [7] -5.328954e-08 -2.038522e-05 -1.198577e-05
[10] -6.257321e-06 -1.073663e-07  2.527789e-08
> fjjcheckexpint(100)
 [1] -4.320013e-10 -2.098228e-10 -2.268147e-10
 [4] -3.533094e-05 -9.879388e-07  2.627358e-08
 [7] -4.582066e-08  3.953909e-08  2.031852e-08
[10] -3.998339e-08  1.111485e-08  2.632293e-08
> fjjcheckexpint(200)
 [1] -4.320013e-10 -2.098227e-10 -2.238392e-10
 [4] -4.090723e-06 -1.107696e-07  2.627360e-08
 [7] -4.582066e-08  3.953909e-08  2.031852e-08
[10] -3.998339e-08  1.111485e-08  2.632293e-08
```

The continued fraction is reputed to converge slowest for small numbers. The algorithm in Press *et al* (1992) uses a series approximation instead of the continued fraction for numbers less than 1. It appears that using 100 terms in the continued fraction is a reasonable compromise between speed and accuracy.

We can also check a few points for complex numbers:

```
> fjjcheckexpintcomp
function(nterms)
{
        z <- c(-2+0.2i, 1i, 2.5+0.6i)
        n <- rep(1, length(z))
        enzl <- c(-4.219228+0.636779i,
                -0.337404+0.946083i,
                0.961532+0.218215i)
        fjjexp.integral(z, n, nterms = nterms) + log(z
                ) - enzl
}
> fjjcheckexpintcomp(100)
```

```
[1] -3.580492e-01+8.532514e-02i
[2]  7.710208e-08+7.036693e-08i
[3] -2.616037e-07-2.712842e-07i
> fjjcheckexpintcomp(1000)
[1]  8.268037e-04-5.364546e-05i
[2]  7.709903e-08+7.036718e-08i
[3] -2.616037e-07-2.712842e-07i
> fjjcheckexpintcomp(5000)
[1] -3.853327e-07-1.648335e-07i
[2]  7.709903e-08+7.036718e-08i
[3] -2.616037e-07-2.712842e-07i
```

We now have good evidence that the function is coded properly, but we are left in the uncomfortable situation of not knowing where in the complex plane the approximation is going to be poor.

Finally, the function can be made to behave more like other mathematical functions. The restriction put on `n` may cause some annoyance, but more likely it will catch blunders.

```
"exp.integral"<-
function(x, n, nterms = 100)
{
        if(!length(x))
                return(x)
        xatt <- attributes(x)
        x <- as.vector(x)
        outlen <- length(x)
        if(length(n) == 1)
                n <- rep(n, outlen)
        else if(length(n) != outlen)
                stop("n must be one long or the length of x")
        n <- round(n)
        tseq <- 0:(nterms - 1)
        num <-  - outer(tseq - 1, n, "+") * tseq
        num[1,  ] <- 1
        den <- outer(2 * tseq, x + n, "+")
        ans <- exp( - x) * continue.fraction(num, den)
        attributes(ans) <- xatt
        ans
}
```

If we needed an accurate calculation of the exponential integral function, then we would want to create code (in C) that would ensure that we got good accuracy for every value. We don't have that here, but we do have an algorithm that works for both numeric and complex arguments that was quick to create.

10.6 Things to Do

Extend the functionality of `numberbase` to non-integers, missing values and infinity. Is there a better way of organizing the computations?

Teach `numberbase` to do mathematics. If the bases of two numbers in an operation are different, then what base should the answer be? How will you test the results?

Write a function based on `scan` that will read in rational numbers printed in the same format that `print.rationalnum` uses. Is it robust to differences in white-space?

Make rational numbers work within arrays, data frames and so on. Include matrix multiplication, and fill in any other holes in the implementation.

There is no error given when there is overflow in the denominator or numerator in `rationalnum` objects:

```
> jjr3 <- rationalnum(1, c(353, 713, 927, 213))
> > jjr3[1] * jjr3[2] * jjr3[3]
[1] 1/233315703
> jjr3[1] * jjr3[2] * jjr3[3] * jjr3[4]
[1] -1/1843362813
```

How would you fix this problem? Are there changes that can be made to reduce the problem, even without a proper fix? Find and fix any other problems with rational numbers.

Revise the `great.common.div` function to use C code. Write both a Euclidean algorithm version and a binary algorithm version of the C code. Is it worth changing to C code? Which algorithm works better? A description of the binary algorithm is given in Knuth (1981).

Try to write a better version of `continue.fraction`.

Create functionality for rational complex numbers.

Make a class of objects of rational numbers in an arbitrary base.

The `polyroot` S-PLUS function finds the roots of a polynomial of order at most 48. Create a function that does not have such a limitation. Young and Gregory (1973, volume 1) has a chapter on roots of polynomials.

Try out at least three ways of evaluating polynomials. Test them for speed and memory use.

Write functions that allow infinite-precision arithmetic.

Create functions that do symbolic mathematics.

Write a suite of functions to do interval arithmetic. That is, instead of each quantity being a single number, you have a range of numbers that the quantity is known to lie in.

10.7 Further Reading

Knuth (1981) *Seminumerical Algorithms* contains a great deal of interesting material related to mathematical computing. It presents pretty much all you are likely to want to know about rational arithmetic. Other topics include radix conversion and continued fractions.

Other books that discuss numerical computation include Young and Gregory (1973) *A Survey of Numerical Mathematics*; Hamming (1973) *Numerical Methods for Scientists and Engineers*. Hamming discusses the polygamma functions, but uses the $\Gamma(x+1)$ form.

Abramowitz and Stegun (1964) *Handbook of Mathematical Functions* contains much information on mathematical functions including the digamma and polygamma functions and the exponential integral. Approximations, tables of values and relationships between functions are just some of what is included.

Press *et al* (1992) *Numerical Recipes in C, Second Edition* provides many numerical algorithms. In particular, they have one for the exponential integral for real values.

Wall (1948). gives an in-depth treatment of continued fractions.

10.8 Quotations

[34]William Butler Yeats "The Cat and the Moon"

[35]W. H. Auden "In Memory of W. B. Yeats"

Chapter 11

Character

Here the focus is character data. *No belly and no bowels, Only consonants and vowels.* [36]

11.1 Perl

This is a function that provides a simple interface to Perl. It assumes that the same action will be performed on each element of the input `x`.

```
"perl"<-
function(x, cmd, preface = "", print = T, trace = F)
{
        ploc <- tempfile("sperl")
        on.exit(unlink(ploc))
        perl.pre <- c("#!/usr/bin/perl", preface,
                "while(<>) {")
        if(print)
                perl.post <- c(";\tprint $_ ;", "}")
        else perl.post <- c(";", "}")
        cat(file = ploc, c(perl.pre, cmd, perl.post),
                sep = "\n")
        if(trace) {
                foo <- unix(paste("cat", ploc), out =
                        F)
        }
        ans <- unix(paste("perl", ploc), x)
        if(length(ans) == length(x))
                attributes(ans) <- attributes(x)
        ans
}
```

There are other ways to use Perl of course, but this is probably the most natural for S. This function starts by creating a file that contains a Perl script. If `trace` is TRUE, then the script is printed—this is useful for debugging purposes. Finally the script is run, and the answer is given the attributes of the input if the lengths match.

The `transcribe` function uses `perl`.

```
"transcribe"<-
function(x, old, new = "", complement = F, delete = F,
        squash = F, sep = "/")
{
        flags <- paste(c("c", "d", "s")[c(complement,
                delete, squash)], collapse = "")
        if(complement)
                old <- paste(old, "\\n", sep = "")
        ptext <- paste("tr", sep, old, sep, new, sep,
                flags, " ;", sep = "")
        if(length(ptext) != 1)
                stop("old and new must each be single strings")
        perl(x, ptext)
}
```

In good textbook fashion, I have presented this in reverse order. In reality, a version of `transcribe` came first—that was the task at hand—then the Perl part of the function was abstracted out and `transcribe` rewritten.

Switching between upper and lower case is one use for `transcribe`.

```
> jjsn <- state.name[21:32]
> jjsn
 [1] "Massachusetts" "Michigan"      "Minnesota"
 [4] "Mississippi"   "Missouri"      "Montana"
 [7] "Nebraska"      "Nevada"        "New Hampshire"
[10] "New Jersey"    "New Mexico"    "New York"
> transcribe(jjsn, "A-Z", "a-z")
 [1] "massachusetts" "michigan"      "minnesota"
 [4] "mississippi"   "missouri"      "montana"
 [7] "nebraska"      "nevada"        "new hampshire"
[10] "new jersey"    "new mexico"    "new york"
> transcribe(jjsn, "a-z", "A-Z")
 [1] "MASSACHUSETTS" "MICHIGAN"      "MINNESOTA"
 [4] "MISSISSIPPI"   "MISSOURI"      "MONTANA"
 [7] "NEBRASKA"      "NEVADA"        "NEW HAMPSHIRE"
[10] "NEW JERSEY"    "NEW MEXICO"    "NEW YORK"
```

Other amusements include:

```
> transcribe(jjsn, "aeiou", delete=T)
 [1] "Msschstts" "Mchgn"     "Mnnst"     "Mssssppp"
 [5] "Mssr"      "Mntn"      "Nbrsk"     "Nvd"
 [9] "Nw Hmpshr" "Nw Jrsy"   "Nw Mxc"    "Nw Yrk"
> transcribe(transcribe(jjsn, "aeiou", delete=T),
+       "a-z", "a-z", squash=T)
 [1] "Mschsts"   "Mchgn"     "Mnst"      "Msp"
 [5] "Msr"       "Mntn"      "Nbrsk"     "Nvd"
 [9] "Nw Hmpshr" "Nw Jrsy"   "Nw Mxc"    "Nw Yrk"
> transcribe(jjsn, "aeiou", "_", comp=T)
 [1] "_a__a__u_e___" "_i__i_a_"      "_i__e_o_a"
 [4] "_i__i__i__i"   "_i__ou_i"      "_o__a_a"
 [7] "_e__a__a"      "_e_a_a"        "_e___a____i_e"
[10] "_e___e__e_"    "_e___e_i_o"    "_e___o__"
> transcribe(jjsn, "aeiou", "_", comp=T, delete=T)
 [1] "aaue" "iia"  "ieoa" "iiii" "ioui" "oaa"  "eaa"
 [8] "eaa"  "eaie" "eee"  "eeio" "eo"
```

When you're done playing, you can use `transcribe` to change names valid in C to valid S names, and vice versa.

```
> transcribe("state_name", "_", ".")
[1] "state.name"
> transcribe("state.name", ".", "_")
[1] "state_name"
```

This is its use in `portoptgen` of page 348.

The `substifile` function uses Perl in a different fashion. Here the intent is not to change S objects, but to change files.

```
"substifile"<-
function(filenames, old, new, sep = "/", backup =
        ".jj")
{
        if(length(old) != 1 || length(new) != 1)
                stop("old and new must be single strings")
        if(nchar(sep) != 1)
                stop("sep needs to be a single character")
        if(any(AsciiToInt(sep) == AsciiToInt(c(old,
                new))))
                stop("sep character in old or new")
        cmd <- paste("perl -p -i", backup, " -e \"s",
                sep, old, sep, new, sep, " ;\" ",
                paste(filenames, collapse = " "), sep
                 = "")
```

```
        unix(cmd, out = F)
}
```

A vector of file names is taken as well as a string to be replaced and a string to substitute in. Some checks are made to ensure that the input is okay, then the command is pasted together, and finally the command is executed. The return value of `substifile` is the return value of the call to `unix` with `output=F`, which is the exit status of the command. The `p` flag to Perl invokes a while loop like the one that the `perl` function creates. The `i` flag means to do the changes in place; if followed by a string, then backup copies of the original files are created, adding the string as a suffix to the filenames.

The `substifile` function is used later (in `portoptgen`, page 348) to modify template files to be specific to a particular case.

An alternative to Perl for manipulating character strings is C. Basic functionality is obtained with the `strings.h` header file.

11.2 Home Brew

Although S is relatively weak on functionality for character data, quite a lot can be done with its in-built functions. Most often of use are `paste`, `substring` and `nchar`.

Here is a function to justify a vector of character strings. The `format` function does this, but you have no control over how it does it.

```
"justify"<-
function(x, type = "r")
{
        type <- unabbrev.value(type, c("right",
                "center", "left"))
        x <- as.character(x)
        ncx <- nchar(x)
        blanks <- paste(rep(" ", max(ncx)), collapse
                 = "")
        blanks <- substring(blanks, 1, max(ncx) - ncx)
        switch(type,
                right = paste(blanks, x, sep = ""),
                left = paste(x, blanks, sep = ""),
                center = {
                        blank.half <- nchar(blanks) %/%
                                2
                        paste(substring(blanks, 1,
                                blank.half), x,
                                substring(blanks,
```

```
                                blank.half + 1), sep
                                 = "")
                }
                )
}
```

The `blanks` variable is first created as a single string of a certain number of blank spaces, then it is replicated to the length of `x` via `substring`. All the rest of the function is quite simple. Below is an illustration of the three types of justification.

```
> as.matrix(justify(dimnames(freeny.x)[[2]], "r"))
                        [,1]
[1,] "lag quarterly revenue"
[2,] "          price index"
[3,] "         income level"
[4,] "     market potential"
> as.matrix(justify(dimnames(freeny.x)[[2]], "l"))
                        [,1]
[1,] "lag quarterly revenue"
[2,] "price index          "
[3,] "income level         "
[4,] "market potential     "
> as.matrix(justify(dimnames(freeny.x)[[2]], "c"))
                        [,1]
[1,] "lag quarterly revenue"
[2,] "     price index     "
[3,] "    income level     "
[4,] "  market potential   "
```

There is a description of "valid" S names on page 4, here is a function that determines if each element of a character vector is a valid name.

```
"valid.s.name"<-
function(x)
{
        xnum <- lapply(x, AsciiToInt)
        the.table <- c(48:57, 46, 65:90, 97:122)
        xmatch <- lapply(xnum, match, table =
                the.table, nomatch = 0)
        good <- unlist(lapply(xmatch, function(z)
        all(z > 10)))
        bad <- unlist(lapply(xmatch, function(z)
        any(z == 0) || all(z < 11) || !length(z)))
        ans <- rep(NA, length(x))
```

```
        ans[good] <- T
        ans[bad] <- F
        ugly <- !good & !bad
        if(any(ugly)) {
                xug <- unlist(lapply(xmatch[ugly],
                        function(z)
                z[z != 11][1] > 11))
                ans[ugly][xug] <- T
                ans[ugly][!xug] <- F
        }
        names(ans) <- names(x)
        ans
}
```

. Although this does get the answer, we are straining ourselves for such a simple request. The first thing is to use `lapply` to effectively vectorize `AsciiToInt`. All of the rest is dealing with subscripting with logicals. Notice that special attention needs to be given to the empty string—this falls under the heading of "test along seams" discussed on page 53.

A use of this function is to make sure that a group of functions are under proper source control. Suppose that you have an object `manifest` that is a vector of the names of all the functions you want to have controlled. Some of these may be operators or assignment functions, and have a different name on the dump file. You can issue a command like:

```
for(i in manifest[valid.s.name(manifest)]) {
  cat("\n", i, "\n")
  print(diffsccs(i))
}
```

Then view the names of the strange ones with:

```
manifest[!valid.s.name(manifest)]
```

These can be checked with `diffsccs` by knowing the naming convention.

The `find.I.of` function does a very specific thing—take a string and find the location of calls to the `I` function in it. Page 301 tells why we would want to do this.

So here's the game: Find where the "I"s are that correspond to calls to the function, then find where the calls end.

```
> find.I.of
function(string, nesting.ok = F)
{
        string.code <- AsciiToInt(string)
```

```
        if(!any(eyes <- string.code == AsciiToInt("I")))
                return(NULL)
        open.par <- AsciiToInt("(")
        close.par <- AsciiToInt(")")
        opens <- string.code == open.par
        Istart <- eyes & c(opens[-1], F)
        if(!any(Istart))
                return(NULL)
        closes <- string.code == close.par
        counts <- cumsum(opens - closes)
        Istart.num <- seq(along = Istart)[Istart]
        ans <- array(0, c(length(Istart.num), 2))
        for(i in 1:length(Istart.num)) {
                loci <- Istart.num[i]
                this.lev <- counts[-1: - loci] ==
                        counts[loci]
                if(!any(this.lev))
                        stop("no closing parenthesis for I")
                ans[i,  ] <- c(0, min(seq(along =
                        this.lev)[this.lev])) + loci
        }
        if(!nesting.ok && nrow(ans) > 1) {
                ans <- ans[c(T, diff(ans[, 2]) > 0),  ,
                        drop = F]
        }
        ans
}
```

The hard part is finding the end of the call. The `counts` variable tells how many opening parentheses haven't yet had a closing parenthesis to match it. The `for` loop finds all of the calls, then later the nested calls are removed if required. Notice the `drop = F` to ensure that we always end up with a matrix.

Here are some examples:

```
> jjs1
[1] "~ jj1 + 2:4 * jjI"
> find.I.of(jjs1)
NULL
> jjs2
[1] "~ jj1 * I((2*jjI)+1)"
> find.I.of(jjs2)
     [,1] [,2]
[1,]    9   20
> jjs3
[1] "~ jj1 * 2:4"
> find.I.of(jjs3)
```

```
NULL
> jjs4
[1] "~ I((2)* I(jee+3)) + 2+I(3^x) + I (y^2)"
> find.I.of(jjs4)
     [,1] [,2]
[1,]    3   18
[2,]   24   29
> find.I.of(jjs4, T)
     [,1] [,2]
[1,]    3   18
[2,]   10   17
[3,]   24   29
```

Notice that the call in `jjs4` that has a space between the "I" and the "(" is not found by the function. But this isn't really much of a problem because the space won't be there if the string is parsed and then deparsed.

11.3 Things to Do

Make a list of desirable functionality for character data. How best can it be achieved?

The comparison operators like `<` work with character data where the ordering comes from ASCII. Create a function that will compare character vectors using an arbitrary ordering of characters. Use this to create a function that will sort according to the arbitrary ordering.

Create a function for playing "hangman".

Write a function that takes a vector and returns a list where each component of the result contains the "words" in the corresponding element of the input. How general can you make it?

Rewrite `valid.s.name` using the `perl` function. Try improving my version of the function using only "native" functionality. How should the reserved words of S be handled?

Write a function that generalizes `find.I.of`, so that it can be a simple call to your function.

11.4 Further Reading

The two books *Programming Perl* by Wall, Christiansen and Schwartz, plus *Learning Perl* by Schwartz provide a good explanation of the Perl language.

11.5 Quotations

[36]John Crowe Ransom "Survey of Literature"

Chapter 12

Arrays

The focus is on arrays that have more than two dimensions, but we start with matrices. See also the section in the chapter on C about dealing with arrays in C code—it starts on page 173.

12.1 Matrices

The square root of a symmetric matrix A is defined as any matrix R such that

$$A = R'R \tag{12.1}$$

where the prime denotes the transpose. The Choleski decomposition provides one square root. This is the particular square root that is upper triangular.

```
> jjfv <- var(freeny.x)
> jjfvc <- chol(jjfv)
> > max(abs(t(jjfvc) %*% jjfvc - jjfv))
[1] 1.387779e-17
```

Sometimes it is desirable to have the symmetric square root. Here is a function that returns it, or its inverse.

```
"symsqrt"<-
function(x, inverse = F, tol = 1e-10)
{
        dx <- dim(x)
        if(dx[1] != dx[2])
                stop("need square matrix")
        if(any(abs(x - t(x)) > tol * max(abs(x))))
                stop("need symmetric matrix")
        xeig <- eigen(x, sym = T)
```

```
        if(inverse)
                xeig$vectors %*% (xeig$values^-0.5 * t(
                        xeig$vectors))
        else xeig$vectors %*% (xeig$values^0.5 * t(
                        xeig$vectors))
}
```

This performs an eigen decomposition of the matrix, takes the square root (or inverse square root) of each eigenvalue, and then puts the matrix back together again. In effect it uses a sparse matrix technique for the diagonal matrix of eigenvalues.

Now test it to see that it does return a square root, and that the result is symmetric.

```
> jjfvs <- symsqrt(jjfv)
> max(abs(jjfvs %*% jjfvs - jjfv))
[1] 1.526557e-16
> max(abs(jjfvs - t(jjfvs)))
[1] 0
```

There are, in fact, a whole group of square roots. The following function returns a random square root.

```
"rmatsqrt"<-
function(x)
{
        xc <- chol(x)
        x[] <- runif(dim(x)[1]^2)
        orthmat <- svd(x)$v
        orthmat %*% xc
}
```

This finds one square root with `chol`, then creates an orthogonal matrix out of a matrix of random numbers. These two are then multiplied to produce the random square root.

Below two different square roots are produced, shown to be different, and then checked to see that they really are square roots of the matrix in question.

```
> jjsv1 <- rmatsqrt(var(freeny.x))
> jjsv2 <- rmatsqrt(var(freeny.x))
> jjsv1
     lag quarterly revenue price index income level
[1,]           -0.21783334  0.09945160  -0.07706364
[2,]           -0.10834290  0.03956840  -0.02901060
[3,]           -0.19116893  0.06958443  -0.08713387
```

```
[4,]           -0.06152129  0.03854587  -0.01156840
     market potential
[1,]     -0.047556536
[2,]     -0.021705228
[3,]     -0.036462742
[4,]     -0.009923954
> jjsv2
     lag quarterly revenue price index income level
[1,]            -0.1831166  0.08579325  -0.06217413
[2,]            -0.1119885  0.04130825  -0.06180107
[3,]            -0.1560023  0.05075554  -0.06170711
[4,]            -0.1706170  0.07836504  -0.05489805
     market potential
[1,]      -0.03474957
[2,]      -0.02301239
[3,]      -0.03051149
[4,]      -0.03863553
> all.equal(t(jjsv1) %*% jjsv1, var(freeny.x))
[1] T
> all.equal(t(jjsv2) %*% jjsv2, var(freeny.x))
[1] T
```

12.2 Array Functions

This section contains functions that make it easier to use higher-dimensional arrays.

Stable Apply

When the function used in `apply` returns a vector, then those vectors fill along the first dimension of the result. For example, sorting the rows of a matrix will give you the transpose of what one would naively expect.

```
> jjmr <- matrix(sample(15), 3)
> jjmr
     [,1] [,2] [,3] [,4] [,5]
[1,]   14    3    1    2   10
[2,]   12   13    9    4   15
[3,]    7   11    8    5    6
> apply(jjmr, 1, sort)
     [,1] [,2] [,3]
[1,]    1    4    5
[2,]    2    9    6
[3,]    3   12    7
```

```
[4,]   10   13    8
[5,]   14   15   11
```

The `stable.apply` function keeps the dimensions of the output in the same order as the input.

```
> stable.apply(jjmr, 1, sort)
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    2    3   10   14
[2,]    4    9   12   13   15
[3,]    5    6    7    8   11
```

Here is the definition of `stable.apply`.

```
"stable.apply"<-
function(X, MARGIN, FUN, ...)
{
        ldx <- length(dim(X))
        if(length(MARGIN) != ldx - 1) {
                warning("stability not performed")
                return(apply(X, MARGIN, FUN, ...))
        }
        ans <- apply(X, MARGIN, FUN, ...)
        if(length(dim(ans)) != ldx)
                ans
        else aperm(ans, order(c((1:ldx)[ - MARGIN], MARGIN)))
}
```

You can see from this that I've overstated the case somewhat—the dimensions are left intact only when a single dimension is being "collapsed". The only real operation except feeding the problem to `apply` is to permute the dimensions with `aperm` (which you can read about on page 87).

Binding Arrays

An operation that is fairly common with higher-dimensional arrays is to bind them together—similar to `rbind` and `cbind` with matrices. Here are a couple of examples.

```
> jjm3 <- array(1:6, c(2,3))
> jjm4 <- jjm3 + 7
> bind.array(jjm3, jjm4, 3)

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

```
[2,]    2    4    6

, , 2
     [,1] [,2] [,3]
[1,]    8   10   12
[2,]    9   11   13
> bind.array(jjm3, jjm4, 2)
     [,1] [,2] [,3] [,4] [,5] [,6]
[1,]    1    3    5    8   10   12
[2,]    2    4    6    9   11   13
```

This last command is the same as a `cbind` call. Here's the definition.

```
"bind.array"<-
function(x, y, margin)
{
        ldx <- length(dx <- dim(x))
        ldy <- length(dy <- dim(y))
        if(ldx != ldy)
                stop("length of dimensions not equal")
        margin <- round(margin)
        if(margin < 1)
                stop("bad value for margin")
        if(margin > ldx) {
                if(margin - ldx == 1) {
                        dx <- c(dx, 1)
                        ldx <- ldx + 1
                        dy <- c(dy, 1)
                        x <- array(x, dx, if(length(
                                dimnames(x))) c(
                                  dimnames(x), list(
                                  NULL)))
                        y <- array(y, dy, if(length(
                                dimnames(x))) c(
                                  dimnames(y), list(
                                  NULL)))
                }
                else stop("bad value for margin")
        }
        if(any(dx[ - margin] != dy[ - margin]))
                stop("arrays not conformable")
        newdim <- dx
        newdim[margin] <- dx[margin] + dy[margin]
        newdimnames <- dimnames(x)
        newdimnames[[margin]] <- c(dimnames(x)[[margin
                ]], dimnames(y)[[margin]])
```

```
        ans <- array(x[1], newdim, newdimnames)
        cmd <- paste("ans[", paste(rep(",", margin - 1
                ), collapse = " "), "1:dx[margin]",
                paste(rep(",", ldx - margin), collapse
                 = " "), "] <- x")
        eval(parse(text = cmd))
        cmd <- paste("ans[", paste(rep(",", margin - 1
                ), collapse = " "),
                "dx[margin] + 1:dy[margin]", paste(rep(
                ",", ldx - margin), collapse = " "),
                "] <- y")
        eval(parse(text = cmd))
        ans
}
```

There is a lot of messing around with details, but essentially all that is done is to create the answer to be the right size and shape, then use the eval-parse-text idiom twice to put the inputs into the proper locations of the answer.

12.3 Things to Do

Find and fix the bugs in `bind.array`. A particular one is that it only accepts two arrays. Check how you are doing by looking at the `abind` function from Statlib submitted by Tony Plate and Rich Heiberger.

Create a function that generalizes a matrix computation to three-dimensional arrays.

12.4 Further Reading

A discussion of computations with matrices is Golub and Van Loan (1983).

Chapter 13

Formulas

Formulas provide a mechanism for giving a function an arbitrarily complex input. It is never necessary to have a function take a formula—the same functionality can be achieved through other means—however, formulas can offer a clean interface to the user.

A formula is a call to the tilde operator. This operator does essentially nothing—the result is a call to the operator that has class `"formula"`. A brief explanation of this is on page 203.

Many of the modeling functions use the `terms` function which digests a formula. Although `terms` is useful in many situations, it is not necessary to use it with formulas—the mathematical graph example later in this chapter, for instance, does not use `terms`.

