available, free-of-charge, to others. There is, in essence, no support provided, other than an effort by Tierney to fix bugs, and minimal on-line help. This, of course, is a mixed blessing, but since the quality of the software is very high, and since there are few bugs, there seems to be little need for support.

The book, of course, is not free, though it is reasonably priced. I have already described all of the chapters in the book, emphasizing that the book is especially strong on dynamic graphics. The book is useful as Lisp-Stat documentation, providing a tutorial and examples of using and extending the system. The book is also a good introduction to functional and object-oriented programming, as used in statistics, and to Lisp.

10. CONCLUSION

Lisp-Stat is the most important, exciting and promising development in computational and graphical statistics in recent years. It provides a foundation on which computational statisticians can build a statistical system offering all types of statistical and data analysis tools—from basic to advanced—to all types of users—from novices to sophisticates. As it stands, Lisp-Stat is not (and does not claim to be) a statistical system that provides a wide range of analysis tools for a wide range of users. However, with the proper extensions, Lisp-Stat could become the standard by which other systems are judged. In sum, Lisp-Stat is the statistical environment "for the best of us," not "for the rest of us"—yet.

Comment: Two Functional Programming Environments for Statistics — Lisp-Stat and S

David J. Lubinsky

1. GENEALOGY

There is a German saying, "Tell me where you come from and I will tell you who you are," and this is perhaps even more true for the two statistical environments that are the subject of this review. They are both the products of many ancestors and each reflects its heritage. Both Lisp-Stat (L_S; the idea of this notation is that Lisp-Stat is a Lisp system specialized for statistics) and S are descendants in the line of interactive, interpretive systems, starting with APL and Lisp; L_S also draws inspiration from S, Smalltalk and dynamic graphics systems (Cleveland and McGill, 1988). The two systems are both interpretive programming environments using functional languages. They each have vectorized arithmetic operations and support a wide set of statistical primitives. In addition each has strong support for graphical display of data.

In the family of statistical and computing systems, L_S and S are very close, and as in all families, there is a natural rivalry, but also a natural affinity between them. S is the older brother, more mature and more complete. Whereas L_S is faster, incorporates many new ideas in graphics and object-oriented programming, but still has a long way to go before it can compete with S in all areas.

They are both designed to be used for more than canned analyses of data. Each allows users to combine standard analyses in nonstandard and flexible ways and, more importantly, to implement and experiment on new techniques.

The rest of this section introduces each system by presenting the same function coded in each language and discusses the general areas of application in which each system is stronger. The following section is a more detailed comparison of languages and primitives in each system. This is followed by a comparison of performance of the systems, and the last section discusses the their documentation.

1.1 A Running Example

To help make the similarities and differences more concrete, Figure 1 shows how one would implement a running smoother using S and L_S. Each function takes two required arguments, **x** and **y**, and returns a set of smoothed values at equally spaced points along the range of the **x**'s. They also take two optional arguments, the function to be used to find local values of the smooth, and the number of points in the returned smooth. Two obvious examples of local smoothing functions would be the mean (the default) and the median. These

David J. Lubinsky is a member of the technical staff at AT&T Bell Laboratories, Room 25524, Crawfords Center Road, Holmdel, New Jersey 07733.

```
running <- function(x,y,fun=mean,
              npoints=50,fraction=0.1){
    firstgte <- function(val,vec)
        (1:length(vec))[vec>=val][1]

    o <- order(x)
    x <- x[o]
    y <- y[o]
    n <- length(x)
    r <- range(x)

    xps <- seq(r[1],r[2],length=npoints)
    span <- round(fraction*npoints)
    lpx <- 1
    rpx <- firstgte(xps[span],x)
    s <- xps


    for(i in 1:npoints) {
        s[i] <- fun(y[lpx:(rpx-1)])
        lp <- max(1,i-span)
        rp <- min(npoints,i+span)
        lpx <- firstgte(xps[lp],x)
        rpx <- firstgte(xps[rp],x)
    }
    list(x=xps,y=s)
}
```

Declare the function, and its arguments. Note that `mean` is a function.

Define a local function, `firstgte` to find the position of the first element in `vec` which is ≥ `val`.

