Lexical Scope and Statistical Computing

Robert GENTLEMAN and Ross IHAKA*

Abstract

The nature of statistical computing has changed a great deal in the past 15 years mainly due to the influence of programming environments such as S (Becker, Chambers, and Wilks, 1988) and Lisp-Stat (Tierney, 1990). Both these programming environments have languages for performing computations, data storage mechanisms and a graphical interface. Their usefulness in providing an interactive interface to data analysis is invaluable. In order to take full advantage of these programming environments statisticians must understand the differences between them. Ihaka and Gentleman (1996) introduced R, a version of S which uses a different scoping regimen. In some ways this makes R behave more like Lisp-Stat. In this paper we discuss the concept of scoping rules and show how lexical scope can enhance the functionality of a language.

Key Words: Statistical Computing; Function Closure; Lexical Scope; Random Number Generators.

1 Introduction

The nature of statistical computing has changed a great deal in the past 15 years mostly because of the influence of programming environments such as S (Becker, Chambers, and Wilks, 1988) and Lisp-Stat (Tierney, 1990). These environments have encouraged statisticians to become programmers so that they can have more effective control over the data analytic process. Statisticians can, therefore, benefit from considering programming paradigms and their implications. In this paper we will discuss scoping rules. These are the rules by which variables, ie. symbols, and values are associated. We will show some of the effect scoping has on evaluation in computer languages and demonstrate how some scoping rules may be beneficial to statisticians.

In every computer language there is a process for obtaining the value of a symbol when a computation is being carried out. If we ask the computer to evaluate `x+5` then there must be a mechanism or algorithm for determining the value represented by `x`. The set of rules used to obtain a value for `x` are called the scoping rules of the language. It is rather surprising how many different

*Senior lecturers in the Department of Statistics, University of Auckland, Private Bag 92019, Auckland, New Zealand.

ways this can be done and how large an effect the method used has on the behavior of the language.

We will call a collection of symbols and associated values an *environment*. Under certain scoping rules it is possible to create function closures. A *function closure* is a function together with an environment. In many programming languages, particularly functional languages, closures are the basic programming construct. By associating different environments with a function or by changing the values of some of the variables in an environment we alter the way the function evaluates. This mechanism is a useful abstraction for many of the programming tasks faced by statisticians.

By way of illustration we use code written in two languages with different scoping rules and compare the results. To this end our examples are coded in both R and S. The syntax and semantics of the two languages are very similar and the reader can easily compare the results. S and R are languages where functions are first class objects. That means that functions can be passed as arguments to functions and returned as values from functions. This ability is rarely used even though it is potentially very powerful.

Our examples include likelihood functions, probability functions, Bayes estimates and random number generators. We show how the scoping rules used affect these. We have chosen to use simple examples and simple data sets to make it easy for the reader to check on the details.

The remainder of the paper is organized as follows. In section 2 we extend our discussion of the scoping rules and discuss several general concepts needed to show how the scoping rules affect evaluation. In section 3 we provide examples of the advantages of function closures. Then in section 4 we comment on assignment and how being able to change the state of a closure can be useful. Finally in section 5 we summarize our findings.

2 Evaluation and Scoping Rules

Most computer languages have values, symbols (or variables), and functions. Often the same symbol appears in several different contexts within a computer program. For example, the symbol `x` can be a global variable, a local variable in a function, and a formal parameter to another function simultaneously. An important part of the computational process is associating symbols with values. The rules that are used to do this are called scoping rules.

The symbols which occur in the body of a function can be divided into two groups; bound variables and free variables. The formal parameters of a function are those occurring in the argument list of the function. Any variable in the body of the function that matches one of the formal parameters is bound to that formal parameter. Local variables, those whose values are determined by the evaluation of expressions in the body of the functions, are bound to the values of the expressions. All other variables are called free variables. At the time that the function is evaluated a binding must be obtained for all variables. It is an error to access an unbound variable in every computer language.

The process of function evaluation is similar across programming languages. The body of a function is the set of statements that will be executed (sequentially) when the function is invoked. At this time a new environment, the evaluation environment, is created and in this environment the function's formal arguments (symbols) are bound to the actual arguments (values). Bindings for free variables are resolved in the environments specified by the scoping rules. Users familiar with the inner workings of S will realize that S's frames are an implementation of environments.

