AT&T Technical Journal
Vol. 64, No. 9, November 1985
Printed in U.S.A.

Design of the S System for Data Analysis*

By R. A. BECKER and J. M. CHAMBERS†

S is a language and system for interactive data analysis and graphics. It emphasizes interactive analysis and graphics, ease of use, flexibility, and extensibility. While sharing many characteristics with other statistical systems, S differs significantly in its design goals, its implementation, and the way it is used. This paper presents some of the design concepts and implementation techniques in S and relates these general ideas in computing to the specific design goals for S and to other statistical systems.

1. BACKGROUND

S is a language and system for the interactive analysis of data, developed at AT&T Bell Laboratories, and currently in use on the UNIX‡ operating system. An extensive user's guide, *S: An Interactive Environment for Data Analysis and Graphics* [7] is available. As of April 1983, about 250 sites had obtained S and over 4,500 copies of the previous user's manual had been distributed. S is being used at universities, research laboratories, and other organizations. While sharing many characteristics with other statistical systems, S differs significantly in its design goals, its implementation, and the way it is used.

The design goal for S is, most broadly stated, *to enable and encourage good data analysis*, that is, to provide users with specific facilities and a general environment that helps them quickly and conveniently look at many displays, summaries, and models for their data, and to follow the kind of iterative, exploratory path that most often leads to a thorough analysis. The system is designed for interactive use with simple but general expressions for the user to type, and immediate, informative feedback from the system including graphic output on any of a variety of graphical devices. In addition, the system is open to change: Even though the current system has many capabilities, a

† Authors are employees of AT&T Bell Laboratories.

‡ UNIX is a trademark of AT&T Bell Laboratories.

Table I—Design features of S

Topic		See Section
Syntax	Expression language	4
	Uniform treatment of function arguments/results	4
	Formal grammar	6
	Precedence	6
Data Structures	Self-describing attribute-value pairs	5
	Hierarchical	5
	Vector structures	5
Implementation	Structured code	9
	Software tools	9
Portability	FORTRAN (as portable assembly language)	9
	Isolation of machine dependencies	9
	Uniform executive/function interface	6
Extensibility	Macroprocessor	7, 9
	User-written functions (interface language)	7
Other	Device-independent graphics	8
	Online documentation	7

variety of ways are available to extend the system as new applications and techniques appear.

The implementation of S draws on a number of modern computing principles and techniques. Table I summarizes some of these. Many, of course, are popular concepts, although few statistical systems apply them together consistently. Some, such as hierarchical data structures, seem to be unique to S among statistical systems. Vector structures and our approach to an interface language are also novel.

Work on S began at Bell Laboratories in 1976; an initial implementation on a large Honeywell mainframe system was in use late that year. Starting in 1978, a version of S was developed for the *UNIX* operating system. Since 1981, this version has been distributed outside Bell Laboratories. S represents both an evolution from earlier statistical computing work at Bell Laboratories, particularly program libraries and graphics software (see [10]), and also our opinions about what was good and bad in the software used for data analysis at the time. (For a more complete description of how S is used in actual data analysis, see [7].)

2. S AND OTHER SYSTEMS

When the design of S began, a group of us at Bell Laboratories considered the then existing statistical software in terms of our goal

of good data analysis, particularly in an interactive, exploratory environment. There were three main approaches to doing statistics on the computer: programming in a conventional language, usually FORTRAN (this had been our own previous approach); mainframe statistical packages such as BMD, SAS, and SPSS; and a few interactive languages, notably APL. We recognized the need for better use of human resources than having to write FORTRAN programs, but found problems with the existing alternatives.

Statistical packages arose in the 1960s and were closely modeled on the idea of sequentially processing a series of records on punched cards or magnetic tape. Relatively recent user guides to BMDP [8] and SAS [16] still picture the user input as a card deck. This model has several bad influences. Good data analysis is highly iterative, responding to important facts observed in the analysis itself. Picturing analysis as processing a sequence of records through a limited set of statistical commands discourages this freewheeling interaction with the data. In particular, interactive use of the statistical packages was either not available or consisted largely of the ability to set up the card deck and run it from a terminal. S, on the other hand, was designed with the model of a language operating on complete data sets, interactively, in a nonsequential manner. A number of modern statistical techniques, e.g., robust estimation, cannot easily be expressed in the sequential form, and are therefore hard to incorporate in some of the packages.

Another result of the batch approach was the tendency to "shotgun" output, printing all the summaries likely ever to be relevant from a particular model or process. Instead, S tries to provide a wide variety of displays, particularly graphical, that can be used interactively to see the summaries that are relevant to the particular user. Graphics, like interaction, was not part of the original design of the mainframe packages. Since 1976, many of them have added graphical facilities; however, the graphics tend to be viewed as "reports," rather than being integrated into the analysis. For example, most of the graphics add-ons do not include graphic *input* which in our opinion is essential for identifying important features observed in the plots.

