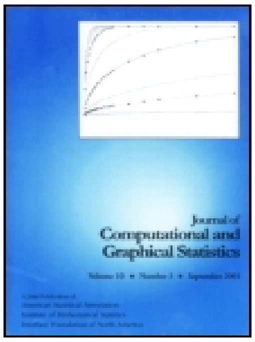

Taylor & Francis
Taylor & Francis Group

Journal of Computational and Graphical Statistics

ISSN: 1061-8600 (Print) 1537-2715 (Online) Journal homepage: https://www.tandfonline.com/loi/ucgs20

R: A Language for Data Analysis and Graphics

Ross Ihaka & Robert Gentleman

R: A Language for Data Analysis and Graphics

Ross Ihaka and Robert Gentleman

In this article we discuss our experience designing and implementing a statistical computing language. In developing this new language, we sought to combine what we felt were useful features from two existing computer languages. We feel that the new language provides advantages in the areas of portability, computational efficiency, memory management, and scoping.

Key Words: Computer language; Statistical computing.

1. INTRODUCTION

This article discusses some issues involved in the design and implementation of a computer language for statistical data analysis. Our experience with these issues occurred while developing such a language. The work has been heavily influenced by two existing languages—Becker, Chambers, and Wilks' S (1985) and Steel and Sussman's Scheme (1975). We felt that there were strong points in each of these languages and that it would be interesting to see if the strengths could be combined. The resulting language is very similar in appearance to S, but the underlying implementation and semantics are derived from Scheme. In fact, we implemented the language by first writing an interpreter for a Scheme subset and then progressively mutating it to resemble S.

We added S-like features in several stages. First, we altered the language parser so that the syntax would resemble that of S. This created a major change in the appearance of the language, but it should be emphasized that the change was entirely superficial; the underlying semantics remained those of Scheme. Next, we modified the data types of the language by removing the single scalar data type we had put into our Scheme and replacing it with the vector-based types of S. This was a much more substantive change and required major modifications to the interpreter. The final substantive change involved adding the S notion of *lazy arguments* for functions.

At this point we had enough of a framework in place to begin building a full statistical language. This process is ongoing, but we feel that we are well on the way to building a complete and useful piece of software. The development of the key portions of language

Ross Ihaka is Senior Lecturer, and Robert Gentleman is Senior Lecturer, Department of Statistics, University of Auckland, Private Bag 92019, Auckland, New Zealand; e-mail: ihaka@stat.auckland.ac.nz.

Journal of Computational and Graphical Statistics, Volume 5, Number 3, Pages 299–314

took two years of part-time effort on our part. This comparatively rapid development has been possible because we were able to make use of the well-documented work of Scheme (and Lisp) implementors and the pioneering work of Becker, Chambers, and Wilks as well as that of Tierney (1990).

The language we have developed differs from S in a number of ways. Some of this difference is just implementation detail, and some is more fundamental. In the following sections of this article we will describe what these differences are and the lessons that we feel are to be learned from our work.

We have named our language R—in part to acknowledge the influence of S and in part to celebrate our own efforts. Despite what has been a nearly all-consuming effort, we have managed to remain on the best of terms and retain our interest in computers and computing.

2. DESIGN ISSUES

2.1 Syntax

A computer language is described by its *syntax* and *semantics*. These provide descriptions of the form and substance of the language. In a sense, the syntax of a language is entirely superficial. What is or is not possible with a language is determined by its semantics. On the other hand, the syntax of a language is important because it determines the way that users of the language express themselves. Indeed, many users differentiate computer languages on the basis of their syntax rather than the underlying semantic differences.

This article concentrates on language structure, but clearly there are other issues that are important in a language for statistical computing. A major one being *tools*. The language—that is, syntax and semantics—plus the tools are combined into what can be called the run-time environment. This environment is the typical user's view of the program. Users want a language that is not only flexible but also provides them with a variety of tools for performing computations. Having good syntactic and semantic structure alone will not yield a useful language nor will their absence create a useless language.