Both R and S are interactive languages. The user invokes the language in some fashion and is then presented with a *prompt*. Usually at this point assignments and evaluation are with respect to an environment which we shall call the *top–level* environment. Behavior of the language in this top–level environment is generally slightly different from its overall behavior. For example, while purely functional languages do not allow assignment they do allow it at top–level; otherwise the user could not define new functions. In R the behavior of functions defined at top–level is different from the behavior of functions defined inside of other functions. In S the behavior of functions is essentially the same no matter where they were defined.

Consider the following function definition.

```
f <-
  function(x) {
    y <- 2 * x
    print(x)
    print(y)
    print(z)
  }
```

In this function `x` is a formal parameter, `y` is a local variable and `z` is a free variable.

Now consider the functions defined below.

```
fun1 <-
  function(x) x + y

fun2 <-
  function() {
    y <- 20
    function(x) x + y
  }
```

Let's first consider `fun1`. It was created at top-level and will behave the same way in both R and S. If `y` has not been defined when `fun1` is invoked an error will be signaled. The value returned by `fun1` will depend on the current value of `y` in the top-level environment.

```
> fun1(3)
Error: y not found
```

```
> y <- 12
> fun1(3)
[1] 15
> y <- 13
> fun1(3)
[1] 16
```

We now turn to the somewhat more interesting case of `fun2`. Notice that it returns a function which is essentially the same as `fun1`. When `fun2` is invoked it returns a function. The behavior of this returned function is different in R and S because of the different scoping rules. In R the scoping rules state that the free variables in a function are resolved in the environment that was active at the time the function was created. In S the scoping rules state that the free variables are resolved at top-level. Let's make the assignment

```
R> fun3 <- fun2()
```

and then examine the values in `fun1` and `fun3`.

```
R> fun1
function (x)
x + y

R> fun3
function (x)
x + y
<environment: 0x14b488>

R> y <- 2
R> fun1(5)
[1] 7
R> fun3(5)
[1] 25
```

In R, when `fun2` is invoked, a new environment is created. Within this environment, `y` is assigned the value 20. Since the function which is ultimately assigned to `fun2` is also defined within this environment, it has access to the variables in the environment. In particular, the value for `y` is found in this environment. In S, `fun3` finds its variable bindings in the top-level environment and so it will evaluate in the same way that `fun1` does.

Notice that R functions that have environments other than the top-level environment associated with them have that environment explicitly printed. This is a visual aid to remind the user that the behavior of such a function may be different from a function with the same body but a different associated environment.

2.1 Scoping Rules

We now discuss scoping rules more generally. For the purposes of the present discussion we identify four types of scoping rules: trivial scope, dynamic scope, static scope and lexical scope. Under trivial scope free variables are not allowed. Under dynamic scope the value associated with a free variable is determined by searching back up the sequence of calling functions and using the most recently defined value associated with that symbol. Under static scope the values of the free variables are determined by a set of global variables. This is the kind of scoping used in C and S. When the values of the free variables are defined by the bindings that were in effect at the time the function was created then the language is lexically scoped. This is the kind of scoping used in R and Scheme.

While the usual definition of static or lexical scope in computer science is that the variable bindings can be determined from a printed copy of the code this definition is not specific enough. Computer scientists tend not to differentiate as finely because their concerns are different. However, if we consider some commonly used computer languages that satisfy this definition of lexical scope we see that there can be large differences between them. For example, in C you are not allowed to nest function definitions so only global variables can be bound to free variables. In S nested function definitions are permitted but S does not resolve the free variable bindings within the calling function instead it resolves them globally. In both R and Scheme the free variable bindings are resolved by first looking in the environment in which the function was created. Since we want to contrast R and S with respect to this difference we have refined the scoping definitions to our purpose.

The following code exemplifies the difference between static scope and lexical scope, as we have defined them.

```
boot <-
  function(x, statistic, bootreps) {
    n <- length(x)
    sapply(1:bootreps,
           function(dummy)
               statistic(sample(x, n, replace = TRUE)))
  }
```

This is a very succinct function that will bootstrap any univariate function. It, however, relies on lexical scope and hence will evaluate properly in R but not in S. In the call to `sapply` there is an anonymous function; that is, a function with no name. When `sapply` is invoked it will evaluate its arguments in the environment of the calling function. The semantics of both R and S are that the anonymous function is defined in the evaluation environment of `boot`. Now, when the anonymous function is evaluated we encounter the symbols `statistic`, `x`, and `n` which are free variables.

