S Programming Techniques

Ross Ihaka

S Programming Workshop
University of Auckland
February 13–14, 2003

The S Language

- The S language has been developed since the late 1970s by John Chambers and his collaborators at Bell Laboratories.
- The language has been through major evolutionary changes, but has been relatively stable since the mid 1990s.
- The language combines ideas from a number of sources (e.g. *APL*, *Lisp*, *Awk*, ...) and provides an environment for quantitative computations.

S Implementations

- *S-PLUS* – a commercialised version of Chambers' work which is marketed by *Insightful*.
- *R* – an independent free-software implementation which was created at the University of Auckland and is now developed by an international collaboration of researchers.
- Each of these versions has advantages and problems.
- What I will talk about in this workshop will generally apply to both implementations. Where there are differences I will try to point them out.

References

- *The New S Language*. (The “Blue” Book.)
  R. Becker, J. Chambers and A. Wilks.

- *Statistical Models in S*. (The “White” Book.)
  J. Chambers and T. Hastie Eds.

- *Programming With Data*. (The “Green” Book.)
  J. Chambers.

- *Modern Applied Statistics with S-PLUS*.
  W. Venables and B. Ripley.

- *S Programming*.
  W. Venables and B. Ripley.

The Nature of Programming

The task of writing a program has two sub-tasks:

1. Describing precisely what is to be done.
2. Describing the data to be used.

These tasks can't be done separately. The choices made in either of the sub-tasks influence the choices made in the other.

algorithms + *data structures* = *programs*
– Niklaus Wirth

Data Structures

- S possesses a rich set of *self-describing* data structures.
- These structures describe the data to be manipulated by the language and also the language itself.
- The fact that the structures are self-describing means that there is no need for a use to declare the types of variables.
- It is possible that in future *optional* type declarations will be introduced to help compile the S language into efficient byte or machine code.

Atomic Data Structures

- The most basic data type in S is the *atomic vector*.
- Such vectors contain an indexed set of values which are all of the same type:
  - *logical*
  - *numeric*
  - *complex*
  - *character*
- The numeric type can be further broken down into *integer*, *single* and *double* types (but this is only important when making calls to C or Fortran.)

Creating Vectors

- Many S functions create vectors to hold the results they compute.
- There are also functions which can be used to create "empty" vectors.

```
> vector("numeric",10)
 [1] 0 0 0 0 0 0 0 0 0 0
> numeric(10)
 [1] 0 0 0 0 0 0 0 0 0 0

> vector("logical", 0)
logical(0)
```

Patterned Vectors

- The functions **`rep`** and **`seq`** can be used to create vectors containing patterns of values.

- Simple replication.

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

- More complex replication.

```
> rep(c("A", "B"), c(2, 3))
[1] "A" "A" "B" "B" "B"

> rep(c("A", "B"), each=3)
[1] "A" "A" "A" "B" "B" "B"
```

Vector Structures

- S retains the notion of *vector structures* from its earliest implementation.

- A vector structure is a vector with some additional information attached to it as an *attribute list*.

- Most uses of vector structures have been deprecated in favour of object-oriented alternatives.

- The major remaining use of vector structures is as the representation of arrays.

Arrays

- S regards an array as consisting of a vector containing the array’s elements together with a dimension (or **`dim`**) attribute.

- A vector can be given dimensions by using the functions **`array`** or **`matrix`**, or by directly attaching them with the **`dim`** function.

- The elements in the underlying vector correspond to the elements of the array with earlier subscripts moving faster.

Examples

- Direct array creation.

```
> x <- 1:10
> dim(x) <- c(2, 5)
> x
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    3    5    7    9
[2,]    2    4    6    8   10
```

- Array creation using **`matrix`**.

```
> x = matrix(1:10, nrow = 2)
```

Naming

- The elements of a vector can be given names by using the **`names`** function.

```
> x = c(10, 20)
> names(x) = c("First", "Second")
> x
 First Second
    10     20
```

- Array extents can be named by using the **`dimnames`** function or the **`dimnames`** argument to **`matrix`** or **`array`**. Extent names are given as a **`list`**, with each list element being a vector of names for the corresponding extent.

