A Brief History of S

Richard A. Becker

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

INTRODUCTION

The S language has been in use for more than 15 years now, and this appears to be a good time to recollect some of the events and influences that marked its development. The present paper covers material on the design of S that has also been addressed by Becker and Chambers (1984b), but the emphasis here is on historical development and less on technical computing details. Also, many important new ideas have come about since that work. Similarly, parts of Chambers (1992) discuss the history of S, but there the emphasis is on very recent developments and future directions.

Why should anyone care about the history of S? This sounds like the question people ask of history in general, and the answer is much the same — the study of the flow of ideas in S, in particular the introduction and dropping of various concepts and their origins, can help us to understand how software in general evolves. Some of the ideas that are so clear today were murky yesterday; how did the insights arise? What things were done well? What conditions helped S to grow into the strong system that it is now? Could we have anticipated the changes that took place and have implemented years ago something more like the current version of S?

This history of S should also be of interest to people who use the commercially available S-PLUS language, because S and S-PLUS are very closely related.

THE EARLY DAYS (1976-80 VERSION OF 'S')

S was not the first statistical computing language designed at Bell Laboratories, but it was the first one to be implemented. The pre-S language work dates from 1965 and is described in Becker and Chambers (1989). However, because it did not directly influence S, we will not elaborate here.

S grew up in 1975-1976 in the statistics research departments at Bell Laboratories. Up to that time, most of our statistical computing, both for research and routine data analysis work, was carried out using a a large, well-documented Fortran library known as SCS (Statistical Computing Subroutines). The SCS library had been developed in-house over the years to provide a flexible base for our research. Because the statistics research departments were charged with developing new methodology, often to deal with non-standard data situations, the ability to carry out just the right computations was paramount. Routine analysis was rare.

Commercial software did not fit well into our research environment. It often used a ''shotgun'' approach — print out everything that might be relevant to the problem at hand because it could be several hours before the statistician could get another set of output. This was reasonable when computing was done in a batch mode. However, we wanted to be able to interact with our data, using Exploratory Data Analysis (Tukey, 1971) techniques. In addition, commercial statistical software usually didn't compute what we wanted and was not set up to be modified. The SCS library provided excellent support for our simulations, large problems, monte-carlo, and non-standard analyses.

On the other hand, we did occasionally do simple computations. For example, suppose we wanted to carry out a linear regression given 20 *x,y* data points. The idea of writing a Fortran program that called library routines for something like this was unappealing. While the actual regression was done in a single subroutine call, the program had to do its own input and output, and time spent in I/O often dominated the

actual computations. Even more importantly, the effort expended on programming was out of proportion to the size of the problem. An interactive facility could make such work much easier.

It was the realization that routine data analysis should not require writing Fortran programs that really got S going. Our initial goals were not lofty; we simply wanted an interactive interface to the algorithms in the SCS library. In the Spring of 1976, John Chambers, Rick Becker, Doug Dunn and Paul Tukey held a series of meetings to discuss the idea. Graham Wilkinson, who was a visitor at the time, also participated in these early discussions. Gradually, we came to the conclusion that a full language was needed. The initial implementation was the work of Becker, Chambers, Dunn, along with Jean McRae and Judy Schilling.

The environment at Bell Laboratories in 1976 was unusual and certainly influenced the way S came together. Our computer center provided a Honeywell 645 computer running GCOS. Almost all statistical computing was done in Fortran, but there was a wide range of experience in using Fortran to implement portable algorithms, as well as non-numeric software tools. For example, the PFORT Verifier (Ryder & Hall, 1974) was a program written in Fortran to verify the portability of Fortran programs. Portability was a big concern then, since the Honeywell configuration was unlikely to be available to others and since the investment we made in software was considerable.

Bell Laboratories in 1976 also provided a unique combination of people, software tools, and ideas. John Chambers had been at Bell Laboratories for 10 years, had experimented with many approaches to statistical computing, and had codified his thoughts in a book (Chambers, 1977). Rick Becker came to Bell Labs in 1974 with experience in interactive computing from work with the Troll (NBER, 1974) system. The statistics research departments were heavily influenced by John Tukey and his approach to Exploratory Data Analysis (Tukey, 1971). Computer science research at Bell Laboratories was still actively working on the UNIX® Operating System (Ritchie and Thompson, 1978) and many of the tools from that work were available on GCOS. In particular, we had access to the C language, based on the Portable C Compiler (Ritchie, et. al. 1978), the YACC and LEX tools for parsing and lexical analysis (Johnson and Lesk, 1978), the Ratfor language for writing structured Fortran (Kernighan, 1975), Struct, for turning Fortran into Ratfor (Baker, 1977), and the M4 macro processor (Kernighan and Ritchie, 1977).