Find the sort order for x, and sort x, and reorder y. Use vectors in LS for quick indexing.

Set `n`, to to the number of points.

In S, use the `range function` to find the `min` and `max`, in LS use `min` and `max`.

Let `xps` be a vector of `npoints` evenly spaced values, along the range of x.

Set the `span` based on the `fraction` of the range. Let `rpx`, be the position of the first x value greater than the `xps` at `span`.

S will contain the smoothed values.

Loop over the `npoints`, with `xps[i]` at the center of each interval. And calculate the function value for the `y's` from the left position `lpx` to the right position `rpx`, and save in `s`.

Then update the pointers in the `xps` vector, and find the corresponding left and right positions in the x vector.

Return the `xps` and smoothed valued. In LS, must first reverse the list, since we added to the front of the list as it was created.

```
(defun running(x y &key (fun #'mean)
                (npoints 50) (fraction 0.1))
 (flet ((firstgte (vec val pos)
          (while (< (select vec pos) val)
              (setq pos (1+ pos)))
          pos))
 (let* ((o (order x))
    (x (coerce (select x o) 'vector))
    (y (coerce (select y o) 'vector))
    (minx (min x))
    (maxx (max x))
    (xps (coerce (rseq minx maxx npoints)
         'vector))
    (span (round (* fraction npoints)))
    (lpx 0)
    (rpx (firstgte x (select xps (- span 1)) 0))
    (s ()) lp rp)

 (dotimes (i npoints)
   (setf s (cons (apply fun
   (select y (iseq lpx (- rpx 1))) ()) s))
   (setf lp (max 0 (- i span)))
   (setf rp (min (- npoints 1) (+ i span)))
   (setq lpx (firstgte x (select xps lp) lpx))
   (setq rpx (firstgte x (select xps rp) rpx))
 )
 (list xps (reverse s)))))
```

FIG. 1. *Programs in S and L_S for computing running smooths.*

functions illustrate the general programming style in each language and, more particularly, how functions can be used as data, how local functions are defined and how vectorized arithmetic and looping are used.

For those who are not familiar with Lisp, the swarm of parentheses in the L_S version might be dismaying, but once you get used to reading Lisp, they are no barrier to understanding the code. Also, many editing systems support the development of Lisp code by automatic indenting and parenthesis matching. Once you get past looking at the parenthesis, you will notice that the algorithms are represented in very similar ways.

The L_S program is longer, consisting of 710 keystrokes (only 134 of them are parentheses), while the S version is 460 keystrokes. The fact that there is less to read and type is not necessarily an argument in favor of S; reading APL code confirms this. However, many statisticians do say that they find the S representation closer to their way of thinking, since it is closer to standard algebraic notation and similar (at least syntactically) to the languages that most statisticians first learn, such as Fortran and C.

In S, the idea was to build an interpreter for a language specially defined for statistical computation. In L_S, the idea was to borrow strength from a well-developed language and add the functions and primitives needed for statistical and graphical computation. This is the fundamental difference between the two approaches.

2. COMPARISON OF CAPABILITIES

2.1 What Is Each System Best For?

After using either system for a long time, the idioms suggested by the language will become the mental constructs used for translating algorithmic steps into computable steps. The idioms available in each language are rich enough to allow compact implementations of statistical ideas. But still, each system has it strengths and weaknesses. To make the distinctions more precise, we compare the two systems on a number of dimensions that are important for any system for data analysis and statistical research.

2.2 Built-in Functions

Both environments include vectorized arithmetic, linear algebra, distribution function primitives and regression computations. But one of the biggest differences between L_S and S is that S provides many more data manipulation and statistical and

graphical routines. These are both of the kind used everyday, such as matching vectors, or drawing barplots, and functions that are used less often such as minimal spanning tree, and cluster analysis.

There is no limit to the number of possible functions that could be included in a statistical programming environment, but the facilities provided by L_S are still too sparse to allow it to be a contender as a replacement for a more fully endowed system such as S.