Thus in R, because it is lexically scoped, the local and appropriate values will be used. In S the top-level environment will be searched for these symbols and it is likely, but not certain, that an error will be signaled. It is relatively

easy to construct a version that will work correctly in S.

```
boot2 <-
  function(x, statistic, bootreps) {
    n <- length(x)
    sapply(1:bootreps,
           function(dummy, data, stat, num)
               stat(sample(data, num, replace = T)),
                   x, statistic, n)
  }
```

In `boot2` there are five arguments passed to `sapply`. The last three arguments; `x`, `statistic`, and `n`, are passed to `sapply` explicitly and the semantics of `sapply` ensure that they are then passed on to the anonymous function. The names of the formal parameters in the anonymous function must match those used in the body of the anonymous function so that there are no free variables in the anonymous function.

2.2 Programming Styles

Looking at languages that have lexical scope one finds that a number of them are functional languages, eg. ML, Miranda and Haskell. Both Lisp and Scheme have lexical scope but are not generally considered to be truly functional languages.

Functional programming is a style of programming where function calls are the primary programming construct. Functional languages tend not to allow assignment (except at top-level). Languages that use assignment, such as C and FORTRAN are called imperative languages.

It should be noted, however, that most functional languages that have come into popular use added some mechanism for assignment, eg Lisp and ML. It is quite difficult to write efficient programs without side–effects. However, good programmers still rely mainly on function closures.

There are stylistic differences between functional languages and imperative languages both in terms of the programs that are written and in terms of the data structures used. In imperative languages arrays are a common and useful data type; this is less true in functional languages. The problem being that it is difficult to directly access arbitrary array elements without some form of assignment. Functional languages tend to rely more on list structures and explicit data types.

Functional languages have gained popularity through their close relation to the theory of computing and to the underlying mathematics that is being implemented. They have also been closely involved in the study of artificial intelligence. Further, in a purely functional language the order in which statements are executed does not affect the final result. This feature has engendered some interest in highly parallel computing.

Consider the two functions below that implement the simple process of summing up the elements of a vector.

```
sum1 <-
  function(x) {
    sum <- 0
    for (i in 1:length(x))
      sum <- sum + x[i]
    sum
  }

sum2 <-
  function(x) {
    if (length(x) == 1)
      x
    else
      x[1] + sum2(x[-1])
  }
```

Both functions will work in either S or R and provide the same results. However, because neither R nor S are properly tail recursive `sum2` can only be used on small examples. The first is an imperative version of sum while the second is a functional version and in `sum2` no assignments are made. Many procedures can be written in either form.

Functional programming provides a different paradigm and a different abstraction. In some cases this abstraction is very useful and in others it is less so. Very little S code has been programmed in a functional style. In part this is due to the fact that it can be done easily in an imperative fashion. In part it is due to the fact that many programmers simply use S to prototype algorithms that they intend to implement in C which is imperative. And, in part, we argue that it is due to the lack of language support in the form of lexical scope.

2.3 Problems

There are other implications of lexical scope and environments. The environments for function evaluation must be maintained, that is, they must be persistent objects. When a function is returned as a value, it must have access to the environment in which it was created. Having persistent environments raises many issues some of which are addressed in Ihaka and Gentleman (1996).

Under static scope environments can be transient, that is they can disappear once the function has returned a value. We can be sure that this environment will never be needed again.

If the language allows the programmer to change the values associated with the variables in the environment of a lexical closure then further difficulties arise. In particular it becomes very difficult to store the functions as files and thus the program, when active, must have all objects reside in memory. The problem arises because several functions may share a single environment. Changes made by one of the functions should be reflected in the behavior of the others. However, in most implementations (and in that of R) environments do not keep

track of the functions associated with them, rather, the functions keep track of their environment.

3 Examples

In this section we will give some examples which indicate the usefulness of function closures. Function closures are function bodies bound together with values for the free variables. In most lexically scoped languages this binding is achieved through the associated environment but the binding can be achieved in many different ways. An example of how to achieve this in S is given in an Appendix.