The APL language, while not designed for statistical computing, offered a very different, and in many ways, more attractive approach. It was intended for interactive use, with users typing expressions that operate on whole data sets and produce immediate output at the terminal. Users can extend the language by defining interpreted "functions" that can then be used in the same way as primitive APL operators. These are all features that contribute to APL's usefulness for data analysis, and which we have incorporated into S. The consistency and functionality of APL's operators is also present in S; however, in S, such operations are normally carried out by functions

rather than operators. The main problems with APL are its syntax, its data structures, and its isolation from other languages. APL has only operators, i.e., functions with one or two arguments, and its precedence rules are different from those of ordinary algebra. For statistical applications, the latter is inconvenient for many users, and the former is a serious drawback. Statistical functions usually have a few main arguments (the data to work on) and any number of additional optional parameters or auxiliary data. They are generally awkward to express as unary or binary operators, as noted in Section 4. In S, we responded by allowing general function calls and by using common algebraic notation for expressions.

The APL data structure is the multiway array, while the result of most statistical functions tends to be less regular. A regression, for example, needs to be described by coefficients, residuals, and summaries of the numerical and statistical methods applied. Fitting this into a single multiway array is unnatural. Allowing completely general, hierarchical data structures in S let the results be expressed naturally, while allowing any data structure to be the value returned by a function hid the structure from users who had no need to extract the pieces explicitly.

The interface to user-written primitive functions discussed in Section 7 allows new functions to be defined when a purely interpretive form would be difficult to write or very inefficient. Both APL and the mainframe statistical packages made the process of interfacing to, say, a new FORTRAN-based algorithm either severely constrained, (e.g., only one user-defined extension) or complicated (involving the implementation details of function interfaces). The substantial number of high-quality algorithms published by journals, such as *Transactions on Mathematical Software* and *Applied Statistics*, makes them an important source of extensions to statistical systems.

Changes in packages and languages since the development of S have often reflected similar concerns to those we felt. Many packages have added graphics and interactive modes. A recent new version of APL moves toward more general data structures. A system built on APL, STATGRAPHICS [24] adds graphics and hides the syntax behind menu-driven interfaces. These are beneficial changes for the users of such systems; however, designing interaction, graphics, and generality in from the beginning makes for a cleaner result.

In retrospect, it is clear that the evolution of S, in many respects, parallels a number of other contemporary computing activities. Our emphasis on user-extensible data structures and operations, and on removing details of data management and implementation from the user is similar to Smalltalk [18]. The approach in S to data structures, dynamic determination of their properties, and a blending of data and

"program" (in macros) has some of the flavor of many LISP-based systems. Speakeasy [17] has some of the S flavor of building an interactive user interface to make mathematical and statistical computations user-friendly, although it is more restrictive in terms of data structures and extensibility.

S represents a growing approach to computing that emphasizes the effectiveness of the *human* as the most important design criterion, as shown by the emphasis on friendly interactive access to computing, on information hiding, and on greater flexibility through delayed binding. Our philosophy is that the effectiveness of the *human* is the most important criterion for design of a computer system.

3. OVERALL ORGANIZATION

An S user types *expressions* that describe the analysis to be done. Some examples are in Table II.

The expressions involve a wide variety of *operators* and *functions* which carry out arithmetic and mathematical operations, statistical analyses, graphics, data manipulation, and other computations. Expressions also use and create *data sets* containing data structures, e.g., vectors, arrays, time series, tables. Data sets are automatically accessed by name. The S *executive* interactively parses expressions and controls their evaluation.

The organization of S resembles that of an interactive operating system: The executive corresponds to a command interpreter, the data sets to files, and the functions to the individual commands. The specific similarity to the UNIX system organization [25] is probably not coincidental, although it was not conscious. There are significant differences, however. The expressions for data analysis need a richer syntax than the commands in an operating system, particularly for algebraic expressions, and data for arguments and results have more structure (for example, commands in the UNIX system operate largely on unstructured streams of bytes).

Table II—Some S expressions

```
  # read a vector of numbers from a file, create data set mydata
mydata ← read("my.data.file")
mydata — mean(mydata)                    # subtract the mean from each value
  # Given a matrix of predictor variables longley.x
  # and a response variable longley.y
  # get the residuals from a multiple linear regression model
r ← regress(longley.x, longley.y)$resid
  # compute the residuals
  # larger than the median absolute residual
r [abs(r) > median(abs(r))]
```

S was designed in a research environment with statisticians who continually develop new techniques, so it was essential that the system be extensible. Some of this extension (macros and new data structures) can be done within the interpretive S language itself. Other extensions involve the creation of new S functions. S includes an *algorithm language* for writing computational algorithms, an *interface language* for describing the interface between the algorithms and the interactive user, and utilities to create new functions. All of these facilities for extension are intended for *users*; they are not restricted to those familiar with the internal workings of S.

4. EXPRESSIONS: THE STATISTICAL LANGUAGE

The user who types expressions into an applications system wants a combination of simplicity and flexibility. Simple requests should be straightforward and brief. At the same time, unusual but sensible requests should not be impossible or unreasonably complicated. Novice and expert users will place different emphasis on the simple or on the unusual.

In S, all user commands follow one general syntax: *Everything is an expression.* The expressions that are given to S may be as short or long as is comfortable for the user.

Expressions in S use functional and algebraic syntax as shown in Table II. (The formal syntax rules are given in Table III.) For users with some background in mathematics, science or engineering, this syntax is readable and familiar. Extensions to ordinary algebraic notation introduce a few special operators, for example, a colon is a sequence operator so that **x:y** is a vector going in steps of ± 1 from **x** to **y**.