13.1 Lazy Evaluation

Formulas are the main trouble spot with lazy evaluation. Formulas have their own form of lazy evaluation that is distinct from but similar to the lazy evaluation of function arguments. The contents of a formula are not evaluated when the formula is evaluated. It is only when the data are really needed that evaluation will happen, and this can happen an arbitrary number of frames away from where the formula is given.

Here is a simplified example of how the problem might arise. We create a formula containing `x` and `y` in our function where they are perfectly behaved objects. However, the formula is passed to another function where it is actually used.

```
"fjjlz1"<-
function(x, y)
{
```

```
        subfun <- function(form)
        {
                lm(form)
        }
        form <- y ~ x
        subfun(form)
}
```

The result is none too good. By the time the variables within the formula are needed, S doesn't know the proper place to look for them.

```
> fjjlz1(jjx, jjy)
Error in subfun(form): Object "y" not found
Dumped
```

There are a couple of remedies. One is to make global variables of those that appear in the formula.

```
"fjjlz2"<-
function(x, y)
{
        subfun <- function(form)
        {
                lm(form)
        }
        form <- y ~ x
        assign("x", x, frame = 1)
        assign("y", y, frame = 1)
        subfun(form)
}
```

We get an answer instead of an error.

```
> fjjlz2(jjx, jjy)
Call:
lm(formula = form)

Coefficients:
 (Intercept)        x1        x2
    0.476645 0.9723329 -0.248944

Degrees of freedom: 10 total; 7 residual
Residual standard error: 0.4326354
```

Perhaps the preferred solution is to keep the data and the formula together. The statistical modeling functions that take formulas expect a data frame of data.

```
"fjjlz3"<-
function(x, y)
{
        subfun <- function(form, df)
        {
                lm(form, data = df)
        }
        form <- y ~ x
        df <- data.frame(x = x, y = y)
        subfun(form, df)
}
```

Now we try it.

```
> fjjlz3(jjx, jjy)
Error in subfun(form, df): Object "x" not found
Dumped
```

This would be the preferred technique if it worked. Let's use `debugger` to see where the trouble lies.

```
> debugger()
Message: Object "x" not found

1:
2: fjjlz3(jjx, jjy)
3: subfun(form, df)
4: lm(form, data = df)
5: eval(m, sys.parent())
6: model.frame.default(formula = form, data = df)
7: subfun(form, df)
Selection: 2
Frame of fjjlz3(jjx, jjy)
d(2)> ?
1: df
2: y
3: x
4: form
5: subfun
d(2)> df
   x.1         x.2         y
 1   1  0.07007709  1.569356
 2   2 -0.53393595  2.983659
 3   3 -0.03948787  3.366048
 4   4  0.49557847  3.768500
 5   5  0.23720871  5.634129
```

```
 6   6 -0.25087103  5.722343
 7   7 -1.16859131  7.272687
 8   8  1.92886521  7.657451
 9   9  1.00915360  9.374802
10  10 -1.23286270 10.767544
```

The problem is that `data.frame` doesn't put a matrix in as a single item, but makes a column for each column of the matrix. (Usually this is the desired behavior, but obviously not always.)

```
"fjjlz4"<-
function(x, y)
{
        subfun <- function(form, df)
        {
                lm(form, data = df)
        }
        form <- y ~ x
        df <- data.frame(y = y)
        df$x <- x
        subfun(form, df)
}
```

This new version presumes that `y` will be a vector, and then puts `x` into the data frame "by hand" in case it is a matrix. Now, once again we get an answer.

```
> fjjlz4(jjx, jjy)
Call:
lm(formula = form, data = df)

Coefficients:
 (Intercept)        x1        x2
    0.476645 0.9723329 -0.248944

Degrees of freedom: 10 total; 7 residual
Residual standard error: 0.4326354
```

There is another mechanism for passing the data along with the formula since modeling functions typically take an integer indicating the frame in which the objects are to be found as well as a data frame of the objects. (Thus the dual use of the word "frame".) Here's our function to do that.

```
"fjjlz5"<-
function(x, y)
{
        subfun <- function(form, df)
        {
```

```
                lm(form, data = df)
        }
        form <- y ~ x
        subfun(form, sys.nframe())
}
```

We want to point to the frame of `fjjlz5`. Even though the `sys.nframe()` is inside the call to `subfun`, it is evaluated in the frame of `fjjlz5` so it is correct. A call to `sys.frame` (giving the frame rather than just the frame number) would work also. An alternative way of doing it would have been to have the command inside `subfun` be:

```
                lm(form, data = sys.parent())
```

Now for the test of the function.

```
> fjjlz5(jjx, jjy)
Error in model.matrix.default(Terms, ..: Invalid data.\
        class(x) ("argument")
Dumped
```

This is a problem with the lazy evaluation of arguments. The mode of something from the formula is still `argument`—meaning it hasn't been evaluated yet—rather than something that a numerical routine might be happy with. So we fix things up.

```
"fjjlz6"<-
function(x, y)
{
        subfun <- function(form, df)
        {
                lm(form, data = df)
        }
        x <- x
        eval(y)
        form <- y ~ x
        subfun(form, sys.frame())
}
```

Both `x` and `y` need to be evaluated. This uses one method for each. And assigning a variable to itself does indeed make a difference.

```
> fjjlz6(jjx, jjy)
Call:
lm(formula = form, data = df)

Coefficients:
```

```
 (Intercept)        x1        x2
    0.476645 0.9723329 -0.248944

Degrees of freedom: 10 total; 7 residual
Residual standard error: 0.4326354
```

13.2 Manipulating Formulas

A formula need not have a left-hand (response) side. In operator terms, the tilde can be either unary or binary.

A formula is really an object of mode `call` (the call is to the tilde operator). The first component of a call is the function, and subsequent components are the arguments. Thus, when both sides of the formula are given, then the formula has length 3—the second component is the left-hand side and the third component is the right-hand side. When only the right-hand side is given, the formula has length 2, and the second component contains the right-hand side.

A common operation is to add the left-hand side to a formula if it doesn't exist—the following code does this, where `form` is a formula:

```
if(length(form) == 2) {
     form[[3]] <- form[[2]]
     form[[2]] <- as.name("z")
}
```

This code puts the component originally in the second position into a new third component, then changes the second component. The second and third components of a formula should be either of mode `name` or of mode `call`.

The operators in formulas often have a special meaning. For example in

```
y ~ x1 + x2
```

`x1` is not being added to `x2`—it is that both are to be included. If you use a formula that is given to the `terms` function, then the "hat" operator as well as addition is special. The formula

```
y ~ x^2
```

will not be interpreted as the square of `x`. If that is what you want, then write:

```
y ~ I(x^2)
```

The `I` construct means to interpret functions in the usual sense of S rather than the "formula" sense.

13.3 Mathematical Graphs

A mathematical graph is a collection of *nodes* (also called *vertices*), and a collection of *edges* that each connect two of the nodes. Edges can be either directed or undirected. A graph that contains only directed edges is called a directed graph.

The `mathgraph` function creates an object representing a mathematical graph. The form of the graph is given to `mathgraph` via a formula. Obviously a graph can be arbitrarily complex, so it makes some sense to use a formula to describe it.

Let's start with some examples.

```
> mathgraph(~ 1:3 / 2:4)
[1]  node 1 <-> node 2
[2]  node 2 <-> node 3
[3]  node 3 <-> node 4

class: mathgraph
> mathgraph(~ 1:3 * 2:4)
[1]  node 1 <-> node 2
[2]  node 2 <-> node 2
[3]  node 3 <-> node 2
[4]  node 1 <-> node 3
[5]  node 2 <-> node 3
[6]  node 3 <-> node 3
[7]  node 1 <-> node 4
[8]  node 2 <-> node 4
[9]  node 3 <-> node 4

class: mathgraph
> mathgraph(~ 1:3 / 4)
[1]  node 1 <-> node 4
[2]  node 2 <-> node 4
[3]  node 3 <-> node 4

class: mathgraph
```

This displays the two main operators in `mathgraph` formulas. The slash operator connects each element in the first vector with the corresponding element in the second—as the last example shows, the usual replication rules hold. The star operator makes a star by connecting each element of the first vector with all of the elements in the second.

There is also the plus operator that concatenates the terms together.

```
>  mathgraph(~ 1:3 / 2:4 + mathgraph(~ c(3,1) / c(2,4), dir=T))
```

```
[1]  node 1 <-> node 2
[2]  node 2 <-> node 3
[3]  node 3 <-> node 4
[4]  node 3  -> node 2
[5]  node 1  -> node 4

class: mathgraph
```

This example shows that a term can not only be a call to `*` or `/`, but also it can be another `mathgraph` object. In this case, the second term is of a directed graph.

The nodes need not be numbers, they can be character strings also.

```
> mathgraph(~ state.name[2:4] * state.name[12:13])
[1]   Alaska <->  Idaho
[2]   Arizona <->  Idaho
[3]   Arkansas <->  Idaho
[4]   Alaska <->  Illinois
[5]   Arizona <->  Illinois
[6]   Arkansas <->  Illinois

class: mathgraph
```

Here is the definition of the `mathgraph` function.

```
"mathgraph"<-
function(formula, directed = F, data = sys.parent())
{
        if(missing(formula)) {
                ans <- NULL
        }
        else {
                ans <- build.mathgraph(formula, data
                        = data)
                adir <- attr(ans, "directed")
                adir[is.na(adir)] <- directed
                attr(ans, "directed") <- adir
        }
        class(ans) <- "mathgraph"
        ans
}
```

First, we allow the possibility of an empty graph. The real work is done by `build.mathgraph`, then a few additions are put on. The `directed` argument allows the edges that are created to be either directed or undirected. The `data` argument provides a way around problems with lazy evaluation as discussed on page 291.

The `build.mathgraph` function looks like:

```
"build.mathgraph"<-
function(formula, data)
{
        cterm <- deparse(formula[[2]])
        allterms <- unpaste(cterm, sep = "+")
        ans <- NULL
        adir <- logical(0)
        for(i in seq(along = allterms)) {
                raw <- allterms[[i]]
                this.parse <- parse(text = raw)
                if(length(this.parse[[1]]) == 1) {
                        this.op <- " "
                }
                else {
                        this.op <- as.character(this.parse[[1]][[1]])
                }
                switch(this.op,
                        "/" = {
                                e1 <- eval(this.parse[[1]][[2]], data)
                                e2 <- eval(this.parse[[1]][[3]], data)
                                this.ev <- cbind(e1, e2)
                                ans <- rbind(ans, this.ev)
                                adir <- c(adir, rep(NA, dim(this.ev)[1]))
                        }
                        ,
                        "*" = {
                                e1 <- eval(this.parse[[1]][[2]], data)
                                e2 <- eval(this.parse[[1]][[3]], data)
                                e1 <- unique(e1)
                                e2 <- unique(e2)
                                le1 <- length(e1)
                                le2 <- length(e2)
                                e1 <- e1[rep(1:le1, le2)]
                                e2 <- e2[rep(1:le2, rep(le1, le2))]
                                this.ev <- cbind(e1, e2)
                                ans <- rbind(ans, this.ev)
                                adir <- c(adir, rep(NA, dim(this.ev)[1]))
                        }
                        ,
                        {
                                this.ev <- eval(this.parse, data)
                                if(inherits(this.ev, "mathgraph")) {
                                  ans <- rbind(ans, this.ev)
                                  adir <- c(adir, attr(this.ev, "directed"))
                                }
                                else stop(paste(
                                    "do not know how to handle term:", raw))
                        }
```

```
                    )
        }
        attr(ans, "directed") <- adir
        ans
}
```

This breaks a character representation of the formula into the individual terms, then creates the representation for each term. It represents the graph with a two-column matrix where each row stands for an edge and the entries are the two nodes connected by the edge.

The `print` method for `mathgraph` produces a fairly clear representation of the graph that also matches the structure of the object quite closely.

```
"print.mathgraph"<-
function(x, prefix.node = if(is.character(xu)) ""
         else "node", ...)
{
        if(length(unclass(x))) {
                xu <- unclass(x)
                if(length(the.nams <- names(x))) {
                        the.nams <- paste(justify(
                                the.nams, "r"), " ",
                                sep = "")
                }
                else {
                        the.nams <- paste("[", format(
                                1:length(x)), "] ",
                                sep = "")
                }
                out <- paste(the.nams, prefix.node, xu[
                        , 1], ifelse(attr(x,
                        "directed"), " ->", "<->"),
                        prefix.node, xu[, 2])
                cat(out, sep = "\n")
                cat("\nclass:", class(x), "\n")
        }
        else {
                cat("mathgraph()\n")
        }
        invisible(x)
}
```

As always, if the last line of a `print` method is not `invisible(x)`, then it is wrong. This essentially just pastes the nodes for each edge together with a symbol for the directions of the edge between. Some of the code (including the call to `justify` which is given on page 278) refers to the names of the graph—

we will get to names shortly. Another part of the code indicates numerical subscripting which we will also get to later.

Let's be a little more adventurous.

```
> jj1 <- 1:3
> mathgraph(~ jj1 * jj1+1)
Error in switch(this.op,: do not know how to handle
        term:  1
Dumped
```

The statement above doesn't work, and it shouldn't. The intention there was to add one to each of the numbers in `jj1`, but the plus sign is interpreted as introducing a second term in the formula rather than as performing addition. The way around this is to surround operations that are to be interpreted in the usual sense instead of the formula sense with a call to `I`.

```
> mathgraph(~ jj1 * I(jj1+1))
Syntax error: end.of.file ("\n") used illegally at this point:
jj1 * I(jj1

Dumped
```

However, this doesn't give us much satisfaction either. Actually, this doesn't even come close—the formula is broken into pieces between plus signs, so we end up with something that doesn't even parse.

We need to revise `build.mathgraph` to handle `I`. Here is the first part of the revised version of `build.mathgraph`.

```
function(formula, data)
{
        cterm <- paste(collapse = "", deparse(formula[[2]]))
        eyes <- find.I.of(cterm)
        if(length(eyes)) {
# calls to I() need to be evaluated
                nI <- nrow(eyes)
                Inames <- paste("Build.mathgraphI",
                        sys.nframe(), 1:nI, sep = ".")
                Iexpr <- substring(cterm, eyes[, 1] +
                        2, eyes[, 2] - 1)
                for(i in 1:nI) {
                        this.val <- eval(parse(text =
                                Iexpr[i]), data)
                        assign(Inames[i], this.val,
                                frame = 1)
                }
                eye.ext <- matrix(c(0, t(eyes), nchar(
```

```
                cterm) + 2), nrow = 2)
            allI.strings <- character(2 * nI + 1)
            allI.strings[2 * (1:nI)] <- Inames
            allI.strings[seq(1, 2 * nI + 1, by = 2
                )] <- substring(cterm, eye.ext[
                1,  ] + 1, eye.ext[2,  ] - 1)
            cterm <- paste(allI.strings, collapse
                 = "")
    }
    allterms <- unpaste(cterm, sep = "+")
```

You may notice that the first line (that uses `deparse`) has been changed. This has nothing to do with `I`, but rather fixes a bug when the formula is long. `deparse` returns a string for each line that it thinks there should be. So the `paste` command ensures that we have just a single string.

After that, this starts with a call to `find.I.of` of page 280, which returns a matrix of the start and end of each call to `I` (if any). When there are instances of `I` to handle, then each of them is evaluated in the appropriate frame using the eval-parse-text idiom, and put into variables in frame 1. The call to `sys.nframe` makes sure that there will not be collisions in the names if there are nested occurrences of this code. The `cterm` string is broken into pieces, and the names for the evaluated `I` expressions are substituted in. This last operation could be abstracted into a separate function, but I'm not sure that it would be especially useful.

Now we can try our example again.

```
> mathgraph(~ jj1 * I(jj1+1))
[1]  node 1 <-> node 2
[2]  node 2 <-> node 2
[3]  node 3 <-> node 2
[4]  node 1 <-> node 3
[5]  node 2 <-> node 3
[6]  node 3 <-> node 3
[7]  node 1 <-> node 4
[8]  node 2 <-> node 4
[9]  node 3 <-> node 4

class: mathgraph
> mathgraph(~ I(jj1*2) / I (jj1+3) + jj1 / I(jj1+1))
[1]  node 2 <-> node 4
[2]  node 4 <-> node 5
[3]  node 6 <-> node 6
[4]  node 1 <-> node 2
[5]  node 2 <-> node 3
[6]  node 3 <-> node 4
```

```
class: mathgraph
```

It is sensible to name edges as well as nodes. The `names` of the `mathgraph` object is natural for this, given the structure. The names for the class can just be the row names of the matrix underlying the object.

```
"names.mathgraph"<-
function(x)
{
        dimnames(unclass(x))[[1]]
}
"names<-.mathgraph"<-
function(x, value)
{
        cl <- class(x)
        x <- unclass(x)
        dimnames(x)[[1]] <- value
        class(x) <- cl
        x
}
```

This makes the assignment form very easy as well. At this point, the unclassing is not necessary since there are no `mathgraph` methods for `dimnames`. However, we don't want our functions to break if code is added later.

We want the length of the graph to be the number of edges contained in the graph.

```
"length.mathgraph"<-
function(x)
{
        x <- unclass(x)
        if(length(x))
                dim(x)[1]
        else 0
}
```

A simpler way to write this function would be to replace the `if` statement by the line:

```
length(x)/2
```

However, it might be slightly safer and not much less efficient to leave it as is.

Subscripting is a very powerful feature of S, and we want subscripting to work for any class that we create if at all sensible. For graphs that means subscripting on the edges.

```
"[.mathgraph"<-
function(x, i)
{
        cl <- class(x)
        x <- unclass(x)
        xdir <- attr(x, "directed")
        if(is.character(i))
                names(xdir) <- dimnames(x)[[1]]
        x <- x[i,  , drop = F]
        attr(x, "directed") <- xdir[i]
        class(x) <- cl
        x
}
```

This is easy to write since we inherit the subscripting from matrices. Not in the technical sense since matrices are not object-oriented in version 3, but it is inheritance nonetheless.

Note the `drop = F`, we don't want to lose the matrixness in case the result has only one edge.

The assignment form of subscripting follows the same pattern.

```
"[<-.mathgraph"<-
function(x, i, value)
{
        if(!inherits(value, "mathgraph"))
                stop("need mathgraph on right-hand side")
        cl <- class(x)
        x <- unclass(x)
        value <- unclass(value)
        ilen <- length(i)
        if(is.logical(i))
                ilen <- sum(rep(i, length = nrow(x)))
        if(nrow(value) != ilen)
                stop("replacement value not correct length")
        xdir <- attr(x, "directed")
        if(is.character(i))
                names(xdir) <- dimnames(x)[[1]]
        x[i,  ] <- value
        xdir[i] <- attr(value, "directed")
        attr(x, "directed") <- xdir
        class(x) <- cl
        x
}
```

The lines involving `ilen` are to avoid the problem described on page 122.

Let's see how they work.

```
> jjmg2
d   node 4 <-> node 1
e   node 5 <-> node 1
f   node 6 <-> node 1
g   node 2 <-> node 4
h   node 3 <-> node 4
i   node 2 <-> node 6
j   node 3 <-> node 6

class: mathgraph
> jjmg2[2:3]
e   node 5 <-> node 1
f   node 6 <-> node 1

class: mathgraph
> jjmg2[c("f","j","f")]
f   node 6 <-> node 1
j   node 3 <-> node 6
f   node 6 <-> node 1

class: mathgraph
> jjmg2[c(F,T)]
e   node 5 <-> node 1
g   node 2 <-> node 4
i   node 2 <-> node 6

class: mathgraph
```

So subscripting is very similar to subscripting a typical vector. Replacement is also similar to the case with vectors.

```
> jjmg2[2:4] <- jjmg1
> jjmg2
d   node 4 <-> node 1
e   node 1 <-> node 2
f   node 2 <-> node 3
g   node 3 <-> node 4
h   node 3 <-> node 4
i   node 2 <-> node 6
j   node 3 <-> node 6

class: mathgraph
```

Sometimes we want to deal only with directed graphs. We can create a generic function to turn non-directed graphs into directed ones. So the result looks like:

```
> jjmg1
a   node 1 <-> node 2
b   node 2 <-> node 3
c   node 3 <-> node 4

class: mathgraph
> alldirected(jjmg1)
a   node 1  -> node 2
b   node 2  -> node 3
c   node 3  -> node 4
a   node 2  -> node 1
b   node 3  -> node 2
c   node 4  -> node 3

class: mathgraph
```

The default method merely creates an error.

```
"alldirected.default"<-
function(x, ...)
{
        stop("do not know how to handle")
}
```

In a sense this is redundant since an error will occur if the default method is needed, but doesn't exist. However, when it is written, it means that the programmer has thought over the situation.

Here is the method for `mathgraph`.

```
"alldirected.mathgraph"<-
function(x)
{
        dir <- attr(x, "directed")
        if(all(dir))
                return(x)
        x <- unclass(x)
        ans <- rbind(x, x[!dir, 2:1])
        attr(ans, "directed") <- rep(T, nrow(ans))
        class(ans) <- "mathgraph"
        ans
}
```

This takes all of the input edges and adds another edge in reverse order for each undirected edge.

Note that the class of the output is not guaranteed to be the same as that of the input. This function could be used by a special type of mathematical graph

that inherits from `mathgraph`. By not passing the class of the input to the output, we are making sure that we don't end up with an object that doesn't make sense—it could be that that special class of graph doesn't make sense for directed graphs. It is subtle distinctions like this that are the hardest part of object-oriented programming. Whether you have a monument or a pile of rubble can depend on getting these decisions right.

We can combine `mathgraph` objects by calling `mathgraph` again.

```
> jjmg3 <- jjmg1
> jjmg4 <- jjmg2
> class(jjmg3) <- c("subA", "mathgraph")
> class(jjmg4) <- c("subA", "mathgraph")
> mathgraph(~ jjmg3 + jjmg4)
a   node 1 <-> node 2
b   node 2 <-> node 3
c   node 3 <-> node 4
d   node 4 <-> node 1
e   node 1 <-> node 2
f   node 2 <-> node 3
g   node 3 <-> node 4
h   node 3 <-> node 4
i   node 2 <-> node 6
j   node 3 <-> node 6

class: mathgraph
```

But it is more convenient to have a method for the `c` function. This also allows us to retain subclasses of `mathgraph` if appropriate.

```
> c(jjmg4, jjmg3)
d   node 4 <-> node 1
e   node 1 <-> node 2
f   node 2 <-> node 3
g   node 3 <-> node 4
h   node 3 <-> node 4
i   node 2 <-> node 6
j   node 3 <-> node 6
a   node 1 <-> node 2
b   node 2 <-> node 3
c   node 3 <-> node 4

class: subA mathgraph
> c(jjmg1, jjmg3)
a   node 1 <-> node 2
b   node 2 <-> node 3
```

```
c   node 3 <-> node 4
a   node 1 <-> node 2
b   node 2 <-> node 3
c   node 3 <-> node 4

class: mathgraph
> jjmg5 <- jjmg1
> class(jjmg5) <- c("subB", "mathgraph")
> c(jjmg5, jjmg3)
a   node 1 <-> node 2
b   node 2 <-> node 3
c   node 3 <-> node 4
a   node 1 <-> node 2
b   node 2 <-> node 3
c   node 3 <-> node 4

class: mathgraph
```

So the class of the result of a call to `c.mathgraph` will be the most specific set of classes that all of the inputs have in common. For many types of object this is the proper behavior. But it isn't necessarily in this case—if the subclass of `mathgraph` represents trees, the combination of two or more trees doesn't guarantee that the result is a tree.

The definition of the `c` method is:

```
"c.mathgraph"<-
function(...)
{
        dots <- list(...)
        the.class <- commontail(lapply(dots, class))
        if(!match("mathgraph", the.class, nomatch = 0)
                )
                stop("not all mathgraph objects")
        amat <- do.call("rbind", dots)
        adir <- unlist(lapply(dots, function(x)
        attr(x, "directed")))
        if(dim(amat)[1] != length(adir))
                stop("garbled object")
        attr(amat, "directed") <- adir
        class(amat) <- the.class
        amat
}
```

There is a check at the end to make sure that the `directed` attribute matches the rest of the object. Without the check, there is the possibility of trouble that will be hard to track down.

We wouldn't mind having an optional argument to determine how to treat the class, but version 3 (at least) of S won't allow it (the default method of `c` gets used when the optional argument is in the call).

The mechanism that decides what the class resulting from `c.mathgraph` should be is the `commontail` function. Its definition is:

```
> commontail
function(x)
{
        the.lens <- unlist(lapply(x, length))
        min.len <- min(the.lens)
        if(min.len == 0)
                return(NULL)
        ans <- NULL
        x <- lapply(x, rev)
        for(i in 1:min.len) {
                this.ans <- unique(unlist(lapply(x,
                        function(y, i)
                y[i], i = i)))
                if(length(this.ans) == 1) {
                        ans <- c(this.ans, ans)
                }
                else {
                        break
                }
        }
        ans
}
```

You'll notice that `lapply` plays prominently in this function. Once we know that there is really some work to do, the order is reversed in each component to make it easier. Inside the `for` loop there is a decision based on there being a unique element in all of the components.

If we don't want to allow edges of the same type between the same nodes, then we can use `unique.mathgraph`.

```
"unique.mathgraph"<-
function(x)
{
        dir <- attr(x, "directed")
        cl <- class(x)
        x <- unclass(x)
        xdup <- duplicated(paste(x[, 1], x[, 2], dir))
        x <- x[!xdup, , drop = F]
        dir <- dir[!xdup]
```

```
        attr(x, "directed") <- dir
        class(x) <- cl
        x
}
```

This uses the usual trick of `unique` which is to delete duplicated items.

The `sort.mathgraph` function makes it easier to look at printed graphs. Note that `sort` is not generic in (at least) lots of versions, so you will probably need to type the full name.

```
"sort.mathgraph"<-
function(x, nodes = T, edges = T)
{
        dir <- attr(x, "directed")
        cl <- class(x)
        x <- unclass(x)
        if(nodes && !all(dir)) {
                x[!dir,  ] <- stable.apply(x[!dir,  ,
                        drop = F], 1, sort)
        }
        if(edges) {
                ord <- order(x[, 1], x[, 2])
                x <- x[ord,  ]
                attr(x, "directed") <- dir[ord]
        }
        class(x) <- cl
        x
}
```

Note that `unclass` is necessary because we are subscripting in the sense of the matrix underlying the `mathgraph` object, not in the sense of subscripting the edges.

This sort routine allows both sorting of the nodes within an undirected edge, and sorting of the edges.

One useful representation of mathematical graphs is an adjacency matrix. This is a square matrix in which the dimension is the number of nodes. There is a "1" in location i, j if there is an edge from node i to node j. So if all of the edges are undirected, then the matrix is symmetric.