Scheme provides a particularly good example of the design of a programming language. It is often regarded as a member of the Lisp family of languages, but Abelson, Sussman, and Sussman (1985) pointed out that it is as closely akin to Algol 60 as to early Lisps. Scheme concentrates on providing a small but powerful set of capabilities that are easily extended and can be used to solve a wide variety of problems.

In designing R, we chose to adopt the underlying evaluation model of Scheme together with the syntax of S. We chose the S syntax for our language because it is a powerful means for data analysts to express the computations that they need to carry out. Indeed, it is possible that the continuing success of S relative to Lisp-Stat (Tierney 1990) is because it provides a syntactic structure that expresses statistical ideas in a manner that statisticians feel comfortable with.

Scheme and S have obvious syntactic differences. Despite this they are actually quite similar in their basic structure. Consider the similarity in two definitions of a simple factorial function.

```
# factorial in S
factorial <- function(n)
    if(n <= 0) 1 else n * factorial(n - 1)

; factorial in Scheme
(define factorial (lambda (n)
    (if (<= n 0) 1 (* n (factorial (- n 1))))))
```

Once one understands that Scheme writes $f(x)$ as `(f x)` and $a - b$ as `(- a b)` and that `define` corresponds to `<-` and `lambda` to `function`, the two definitions are seen to be essentially identical. In fact, turning S expressions into the s-expressions of Scheme generally involves only simple rearrangements.

In what follows, we will suppose that the reader is familiar with S syntax. A description of that syntax can be found in Becker, Chambers, and Wilks (1985).

2.2 EVALUATION AND ENVIRONMENTS

A mathematical expression of the form $\cos(\pi/4)$ can be thought of as a symbolic description of how to compute a numerical value. Such an expression is meaningful only because we understand what its component symbols mean. In the case of this expression, we understand that cos is a mathematical function that computes cosines, π is a numerical constant that gives the ratio of a circle's circumference to its diameter, 4 is a small integer, and / is a binary operator (function of two arguments) that performs division. Similarly, an expression written in a computer language is not sufficient in itself to define the value to be returned by a computation. In addition, values must be associated with the symbols that appear in the expression.

In both Scheme and S the association between symbols and values is provided by what we will call an *environment*. An environment consists of a list of environment *frames*, each of which can be thought of as a list of symbol/value pairs. When a value is required for a symbol, the environment is searched frame-by-frame, pair-by-pair until a matching symbol/value pair is found. Because an environment is required by the evaluation process, we will speak of evaluating an expression in an environment. The same expression can yield different values when evaluated in different environments.

Environments are created during the process of function evaluation. For example, suppose we have a function `square` defined by

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

and we use it in the expression

```
square(10)
```

When this is evaluated, an environment frame is created that associates the symbol `x` with the value `10`. This becomes the first frame of an environment which we will write as

$$\{x \mapsto 10; \rho_p\},$$

where ρ_p denotes the following or *parent* frames of the environment. The body of the function (i.e., x * x) is then evaluated in this environment.

During the evaluation, values are sought for the symbols x and *. The value of x is found in the first environment frame, but the value of * is not defined there and must be sought in the other frames of the environment.

In Scheme, these other environment frames are those that existed when the function square was defined. This is in contrast with S, where the environment frames are a fixed set of *global* frames. *This apparently minor difference has major ramifications and arguably provides the major difference between the two languages.* Because R is derived from Scheme, it has adopted the Scheme evaluation model.

To see that there is a difference between the two evaluation models, consider the following example (which can be run in either R or S).

```
y <- 123
f <- function(x) {
    y <- x * x
    g <- function() print(y)
    g()
}
f(10)
```

When this code fragment is executed, S prints 123 and R prints 100. This points to a major difference between R and S that involves *free variables*; variables used in functions that are not defined by the formal parameters to the function. The manner in which values are attached to these free variables differs between R and S.

It is informative to follow the evaluation process through for this code fragment. In S the evaluation proceeds as follows. The initial assignment y <- 123 defines the value of y to be 123 in the global environment. We will represent this initial environment as

$$\rho_0 = \{y \mapsto 123; \rho_s\},$$

where ρ_s is the parent environment consisting of system-defined functions and constants. Defining the function f modifies ρ_0 to

$$\rho_0 = \{y \mapsto 123, f \mapsto \phi; \rho_s\},$$

where ϕ represents the function assigned to f.

