Fundamentals

- Language notions.
- Functions and their actions. Vectorization.
- The S search path, R scoping and environments.
- Accessing specified frames with `assign` and `get`.
- Lazy evaluation and its consequences.

Language Characteristics

- A function language. Most operations are function calls. No ‘main program’ concept.
- An interpreted language rather than a compiled one. Special implications for programming.
- A language designed for interactive use as well as for programming.

A quick comparison of R and S

- Object permanency. R works “in memory” whereas S works more “on the disc”.
- Scoping rules. In R an expression is created within an environment, and if the value of the expression depends on its environment, the environment itself is part of the result.
  In S an expression is evaluated within an existing system of evaluation frames with strict visibility rules. If the frames change the expression may still exist but evaluate differently.
- Lazy evaluation. Both S and R use it but the details are slightly different.
R original objects v. methods package

- Backward compatibility attempts to be complete. At the interactive level, the casual user (in theory) should be able to work as before with little noticeable change.
- Major language changes are in OOP. Classes must be explicitly created and inheritance relationships explicitly established. Method dispatch is by signature and methods explicitly created. Much of the detail is held in the meta-data thus avoiding namespace conflicts.
- Connections are S objects generalizing the notion of files and extending the possibility for interprocess communication.
- Some syntactic sugar, notably the use of = as the (left pointing) assignment operator in most (but certainly not all) contexts.

What kinds of objects?

- Atomic objects: mode numeric, complex, character or logical.
- Language objects: functions, names, calls, expressions, grouped statements (objects of mode "{}"), assignments, quits.
- Recursive objects: lists, expressions, calls, functions.

No fundamental distinction is made between the objects of data analysis and the objects of the language itself.

Objects in S are normally permanent until deliberately discarded.

Objects in R are normally temporary unless the memory image is preserved.

The search path determines the (default) visibility of named objects.

Language Concepts

S is a language for manipulating objects.

What kind of language?

- A function language,
- An interpreted language,
- An interactive language.
- An object oriented language.

The goals of S: Efficient, safe and effective data analysis and graphics, algorithm development and technology transfer.

It is not necessarily the best environment for large, routine production jobs on present day hardware.

Objects and their properties

The object paradigm in S 3 is the structure.

This consists of a basic object, (called the .Data component) to which may be attached attributes.

The basic object has two properties: mode and length.

The attributes allow the object to be used in various special ways (or in some cases prevent certain actions).

Important attributes an object may have include class, names, dim, dimnames, levels.
Atomic and recursive objects

Atomic objects are of mode numeric, character, complex or logical. All components are of identical structure.

The prototype recursive object is one of mode list, but CALL, function, expression, and other modes are also in this category.

The components may be of a variety of different modes (in the case of a list, any mode).

Single and double square bracket operators behave the same for atomic objects but differently for recursive:

- Single square bracket: extract/replace a substructure of the same mode (typically list).
- Double square bracket: extract/replace a single component, which may be of any mode.

Examples

- A dim attribute allows access to components through a multiple index system (including the often forgotten ‘matrix index’),
- A names attribute allows access to components by names,
- A dimnames attribute allows access to index positions by name,
- A class attribute enables method dispatch by generic functions.
- Being of class factor prevents arithmetic on the object (which is still of mode numeric),
- Having a dim attribute restricts matrix operations to those that are matrix-wise coherent.

Environments

In R each object is evaluated within the environment within which it was created.

Environments form nested sequences all enclosed within the Global environment.

attach and library calls add to the global envirnoment in a nested way.

Function calls initiate new environments, inheriting from the global environment.

A function defined within another inherits its parent environment as well.
**Consequences of restricted visibility**

Consider a function to section each column of a matrix according to a factor and apply a function to each section.

```r
f1 <- function(X, f, FUN = mean) {
  v <- apply(X, 2, function(x)
    tapply(x, f, FUN))
  v
}
```

This will not work in S since `f` and `FUN` are not visible from within the frame of the innermost function.

It will work in R since the (anonymous) function defined in the call to `apply` inherits the environment in which it was defined when it is passed on.

**Note**

`sapply` and `tapply` provide some pre- and post-processing to `lapply` which can be expensive.

In practice this extra processing is often not needed and combinations of `lapply`, `unlist` and `split` can be used.

The following simple front-ends to `lapply` often suffice.

```r
> Sapply <- function(...) unlist(lapply(...))
> Tapply <- function(v, f, FUN, ...) unlist(lapply(split(v, f), FUN, ...))
```

Similarly `table` is a convenient interface function to `tabulate`.

Where speed is necessary `tabulate` should be used as directly as possible.

**Lazy evaluation**

Arguments are only evaluated if and when they are first used.