Here's a function to transform a `mathgraph` object into the equivalent adjacency matrix.

```
"adjamat.mathgraph"<-
function(x, general = F)
{
```

```
        x <- unclass(x)
        xdir <- attr(x, "directed")
        ischar <- is.character(x)
        if(ischar) {
                nnam <- unique(x)
                nnode <- length(nnam)
                has.names <- T
        }
        else {
                if(!is.numeric(x))
                        stop("nodes must be character or numeric")
                nnode <- max(x)
                nnam <- paste("node", 1:nnode)
                has.names <- F
        }
        ans <- array(0, c(nnode, nnode), list(nnam,
                nnam))
        if(ischar) {
                dx <- dim(x)
                x <- match(x, nnam)
                dim(x) <- dx
        }
        ans[x] <- 1
        if(any(!xdir))
                ans[x[!xdir, 2:1]] <- 1
        if(general) {
                stop("general version not implemented")
        }
        attr(ans, "call") <- match.call()
        attr(ans, "has.names") <- has.names
        class(ans) <- "adjamat"
        ans
}
```

Making the function above a method for a generic function is sensible.

```
"adjamat"<-
function(x, ...)
UseMethod("adjamat")
```

An example where the nodes are character is:

```
> adjamat(mathgraph(~ state.name[1:3]/state.name[2:4]))
         Alabama Alaska Arizona Arkansas
 Alabama       0      1       0        0
  Alaska       1      0       1        0
```

```
 Arizona       0      1       0        1
Arkansas       0      0       1        0
attr(, "call"):
adjamat.mathgraph(x = mathgraph( ~ state.name[1:3]/
        state.name[2:4]))
attr(, "has.names"):
[1] T
attr(, "class"):
[1] "adjamat"
```

Another representation of a graph is an incidence matrix. This matrix has as many rows as nodes in the graph, and the number of columns is equal to the number of directed edges plus twice the number of undirected edges.

Incidence matrices have an advantage over adjacency matrices in that the latter destroy information about the edges—we can't retain the names of the edges in an adjacency matrix without essentially keeping the original `mathgraph` object. However, there are typically more edges than nodes in a graph, so incidence matrices are usually going to be bigger than adjacency matrices. Neither are very frugal with space.

Here is the function to transform a `mathgraph` object into an incidence matrix. This is simplified by taking out the code that makes sure that loops (an edge that begins and ends on the same node) behave properly. As is common, an unimportant case like this involves a major part of the function.

```
"incidmat.mathgraph"<-
function(x, expand = T, general = F)
{
        x <- unclass(x)
        xdir <- attr(x, "directed")
        nedge <- dim(x)[1]
        eseq <- 1:nedge
        cnam <- dimnames(x)[[1]]
        has.names <- c(nodes = is.character(x), edges
                 = T)
        if(!length(cnam)) {
                has.names["edges"] <- F
                cnam <- paste(ifelse(xdir, "arc",
                        "edge"), eseq)
        }
        ischar <- is.character(x)
        if(ischar) {
                nnam <- unique(x)
                nnode <- length(nnam)
                dx <- dim(x)
                x <- match(x, nnam)
```

```
                dim(x) <- dx
        }
        else {
                if(!is.numeric(x))
                        stop("nodes must be character or numeric")
                nnode <- max(x)
                nnam <- paste("node", 1:nnode)
        }
        loops <- x[, 1] == x[, 2]
        if(expand && !all(xdir)) {
                cnam <- rep(cnam, 2 - xdir)
                ans <- array(0, c(nnode, length(cnam)),
                        list(nnam, cnam))
                reseq <- match(eseq, rep(eseq, 2 -
                        xdir))
                ans[cbind(x[, 2], reseq)] <- -1
                ans[cbind(x[, 1], reseq)] <- 1
                ans[cbind(x[!xdir, 1], reseq[!xdir] +
                        1)] <- -1
                ans[cbind(x[!xdir, 2], reseq[!xdir] +
                        1)] <- 1
        }
        else {
                ans <- array(0, c(nnode, nedge), list(
                        nnam, cnam))
                ans[cbind(x[, 1], eseq)] <- 1
                ans[cbind(x[, 2], eseq)] <- ifelse(
                        xdir, -1, 1)
        }
        attr(ans, "has.names") <- has.names
        attr(ans, "call") <- match.call()
        class(ans) <- "incidmat"
        ans
}
```

After some initial bureaucracy, the work really gets underway in the `if(expand)`. Expanding means that undirected edges are broken into two directed edges—so `cnam` is replicated accordingly. The `reseq` object is the indices corresponding to the first column representing each edge. This is then used to put values in the appropriate places, using a matrix as the object inside the subscripts. The `else` clause is similar.

As stated earlier, this listing leaves out how loops are handled—not because it is hard, but because it is a lot of code for not much gain. My first attempt at this function tried to be clever, and handle loops along with everything else in only a couple more lines than you see here. The problem with that was that it didn't work for all cases.

So there are two traps to fall into:

• You can subscribe always to cleverness, in which case your code may have surprising bugs. Attempts to fix these bugs may lead to further knots of code that can easily create new bugs.

• You can avoid cleverness, thus making code that is long, tedious, and perhaps inefficient. As more functionality is needed, the code gets ever longer, because simple solutions are hidden in the volumes of code.

Don't do either.

The `incidmat` function is generic, we call it to use the `incidmat.mathgraph` function.

```
> incidmat(mathgraph(~ 1:3/2:4))
       edge 1 edge 1 edge 2 edge 2 edge 3 edge 3
node 1      1     -1      0      0      0      0
node 2     -1      1      1     -1      0      0
node 3      0      0     -1      1      1     -1
node 4      0      0      0      0     -1      1
attr(, "has.names"):
 nodes edges
     F     F
attr(, "call"):
incidmat.mathgraph(x = mathgraph( ~ 1:3/2:4))
attr(, "class"):
[1] "incidmat"
> incidmat(mathgraph(~ 1:3/2:4), ex=F)
       edge 1 edge 2 edge 3
node 1      1      0      0
node 2      1      1      0
node 3      0      1      1
node 4      0      0      1
attr(, "has.names"):
 nodes edges
     F     F
attr(, "call"):
incidmat.mathgraph(x = mathgraph( ~ 1:3/2:4), expand = F)
attr(, "class"):
[1] "incidmat"
```

Mathematical graphs are built not for their own sake, but to learn things about them. For example, we may be interested in a path from one node to some other node. The `getpath.adjamat` function does this when the graph is represented by an adjacency matrix. This is very similar to `getpath.incidmat` which is shown here.

```
"getpath.incidmat"<-
function(x, start, end)
{
        if(start == end)
                return(mathgraph())
        x <- unclass(x)
        dircheck <- rep(1, dim(x)[1]) %*% x
        if(any(dircheck))
                stop("need matrix for directed graph")
        has.names <- attr(x, "has.names")
        if(has.names["edges"])
                enames <- dimnames(x)[[2]]
        else enames <- NULL
        if(has.names["nodes"])
                node.names <- dimnames(x)[[1]]
        else node.names <- NULL
        if(is.character(c(end, start))) {
                start <- match(start, node.names, nomatch = NA)
                end <- match(end, node.names, nomatch = NA)
                bad.in <- c("start", "end")[is.na(c(start, end))]
                if(length(bad.in))
                        stop(paste(paste(bad.in, collapse = " and "),
                                "not right"))
        }
        tset <- start
        prev <- 0
        edges <- 0
        unchecked <- T
        nseq <- 1:dim(x)[1]
        eseq <- 1:dim(x)[2]
        repeat {
                this.index <- (1:length(unchecked))[unchecked][1]
                newe <- eseq[x[tset[this.index],  ] > 0.5]
                if(length(newe)) {
                        newn <- row(x[, newe, drop = F])[as.vector(x[, newe] <
                                -0.5
                                )]
                        newt <- !duplicated(newn)
                        newn <- newn[newt]
                        newe <- newe[newt]
                        newt <- match(newn, tset, nomatch = 0) == 0
                        newn <- newn[newt]
                        newe <- newe[newt]
                }
                else newn <- NULL
                unchecked[this.index] <- F
                if(n <- length(newn)) {
                        if(endind <- match(end, newn, nomatch = 0)) {
# have a path
                                tset <- c(tset[!unchecked], end)
```

```
                                prev <- c(prev[!unchecked], tset[this.index])
                                edges <- c(edges[!unchecked], newe[endind])
                                pseq <- 1:length(tset)
                                path <- this.index <- length(tset)
                                this.node <- end
                                while(prev[this.index] != start) {
                                  this.index <- pseq[tset == prev[this.index]]
                                  path <- c(this.index, path)
                                }
                                if(has.names["nodes"]) {
                                  ans <- mathgraph( ~ node.names[prev[path]]/
                                    node.names[tset[path]], dir = T)
                                }
                                else {
                                  ans <- mathgraph( ~ prev[path]/tset[path],
                                    dir = T)
                                }
                                if(has.names["edges"]) {
                                  names(ans) <- enames[edges[path]]
                                }
                                return(ans)
                        }
                        tset <- c(tset, newn)
                        edges <- c(edges, newe)
                        prev <- c(prev, rep(tset[this.index], n))
                        unchecked <- c(unchecked, rep(T, n))
                }
                if(!any(unchecked))
                        return(NULL)
        }
}
```

By seeing a `repeat` loop in which there is a great deal of creating and enlarging objects, you might have the idea that S is not the ideal place for this code. You'd be right. A C implementation is going to be miles more efficient than the S code, however, it will take substantially longer to code also. A good implementation would have `getpath.mathgraph` pass the matrix of the `mathgraph` object down into C to do the work and pass the answer back, at which point S would construct the appropriate object to return. The current `getpath.mathgraph` makes an incidence matrix of the graph and then uses the `incidmat` method of `getpath`.

Here is `getpath` in action.

```
> jjmgs2 <- mathgraph(~ state.name[1:3]*state.name[2:4] +
+    state.name[22:23]*state.name[24:25] +
+    state.name[2]/state.name[23], dir=T)
> getpath(jjmgs2, "Alabama", "Missouri")
[1]   Alabama  ->  Alaska
```

```
[2]   Alaska  ->  Minnesota
[3]   Minnesota  ->  Missouri

class: mathgraph
```

Of course it is natural to think of mathematical graphs visually, which means that you should be able to plot them. Here is a quick and decidedly dirty `plot` method for `mathgraph` objects.

```
"plot.mathgraph"<-
function(x, ...)
{
        x <- unclass(x)
        xdir <- attr(x, "directed")
        if(ischar <- is.character(x)) {
                node.names <- unique(x)
                maxx <- length(node.names)
                dx <- dim(x)
                x <- match(x, node.names)
                dim(x) <- dx
        }
        else {
                maxx <- max(x)
                node.names <- as.character(1:maxx)
        }
        px <- cos((2 * 0:(maxx - 1) * pi)/maxx)
        py <- sin((2 * 0:(maxx - 1) * pi)/maxx)
        plot(px, py, axes = F, xlab = "", ylab = "",
                xlim = c(-1.04, 1.04), ylim = c(-1.04,
                1.04), ...)
        box()
        px <- 0.98 * px
        py <- 0.98 * py
        if(!all(xdir))
                segments(px[x[!xdir, 1]], py[x[!xdir,
                        1]], px[x[!xdir, 2]], py[x[!
                        xdir, 2]])
        if(any(xdir))
                arrows(px[x[xdir, 1]], py[x[xdir, 1]],
                        px[x[xdir, 2]], py[x[xdir, 2]]
                        )
        text(px * 1.07, py * 1.07, node.names)
        invisible()
}
```

13.4 Things to Do

There is a bug in the version of `unique.mathgraph` listed on page 309—trust me, you shouldn't trust me. What is the bug? How would you fix it?

Write `adjamat.incidmat` and `incidmat.adjamat`.

Is subscripting feasible for classes `adjamat` and `incidmat`?

Are there operations that should be added to the formulas for `mathgraph`? How should adjacency and incidence matrix representations be coerced to `mathgraph` objects? What other additions should be made to the `mathgraph` implementation?

In many applications the edges of a graph have a capacity—for example, a telephone system. Create a new class that inherits from `mathgraph` that has capacities for the edges.

Write a `plot.mathgraph` that does what we really want to happen. Can you allow the user to have control over the exact output?

Write a suite of functions to perform various operations with mathematical graphs. That is, functions similar in spirit to `getpath`. How do you test these functions?

Write a function that generates random graphs.

Write a function that can benefit from having at least one argument be a formula. What differences would there be in an implementation without formulas?

13.5 Further Reading

A book on mathematical graphs is Chachra, Ghare and Moore (1979) *Applications of Graph Theory Algorithms*.

Chapter 14

Functions

This chapter concentrates on functions that have functions as arguments, or that return functions.

14.1 Numerical Integration

Numerical integration (also known as quadrature) is often useful. The following function performs contour integrals in the complex plane.

```
"fjjline.integral"<-
function(FUN, POINTS, EVALS = 100, ...)
{
        n <- length(POINTS)
        if(n < 2)
                stop("POINTS must have length at least 2")
        if(is.character(FUN))
                FUN <- get(FUN, mode = "function")
        ans <- 0
        for(i in 2:n) {
                this.seq <- seq(POINTS[i - 1], POINTS[
                        i], length = EVALS)
                this.ans <- FUN(this.seq, ...)
                ans <- ans + (sum(this.ans) - 0.5 *
                        sum(this.ans[c(1, EVALS)])) * (
                        this.seq[2] - this.seq[1])
        }
        ans
}
```

This function takes a function, a vector of points that are the vertices of the contour along which the integration takes place, and the number of function

evaluations to do in each segment of the contour. Additional arguments for the function being integrated are also accepted. The names of the arguments to `fjjline.integral` are all in upper case so that their names will not interfere with the names of arguments of the function being integrated.

The method of integration is quite naive—in a closed contour like the example below, each point that is evaluated is given the same weight.

```
> fjjline.integral(function(z) 1/z, complex(re=c(1,1,-1,-1,1),
+         im=c(-1,1,1,-1,-1)))
[1] 1.25601e-16+6.283049i
> pi*2i
[1] 0+6.283185i
```

From theory we know that the true value of this integral is $2\pi i$ so we can see how close the numerical integration is to the right answer—it seems to be trying to get it right, but is not terribly accurate.

Now try another integral for which we know the correct answer.

```
> fjjline.integral(function(z) 1, complex(re=c(1,2),
+         im=c(-1,1)))
[1] NA
```

This time we are met with failure. The problem is that the integrand is required to be vectorized, and the function in this example is not.

With this experience in hand, we can attempt a better version. Not wanting to appear naive, we call the method in the previous version the "trapezoid" method.

```
"line.integral"<-
function(FUN, POINTS, EVALS = 100, METHOD = "simpson",
        ...)
{
        METHOD <- unabbrev.value(METHOD, c("trapezoid",
                "simpson"))
        n <- length(POINTS)
        if(n < 2)
                stop("POINTS must have length at least 2")
        switch(METHOD,
                simpson = {
# want odd number of evaluation points
                        if(EVALS %% 2 == 0) EVALS <-
                                EVALS + 1
                }
                )
        if(is.character(FUN))
                FUN <- get(FUN, mode = "function")
```

```
        ans <- 0
        for(i in 2:n) {
                this.seq <- seq(POINTS[i - 1], POINTS[
                        i], length = EVALS)
                this.ans <- FUN(this.seq, ...)
                if(length(this.ans) != EVALS)
                        stop("FUN not properly vectorized")
                ans <- ans + switch(METHOD,
                        trapezoid = {
                                (sum(this.ans) - 0.5 *
                                  sum(this.ans[c(1,
                                  EVALS)])) * (
                                  this.seq[2] -
                                  this.seq[1])
                        }
                        ,
                        simpson = {
                                sum(this.ans * c(1,
                                  rep(c(4, 2), length
                                   = EVALS - 2), 1))/3 *
                                  (this.seq[2] -
                                  this.seq[1])
                        }
                        )
        }
        ans
}
```

The result of both of the previous examples is now much improved.

```
> line.integral(function(z) 1/z, complex(re=c(1,1,-1,-1,1),
+         im=c(-1,1,1,-1,-1)))
[1] 1.480297e-16+6.283185i
> pi * 2i
[1] 0+6.283185i
> line.integral(function(z) 1, complex(re=c(1,2),
+         im=c(-1,1)))
Error in line.integral(function(z): FUN not properly
        vectorized
Dumped
```

Note that the check on vectorization is not foolproof—if you use `max` where you should use `pmax`, for instance, the function can give answers that are the correct length but are wrong.

Real-valued integrations can be done by both `line.integral` and the S-PLUS function `integrate`, so the results can be compared. In the following

case, we are subtracting a very accurate way of evaluating a value from the numerical integration.

```
> integrate(dnorm, -1, 1)$integral - diff(pnorm(c(-1,1)))
[1] -2.220446e-16
> line.integral(dnorm, c(-1, 1), EV=5000) - diff(pnorm(c(-1,1)))
[1] -7.249756e-14
```

Even with a lot of evaluations, Simpson's method is significantly less accurate than the adaptive technique that `integrate` employs. However, `line.integral` takes only about half as much time as `integrate` for this problem.

```
> p.unix.time(for(i in 1:100) line.integral(dnorm,
+       c(-1, 1), EV=5000))
 comp time clock time  memory
      6.89          7 1763672
> p.unix.time(for(i in 1:100) integrate(dnorm, -1, 1))
 comp time clock time  memory
     12.85         13 2511336
```

The difference in both accuracy and computation time will vary from problem to problem—my suspicion is that this probably overstates the time advantage of `line.integral` for the typical problem.

In applications where speed is more important than accuracy, it could be profitable to substitute `line.integral` for `integrate`. However, if the integrand is not nicely behaved, then `integrate` is likely to do much better.

14.2 Interpolation

Function interpolation is useful when it is expensive to evaluate the function. The strategy is to evaluate the function exactly at a few points, and then to approximate the function at other points within the interval in which the exact values are known.

One of the better methods is Lagrange interpolation. If we know the value of function f at n points x_i, then the Lagrange interpolation formula for the approximating function $\tilde{f}$ is

$$\tilde{f}(x) = \sum_{k=1}^{n} l_k(x) f(x_k) \tag{14.1}$$

where

$$l_k(x) = \frac{\prod_{i \neq k} (x - x_i)}{\prod_{i \neq k} (x_k - x_i)} \tag{14.2}$$

To get a function that interpolates for the sine function we need to start with a vector of x-values and the sine of those values. The next task is to translate Lagrange's formula into S. First we get the known values:

```
> jjsx <- seq(0, pi/2, len=5)
> jjsy <- sin(jjsx)
```

Then we write the function to do interpolation with them:

```
function(x)
{
        x.know <- jjsx
        y.know <- jjsy
        if(length(x) != 1)
                stop("x must have length 1")
        if(x < min(x.know) || x > max(x.know))
                stop("x not in interval")
        tab <- outer(x.know, x.know, "-")
        diag(tab) <- 1
        den <- apply(tab, 1, prod)
        tab <- outer(rep(x, length(x.know)), x.know,
                "-")
        diag(tab) <- 1
        num <- apply(tab, 1, prod)
        sum((y.know * num)/den)
}
```

Obviously we don't really care about having a function that approximates the sine—S already has a function that approximates the sine very well. What we do want is a way of creating an S function that approximates a function of our choice. The following function takes a vector of x values and a vector of corresponding values of the mathematical function of interest and returns an S function that is the Lagrange interpolation of the data.

```
"fjjinterpolator.lagrange"<-
function(x, y)
{
        if(length(x) != length(y))
                stop("x and y must be the same length")
        ans <- function(z)
        {
        }
        body <- expression(if(length(z) != 1) stop(
                        "z must have length 1"), NULL,
                NULL, NULL, NULL, sum(y.den * apply(
                tab, 1, prod)))
        mode(body) <- "{"
```

```
        body[[2]] <- substitute(if(z < minx || z >
                maxx) stop("z not in interval of known x values"
                        ), list(minx = min(x), maxx =
                max(x)))
        tab <- outer(x, x, "-")
        diag(tab) <- 1
        den <- apply(tab, 1, prod)
        body[[3]] <- call("assign", "y.den", y/den)
        body[[4]] <- substitute(tab <- outer(rep(z,
                xlen), x, "-"), list(xlen = length(x),
                x = x))
        body[[5]] <- parse(text = "diag(tab) <- 1")[[1
                ]]
        ans[[length(ans)]] <- body
        ans
}
```

The result of this function will be a function of one argument, so the `ans` variable is assigned a one-argument function—the argument will be called `z`. What the function actually does comes later. The start of this is to create `body` that is an expression of the correct length and has a couple of the commands filled in. Then `body` is coerced to be of mode "brace"—this is the mode we want it to end up being, and mode expression has exhausted its usefulness at this point. Skipping to the end, `body` is attached to `ans` and `ans` is returned. *Body my house my horse my hound* [37]

In the middle of the function there are three methods of adding commands to `body`. `substitute` provides very useful functionality—it takes an expression as its first argument and a named list as its second. Values of variables that are in the list are substituted into the expression; other names are left as is. When a command is to be created and there is nothing to substitute in, then you can use `call`, or you can use the `text` argument in `parse`.

Here is the result with the sine data.

```
> fjjinterpolator.lagrange(jjsx, jjsy)
function(z)
{
        if(length(z) != 1)
                stop("z must have length 1")
        if(z < 0 || z > 1.5707963267949)
                stop("z not in interval of known x values")
        assign("y.den", c(0, -2.68193882509428,
                7.43336516385593, -6.47477308499758,
                1.75206097146613))
        tab <- outer(rep(z, 5), c(0, 0.392699081698724,
                0.785398163397448, 1.17809724509617,
                1.5707963267949), "-")
```

```
        diag(tab) <- 1
        sum(y.den * apply(tab, 1, prod))
}
```

The major shortcoming of this is that it is not vectorized. So we go to work and come up with a function to interpolate sine that is vectorized.

```
function(z)
{
        out <- (z < min(jjsx) | z > max(jjsx))
        z[out] <- NA
        tab <- outer(jjsx, jjsx, "-")
        diag(tab) <- 1
        den <- apply(tab, 1, prod)
        zlen <- length(z)
        tab <- outer(array(rep(z, rep(length(den),
                zlen)), c(length(den), zlen)), jjsx,
                "-")
        tab <- aperm(tab, c(1, 3, 2))
        tab[row(tab) == col(tab)] <- 1
        num <- apply(tab, c(1, 3), prod)
        drop(rep(1, length(jjsy)) %*% ((jjsy * num)/
                den))
}
```

The single-value version required `outer` to create a matrix. This merely adds another dimension for the values in the input, so now `outer` creates a three-dimensional array. Once this function is written and we know that it works, we can use it as a template.

```
"interpolator.lagrange"<-
function(x, y)
{
        if(length(x) != length(y))
                stop("x and y must be the same length")
        ans <- function(z)
        {
        }
        body <- expression(zlen <- length(z), NULL,
                NULL, z[zout] <- NA, NULL, tabz <-
                array(rep(z, rep(ilen, zlen)), c(ilen,
                zlen)), NULL, tab <- aperm(tab, c(1, 3,
                2)), tab[row(tab) == col(tab)] <- 1,
                num <- apply(tab, c(1, 3), prod), ans <-
                rep(1, ilen) %*% (y.den * num),
                attributes(ans) <- attributes(z), ans)
```

```
        mode(body) <- "{"
        body[[2]] <- substitute(ilen <- length.x, list(
                length.x = length(x)))
        body[[3]] <- substitute(zout <- (z < minx | z >
                maxx), list(minx = min(x), maxx = max(
                x)))
        tab <- outer(x, x, "-")
        diag(tab) <- 1
        den <- apply(tab, 1, prod)
        body[[5]] <- call("assign", "y.den", y/den)
        body[[7]] <- substitute(tab <- outer(tabz, x,
                "-"), list(x = x))
        ans[[length(ans)]] <- body
        test <- all.equal(ans(x), y)
        if(!is.logical(test) || !test)
                stop("function didn't build correctly")
        ans
}
```

This includes a test at the end to see if the resulting function looks reasonable. We know that the interpolation should be exact at the input points, and this is an easy thing to check. When we humans solve a problem, it is good practice to check if the answer seems sensible. There's no reason that programs shouldn't do the same.

Here we try it out.

```
> fjjsin.il <- interpolator.lagrange(jjsx, jjsy)
> fjjsin.il
function(z)
{
        zlen <- length(z)
        ilen <- 5
        zout <- (z < 0 | z > 1.5707963267949)
        z[zout] <- NA
        assign("y.den", c(0, -2.68193882509428,
                7.43336516385593, -6.47477308499758,
                1.75206097146613))
        tabz <- array(rep(z, rep(ilen, zlen)), c(ilen,
                zlen))
        tab <- outer(tabz, c(0, 0.392699081698724,
                0.785398163397448, 1.17809724509617,
                1.5707963267949), "-")
        tab <- aperm(tab, c(1, 3, 2))
        tab[row(tab) == col(tab)] <- 1
        num <- apply(tab, c(1, 3), prod)
        ans <- rep(1, ilen) %*% (y.den * num)
```

```
        attributes(ans) <- attributes(z)
        ans
}
```

We can see how it does by plotting the error of the Lagrange interpolation and the error of linear interpolation.

```
> jjx1 <- seq(0, pi/2, len=200)
> matplot(jjx1, cbind(approx(jjsx, jjsy, xout=jjx1)$y,
+       fjjsin.il(jjx1)) - sin(jjx1), type="l")
```

Note that you can get terrible performance from interpolation if your function does not behave the way that the interpolation method presumes of functions. Interpolation is not something to do thoughtlessly.

14.3 Genetic Algorithms

Genetic algorithms are a type of random algorithm that are used for optimization (among other things). The basic idea is to have a population of solutions that improves relative to the objective through a "survival of the fittest" mechanism. Reasons to use a genetic algorithm rather than something like a Newton-Raphson or conjugate gradient technique are:

- There are multiple local optima.
- The objective function is not differentiable.
- The number of parameters is very large, so gradient techniques are very slow.

As in Hollywood movies, the main ingredients for a genetic algorithm are sex and violence. There needs to be a mechanism for children to be born based on a random selection of attributes of members of the population. Then there needs to be a way to decide who is to live so that the population size remains constant (or at least relatively constant).