When an expression is given to S, it is evaluated. The result may be assigned a name and thus saved as a data set. If the result of an expression is not assigned or used inside another expression, it is printed for the user.

Algebraic notation, i.e., prefix or infix operators, is natural for functions with one or two arguments. However, data analysis quickly becomes involved with functions having many arguments. Functions in S can have arbitrarily many arguments which can be specified positionally or by name. Typical functions to carry out statistical or graphical analysis will have a few arguments to say what data is to be analyzed or plotted as well as many optional arguments to control details. Options are most easily supplied in the form *name=value*; the options of interest can be specified in any order. Functions return data structures that may have arbitrarily many named components; thus, functions may have any number of inputs and produce any number of outputs.

Table III—S grammar rules*

```
expr      : NAME ( arg.list )                          #function call
          | expr OP expr                               #binary operator
          | UNARY expr                                 #unary operator
          | sub.expr [ arg.list ]                      #subset
          | asn.expr ← expr                            #assign or replace
          | expr → asn.expr                            #same, to the right
          | INT                                        #literals
          | REAL
          | STRING
          | sub. expr                                  #can be subsetted
          | asn.expr                                   #can receive assignments
          | control                                    #iteration, conditional, etc.
          ;
asn.expr  : com.name                                   #name or component
          | com.name [ arg.list ]                      #subsetted
          ;
com.name  : NAME
          | com.name $ NAME
          | com.name $ [ expr ]
          ;
sub.expr  : ( expr )                                   #anything in parens
          | NAME ( arg.list )                          #function call
          | sub.expr $ NAME                            #component
          | sub.expr $ [ expr ]                        #numbered component
control   : if ( expr ) expr
          | if ( expr ) expr else expr
          | for ( NAME in expr ) expr                  #iteration
          | while ( expr ) expr
          | repeat expr
          | break                                      #loop control
          | next
          | { exp.list }                               #compound
exp.list  : expr
          | exp.list ; expr
          ;
arg.list  : arg
          | arg.list , arg
          ;
arg       : #empty                                     #empty
          | expr
          | NAME =
          | NAME = expr
          ;
```

* Lexical tokens are capitalized; key words are in boldface.

One of the most powerful functions in the S language is represented by the subscripting operator. Since S deals with vectors, it is natural that subscripts are also vectors. Thus

x[1:5]

returns the first five values in **x**. Since it is frequently necessary to exclude data from statistical analysis, negative subscripts specify the values to be excluded:

x[−6]

returns **x** with the sixth value omitted.

Subscripting can also be used to answer database-like queries. Logical expressions used as subscripts cause the selection of data corresponding to TRUE values in the subscript.

```
name[ salary > 30000 & age < 25 ]
```

The operation extends naturally to multiway arrays, and in this context, an empty subscript denotes all values in that subscript position. For a matrix **y**

```
y[ , 6:2 ]
```

returns all rows of columns six through two. As this example illustrates, the subscript operator can also permute data values (here reordering columns 6 through 2).

The function **order** generates subscripts corresponding to a sorted version of its argument. Thus

```
x[ order(x) ]
```

is equivalent to

```
sort(x)
```

Using **order**, it is possible to do passive sorting simply:

```
name[ order(salary) ]
```

lists names in increasing order of salaries.

The **print** function, implicitly invoked whenever a result is not assigned, represents numerical results to the appropriate number of decimal places and can neatly lay out matrices, time series, multiway tables, and character data.

The function **apply** (similar to "mapfun" in Lisp [23]) is able to invoke another function repeatedly on portions of data structures. In its simplest form, **apply** invokes a function on each of the rows or columns of a matrix. Thus

```
apply( y, 1, "mean" )
```

invokes **mean** once on each row (Dimension 1) of the matrix **y** and returns the vector of row means. With other choices for its second argument, **apply** can deal with slices of multiway arrays. Functions can also be applied over hierarchical data structures and ragged arrays.

5. DATA STRUCTURES AND DATA MANAGEMENT

Data sets in S contain self-describing, hierarchical (list-like) data structures. Data sets are created automatically by assignment expressions; no user control of storage is required. The elementary data structures are *vectors* of numbers, logical values or character strings:

```
> response
  1.01  0.97  3.1  7.21
> response > 2.5
  F  F  T  T
> species.name
  "Setosa"  "Virginica"  "Versicolor"
```

(Here the ">" is the S prompt for an expression.)

The numeric *data modes* are "real" and "integer," but for the most part, the distinction is unimportant to the user. In S, the value of the expression "3/2" is 1.5, even though in many programming languages, integer arithmetic would produce an integer result of 1. A special operator is provided for integer division when it is needed.

There is a special value, NA (not available), which can be used to signify missing data. Any arithmetic operations on NAs produce NAs.

General data structures consist of any number of components, each component being either a vector or another general data structure. Each component has a *component name*; syntactically, the component named **Label** of a structure **z** is denoted **z$Label**.