Function closures allow you to encapsulate the data. When considering likelihoods and Bayes estimates we will see how this allows us to ensure that the correct data and probability models are always being used. There is a strong relationship between function closures and object–oriented programming. In some sense you can argue that a function closure is an instance of an object. While one can use lexical scope to build an object system that is probably not the best way to do it.

A second reason to use function closures is that they allow you to keep the code closer to the underlying mathematics. This should make it easier to write and understand the code. We demonstrate this feature in both the Bayesian estimation and numerical integration sections below.

3.1 Likelihoods

Suppose we observe a sample of size n which we believed to be from the Exponential density, $f(x) = \mu \exp(-x\mu)$ where both μ and x must be positive. In order to estimate μ one can use the likelihood principle. The log likelihood function for a sample, $(x_1, \ldots, x_n)$, from an Exponential(μ) distribution is $l(\mu) = n \log(\mu) - \mu \sum(x_i)$. The maximum likelihood estimate is the value of μ which maximizes this function.

Likelihood functions are commonly used in both research and teaching. It would be convenient to have some means of creating a likelihood function. This means that we want to have some function, which we will call a *creator*, that we pass data to and get back a likelihood function. We will call this function the *returned function*. This likelihood function would then take as arguments values of the parameter (μ in the case above) and return the likelihood at that point for the data that was supplied to the creator. To do so the returned function needs to have access to the values of the data that were passed to the creator.

If the programming language has lexical scope there is no problem because the returned function is created inside the creator and hence has access to all variable definitions that were in effect at the time that it was created.

In the following example `Rmlfun` is a creator. It sets up several local variables which will be needed by the likelihood function and whose values depend on the data supplied. Then the likelihood function is created and returned. The

environment associated with the returned function is the environment that was created by the invocation of `Rmlfun` which means that the variables `n` and `sumx` will have bindings in that environment.

```
Rmlfun <-
  function(x) {
    sumx <- sum(x)
    n <- length(x)
    function(mu)
      n * log(mu) - mu * sumx
  }
```

Subsequent evaluation of `Rmlfun` causes the creation of a new environment with bindings to `n` and `sumx` which depend on the arguments supplied to `Rmlfun`. This environment does not interfere in any way with any environment created by previous invocations of `Rmlfun`.

```
R> efun <- Rmlfun(1:10)   # efun is a function!
R> efun(3)
[1] -154.0139

R> efun2 <- Rmlfun(20:30)
R> efun2(3)
[1] -812.9153

R> efun(3)                # nothing has changed for efun
[1] -154.0139
```

This example does not work in S. In S `Rmlfun` returns a function with the correct body but when `efun` is evaluated an error occurs because the environment in which `n` and `sumx` were bound evaporated when `Rmlfun` returned its value. They are free variables in `efun` and unless there are global variables with the same names an error will be signaled. However, the function MC, given in an Appendix, can be used to provide similar functionality in S.

```
S> Smlfun <-
   function(x) {
     sumx <- sum(x)
     n <- length(x)
     rfun <- function(mu)
       n * log(mu) - mu * sumx
     MC(rfun, list(sumx = sumx, n = n))
   }
S> efun <- Smlfun(1:10)
S> efun(3)
[1] -154.0139
```

3.2 Function Optimization

In this section we will extend the example given above slightly to indicate one of the areas where lexical scope can provide great simplifications of the code. We will use simple examples and naive implementations of them so that the points regarding lexical scope are not lost amid the complexity of function optimization. For the reader this can be paraphrased as, do not use these methods, they are only examples and there are better ways to solve these problems. However, even the better solutions benefit from lexical scope so we lose nothing and gain simplicity for our purpose.

A somewhat simple method for finding the zero of an arbitrary function, $f(x)$, of one variable is Newton's method. If a is an initial guess as to the value of x such that $f(x) = 0$ then an improved guess is obtained via

$$x_{new} = a - f(a)/f'(a). \tag{1}$$

This process can then be iterated until a value of x_{new} is obtained such that $f(x_{new})$ is sufficiently close to zero.

Optimization problems frequently arise in all areas of statistics and one common problem is in finding the maximum likelihood estimate. In many cases the likelihood is convex in the parameters and hence has a single maximum. In that case the maximum likelihood estimate can be obtained by finding the place where the score function (the first derivative of the likelihood) is zero.