The following function implements a crude genetic algorithm. The basic form of it, though, resembles many of the algorithms that I use in practice. Two members of the population are selected at random, a child is created by mixing the elements from the two parents. The best two of the three survive. Someone once called this the Oedipus algorithm—kill your father, marry your mother. If a child survives, then it is "jittered" in an attempt to find a better solution nearby. This function operates on logical vectors, as traditional genetic algorithms do.

```
"fjjgenop1"<-
function(FUN, POP, BIRTHS = 100, JITTERS = 3, PROB = 0.3, MUTATE =
        0.02, TRACE = T, ...)
{
```

```
random.seed <- .Random.seed
if(is.character(FUN))
        FUN <- get(FUN)
if(!is.matrix(POP))
        stop("bad input for POP")
if(!is.logical(POP))
        stop("this is for logical populations only")
popsize <- ncol(POP)
objective <- numeric(popsize)
for(i in 1:popsize) {
        objective[i] <- FUN(POP[, i], ...)
}
funevals <- popsize
npar <- nrow(POP)
if(PROB <= 0 || PROB >= 1)
        stop("bad value for PROB")
PROB <- c(PROB, 1 - PROB)
if(MUTATE <= 0 || MUTATE >= 1)
        stop("bad value for MUTATE")
MUTATE <- c(MUTATE, 1 - MUTATE)
if(TRACE)
        cat("objectives are from", min(objective), "to", max(
                objective), "\n")
for(i in 1:BIRTHS) {
        parents <- sample(popsize, 2, rep = F)
        child <- POP[, parents[1]]
        cloc <- sample(c(F, T), npar, rep = T, prob = PROB)
        if(all(cloc))
                cloc[sample(npar, 1)] <- F
        else if(all(!cloc))
                cloc[sample(npar, 1)] <- T
        child[cloc] <- POP[cloc, parents[2]]
        child.obj <- FUN(child, ...)
        funevals <- funevals + 1
        survive <- child.obj < max(objective[parents])
        if(TRACE)
                cat("parents:", format(objective[parents]),
                        "child:", format(child.obj), if(
                        survive) "(improve)", "\n")
        if(survive) {
                if(objective[parents[1]] > objective[parents[
                        2]])
                        out <- parents[1]
                else out <- parents[2]
                if(TRACE && child.obj < min(objective))
                        cat("new minimum\n")
                POP[, out] <- child
                objective[out] <- child.obj
                for(i in seq(length = JITTERS)) {
                        cloc <- sample(c(T, F), npar, rep = T,
```

```
                                prob = MUTATE)
                            if(all(!cloc))
                              cloc[sample(npar, 1)] <- T
                            jchild <- child
                            jchild[cloc] <- !child[cloc]
                            jchild.obj <- FUN(jchild, ...)
                            funevals <- funevals + 1
                            if(jchild.obj < child.obj) {
                              if(TRACE) {
                                cat("jitter successful,",
                                  "new objective", jchild.obj,
                                  "\n")
                                if(jchild.obj < min(objective))
                                  cat(" new minimum\n")
                              }
                              POP[, out] <- child <- jchild
                              objective[out] <- child.obj <-
                                jchild.obj
                            }
                        }
                    }
            }
            ord <- order(objective)
            list(population = POP[, ord], objective = objective[ord],
                    funevals = funevals, random.seed = random.seed, call
                     = match.call())
}
```

The main purpose for writing this function is to compare how well it does on one of the problems studied by Jennison and Sheehan (1995). They studied the simple problem of maximizing

$$g(x) = \sum_{i=1}^{p} a_i x_i \qquad x_i \in \{0, 1\}$$

All of the a_i are positive so the function is obviously maximized when all of the x_i are 1 (or `TRUE` in our function). Jennison and Sheehan give the complete details only for their 20 dimensional problem, so that is the one we'll use. Here is the S function to be minimized.

```
> fjj.jsa20
function(x)
 - sum(x * jsa20)
> jsa20
 [1] 2.10 1.89 1.03 1.69 1.24 1.02 3.36 1.43 2.15 1.43
[11] 1.54 2.35 2.39 1.12 1.00 3.66 1.09 1.10 3.01 1.18
```

The following function keeps track of how well `fjjgenop1` does on the problem for a number of evaluations.

```
"fjjgentest1"<-
function(trials = 100, ...)
{
        funev <- numbest <- numeric(trials)
        for(i in 1:trials) {
                this.one <- fjjgenop1(fjj.jsa20,
                        matrix(sample(c(F, T), rep = T,
                        400), 20), ...)
                funev[i] <- this.one$funeval
                numbest[i] <- sum(this.one$popula[, 1])
        }
        list(funevals = funev, numbest = numbest, call
                 = match.call())
}
```

The results look reasonable using the defaults. The first thing to look at is the number of elements of the best solution that were `TRUE` in each of the trials. 77 of the 100 evaluations found the correct answer, and only one time did the best solution contain more than one `FALSE`.

```
> table(gentest1.res1$numbest)
 18 19 20
  1 22 77
> summary(gentest1.res1$funeval)
 Min. 1st Qu. Median Mean 3rd Qu. Max.
  306     321    330  330     336  351
```

When jittering is used, the number of function evaluations is random. The call to `summary` indicates the distribution of the number of function evaluations used.

This performance is much better than that reported by Jennison and Sheehan with the traditional genetic algorithm that they used. The performance of `fjjgenop1` seems adequate to me.

Since the problem is so simple, I hypothesized that the jittering might be slowing down the algorithm. To test this, I set the number of jitters to zero and the number of births to 400.

```
> table(gentest1.res0.4$numbest)
 17 18 19 20
  3  4 31 62
> summary(gentest1.res0.4$funeval)
 Min. 1st Qu. Median Mean 3rd Qu. Max.
  420     420    420  420     420  420
```

The result is worse while using more function evaluations. So jittering is useful even in this artificially simple problem.

The `genopt` function, listed below, is a more useful function that minimizes a function via a genetic algorithm. First off, it presumes that the parameters are real-valued rather than binary—a much more common case. Box constraints are allowed on the parameters; this is easy to do, and decidedly useful enough to be worth the effort.

In `fjjgenop1` the argument names were all in upper case to avoid conflicts with additional arguments for the function being optimized. With so many arguments capitalized, the solution becomes almost as ugly as the problem. The `genopt` function has the argument `add.args` that is expected to be a list of the additional arguments. This avoids the possibility of name conflicts altogether, and frees up the three-dots for a different task.

The `population` argument to `genopt` can be a matrix whose columns are each a parameter vector in the initial population, or it can be the result of a previous call to `genopt`. In the latter case it is of importance that the function being optimized is not re-evaluated for the population. Thus, `genopt` can be called repeatedly with the `population` equal to the result of the last call, so the effect is like asking for a large number of births but getting intermediate results.

Another addition in `genopt` is a purely random phase. Parameter vectors are created by some random mechanism that does not care about the parameters within the population. A parameter vector that has a better objective than the worst member of the population replaces that member. The improvement in the population resulting from the random phase can greatly improve the efficacy of the genetic phase. The creation of the random vectors in `genopt` is quite *ad hoc*—a better method should be used for specific applications.

`genopt` saves the random seed that it started with, so that the computations can be reproduced, if desired.

Without further ado, here is the definition of `genopt`.

```
"genopt"<-
function(fun, population, lower =  - Inf, upper = Inf, scale =
        dcontrol["eps"], add.args = NULL, control = genopt.control(
        ...), ...)
{
        random.seed <- .Random.seed
        if(is.character(fun))
                fun <- get(fun, mode = "function")
        fun.args <- c(list(NULL), add.args)
        go.rectify <- function(pars, lower, upper)
        {
                pars[pars < lower] <- lower[pars < lower]
                pars[pars > upper] <- upper[pars > upper]
                pars
        }
        if(is.list(population)) {
                objective <- population$objective
                funevals <- population$funevals
```

```
        population <- population$population
        popsize <- ncol(population)
        if(is.null(popsize) || length(objective) != popsize)
                stop("bad input population")
        if(!is.numeric(funevals) || is.na(funevals)) {
                funevals <- 0
                warning("funevals starting at 0")
        }
}
else {
        if(!is.matrix(population))
                stop("bad input population")
        popsize <- ncol(population)
        objective <- numeric(popsize)
        npar <- nrow(population)
        lower <- rep(lower, length = npar)
        upper <- rep(upper, length = npar)
        if(any(upper < lower))
                stop("upper element smaller than lower")
        for(i in 1:popsize) {
                population[, i] <- fun.args[[1]] <-
                        go.rectify(population[, i], lower,
                        upper)
                objective[i] <- do.call("fun", fun.args)
        }
        funevals <- popsize
}
icontrol <- control$icontrol
dcontrol <- control$dcontrol
trace <- icontrol["trace"]
minobj <- min(objective)
npar <- nrow(population)
if(trace) {
        cat("objectives go from", format(minobj), "to",
                format(max(objective)), "\n")
}
if(icontrol["random.n"]) {
        par.range <- apply(population, 1, range)
        par.range[2, par.range[2,  ] == par.range[1,  ]] <-
                par.range[2, par.range[2,  ] == par.range[1,
                ]] + dcontrol["scale.min"]
        maxobj <- max(objective)
        for(i in 1:icontrol["random.n"]) {
                fun.args[[1]] <- runif(npar, par.range[1,  ],
                        par.range[2,  ])
                this.obj <- do.call("fun", fun.args)
                if(this.obj < maxobj) {
                        maxind <- order(objective)[popsize]
                        population[, maxind] <- fun.args[[1]]
                        objective[maxind] <- this.obj
```

```
                        maxobj <- max(objective)
                }
        }
        if(trace) {
                cat("objectives go from", format(minobj),
                        "to", format(maxobj), "\n")
        }
}
njit <- icontrol["jitters.n"]
lower <- rep(lower, length = npar)
upper <- rep(upper, length = npar)
if(any(upper < lower))
        stop("upper element smaller than lower")
scale[scale < dcontrol["scale.min"]] <- dcontrol["scale.min"]
scale <- rep(scale, length = npar)
prob <- dcontrol["prob"]
prob <- c(prob, 1 - prob)
maxeval <- icontrol["maxeval"]
for(i in 1:icontrol["births"]) {
        if(funevals >= maxeval)
                break
        parents <- sample(popsize, 2)
        child <- population[, parents[1]]
        cloc <- sample(c(T, F), npar, rep = T, prob = prob)
        if(all(cloc))
                cloc[sample(npar, 1)] <- F
        else if(all(!cloc))
                cloc[sample(npar, 1)] <- T
        child[cloc] <- population[cloc, parents[2]]
        fun.args[[1]] <- child
        child.obj <- do.call("fun", fun.args)
        funevals <- funevals + 1
        parent.obj <- objective[parents]
        survive <- child.obj < max(parent.obj)
        if(trace) {
                cat(i, "parents:", parent.obj, "child:",
                        format(child.obj), if(survive)
                          "(improve)", "\n")
        }
        if(survive || (child.obj == parent.obj[1] &&
                child.obj == parent.obj[2])) {
                if(parent.obj[1] > parent.obj[2])
                        out <- parents[1]
                else out <- parents[2]
                population[, out] <- child
                objective[out] <- child.obj
                if(trace && child.obj < minobj) {
                        minobj <- child.obj
                        cat("new minimum\n")
                }
```

```
                        for(i in seq(length = njit)) {
                                fun.args[[1]] <- jchild <- go.rectify(
                                  rnorm(npar, child, scale), lower,
                                  upper)
                                jchild.obj <- do.call("fun", fun.args
                                  )
                                funevals <- funevals + 1
                                if(jchild.obj < child.obj) {
                                  child <- population[, out] <-
                                    jchild
                                  child.obj <- objective[out] <-
                                    jchild.obj
                                  if(trace) {
                                    cat("jitter successsful:", format(
                                      jchild.obj), "\n")
                                    if(jchild.obj < minobj) {
                                      cat("new minimum\n")
                                      minobj <- jchild.obj
                                    }
                                  }
                                }
                        }
                }
        }
        ord <- order(objective)
        list(population = population[, ord], objective = objective[
                ord], funevals = funevals, random.seed = random.seed,
                call = match.call())
}
```

It would be much nicer if each birth and the jittering were abstracted into functions. The difficulty with this is how information is to be passed in and out of the functions.

An important feature of `genopt` is that it uses the "control" paradigm that was introduced into S with some of the statistical modeling functions (Chambers and Hastie, 1992). There is a `control` argument that expects an object containing a number of items that control the computation. This list can be created with a separate function—in this case called `genopt.control`. Often, as in `genopt`, the three-dots of the main function allows individual control arguments to be given. This is a convenient and powerful convention.

```
"genopt.control"<-
function(births = 100, random.n = 0, jitters.n = 3,
        trace = T, eps = 0.1, prob = 0.3, scale.min =
        1e-12, maxeval = Inf)
{
        dcon <- c(eps = eps, prob = prob, scale.min =
                scale.min)
```

```
        icon <- c(births = births, random.n = random.n,
                jitters.n = jitters.n, trace = trace,
                maxeval = maxeval)
        list(icontrol = icon, dcontrol = dcon)
}
```

For use within S, the division between integer and floating-point numbers is unnecessary. However, in C it matters greatly. This function would not need to change if the optimization were put into C.

The `genopt` function can be used as a model for a function that specializes the optimization. Such a function could use `genopt.control` if no new control arguments were necessary (unused ones pose no problem). A new control function would be required if new controls were used.

The Jennison and Sheehan problem can be made a little harder by letting the parameters be real-valued between 0 and 1, instead of only 0 or 1. Here's an approach to solving this problem with `genopt`. The first command indicates what the true minimum is.

```
> -sum(jsa20)
[1] -35.78
> jj1 <- genopt(fjj.jsa20, matrix(runif(400),20), lower=0,
+         upper=1, trace=F)
> summary(jj1$objective)
   Min. 1st Qu. Median   Mean 3rd Qu.   Max.
 -31.45  -30.06 -29.32 -29.09  -28.49 -25.32
> jj2 <- genopt(fjj.jsa20, jj1, lower=0, upper=1, trace=F)
> summary(jj2$objective)
   Min. 1st Qu. Median   Mean 3rd Qu.   Max.
 -33.72     -33 -32.44 -32.49  -32.08 -31.16
> summary(jj2$population[,1])
   Min. 1st Qu. Median   Mean 3rd Qu. Max.
 0.7201  0.8724 0.9713 0.9248       1    1
> jj2$funeval
[1] 646
```

This is obviously harder than the binary problem, but we're not doing so terrible with 650 function evaluations.

The problem can be made more realistic by adding a constraint to the problem. Suppose that we still want to minimize `fjj.jsa20` but there is the constraint that the sum of the parameters can be no more than 10. This new problem is known as a knap-sack problem; there is an S function called `napsack` for one class of knap-sack problems. An inexact but often effective way of dealing with constraints is to add a penalty to the objective if constraints are violated.

Here is a new function to be optimized that includes a penalty for the constraint.

```
> fjj.jsa20sum
function(x, limit = 10, cost = 2)
{
        objective <-  - sum(x * jsa20)
        violation <- max(0, sum(x) - limit)
        objective + violation * cost
}
```

Naively you may think that the cost parameter should be infinite. An infinite cost causes a discontinuity, so the cost obviously can't be infinite when using this trick with derivative-based optimizers. Genetic algorithms don't fail on account of discontinuity, but you still don't want the cost to be excessively high because you want to guide the algorithm toward the right place.

Here we see what the true optimum is, and then evaluate how well we are doing with the constraint and the objective.

```
> -sum(sort(jsa20)[11:20])
[1] -24.14
> jj3 <- genopt(fjj.jsa20sum, matrix(runif(400),20),
+         lower=0, upper=1, trace=F, random=100)
> apply(jj3$population, 2, sum)
 [1] 10.344031 10.250866 10.020323 10.024000  9.676853
 [6] 10.319002 10.850953 10.436489 10.074452 11.640020
[11] 10.594859 10.476254 10.747117  9.692068 11.048758
[16] 10.157180 10.057599 11.963033 10.322911 10.842347
> summary(jj3$objective)
  Min. 1st Qu. Median   Mean 3rd Qu.   Max.
 -22.1  -21.65 -21.23 -21.24  -20.95 -19.59
```

At this point the constraint isn't obeyed very well, but not egregiously violated, and the objectives aren't so bad. The default cost is almost surely too small, so we give a different value of `cost` to `fjj.jsa20sum`.

```
> jj4 <- genopt(fjj.jsa20sum, jj3, lower=0, upper=1,
+         add.arg=list(cost=50), trace=F)
> apply(jj4$population, 2, sum)
 [1]  9.934612  9.639238  9.803568  9.894391  9.854145
 [6] 10.344031 10.250866 10.020323  9.640379  9.880965
[11]  9.858581  9.731130 10.074452  9.991200 11.640020
[16]  9.979223 10.594859 10.476254  9.819179 10.057599
> summary(jj4$objective)
   Min. 1st Qu. Median   Mean 3rd Qu.   Max.
 -22.32  -22.11 -21.62 -21.65  -21.31 -20.84
```

```
> jj4$funeval
[1] 442
```

Often a much better solution can be found by inserting into a population the best solution from each of two or more independent runs of the genetic algorithm. This is the strategy with `jj4` and `jj5` that arrives at `jj6`.

```
> jj5 <- genopt(fjj.jsa20sum, matrix(runif(400),20),
+         lower=0, upper=1, trace=F, random=100,
+         add.arg=list(cost=50))
> apply(jj5$population, 2, sum)
 [1] 9.975258 9.975258 9.871977 9.756353 9.206575
 [6] 9.287558 9.808857 9.701156 8.964027 9.923369
[11] 9.538543 9.761804 9.400245 9.392638 9.536091
[16] 9.811276 9.800468 9.985085 9.809348 9.582101
> summary(jj5$objective)
   Min. 1st Qu. Median   Mean 3rd Qu.   Max.
 -21.72  -20.51 -19.78 -19.97  -19.24 -18.59
> jjp1 <- cbind(jj4$pop[,1], jj5$pop[,1], matrix(
+         runif(400),20))
> jj6 <- genopt(fjj.jsa20sum, jjp1, lower=0, upper=1,
+         trace=F, random=100, birth=500,
+         add.arg=list(cost=50))
> apply(jj6$population, 2, sum)
 [1] 9.899633 9.890572 9.932042 9.881737 9.919993
 [6] 9.969054 9.934612 9.785121 9.970647 9.679190
[11] 9.848886 9.852910 9.947428 9.570757 9.961821
[16] 9.961821 9.972682 9.931718 9.876357 9.995824
[21] 9.979015 9.966458
> summary(jj6$objective)
   Min. 1st Qu. Median   Mean 3rd Qu.   Max.
 -22.82  -22.33 -22.16 -22.16  -21.92 -21.46
```

For many applications, S is the wrong place for a genetic algorithm. To be effective a genetic algorithm requires a great number of function evaluations. If the function being optimized does not involve heavy computing, then the optimization should be in C or another compiled language. But if each evaluation will take a long time anyway, then the overhead of the genetic algorithm in S will be negligible.

The one problem that a genetic algorithm can have is that it doesn't go to the optimum. One way this occurs is that the population initially moves well but too soon everyone in the population is very similar—premature convergence. Another popular way is for the population not to move much at all. As the population size goes from tiny to huge, you will go from the first of these problems to the second. So choosing the population size is important—the proper choice depends on the particular problem and on the use to which the result is put.

On one extreme, I've chosen a population of size 10 in an application. A genetic algorithm is used to get a starting point for a derivative-based optimizer because there are local minima, and because the number of parameters can be very large, and each function evaluation is reasonably expensive. In cases where the result needs to be close to the true optimum and the cost of function evaluations is insignificant in comparison, then a population size of hundreds is probably not too many. In run-of-the-mill situations, I tend to use population sizes of 20 to 50, but I have no basis in fact for doing so.

The main thing to get right in a genetic algorithm is the genetics. If what you are using in the population to represent each solution does not map well into the objective to be optimized, then the algorithm will be ineffective. One good reason to use S for genetic algorithms is because S makes it easy to use whatever is best at representing the solutions. The development of genetic algorithms is limited primarily by our imaginations.

14.4 Optimization via PORT

The PORT library is some of the best optimization software that is freely available. This comes out of Bell Labs and was written by David Gay and associates. Both of the S-PLUS functions `nlminb` and `nlmin` are based on PORT routines.

A drawback of these two functions is that they are slow because they use S functions. The `portopt1` and `portoptgen` functions discussed below provide a way of optimizing a function written in C or Fortran but getting the results in S.

Simple interface

The `portopt1` function allows you to minimize a C or Fortran function that has just two arguments, the vector of parameters over which the optimization occurs and the length of the vector. This is not terribly useful functionality, but it gives us a foothold on the problem.

```
"portopt1"<-
function(fun.address, start, lower =  - Inf, upper = Inf, scale = 1,
        control = portopt.control(...), ...)
{
        if(fun.address == 0)
                stop("symbol not loaded")
        p <- length(start)
        if(length(scale) != 1 && length(scale) != p)
                stop("bad length for 'scale' argument")
        if(any(scale) <= 0)
                stop("nonpositive scale vector")
        scale <- rep(scale, length = p)
```

```
if((length(lower) != 1 && length(lower) != p) || (length(
        upper) != 1 && length(upper) != p))
        stop("bad length for lower or upper")
big <- 1e+30
lower[lower <  - big] <-  - big
upper[upper > big] <- big
lower <- rep(lower, length = p)
upper <- rep(upper, length = p)
if(any(lower > upper))
        stop("box constraints are infeasible")
liv <- 59 + p
lv <- 77 + (p * (p + 23))/2
init <- .Fortran("divset",
        as.integer(2),
        iv = integer(liv),
        as.integer(liv),
        as.integer(lv),
        v = double(lv))[c("iv", "v")]
if(init$iv[1] != 12)
        stop("abnormal return from divset")
icontrol <- control$icontrol
init$iv[17:18] <- icontrol[c("eval.max", "iter.max")]
dnam <- c("abs.tol", "rel.tol", "x.tol", "step.min",
        "step.max", "sing.step", "sing.tol", "scale.tol",
        "scale.mod", "scale.fac", "diff.grad")
dcvec <- control$dcontrol[dnam]
dmiss <- is.na(dcvec)
init$v[c(31:37, 39:42)[!dmiss]] <- dcvec[!dmiss]
names(upper) <- names(start)
limits <- rbind(lower = lower, upper = upper)
storage.mode(limits) <- "double"
ans <- .C("portopt_one_Sp",
        as.integer(liv),
        as.integer(lv),
        iv = as.integer(init$iv),
        v = as.double(init$v),
        as.integer(icontrol),
        objective = double(1),
        npar = as.integer(p),
        limits = limits,
        scale = as.double(scale),
        as.integer(fun.address),
        parameters = as.double(start))[c("parameters",
        "objective", "iv", "limits")]
msg <- switch(ans$iv[1] - 2,
        "X convergence",
        "relative function convergence",
        "both X and relative function convergence",
        "absolute function convergence",
        "singular convergence",
```

```
                "false convergence",
                "function evaluation limit reached",
                "iteration limit reached",
                stop("invalid output code 11"),
                stop("invalid output code 12"),
                stop("invalid output code 13"),
                stop("LIV is too small"),
                stop("LV is too small"),
                stop("invalid output code 17"),
                stop("negative component in scale vector"),
                stop("cannot complete optimization"))
        ans$convergence <- msg
        ans$call <- match.call()
        ans$iv <- NULL
        ans
}
```

This is really a simple function despite its length. It starts out checking for misspecification of arguments and getting those arguments ready to be used in the optimization. Then it calls a Fortran function that initializes the two special arrays for the optimizer, and then changes control arguments as needed. Then the optimizer is invoked through the call to C, and finally the result is fixed up for the return.

The novel part is the C code that arranges for the optimization to take place.

```
#include <S.h>

void
portopt_one_Sp(liv, lv, iv, v, icontrol, objective, npar, limits,
        scale, f_addr, par)
long *liv, *lv, *iv, *icontrol, *npar;
double *v, *objective, *limits, *scale, *par;
double (**f_addr)();
{
        long i, iter_max = iv[17], trace = icontrol[2], iternum=-1;

        if(iter_max) {
                F77_CALL(mnb0)(npar, par, scale, limits,
                        objective, liv, iv, lv, v);
                *objective = (**f_addr)(par, npar);
                if(is_na(objective, DOUBLE) ||
                                is_inf(objective, DOUBLE)) {
                        iv[1] = 1;
                }
                while(iv[0] < 3 && iternum < iter_max) {
                        if(iv[30] != iternum) {
                                iternum = iv[30];
                                if(trace) {
```

```
                                printf("iter: %d  objective: %g\n",
                                        iternum, *objective);
                                fflush(stdout);
                        }
                }
                F77_CALL(mnb0)(npar, par, scale, limits,
                        objective, liv, iv, lv, v);
                *objective = (**f_addr)(par, npar);
                if(is_na(objective, DOUBLE) ||
                                is_inf(objective, DOUBLE)) {
                        iv[1] = 1;
                }
        }
    }
    *objective = (**f_addr)(par, npar);
}
```

The first thing you may notice is that the code is complicated by a test to see if the maximum number of iterations is zero. I've found the extra bother to be well worth it—there are a surprising number of instances in which doing nothing is desired. The operative part of this function is a `while` loop that does four things. It updates the iteration number if necessary and prints the iteration information if asked to. It then calls the PORT optimizer, the Fortran function `mnb0`. Next it calls the function to be optimized with the parameter vector that the optimizer set. Finally it notifies the optimizer if trouble is afoot.

Perhaps you are puzzled by the declaration:

```
double (**f_addr)();
```

The number that is contained in the S vector is the address of the function, that is, it is a pointer to the function. Whatever S passes into C is a pointer to that information, so we have a pointer to a pointer to the function. S always thinks in terms of vectors—if there were addresses for a number of functions contained in the S vector passed into C, the declaration would remain the same.

`portopt1` follows the convention of a `control` argument that is supplied by a "something.control" function. The `portopt.control` function is a little different in that it uses missing values for some of its defaults because the real defaults are created elsewhere.

```
"portopt.control"<-
function(eval.max = 5000, iter.max = 150, abs.tol = NA,
        rel.tol = NA, x.tol = NA, step.min = NA,
        step.max = NA, sing.step = NA, sing.tol = NA,
        scale.tol = NA, scale.mod = NA, scale.fac = NA,
        diff.grad = NA, trace = T)
{
        icontrol <- c(npar = NA, eval.max = eval.max,
```

```
                iter.max = iter.max, trace = trace)
        dcontrol <- c(abs.tol = abs.tol, rel.tol =
                rel.tol, x.tol = x.tol, step.min =
                step.min, step.max = step.max,
                sing.step = sing.step, sing.tol =
                sing.tol, scale.tol = scale.tol,
                scale.mod = scale.mod, scale.fac =
                scale.fac, diff.grad = diff.grad)
        list(icontrol = icontrol, dcontrol = dcontrol)
}
```

The `symbol.address` function is a slight modification of the `is.loaded` S-PLUS function. It returns the addresses of a vector of symbols—zero means that the symbol is not currently loaded (or you didn't get the symbol right).

```
symbol.address
function(symbol)
{
        symbol <- as.character(symbol)
        .C("S_get_entries",
                symbol,
                integer(length(symbol)),
                length(symbol))[[2]]
}
```

As a test, I use a computer intensive way of calculating two means. At this stage of testing, we really do want to know the right answer beforehand.

```
double
jjtest1(pars, n)
long *n;
double *pars;
{
        long i;
        double dif, ans=0.0;
        static double data[6] = { 3.4, 6.3, 9.2, -.7, 2.1, 8.8};

        for(i=0; i < 3; i++) {
                dif = data[i] - pars[0];
                ans = ans + dif * dif;
        }
        for(i=3; i < 6; i++) {
                dif = data[i] - pars[1];
                ans = ans + dif * dif;
        }
        return(ans);
}
```

Now compile the code, dyn load it and use it in the optimization function.

```
> !Splus COMPILE jjtest1.c
targets= jjtest1.o
make -f /usr/lang/sp/newfun/lib/S_makefile  jjtest1.o
cc -c -I${SHOME-`Splus SHOME`}/include -O2   jjtest1.c
> dyn.load("jjtest1.o")
> portopt1(symbol.address(symbol.C("jjtest1")), c(0,0))
iter: 0  objective: 218.23
iter: 1  objective: 178.277
iter: 2  objective: 64.48
iter: 3  objective: 64.48
$parameters:
[1] 6.300002 3.400020

$objective:
[1] 64.48

$limits:
        [,1]   [,2]
lower -1e+30 -1e+30
upper  1e+30  1e+30

$convergence:
[1] "relative function convergence"

$call:
portopt1(fun.address = symbol.address(symbol.C(
        "jjtest1")), start = c(0, 0))

> 10.2/3
[1] 3.4
> 18.9/3
[1] 6.3
```

The answers are correct, and the algorithm converged okay.