S has a much richer set of primitives of all kinds. This leads to two improvements. First, the provided primitives do not need to be rewritten and tested from scratch, which makes it easier to program many applications in S, and second, often the primitives are implemented at the C level, so they run faster than the equivalent code in S or Lisp. For example, S has a function called `match` that returns the position of each element of its first argument in the second. It is easy to code `match` in L_S: `(defun match(x y)(mapcar # '(lambda (v) (position v y)) x))`. But this function is comparatively slow. Other functions which are not provided by L_S would be much more difficult to implement.

The notation and functions supplied with the system are those that the authors thought necessary for a high level statistical environment, and it is these features that users will be most likely to use. The notation provided in a particular language becomes a tool for thought. After using a language for long enough, people begin to use a subset the primitives provided in the language in their own thinking. So it is essential that features be provided from the start, rather than seen as possible later additions.

In the beginning of the L_S book, there is an acknowledgment that L_S has used many of the ideas from S, and this is a good thing. Certainly many of the forms, primitives and data structures in S are generally useful in any statistical programming environment. But, as a seasoned S programmer, I often find myself wishing that L_S was even more like S.

2.2.1 Statistical Modeling. Both S and L_S provide functions for fitting regression and nonlinear models and examining various types of diagnostic information about the model. The 1991 version of S also has functions for fitting many other kinds of models (ANOVA, GLMs, generalized additive models, tree-based models and local regression models). L_S is not as complete, but a GLM package is already available, and perhaps as the system attracts a larger user base, more modeling routines will be added.

Both systems return objects that describe the model fit, rather than only printing out the results. This is important both for implementing techniques using model fitting as one stage, and for examining the model.

2.2.2 Graphics. L_S's unique capabilities are most apparent in the fact that its uses a bitmapped graphics model that supports dynamic graphics. The graphics model requires an underlying window system and creates a separate window for each plot. These windows can be interacted with separately and can be customized so that they execute specified functions for each of a number of types of interaction (including mouse and keyboard actions). This allows plots to be easily linked in different ways and dynamic displays to be programmed directly. Some of the most exciting material in the L_S book is contained in the last chapter describing dynamic graphics and animation applications.

In S, the graphics model is that output goes to a plotter, a pen writing on paper. This means that S is ideally suited for creating paper graphics and provides many functions for creating standard and customized plots. One measure of this is the number of conference and journal papers one sees in which the graphics are clearly from S (they are identified by the Helvetica font, which S uses as its default for Postscript output). Figure 2 shows an example of a customized plot produced with S. In L_S, the only way to get a hard copy is a screen dump of a window.

It would be most useful for users if these two models could be neatly combined. One simple suggestion would be for L_S to keep track of all the commands used in creating a window's image, and then, when printing the window, instead of just

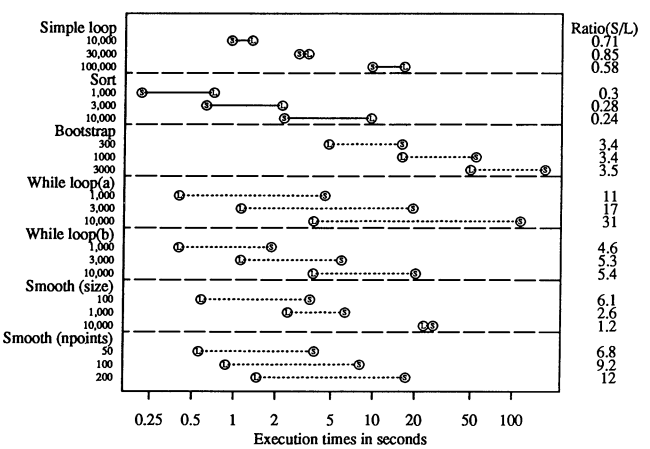

FIG. 2. *Comparison of S and Lisp-Stat (L), on a number of tasks. The dotted lines show cases where L_S is quicker and solid lines where S is quicker. The X-axis is on a log scale, so differences are proportional to the logs of the ratios in execution speed. The ratios of execution times are shown to the right.*

copying the bitmap, Postscript equivalents of the graphics commands could be generated.

Another limitation of the L_S graphics is that only one font is provided. This is surprising, given the range of fonts supported in both the Macintosh and X environments. Having only size of text limits the amount and type of textual information that can be displayed by L_S.