Evaluation is done in the parent frame.

Default values are evaluated in the local frame.

Example:

```r
> rm(a)
> f0(a <- 1, a) # OK since x eval. first
[1] 2
> rm(a)
> f1(a <- 1, a) # fails default not used
Error: Object "a" not found
```

```r
> rm(a)
> f0(a <- 1)   # OK since x eval. first
[1] 2
> rm(a)
> f1(a <- 1)   # OK since default first
[1] 2
```
The ‘whole object’ view

Iteration can be inefficient for interpreted interactive languages:

- The same code has to be re-interpreted at every step of the calculation,
- There can be large memory build-up to enable back-out of a calculation that fails.

The only way to avoid this is to make as much use as possible of compiled code. Two ways of doing this are

- Dynamic loading,
- Make as much use as possible of built-in operators.

This generally amounts to viewing a calculation as a single operation on an entire object rather than a sequence of operations on bits of it.

Dynamic loading

Use of compiled routines for key steps in the calculation:

Easy in S-PLUS 5.0,
Easy enough in S-PLUS 3.4 and R,
Tedious and awkward in S-PLUS 4.5 and exacting in its requirements.

Built-in operators

Perform standard tasks on entire objects. These include

- Indexing and low level functions such as match, order, sort, duplicated, unique, &c.
- The apply family, in particular lapply (in S-PLUS 4.x and earlier), together with unlist, split, &c.
- Matrix operators and allied functions, such as outer, %o%, tabulate, &c.

An example

Find the frequency of each possible value of \(ij - kl\) where \(i, j, k\) and \(l\) are independent non-negative single digit integers.

1. A loop solution:

```r
fr <- numeric(163) # initially zero
for(i in 0:9)
  for(j in 0:9)
    for(k in 0:9) {
      for(l in 0:9) {
        m <- i*j - k*l + 82
        fr[m] <- fr[m] + 1
      }
    }
```

2. A whole object solution:

```r
fr <- outer(0:9, 0:9)
fr <- tabulate(outer(fr, fr, "-")) + 82
```

Graph the results (in R):

```r
> postscript(file = "freq.eps")
> plot((-81):81, fr, type="h",

   xlab = expression(m[i]),
   ylab = expression(Fr(m[i])))
> dev.off()
```
An example of the use of `match`

Problem: Given a data frame, `mydata`, containing a factor `Foo`, find the sub-data frame `subdata` got by excluding all levels of `Foo` with fewer than `k` members.

```r
> dim(cuckoo)  # data on Cuculus canoris eggs
[1] 243  3
> names(cuckoo)
[1] "Length"  "Breadth"  "Host"
> attach(cuckoo)
> tHost <- table(Host)
> table(tHost)
     1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20
   1 1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1
   2  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1
   3 10  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1
   4  2  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1
   5  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1
   6  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1
   7  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1
   8  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1
   9  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1
  10 1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1
  11 1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1
  12 1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1
  13 1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1
  14 1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1
  15 1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1
  16 1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1
  17 1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1
  18 1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1
  19 1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1
  20 1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1
> rare <- names(tHost)[tHost < 9]
> keep <- is.na(match(Host, rare))
> Cuc <- cuckoo[keep,]
> Cuc$Host <- factor(as.character(Cuc$Host))
> Cuc$Host <- cuckoo$Host[keep, drop=T] # alt
> detach()
> attach(Cuc)
```

> table(table(Host))

```
  9  10  14  15  16  20  45
 1   1   1   1   1   1   1
```

An example of the use of `match`

Graph the results (in R):

```
-50  0  50
 0  100  200  300  400  500
mi
Fr(mi)
```

```r
> dim(cuckoo)  # data on Cuculus canoris eggs
[1] 243  3
> names(cuckoo)
[1] "Length"  "Breadth"  "Host"
> attach(cuckoo)
> tHost <- table(Host)
> table(tHost)
     1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20
   1 1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1
   2  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1
   3 10  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1
   4  2  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1
   5  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1
   6  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1
   7  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1
   8  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1
   9  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1
  10 1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1
  11 1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1
  12 1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1
  13 1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1
  14 1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1
  15 1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1
  16 1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1
  17 1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1
  18 1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1
  19 1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1
  20 1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1
> rare <- names(tHost)[tHost < 9]
> keep <- is.na(match(Host, rare))
> Cuc <- cuckoo[keep,]
> Cuc$Host <- factor(as.character(Cuc$Host))
> Cuc$Host <- cuckoo$Host[keep, drop=T] # alt
> detach()
> attach(Cuc)
```

> table(table(Host))

```
  9  10  14  15  16  20  45
 1   1   1   1   1   1   1
```