If you were paying attention, then you have already chastised me for passing a pointer for the parameter length in the calls to the function to be optimized in `portopt_one_Sp`. It is true that this is a little unnatural in C, but it allows Fortran to be used as well as C. Here is the test function translated into Fortran.

```
function jjtest2(pars, n)
implicit none
integer n
double precision pars(n)
integer i
double precision dif, jjtest2, tdat(6)
tdat(1) = 3.4
tdat(2) = 6.3
```

```
      tdat(3) = 9.2
      tdat(4) = -.7
      tdat(5) = 2.1
      tdat(6) = 8.8

      jjtest2 = 0.0

      do 10 i=1, 3
              dif = tdat(i) - pars(1)
              jjtest2 = jjtest2 + dif * dif
10    continue
      do 20 i=4, 6
              dif = tdat(i) - pars(2)
              jjtest2 = jjtest2 + dif * dif
20    continue
      return
      end
```

It is used like:

```
> portopt1(symbol.address(symbol.For("jjtest2")), c(0,0))
```

I foresee `portopt1`—if it is useful at all—not being used much directly, but put into a function that insures that the code computing a specific function is loaded and then calls `portopt1`.

General interface

The `portoptgen` function provides a way to build an S function that minimizes a function in a compiled language that has several arguments. For example, a C function that not only has a vector of parameters over which to be optimized, but data inputs also. `portoptgen` creates a file of C code containing a function to be called by the `.C` function, and returns an S function that makes that call. Let's start with an example, and go through how it works later.

The C function `jjtest3` is the function that we want to minimize. This, like the code in the last subsection, is a computationally intensive way to find the means of numbers in some groups. However, in this case it is much more flexible in what it can do.

```
> !cat jjtest3.c
#include <S.h>

double
jjtest3(pars, n, lengs, data)
long n, *lengs;
double *pars, *data;
{
```

```
        long i, j, count=0, top_count=0;
        double dif, ans=0.0;

        for(i=0; i < n; i++) {
                top_count += lengs[i];
                for(j=count; j < top_count; j++) {
                        dif = data[j] - pars[i];
                        ans = ans + dif * dif;
                }
                count = top_count;
        }
        return(ans);
}
```

This function has arguments that are the array of parameters over which the optimization takes place (they will be the means), the number of parameters, an array giving the number of points in each group, and an array containing all of the data. Now that we know what the function looks like, we are ready to use `portoptgen`.

```
> fjjtest3 <- portoptgen("jjtest3", c(v.2="double", n="long",
       lengs="long", data = "double"), nonpo="n", make="n")
Creating file port_opt_jjtest3.c
```

The first argument to `portoptgen` is the name of the function to minimize. The second argument is a character vector containing the declarations for the arguments to the function—these can either be C declarations ("long", for example), or S storage modes ("integer"). The names of this vector are used by S and C as variable names. This call also uses two of the optional arguments.

Since `n` in `jjtest3` is not a pointer, we indicate that in the `nonpointers` argument to `portoptgen`. All of the variables are presumed to be passed by address into the C function unless noted in `nonpointers`. In general, it is not a good idea to force users into a double negative as in `nonpointer=F` means "pointer". It makes it more confusing, and hence more likely that bugs will occur. But in this case the ease of use overrules this—in my mind, at least. An improvement would be to find a better name in place of "nonpointer".

The `n` variable is also in the `make` argument to `portoptgen`. This is merely the length of the vector of parameters, so we don't want to have this as an argument to the S function—S can assign it within the function. Arguments that are in `make` are not arguments to the S function that is created (hence they need to be made inside the function).

In the call to `portoptgen` above, two things happened: the S function was put into `fjjtest3`, and a file of C code was created. Here is the top of the file of C code.

```
> !head -20 port_opt_jjtest3.c
```

```
#include <S.h>
void
port_opt_jjtest3_Sp(liv, lv, iv, v, icontrol, objective, limits, scale,
v_2, n, lengs, data
)
long *liv, *lv, *iv, *icontrol;
double *v, *objective, *limits, *scale;
double *v_2;
long *n;
long *lengs;
double *data;
{
        long i, iter_max = iv[17], trace = icontrol[3], iternum=-1;
        long *npar = icontrol;
        extern double jjtest3();
        if(iter_max) {
                F77_CALL(mnb0)(npar, v_2, scale, limits,
                        objective, liv, iv, lv, v);
                *objective = jjtest3(v_2, *n, lengs, data);
                if(is_na(objective, DOUBLE) ||
```

It is a function that is called by `fjjtest3` through `.C`. The arguments to `jjtest3` are passed in along with the other arguments needed for the optimizer, and the call to `jjtest3` is done correctly. There is nothing that needs to be done to this except compiling it. (Note that `v.2` is translated to `v_2` in C.)

The S function that `portoptgen` returns does usually need a little work before it is usable. Comments at the top indicate what needs to be done.

```
> fjjtest3
function(v.2 = NULL, lengs = NULL, data = NULL, lower
         =  - Inf, upper = Inf, scale = 1, control =
        portopt.control(...), ...)
{
# need to ensure that jjtest3 is loaded, possibly with
# if(!is.loaded(symbol.C('jjtest3')))
#     dyn.load('jjtest3.o')
#
# need to ensure that port_opt_jjtest3_Sp is loaded, possibly with
# if(!is.loaded(symbol.C('port_opt_jjtest3_Sp')))
#     dyn.load('port_opt_jjtest3.o')
#
# need to fix up n
        p <- length(v.2)
        if(length(scale) != 1 && length(scale) != p)
                stop("bad length for 'scale' argument")
  . . .
```

(There is lots more of the function.) In a lot of situations, you can probably just

uncomment the lines involving `dyn.load` to take care of the issue of loading. Any variable that appears in the `make` argument to `portoptgen` will need to be taken care of. The function as returned by `portoptgen` will create an error when any of these is attempted to be used (in the `.C` call). It would usually be good form to make the arguments to the S function that are also arguments to `cfun` be required arguments—that is, remove the `NULL` default value.

Once the S function editing is done and the C code compiled, we can compute our means.

```
> fjjtest3(c(1, 3), c(3, 5), 1:8)
iter: 0  objective: 60
iter: 1  objective: 34.329
iter: 2  objective: 24.1624
iter: 3  objective: 12
iter: 4  objective: 12
iter: 5  objective: 12
$parameters:
[1] 2.000000 5.999999

$objective:
[1] 12

$limits:
        [,1]   [,2]
lower -1e+30 -1e+30
upper  1e+30  1e+30

$convergence:
[1] "relative function convergence"

$call:
fjjtest3(v.2 = c(1, 3), lengs = c(3, 5), data = 1:8)
```

Here are all of the arguments to `portoptgen`.

```
> args(portoptgen)
function(cfun, args, make = F, nonpointers = F,
        parameter = 1, fortran = F)
NULL
```

There are two that we haven't discussed yet. The `parameter` argument indicates which argument to `cfun` is the one over which optimization is to occur. As you might guess, the `fortran` argument is a logical that tells whether `cfun` is Fortran or C.

DANGER. The return value of the `cfun` function that `portoptgen` uses must be double precision.

Now, how it all works. The first ingredient is an object that contains the template for the C code. Here are the first few elements of it.

```
> portoptgen.ctemplate[1:16]
 [1] "#include <S.h>"
 [2] "void"
 [3] "port_opt_cFsUF_Sp(liv, lv, iv, v, icontrol, obje
ctive, limits, scale, "
 [4] "cFaRGS"
 [5] ")"
 [6] "long *liv, *lv, *iv, *icontrol;"
 [7] "double *v, *objective, *limits, *scale;"
 [8] "cFdEC"
 [9] "{"
[10] "\tlong i, iter_max = iv[17], trace = icontrol[3],
iternum=-1;"
[11] "\tlong *npar = icontrol;"
[12] "\textern double cFfUNDEC();"
[13] "\tif(iter_max) {"
[14] "\t\tF77_CALL(mnb0)(npar, cFpAR, scale, limits,"
[15] "\t\t\tobjective, liv, iv, lv, v);"
[16] "\t\t*objective = cFcALL"
```

This should be familar from the C code shown above except that there are funny bits like "cFsUF". These will be replaced with text that depends on the function that is being optimized.

The `portoptgen.ctemplate` object was created by typing in a file, then—when it was correct—reading it into S with the command:

```
portoptgen.ctemplate <- scan("portopt_gen.c",
       what = "", sep = "\n")
```

The only trick is that you want to have a double backslash in the file anywhere that you would have a backslash in C.

Another ingredient is `portoptgen.stemplate`. This is the model on which the return value of `portoptgen` is built. It is quite similar to the `portopt1` function.

Here is the definition of `portoptgen`.

```
> portoptgen
function(cfun, args, make = F, nonpointers = F, parameter = 1, fortran = F)
{
        if(!is.character(cfun) || length(cfun) > 1)
```

```
        stop("cfun must be a single string")
if(fortran)
        nonpointers <- F
bnam <- paste("v", 1:length(args), sep = "")
if(!length(anam <- names(args)))
        names(args) <- bnam
else {
        nonam <- nchar(anam) == 0
        names(args)[nonam] <- bnam[nonam]
}
if(any(is.na(match(args, c("double", "single", "float", "integer",
        "long", "character", "char", "complex"), nomatch = NA))))
        stop("bad type in 'args' argument")
cargs <- args
cam <- match(cargs, c("single", "integer", "character"), nomatch = 0)
if(any(cam)) {
        cargs[cam == 1] <- "float"
        cargs[cam == 2] <- "long"
        cargs[cam == 3] <- "char"
}
names(cargs) <- transcribe(names(cargs), ".", "_")
if(any(duplicated(names(cargs))))
        stop("duplicated names in arguments")
c.outnames <- c("lv", "liv", "v", "iv", "icontrol", "objective",
        "scale", "limits")
if(sum(match(names(cargs), c.outnames, nomatch = 0)))
        stop(paste("Can't have C argument names in:", paste(c.outnames,
                collapse = ", ")))
sargs <- cargs
sam <- match(sargs, c("float", "long", "char"), nomatch = 0)
sargs[sam == 1] <- "single"
sargs[sam == 2] <- "integer"
sargs[sam == 3] <- "character"
names(sargs) <- transcribe(names(sargs), "_", ".")
arglen <- length(args)
switch(mode(make),
        logical = {
                make <- rep(make, length = arglen)
        }
        ,
        character = {
                maknam <- make
                make <- rep(F, arglen)
                names(make) <- names(args)
                make[maknam] <- T
                if(length(make) != arglen)
                        stop("bad input for make")
        }
        ,
        stop("invalid input for make"))
```

```
        names(make) <- names(sargs)
        if(is.numeric(parameter))
                parameter <- names(sargs)[parameter]
        else if(!match(parameter, names(sargs), nomatch = 0)) {
                par.m <- match(parameter, names(args), nomatch = NA)
                if(is.na(par.m))
                        stop("bad value for 'parameter' argument")
                parameter <- names(sargs)[par.m]
        }
        if(sargs[parameter] != "double")
                stop("parameters need to be double precision")
        if(make[parameter])
                stop("parameter must be passed in, not made")
        switch(mode(nonpointers),
                logical = {
                        nonpointers <- rep(nonpointers, length = arglen)
                }
                ,
                character = {
                        pointnam <- nonpointers
                        nonpointers <- rep(F, arglen)
                        names(nonpointers) <- names(args)
                        nonpointers[pointnam] <- T
                        if(length(nonpointers) != arglen)
                                stop("bad input for nonpointers")
                }
                ,
                stop("invalid input for nonpointers"))  #
#
# create and modify file of C code
#
        cfile.name <- paste("port_opt_", cfun, ".c", sep = "")
        cfile.test <- filetest(cfile.name, dir = T, wr = T, r = T)
        if(cfile.test["exists"]) {
                if(cfile.test["dir"])
                        stop(paste(cfile.name, "is a directory"))
                if(!cfile.test["wr"])
                        stop(paste(cfile.name, "is unwriteable"))
                if(!cfile.test["read"])
                        stop(paste(cfile.name, "is unreadable"))
                cat("Overwriting file", cfile.name, "\n")
        }
        else {
                cat("Creating file", cfile.name, "\n")
        }
        cfun.args <- paste(names(cargs), collapse = ", ")
        cfun.dec <- paste(cargs, " *", names(cargs), ";", sep = "", collapse =
                "\n")
        if(fortran)
                cfun.cn <- paste("F77_CALL(", cfun, ")", sep = "")
```

```
        else cfun.cn <- cfun
        cfun.call <- paste(cfun.cn, "(", paste(ifelse(nonpointers, "*", ""),
                names(cargs), sep = "", collapse = ", "), ");", sep = "")
        cfun.par <- names(cargs)[match(parameter, names(sargs))]
        cat(portoptgen.ctemplate, file = cfile.name, sep = "\n")
        substifile(cfile.name, "cFaRGS", cfun.args)
        substifile(cfile.name, "cFdEC", cfun.dec)
        substifile(cfile.name, "cFcALL", cfun.call)
        substifile(cfile.name, "cFpAR", cfun.par)
        substifile(cfile.name, "cFfUNDEC", cfun.cn)
        substifile(cfile.name, "cFsUF", cfun)   #
#
# get S function right
#
        new.sargs <- names(sargs)[!make]
        ans <- portoptgen.stemplate
        ans.len <- length(ans)
        ans[[c(ans.len, 1, 2, 2)]] <- as.name(parameter)
        the.comm <- c(paste("# need to ensure that", cfun,
                "is loaded, possibly with"), paste("# if(!is.loaded(symbol.",
                if(fortran) "For" else "C", "('", cfun, "')))", sep = ""),
                paste("#     dyn.load('", cfun, ".o')", sep = ""), "# ", paste(
                "# need to ensure that port_opt_", cfun,
                "_Sp is loaded, possibly with", sep = ""), paste(
                "# if(!is.loaded(symbol.C('port_opt_", cfun, "_Sp')))", sep =
                ""), paste("#     dyn.load('", substring(cfile.name, 1, nchar(
                cfile.name) - 1), "o')", sep = ""))
        if(any(make)) {
                the.comm <- c(the.comm, "# ", paste("# need to fix up", names(
                        make)[make]))
        }
        the.comm <- eval(parse(text = c("function() {", the.comm, "}")))
        ans[[ans.len]] <- c(the.comm[[1]], ans[[ans.len]])
        new.fun <- vector("function", length(new.sargs) + 1)
        names(new.fun) <- c(new.sargs, "")
        ans <- c(new.fun[ - length(new.fun)], ans)
        the.dotC <- expression(ans <- .C("portopt_one_Sp",
                as.integer(liv),
                as.integer(lv),
                iv = as.integer(init$iv),
                v = as.double(init$v),
                as.integer(icontrol),
                objective = double(1),
                limits = limits,
                scale = as.double(scale))[c("parameters", "objective", "iv",
                "limits")])
        new.dotC <- as.call(parse(text = paste("foo(", paste(names(sargs), "=",
                ifelse(make, paste("stop('fix up", names(sargs), "')"), paste(
                "as.", sargs, "(", names(sargs), ")", sep = "")), collapse =
                ", "), ")")))
```

```
        names(new.dotC[[1]])[match(parameter, names(new.dotC[[1]]))] <-
                "parameters"
        the.dotC[[c(1, 2, 2)]] <- c(the.dotC[[c(1, 2, 2)]], new.dotC[[1]][-1])
        the.dotC[[c(1, 2, 2, 2)]] <- paste("port_opt_", cfun, "_Sp", sep = "")
        ans.len <- length(ans)
        dotC.num <- match("the..C.call", as.character(ans[[length(ans)]]))
        ans[[c(ans.len, dotC.num)]] <- the.dotC[[1]]
        ans
}
```

There's a lot to explain here—we'll just go in order. The first section makes sure that the arguments are right, or can be fixed up to be right. All of the error creation is near the top of the function. In general this is a good idea so that the user doesn't need to wait around merely to be told that there was an error. In this case it is more than a good idea because we don't want to have the side effect of creating a permanent file, and then abort the function.

The second section builds the C code. It starts out by testing if the file that would be used is suitable—this uses `filetest` of page 162. Then the template is catted into the file, the required strings pasted together, and then substituted into the file with `substifile` of page 277.

The last section creates the S function to be returned. Here we come across the statement:

```
        ans.len <- length(ans)
        ans[[c(ans.len, 1, 2, 2)]] <- as.name(parameter)
```

Subscripting with a vector or list inside `[[` is described on page 203. The first few lines of `portoptgen.stemplate` (which is the same as `ans` here) are:

```
function(lower =  - Inf, upper = Inf, scale = 1,
        control = portopt.control(...), ...)
{
        p <- length(This.Should.Be.Gone)
        if(length(scale) != 1 && length(scale) != p)
```

Recall that the last component of a function is the body of the function. So the replacement into `ans` is picking out the second part of the second part of the first statement in `ans`, and putting in the name of the parameter.

Next comes `the.comm` which creates the comments at the start of the `ans` that is returned. `the.comm` starts out life as a character vector that has been pasted together. It is then pasted some more, parsed, and evaluated to form a function. This move to a function is basically so we have a container to hold the comments—they tend to run through your hands if you don't have a proper bucket. Next the two objects of mode "brace" are concatenated together with `c` to form the new body of `ans`.

Now comes:

```
new.fun <- vector("function", length(new.sargs) + 1)
names(new.fun) <- c(new.sargs, "")
ans <- c(new.fun[ - length(new.fun)], ans)
```

This creates a function that has arguments that are those that we want to add to the answer. Then the two functions are concatenated together, except that we take away the body of `new.fun`.

The final task to accomplish is to create the correct call to `.C` and place it in `ans`. This follows the same pattern—we want to add to a call, so we create another call and use `c` to combine them. Placing the call to `.C` is easy since the spot in `ans` where it should go is the character string `"the..C.call"`.

If gradients or Hessians are computable, then `portopt1` and `portoptgen` can be modified to use different PORT routines.

14.5 Things to Do

Add an integration method to `line.integral`.

Write a function similar to `interpolate.lagrange` or `portoptgen` that creates functions that perform some other task.

Rewrite `genopt` so that births, jittering, and so on are all abstracted (there are separate functions for each).

Design a genetic algorithm for an optimization problem that you have.

14.6 Further Reading

You can learn about integration in the complex plane from a text on complex analysis, one that I happen to have is Bak and Newman (1982) *Complex Analysis*. Discussions of numerical integration are in numerical analysis books. One such book is Mathews (1987).

Interpolation is discussed in many numerical analysis books, for example, Hildebrand (1974) *Introduction to Numerical Analysis*.

A good source on genetic algorithms is D. E. Goldberg (1989) *Genetic Algorithms in Search, Optimization & Machine Learning*. I also like his article (1989) "Zen and the art of genetic algorithms." John Holland introduced genetic algorithms; his book is (1975) *Adaptation in Natural and Artificial Systems*.

Gay (1990) gives details about how to use some of the PORT routines, but I haven't found anything that gives a good explanation of how the optimization is performed.

14.7 Quotations

[37]May Swenson “Question”

Chapter 15

Large Computations

How to live well on nothing a year.

15.1 Backups

When the computations are large in the sense of time-consuming but the answer is not especially large, I use a backup scheme. This saves partial results in case the job stops for some reason.

Here is an example of the technique.

```
"fjjsimbk"<-
function(n, dof, trials = 100)
{
        backup <- paste("sbk.BACKUP", unix("echo $$"),
                sep = ".")
        simbk.sub <- function(n, dof)
        {
                sum(rt(n, dof))
        }
        ans <- numeric(trials)
        attr(ans, "call") <- match.call()
        for(i in 1:trials) {
                ans[i] <- simbk.sub(n, dof)
                if(i %% 10 == 0)
                        assign(backup, ans, where = 1,
                                immediate = T)
        }
        ans
}
```

In a real application the sub-function would not be so trivial (I would hope). The two statements involving `backup` implement the idea. The first statement creates an object name that will be essentially unique from call to call of the function. The second statement assigns the backup object to the working database at appropriate intervals. The `immediate=T` is key—this makes sure that the object is actually written to disk.

To use this, we might make two different calls to the function, probably in two different files input to different `BATCH` jobs.

```
ts.10.5 <- fjjsimbk(10,5)

ts.20.3 <- fjjsimbk(20,3)
```

At a later time we can see what exists in our database:

```
> objects()
[1] ".First"           ".Last.value"
[3] ".Random.seed"     "sbk.BACKUP.13517"
[5] "sbk.BACKUP.13843" "ts.10.5"
```

Here we see that there is a backup object for each process, but only one "real" result from the function. It could be that the other call has merely not finished yet, or it may have crashed. The `call` attribute allows us to sort out which backup object is for which call.

```
> attr(sbk.BACKUP.13517, "call")
fjjsimbk(n = 10, dof = 5)
> attr(sbk.BACKUP.13843, "call")
fjjsimbk(n = 20, dof = 3)
```

This scheme is useful for simulations and similar situations in which there are a number of results. It is harder to apply where there is really just a single answer.

An alternative approach would be to have the function not return the object, but rather use the object that is assigned internally be the "result" of the function. In general, I'd rather avoid this—the hidden side effects can create confusion. Of course, `fjjsimbk` has hidden side effects also, but the side effects are of secondary importance and are unlikely to cause trouble. On the other hand, if the answer is a large object, then it may be wiser not to have two copies of it. *For all averred, I had killed the bird* [38]

15.2 Reduce and Expand

I have created the generic functions `reduce` and `expand` to relieve memory problems. My first application was to minimize the use of disk-space by removing components of objects that can be recreated. For example, we could create

an `lm.reduced` class that saves space by deleting the residuals. The residuals can be recreated by the `expand` method assuming that the original response is still available and unchanged.

The default methods for `reduce` and `expand` both just return the first argument unchanged. This allows you to try to reduce an already reduced object, or expand an unreduced object with no harm done.

```
"reduce.default"<-
function(x, ...)
x
```

Another use for `reduce` is to save on memory in computations. The reduced object is passed in, and only expanded at the last minute when needed.

```
"fjjtwomat"<-
function(n, m, k)
{
        seed <- .Random.seed
        ans <- list(x = array(rnorm(n * m), c(n, m)),
                y = array(rnorm(m * k), c(m, k)))
        ans$seed <- seed
        class(ans) <- "twomat"
        ans
}
```

The function above produces an object of a specific class that holds two matrices. Below is a matrix multiplication method for this class.

```
"%*%.twomat"<-
function(x)
x$x %*% x$y
```

The next function is a `reduce` method that stores the two matrices in new locations, and remembers those locations.

```
"reduce.twomat"<-
function(x, immediate = F)
{
# attempt to make unique name
        basename <- paste("rEdUtwomat", unix("echo $$"
                ), sep = ".")   #
# ensure that it is unique
        xname <- paste(basename, "x", sep = ".")
        count <- 0
```

```
        bnchar <- nchar(basename)
        while(exists(xname, where = 1)) {
                count <- count + 1
                basename <- paste(substring(basename,
                        1, bnchar), count, sep = ".")
                xname <- paste(basename, "x", sep =
                        ".")
        }
        yname <- paste(basename, "y", sep = ".")
        dbname <- search()[1]   # need absolute path
        if(AsciiToInt(dbname)[1] != 47) {
                dbname <- paste(unix("pwd"), dbname,
                        sep = "/")
        }
        assign(xname, x$x, where = 1, immediate =
                immediate)
        assign(yname, x$y, where = 1, immediate =
                immediate)
        x$x <- c(name = xname, db = dbname)
        x$y <- c(name = yname, db = dbname)
        class(x) <- c("twomat.reduced", class(x))
        x
}
```

Most of the work is in ensuring that the names are unique and the full name of the working database is known. (Actually the code takes the liberty of assuming that if the "x" name doesn't exist, then neither does the "y" name; this may not be a good idea depending on the mechanism used for removing these objects.) Below is an object of the reduced class—this is quite a small object, and is the same size no matter how large the matrices it represents.

```
> jjtr
$x:
                name                                db
 "rEdUtwomat.18173.x" "/user/users/burns/spo/.Data"

$y:
                name                                db
 "rEdUtwomat.18173.y" "/user/users/burns/spo/.Data"

$seed:
 [1] 41 57 51  7  5  2 21 49 27 17 30  0

attr(, "class"):
[1] "twomat.reduced" "twomat"
```

Here is the method for expanding the reduced object. It merely gets the real

matrices, and peels off the first element of the class.

```
"expand.twomat.reduced"<-
function(x)
{
        xn <- x$x
        yn <- x$y
        x$x <- get(xn["name"], where = xn["db"],
                immediate = T)
        x$y <- get(yn["name"], where = yn["db"],
                immediate = T)
        class(x) <- class(x)[-1]
        x
}
```

It is a good idea to check that the methods work in the sense that reducing and then expanding gets you back to the same thing.

```
> all.equal(jjt, expand(reduce(jjt, im=T)))
[1] T
> all.equal(jjt, expand(reduce(jjt, im=F)))
Error in get.default(x$x["name"], whe..: Object "rEdU\
        twomat.18201.x" not found
Dumped
```

This shows the usefulness of the `immediate` argument. If `immediate` is FALSE, then the objects that the `reduce` method create are not actually written (in S jargon, not committed) until just before the next S prompt is given. In the example above where an error occurs, the objects are merely marked to be written, but do not actually exist at the time that `expand` demands them.

Below are some ingredients for testing how much memory is saved.

```
> fjjmul1
function(x)
{
        cat("memory at start  ", fjjcomma(memory.size(
                )), "\n")
        y <- x
        cat("memory before fjjmul2  ", fjjcomma(
                memory.size()), "\n")
        ans <- fjjmul2(y)
        cat("memory after fjjmul2  ", fjjcomma(
                memory.size()), "\n")
        ans
}
> fjjmul2
```

```
function(y)
{
        sum("%*%"(expand(y)))
}
> jjtmbig <- fjjtwomat(100, 200, 200)
> fjjcomma(object.size(jjtmbig))
[1] "480,475"
```

And now we do the test, starting a new session each time.

```
> fjjmul1(jjtmbig)
memory at start   633,296
memory before fjjmul2   1,321,424
memory after fjjmul2   2,783,696
[1] -1066.744
```

Now, with the reduced version.

```
> fjjmul1(jjtmbig.red)
memory at start   633,296
memory before fjjmul2   834,000
memory after fjjmul2   2,296,272
[1] -1066.744
```

So we have saved one copy of the matrices in this example.