Example

```
> x <- array(1:8, dim=c(2,2,2))
> dimnames(x) <- list(c("A", "B"), NULL,
+                     c("X", "Y"))
> x

, , X
  [,1] [,2]
A    1    3
B    2    4

, , Y
  [,1] [,2]
A    5    7
B    6    8
```

Subsetting

- One of the most powerful features of S, is its ability to manipulate subsets of vectors and arrays.
- The S subsetting facility is derived from and extends that of *APL*.
- Subsetting is indicated by `[ ]`.

Subsetting With Positive Indexes

- A subscript consisting of a vector of positive integer values is taken to indicate a set of indexes to be extracted.

```
> x <- 1:10
> x[1:3]
[1] 1 2 3
```

- A subscript which is larger than the length of the vector being subsetted produces an **`NA`** in the returned value.

```
> x[9:11]
[1]  9 10 NA
```

Subsetting With Positive Indexes

- Subscripts which are zero are ignored and produce no corresponding values in the result.

```
> x[0:1]
[1] 1
```

- Subscripts which are `NA` produce an `NA` in the result.

```
> x[c(1, 2, NA)]
[1]  1  2 NA
```

Assignments With Positive Indexes

- Subset expressions can appear on the left side of an assignment. In this case the given subset is assigned the values on the right (recycling the values if necessary).

```
> x[1:3] <- 10
> x
 [1] 10 10 10  4  5  6  7  8  9 10
```

- If a zero or **`NA`** occurs as a subscript in this situation, it is ignored.

Subsetting With Negative Indexes

- A subscript consisting of a vector of negative integer values is taken to indicate the indexes which are not to be extracted.

  ```
  > x[-(1:3)]
  [1]  4  5  6  7  8  9 10
  ```

- Subscripts which are zero are ignored and produce no corresponding values in the result.

- **`NA`** subscripts are not allowed.

- Positive and negative subscripts cannot be mixed.

Assignments With Negative Indexes

- Negative subscripts can appear on the left side of an assignment. In this case the given subset is assigned the values on the right (recycling the values if necessary).

```
> x <- 1:10
> x[-(1:3)] <- 10
> x
 [1]  1  2  3 10 10 10 10 10 10 10
```

- Zero subscripts are ignored.

- **`NA`** subscripts are not permitted.

Subsetting By Logical Predicates

- Vector subsets can also be specified by a logical vector of trues and falses.

```
> x <- 1:10
> x[x > 5]
[1]  6  7  8  9 10
```

- **`NA`** values used as logical subscripts produce **`NA`** values in the output.

- The subscript vector can be shorter than the vector being subsetted. The subscripts are recycled in this case.

- The subscript vector can be longer than the vector being subsetted. Values selected beyond the end of the vector produce **`NA`**s.

Subsetting By Name

- If a vector has named elements, it is possible to extract subsets by specifying the names of the desired elements.

```
> x <- 1:10
> names(x) <- LETTERS[1:10]
> x[c("A","B")]
 A B
 1 2
```

- If several elements have the same name, only the first of them will be returned.
- Specifying a non-existent name produces an `NA` in the result.

Exercises

1. Determine (precisely) how S handles non-integer subscripts (e.g. `1.2`). How might this produce problems?

2. What value do the following expressions produce?

```
> x <- 1:10
> x[-11]
```

3. How could you choose all elements of a vector which have odd subscripts? Even subscripts?

4. How are complex subscripts treated?

Subsetting Arrays

- Rectangular subsets of arrays obey similar rules to those which apply to vectors.
- One point to note is that arrays can be treated as either matrices or vectors. This can be quite useful.

```
> x <- matrix(1:9, ncol = 3)
> x[x > 6]
[1] 7 8 9

> x[row(x) > col(x)] <- 0
> x
     [,1] [,2] [,3]
[1,]    1    4    7
[2,]    0    5    8
[3,]    0    0    9
```

Mode and Storage Mode

- The functions **`mode`** and **`storage.mode`** return information about the *types* of vectors.

```
> mode(1:10)
[1] "numeric"
> storage.mode(1:10)
[1] "integer"

> mode("a string")
[1] "character"

> mode(TRUE)
[1] "logical"
```

Automatic Type Coercion

- S will automatically coerce data to the appropriate type when this is necessary.

  ```
  > 1 + T
  [1] 2
  ```

  Here the logical value `T` has been coerced to the numeric value `1` so that addition can take place.