We designed S so that most users are unaware of the details of data structures, but also so that structures can be defined and manipulated easily to handle new analyses. Simplicity for the user is obtained by having all functions that deal with a given type of data structure (e.g., matrices, time series or tree structures from clustering) recognize the structure type by looking for components with certain specific names. Functions that produce such structures as their value simply return structures with the appropriately named components. For example, a multiway array is defined as a structure with two vector components: one named **Data** containing the data values for the array (listed column by column), and one named **Dim** containing the extents of the array on each dimension. A 2 by 3 matrix, **x**, with data value $2i + j$ in the $[i, j]$ position corresponds to the following list representation:

```
( "x" STR
   ( "Dim" INT 2 3 )
   ( "Data" REAL 3 5 4 6 5 7 )
)
```

Certain functions make use of a list representation of S data structures to enable structures, or entire databases, to be written to files in character form and subsequently read back in.

The ordinary user does not see this structure, however; **x** just appears to be a matrix. When a matrix or array is printed, it is laid out conventionally with no explicit reference to the components of the structure:

```
>x
Array:
2 by 3
     [,1] [,2] [,3]
[1,]    3    4    5
[2,]    5    6    7
```

Matrices and arrays are created and manipulated by a large number of S functions. Data structures such as arrays or time series are so widely recognized that they are considered to be built into the language. In particular, they can be declared as special structures in the interface language (see Section 7) that defines S functions. Most of the basic functions, such as arithmetic, logic, printing, and plotting include some special facilities for treating these structures sensibly. For example, the result of adding together two time series is a time series on the intersection of the two time domains.

A broader special class consists of *vector structures*: data structures that *can* act like vectors, but have special structure in addition. Such structures can be used in arithmetic, and in general, can act as a vector argument to any S function. Arrays and time series are examples of vector structures, but the class is open-ended. Internally, any structure with a vector component named **Data** is considered a vector structure. The **Data** component is the part that acts like a vector when necessary. Functions that operate element-by-element on a vector structure change the data values but leave the other components unaltered. If **x** is the matrix above, **sin(x)** produces a 2 by 3 matrix with data **sin(3)**, etc., and **x<4** is a matrix of logical values. Functions that rearrange the order of elements, on the other hand, throw away the structure and leave just the data: **sort(x)** sorts the data values in the matrix but its result is a simple vector. Since the original design of S, vector structures have been added to represent such structures as distance measures, categorical variables, and multiway tables. These structures can be used as vectors throughout the language with no modification of the various S functions involved.

Other structures may not be interpretable as a vector, but may still be recognized by groups of functions. For example, a tree structure is used to represent the result of hierarchical cluster analysis [22]. Its components are a matrix describing the merging that took place in the analysis, a vector of the distances at which merges took place, and a vector giving the reordering of the original objects needed to plot the tree. S functions exist to produce such trees, plot them, and extract subtrees or nonhierarchical clusterings. Users of these functions need not know how the tree is represented, so long as the various functions agree among themselves.

6. THE EXECUTIVE

The S executive performs tasks roughly comparable to an operating-system command interpreter. It controls most interactions with the user, parses user expressions, schedules the execution of various functions, and handles interrupts and error recovery. The expressions and the data structures in S are considerably more general than the commands and files handled by most operating systems, so that parsing and data transmission in the S executive must be correspondingly more general.

User expressions are accepted by a parser built using a version of the YACC compiler-compiler [19] with a customized lexical analyzer. The use of YACC allows the syntactic rules in the language to be compact and relatively readable; with just a little cleaning up, they are included in Table III. Occasional changes in the syntax are made easier to implement as well.

The result of the parse is a hierarchical structure representing the parse tree; in fact, it is another of the general S data structures. It is evaluated by performing a depth-first traversal of the tree. When parsed, each function invocation becomes a structure with one component for each argument: As the functions are evaluated, the *value* returned by each function replaces this substructure in the parse tree. When the traversal is complete, the parse tree has been replaced by the value of the expression.

Down to function evaluation, the process is essentially portable. However, the process by which the executive invokes an S function is crucially system-dependent. S consists of a large collection of functions (currently around 300). Furthermore, users must be free to write and use their own functions. The facilities of the operating system running S determine how such a collection can be maintained and used in a reasonably efficient way. Table IV lists advantages and disadvantages of various implementation techniques that can be used to execute functions. Operating system constraints have forced us to use several different strategies. For the original version, on a Honeywell computer with a relatively primitive operating system (no virtual memory or process control), we wrote our own dynamic loader. Each S function was an overlay, read in by the executive; control was passed by a standardized transfer vector.

Running on 16-bit hardware without virtual memory, the major constraint is that program address space is limited. For this environment, we implement S functions as independent programs. Control of execution and data transmission, respectively, are handled by simple process-signaling facilities and by file input/output.

The implementation on 32-bit hardware exploits the larger address space to incorporate some, or all, of the S functions as part of the

Table IV—Advantages and disadvantages of implementation of S executive

Overlay Loading	
Common code resides in the executive.	Execution-time overlay loading is relatively slow. Many loaders do not allow partial linking—the entire system must be linked at one time. Tools are required to allow user-written overlays. Functions which get into trouble may harm the executive.
Dynamic Linking	
The operating system does most of the work.	Most operating systems do not have this facility. Functions which get into trouble may harm the executive. First-time invocation is slow because of resolving external references.
Large Virtual Memory Executive	
The operating system does most of the work.	Functions which get into trouble may harm the executive. The entire system must be linked together. Recursive calls involving the **apply** function may be needed. A facility for user-written functions is necessary.
Independent Processes	
It is easy to create functions—user and system—since they are independent. The executive is insulated from functions.	Common object code in many modules means large disk storage for executables; it is hard to correct bugs in common code. Function invocation is slow due to process creation/startup. Interprocess communication is needed.

program containing the executive. We speculate that an ideal environment may be one in which dynamic linking is combined with virtual-memory facilities, although we have no experience with such a system.