When S evaluates f(10) it creates a new environment frame that associates x with the value 10 and which is parented by ρ_0,

$$\rho_1 = \{x \mapsto 10; \rho_0\}.$$

The body of f is then evaluated in ρ_1. The first assignments of the function body modify ρ_1 with assignments to y and g.

$$\rho_1 = \{x \mapsto 10, y \mapsto 100, g \mapsto \gamma; \rho_0\},$$

where γ is a function that prints the value of the symbol y. When the function g is invoked, a new environment frame is created. Because g has no formal arguments, this environment is given by

$$\rho_2 = \{\ ; \rho_0\}.$$

Note that the parent environment here is ρ_0. The body of g is now evaluated. During this process a value for y is sought. The search (starting in ρ_2) finds the value 123 in the first frame of ρ_0 and so it is this value that is printed.

In R, the evaluation process is identical up to the evaluation of g(). At this point the body of g is evaluated in the environment

$$\rho_2' = \{\ ; \rho_1\},$$

because it was ρ_1 that was in effect when g was defined. The search for y now finds the value 100 in the first frame of ρ_1.

One might think that the difference between R and S is that, in S, matching of variables to values takes place after the function is executed, but in R it takes place when the function is created via the associated environment. This, however, is not true. In R there is no association between symbols and values prior to function invocation. The associated environment is simply a repository for possible values; these can be changed directly by accessing the environment or indirectly via the actual arguments to the function. The real difference is that there is this associated environment that can be used to store information between function calls.

2.3 MAINTAINING STATE WITHIN FUNCTIONS

In both S and R, functions are important because they are the means by which computation takes place. Functions are first-class objects in both languages. This means that they are treated in the same way as other objects in the language (numbers, strings, etc). They may be returned as values by other functions and can be assigned as the value of a symbol.

R and S differ fundamentally in their ability to maintain state information within functions. As mentioned in the previous section, each function in R has an associated environment, namely the environment that was in effect when the function was defined. When a function is invoked, a new environment frame is created that associates the formal and actual arguments for that function. The body of the function is then evaluated in the environment obtained by inserting this frame before the first frame of the function's associated environment.

Assignments in the function body using the <- operator change the first frame of this environment. Both R and S provide a second assignment operator, <<-; y<<-10, for example. The effect of this kind of assignment in S is that a change is made directly to the global environment. In R, a search is made, starting with the parent environment and traversing upwards through the parent environments until the global environment is reached. The value of the first y encountered is changed. If no y symbol is encountered, then the global environment is modified to include a definition for y.

The combination of the Scheme/R evaluation model and the <<- operator provides a means for functions to maintain local state. Roughly speaking, this means that functions are able to remember the values of variables between invocations. The actual implications are much more profound.

To see how this works, consider the following function which models the way a bank account might work. Although this example is not statistical in nature there are many statistical problems, such as random number generation, for which such capability is important.

```
account <- function(total)
    list(
        balance = function() total,
        deposit = function(amount) total <<- total + amount,
        withdraw = function(amount) total <<- total - amount
    )
```

The function account takes a numerical argument total and returns a list containing three functions. Because these functions are defined in an environment that contains total, they will have access to its value. *This will be true even after the function* account *has returned.*

The effect of this function can be seen in the following dialog.

```
> Robert <- account(1000)
> Ross <- account(500)
> Robert$deposit(100)
> Ross$withdraw(150)
> Robert$balance()
[1] 1100
> Ross$balance()
[1] 350
```

The accounts for Robert and Ross are maintained separately as we would expect, with deposits, withdrawals, and balance inquiries for each Robert and Ross taking place on the appropriate account. This separation is possible because there is a distinct value of total for each Robert and Ross. The two totals exist in different environments. The first of these environments is created when account is invoked for the first time and an environment frame is created that associates total with the value 1,000. An extended environment that includes this frame is associated with the three functions returned by account. An environment with a different first frame is created by the second invocation of account and associated with the functions returned from it. These functions then manipulate the value of total in their associated environment. This manipulation is carried out with the <<- form of assignment. The three functions returned by a call to account share common persistent state information in the form of the value of the symbol total.