- Some common coercions are

  logical → numeric
  logical, numeric → complex
  logical, numeric, complex → character
  numeric, complex → logical

Type Coercion and NA Values

- Logical values can be coerced to any other atomic mode. Because of this, the constant **`NA`** has been made a logical value.

```
> mode(NA)
[1] "logical"
```

- When **`NA`** is used in an expression, the mode of the result is usually determined by the mode of the other operands.

```
> 1 + NA
[1] NA
> mode(1 + NA)
[1] "numeric"
```

An R / S-PLUS Difference

- S-PLUS does not have an **`NA`** indicator for character strings. It coerces **`NA`** values to the character string **`"NA"`**. There are potential problems with this approach.

  ```
  > is.na(as.character(NA))
  [1] F
  ```

- R does have a special **`NA`** value for character strings and so does differentiate **`NA`** and **`"NA"`**.

  ```
  > is.na(as.character(NA))
  [1] TRUE
  ```

Explicit Type-Coercion

- The function **`as.logical`**, **`as.integer`**, etc., return a copy of values passed to them, coerced to the specified type.

```
> as.numeric(c("1","10.5","text"))
[1]  1.0 10.5   NA
```

- **Warning**: These functions discard all labelling and dimensioning information.

```
> x <- 1:5
> names(x) <- LETTERS[1:5]
> as.character(x)
[1] "1" "2" "3" "4" "5"
```

Explicit Type-Coercion

- The functions **`mode`** and **`storage.mode`** (or more precisely **`mode<-`** and **`storage.mode<-`**) can be used to alter the storage mode of a variable.

```
> x <- 1:5
> names(x) <- LETTERS[1:5]
> x
 A B C D E
 1 2 3 4 5
> storage.mode(x) <- "character"
> x
   A   B   C   D   E
 "1" "2" "3" "4" "5"
```

- These functions preserve attributes like labelling and dimensioning.

Lists

- In addition to atomic vectors, S has a number of *recursive* data structures. The most important of these is the *list*.
- A list is a vector which can contain vectors and other lists as its elements.

```
> lst <- list(a = 1:3, b = "a list")
> lst
$a:
[1] 1 2 3

$b:
[1] "a list"
```

Subsetting and Lists

- Lists are useful as containers for grouping related things together (many S functions return lists as their values).
- Because lists are a recursive structure it is useful to have two ways of extracting subsets.
- The `[ ]` form of subsetting produces a sub-list of the list being subsetted.
- The `[[ ]]` form of subsetting can be used to extract a single element from a list.

List Subsetting Examples

- Using the `[ ]` operator to extract a sublist.

```
> lst[1]
$a:
[1] 1 2 3
```

- Using the `[[ ]]` operator to extract a list element.

```
> lst[[1]]
[1] 1 2 3
```

- As with vectors, indexing using logical expressions and names are also possible.

List Subsetting Syntactic Sugar

- The dollar operator provides a short-hand way of accessing list elements by name. The expression

  ```
  > lst[["a"]]
  ```

  is completely equivalent to the expression

  ```
  > lst$a
  ```

- The abbreviation is provided because accessing list elements by name is a very common operation in S.

Data Frames

- Data frames are a special S structure used to hold a set of related variables. They are the S representation for a statistical *data matrix*.

- Data frames can be treated like a matrix, and indexed with two subscripts. The first subscript refers to the observation, the second to the variable.

- In fact, this is an illusion maintained by the S object system. Data frames are really lists, and list subsetting can also be used on them.

Control-Flow

- S has a number of special control-flow structures which make it possible to express quite complex computations in the S language.
- Iteration is provided by the **`for`**, **`while`** and **`repeat`** statements.
- Conditional evaluation is provided by the **`if`** statement and the **`switch`** function.
- Of these capabilities, **`for`** and **`if`** are by far the most commonly used.

For Statements

- For statements have the basic form:

  `for(`_var_ **`in`** _vector_**`)`** {

  statements

  }

  The effect of this is to set the value of the variable *var* successively to each of the elements in *vector* and then evaluating *statements*.

- This looks similar to the *for* statement found in languages such as *C* and *C++*, but it is closer to the *foreach* statement of *Perl*.

Examples

- Summing the values in a vector (*C* style).

```
sum <- 0
for(i in 1:length(x)) {
  sum <- sum + x[i]
}
```

- Summing the values in a vector (*Perl* style).

```
sum <- 0
for(elt in x) {
  sum <- sum + elt
}
```

- The second of these is more efficient.