For our goals of flexibility and extensibility, it is essential that these changes in implementation affect only the executive, not the source code for the individual functions. Even in the executive, only a relatively small fraction of the code is system-dependent. However, this code has an importance to the reliability and efficiency of the system much greater than its size. Adapting the control of such a large-application software system to the features of an interactive computer system is likely to be the most important and difficult implementation step.

7. FUNCTIONS: ALGORITHMS AND INTERFACES

There are basically two ways to extend the S language to perform

Table V—Comparison of macros and functions

Macros
Easy to write
Little programming skill needed
Uses existing S functions—a limitation
Slow execution
Functions
Programming needed
Harder to create than macros
Creation process is slower
Unlimited flexibility
Fast execution

new computations: macros and functions. A comparison of macros and functions is included in Table V.

Macros combine existing S expressions into a named entity. They are easy to write and are often useful for providing sets of operations for specific applications. As an example, consider a macro designed to produce a scatter plot which includes a fitted regression line:

```
MACRO line(x, y)
plot(x, y)
abline(reg(x, y))
END
```

The body of the macro, with appropriate parameter substitutions, is inserted when the user types an expression such as

```
?line(x, y)
```

Details of macro writing are given by Becker ([7], Chapter 6). The macroprocessor itself is an extension of the M4 processor [21].

Some operations are difficult or inefficient to carry out using existing S functions. Sometimes, new algorithms appear that would be useful in S. In these cases, it is desirable to write a new S function.

All S functions are defined by *interface* routines. The interface routine describes the arguments that the user may supply, how these arguments are to be interpreted, and what default actions will be taken when arguments are omitted. It checks for errors in the arguments that would prevent successful execution. It allocates space dynamically for data structures needed for the value of the function or for temporary storage. It invokes *algorithms* to do the actual computation (numerical, graphical, etc.) and returns the appropriate result. The interface routine is written in an interface language which is compiled using FORTRAN as an intermediate language. The function **gs**, for example, takes a matrix and uses a Gram–Schmidt algorithm, which could be written in the algorithm language, FORTRAN or C, to

decompose it into the product of an orthogonal matrix **q** and an upper-triangular matrix **r**. The corresponding interface routine is:

```
FUNCTION gs(x/MATRIX/)
STRUCTURE( q/LIKE(x)/,
  r/MATRIX,NCOL(x),NCOL(x)/ )
call gs(n,NROW(x),NCOL(x),q,r)
RETURN(r,q)
END
```

The first line says that the function has one argument and that this argument should be interpreted as a matrix, if possible. The next statement allocates two structures whose sizes are determined at execution time, the fourth line invokes an algorithm (subroutine) to do the calculations, and the fifth line specifies the result of the function as a structure with the two components **q** and **r**.

It is important to our design that interface routines be as simple and readable as possible. Users, not just system programmers, should write them. Most interface routines are longer than this example, since most S functions have several arguments. However, more detailed or complex interface routines are usually still readable.

S also provides a tool to create a partial documentation file from an interface routine. This helps ensure accuracy in documenting arguments and results, and also makes it easier for the author of a function to document it in a manner consistent with the standard S online documentation. (A similar tool is also available for user documentation of macros and data sets).

The interface routine is also an interface between the S data structures and the vectors and arrays that the FORTRAN-based algorithms can handle. Typically, an algorithm will require both the *data* part of a structure and some *attributes* such as the length of a vector or the number of rows and columns of a matrix. These attributes are implicit in the self-describing S data structures and are supplied to algorithms by attributes defined through the interface language.

An extensive set of computational algorithms is required to support a large collection of functions. The algorithms need to be of good quality, they should if possible *not* be restricted to use within S itself, and the human effort required for producing and maintaining them should be kept small. In particular, we take advantage of published or otherwise generally available algorithms. Our algorithms are generally written in or converted to our *algorithm language.* This language is a combination of RATFOR [21] or EFL [15] with a set of macros to provide machine constants, error handling, user messages, dynamic storage, and various specialized facilities. Algorithms can be converted mechanically from FORTRAN into RATFOR [1], but additional effort

is usually required to make error handling, use of scratch space, and other environmental interfacing compatible. Ideally, this does not require knowledge of the details of the algorithm.

In addition to imported algorithms, S contains a large collection of graphics routines, facilities to support reading and printing of data, and algorithms to handle the S data structures.

Our experience shows that a large body of algorithms to support a scientific system can be developed by a small programming group if: (1) advantage can be taken of the many existing algorithms: and (2) there is a good collection of tools to make error handling, dynamic storage, and other support features easy to implement.

8. GRAPHICS

Data analysts use plots iteratively as an intimate part of their study of data. The unique role of plots comes from their information content: No other form of output conveys so much information so quickly. Users often react to plots by finding the unexpected and using this new information to shape the subsequent analysis. A variety of graphical techniques for data analysis is presented in a recent book [11].