Since the chief advantage of L_S is that it uses a pixmap rather than a plotter model, it is also surprising that it lacks many basic primitives for handling pixmaps. The only feature provided by L_S is to copy a pixmap represented by an integer array of 0's and 1's, onto the screen. Some of the features that are missing are: functions for copying pixmaps in compressed format from memory onto the screen, and from the screen to memory, moving regions of the screen in primitive operations and dealing with color pixmaps. Implementing any of these operations at the Lisp level would be prohibitively slow (requiring nested loops, to loop over the values of pixels), so this is a defect that would not be easy for the user to remedy. Also, the X version of L_S does not implement backing store for windows, so that, when window are uncovered, they must be redrawn which can be time consuming for complex images.

2.3 Data Manipulation

Data are the stuff of statistics and, to a large extent, the ease with which data can be structured and manipulated will determine the usefulness of a statistical system. In the following sections, we discuss a number of issues related to representation and manipulation of data.

2.3.1 Vectors and Lists. L_S has two ways of representing sequences of values, lists in which each element points to the next one, so only sequential access is possible, and vectors, where elements are adjacent and of constant size, so arbitrary access is possible. It is much quicker to access random elements of a vector than of a list, but it is not clear which functions work only on lists and not on vectors. I would have appreciated a bit more advice on when to use which. Many of the built-in functions demand list arguments and some can also work on vectors, but, except for the functions explicitly for dealing with vectors, none of the functions demand vectors. So, good advice would be: Unless you are accessing elements randomly, use lists. And when accessing list elements sequentially use `dolist` or `mapcar`. It would be nice, though, if all functions that use sequences could accept vectors or lists.

S also has vectors and lists, but the meaning is somewhat different. In S, vectors contain elements of one of the basic data types, while lists are recursive data structures that may have anything as elements. In L_S, both lists and vectors can have arbitrary valued elements.

2.3.2 Type Coercion. S has only three types for representing data (other types represent functions, expressions and so on), `logical`, `numeric` and `character`, and has automatic coercion among them whenever it is sensible. This can be very useful, and, since S does not do conversions that lose accuracy, one is usually not harmed by them. The conversion of booleans to 0's and 1's is particularly useful.

L_S has an extra data type, `character` (for single characters) and it also distinguishes between `fixnums` for integers and `flonums` for reals. L_S does not specifically have a Boolean type, but in conditionals, any value except `NIL` is treated as true.

Since L_S vectors and lists can have anything in each element, there is not as much need for coercion, as in S, where each element of a vector must be of the same type.

2.3.3 Indexing. Indexing in S is much richer than in L_S and gives programming in S a distinct and pleasing flavor. Data objects acquire a plasticity that allows them to be manipulated in surprising and elegant ways. S has four indexing modes. A set of positive integers selects the indicated elements. A set of negative integers selects all but the elements indicated by the absolute values of the indices. A vector of logical values selects only those elements corresponding to TRUE values in the index vector, and a set of strings selects those elements whose names correspond to the string values. This last feature allows S arrays to have the flavor of associative arrays.

In L_S, only the first indexing mode is provided. Also, when indexing arrays in S, leaving out the values for a dimension implies selecting all elements along that dimension. In L_S, all dimensions must be specified.

2.4 Missing Values

In S, missing values are represented by a special value and are propagated by arithmetic operations. There is no mention at all of missing values in L_S. The convention could be adopted by users, to represent missing values by say NIL, but applying the arithmetic operators to any nonnumeric value causes an error. So, except for rewriting all the arithmetic functions, it is not possible to propagate missing values in L_S.

A related issue is how the two systems deal with arguments that are out of the domain of a function (e.g., division by zero). In S this returns an NA

(missing value), whereas in L_S it is an error. There are arguments for both approaches, but the S approach is more flexible. In S, for example, it is convenient that taking the mean of a zero length vector returns NA. Whereas in L_S, this would result in an error, so taking a mean in a program would have to be guarded by a conditional.

2.5 Pointers

Both S and L_S are functional programming environments. Each system is deeply influenced by the idea of computer programming as a set of functional transformations of data. In fact, even though L_S is based on Lisp, the grandfather of functional programming languages, the designers of S have taken the idea even further. Global assignments in S are not encouraged, and thanks to its design not really needed. And, more dramatically, S has no pointers, so the only way an object can be modified is when its name appears in a expression on the left hand side of an assignment.