The ability to preserve state information between function invocations is a very useful feature for a programming language to have. Those with a deeper knowledge of programming systems will see immediately that it would be possible to build an *object-*

oriented system using precisely this kind of local state. We discussed some statistical applications of these ideas in Gentleman and Ihaka (1994).

Functions in S do not have the ability to preserve state information in this straightforward manner. This is because they do not have environments associated with them. In fact they can be thought of as *literals* in S. In other regards, functions in R and S are very similar. Both match formal and actual arguments in the same way and both allow default values that will give a value to a formal when no matching actual is supplied.

2.4 LAZY ARGUMENTS AND DEFAULT VALUES

Scheme uses a policy of *eager evaluation* for function arguments. All the arguments to a function are evaluated before the function body is evaluated. Some special functions, such as `if`, are exceptions to this general rule. By contrast, S uses a policy of *lazy evaluation* for function arguments. This means that the expressions given as function arguments are not evaluated before the function is called. Instead, the expressions are packaged together with the environment in which they should be evaluated and it is this package that is passed to the function. Evaluation takes place only when the value of the argument is required. Such a package can be thought of as a *promise* to evaluate should the need arise.

A policy of *lazy arguments* is very useful because it means that, in addition to the value of an argument, its symbolic form can be made available in the function being called. This can be very useful for specifying functions or models in symbolic form. For example, consider the following function which can be used to draw smooth curves, specified as a function of x.

```
curve <- function(expr, from, to) {
    x <- seq(from, to, length=500)
    y <- eval(substitute(expr))
    plot(x, y, type="l")
}
```

To draw the graph of the function $f(x) = x^2 - 1$ over the interval $[-2, 2]$, this function would be invoked as follows.

```
curve(x^2 - 1, -2, 2)
```

The unevaluated first argument to `curve` is returned by `substitute(expr)`. This is then evaluated by `eval` in the environment that the body is being evaluated in. This environment contains the value of x created by the call to `seq`.

There are several means of providing default values for missing arguments. In the first example, the function `missing` is used to determine whether an argument was supplied. The function `missing` when applied to a formal parameter returns true if the argument was not supplied and false if the argument was supplied.

```
sumsq <- function(y, about) {
    if(missing(about))
        sum((y - mean(y))^2)
    else
```

```
                sum((y - about))^2)
        }
```

If the parameter about is supplied, then it is used to center the y's and if not, the mean of the values is used as a centering value. A second method of providing default values is to assign them in the argument list using the = operator.

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

If no value is specified for about in a call of sumsq, the mean of the y values is used. In fact, the expression mean(y) is packaged in an evaluation promise together with the environment that the function body will be evaluated in. This means that the default argument is not evaluated until the argument is used for the first time. This delay of evaluation can be quite useful as the next version of sumsq shows.

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

The evaluation of the centering constant does not take place until after the NA's have been processed. An NA represents a missing value. Computations on vectors that contain NA's generally result in NA so they are often removed before the calculation is carried out. The default argument for about, in our example, has the correct value of the centering constant regardless of whether na.rm is set to true or false.

Another advantage of lazy evaluation is that functions like if do not need to be special. In Scheme if is a special form. With lazy evaluation it can be implemented as a standard function that chooses which of its arguments to evaluate.

Lazy argument evaluation is a very useful idea. Our original implementation of R omitted it, but it quickly became clear that other methods of providing the same functionality were much less elegant. There is one disadvantage to lazy evaluation, however. If assignments are made in function invocations, unusual behavior may result, as the following example shows.

```
> silly <- function(x,y) {
+   if(y < 10) print(x)
+   else print(y)
+ }
> silly(z <- 3, 14)
[1] 14
> z
Error: Object "z" not found
> silly(z <- 3, 4)
[1] 3
> z
[1] 3
```

Notice that the assignment, `z<-3`, takes place in the environment that `silly` was called from. The difference between the first call to `silly` and the second is that in the first, because `y` was larger than 10, the first line of code was not evaluated and hence the argument `x` was never evaluated. So the value 3 was never assigned to `z`. In the second case `x` was evaluated and hence the assignment of the value 3 to `z` did take place. Because of this possibility it is recommended that one not use assignments (or indeed any expressions with side-effects) in function calls.