15.3 Things to Do

Revise or add to the `reduce.twomat` example so that there is not a problem with stored components existing that are no longer pointed to by a `twomat.reduced` object.

15.4 Further Reading

Vanity Fair by William Makepeace Thackeray.

15.5 Quotations

[38]Samuel Taylor Coleridge "The Rime of the Ancient Mariner"

Chapter 16

Esoterics

Where we sail off the edge.

16.1 Magnitude

We have come all this way without a theorem, I now eliminate the deficiency.

Theorem 1 *An S object with at least one attribute consumes an infinite amount of space.*

Proof: The demonstration uses the in-built matrix `freeny.x`, but the particular object—as long as it has an attribute—is immaterial. Consider the sequence:

```
> attributes(attributes(freeny.x))
$names:
[1] "dim"      "dimnames"

> attributes(attributes(attributes(freeny.x)))
$names:
[1] "names"

> attributes(attributes(attributes(
+         attributes(freeny.x))))
$names:
[1] "names"
```

Obviously this continues *ad infinitum* so the size of the object is bounded below by:

```
> object.size(attributes(attributes(attributes(
+         freeny.x)))) * Inf
```

```
[1] Inf
```

Q.E.D. ♠

16.2 Dostoevsky

> Twice-two-makes-four is a farcical, dressed-up fellow who stands across your path with arms akimbo and spits at you. Mind you, I quite agree that twice-two-makes-four is a most excellent thing; but if we are to give everything its due, then twice-two-makes-five is sometimes a most charming little thing, too.
>
> Fyodor Dostoevsky *Notes from the Underground*
> translated by David Magarshack

The object of the game is to make twice two equal five.

My first method is rather cheap and unsatisfactory. It gives the required answer:

```
> 2 * 2
[1] 5
```

but is clumsy elsewhere.

```
> 2 * 0:9
 [1]  1  3  5  7  9 11 13 15 17 19
```

Here's the adjustment:

```
> get("*")
function(e1, e2)
.Internal(e1 * e2, "do_op", T, 4) + 1
```

The next attempt is slightly better, though it goes to the bother of writing a new function.

```
> twice(2)
[1] 5
> twice(1:9)
[1]  2  5  6  8 10 12 14 16 18
```

Behind the scenes is a little work involving two's and five's:

```
"twice"<-
function(x)
{
        2 * x + dnorm(x - 2, sd = 1/sqrt(2 * pi)/5)/5
}
```

This definition of `twice` gives answers close to the usual except for inputs close to 2.

16.3 Things to Do

Make $2 * 2 = 5$.

Develop a topology for S objects.

16.4 Further Reading

Catch-22 by Joseph Heller.

Epilogue

If I've convinced you that abstraction and consistency are your friends, that redundancy is your enemy, that artistry in programming is worth seeking—that is, if you have become a proud, lazy, impatient, distrustful simpleton—then in some measure I have succeeded. *it's not elves exactly* [39]

It is no accident that abstraction is given top billing in the previous paragraph. Unix is a powerful operating system because it has simple but general abstractions. C has become a major programming language because it has efficient abstractions that allow the user effective control of the machine. Your code, too, can be potent if you create the right abstractions. Perhaps I haven't mentioned abstraction enough. Abstraction, abstraction, abstraction.

So, what are the steps of programming?

- Think about what functionality is desirable. Then think about how to generalize that.
- Find algorithms that are as simple and general as possible.
- Implement the algorithms as simply and generally as you can.
- Document your code clearly and completely.
- Figure out how to test your code thoroughly.
- Write a verification suite.
- Put your code, documentation and tests under a source code control scheme.
- Revise as necessary.

Notice that the actual code generation is but a fraction of the time involved (unless perhaps you are just learning the language). It isn't hard to get something to happen, but it can be devilish to get the right thing to happen.

```
q()
```

or just some human sleep [40]

Quotations

[39]Robert Frost "Mending Wall"
[40]Robert Frost "After Apple Picking"

Bibliography

[1] Abelson, H., Sussman, G. J., and Sussman, J. *Structure and Interpretation of Computer Programs, Second Edition.* MIT Press, Cambridge, Massachusetts, 1996.

[2] Abramowitz, M., and Stegun, I. *Handbook of Mathematical Functions.* Dover, New York, 1972.

[3] Bak, J., and Newman, D. J. *Complex Analysis.* Springer-Verlag, New York, 1982.

[4] Bates, D., and Watts, D. G. *Nonlinear Regression Analysis and Its Applications.* John Wiley and Sons, New York, 1988.

[5] Becker, R. A., and Chambers, J. M. *S: An Interactive Environment for Data Analysis and Graphics.* Wadsworth, 1984.

[6] Becker, R. A., and Chambers, J. M. *Extending the S System.* Wadsworth, 1985.

[7] Becker, R. A., Chambers, J. M., and Wilks, A. R. *The New S Language.* Chapman and Hall, 1988.

[8] Bentley, J. L. *Programming Pearls.* Addison-Wesley, Reading, Massachusetts, 1986.

[9] Bentley, J. L. *More Programming Pearls: confessions of a coder.* Addison-Wesley, Reading, Massachusetts, 1988.

[10] Beran, J. *Statistics for Long-Memory Processes.* Chapman and Hall, New York, 1994.

[11] Breiman, L., Friedman, J. H., Olshen, R. A., and Stone, C. *Classification and Regression Trees.* Wadsworth, Belmont, California, 1984.

[12] Chachra, V., Ghare, P. M., and Moore, J. M. *Applications of Graph Theory Algorithms.* Elvesier North Holland, New York, 1979.

[13] Chambers, J. M., and Hastie, T. *Statistical Models in S.* Chapman and Hall, 1992.

[14] Cleveland, W. *Visualizing Data*. Hobart Press, 1993.

[15] Darnell, P. A., and Margolis, P. E. *C: A Software Engineering Approach, Third Edition*. Springer-Verlag, New York, 1997.

[16] Davis, A. M. *201 Principles of Software Development*. McGraw-Hill, New York, 1995.

[17] Gay, D. M. Usage summary for selected optimization routines. Tech. rep., ATT Bell Laboratories, Murray Hill, New Jersey, 1990.

[18] Goldberg, D. E. *Genetic Algorithms in Search, Optimization & Machine Learning*. Addison-Wesley, Reading, Mass., 1989.

[19] Goldberg, D. E. Zen and the art of genetic algorithms. In *Proceedings of the 3rd International Conference on Genetic Algorithms* (1989), pp. 80–85.

[20] Golub, G. H., and Van Loan, C. F. *Matrix Computations, Third Edition*. Johns Hopkins University Press, Baltimore, Maryland, 1996.

[21] Hamming, R. W. *Numerical Methods for Scientists and Engineers*. Dover, New York, 1973.

[22] Hastie, T., and Tibshirani, R. *Generalized Additive Models*. Chapman and Hall, 1990.

[23] Hildebrand, F. B. *Introduction to Numerical Analysis*. Dover, 1974.

[24] Holland, J. *Adaptation in Natural and Artificial Systems*. University of Michigan Press, Ann Arbor, Michigan, 1975.

[25] Jennison, C., and Sheehan, N. Theoretical and empirical properties of the genetic algorithm as a numerical optimizer. *Journal of Statistical Graphics and Computing 4* (1995), 296–318.

[26] Kernighan, B. W., and Plauger, P. J. *The Elements of Programming Style*. McGraw-Hill, New York, 1974.

[27] Kernighan, B. W., and Ritchie, D. M. *The C Programming Language, 2nd Edition*. Prentice-Hall, Englewood Cliffs, New Jersey, 1988.

[28] Knuth, D. E. *Fundamental Algorithms, Second Edition*. Addison-Wesley, Reading, Massachusetts, 1973.

[29] Knuth, D. E. *Seminumerical Algorithms, Second Edition*. Addison-Wesley, Reading, Massachusetts, 1981.

[30] Libes, D. *Obfuscated C and Other Mysteries*. John Wiley and Sons, New York, 1993.

[31] Mathews, J. H. *Numerical Methods for Computer Science, Engineering, and Mathematics*. Prentice-Hall, Englewood Cliffs, New Jersey, 1987.

[32] MathSoft, Inc. *S-PLUS Programmer's Manual*. MathSoft, Inc., 1994.

[33] McCullagh, P., and Nelder, J. A. *Generalized Linear Models, Second Edition*. Chapman and Hall, New York, 1989.

[34] Nelder, J. A., and Mead, R. A simplex method for function minimization. *Computer J. 7* (1965), 308–313.

[35] Peek, J., O'Reilly, T., and Loukides, M. *Unix Power Tools, Second Edition*. O'Reilly and Associates, Sebastopol, CA., 1997.

[36] Polya, G. *How to Solve It, Second Edition*. Penguin, 1990.

[37] Press, W. H., Teukolsky, S. A., Vetterling, W. T., and Flannery, B. P. *Numerical Recipes in C, Second Edition*. Cambridge University Press, Cambridge, 1992.

[38] Ripley, B. *Pattern Recognition and Neural Networks*. Cambridge University Press, Cambridge, 1996.

[39] Schwartz, R. L. *Learning Perl*. O'Reilly and Associates, Sebastopol, California, 1993.

[40] Seber, G. A. F., and Wild, C. J. *Nonlinear Regression*. John Wiley and Sons, New York, 1989.

[41] Spector, P. *An Introduction to S and S-Plus*. Wadsworth, Belmont, California, 1994.

[42] Venables, W., and Ripley, B. *Modern Applied Statistics with S-Plus, Second Edition*. Springer-Verlag, New York, 1997.

[43] Wall, H. S. *Analytic theory of continued fractions*. Van Nostrand, New York, 1948.

[44] Wall, L., Christiansen, T., and Schwartz, R. L. *Programming Perl*. O'Reilly and Associates, Sebastopol, California, 1996.

[45] Young, D. M., and Gregory, R. T. *A Survey of Numerical Mathematics*. Dover, New York, 1973.

Glossary

abscissa 1) *Math.* the x-coordinate.

address 1) *Comp.* a number that represents a location in memory.

aggregate 1) *Stat.* to combine consecutive observations in a **time series**; for example, a monthly time series of amount sold can be aggregated into an annual series.

AIC 1) *Stat.* "An Information Criterion", a method of combining the **likelihood** and the number of parameters so that non-nested models can be compared, due to H. Akaike. AIC has been criticized for suggesting models with too many parameters. See also **SIC**.

algorithm 1) *Comp.* a recipe for a computation. This can either be specific as in Euclid's algorithm for the greatest common divisor, or be a general way of approaching a problem as in the **EM algorithm**.

alias 1) *Unix.* a name that expands into some (part of a) command; similar to a **macro** in C. 2) *Stat.* in fractional experimental designs, some effects are aliased (or confounded) with others.

alpha version 1) *Comp.* a prototype of some software, numerous and serious **bugs** are to be expected. See **beta version**.

alphanumeric 1) the alphabetic characters plus the ten digits.

ampersand 1) the character "&", a symbol used in S, C and Unix.

argument 1) *Comp.* an input into a function or subroutine. 2) *Math.* the angle of a **complex number** from the positive real axis, calculated in S with `Arg`.

argument matching 1) *Comp.* the process of deciding which arguments in a particular call corresponds to the formal arguments of the routine being called. This is a trivial exercise except in languages like S that allow default values for arguments.

array 1) *S.* an object that has a `dim` **attribute**, and represents a (hyper) rectangle of objects. Also a function for creating those objects. 2) *C.* an object that is the equivalent to a **vector** in S. See also **ragged array**.

ASCII 1) *Comp.* a standard for encoding characters. Pronounced "ASS-key", it is an acronym for American Standard Code for Information Interchange. Each character uses 8 bits (1 byte), but many programs (including S) only deal well with the first $128 = 2^7$ characters.

aspect ratio 1) *Comp.* the ratio of the x-axis to the y-axis. This can refer either to the physical size of the graph, or to the units represented by the graph.

assembly code 1) *Comp.* a program in the machine language for a particular chip—very fast, but untidy.

assignment 1) *Comp.* the process of giving a name to some entity. Performed in S with "`<-`".

assignment function 1) *S.* the definition of a function that appears on the left of an assignment. For example:

```
names(x) <- letters
```

The name of the `names` assignment function is "`names<-`". If `fjj` is a generic assignment function, then "`fjj<-.middle`" would be the assignment function for `fjj` for class `middle`.

association 1) *Math.* an operator $\circ$ is associative if $(x \circ y) \circ z = x \circ (y \circ z)$. 2) *Comp.* an operator $\circ$ associates from left to right if $x \circ y \circ z$ is evaluated the same as $(x \circ y) \circ z$, and it associates from right to left if it evaluates as $x \circ (y \circ z)$.

asymptotic 1) *Stat.* an approximation of a quantity based on the limit as the number of observations goes to infinity. For example, the asymptotic variance of an estimator.

at sign 1) the character "@", a symbol not used in S version 3, but is used to access **attributes** in version 4.

atomic 1) *S.* an object that is one of the five modes `numeric`, `logical`, `character`, `complex`, `null`. All elements of an atomic vector are of the same type.

attach 1) *S.* the process of adding a **database** to the **search list**. This is done with functions `attach`, `library`, `module`.

attributes 1) *S.* a named list that is a subsidiary part of an object. Examples of attributes include `class`, `names`, `dim`. Also the function that retrieves or changes this list.

audit 1) *Comp.* a record of the commands performed, generally with enough information that all states can be reproduced. S keeps such a record in the `.Data/.Audit` file.

axis 1) *Math.* the line in a graph along which one variable is zero. 2) *S.* the information such as tick marks and tick labels that are put along an axis. The axes are numbered 1 for the bottom, 2 for the left, 3 for the top, and 4 for the right.

background 1) *Unix.* a process is in the background if it is running, but not listening to **standard-in**. An example is an S `BATCH` job. A command can be put in the background by placing an **ampersand** at the end of the command.

background color 1) *S.* the color of a blank graph, in S this is denoted color 0.

backquote 1) the symbol `, unused in S, used in Unix to surround commands that are to be interpreted. See also **single quote**.

backslash 1) the symbol " \ " used as an **escape** character by many languages.

backward-compatibility 1) *Comp.* the state such that old code will run properly using a new version of the software.

bang 1) *Comp.* slang for the exclamation mark "!". Also called "screamer".

batch 1) *Comp.* a style of computing in which all instructions are submitted and then results are obtained; compare to **interactive**.

benchmark 1) *Comp.* the process of testing the speed of one computer (or piece of software) relative to another. A task that is harder than it seems since the relative speed of different tasks is often not very similar.

beta test See **beta version**.

beta version 1) *Comp.* a version of some software that is thought to be close to release quality, but which has only been used by the developers. Allowing a group of people to use the version in return for reporting bugs and giving general comments is the *beta test*.

bias 1) *Stat.* the difference between the mean value of an estimator and the true (unknown) value that is being estimated. Often there is a trade-off between bias and variance when choosing an estimator.

binary file 1) *Comp.* a file that is in a format suitable to some program as opposed to a format for being read by humans. Generally, any file that is not in ASCII format is said to be a binary file. 2) *Comp.* in particular, **object code**.

binary number 1) *Math.* a number in base 2, composed only of the digits 0 and 1.

binary tree 1) *Math.* a relative of a mathematical **graph** in which each **node** has zero, one or two children (other nodes), and "left" children are distinguished from "right" children. It is the distinction of left and right that makes a binary tree different from an ordinary mathematical **tree**.

bit 1) *Comp.* a binary digit. 2) *Comp.* the amount of information contained in a binary digit.

bivariate 1) *Stat.* a situation involving two random variables. The smallest case of **multivariate**.

BLAS 1) *Comp.* acronym for Basic Linear Algebra Subroutine. If your computer has the BLAS, it does not have a case of ennui but rather it has subroutines for doing some linear algebra tasks that are optimized for that machine.

blue book 1) *S.* Becker, Chambers and Wilks (1988) *The New S Language.*

body 1) *S.* the part of a function where the computations are, as opposed to the **arguments**.

bootstrap 1) *Stat.* a way of evaluating the variability of an estimate by resampling the input data. 2) *Comp.* starting a simple process that allows a similar but more complex process to work.

Bourne shell 1) *Unix.* one of the most common Unix **shells**, very popular for programming.

box 1) *S.* the lines around the **plot area** of a **figure**.

box constraint 1) *Math.* constraints on a vector such that there is a minimum and a maximum for each element (independently of the other elements). Compare **linear constraint**.

boxplot 1) *Stat.* a graph indicating the distribution of some data. There are various definitions, but a common one (and the one used in S) is: The ends of the box are the first and third quartiles, the mark within the box is the median. The "whiskers" stretch to the farthest datapoint that is within distance d of the near end of the box, where d is some number times the length of the box. Points farther than d from the box are individually marked.

brace 1) one of the symbols "{" or "}", used in S and in C to bind several commands into one. See also **bracket**, **parenthesis**.

bracket 1) one of the symbols "[" or "]", used in S and in C for **subscripting**. Sometimes called "square bracket" to distinguish from the more general use of the term "bracket" that includes **braces** and **parentheses**.

brain-dead 1) *Comp.* slang for something that is ill-planned and/or ill-executed.

buffer 1) *Comp.* a chunk of memory set aside for a specific purpose. For example printed output in C uses a buffer—characters to be printed are put in the buffer and printing actually takes place when the buffer is full; then the buffer is marked empty so it can accept more characters.

bug 1) *Comp.* a problem with some software or hardware. 2) *Comp.* pejorative for **feature**.

byte 1) *Comp.* the basic unit of memory for a computer, in most cases consisting of 8 **bits**.

C shell 1) *Unix.* a popular **shell** for interactive use, less popular for programming.

cache 1) *Comp.* a storage area that is treated specially in some way. Almost always the term is used to mean something that is time efficient—it can be a memory location, a **hash table**, a **lookup table**, etc.

call 1) *S.* the action of invoking a function. 2) *S.* a **mode**, an object of mode `call` contains the information for a specific call to a function. 3) *S.* a **component** or **attribute** of many S objects that is the call that created the object of which it is a part (the mode of this component or attribute is generally `call`).

canonical correlation 1) *Stat.* a way of assessing the realationship between two multivariate sets of data.

carriage return 1) *Comp.* an **ASCII** character rather like **newline**, but not quite the same.

case-sensitive 1) *Comp.* a program is case-sensitive if capital letters are considered to be distinct from lower-case letters. S, Unix and C are case-sensitive; Fortran and DOS are not.

category 1) *Stat.* a variable that takes on a finite number of values and the values are generally non-numeric. Compare to **continuous variable**. The levels can be ordered as in {"low", "medium", "high"}, or not as in {"corgi", "kitty", "human"}. 2) *S.* an object that represents a category; the preferred type of object is one that inherits from class `factor`.

ceiling 1) *Math.* the smallest integer greater than or equal to a number. Compare **floor**.

central processing unit 1) *Comp.* the computer chip that does the actual computation. Abbreviated as "CPU".

chaos 1) *Math.* a process that appears to be random, but is really deterministic. **Pseudorandom numbers** are an example of chaos.

character 1) *S.* an **atomic** mode, each element of a character vector is a character string of arbitrary length.

characteristic 1) *Comp.* the exponent of a number stored in **floating-point**. The other part is the *mantissa*. 2) see **eigen**.

characteristic function 1) *Stat.* the Fourier transform of the density function of a random variable. One use is to compute moments of the distribution.

child 1) *S.* a **memory frame** is the child of the frame that caused it to come into existence. Frame 1 is no one's child. 2) *Unix.* a **process** is the child of the process that spawned it.

class 1) *S.* an **attribute**, the `class` attribute of an object determines which **method** of a **generic function** is dispatched when the object is a particular argument in the function call.

classification tree 1) *Stat.* a statistical model that describes a **categorical** variable by means of a recursive partition of the **explanatory variables** (where each partition is of a single explanatory variable). Fit in S with `tree`. See also **regression tree**.

clustering 1) *Stat.* a technique for grouping datapoints by their similarity; this is called "unsupervised learning" in the machine-learning literature.

coercion 1) *Comp.* the conversion of an object from one concept to another, such as from character to numeric, or from double-precision floating-point to single-precision floating-point.

collision 1) *Comp.* the occurrence of more than one value being encoded to the same location in a **hash table**.

colon 1) the character ":" which is used in S to produce sequences.

column-major order 1) *Comp.* the storage order of the elements of a matrix in which all of the elements of the first column are together, followed by the elements of the second column, etc. S and Fortran store matrices (and higher dimensional arrays) in column-major order. Compare to **row-major order**.

combination 1) *Math.* a selection without replacement of a certain number of elements from a finite set where the order of the selection is ignored. The binomial coefficient gives the number of combinations. Compare with **permutation**.

command line editing 1) *Comp.* the ability to recall past commands, edit them and execute the modified command. In S this can be accomplished with S-mode for emacs, or through the `-e` flag to S-PLUS.

command 1) a non-technical term—something that will make the computer do something.

comment 1) *Comp.* text in software that is ignored in computations, and is used to inform humans reading the code. In S, text following a **pound sign** on a line is a comment.

commitment 1) *S.* an **assignment** of an object to a permanent location is committed when it is actually written to disk, which often is not until S is about to give a prompt rather than at the time the assignment is made. In the interim S acts as if the object were in the specified location even though it isn't physically there.

commute 1) *Math.* an operator $\circ$ commutes if $x \circ y = y \circ x$.

compacting 1) *S.* term used to mean **garbage collection**. Compacting usually occurs only inside of loops.

comparison operator 1) *Comp.* an operator that tests equality or inequality. The comparison operators in S are `==`, `!=`, `>`, `<`, `>=`, `<=`.

compile 1) *Comp.* the act of converting **source code** into the corresponding instructions for a particular computer chip and operating system. The result is a file of **object code**.

compile-time 1) *Comp.* the time at which a program was **compiled**, as opposed to when it is used which is called **run-time**.

compiled language 1) *Comp.* a language, C or Fortran for example, in which the source code is **compiled** into **object code** which can then be executed. Compare to **interpreted language**.

complex 1) *S.* an **atomic** mode, an object of this mode contains **complex numbers**.

complex number 1) *Math.* a number which is a real number plus a real number times the square root of -1. The square root of -1 is often denoted "i", which is the case in S.

component 1) *S.* an item in a **list** or other recursive object. The length of a list is the number of components it has. Compare with **element**.

condition number 1) *Comp.* a number that states the degree of ill-conditioning in a computation. By convention, small numbers mean good results, and large or infinite values mean poor results. For example, the condition number for inverting a matrix is often defined as the ratio of largest to smallest singular value.

conditional distribution 1) *Stat.* a probability distribution that assumes some condition holds. For example, the distribution of height given that weight is less than some specific amount.

conditioning plot 1) *Comp.* a plot of some sort that is conditional on a value or set of values of a variable that is not represented in the plot. The `coplot` function does this in S.

confounded 1) *Stat.* two effects are confounded if the data at hand can not distinguish if it is only one or the other that has influence on the **response**. For example, if you only observe people who either use both alcohol and tobacco or use neither, then you won't be able to discern the health effects of one of the drugs. Confounding is often a matter of degree, not just totally confounded versus not confounded at all.

confusion matrix 1) *Stat.* when a **categorical** variable is being modeled, a matrix that is a **contingency table** of the true categories versus the predicted categories from the model of the data.

conjugate 1) *Math.* a number is the (complex) conjugate of another if it has the same real part but the opposite imaginary part, so $a - bi$ is the conjugate of $a + bi$. Computed in S with `Conj`. 2) *Stat.* in Bayesian statistics there are conjugate distributions, meaning that the computations simplify when the two distributions are used together as **prior** and **likelihood**.

constant 1) *S.* an object—like the number 5.13—that is fixed. 2) *Math.* an object that is fixed, but not necesarily known.

constraint 1) *Math.* a condition placed on a problem. Two common classes of constraint are **box constraint** and **linear constraint**.

contingency table 1) *Stat.* a table of counts of the number of occurrences in the combination of levels of two or more **categorica**l variables. When there are two variables, then the table is represented as a matrix. More variables means higher dimensional arrays. These are created in S with `table`. 2) where you eat when your kitchen table has a broken leg.

continuation 1) *S.* an incomplete statement may be given to S and the continuation of the command given on one or more additional lines. The default continuation prompt is "`+ `". 2) *Math.* a means of extending a function over more of the complex plane.

continuous variable 1) *Stat.* a variable that can (conceptually) take on any value in some range of the real numbers. Compare to **category**.

contour plot 1) *Comp.* a graphic that shows the value of a function of two variables by drawing lines of equal value of the function.

contrast 1) *Stat.* a coefficient vector that sums to zero. Important because they are **orthogonal** to the vector of all 1's which corresponds to the mean. 2) *S.* an **attribute** or **component** of some objects representing a statistical model that shows the set of contrasts that was used in the fitting process. (These are not necessarily contrasts in the sense of meaning 1.)

control 1) *Comp.* the act of keeping a series of backups of objects such as files of code or documentation. In Unix this is done with SCCS or RCS (and probably other methods).

control operator 1) *S.* one of the operators `&&` or `||` which perform non-vectorized "and" and "or", respectively.