In Lisp, on the other hand, assignment only copies a scalar or a pointer value. So it is possible (and often the cause of hard-to-find bugs) to have two names sharing the same structure, and by modifying one value, the other is modified as well. This allows for more efficient programming in many cases, especially with deeply nested data structures, and structures which often change size. This is also important for implementing object-oriented models of low level objects such as screen windows which need to be rapidly updated.

2.6 Delayed Execution

To motivate this section we begin with two short programs listed in Table 1. Each of these little programs implements an increment operation, but they do it in very different ways. In Lisp, `incr` is a macro, which works by textual substitution, so a call like `(incr x)` is identical to `(setf x (1+x))`. But in S, `incr(x)` is an ordinary function call. Two features of S allow this to work. First, S does not evaluate the arguments of functions till they are needed (this is known as lazy evaluation), and the function `substitute` replaces each occurrence of the argument with the expression in the call (note, the expression, not the value). Then `eval` evaluates the new expression in the environment of the caller. This is more complex than the Lisp version, but also much more flexible. Lazy evaluation has many applications, one which is to label plots with the expressions used in creating them.

TABLE 1
Increment operator program

L_S	S
(defmacro incr(a) ' (setf, a (1 + , a)))	incr ← function (a) eval(substitute(a ← a + 1), local = sys.parent())

2.7 Data Structuring Facilities

The title of N. Wirth's (1976) famous book is *Data Structures + Algorithms = Programs.* In statistical computing, we too often forget the data structures and think only about the algorithms. This leads to opaque data representations, and programs that are hard to understand or modify and re-use.

One contribution of both S and L_S is that they provide us with much greater flexibility in building data structures than is available in the more traditional numeric computing languages such as Fortran and APL.

For example, a binary sort tree would be represented in Fortran as an array with one row for each node, and three columns, the first containing the value, and the second and third the row index of the left and right children.

In L_S and S, you might represent the tree, by creating an object which has three slots, `value`, `left` and `right`. In L_S, this object might be an object in the class system (or a simple list), in S, you could also implement a class, but would be more likely to use a list. Then, the `value` slot can contain any value that can be ordered and is not restricted to an integer. Also, the routines to insert and delete nodes can be written much more clearly, since they deal with node objects directly rather than the matrix.

One disadvantage with S is that you have to rebuild the whole tree every time you insert a node, whereas in Lisp you can (for the sake of efficiency) set the value of the appropriate leaf, without the added expense of recreating the tree. So, if you wanted to create a tree containing many nodes you would be forced to use the Fortran approach in S.

This comment does not negate in any way S's facility for representing data structures. Rather, it is a warning that these facilities are only of use for data structures that are not updated too often.

2.8 Object-Oriented Facilities

L_S and the 1991 release of S both include facilities for object-oriented programming. In L_S, this is at the very heart of the design. The subtitle of the L_S book is "An Object-Oriented Environment for Statistical Computing and Dynamic Graphics." In S, the object-oriented facilities have only been added recently.

In S, any object may be given a `class` attribute that lists the set of classes, which the object inherits from. If, for example, an object named `a` has class `point`, the function call `describe(a)` will be translated into a call to the function `describe.point(a)`. The message dispatching facility is built directly on S objects. In S, the class system is currently used mainly in model objects.

The L_S object-oriented system is more formal, in that new objects can only be created by sending a `:new` message to an already existing object. Each class is represented by a special object called a `prototype` from which new objects of the class can be created. Prototypes can inherit from other prototypes and contain slots for values particular to the class. The inheritance system of L_S is particularly useful for the graphics prototypes, so that graphics windows can inherit appropriate methods, for common functions. For example, all graphics windows can inherit a `close-window` method from the basic `window-prototype`.

The ability of object-oriented systems to encapsulate data and methods for classes of objects is only now becoming a reality in statistical computing systems but will be increasingly important, and both systems allow researchers to experiment with these important ideas.