An important precondition for an interactive computing system was a good graphics facility. Becker and Chambers (1976, 1977) (and Chambers, 1975) wrote a subroutine library called GR-Z to provide flexible, portable, and device-independent graphics. It used a computer-center supplied library, Grafpac, to produce output on batch devices such as microfilm recorders and printer/plotters. Interactive devices, such as Tektronix graphics terminals, line printers, and others were supported by device driver subroutines written for GR-Z. Its graphical parameters controlled graph layout and rendering, and were an extension of those provided by Grafpac. The system provided for graphic input as well as output, and became the cornerstone for graphics in S.

Another enabling experiment was one in separate compilation of portions of a system. Development of a statistical computing system under GCOS required the use of overlays, where portions of the instructions were brought into memory when they were needed. Unlike modern virtual memory systems, space on the Honeywell machine was severely limited, and compilation and linking was slow. Development of the system was likely to require lots of work on the various S functions, and it would be unacceptably slow and expensive to recompile or relink the entire system for every change in one of the functions. To get around this, we developed a pair of assembler language routines that implemented transfer vectors. The main program would ensure that jump instructions to various subroutines were placed at known fixed memory locations. Overlays could then be compiled and linked separately, knowing exactly where they could find various needed subroutines. Similarly, the overlays had a vector of transfer instructions to the overlay's functions stored in another set of fixed addresses. Thus, the main program could always be sure of the address to use when calling the subroutines. This separate compilation/linkage process fit right in with the notion that S would be based on a collection of functions; each function or related group of functions could reside in a single overlay.

A feature of the computing environment on GCOS was the QED text editor (Ritchie, 1970). It was used as a command interpreter, doing textual computations to build an operating system command and finally executing it. Thus, QED provided a primitive analog to the Unix system shell command interpreter. QED was used extensively behind the scenes, to control compilations and overlay loading; it provided the scaffolding that allowed us to build S.

Initial S Concepts

What were the basic ideas involved in S? Our primary goal was to bring interactive computing to bear on statistics and data analysis problems. S was designed as an interactive language based entirely on functions. A few of these functions would be implemented as prefix or infix operators (arithmetic, subscripting, creating sequences), but most would be written with a function name followed by a parenthesized argument list. Functions would be allowed an arbitrary number of positional and keyword arguments and would return a data structure that would allow a collection of named results. Arguments that were left out would have default values. The basic data structures would be vectors and a hierarchical structure, a combination of vectors and/or other such structures (Chambers, 1978). Entire collections of data would be referred to by a single name.

The power of S and its functions would never be used, however, unless the system were extremely flexible and provided the operations that our colleagues wanted. In order to access the computations that were presently available as subroutines in the SCS library (and those that would later be written) we planned to import subroutines written in a lower-level programming language (Fortran) and we planned to allow ordinary *users* to do the same thing. Of course, to import an algorithm would require some sort of interface, to translate between the internal S data structures and those of the language, as well as to regularize the calling sequence. Our model for this was a diagram that represented an algorithm as a circle, with a square interface routine wrapped around it, allowing it to fit into the square slot provided for S functions.

Another important notion that came early was that the language should be defined generally, based on a formal grammar, and have as few restrictions as possible. (We were all tired of Fortran's seemingly capricious restrictions, on the form of subscripts, etc.) English-like syntax had been tried in many other contexts and had generally failed to resemble a natural language in the end, so we were happy to have a simple, regular language. Other applications languages at the time seemed too complicated — although a formal grammar might describe the basic expression syntax, there were generally many other parts. We wanted the expression syntax to cover everything. General functions and data structures were the key to this simplicity. Functions that could take arbitrary numbers of arguments as input, and produce an arbitrarily complex data structure as output should be all the language would need. Infix operators for arithmetic and subscripting provided very natural expressions, but were simply syntactic sugar for underlying arithmetic and subscripting functions.

From the beginning, one of the most powerful operations in S was the subscripting operator. It comes in several flavors. First, a vector of numeric subscripts selects corresponding elements of an object. Even here there is a twist, because the vector of subscripts can contain repeats, making the result longer than the original. Negative subscripts may be an idea that originated with S: they describe which elements should *not* be selected. Logical subscripts select elements corresponding to `TRUE` values, and empty subscripts select everything. All of these subscripting operations generalize to multi-way arrays. In addition, any data structure can be subscripted as if it were a vector.†

S began with several notions about data. The most common side effect in S is accomplished by the assignment function; it gives a name to an S object and causes it to be stored. This storage is persistent — it lasts across S sessions.