S emphasizes interactive graphics as one of the most important tools in data analysis. Graphics functions in S provide the simple displays that are predominant in statistical graphics, most notably the scatter plot, in a flexible and easy-to-use form. For example:

```
plot(x,y)          # scatter plot
qqnormal(x)        # Normal probability plot
```

The general data structures and expressions in S help to provide graphical output from a variety of sources. Many statistical analyses produce results that define a scatter plot; for example, a *probability plot* [11] shows an ordered set of data plotted against corresponding quantiles of a probability distribution. Deviations from a straight-line pattern help assess distributional assumptions. Rather than duplicating scatter-plot software for each such plot, S functions return as their value a *plotting data structure*, which is passed automatically to the **plot** function to be displayed. The expression

```
qqnorm(mydata)
```

produces a probability plot of **mydata** against quantiles from the standard normal distribution. Internally, **qqnorm** only generates the plotting data structure and then invokes the scatter-plot function; **qqnorm** need know nothing about plotting. The data structure consists of two vector components for the **x** and **y** coordinates of the points to plot. Once the probability plot is seen as a data structure, it

is straightforward to use this structure for further analysis, for example, by fitting some suitable line to the points in the plot.

While statistical graphics characteristically make wide use of simple plots, such as the scatter plot, there is also the need to develop and use other special displays. The interface language and underlying graphical algorithms provide support to make the writing of new graphical functions straightforward. There are high-level graphical algorithms for generating complete displays such as scatter plots, low-level algorithms to produce components of a plot (lines, points, axis labeling, etc.) and a set of macros in the algorithm language to specify and query *graphic parameters* (Table VI) controlling details of how plotting is to be done. Parameters include specifics such as the symbol to use for a scatter plot, and generalities such as the style of axis labeling. Parameters can also be controlled by the user of S, by supplying them as optional arguments to S graphics functions. For example, a scatter plot in (device-dependent) color 6 with "O" as the plotting character could be produced by

```
plot(x,y,col=6,pch="O")
```

The goal, again, is to provide the maximum flexibility to both the interactive user and the writer of new S functions, while still keeping the ordinary use of plotting simple. Graphic input is supported in S and is important when data values or areas of interest are to be identified on the plot, as a guide to further analysis.

The graphical functions are *device-independent* in that both the user-typed expression and the underlying interface routines and algorithms are written independently of specific graphic devices. Actual graphical output is produced through a *device driver* which converts the graphics output, at a relatively low level, into commands for a particular device (see Figure 1). Drivers exist for devices including ordinary printing terminals and a range of interactive plotting terminals. A driver is written by implementing routines to carry out a specified set of graphic primitives (e.g., draw a line or plot a character),

Table VI—Some graphical parameters

User (world) coordinate system
Viewport
Plot aspect ratio
Size of margin surrounding plot (used for labelling, titles, etc)
Color
Line style
Character rotation
Character size
Plotting character
String justification (left, right, center)

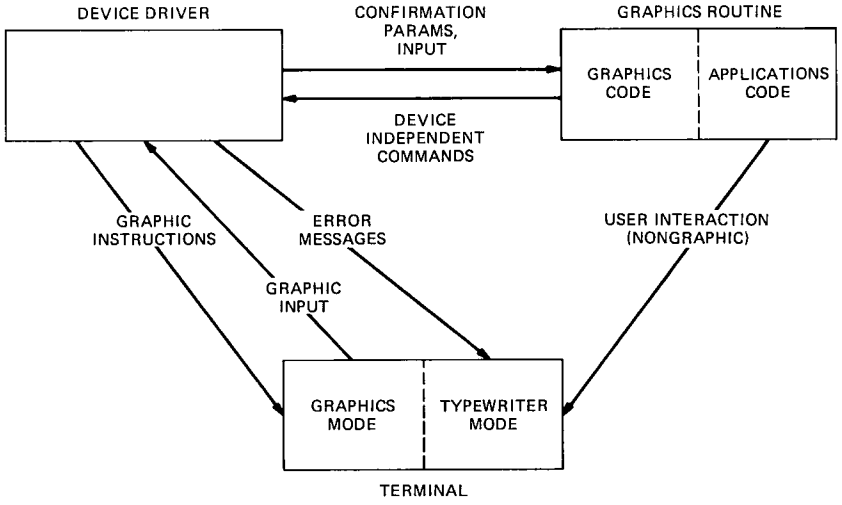

Fig. 1—Operation device-independent graphics.

and by providing a definition of the device in terms of basic graphic parameters (e.g., the device coordinate system, raster size). Incorporating a new device typically takes a few days or less; the process is sufficiently straightforward that we include instructions in the user's manual. Users can write their own device drivers.

The intimate role of graphics in interactive data analysis means that users should have interactive graphic terminals available locally. The terminals must not be too expensive and they should be of sufficient quality that the detailed information in many statistical plots can be seen clearly. Both scope terminals and pen plotters are popular with some users (mainly for rapid plotting and good graphical quality, respectively).