In most of the problems that arise in statistics the objective function depends not only on the parameter that we are optimizing over but on many other variables (usually the data). Because of that one can never really use the simple form of Equation 1. In most implementations there must be some means of passing the extra information to the optimizer. This generally complicates the code and often results in code that is not easily extensible.

However, when the language has lexical scope, the simple form can be used for many problems. Consider the slightly extended likelihood function generator given below.

```
Rmklike <-
  function(data) {
    n <- length(data)
    sumx <- sum(data)
    lfun <- function(mu) n * log(mu) - mu * sumx
    score <- function(mu) n / mu - sumx
    d2 <- function(mu) -n / mu^2
    list(lfun = lfun, score = score, d2 = d2)
  }
```

In this function we return not only the likelihood function but also functions to obtain the score and the second derivative.

The optimizer can then be written in the following way,

```
newton <-
```

```
function(lfun, est, tol = 1e-7, niter = 500) {
  cscore <- lfun$score(est)
  if (abs(cscore) < tol)
    return(est)
  for (i in 1:niter) {
    new <- est - cscore / lfun$d2(est)
    cscore <- lfun$score(new)
    if (abs(cscore) < tol)
      return(new)
    est <- new
  }
  stop("exceeded allowed number of iterations")
}
```

The function `newton` can be used to find the zero of any univariate function provided that the function passed in adheres to the protocol that the zero function is stored in the list as `score` and its derivative is stored in the list as `d2`.

This functionality can be used in S as well. One simply modifies `Rmklike` to use MC to create each of the functions that are to be returned. It should be noted that this approach works because none of the returned functions alter the values of `n` or `sumx`. If we were working on a problem where values of some constants change between function calls then lexical closures can be used in R but the method described for S will not work. The difference is that in S the returned functions will each have their own copies of the state variables while in R they will share one set.

3.3 Probabilities and Related Concepts

Suppose we want to take a set of data and make an empirical cumulative distribution function from it. One representation of an ecdf is as $\Pr(X \leq t)$. A functional version of the ecdf accepts as input a real number t and returns a value between 0 and 1.

```
mkecdf <-
  function(x) {
    n <- length(x)
    function(t)
      sum(x <= t) / n
  }
```

The local variables in the returned function are bound to the values they held at the time the returned function was created. These examples are useful in the teaching of statistics and probability since they function in a manner consistent with our perceptual model and hence allow for easy abstraction to related problems.

In fact these functions and concepts are of more general use. Gentleman and Crowley (1989) examine their use for smoothing data. Consider a scatter plot

with a response variable plotted in the y direction and a covariate plotted in the x direction. A *scatterplot smoother* is a line added to the plot which represents an estimate of the mean response conditional upon the value of the covariate. Adding a scatterplot smoother enhances our perception of the dependence between the response and the covariate. An example of a smoother is a running mean which is given by

$$\hat{m}_x = \frac{1}{k} \sum_{N_x} x_i$$

where N_x is the list of indices of the k nearest neighbors of the point x. This scatter plot smoother shows how the mean response changes as the value of the covariate changes.

Many smoothers may be written as functionals of the estimated conditional distribution function. For example, the running mean can be written as

$$\hat{m}_z = \int u d\hat{F}_z(u \mid k)$$

where $\hat{F}_z(\cdot \mid k)$ is the empirical distribution function based on the k nearest neighbors of z. This latter representation of the running mean can easily be generalized to use estimates of F other than the empirical distribution function. It also lends itself to easily being adapted to providing estimates other than a running mean (eg. running quantiles) by simply changing the integrand.

This formulation is also useful computationally. Take the entire data set, break it down into neighborhoods and estimate a conditional distribution in each. Return a vector or a list of all the conditional distributions. Now writing a function which takes a conditional distribution function as input and returns the desired quantity yields the appropriate smoother with little additional computational effort.