3. PROGRAMMING ENVIRONMENT

The usefulness of a system is determined, not only by the language and primitives it provides, but also by the environment it provides for developing and testing programs. In the following sections, we discuss some of the factors relating to this environment.

3.1 Debugging

Debugging in S and L_S are similar. Since they are both interpreters, the execution stack at the time the error occurred is available to be examined. In L_S, the execution is suspended, in the environment in which the error occurred. However, there is no way of examining the values of local variables other than those in the function that contained the error.

S allows any action to be defined at the time of an error, and the default action is to dump a copy of all of the frames (values of local variables) in the stack at the time of the error, and this dump can be examined using a special `debugger` function, that allows any of the local variables to be examined.

The debugger in L_S is primitive compared to what is available in more professional Lisps. If L_S is moved to Common Lisp, one of the benefits will be the improved debugging facilities in professional Lisp systems. This is part of the tradeoff in deciding to use a public domain Lisp system.

Also, both systems provide facilities for tracing functions and for setting break-points at which variable values can be examined.

3.2 Persistent Objects

Another difference in philosophy between S and L_S is the approach to persistent objects. S shares with APL the idea that objects assigned at the top level become permanent and can be accessed during later sessions. S extends APL's single workspace concept by allowing multiple directories in a search path. This is a useful for organizing functions and data objects by projects.

In L_S, plain text files describing the functions and data in Lisp syntax are loaded at the beginning of each session. This can be automated somewhat by putting the names of the files to be loaded in a file that is automatically loaded at the start of the session. In fact, each time L_S starts, it loads a number of files containing the functions that extend Lisp. Variables that are created during the session can be explicitly saved in a text format and loaded again at the next session.

The S approach takes less organization on the part of the user and also prevents important values from being lost by mistake, but it can waste space by saving unneeded objects.

3.3 Interfaces

The UNIX operating system is designed to allow complex manipulations to be constructed from simple tools, and it provides many such tools. So, it is important that systems running under UNIX have a smooth integration with the operating system and are able to take advantage of these tools. With S, it is easy to use the tools provided by the operating system and get their results. A similar though somewhat less convenient facility is also provided in L_S.

Also, since much statistical/numeric code already exists in Fortran and C, it is important to be able to call new routines from within the system. L_S and S share similar methods for loading and calling compiled functions from within the interpreter.

4. TIMING COMPARISONS

One of the keys to usefulness is execution speed, and, as quicker computers become available, more demanding applications are invented. We will always tax our computers to their fullest, and in any comparison of systems it is the timing comparison

that get most attention. The rest of this section is a comparison of the execution speed of a number of tasks, but it should be pointed out that the results hold only for the programs discussed on a particular computer, with the particular versions of each system.

4.1 Description of Tasks

Table 2 describes the code used in the timing tests. We also used the running smooth code shown in Figure 1 in two of the tests.

Figure 2 summarizes the results of the comparison. All reported times are the sum of the system time and the user time reported by UNIX, on a SparcStation 1+ with 16 M of memory. These times were within one second of the elapsed time for all comparisons shown although it is possible to get both systems into states, where swapping dominates the computation and the system times would be much less than elapsed time. In all the comparisons, I used the December 1989 AT&T version of S and version 2.1 of L_S.

In order to give the best comparison, I set the `alloc` size for L_S to 20,000 (the size of allocated blocks). Tuning the memory parameters of S did not seem to affect performance much.

4.2 Evaluation of Results

For the simple, vector-oriented test, *sort*, S is three to four times quicker. Executing an empty loop over a vector, S is slightly faster. When the vector has 100,000 elements, S is almost twice as fast, since S vectors have much less overhead, and hence require less swapping.

For applications involving any computations at the interpreter level, the S interpreter is much slower than the Lisp interpreter. For the simple bootstrap example, L_S is more than 3 times faster. The cost of executing the while loop grows linearly for L_S, but S slows down, so it takes more than 30 times longer to perform 10,000 iterations. The second *while* loop example gets much better performance since S cleans up storage at the end of loops, so many, shorter loops are much quicker. This is a strange quirk of the S garbage collection system, but it emphasizes the point that, when using interpretive systems, memory management is the key to efficiency, and whatever system one uses, one has to be careful about how memory is used.