Our graphics routines have much in common with the CORE graphics standard [13], although our work was independent [2, 3, 5]. Relative to CORE, graphics for data analysis consists mostly of applications software; the quantity of device support software is not large. A number of the more elaborate features of the standard, on the other hand, are not often used in data analysis, e.g., retained segments. A CORE implementation would be more than sufficient to define one of our device drivers, but would not provide much of our device-independent support code.

9. TOOLS: THE OPERATING SYSTEM

The discussion so far has shown the basic design of S and the components of its implementation: execution of user's expressions; data structures and management; source code for interface routines

and algorithms. The complete system contains about 6,000 lines of interface language, 35,000 lines of algorithm language, and 9,000 lines of C code. Development and maintenance of S by a small group of people requires efficient use of time. Our experience is that three aspects of the design particularly affect human efficiency: the *languages* in which programming is done, the *tools* for maintaining the application system, and the *operating system interface.*

The approach to language was described in Section 7. Developing our own interface language and algorithm language may have taken perhaps 10 to 15 percent of the total effort, but this development has been cost-effective. If interface routines were written directly in a general language like FORTRAN, they would be much more complicated and error-prone, and all but the most sophisticated users would find it impossible to write their own S functions. During compilation, an interface routine typically expands into a much larger FORTRAN routine (an order of magnitude more lines of source code). Much of this expansion reflects inherent clumsiness in using FORTRAN to express the argument processing, dynamic storage management, and generation of results in an S function. At the same time, the use of FORTRAN as an intermediate language is important. We could not reimplement all the basic statistical algorithms previously written in FORTRAN.

The use of software tools is essential for creating and maintaining a system such as S. Compiler-compilers, macroprocessors, and more specialized tools ease the burden of system development. The interface language goes through our own simple compiler, two passes of the M4 macroprocessor, RATFOR, and FORTRAN during compilation. Obviously, we are not trying to optimize compilation time. However, this multistep process leaves us able to modify individual steps as our needs change.

Other tools are used to provide specific utilities for S developers. The *make* system for maintaining programs [14] is used to generate the S executive and the individual functions.

For tools to be useful in large applications systems, they should themselves be easily adaptable. For example, our use of *make* is highly specialized. The interface routines and the support programs, whether based on RATFOR or on the C language, all take advantage of special S facilities. We therefore replace and extend *make*'s built-in rules for compiling to include these special features. The result is a customized tool of our own (itself built from a number of tools).

The ease with which tools are put together is also a function of the operating system environment. The UNIX environment is convenient for developing a system such as S, both because of specific facilities and because the operating system tries not to be unnecessarily restric-

tive. Facilities such as pipes and a flexible command interpreter make the creation of customized tools much easier. The *absence* of complex rules about file formats and interprocess protocols, on the other hand, has meant fewer barriers to our implementation.

The dependence of the current version of S on its operating system environment involves both the internal dependencies and the use of operating system features in the tools. The dependencies on computer *hardware*, such as machine accuracy, are relatively easy to handle. The large majority of S code passes through FORTRAN during compilation. Nonportable features such as the choice of special characters and machine precision are isolated in the macroprocessing phase and kept in a single file.

The use of FORTRAN as an intermediate language and the parametrization of machine dependencies make the S source code quite portable. On the other hand, implementing and using a system like S benefits from a good general computing environment. Among current computer systems, the UNIX system is relatively well-designed, allowing us to combine and modify tools to put together a system like S. In a more restrictive system, we would be obliged to provide more of the support environment. We also observe substantial interest in porting the UNIX system to a variety of new computer systems, and when that is done, S goes along for free.

10. EXPERIENCE AND EVOLUTION

A significant fraction of the evolution of S has come from users' activities and experience. By far, the majority of our users are not professional statisticians. Instead, they are professionals in other areas with a need for data analysis, graphics or other S facilities to enhance their own work. In a number of cases, their specialized use of S has led them to develop, in effect, specialized systems for their own user community, built on S. This is usually done by creating a set of S macros to translate users' requests from the terminology of specific applications into the S expressions generating the results. Less frequently, more ambitious projects may include user-written S functions or interfaces between S and other large application systems. The relative simplicity of writing S macros means that a new system tailored to a particular user community can be written with a fraction of the programming effort required for a corresponding project using a general programming language. Also, the system development effort grows naturally (often unintentionally at first) out of direct use of S to solve the user's problems; there is no large initial investment in programming before any of the proposed uses can be tested and evaluated.

One of the more difficult tasks in user training has been convincing FORTRAN programmers that it is ordinarily not necessary to write explicit loops to operate on collections of data. Most of the nonprogrammers, however, seem to have little difficulty with the implicit iteration provided by S.

Specific user suggestions and our general recognition of the pattern of use have contributed many of the enhancements in S. An early user suggestion was the inclusion of a right-facing assignment arrow, →, for the occasion when one decides to save a result *after* typing a long expression. Our use of tools like syntax-driven parsing makes such changes easy. Interestingly, the interactive use of S in this case has direct implications for the syntax. Other enhancements in response to user needs include: a simple mechanism to edit and rerun expressions after errors; a "diary" feature to provide a history of the expressions executed during a session; tools to help users create online documentation for their macros, data sets, and new S functions; and facilities for moving large collections of S data sets among different machines in a portable way. We have also provided a mechanism for running S noninteractively for large or repetitive analyses, and a technique for creating device-independent graphics metafiles which can be plotted later on interactive or batch devices. The ability to provide such facilities with a limited expenditure of our own time derives from our modular, tool-oriented design and from the similar orientation of the UNIX environment.