3.4 Numerical Integration

To see how a functional approach puts you closer to the mathematics consider numerical integration. To integrate a function in one dimension the midpoint rule can be easily used. This rules states that

$$\int_a^b f(x)dx \approx \frac{b-a}{n} \sum_{i=1}^{n} f(x_i),$$

where $x_i = a + h/2, a + 3h/2, \ldots, b - h/2$ and $h = (b-a)/n$. This translates into the program,

```
midpoint <-
  function(f, a, b, n = 100) {
    h <- (b - a) / n
    (b-a) * mean(sapply(seq(a + h / 2, b - h / 2, len = n),f))
  }
```

To compute a bivariate integral,

$$\int_{a_1}^{b_1} \int_{a_2}^{b_2} f(x,y)dxdy$$

mathematically we think of this as first integrating with respect to x and then with respect to y. Programmatically we would like to duplicate this, the following code segment does just that.

```
integrate <-
  function(f, a, b, n = 100, rule = midpoint) {
    g <- function(y) {
      fx <- function(x) { f(x, y) }
      rule(fx, a[2], b[2], n)
    }
    rule(g, a[1], b[1])
  }
```

Note that the function `integrate` relies heavily on lexical scope. When `fx` is evaluated in `g` it must find the correct value for `y`. In S it will not. This is a very natural implementation of the iterated integral, we have not had to make provision for passing extra parameters and we have suppressed arguments that do not change.

3.5 Bayesian Estimation

These ideas can be further extended to some simple examples in Bayesian statistics. In a Bayesian setting inference about the unknown parameter(s), θ, is based on both the observed data and a *prior* belief about possible values of the unknown parameter(s). This prior belief is generally expressed in terms of a probability distribution on the possible values of the parameter(s). This distribution is termed the prior distribution and we will label it $\pi(\theta)$. In a Bayesian setting one uses the prior distribution, the assumed form of the distribution of the data and Bayes Theorem to obtain a posterior distribution for the parameter given the data. Inference about the parameter(s) is then based on this posterior distribution.

Since the posterior, prior and conditional distribution of the data given the parameters are all functions this is a natural candidate for functional programming. From a pedagogical point of view Bayesian statistics was often given reduced emphasis due to the computational difficulties inherent in the calculation of the posterior distribution. Recently Markov Chain Monte Carlo methods have overcome this but simple application of functional programming methods can also yield useful results.

Mathematically, we begin with a prior distribution on θ called $\pi(\cdot)$. The conditional distribution of the data given θ is denoted $f(x|\theta)$. Under the Bayesian assumptions x and θ have a joint distribution, $f(x,\theta)$ and the marginal distri-

bution of x can be computed as,

$$f(x) = \int_{\Theta} \pi(\theta) f(x|\theta) d\theta,$$

where $\theta \in \Theta$. Then the posterior distribution is computed as,

$$f(\theta|x) = \pi(\theta) \frac{f(x|\theta)}{f(x)}.$$

To see how we can implement this in a functional way we first make the setting more concrete. Let's suppose that we have x successes in n Bernoulli trials and that we wish to estimate θ which is the probability of a success. It is well–known that if θ is assumed to have a Uniform prior distribution then the posterior distribution is,

$$f(\theta|x) = \frac{\theta^x (1-\theta)^{n-x}}{B(x+1, n-x+1)},$$

where $B()$ is the Beta function. Hence we see that the posterior distribution is a Beta distribution. If one wants to choose a different prior the mathematics become somewhat more difficult.

A computational solution to this problem is given by the functions below.

```
piunif <-
  function(x)
  ifelse (0 <= x & x <= 1, 1, 0)

bayesunif <-
  function(x, n, prior = piunif) {
    cprob <- function(theta) dbinom(x, n, theta)
    integrand <- function(theta) prior(theta) * cprob(theta)
    px <- midpoint(integrand, 0, 1)
    function(theta)
      prior(theta) * cprob(theta) / px
}
```

The first function, `piunif` is simply a functional form of the Uniform density. The second function, `bayesunif`, returns as a function the posterior density of θ given the data. This can be evaluated for any value of θ. In contrast to the theoretical approach the use of any other prior is straightforward. One simply writes the appropriate substitute for `piunif` and calls `bayesunif` supplying it as the prior.

Examining `bayesunif` we see that the functional programming style has been used to keep us closer to the mathematics. The function `cprob` is the conditional distribution of x given θ, the function `integrand` is the integrand needed to evaluate the marginal distribution of x, and we reuse `midpoint` defined previously, to compute the constant $f(x)$; remember that the data are fixed. Finally, the posterior is returned.

This can be plotted, it can be used to find $F(\theta|x)$, again using `midpoint`. One can generate observations from this density using rejection algorithms and compute confidence intervals for θ. It can also be used to find the mean of the posterior which can be used as an estimate of θ.