CPU 1) *Comp.* acronym for **central processing unit**.

crash 1) *Comp.* slang for the unintentional or unexpected halt of a program or of hardware.

cross-validation 1) *Stat.* a process of estimating the quality of a model by comparing observations left out of the fitting process with the prediction of that fit for the observations. For example, the available observations are randomly divided into 10 groups, the fitting is performed 10 times with a different group dropped each time. Read carefully when this term comes up—there are some confusing uses of it.

cursor 1) *Comp.* the marker on a computer screen that indicates where input is to go, or the spot where the mouse is pointing.

cut and paste 1) *Comp.* the action when using a window system of marking some text in one spot and making that text appear in another spot.

data frame 1) *S.* an object of class `data.frame` that represents a rectangle of values in which the type of value may be different from column to column. Compare to **matrix**.

database 1) *S.* an item on the **search list**, or something that might be. 2) *Comp.* a program designed to store information—generally in the form of **fields** and **records**, and often using **SQL**.

debug 1) *Comp.* the process of tracking down and fixing a problem. [The entomology of this word is reputed to be that a moth flew into the Mark I computer in 1945, crashing the machine. The problem was fixed by "debugging" the relay that failed. If this story isn't true, it should be.]

decimal number 1) *Math.* a number written in base ten.

declaration 1) *Comp.* the specification of the type of object that a variable is to be. This is done in compiled languages like C and Fortran, but essentially non-existent in S.

decomposition 1) *Math.* writing a matrix as the product of two or more other matrices of special types; for example, **QR decomposition**.

default 1) *S.* the value of an optional argument that is not given in the call.

density 1) *Stat.* a function that defines a probability distribution. If the distribution is discrete, then the density at a point is the probability of the distribution taking that value. If the distribution is continuous, then the density is such that the integral between any two points of the density is the probability of the distribution landing in that interval.

deparse 1) *S.* to change a language object into one or more character strings that represents the object.

dependent 1) *Stat.* two random variables are dependent if knowing information about one helps predict the other. 2) *Math.* a dependent variable is a function of one or more other variables. 3) *Stat.* a **response** in a statistical model. This is in analogy to meaning 2, but confusing because of meaning 1.

deque 1) *Comp.* a data structure similar to a **stack** or a **queue** except that both adding and deleting are allowed at both ends. [I believe the name derives from "double-ended queue."]

dereference 1) *C.* the act of getting the value pointed to by a **pointer**, done with the $*$ operator.

derivative 1) *Math.* the slope of a function at any given point.

deviance 1) *Stat.* a measure of the distance of the data from its fit to a model. The form of the measurement depends on the assumed **error** distribution.

diagnostics 1) *Stat.* statistics and graphics intended to show how well a statistical model fits the data—an important part of the modeling process.

diagonal matrix 1) *Math.* a matrix that is zero except on the diagonal (where the row number is equal to the column number).

dim 1) *S.* the **attribute** that describes the shape of an array. Also the function that retrieves or changes this attribute.

directory 1) *Comp.* a location that contains files and (possibly) sub-directories.

distribution 1) *Stat.* a description of how a particular random variable behaves.

distribution function 1) *Stat.* a function of x that is the probability that a random variable will be less than or equal to x.

distribution-free test 1) *Stat.* a class of hypothesis tests which use ranks of the data. An example is the Wilcoxon signed-rank test. Some times called "nonparametric".

dot plot 1) *Comp.* a plot that displays information as plotting symbols that are each some distance along their respective lines—used as an alternative to barplots and pie charts. These are created in S with `dotchart`.

dot product 1) *Math.* a number which is the sum of the product of corresponding elements of two vectors.

$$\mathbf{u} \cdot \mathbf{v} = \sum_{i=1}^{n} u_i v_i$$

where n is the length of the vectors. This can be done in S via `sum(u*v)`.

double quote 1) the character `"`. See also **backquote** and **single quote**.

double-precision 1) *Comp.* a **floating-point number**, usually consisting of 8 **bytes**. 2) *Comp.* computation performed with double-precision numbers.

dump 1) *S.* an **ASCII** representation of an S object created by `dump` or `data.dump` (these two representations are distinctly different). 2) *S.* the action of representing the state of S when an error occurs. This generally goes to the object `last.dump`. 3) *Comp.* an error action as in a core dump.

dynamic graphics 1) *Comp.* graphics that change spontaneously or from input by the user, usually to display high-dimensional data.

dynamic load 1) *Comp.* the process of adding code to a program that is already running. This is performed in S with `dyn.load` and its relatives. Compare **static load**.

dynamic programming 1) *Comp.* a technique for optimizing over n time points. Given the state at time $n-1$, the best decision to take is computable; so work backwards through time to the present.

echo 1) *Comp.* the act of the computer repeating input so it can be seen. Passwords are not echoed when they are typed in.

edge 1) *Math.* a part of a mathematical **graph** that connects two **nodes**.

effect 1) *Stat.* in 2^k designs, twice the estimated coefficient for a factor or interaction. 2) *S.* a component of an object of class `aov` that is a vector of orthogonal single degree-of-freedom values. This is not the same as definition 1.

efficiency 1) *Comp.* some measure of the amount of computing time, memory or other resources that a program uses. 2) *Stat.* the variance of an estimator (usually) measured relative to the variance of the best possible estimator given the model.

eigenvalue 1) *Math.* When an equation of the form $A\mathbf{x} = \lambda\mathbf{x}$ is satisfied, then λ is an *eigenvalue* of the square matrix A, and $\mathbf{x}$ is an eigenvector of A. If A is real-valued and symmetric, then all of the eigenvalues are real-valued. Eigenvalues are sometimes referred to as "characteristic values".

eigenvector See **eigenvalue**.

element 1) *S.* an item of an **atomic** vector. The **length** of an atomic vector is the number of elements that it contains.

ellipsis 1) the set of characters "...", also known as **three-dots** (which see).

EM algorithm 1) *Stat.* an approach to estimating a statistical model in which there is some form of missing data. The expected value of the missing data is taken (the E step), then the model **likelihood** is maximized given these values (the M step). These two steps are iterated until convergence. Convergence can be slow, but the code can be easier to write than alternatives (if there are any).

environment variable 1) *Unix.* variable that a **process** passes on to its children.

EOF 1) *Comp.* acronym for "End Of File."

error 1) *Stat.* a random disturbance, or the difference between the data and a **model** of the data. In this sense of the word, it does not mean "mistake". Most statistical models specify the distribution of the errors. 2) *Comp.* a condition that causes some program to stop execution—see **error handling** and **warning**. 3) *Comp.* non-technically, a **bug**. See also **numerical error** ,

error handling 1) *Comp.* the actions that happen when a program hits an error condition. See **dump** (meaning 2) for what happens in S.

escape 1) *Comp.* to temporarily leave a program to give an operating system command, done in S with the `!` form. 2) *Comp.* a character that allows the following character (or set of characters) to have a meaning other than usual. In S the back-slash is the escape character in character strings.

evaluation 1) *S.* the process of doing the actual computation, after the command has been **parsed**.

evaluation frame 1) *S.* see **memory frame**.

event 1) *Comp.* an action that happens on a computer, such as a mouse click or the cursor entering a certain region. **Version 3** of S knows nothing about events, but **version 4** does.

executable 1) *Comp.* (adjective) the condition when a file can be executed, that is, will perform some action. 2) *Comp.* (noun) the file that executes a program, `Sqpe` is the executable of S.

expected value 1) *Stat.* the mean of a probability distribution.

explanatory variable 1) *Stat.* a variable in a statistical **model** that is used to approximate the **response**. This is also known as a "predictor" or the confusing "independent variable."

exponential notation 1) the same as **scientific notation**.

expression 1) *S.* a **recursive** mode in S that is the **parsed** form of one or more commands. 2) *Comp.* used non-technically to mean a command or part of a command. See also **regular expression**.

extraction See **subscript**.

factor 1) *S.* class of an object that represents a **categorical** variable. 2) *Stat.* a categorical variable in an experimental design (hence meaning 1). 3) *Stat.* a (hypothetical) latent variable in, for example, a factor analysis model. 4) *Math.* relating to a product, as in "an integer can be factored uniquely as the product of prime numbers."

factorial 1) *Math.* n-factorial, written $n!$, is the product of the integers from 1 to n. In S this is found with `gamma(n+1)`. Because the numbers quickly get large, `lgamma(n+1)` is usually more useful.

false convergence 1) *Comp.* state of a derivative-based optimizer in which it knows that it has not found an optimum, but doesn't know in which direction to go. This can be caused by various forms of ill-conditioning, but the surest way to achieve it is to get the sign of the gradient wrong.

family 1) *S.* a class of object used in `glm` and `gam` that describes the **link function** and **error** distribution.

FAQ 1) *Comp.* acronym for Frequently Asked Questions.

feature 1) *Comp.* euphemism for **bug**.

field 1) *Comp.* area of an **ASCII** file that is to be read as a unit, one or more fields comprise a **record**. For example, there could be two fields, species and weight, and each record would correspond to a different individual. 2) *Comp.* a similar concept in any **database**.

FIFO 1) *Comp.* acronym for "First In, First Out", a **queue**.

figure 1) *S.* conceptual area of a **graphics frame** that contains a **plot area** surrounded by a **margin**. There can be one or more figures in a frame, all of which are surrounded by the **outer margin**.

file 1) *Comp.* an object of the operating system that holds information (and often has a name).

FILO 1) *Comp.* acronym for "First In, Last Out", a **stack**.

fixed format 1) *Comp.* a style of **ASCII** file in which the **fields** are each a specified number of characters and in a specific location relative to the left border. This is opposed to a file which divides the fields by a delimitor such as a semicolon.

fixed-width font 1) *Comp.* a font in which each character is the same width.

flag 1) *Unix.* an optional argument to a Unix command, introduced by a dash. For example the "t" in `ls -t`. 2) *Comp.* in programming, a logical variable that signals a certain condition. For example, `has.names` in `c.rationalnum` on page 251 is a flag.

floating-point number 1) *Comp.* a computer representation of a real number composed of a fractional part, called the *mantissa* and an exponent, called the *characteristic*. Because the representation is finite, few numbers are represented exactly, and hence **numerical error** will usually result during computations.

floor 1) *Math.* the largest integer less than or equal to a number. Compare **ceiling**.

font 1) *Comp.* a set of descriptions of how characters are to be represented. This can be either rather general, as in "computer modern font"; or very specific, as in "10 point italic computer modern font."

formula 1) *S.* class of object that results from an expression that uses the **tilde** operator. Used to provide an argument of arbitrary complexity. Operators generally have special meanings within formulas.

fragmented memory 1) *Comp.* the condition of **RAM** in which there are numerous unused portions, making the total use of memory greater.

frame 1) *S.* there are (at least) three types of "frame" in S: **memory frame** (also called "evaluation frame"), **data frame**, and **graphics frame**.

function 1) *S.* **mode** of an object that defines computations. The computations are performed when a call to the function is evaluated. 2) *Math.* a correspondence from one set into another set.

functional language 1) *Comp.* a class of computer languages that avoids **side effects**. S is essentially a functional language, but would not be considered one by a functional language purist.

garbage collection 1) *Comp.* the act of freeing up memory that is no longer being used.

Gaussian distribution 1) *Stat.* the most common probability distribution in Statistics. Also known as "Normal" and "bell curve".

generalized additive model 1) *Stat.* a statistical model in which the response is approximated by a **link function** of the sum of nonlinear **smooths** of each explanatory variable. The **error** distribution can be one of several. Fit in S with `gam`.

generalized linear model 1) *Stat.* a statistical model in which the response is approximated by a **link function** of a linear combination of the explanatory variables. The **erro**r distribution can be one of several. Fit in S with `glm`.

generic function 1) *S.* a function whose behavior is determined by the **class** of one or more of its arguments. The class of the relevant argument(s) determines which **method** of the generic function is used.

genetic algorithm 1) *Comp.* a form of **random algorithm** used for optimization that imitates a natural selection process.

gigabyte 1) *Comp.* 10^9 bytes. 2) *Comp.* 2^{30} bytes.

global variable 1) *Comp.* (in S terminology) a variable that is not created locally, and is not passed in as an argument.

gotcha 1) Short for "got you", the "Choppy Water" chapter contains a number of gotchas.

gradient 1) *Math.* vector which is the first derivative of a multivariate function.

graph 1) *Math.* a collection of **nodes** (also called *vertices*) and **edges** which each connect two nodes. A particular type of graph is a **tree**. 2) *Comp.* (or "graphics") a pictorial representation of information. In S you must start a **graphics device** before creating graphics.

graphical parameter 1) *S.* same as **graphics parameter**.

graphical user interface 1) *Comp.* a system of how human and computer communicate, generally consisting of windows, menus, etc. The Motif flavor of X-Windows is an example.

graphics device 1) *S.* a program that renders graphics in some form. For example, some create a window and draw graphics in the window. The `postscript` device translates the graphics into PostScript and then sends it to a printer or a file. 2) *S.* an S function that starts such a program.

graphics frame 1) *S.* if the graphics are printed, this is equivalent to a single page. A frame contains one or more **figures**, and an **outer margin**.

graphics parameter 1) *S.* a number, string or logical value that controls some aspect of how a graph will look, or that describes the state of the graphics device. The `par` function queries and changes graphics parameters.

gray scale 1) *Comp.* the rendering of values by means of different shades of gray. This can be done in S-PLUS with `image`.

greedy algorithm 1) *Comp.* (not strictly an algorithm, but more a description of some algorithms) an optimization algorithm that always does the best it can on each step, but does not look ahead. For some problems this will produce the true optimum, but for lots of problems it will not.

group 1) *S.* a collection of **generic functions** that may have one **method** written for all of them. 2) *Math.* a set for which there is an associative operation, the identity for the operation is in the set, and the inverse of every element in the set is also in the set.

GUI 1) *Comp.* acronym for **Graphical User Interface**.

hack 1) *Comp.* to illegally enter (or attempt to enter) a computer. 2) *Comp.* to discover how some undocumented program works by experimenting with it. 3) *Comp.* a **kludge** (meaning 1).

hacker 1) *Comp.* a person who "hacks" (meaning 1). 2) *Comp.* slang for programmer, as in "S hacker."

Hadamard product 1) *Math.* element by element product of matrices. The operation performed by `*` on matrices in S (the `%*%` operator is used for matrix multiplication).

hang 1) *Comp.* said of a computer or program when nothing happens any more, as in "S hangs when I execute this command." This is probably caused by one of three things: it is not doing anything, there is an infinite loop, the computations take muxh longer than expected.

hard-coded 1) *Comp.* a part of a program that is not easily changed. For example, the functionality performed in a `.Internal` function in S is hard-coded since it can not be changed by users. Another example is a concept in a program that is always used as a constant rather than abstracted into a variable.

hash symbol 1) the character "#", also called "pound sign."

hash table 1) *Comp.* a table of values in which each record has a key on which a mathematical operation is performed to give an address for the record. This provides fast access to any record for which the key is known.

hat 1) slang for the caret character, ˆ. Used in Statistics to mean an estimate, often specifically the **MLE**.

hat matrix 1) *Stat.* the matrix (in linear regression) that puts the "hat" on the vector of observations, that is, the hat matrix times the observation vector equals the fitted observations. $\hat{y} = Hy$. It is mainly the diagonal of the hat matrix that is of interest in diagnostics.

header file 1) *C.* a file usually ending in ".h" that contains definitions of general use—intended to be included in code. The `S.h` file, which can be used for C code to be called by S, is an example.

help file 1) *S.* the on-line documentation on individual objects. The corresponding term in Unix is "man page".

Hermitian matrix 1) *Math.* a matrix that is equal to the **transpose** of its complex **conjugate**. Also called "self-adjoint." For real-valued matrices, this is the same as symmetric.

Hessian 1) *Math.* matrix which is the second derivative of a multivariate function.

hexadecimal 1) *Math.* referring to base 16.

high-level graphics function 1) *S.* a function that makes a new **graphics frame**, and then creates a graph.

high-level graphics parameter 1) *S.* a **graphics parameter** that may not be given to `par`.

histogram 1) *Stat.* a plot showing the distribution of a variable by means of boxes that show the count of observations in a number of adjacent intervals. Created in S by `hist`.

hook 1) *Comp.* a feature of a program that allows some functionality to be easily added later if desired. As in, "`line.integral` has a hook for additional methods of integration."

idempotent 1) *Math.* an object such that the product of itself with itself equals itself. 0 and 1 are idempotent, as are some matrices.

independent 1) *Stat.* two distributions are independent if information on one gives you no information about the other. Mathematically this is expressed by the density of the **joint distribution** being equal to the product of the densities of the **marginal distributions**. 2) *Math.* an independent variable is in the domain of a function. 3) *Stat.* the term "independent variable " is used to mean an **explanatory variable** in analogy with meaning 2, but confusing because of meaning 1.

index 1) *S.* an element or component number. In the S command `x[4]`, 4 is the index into `x`.

indirection 1) *Comp.* a form of abstraction in which a variable is created to avoid **hard-coding** something. For instance, creating a variable for the first part of a path to a file.

information matrix 1) *Stat.* matrix that measures the variability of estimated parameters. In **maximum likelihood estimation**, this can be approximated using the **Hessian** at the optimum.

inheritance 1) *Comp.* the feature of **object-oriented programming** in which a subclass of objects can use some of the methods for the main class. In S inheritance results when a **class** attribute has length greater than one.

input/output 1) *Comp.* also "I/O", functionality related to information entering or leaving a program.

integer 1) *Math.* one of the numbers $0, \pm 1, \pm 2, \ldots$ 2) *Comp.* a value known to be an integer. Compare to **floating-point**. 3) *S.* sometimes a vector with storage mode `integer`; sometimes a floating-point number that is logically integer.

integer programming 1) *Comp.* an optimization problem with (linear) constraints in which the elements of the answer must be integer-valued.

interactive 1) *Comp.* a program that converses with the user. S is an interactive language since it takes input from the user, responds to it, and then asks for more input. Opposed to **batch**.

interpreted language 1) *Comp.* a language that is not compiled, so that the interpretation into machine instructions is performed on the fly. S is an interpreted language. Compare **compiled language**.

interquartile range 1) *Stat.* the distance between the first quartile (25th percentile) and the third quartile (75th percentile) of a probability distribution or empirical sample.

interrupt 1) *Comp.* a **signal** that tells a program to quit doing what it is doing at the moment. Control-c is the usual key binding for the interrupt signal. Sometimes "interrupt" is used in a less technical sense to include other signals as well, such as control-**backslash**.

invisible 1) *S.* a value that is not printed automatically.

IQR 1) *Stat.* acronym for the **interquartile range**.

iteration 1) *Comp.* performing an operation repetitively in a loop while some things change. Compare to **recursion**. 2) *Comp.* one pass through an iterative computation.

joint distribution 1) *Stat.* a multivariate probability distribution of all of the variables in question. Compare with **marginal distribution**.

julian date 1) *Comp.* an integer that represents the number of days from a certain date.

justify 1) *Comp.* to place text in a certain alignment. For example, left justify means to place along the left margin.

K 1) *Comp.* slang for **kilobyte**.

kernel 1) *Unix.* the heart of the Unix system, the part that is the same no matter what **shell** is used. 2) *Stat.* a function that produces the weights for a moving average-type (kernel) **smooth**.

key binding 1) *Comp.* the association of a key or key combination on the keyboard to a specific meaning such as "erase" or "interrupt."

kilobyte 1) *Comp.* $1024 = 2^{10}$ bytes. 2) *Comp.* 1000 bytes.

kludge 1) *Comp.* a piece of programming that is ugly. 2) *Comp.* a piece of programming that is so ugly, it's sort of cute.

knap-sack problem 1) *Math.* a specific class of optimization problem. The prototypical problem is to maximize the value of what goes into your knap-sack while keeping the weight below a certain limit.

Kronecker product 1) *Math.* a product of two matrices in which the result has dimensions that are the products of the dimensions of the matrices being multiplied. The S-PLUS `kronecker` function generalizes the concept beyond multiplication.

labels 1) *S.* a generic function that creates labels for various objects. 2) *S.* an argument to `factor` and `ordered` that controls the value of the `levels` attribute of the output.

language 1) *S.* an object that possesses one of several **modes** that correspond to the language itself rather than to data. See also **functional language**, **interpreted language**, **compiled language**, **natural language**.

LAPACK 1) *Comp.* a package of Fortran routines for performing linear algebra, available from **netlib**. Some LAPACK **symbols** are in S-PLUS.

lazy evaluation 1) *Comp.* strategy in which objects are not actually **evaluated** until necessary. S uses lazy evaluation.

least absolute deviations 1) *Stat.* principle of estimation in which the sum of the absolute value of the estimated errors is minimized. This typically overcomes some of the statistical **robustness** problems of least squares, but has other statistical deficiencies and is a little more expensive to compute. Also known as LAD, least sum of absolute errors, LSAE, $\mathcal{L}_1$.

least squares 1) *Stat.* principle of estimation where the sum of squares of the estimated errors is minimized. Useful because of its computational simplicity in many situations, and its theoretical optimality given a Gaussian distribution, but it often fails to be statistically **robust**.

legend 1) *Comp.* a part of a graph that explains the meaning of the symbols used in the main part of the graph.

length 1) *S.* a fundamental property of an S object. In numeric vectors, for example, it states how many numbers the object contains.

levels 1) *S.* **attribute** of a categorical object that is a character vector containing the possible values that the variable can take on. The `labels` argument to `factor` and `ordered` can be used to control the resulting `levels` attribute.

library 1) *S.* a directory containing S objects that is set up so that the `library` function can **attach** it. 2) *Comp.* a set of compiled routines. A user's **object code** is **linked** against a library to get **symbols** that are defined in the library.

likelihood 1) *Stat.* the density of a statistical model thought of as a function of the parameters of the distribution with the data known. The **maximum likelihood estimate** is the value of the parameters that maximizes this function. 2) *Stat.* the value of the likelihood (previous meaning) or its logarithm when it is maximized.

limitation 1) *Comp.* something that a program might be expected to do but doesn't. A **bug** of omission.

linear constraint 1) *Math.* a constraint on vectors $\mathbf{x}$ that can be written as $\mathbf{cx} < a$ or $\mathbf{cx} \leq a$ or $\mathbf{cx} = a$. A particularly simple and common set of linear constraints is a **box constraint**.

linear programming 1) *Comp.* an optimization problem in which the objective function is a linear function of the variables, and there are linear constraints on the variables. Abbreviated as "LP".

link 1) *Comp.* to combine files of **object code** together so that **symbols** from one that are used in another are resolved. 2) *Unix.* a file name that points to a file with another name. This can be used for abstraction, or to save space.

link function 1) *Stat.* in a **generalized linear model**, the function of the linear combination of the **explanatory variables** that approximates the **expectation** of the **response**. A function that performs a similar task in other models, such as **generalized additive models**.

linked list 1) *Comp.* a data structure in which each item in the list contains a pointer to the location of the next item. Specializations include doubly linked lists in which each item has a pointer to the previous item as well as the next item, and circular lists in which the final item has a pointer to the first item.

literal 1) *Comp.* in a language, text that represents some specific thing, as opposed to a variable. For example, "3.1" is a literal for the number 3.1.

load 1) *Comp.* to make compiled routines available to a program by making sure that the **symbol table** of the program contains the routines. This can be done statically or dynamically. See **dynamic load**, **static load**.

logical 1) *Comp.* something that is True or False. In S a third logical value is `NA`. 2) *S.* an **atomic** mode.

long-tailed distribution 1) *Stat.* a probability distribution that has a higher probability of points far from the mean than a **Gaussian** distribution with the same variance. Such distributions are problematic for **least squares** estimators.

lookup table 1) *Comp.* a table of values used in a computing strategy in which values are stored and retrieved rather than being computed each time.

low-level graphics function 1) *S.* a function that does not switch to a new **graphics frame**, but adds to a graph instead. Examples include `points`, `abline`, `axis`.

LP 1) *Comp.* acronym for **linear programming**.

machine dependent 1) *Comp.* something that changes between types of computers. For example, the order of bits in a **floating-point number** is different on some machines than on others. The order of bits for **ASCII** characters is the same on all computers, so ASCII is not machine dependent (which is the main point of ASCII).

macro 1) *C.* a definition that is substituted into the code in the **preprocessing**.

man page 1) *Unix.* On-line help for a Unix command.

mantissa 1) *Comp.* the fractional part of a number in a **floating-point number**. The exponential part is called the *characteristic*.

margin 1) *S.* the conceptual part of a graph that surrounds the **plot area** but is within a **figure**. Axis labels and titles are in the margin. See also **outer margin**.

marginal distribution 1) *Stat.* the univariate probability distribution of a variable (within a multivariate context). Compare with **joint distribution**.

masking 1) *S.* an object masks another object if they have the same name (and sometimes the same **mode**), and the first is earlier on the **search list** than the second.

mathematical graph 1) *Math.* See **graph**.

matrix 1) *Math.* a rectangular display of numbers that can be treated as a single entity. 2) *S.* an object that has a `dim` of length two. (This is the definition `is.matrix` uses, includes **data frames**.) 3) *S.* an object that has a `dim` attribute of length two and is also an **array**. (This is the definition `as.matrix` uses, excludes data frames.)

maximum likelihood estimation 1) *Stat.* a principle of estimation which uses the most probable value of a set of parameters given the data and a statistical model. In some circumstances, models exist such that the maximum likelihood estimator is equivalent to **least squares**, or to **least absolute deviations**.

mean 1) *Stat.* a measure of the central location of a probability distribution, for empirical data it is the sum of the observations divided by the number of observations. For a single sample it is a **least squares** estimate.

median 1) *Stat.* a measure of the central location of a probability distribution, it is the 50th percentile. For empirical data, at least half the data are bigger than or equal to the median, and at least half are smaller than or equal to it. The median is not uniquely defined for all datasets. It is a **least absolute deviation** estimate for a single sample.

meg 1) *Comp.* slang for megabyte. As in, "My S function gobbled 55 megs."

megabyte 1) *Comp.* 2^{20} bytes. 2) *Comp.* 1,000,000 bytes.

memory 1) *Comp.* storage of objects. This is often **RAM**, but may be **swap-space**, disk-space, etc.

memory frame 1) *S.* the collection of values that are local to a particular function during an evaluation. Almost every function call results in a new frame as it is evaluated. Also called "evaluation frame."

menu 1) *Comp.* a device in interactive computing that provides a number of choices to the user from which the user is to select.

method 1) *S.* a function that performs the computation of a **generic function** for a specific **class**.

micro- 1) a prefix meanining one-millionth (10^{-6}).

missing 1) *S.* **mode** of an object that is an **argument** which was not included in the **call**.

missing value 1) *S.* a value written `NA` that represents a value that is unknown. See also **not-a-number**.

MLE 1) *Stat.* acronym for **Maximum Likelihood Estimate**.

mode 1) *S.* a fundamental property of an S object which states what type of object it is. 2) *Stat.* the most likely value of a probability distribution. One way to estimate this for continuous distributions is to find the maximum of a density estimate of the data. For discrete distributions use `table`.

model 1) *Stat.* a hypothetical picture of how some particular set of data is generated, usually consisting of a functional description plus a probability distribution for the errors from the functional part. G. E. P. Box says, "All models are wrong, some models are useful."

modular programming 1) *Comp.* discipline in programming that allows pieces of the code to work independently of each other.

modulo 1) *Math.* a number which can be thought of as the remainder after a division. The `%%` operator performs this in S.

modulus 1) *Math.* the distance of a complex number from the origin, equivalent to the absolute value for real numbers. In S the `Mod` function returns this.

monadic 1) *Math.* **unary**.

Monte Carlo 1) *Stat.* evaluation by means of a random mechanism. Basically, a synonym for simulation. For example, a multivariate integral can be approximated by sampling uniformly over the integration space and averaging the function values of those points—this is Monte Carlo integration.

multitasking 1) *Comp.* Using the same resources for more than one job by a time-sharing mechanism. Unix arranges for its processes to be multitasked.

multivariate 1) *Stat.* pertaining to more than one variable. Opposed to **univariate**, but includes **bivariate**.

NA 1) *S.* how **missing values** and **not-a-numbers** are printed.

name 1) *S.* **mode** of an object that is a variable within an **expression**. Not to be confused with the **names** attribute.

names 1) *S.* an **attribute** of many objects that labels the elements or components of the object. Also the function that retrieves or sets this attribute.