If Statements

- If statements have the basic form

  `if(` *test* **`)`** {
      statements
  } **`else`** {
      statements
  }

- If the first element of *test* is true, the first group of statements is executed, otherwise, the second group of statements is executed.

- The **`else`** clause is optional.

Examples

- Here is a typical use of **`if`**.

```
if (any(x < 0))
    stop("negative values encountered")
```

- Here is a choice between actions.

```
r <- if (all(x >= 0))
           sqrt(x) else
           sqrt(x + 0i)
```

  The layout here is important. The **`else`** must fall on the same line as the preceding statement (assuming the code above is not enclosed within **`{`** and **`}`**).

The Switch Function

- The `switch` function uses the value its first argument to determine which of its remaining arguments to evaluate and return. The first argument can be either an integer index, or a character string to be used in matching one of the following arguments.

  ```
  centre <- function(x, type) {
    switch(type,
           mean = mean(x),
           median = median(x),
           trimmed = mean(x, trim = .1))
  }
  ```

- Calling `centre` with `type=1` or `type="mean"` produces the same result.

Efficiency Issues

- S provides a full set of control-flow statements but they execute very slowly because S is (currently) an interpreted language.
- *R* is somewhat faster than *S-PLUS* at looping, but it is still two orders of magnitude slower than compiled *C* or *Fortran*.
- For time-critical applications, it can be useful to obtain measures of how fast a particular piece of code runs as a guide choosing a good computational method.
- The functions **`dos.time`**, **`unix.time`** (in *S-PLUS*) and **`system.time`** (in *R*) provide a way of timing how long it takes to evaluate a given expression.

Timing Experiments

- Timing experiments can be a good way of checking alternative ways of carrying out computations.

```
> sum <- 0
> x <- rnorm(10000)
> unix.time({s <- 0
+            for(i in 1:length(x))
+              s <- s + x[i]})
[1] 0.50 0.00 0.52 0.00 0.00

> unix.time({s <- 0
+            for(v in x)
+              s <- s + v})
[1] 0.19 0.00 0.19 0.00 0.00
```

The “Apply” Family

- Because looping tends to be slow in S, there is a family of functions which can be used to avoid explicit looping.

- The members of the family differ in the types of data structure they work on and in the degree to which they simplify the answers returned.

- The members are:

    - **`apply`** for *arrays*
    - **`tapply`** for *ragged arrays*
    - **`lapply`** and **`sapply`** for *lists*

Using Apply

- **`apply`** applies a function over the margins of an array.
- For example, the call:

  ```
  > apply(x, 2, mean)
  ```

  computes the column means of a matrix **`x`**, while

  ```
  > apply(x, 1, median)
  ```

  computes the row medians.
- **`apply`** is implemented in a way which avoids the overhead associated with explicit looping.

An Additive Table Decomposition

- Given data in a matrix `x`, this code carries out an *overall* + *row* + *column* decomposition.

```
overall <- mean(x)
row <- apply(x, 1, mean) - overall
col <- apply(x, 2, mean) - overall
res <- x - outer(row, col, "+") - overall
```

- The generalised outer product function **`outer`** is used here to produce a matrix, the same shape as `x`, containing the appropriate sums of row and column effects.
- Something similar can be used to produce a simple implementation of median polish.

Writing Functions

- Writing S functions provide a means of adding new functionality to the language.
- Functions that a user writes have the same status as those which are provided with S.
- Reading the functions provided with the S system provides a good way of learning how to write functions.
- If a user chooses, she/he can make modifications to the functions provided by the system and use the modified versions in preference to the system ones.

A Simple Function

- Here is function which squares its argument.

```
> square <- function(x) x * x

> square(10)
[1] 100
```

- Because the underlying arithmetic in S is vectorised, so is this function.

```
> square(1:4)
[1]  1  4  9 16
```

Composition of Functions

- Once a function is defined, it is possible to call it from other functions.

```
> sumsq <- function(x) sum(square(x))
> sumsq(1:10)
[1] 385
```

Example: Factorials

- Iteration.

```
fac <- function(n) {
  ans <- 1
  for(i in seq(n)) ans <- ans * i
  ans
}
```

- Recursion.

```
fac <- function(n)
if (n <= 0) 1 else n * fac(n - 1)
```

Example: Factorials

- Vectorised arithmetic.
  ```
  fac <- function(n) prod(seq(n))
  ```
- Using special functions.
  ```
  fac <- function(n) gamma(n+1)
  ```
- The version of `fac` based on the gamma function is one of the fastest and is the most flexible.

