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"