For the running smooth examples, the first example shows how the performance changes as the size of the vectors to be smoothed increases. S and L_S approach parity for vectors of 10,000 elements. In the second example, as we increase the number of points at which the smooth is evaluated, the advantage of L_S increases.

4.3 Why S Is Slower

Since S stores its functions in a pre-parsed form, the greater syntactic complexity of the S language is no hindrance. One reason that S is slower is that its memory management is inefficient. Every time

TABLE 2
Code used in the timing tests

Simple loop	Do nothing in a loop that executes n times.
L_S	`(defun 1(n) (dotimes (i n)))`
S	`1 ← function(n) for (i in 1:n) { }`
Sort	Create a vector of random uniforms of length n and sort it.
L_S	`(sort (uniform-rand m) #'<)`
S	`sort(runif(n))`
Bootstrap	Find the medians of n samples of 100 uniform numbers.

L_S	`(defun f(n)`
	`(mapscar #'(lambda (x) (median (uniform-rand 100)))`
	`(iseq n)))`

S	`f ← function(n) {`
	`a ← 1:n`
	`for(i in seq(a)) a[i] ← median(runif(100))`
	`a`
	`}`

While loop	Execute a while[a] loop with a simple addition, n times.
L_S	`(defun m(n) ((i 0)) (while (< i n) (setq i (1 + 1))))`

S	`a)m ← function(n) {i ← 1; while(i <= n) i ← i + 1}`
	`b)m2 ← function(n) { i ← −1; while(i <= n/100) {m1(); i ← i + 1}}`
	`m1 ← function( ) {i ← 1; while(i <= 100) i ← i + 1}`

[a] *While* is not a part of the L_S syntax, but is defined by the following macro: `(defmacro while(test &rest body) '(do () ((not ,test)) ,@body))`.

an object is assigned, a new copy is created. Besides the overhead in creating the new copy, this also increases the amount of space S uses, and paging activity soon follows. And heavy paging kills performance. In contrast, everything in Lisp is either an atom or a pointer, so all assignments are quick. In S, even indexed assignment, which should not demand that the entire object be copied, does cause the size to grow as if a copy had been made.

The second, related, problem is that S does not have an efficient garbage collection mechanism. This also causes the S process to grow, sometimes without bound. This is the reason that the ratios for the *While (a)* and *Smooth (npoints)* examples increase as the number of iterations increases.

Although S is generally slower, "slow" is only relative to the application and for most applications; there is no difference between a response of a third of a second and a second. S's popularity is the best proof that the speed that really matters is the speed in which the overall problem gets done, not compute speed per se.

Execution speed is particularly important though for dynamic graphics, where substantial amounts of computing must be done within the integration time of the eye (about 1/20 of a second) to create the illusion of continuous motion. On current hardware, S would not be fast enough to support such computations, unless the primitives were written in a compiled language.

Both S and L_S are interpreters, so they are fundamentally slower than compiled languages, but the convenience is worth it in all but the most demanding applications.

5. CUSTOMER PROGRAMS

Neither S nor L_S is yet at the stage where it can be called "production software"; both systems have bugs of different types, and workarounds have to be invented. But both systems are so powerful that there is a strong temptation to use them to develop programs to be used by others.

This is an increasingly important area since, as computing facilities become more available and statistical expertise more needed, statisticians, will write more and more programs to be used by others.

5.1 Interaction with the User

An important part of any program written for nonexpert user is the user interface. The emphasis in S is on writing code to be used only by the author, and only a primitive text menu facility is provided. In L_S, there are a number of functions that take advantage of the bit-mapped display and mouse to allow a pleasant user interface to be created using button, menus and dialog screens.

5.2 Error Trapping

Both systems provide error trapping facilities to allow a function to continue running after an error has occurred. But in both systems the facilities provided are basic. There is no way to tell exactly where the error occurred or what the error was. So a user interrupt would be treated in the same way as an execution error.

5.3 Reading in Data

One common step in a system designed for statistical analysis is to read in data, usually stored in text files, to be used in the analysis. Here, too, both systems provide only basic support, with functions for reading in data stored in columns. UNIX tools, like AWK, can be used to supplement these facilities, but if the data input has to be very robust, this will involve writing a long AWK, program, and it would be more pleasant if all of the programming could be in a single language.