NaN 1) *Comp.* a common way that **not-a-number** is printed. In S they are printed `NA`, but can be distinguished from ordinary missing values with `is.nan`.

nano- 1) a prefix meaning one-billionth (10^{-9}), as in "π seconds are a nanocentury."

natural language 1) a language that humans have developed to communicate with each other. Examples are English, French and Northern Sahaptin. Opposed to computer language.

netlib 1) *Comp.* on-line repository of a large amount of software and other information. Much of the software is numeric. The world wide web address is `http://www.netlib.org/`.

newline 1) *Comp.* a character that means a new line of text is to be started. Represented in S and in C by backslash-n. It is different than **carriage return**.

node 1) *Math.* a part of a mathematical **graph** that is usually represented as a point. Also called "vertex".

nonparametric 1) *Stat.* a means of estimation that does not depend on optimizing parameters, many **smooths** are nonparametric. 2) *Stat.* a **distribution-free** procedure.

norm 1) *Math.* a number which gives the "size" of something. There are various norms for vectors, and for matrices.

not-a-number 1) *Comp.* one of a class of "numbers" that represents a limit that does not exist. Zero divided by zero, and infinity minus infinity are examples. Often printed as `NaN`, but in S printed as `NA`. S has the function `is.nan` to distinguish where not-a-numbers are.

NP-complete 1) *Comp.* the definition is a convoluted statement involving polynomial-time algorithms. The practical effect is that if you have a problem that is NP-complete, then the best you are likely to do is get a reasonable approximation to the solution. To be sure that you have the real solution is too hard in all but the smallest of problems.

null 1) *S.* **mode** of the object of zero length that is printed `NULL`.

numeric 1) *S.* **atomic** mode that represents real numbers. This contains the storage modes `double`, `single` and `integer`. Complex numbers are excluded.

numerical differentiation 1) *Comp.* approximating the derivative of a function at specific locations by performing arithmetic on function values. Not a trivial task since it involves dividing by approximately zero.

numerical error 1) *Comp.* the difference between an actual result as computed compared to what the answer would be if computations were performed with numbers of infinite precision. Note that "error" here is not used in the sense of a **bug**.

numerical integration 1) *Comp.* the approximation of an integral by some means. Also called "quadrature."

object 1) *S.* almost everything in S is an object—if it has a **mode**, it is an object.

object code 1) *Comp.* the **compiled** version of some code in a **compiled language**. Typically the file name ends with `.o` in Unix.

object-oriented programming 1) *Comp.* a style of software in which the action of some "message" (or function) depends on the type of object that receives the message. In S this is accomplished with **generic functions**, **methods** for generic functions and the `class` attribute.

octal 1) *Math.* pertaining to numbers in base 8.

OOP 1) *Comp.* acronym for **Object-Oriented Programming**.

operating system 1) *Comp.* software that controls the general operation of a computer. Unix and MacOS are operating systems.

operator 1) *S.* a function that does not need to use the usual format when called. Examples are `+`, `$` and `!` (when it means "not").

option 1) *S.* one of several values that control some aspect of the S session. These are queried and set by the `options` function.

optional argument 1) *Comp.* an argument to a function that need not be given. When it isn't given, then usually some default value is used for it.

orthogonal matrix 1) *Math.* a square matrix A is orthogonal (also called "unitary") if

$$A'A = I$$

that is, if the transpose of A is equal to the inverse of A. Some call this "orthonormal" and allow an orthogonal matrix B to be such that $B'B$ is some diagonal matrix.

orthogonal vectors 1) *Math.* two vectors are orthogonal if their **dot product** is zero.

outer margin 1) *S.* the coneptual area that is at the very edges of a **graphics frame**. This is usually only used (that is, has non-zero area) when there is more than one **figure** in the frame. See also **margin**.

outlier 1) *Stat.* a datapoint that is far enough from its predicted value in a statistical model that it strains the credibility of the model for that point.

overflow 1) *Comp.* the condition when an operation causes the answer to go out of bounds of the range of the number representation. This most commonly happens with **integers** since the range is relatively small, but may also happen with **floating-point numbers**. In S at least, overflow with floating-point numbers is little problem since the value gets set to infinity; but with integers the value is capricious.

overload 1) *Comp.* the action of making the same thing do more than one job. **Generic functions** in S are overloaded.

p-value 1) *Stat.* in hypothesis testing, the probability that something more extreme than the test statsitic would be observed if the null hypothesis were true. A small p-value is evidence against the null hypothesis being true (if the conditions assumed for the calculation of the p-value are close enough to being true).

packed 1) *Comp.* one of numerous ways of writing a **sparse** or repetitious matrix in a small amount of space.

paging 1) *Comp.* see **swapping**.

parent 1) *S.* the **memory frame** that caused the memory frame in question to come into existence. 2) *Unix.* the **process** that caused the process in question to come into existence.

parenthesis 1) either of the characters "(" or ")", which are respectively an "opening parenthesis" and a "closing parenthesis". Can also be used in the broader sense that includes **braces** and **brackets**.

parse 1) *Comp.* the process of breaking a group of characters into parts with respect to the **syntax** of the language.

paste 1) *Comp.* see **cut and paste**. 2) Both S and Unix have `paste` routines to combine text.

permission 1) *Unix.* the ability of the owner of a file, members of the owner's group and users at large to read, write and execute the file.

permutation 1) *Math.* a particular ordering of objects. There are n-**factorial** permutations of n distinguishable objects. See also **combination**.

permutation test 1) *Stat.* a test of "no effect" that is performed by permuting observations and comparing the real statistic with the distribution of statistics from the permuted data. It has the advantages that it is valid under very broad assumptions and it is easy to compute, but is limited in applicability. It is similar to the **bootstrap**.

pico- 1) prefix meaning 10^{-12}, as in picosecond.

pipe 1) *Unix.* a command that uses the `|` operator to take the **standard-out** from the process on the left as the **standard-in** for the process on the right.

pivot 1) *Comp.* there are various meanings, but commonly the reordering of computations in order to ensure the computation is as numerically **stable** as possible.

platform 1) *Comp.* non-technical term meaning the type of hardware, and possibly **operating system**. For example, Pentium machine running Linux.

plot area 1) *S.* the portion of a graph where the actual data go, surrounded by the **margin**. See also **figure**, **graphics frame**.

pointer 1) *C.* an object that holds the address of the real object of interest.

polar coordinates 1) *Math.* the designation of points by angle and distance from the origin. Compare to **rectangular coordinates**.

pop 1) *Comp.* the process of removing an item from a **stack**.

portable 1) *Comp.* code that can easily be adapted to numerous computers.

positive definite 1) *Math.* a symmetric matrix A is positive definite if $\mathbf{x}'A\mathbf{x} > 0$ for all $\mathbf{x}$. This is equivalent to all of the eigenvalues of A being positive.

positive semidefinite 1) *Math.* a symmetric matrix A is positive semidefinite if $\mathbf{x}'A\mathbf{x} \geq 0$ for all $\mathbf{x}$. This is equivalent to all of the eigenvalues of A being non-negative.

posterior distribution 1) *Stat.* in Bayesian statistics, the distribution of the model parameters after taking the data into account. The posterior is a combination of the **prior distribution** and the **likelihood**.

PostScript 1) *Comp.* a language that describes how text and graphics are to be rendered.

pound sign 1) The character "#", also called "hash symbol", used for comments in many languages including S and Unix, but not C.

precedence 1) *Comp.* the order in which calculations take place in an otherwise ambiguous context. For example `*` takes precedence over `+` in S and in C.

predictor 1) *Stat.* synonym for **explanatory variable**.

preprocessor 1) *C.* a process that changes the text of a source file before it is **compiled**. This is used, for example, to handle machine dependencies. The **pound sign** is used to indicate preprocessor commands.

principal components 1) *Stat.* a multivariate technique that creates a set of new variables (the principal components) that are uncorrelated with each other and are ordered such that the variance of each principal component is no larger than the preceeding one. The first principal component has as large of variance as possible.

prior distribution 1) *Stat.* in Bayesian statistics, the hypothesized distribution of the model parameters before the data are consulted. See **posterior distribution**.

process 1) *Unix.* the basic "being" in Unix, executing a command starts one or more processes.

projection 1) *Math.* a specific way of mapping a space into a smaller subspace. For example, the least squares **hat matrix** is a projection matrix.

projection pursuit 1) *Stat.* a multivariate technique that searches for "interesting" linear combinations of the variables. There are a few such techniques in use, for example projection pursuit regression.

prompt 1) *Comp.* characters used to indicate that an interactive program is ready for input from the user. The default prompt in S is "`> `".

pseudorandom number 1) *Comp.* one of a series of numbers generated by a mathematical function. The series is hoped to possess the most important empirical features of a truly random series of numbers.

push 1) *Comp.* the process of adding an item to a **stack**.

QP 1) *Comp.* acronym for **quadratic programming**.

QQplot 1) *Stat.* a plot to compare an empirical distribution to a theoretical probability distribution (or another empirical distribution). If the points are along a straight line, then the two distributions match (except possibly different means and variances). Points in the tails are much more variable than those in the middle. Functions in S for this include `qqnorm`, `qqplot` and `ppoints`.

QR decomposition 1) *Comp.* a class of algorithms for converting a rectangular matrix into an **orthogonal matrix** times an upper **triangular matrix**. This is convenient and numerically **stable** for least squares regression calculations.

quadratic programming 1) *Math.* the solution to a problem such as the minimum over all $\mathbf{x}$ of $\mathbf{g}'\mathbf{x} - .5\mathbf{x}'H\mathbf{x}$ subject to a set of linear constraints on $\mathbf{x}$.

quadrature 1) *Comp.* univariate **numerical integration**.

quantile 1) *Stat.* a value such that a random variable is less than the value a given percent of the time. The 95 percent quantile of a standard Gaussian distribution is approximately 1.64. For empirical data there are a number of ways of calculating the quantiles which are slightly different from each other.

quartile 1) *Stat.* a **quantile** that is at the 25th, 50th or 75th percentile. See also **interquartile range**.

queue 1) *Comp.* a data structure in which items are added at one end and removed from the other. Also called "FIFO" for "first in, first out". Compare to **stack**.

quick call 1) *S.* an S function that does not create a separate **memory frame**.

R 1) *Comp.* a language that is very similar to S—much S code can run properly in R. It is free software available via **statlib**, and was originally developed by Robert Gentleman and Ross Ihaka.

radian 1) *Math.* a unit of measure for angles, there are 2π radians in a circle. The trigonometric functions in S use radians.

radix 1) *Math.* the base of the representation of a number. Examples include decimal, binary, hexadecimal.

ragged array 1) *Comp.* a data structure in which there is some number of dimensions (as in an **array**), but not all of the "cells" contain the same number of items. These are discussed further on page 79.

RAM 1) *Comp.* acronym for **random access memory**.

random access memory 1) *Comp.* also called "main memory", the location where data needs to be in order to be accessed by a **central processing unit**. In S all of the objects involved in a computation must be held in RAM. 2) *Comp.* more generally (and accurately) a memory bank that can be accessed by address rather than sequentially.

random algorithm 1) *Comp.* an algorithm that uses (psuedo) random numbers, and hence often arrives at different solutions on different runs. Examples are **genetic algorithms** and **simulated annealing**. 2) *Comp.* used loosely to mean an algorithm that uses randomness, but all of the randomness is **independent**, unlike in a genetic algorithm or simulated annealing.

random seed 1) *Comp.* an object that holds the state of a pseudorandom number generator. In S the random seed is kept in the object `.Random.seed`.

range 1) *Stat.* the minimum and maximum of a dataset, or the distance between the two. 2) *Math.* set of values taken on by a function.

rank 1) *Math.* the maximum number of independent linear combinations that can be made with rows or with columns of a matrix. In mathematics there is a definite rank to a matrix. Computationally the rank is a much fuzzier concept due to **numerical error**—there is not a universally applicable algorithm to get the rank of a matrix. 2) *Stat.* the spot in the ordered sample that an observation falls; ties create fractional values. These are often used for **distribution-free** tests and statistically **robust** estimates. Computed in S with `rank`.

real-time 1) *Comp.* calculations that pertain to the situation at the moment that the calculations are available. For example, a monitor that shows current temperature is real-time, while one that shows the history

of yesterday's temperature is not. The time lag longer than which the computation is not "real-time" is application dependent.

record 1) *Comp.* in a **database**, a collection of **fields** constituting one observation.

rectangular coordinates 1) *Math.* a coordinate system in which the axes are linear and perpendicular to each other. Compare to **polar coordinates**.

recursion 1) *Comp.* a computation that is self-referential—a recursive function calls itself. Some problems are easily solved with recursion, but recursive algorithms tend to be inefficient in terms of computing resources. A recursion can always be recast as an iterative process.

recursive object 1) *S.* an object of one of several **modes** such as `list` and `expression` that can contain an object of the same mode. A list, for example, can have a **component** that is a list.

redirect 1) *Unix.* the action of changing how **standard-in**, **standard-out** and/or **standard-erro**r behave. For example standard-out can be redirected to a file named `my.out`.

register 1) *Comp.* a special memory location where access by the **central processing unit** is extremely fast.

regression test 1) *Comp.* a test of the quality of software that checks if a **bug** occurs that was previously found in the software. 2) It is certainly feasible that this phrase could appear in a statistical context.

regression tree 1) *Stat.* the same as a **classification tree** except that the **response** is **continuous** rather than **categorical**, fit in S with `tree`.

regular expression 1) *Unix.* the pattern matching facility used by `grep` and several other commands. This is not the same as the **wildcarding** that Unix uses with, for example, the `ls` command.

replacement 1) *S.* the use of **subscripts** on the left of an **assignment** to change the values in part of an object. This is using subscripting as an **assignment function**.

required argument 1) *Comp.* an argument to a function that must be given in its calls. When it isn't given, then an **error** results.

reserved word 1) *Comp.* a group of characters that can not be used as a variable name in a language because they mean something particular in the language. Examples of reserved words in S are `return`, `break`, `T`, `NULL`.

response 1) *Stat.* a variable that is approximated in a statistical **model** by a function of one or more **explanatory variables**. This is also confusingly called "dependent variable".

return value 1) *Comp.* the value that is left in place of a function call after it is evaluated. Compare **side effect**.

robust 1) *Stat.* an estimate or hypothesis test that is relatively reliable even when (some) assumptions are violated. For example, the median is robust to outliers while the mean is not. 2) *Comp.* code that works even given extreme conditions. Often this refers to code that is smart enough to retain as much precision as possible. The opposite of **unstable**. 3) *Comp.* an algorithm that works even under adverse conditions. For example, **genetic algorithms** are said to be robust because they work even with multiple local optima and if the objective is not differentiable.

root 1) *Math.* a value that makes a function zero. 2) *Math.* a **node** in a **tree** that is treated specially as the primary node.

rounding 1) *Math.* the process of dropping the least significant digits at some particular location relative to the decimal point. If a 5 is the single digit to be dropped, then it is not clear what to do and there is more than one convention—round to even, and round up are the two most common. Compare **truncate**.

rounding error 1) *Comp.* same as **numerical error**.

row-major order 1) *Comp.* the order of elements in a matrix in which the first row is filled, then the second row, etc. The IMSL C routines use row-major ordering. Compare **column-major ordering**.

RTFM 1) *Comp.* a suggestion to Read The Manual.

rug 1) *Comp.* marks along an **axis** of a plot indicating the **marginal distribution** of the data along that axis.

run-time 1) *Comp.* the time at which a program is used, as opposed to when it is compiled, which is called **compile-time**.

S-news 1) a mailing list for discussion of and questions about S and S-PLUS. At press time, the mechanism to subscribe is to send email to:
`S-news-request@wubios.wustl.edu`
with the body of the message equal to:
`subscribe s-news`
You should get back confirmation in an explanatory message.

sample 1) *Stat.* a selection of observations from some population. Usually used in the sense of a selection from a known population with a specific random mechanism.

scalar 1) *Math.* a single number, as opposed to a **vector** or **matrix**.

scientific notation 1) *Math.* a number written as a number within a certain range times 10 to some power. The number 123.4 is written in scientific notation as 1.234×10^2 or as $.1234 \times 10^3$, depending on the convention used. S (and many other programs) would write it as `1.234e2`.

script 1) *Unix.* a program written in a **shell** language.

search list 1) *S.* the collection of **databases** that an S **session** will search, in order, for objects. The analogue in Unix is the `path` variable.

seed 1) *Comp.* see **random seed**.

selection 1) *Comp.* see **subscript**.

semantics 1) *Comp.* the meaning attached to expressions in a language. The semantics affects the **evaluation** of an expression. Compare **syntax**.

semicolon 1) the character ";" used in S and in C to separate commands.

session 1) *S.* a process that starts when S is started (interactively or in batch) and ends when S is exited.

shell 1) *Unix.* the program that controls the interaction with the user. Examples are the **Bourne shell** and the **C shell**. It is a "shell" because it covers the **kernel** which is unchangeable.

SIC 1) *Stat.* the "Schwarz Information Criterion", sometimes denoted as BIC (but there is another slightly different BIC also). This, like **AIC**, is a way of combining the **likelihood** and the number of parameters so that non-nested models may be compared. In practice, this often suggests smaller models than does AIC.

side effect 1) *Comp.* any action of a program (function) except returning a value. Compare **return value**.

signal 1) *Comp.* a special message given to a program to change what it should do. An **interrupt** signal is an example.

significant digits 1) *Stat.* the number of digits in a number that are not just random noise. 2) *Comp.* used as in "most significant digits" or "least significant digits" (or bits) to mean the digits representing the left-most or right-most places.

simplex algorithm 1) *Comp.* a specific method of solving a **linear programming** problem. 2) *Comp.* an algorithm introduced by Nelder and Mead (1965) to solve general optimization problems without the assumption of differentiability. The algorithm starts with one more solution vector than there are parameters; each step replaces one of the solutions with a new one until a local minimum is found.

simulated annealing 1) *Comp.* a class of optimization algorithm that randomly explores around the current best value, and the size of the random jumps tends to be smaller and smaller as the optimization progresses. The analogy is to a material finding its minimum energy state as it cools. A competitor to **genetic algorithms**.

single quote 1) the character ', almost always on the same key as the **double quote**, not to be confused with the **backquote**.

single-precision 1) *Comp.* a **floating-point number**, generally using 4 bytes. 2) *Comp.* computations done with single-precision numbers.

singular matrix 1) *Math.* a square matrix that is not invertible (has determinant zero). 2) *Comp.* a square matrix that is numerically not invertible. The distinction between singular and non-singular is fuzzy with finite-precision numbers.

singular value decomposition 1) *Math.* the decomposition of a rectangular matrix into the product of an **orthogonal matrix** times a **diagonal matrix** times another orthogonal matrix.

slash 1) the character "/". Used in S and in C to mean division, and in Unix within path names. 2) *Stat.* a probability distribution that is equivalent to a standard Gaussian divided by a uniform(0,1).

smooth 1) *Stat.* a non-linear model of data, often created with a **non-parametric** method. This is most commonly **univariate**, but need not be.

source 1) *S.* the process of making S objects from an **ASCII** representation of the objects. This is done with the `source` function, but more loosely may refer to use of `restore`, `data.restore` and **redirection** of **standard-in**.

source code 1) *Comp.* the software as written in a language like C or Fortran that created a program. This usually refers to code for a **compiled language**, but may also refer to code from an **interpreted language** like S. See also **compile**, **object code**.

sparse matrix 1) *Math.* a matrix that contains a large number of zeros.

special value 1) *Comp.* a number-like entity: an infinity or **not-a-number**.

spline 1) *Math.* a function that is a piece-wise polynomial of a certain degree, often with the constraint that the function and a certain number of its derivatives are continuous.

SQL 1) *Comp.* acronym for "Structured Querry Language", the lingua franca of **database** management programs.

stable 1) *Comp.* a procedure that is **robust** (meaning 2).

stable distribution 1) *Stat.* a family of distributions that is very interesting from a theoretical view, but has found little practical use except for the **Gaussian** and Cauchy distributions which are special cases.

stable sort 1) *Comp.* any sorting method that ensures that, in the case of ties in the variables used to sort, observations within a tie retain their original order.

stack 1) *Comp.* a data structure in which items are added to and extracted from the same place, also called FILO for "first in, last out". The image is of a spring-loaded container for plates—new items are pushed onto the stack, and items are popped off of the stack. Compare **queue**. See also **push**, **pop**.

standard deviation 1) *Stat.* the variability of a probability distribution, defined as the square root of the variance.

standard error 1) *Stat.* the variability of an estimate, defined as the square root of the variance.

standard-error 1) *Unix.* the place where error messages go. Usually this is the screen.

standard-in 1) *Unix.* the place where information comes from. Usually this is the keyboard.

standard-out 1) *Unix.* the place where information goes. Usually this is the screen.

star 1) *Comp.* slang for the asterisk, the character "*". Often used to mean multiplication, but it has other meanings as well.

static load 1) *Comp.* the addition of code (**symbols**) to the **executable** of a program when it is not running. That is, a new, larger executable is created. In S this is done with the `LOAD` utility. Compare **dynamic load**.

statlib 1) a repository of software and other information pertaining to Statistics. It includes a large section of contributed software for S. Accessible on the world wide web with `http://lib.stat.cmu.edu/`.

status 1) *Unix.* the **return value** of a Unix program that indicates the **error** state—zero, by convention, means no error.

storage mode 1) *S.* sub-types of mode `numeric`. The possibilities are `integer`, `double` and `single`.

stream 1) *Comp.* a conceptually unending series of pieces of information. An example is input from a keyboard.

string 1) *Comp.* a sequence of characters.

subscript 1) *Comp.* the extraction (or replacement) of part of an object. In S and C this is done with **brackets**. 2) text that is written lower (and generally in a smaller **font**) than the regular text. In this sense, opposed to superscript.

swap-space 1) *Comp.* portion of disk-space reserved for overflow from **RAM**.

swapping 1) *Comp.* the process of switching information between **RAM** and **swap-space** so that all information required for the pending computation is in RAM. Also known as "paging".

symbol 1) *Comp.* a character string that identifies a **compiled** routine—often with **underscores** added to the routine's name. 2) *S.* the mark put at the location of a datapoint in a plot.

symbol table 1) *Comp.* a table kept by a program that relates **symbols** to **addresses**.

symbolic computation 1) *Comp.* mathematical computation, such as simplification and factoring, where some of the "numbers" are variable names.

symmetric matrix 1) *Math.* a square matrix that is equal to its own **transpose**. Compare **Hermitian matrix**.

syntax 1) *Comp.* the rules governing how expressions are made in a language, without regard to what the expressions mean. The syntax affects the **parsing** of an expression. Compare **semantics**.

system terminating 1) *S.* the action that S takes when it thinks something bad (usually inside calls to `.C` or `.Fortran`) has happened. S kills itself to make sure that no permanent data become corrupted.

tab 1) character written as backslash-t in S and C that represents some number of blank spaces. Counts as **white space**.

table See **contingency table**, **symbol table**, **lookup table**.

tera- 1) prefix meaning 10^{12} (or 2^{40}), as in terabytes of data.

three-dots 1) *S.* the construct "`...`" used to accept an arbitrary number of **arguments** in a function.

tick 1) *Comp.* mark along the **axis** of a graph indicating a value for that location.

tilde 1) the character "~" used in S to denote a **formula**.

time 1) *Comp.* an ambiguous concept on time-sharing computers since there is a difference between the total elapsed time from start to finish of a task ("wall time") and the time actually expended on the task ("computer time").

time series 1) *Stat.* a dataset where observations are at a sequence of time points, generally evenly spaced time points.

title 1) *S.* one of two **high-level graphics parameters** that label a graph. The `main` title is written at the top of the **figure**, the `sub` title is at the bottom. The titles appear in the **margin**.

token 1) *Comp.* an element of a **parsed** expression.

trace 1) *Math.* the sum of the elements on the diagonal of a matrix. 2) *S.* the process of modifying functions so that they produce some information when called, used for debugging.

transpose 1) *Math.* operation in which the rows of a **matrix** are interchanged with its columns.

tree 1) *Math.* a (connected) mathematical **graph** that contains no loops. See also **binary tree**, **classification tree**, **regression tree**.

trellis 1) *S.* system of **graphics functions** that focus on displaying **multivariate** data in a simple and comprehensible manner.

triangular matrix 1) *Math.* a matrix in which there are all zeros above the diagonal (lower triangular), or below the diagonal (upper triangular).

tridiagonal matrix 1) *Math.* a matrix in which the only non-zero elements are on the diagonal and the diagonals immediately above and below the diagonal. Compare to **diagonal matrix**.

truncate 1) *Math.* a relative of **rounding**, but the least **significant** digits or bits are just dropped with no change to what is left. Comare **rounding**.

type 1) *S.* a **high-level graphics parameter** that specifies how the data are to be represented in the graph. Choices include points, lines, **high-density** or none.

unary 1) *Math.* an operation on a single entity, for example "unary minus" to change the sign of a number. Also called "monadic".

underflow 1) *Comp.* the event in which a non-zero number becomes indistinguishable from zero due to finite-precision arithmetic. Generally, this is of little consequence.

underscore 1) the character "_", used in S to mean **assignment**, used in C within variable names.

univariate 1) *Stat.* a situation involving a single distribution. Compare to **multivariate**.

unstable 1) *Comp.* a computation that can be inaccurate due to **numerical error** or some other phenomenon. For example, dividing by a number close to zero is an unstable operation. The opposite of **robust** (meaning 2).

utility 1) *S.* an operating system command started by S. Examples are `BATCH` and `LOAD`.

value 1) *Comp.* often used in the sense of what is returned by a function (**return value**), as opposed to its **side effects**.

vector 1) *S.* an **atomic** object (common usage). 2) *S.* an object with no **attributes** (the definition that `as.vector` and `is.vector` use). 3) *S.* an object of one of many **modes** that allows an arbitrary **length**. 4) *Math.* an ordered collection of numbers.

vectorization 1) *S.* a function is vectorized when the output can contain an arbitrary number of answers. 2) *Comp.* computations are vectorized by some (super) computers. 3) *Math.* the columns of a **matrix** stacked on top of each other to form a **vector**. This is indicated by the "vec" operator, and is equivalent to an S matrix being treated as a simple vector.

version 3 1) *S.* the version of S that has existed since the early 1990's.

version 4 1) *S.* the version of S first appearing in 1998 that allows the equal sign to mean assignment and numerous other changes. Confusingly, S-PLUS Version 4.0 is based on Version 3 of S.

vertex (plural *vertices* or *vertexes*.) 1) *Math.* a part of a mathematical **graph** that is usually represented by a point—also called *node*.

virtual memory 1) *Comp.* **swap-space**.

wall time 1) see **time**.

warning 1) *Comp.* an **error**-like condition that does not interrupt computation, but merely issues a message.

white book 1) *S.* Chambers and Hastie (1992) *Statistical Models in S.*

white space 1) *Comp.* any of the characters "space", "tab", and sometimes "newline".

wildcard 1) *Unix.* The style of specification of multiple file names used by `ls` and some other commands. Compare to **regular expression**.

word 1) *Comp.* a **byte**.

workaround 1) *Comp.* a temporary fix for a **bug**. 2) *Comp.* a method of avoiding a bug by going a different route.

working database 1) *S.* the **database** that is first on the **search list**.

WYSIWYG 1) *Comp.* (pronounced WIZZ-ee-Wig) an acronym for "What you see is what you get."

Index