4 Mutable State

One could describe the association of an environment with a function as giving that function state. We next explore the effects of being able to change that local state information programmatically.

Both S and R are at heart imperative languages. Part of the problem with them is that the assignment operator, `<-` is *overloaded*. The form of the assignment expression is generally, *left–hand–side* `<-` *right–hand–side*. Where the *left–hand–side*, when evaluated, yields a symbol while the *right–hand–side* yields a value.

An operator is said to be *overloaded* if it performs more than one task. In R and S the assignment operator performs two tasks. If the left side is a symbol that already exists in the current environment then the value of that symbol is changed to the right hand side. If the symbol does not exist in the current environment then it is created and the value of the right hand side is assigned to it.

Most programming languages separate these two tasks. Overloading them can make programming easier but can also create serious problems in writing compilers and in understanding the code. Consider the following unusual, but legal code segment,

```
foo <-
  function(x) {
    if (x < 10)
      y <- 12
    x + y
  }
```

Now, one cannot determine whether `y` is a local variable or a global variable until the function is evaluated. If `x<10` then `y` in `x+y` is a local variable and otherwise it is a global variable.

It can be argued that this overloading of `<-` forces us to have a second type of assignment, `<<-`. In S the semantics of this operator are as follows. In the top-level environment the symbol on the left hand side of the operator is assigned the value given by the right hand side. If necessary the variable is created, otherwise its value is changed. In R, the semantics are slightly different. Starting with the current environment and searching up through the enclosing environments the symbol on the left hand side is sought. If it is found then the value of the right hand side is associated with it. If it is not found once the top-level environment has been searched then the symbol on the left hand side is assigned to it.

4.1 Random Number Generators

Pseudo-random number generation is an important part of statistical computing. Two common uses are simulation and bootstrapping. Pseudo-random number generators require a seed and for a given seed they produce a sequence of numbers that, hopefully, has all the properties of a sequence of truly random numbers. The difference between pseudo-random number generators and random number generators is that given the same seed the pseudo-random number generator will reproduce the same sequence. Since we only discuss pseudo-random numbers we will drop the pseudo prefix.

With most random number generators the seed is updated for each random number generated and in order to get the next number in the sequence you need the last value of the seed. This implies that in order to implement a simulation all routines which handle the random numbers need to have an extra parameter passed to them, the seed. This is often not practical and instead the solution of making the seed a global variable is taken. Having the seed be a global variable can have some undesirable effects on the simulation since it makes the seed accessible to every function and hence increases the likelihood of some function inadvertently changing the seed. There is also a chance that the seed could be reset to the original starting value. These sorts of occurrences can and usually do invalidate the results of the simulation and hence should be guarded against. Unfortunately there is seldom ever any evidence that such an event has occurred and the results are accepted as if they were valid.

These problems can be overcome by the use of lexical scope. Lexical scope allows an instance of the seed to be bound to a particular random number generator and to be inaccessible to any other random number generator. Thus, it cannot be altered inadvertently. The binding mechanism is sufficiently general to allow the user to set the seed, query the seed and generate the next number in the sequence through a collection of returned functions. But, the seed is accessible only through these functions.

An example will clarify the situation. Consider the function: `make.random`.

```
make.random <-
  function(seed)
    list(rand =
           function() {
             seed <<- (9 * seed + 5) %% 1024
             seed },
         setseed =
           function(nseed) {
             seed <<- nseed },
         getseed =
             function() {
               seed
             }
         )
```

The function `make.random` is a function that has one argument, `seed` and it returns a list containing three functions: one to generate random numbers, `rand`, one to set the seed of the random number generator, `setseed`, and one to get the current value of the seed, `getseed`.

When `make.random` is invoked an environment is created and in this environment the symbol `seed` is bound to the value supplied as an argument. Next the list of functions is created. Each of these functions has the environment with a binding for `seed` as its associated environment and hence can access the current value associated with `seed`.

```
R> rand <- make.random(1)
R> rand$rand()
[1] 14
R> rand$rand()
[1] 131
R> rand$getseed()
[1] 131
```

Now `rand$rand` is a function and evaluating it produces a sequence of random numbers. Again it is essential that in `rand$rand` the special assignment operator `<<-` is used since this ensures that rather than creating a new variable called `seed` the variable in the associated environment has a new value bound to it.