It would be easy to modify the grammar of R to disallow assignments as function arguments. This would also introduce the possibility of using = as the assignment operator instead of, or in addition to, `<-`. We have chosen not to do this mainly for compatibility reasons. If S or R used = as the assignment operator under the current syntactic paradigm there would be confusion. It would not be clear whether the function invocation `foo(x=5)` meant invoke the function `foo` with formal argument `x` bound to `5` or to invoke `foo` with the first formal argument bound to `5` and set the global variable `x` equal to `5`.

3. IMPLEMENTATION ISSUES

3.1 PORTABILITY

We planned to carry out development of R on Unix workstations, but wanted to be able to use the production code in an instructional laboratory consisting of Macintosh computers. From the outset, then, portability was a major concern for us. Although it would have been possible to create a version of R that would run on smaller computers (such as those the based on the Intel 286 processor), this would have introduced unwanted complications and we decided to target the implementation at machines with at least 32-bit addresses. In addition, we decided to develop for an environment that provided two or more megabytes of directly addressable memory. Machines of this type are now the rule rather than the exception in computing.

To further foster portability, we chose to code R in ANSI standard C since this is now the standard programming environment for most computing platforms. C makes it possible to write efficient, compact code and provides easy access to all but the lowest level of machine capabilities. It might be argued that C++ would provide a better development vehicle, but because we planned on implementing higher-order functionality in R itself, it is unlikely that we would have taken advantage of the additional abstraction facilities of C++.

The decision to code in C meant that large amounts of useful FORTRAN applications code such as LINPACK (Dongarra, Bunch, Moler, and Stewart 1978) were not directly available to us. The availability of the AT&T/Bellcore `f2c` FORTRAN-to-C translator (Feldman 1990) provided a convenient solution to this problem. On platforms where C/FORTRAN interlanguage calling is either not available or inconvenient, we use `f2c` to obtain easy access to such FORTRAN code.

3.2 IMPLEMENTATION STRATEGY

We began the implementation of R by creating an interpreter for a subset of Scheme (continuations were the most notable omission from the subset). Implementing a Scheme interpreter is a comparatively straightforward task. Useful pointers on how to do it can be found in the books by Abelson, Sussman, and Sussman (1985) and Kamin (1990). We particularly recommend the latter for those interested in the practicalities of interpreter design. Our initial interpreter contained basic mechanisms for symbol-table management, memory management, and evaluation. It consisted of roughly 1,000 lines of C code. Despite its small size, the interpreter was quite powerful.

The first step in mutating the interpreter into something S-like was to add a parser to convert expressions in S syntax into Scheme. As mentioned in Section 2, this involves only trivial rearrangement of expressions, and is a straightforward programming task. We used the yacc parser generator (Johnson 1975) to generate computer code to carry out the rearrangement.

To this base we added the vector data types of S. These types are vectors that consist homogeneously of elements which are either logicals, integers, reals, or character strings. We chose not to implement a single-precision real type and have not yet implemented a complex type, but we did include an enum type for the representation of categorical variables. Integrating these vector types and providing the capability to access and mutate subsets of vector objects took a considerable amount of time. The final change we made was to add lazy evaluation and default values for function arguments. This involved comparatively minor changes to the syntax and evaluator.

3.3 INTERNAL STRUCTURE

The internal structure of R will be familiar to Lisp implementors. A common data structure is shared by the internal representation of most elements of the language (we will call these *basic language elements* or BLEs). The data structure consists of five machine words. The first of these is a tag that contains type information and some miscellaneous bit-fields that are used for internal purposes. The second word contains a (possibly null) pointer that is used to attach a list of attributes to objects. The contents of the remaining three words are interpreted according to the type of object being represented. The machine words are:

tag	*attr*	w_1	w_2	w_3

In all there are about 20 distinct types of BLE. In this section we will show the structure of a few of them.