11. FUTURE PLANS

Future plans for S concentrate on improving the human interface, particularly for use with the new generation of work stations. These offer substantial local computing power, high-quality graphics, and often, novel forms of interface using multiple windows and input devices such as the "mouse." These features allow design of nonprogramming interfaces to statistical systems with greater flexibility and more sophisticated user support than previously possible. We also plan to make the underlying S language itself more efficient and to simplify the user's view of writing new functions.

REFERENCES

1. Baker, B. S. An algorithm for structuring flowgraphs. *J. ACM 24,* (Jan. 1977), 98–120.
2. Becker, R. A. and Chambers, J. M. On structure and portability in graphics for data analysis. In: *Proceedings of the Ninth Interface Symposium on Computer Science and Statistics,* 1976.
3. Becker, R. A. and Chambers, J. M. GR-Z: A system of graphical subroutines for data analysis. In: *Proceedings of the Tenth Interface Symposium on Computer Science and Statistics,* National Bureau of Standards Special Publication 503, 1977, 409–415.

4. Becker, R. A. and Chambers, J. M. Design and implementation of the S system for interactive data analysis. In: *Proceedings COMPSAC 78,* IEEE 78CH1338-3C, 1978, 626–629.
5. Becker, R. A. and Chambers, J. M. Computer graphics for interactive statistics. In: *Proceedings of NCGA,* 1980.
6. Becker, R. A. and Chambers, J. M. *S: A Language and System for Data Analysis,* Bell Laboratories, Jan. 1981.
7. Becker, R. A. and Chambers, J. M. *S: An Interactive Environment for Data Analysis and Graphics,* Belmont, CA: Wadsworth, 1984.
8. *BMDP Statistical Software: 1981,* Dixon, W. J., ed., Berkeley: University of California Press, 1981.
9. Chambers, J. M. *Computational Methods for Data Analysis,* New York: John Wiley & Sons, 1977.
10. Chambers, J. M. Statistical computing: History and trends. *The American Statistician 34,* 4 (Nov. 1980), 238–243.
11. Chambers, J. M., Cleveland, W. S., Kleiner, B., and Tukey, P. A. *Graphical Methods for Data Analysis,* Belmont, CA: Wadsworth, 1983.
12. Cleveland, W. S. Robust locally weighted regression and smoothing scatterplots. *J. American Statistical Association 74,* 829–836.
13. CORE, Status report of the graphics standards planning committee. *Comput. Gr. 13,* 3 (Aug. 1979).
14. Feldman, S. I. Make—A program for maintaining computer programs. *Software—Practice and Experience 9,* (1978), 255–265.
15. Feldman, S. I. Bell Laboratories Memorandum: The programming language EFL. 1979.
16. Helwig, J. T. *SAS Introductory Guide,* Raleigh, NC: SAS Institute, Inc., 1978.
17. Hynes, G. C. *Speakeasy User's Manual,* Tech. Rpt. BDX-613-2569, Bendix Corporation, 1981.
18. Ingalls, D. H. H. Design principles behind smalltalk. *Byte 6,* 8 (Aug. 1981), 286–298.
19. Johnson, S. C. and Lesk, M. E. Language development tools. *Bell Syst. Tech. J. 57,* 6 (July–Aug. 1978), 2155–2175.
20. Kernighan, B. W. RATFOR—A preprocessor for a rational Fortran, *Software—Practice and Experience 5,* (1975), 395–406.
21. Kernighan, B. W. and Plauger, P. J. *Software Tools,* Reading, MA: Addison-Wesley, 1976.
22. Mardia, K. V., Kent, J. T., and Bibby, J. M. *Multivariate Analysis,* London: Academic Press, 1979.
23. McCarthy, J. et al., *LISP 1.5 Programmer's Manual,* Cambridge, MA: MIT Press, 1962.
24. Polhemus, N. W. Interactive statistical graphics in APL: Designing a versatile user-efficient environment for data analysis. In: *Computer Science and Statistics: Proceedings of the 14th Symposium on the Interface,* New York: Springer-Verlag, 1983, 10–19.
25. Ritchie, D. UNIX: A retrospective. *Bell Syst. Tech. J. 57,* 6 (July–Aug. 1978), 1947–1970.

CR Categories and Subject Descriptors: D.3.2 [**Programming Languages**]: Language Classifications—*very high-level languages*; G.3 [**Mathematics of Computing**]: Probability and Statistics—*statistical computing, statistical software*; I.3.4 [**Computer Graphics**]: Graphics Utilities—*application packages*; I.3.6 [**Computer Graphics**]: Methodology and Techniques—*device independence*; J.2 [**Computer Applications**]: Physical Sciences and Engineering—*mathematics and statistics*
General Terms: Algorithms, Languages
Additional Key Words and Phrases: data analysis

Received 7/82; revised 9/83; accepted 11/83

Authors' Present Address: Richard A. Becker and John M. Chambers, Statistics and Data Analysis Research Department, AT&T Bell Laboratories, 600 Mountain Avenue, Murray Hill, NJ 07974.