6. DOCUMENTATION

The final dimension on which I will compare the two systems is the material provided for learning to use them. Each system is documented by a book that serves both as introduction and reference. This documentation is augmented by online help describing the provided functions. This help information is also given in printed form as the second half of the S book (Becker, Chambers and Wilks, 1988). Having it there is useful, both since a book is more portable than a computer and because books are still easier to read. The L_S online descriptions of the functions are too terse and do not contain any examples or references to related functions.

Both S and L_S suffer from requiring the user to invest a lot of time and effort before reaching a level of competence adequate even for simple data analysis. One of the biggest problems is that each of them have so many different functions, some with similar purposes, and it is often not obvious which of the choices is the best. The `help*` and `apropos` functions in L_S and the `help` function and the summary section at the end of the S book are useful, but still one of the most frequently asked questions to a local S expert is "I think there is a function to ..., but I can't find it in the manual." There are 836 different functions and keywords that have help information in L_S. In S there are 546 help files in the main help directory, and about as many in the statistics library. In both systems, these functions are all in a single, flat namespace.

The usefulness of the L_S book would be greatly increased by including more complete descriptions of each function, organized either alphabetically as in the S book or by topic as is done in the description of the Common Lisp language (Steele, 1984).

I end with some minor criticisms of the L_S book.

First, there is disconcerting ambiguity throughout the book about which version of Lisp underlies L_S. It would have been better to commit to presenting L_S as it is implemented in XLISP, and then if/when it is released for Common Lisp, this could be accompanied by a document describing the differences. It is confusing that some ideas are presented generally when only one case applies to the current implementation of L_S. Examples of this are the discussion of lexical and dynamic scoping, and tail recursion, neither of which are relevant to L_S.

Second, the coverage of Lisp is varied. For example, though macros are mentioned obliquely in the text, their use is never discussed. This might be have been a concious choice since Abelson and Sussman (1985) deprecate the use of macros in their description of Lisp. But macros play an important part in Lisp programming and should not have been ignored.

Third, the index is not complete (for example, no entry on eval), but this applies to both books.

Fourth, there is too much Lisp code. I am someone who has a high tolerance for reading code and loves nothing better than wallowing around in piles of parentheses, but the density of Lisp code was too much even for me in some of the sections. Much better to my mind would have been to supply all of the Lisp code with the L_S system, and refer to relevant new ideas as they are introduced. This is worst in the last chapter, discussing dynamic graphics examples. But also in some of the earlier chapters, there is too much reliance on presenting code. Sometimes an entire function is presented many times as its evolution is discussed. Again, it would have been better just to present the relevant changes to the function in each new version.

7. CONCLUSION

By providing a broad range of statistical and mathematical primitives and an interpretive language, combined with good graphical facilities, together, these two systems define the state of the art in computing environments for statisticians. Each system is better suited to certain applications, and for data analysis and research, statisticians can only benefit by acquainting themselves with both.

ACKNOWLEDGMENTS

Thanks are due to John Chambers, Rick Becker, Allan Wilks and Luke Tierney who all provided me with valuable comments on this review.

Rejoinder

Luke Tierney

I would like to thank the reviewers for their comments and for their efforts in working through the book and the software, and I would like thank the editor for this opportunity to comment briefly on a few of the issues raised in the reviews.

FUNCTIONALITY AND EXTENSIBILITY

Several of the reviews point out that the basic Lisp-Stat system does not include a wide range of specialized analyses. This is quite deliberate. A major advantage of an extensible system is that it allows experts in using and developing a particular methodology to provide tools for implementing the methodology. If the Lisp-Stat system is found to be useful then, over time, this should lead to a wider set of better tools than can be provided by a single implementor or small group of implementors.

A comparison with the evolution of the S system may be helpful. The basic S system as described in Becker, Chambers and Wilks (1988) also does not directly support a side range of different analyses. But few sites now provide only the basic S system. Most augment it with the facilities of S-Plus, a variety of locally written code, selections of code

Luke Tierney is Professor, School of Statistics, University of Minnesota, Minneapolis, Minnesota 55455.