3.3.1 Symbols

Symbols are the language elements used to name objects in R. In a symbol's BLE, w_1 is a pointer to a character string that gives the symbol's (print) name and w_2 and w_3 are pointers that can be used to associate system level objects with the symbol. Distinct

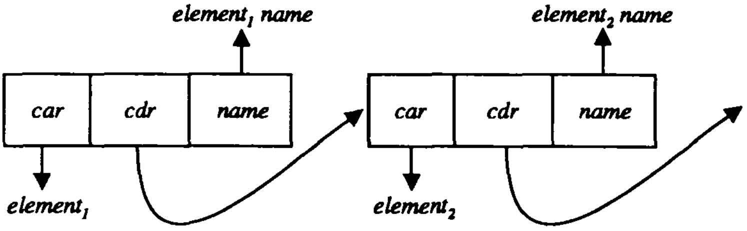

Figure 1. R Internal Memory Organization.

symbol BLEs are guaranteed to have different aming strings. This means that symbol comparisons can be done using the address of the BLE rather than the string comparison. Symbols and values are associated mainly through environments and not through w_2 and w_3.

3.3.2 Cons cells

These objects are the basic "glue" that holds the internal structures of the interpreter together. In particular, they are the building blocks of lists. In cons cells, w_1 and w_2 are interpreted as Lisp *car* and *cdr* pointers and w_3 is taken to provide an optional name for the object pointed to by the *car* pointer. Lists are assembled from cons-cells as shown in Figure 1.

3.3.3 Environments

The role of environments in the evaluation process has been described earlier in this article. An environment, BLE, has w_1, a pointer to a pointer to a frame (which is a list of names and associated values), and w_2, a pointer to a parent environment. w_3 is unused.

3.3.4 Vectors and Strings

These are variable-size objects that must be stored in contiguous blocks of memory. Each such block is referenced by a header stored in another five-word BLE. Such BLEs have the array length stored in w_1 and a pointer to the values in w_2.

3.4 MEMORY MANAGEMENT

The basic functionality that any computer language must provide is data management; the ability to store and retrieve data from computer memory. It is important that this base level *memory management* be as efficient as possible because all other functionality will build upon this layer. The cost of using a bad memory-management strategy can be very high, particularly in virtual memory systems. Such systems allow executing programs to (apparently) use more memory than is physically present in the computer. This is achieved by partitioning the executing image into memory *pages* and only keeping a small subset of the pages in memory at any given time. If consecutive memory references are localized this works well, but if the references are widely dispersed through the image, system performance can suffer drastically. This happens because memory pages must be copied to and from disk in a process called *paging*. Because disk access is extremely slow in comparison to memory access, programs that page heavily tend to run very slowly. Therefore, languages should use as little memory as possible to avoid paging and increase performance. S arguably uses more memory than is strictly necessary.

The problems of memory management are not unique to statistical software. Considerable effort has been expended by computer scientists in producing good memory-management strategies. An extensive review of techniques for uniprocessor systems can be found in Wilson (1990). A key point is that the amount of virtual memory allocated by a program should not exceed the amount of physical memory available. If this is not possible, it is important that memory references be as localized as possible.

We have used relatively simple memory management techniques in R. When R begins execution, a large array of BLEs is allocated in the memory heap. At the same time, another large portion of memory is allocated for the storage of variable sized objects (vectors and strings). The array of BLEs is linked together into a list structure and it is from this list that free BLEs are allocated as the interpreter needs them. Vectors and strings are stored in the memory set aside for variable sized objects.

When the list of free BLEs or the space for vectors is exhausted, a process called *garbage collection* is initiated. Garbage collection takes place in two phases. During the *marking* phase, all BLEs that can potentially take part in future computations are detected and marked. The BLE array is then *swept* and all unmarked BLEs are assembled back into a free BLE list structure. Simultaneously, all vector elements are moved into contiguous memory in a process called *compaction*.

The combination mark-sweep/compaction strategy is simple and effective. Garbage collection can happen at any time, allowing the best possible use of limited memory. Compaction of the vector area means that heap fragmentation is avoided and that a maximally large piece of contiguous memory can always be made available for vector allocation. Using these strategies we have been able to run R on small- to moderate-sized problems with less than two megabytes of memory allocated for object storage.