Several versions of `rand` may exist simultaneously and as they will all be the result of separate invocations of `make.random` their associated environments will be distinct and each will have its own protected version of `seed`.

```
> rand1 <- make.random(1)
> rand2 <- make.random(101)
> rand1$rand()
[1] 14
> rand2$rand()
[1] 914
> rand1$rand()
[1] 131
```

We have created two random number generators, with different seeds, that do not interfere with each other. Even with the call to `rand2$rand` between the two calls to `rand1$rand` we see that we get the same sequence as above for `rand1$rand`.

It is important to emphasize that this method ensures repeatability of the simulation. There is no chance that the seed is corrupted by other outside functions. But there is the possibility here (as with all other simulations) that two different seeds generate sequences that have substantial overlap.

5 Concluding Remarks

In this paper we have argued the usefulness of lexical scoping and its consequences in several situations that are important to statisticians. The use of function closures being one of the more beneficial. These make it much easier for statisticians to program complicated algorithms without having to delve too deeply into the basic functioning of the language. In some ways one could argue that this is one of the true strengths of S, that you can write programs without having to worry about memory allocation or type-checking of variables. We believe that lexical scope has large advantages and have incorporated it in R.

S has become a popular tool for statistical programming and research. It has many strengths, among these the fact that it is a functional language with functions as first class values. This means that functions can be passed to other functions as arguments and they can be returned as values. We argue that this latter property is very important but has been neglected mainly because of the scoping rules used in S. Most modern functional languages use lexical scope. A major reason for this is that these languages rely on the lambda calculus for their theoretical underpinnings and the lambda calculus uses lexical scope.

One place where lexical scope has not been used but could potentially improve the situation is in the modeling language used in both S and R. In the current implementation the formula, `y~x` is simply a quoting mechanism. The symbols `y` and `x` are stored and when the formula is evaluated they are treated as symbols and the standard mechanism for matching them takes over. There is no guarantee that they will be matched to the correct values. This problem often catches the unwary. If formulae captured the current environment through lexical scope then the modeling process would be much safer.

Acknowledgments

The authors would like to thank an Associate Editor and two referees for their careful reading of a prior version of this manuscript. In particular for their helpful suggestions that improved our examples and in some cases provided us with better ones. We would also like to thank John Chambers, Mike Meyer and Duncan Murdoch for their helpful comments on a draft of this paper.

6 References

Abelson, H., Sussman, G. J. and Sussman J. (1985). *Structure and Interpretation of Computer Programs.* MIT Press.

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988). *The new S Language: A programming environment for data analysis and graphics.* Wadsworth & Brooks/Cole.

Gentleman, R. and Crowley, J. (1990). Smoothing censored data. Technical Report 90-13, Department of Statistics and Actuarial Science, University of

Waterloo.

Ihaka, R. and Gentleman, R. (1996). R: A language for data analysis and graphics. *The Journal of Computational and Graphical Statistics,* **5**, 299–314.

McCarthy, John (1960). Recursive functions of symbolic expressions and their computation by machine, part 1. *Communications of the ACM,* **3**,185–95.

Tierney, L. (1990). *Lisp-Stat: An Object Oriented Environment for Statistical Computing and Data Analysis.* John Wiley and Sons.

A Lexical Scope in S

The following method of creating function closures in S closely follows suggestions made by Luke Tierney on various mailing lists. A function closure can be created in S by taking a base function and extending its formal arguments to include the new bindings. The function should be written so that the body contains free variables that will later have bindings supplied. An implementation of this is given by the function `MC` below. It should be noted that this function relies very heavily on the implementation of functions in S.

```
MC <-
  function(f, env = NULL) {
    env <- as.list(env)
    if (mode(f) != "function")
      stop(paste("not a function:", f))
    if (length(env)>0 && any(names(env) == ""))
      stop(paste("all arguments are not named:", env))
    fargs <- if (length(f) > 1) f[1:length(f) -  1]
             else NULL
    fargs <- c(fargs, env)
    if (any(duplicated(names(fargs))))
      stop(paste("duplicated arguments:",
                 paste(names(fargs),
                       collapse=", ")))
    fbody <- f[length(f)]
    cf <- c(fargs, fbody)
    mode(cf) <- "function"
    cf
  }
```