Exercise

Time each of the four factorial functions shown above. This is a little trickier than it sounds.

General Functions

- In general, as S function has the form:

  `function(` *arglist* **`)`** *body*

  where *arglist* is a comma-separated list of formal parameters and *body* is an S expression which computes the value of the function.

- Functions are evaluated by associating the values of the arguments with the names of the formal parameters and then evaluating the body of the function using these associations.

The Evaluation Process

If the function **`hypot`** defined by:

```
hypot <- function(a, b)
    sqrt(a^2 + b^2)
```

the S expression **`hypot(3, 4)`** is evaluated as follows.

- Temporarily create variables **`a`** and **`b`**, which have the values **`3`** and **`4`**.
- Use these variable definitions to evaluate the expression **`sqrt(a^2 +b^2)`** to obtain the value **`5`**.
- When the evaluation is complete remove the temporary definitions of **`a`** and **`b`**.

Optional Arguments

- S has a notion of default argument values.
- These make it possible for arguments to take on reasonable default values if no value was specified in a call to the function.
- In the following function, the second argument takes on the value **`0`** if no argument is specified.

```
sumsq <- function(x, about=0)
    sum((x - about)ˆ2)
```

- This means that the expressions **`sumsq(1:10, 0)`** and **`sumsq(1:10)`** will return the same value.

Optional Arguments

- The default values for arguments can be specified by an S expression involving the variables available inside the body of the function.

  ```
  sumsq <- function(x, about=mean(x))
    sum((x - about)^2)
  ```

- Recursive references within default arguments are not permitted. E.g. At least one argument must be provided to the following function.

  ```
  silly <- function(a=b, b=a) a + b
  ```

Argument Matching

- Because it is not necessary to specify all the arguments to S functions, it is important to be clear about which argument corresponds to which formal parameter of the function.
- The solution is to indicate which formal parameter is associated with an argument by providing a (partial) name for the argument.
- In the case of the **`sumsq`** function, the following are equivalent specifications.

```
sumsq(1:10, mean(1:10))
sumsq(1:10, about=mean(1:10))
sumsq(1:10, a=mean(1:10))
```

Lazy Evaluation

- S differs from many computer languages because the evaluation of function arguments is *lazy*.
- In other words, arguments are not actually evaluated until they are required.
- It can even be the case that arguments are *never* evaluated.

Example

- Here is a variation of the **`sumsq`** function.

```
sumsq <- function(x, about=mean(x)) {
  x <- x[!is.na(x)]
  sum((x - about)^2)
}
```

- This function first removes any **`NA`** values from **`x`** before computing its answer.

- Lazy evaluation means that the **`about`** value is computed from the cleaned **`x`**.

Exercises

1. Modify the **`sumsq`** function so that the removal of **`NA`** values is optional.

2. Write a new function which computes the deviations of the values in **`x`** about **`about`**. The value returned by the function should be “just like” **`x`**. How should missing values be handled?

Reading System Functions

- The built-in functions supplied with S form a valuable resource for learning about S programming.

- In many cases you may be surprised by the complexity of what appear to be trivial functions (try **`factorial`** or **`choose`**). Such complexity is usually introduced over time as a result of user feedback.

- Be warned that there can still be bugs in system functions.

Example: The Ifelse Function

```
> ifelse
function(test, yes, no)
{
  answer <- test
  test <- as.logical(test)
  n <- length(answer)
  if(length(na <- which.na(test)))
    test[na] <- F
  answer[test] <- rep(yes, length = n)[test]
  if(length(na))
    test[na] <- T
  answer[!test] <- rep(no, length = n)[!test]
  answer
}
```

Exercise

Look at these results from the S-PLUS **`ifelse`** function (the results from R are identical).

```
> ifelse("TRUE", 1, 0)
[1] "1"
> ifelse("FALSE", 1, 0)
[1] "0"
```

What is causing this problem and how can it be fixed?

Computing on the Language

- Because of argument evaluation is lazy, S allows programmers to get access to the unevaluated arguments.