In the future, as R grows in size, we may have to resort to more sophisticated memory-management techniques. At present, however, it appears that the simple method described gives satisfactory performance.

4. TIMING

One of our goals in writing R was to provide better performance than S. In this section we report some comparisons. Unfortunately, it is very difficult to provide meaningful timing comparisons for R and S. There are a multitude of problems that cannot easily be overcome. A major problem is how time should be measured. It could certainly be argued that the real time it takes an individual to perform a task is the relevant time on which to make comparisons, but even this will be problematic. The time used by the CPU is not a practical measure because time used paging will not be measured if we concentrate

Table 1. Timing Comparisons

	1000:1	*1500:1*	*2000:1*	*Int*
S	94.1 (.18)	180.4 (.34)	291.2 (.33)	51.5 (.17)
R	18.1 (.18)	28.9 (.28)	40.1 (.35)	9.5 (.17)
Ratio	5.2	6.2	7.3	5.4

on CPU time. For R the amount of time required to do certain tasks will depend on the amount of memory available. The more times garbage collection occurs the more time a task will take. There is a similar problem with S; as a session proceeds it appears that some fragmentation occurs and that subsequent operations are slower. Although we have chosen to report real time on a particular machine with a clean image (both R and S are in pristine condition at the start of the test), the reader is cautioned not to over-interpret the results reported. Another major problem is that R is evolving. As we find procedures that can be simplified and memory allocation that can be avoided, we change the program (as do the developers of S). Thus, the timing comparisons reported here will probably be for a version of R that is obsolete by the time this article is published.

For many algorithms, both languages use internal calls to a compiled language and hence all we would compare in this case would be the algorithms rather than the languages and these are better compared elsewhere. We have decided to program some tasks in the interpreted language and to compare R and S (actually S-Plus, ver. 3.1) on these. The two tasks that we report on are heap sort and numerical integration. They have been coded in a form that runs under both systems. A Sparcstation IPX was used to run the comparisons with both R and S binaries located on a remote file server. For heap sort, the task is to sort the integers from 1,000 down to 1 into ascending order. For numerical integration, the function $f(x) = (100/x^2)\sin(10/x)$ is integrated for $x \in [1, 3]$. Both functions and associated code are given in the appendix. Their inclusion is for completeness; neither is intended to be a definitive implementation.

The results in Table 1 are mean time in seconds with the standard error (of the mean) indicated in parentheses. These results indicate that R is faster than S. Furthermore, the larger the problem the more the gain since the ratio of the times taken for heap sort are 5.1, 6.2, and 7.3. To some extent this reflects savings due to a conservative approach with copying in R, but they also reflect better memory management. A big cost in performance comes from copying all arguments to functions. In R we have adopted a conservative approach to this. Basically we attempt to only copy an object when it is changed (mutated) in the body of the function. Objects that cannot be accessed from the symbol table can be mutated directly rather than copied and then mutated. This approach can save a great deal of time especially if the structure being copied/mutated is large.

5. CONCLUSIONS

5.1 THE COST OF R

So far we have concentrated on the advantages of R. We would now like to indicate some of the problems, or costs, that are associated with the decisions we have made. Perhaps the largest problem that results from our design decisions is the fact that everything must be stored internally. This is in contrast to S, where most data objects are stored in files on disk. If R crashes, there is no method of saving the current environment because it is almost surely corrupted. This suggests that it is a good strategy to save images fairly often so that only a small amount of work is lost.

Another problem is caused by the scoping method used in R. The fact that functions have an associated environment means that there is no simple way in which functions can be saved. They must be stored together with their environment, and any other functions that refer to that environment, their associated environments, and so on.

In fact, the only way to save portions of work from an R session is to save the entire memory image from the session (including system functions). This makes saved R images quite large.

5.2 Future Developments

In any project of this size—and in particular in the area of computing—there are lots of extensions that can be made. Some that are particularly relevant are source-level debugging and compiling. Again, we hope to adapt procedures already common in Lisp implementations to our situation. Most Lisp implementations have very good, intelligent compilers and with some work it is feasible that these can be adapted to R. There are reasons that this may not result in as great an increase in speed as for other languages though. Part of the success of S is that the looping facilities are very general. One may iterate over lists containing virtually any type of object. In order to write a compiler it will be difficult to obtain good, fast code while maintaining this generality. That does not preclude the possibility of selectively compiling some functions while leaving others as interpreted functions. If the compiler is sufficiently specific, it may be the case that most users get more speed and functionality through a simplified foreign function interface.

APPENDIX: PROGRAM LISTING

A.1 Heap Sort

```
heap <- function(ra) {
        n<-length(ra)
        l <- floor(n/2)+1
        ir <- n
        while (5 == 5) {
                if (l > 1){
                        l<-l-1
                        rra <- ra[l]} else {
                                rra <- ra[ir]
                                ra[ir] <- ra[1]
                                ir <- ir-1
                                if (ir == 1) {
                                        ra[1] <- rra
                                        return(ra)
                                }
                }
                i <- 1
                j <- 1*2
                while (j <= ir) {
                        if (j < ir)
                                if (ra[j] < ra[j+1]) j<-j+1
                        if (rra < ra[j]) {
                                ra[i] <- ra[j]
                                i<- j
                                j <- j + i
                        } else j <- ir+1
                }
                ra[i] <-rra;
        }
}
```

A.2 INTEGRATION

```
EPS <- 1.0e-5
jmax <- 20

qtrap<-function(f,a,b)
{
        olds<-(-1.0e30)
        for (j in 1:jmax) {
                s<-trapzd(f,a,b,j)
                if (abs(s-olds) < EPS*abs(olds)) return(s)
                olds<-s
        }
        print("too many iterations")
}

trapzd<-function(f,a,b,n) {
        if(n==1) {
                qit<<-1
                strap<<- 0.5*(b-a)*(f(a)+f(b))
                return(strap)
        }
        else {
                tnm<-qit
                del<-(b-a)/tnm
                x<-a+0.5*del
                sum<-0
                for(j in 1:qit) {
                        sum<-sum+f(x)
                        x<-x+del
                }
                qit<<-qit*2
                strap<<-0.5*(strap+(b-a)*sum/tnm)
                return(strap)
        }
}
```

ACKNOWLEDGMENTS

The authors thank John Chambers, Mike Meyer, and Duncan Murdoch for their helpful comments on a draft of this article. We also thank our colleagues and students that were and still are our guinea pigs.

[Received May 1995. Revised April 1996.]

REFERENCES

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

Becker, R. A., Chambers, J. M., and Wilks, A. R. (1988), *The New S Language*, Pacific Grove, CA: Wadsworth.

Dongarra, J., Bunch, J. R., Moler, C. B., and Stewart, G. W. (1978), *LINPACK Users Guide*, Philadelphia, PA: SIAM Publications.

Feldman, S. I., Gay, D. M., Maimone, M. W., and Schryer, N. L (1990), "A Fortran-to-C Converter," Computing Science Technical Report No. 149, AT&T Bell Laboratories, Murray Hill, NJ 07974.

Gentleman, R. C., and Ihaka, R. (1994), "Lexical Scope and Statistical Computing," Technical Report No. 2, University of Auckland, Dept. of Statistics.

Johnson, S. C. (1975), "Yacc: Yet Another Compiler Compiler," Computing Science Technical Report No. 32, AT&T Bell Laboratories, Murray Hill, NJ 07974.

Kamin, S. N. (1990), *Programming Languages*. Reading, MA: Addison-Wesley.

Steel, G. L., and Sussman, G. J. (1975), "Scheme: An Interpreter for the Extended Lambda Calculus," Memo 349, MIT Artificial Intelligence Laboratory.

Tierney, L. (1990), *Lisp-Stat*, New York: John Wiley.

Wilson, P. R. (1990), "Uniprocessor Garbage Collection Techniques," in *The Proceedings of the 1992 International Workshop on Memory Management*, Springer-Verlag Lecture Notes on Computer Science.