- This is made possible by the **`substitute`** function.

```
> g <- function(x) substitute(x)
> g(x[1]+y*2)
x[1] + y * 2
```

- **`substitute`** is used conjunction with **`deparse`** to obtain a character string representation of an argument.

```
> g <- function(x) deparse(substitute(x))
> g(x[1]+y*2)
"x[1] + y * 2"
```

Computing on the Language

- The substitute function can take a call and substitute the symbolic representation of several arguments.

```
> g <- function(a, b) substitute(a+b)
> g(x*x, y*y)
x * x + y * y
```

- One particularly useful trick is to use the `...` argument in a substitute expression.

```
> g <- function(...) substitute(list(...))
> g(a=10, b=11)
list(a = 10, b = 11)
```

Manipulating Language Calls

- The objects returned by **`substitute`** are vectors of mode **`call`**.
- Calls are similar to lists in their behaviour and can be subscripted in the same way.
- The call **`a+b`** has three elements which are in order **`+`**, **`a`** and **`b`** (i.e. a lisp-like representation is used).
- The variable names appearing in calls are special S objects of mode **`name`**. They can be created from character strings with the function **`as.name`**.

Creating Calls

- Calls can be created with the function **`vector`**.

  ```
  > u = vector("call" 3)
  > u
  (,  )
  > u[[1]] <- as.name("f")
  > u[[2]] <- as.name("x")
  > u[[3]] <- as.name("y")
  > u
  f(x, y)
  ```

  but usually manipulations are carried out existing calls.

Evaluating Calls

- Given a call it can be *very* useful to evaluate that call. This is done with the **`eval`** function.

- **`eval`** takes the call, together with values for any variables present in the call and produces the value that this defines.

```
> u <- substitute(a+b)
> eval(u, list(a=10, b=20))
[1] 30
```

- A third argument to eval can be used to supply additional places which can be used to find values for variables.

Example: Transforming Data Frames

- Peter Dalgaard has written a small function to make it easy to manipulate the variables in a data frame.

- This function will transform and replace existing variables or create new ones to be added.

- Here is an example of applying this function to the S data set **`air`**, which gives information about air pollution.

```
> new.air <- transform(air,
+   new = -ozone,
+   temperature = (temperature-32)/1.8)
```

Example: The Transform Function

```
transform <- function (x, ...) {
    e <- eval(substitute(list(...)), x,
              sys.frame(sys.parent()))
    tags <- names(e)
    inx <- match(tags, names(x))
    matched <- !is.na(inx)
    if (any(matched)) {
        x[inx[matched]] <- e[matched]
        x <- data.frame(x)
    }
    if (!all(matched))
        data.frame(x, e[!matched])
    else x
}
```

Scoping

- We’ve seen that evaluation is the process of determining the value of a symbolic expression.
- In order for evaluation to take place, values must be determined for the variables in the expression.
- The scope of a variable is that portion of a program where that variable refers to the same value.
- The two dialects of S differ in their scoping rules.

Example

- In the following fragment:

```
x <- 10
y <- 20

f <- function(y) {
  x + y
}
```

There is global variable called x.

There is global variable called y and a local variable called y.

Scoping In S-PLUS

- The scoping rules in S-PLUS are simple.
- Variables are either local to the function they are defined in or they are global.
- The process of determining the value of a variable is as follows.
  1. Look for a local variable – if there is one, use its value.
  2. If there is no local variable, use the value of the global variable.
- There are some effects of these scoping rules which are counter-intuitive.

Scoping Problems

- The follow implementation of binomial coefficients does not work in S-PLUS.

```
choose <- function(n, k) {

  fac <- function(n)
           if(n <= 1) 1
           else n * fac(n - 1)

  fac(n) / (fac(k) * fac(n - k))

}
```

- Why does the function fail?

Consequences of S-PLUS Scoping

- The scoping rules of S-PLUS encourage the use of many globally defined functions, even when those functions are never called directly.
- This is because it is difficult to hide related helper functions inside "wrapper" functions.
- The use of this style produces *namespace clutter* and effects like the accidental masking of functions.
- Object-oriented programming extensions help a little.

Scoping in R

- R uses what is called static or lexical scoping (another term is block structure).
- Variables defined in outer blocks are visible inside inner blocks.
- This is a natural extension to the S-PLUS way of scoping.
- The hiding of helper functions within wrappers is encouraged.
- This promotes better software design and alleviates namespace clutter.
- It also has some more “interesting” consequences.

Example: Gaussian Likelihoods

```
mkNegLogLik <- function(x) {

  function(mu, sigma) {
    sum(sigma + 0.5 * ((x - mu)/sigma)^2)
  }

}

q <- mkNegLogLik(rnorm(100))
```