The basic data structure in S is a vector of like-elements: numbers, character strings, or logical values. Although the notion of an *attribute* for an S object wasn't clearly implemented until the 1988 release, from the beginning S recognized that the primary vector of data was often accompanied by other values that described special properties of the data. For example, a matrix is just a vector of data along with an auxiliary vector named `Dim` that tells the dimensionality (number of rows and columns). Similarly, a time series has a `Tsp` attribute to tell the start time, end time, and number of observations per cycle. These vectors with attributes are known as *vector structures*, and this distinguishes S from most other systems. For example, in APL, everything is a multi-way array, while in Troll, everything is a time series. The LISP notion of a property list is probably the closest analogy to S attributes. The general treatment of vectors with other attributes in S makes data generalizations much easier and has naturally extended to an object-oriented

---

†Later, for arrays, we added one further twist: a matrix subscript with k columns can be used to select individual elements of a k-way array.

implementation in recent releases (but that's for later).

Vector structures in S were treated specially in many computations. Most notably, subscripting was handled specially for arrays, with multiple subscripts allowed, one for each dimension. Similarly, time series were time aligned (influenced by Troll) for arithmetic operations.

Another data notion in the first implementation of S was the idea of a hierarchical structure, which contained a collection of other S objects.† Early S structures were made up of named *components*; they were treated specially, with the ''$'' operator selecting components from structures and functions for creating and modifying structures. (There was even a nascent understanding of the need to iterate over all elements of a structure, accomplished by allowing a number instead of a name when using the ''$'' operator.) By the 1988 release we recognized these structures as yet another form of vector, of mode *list*, with an attribute vector of names. This was a very powerful notion, and it integrated the treatment of hierarchical structures with that of ordinary vectors.

Implementation

Persistent data in S was stored in a system known as *scatter storage*, implemented in routines on the SCS library by Jack Warner. Earlier projects in statistics research had needed what amounted to a large flat file system, held in one operating-system file. Scatter storage provided that: give it a hunk of data, its length, and a name and it would store the data under the name. Given the name, it could retrieve the data.

In memory, S objects were stored in dynamically allocated space; S used SCS library Fortran-callable utilities that enabled a program to request space from the operating system and use that space for holding Fortran arrays.

The implementation was done in Fortran for many reasons. Primarily, there was little other choice. C was only beginning to get a start on the GCOS system and at the time, very little numerical work had been done in C. Assembler language was obviously difficult and non-portable. Also, by using Fortran, we were able to make good use of the routines already available on the SCS library. S was already a formidable implementation job and in the beginning we wanted to make the best use of the already-written components. (As S has been re-implemented, all of the reliance on SCS and most of the Fortran has gone away.) On the other hand, even though we were using Fortran, we were loath to give up modern control structures (IF-ELSE) and data structures. As a compromise, we used Ratfor along with the M4 macro processor to give us an implementation language that was quite reasonable, especially before the advent of Fortran 77; we called it the *algorithm language*.

As S was being built, there was a long-held belief at Bell Labs in the importance of software portability and we made a number of attempts to provide it. The Fortran code we used was intended to be portable, not only for numerical software, but also for character handling, where M4 was used to parameterize machine-specific constants such as the number of characters that could be stored in an integer. The GCOS environment certainly encouraged thoughts of portability, because it provided, on one machine, two different character sets (BCD and ASCII) and two different sets of system calls (for batch and time-sharing). To provide even more for portability, we tried to write code that would encapsulate operating-system dependencies: file opening, memory management, printing.

Because S is based so heavily on functions that carry out particular computational tasks, we realized as soon as we had a parser and executive program that it would be necessary to implement lots of functions. Although many of the functions had at their core an SCS subroutine, it soon became apparent that implementation of the necessary interface routines was repetitive, error-prone, and boring. As a result, a set of M4 macros, the *interface macros*, was developed to help speed the implementation of interface routines. We also developed some new algorithms for S; many of them, involving for example, linear algebra and random numbers, reflected ideas described in Chambers, 1977.

From the beginning, S was designed to provide a complete environment for data analysis. We believed that the raw data would be read into S, operated on by various S functions, and results would be

† In this discussion we will refer to S data structures as *objects*, even though this nomenclature came later. The connotations of the word are appropriate: an S object can contain arbitrary data, is self-describing, and stands on its own without requiring a name.

produced. Users would spend much more time computing on S objects than they did in getting them into or out of S. This is reflected in the relatively primitive input/output facilities for S. Basically, one function was provided to read a vector of data values into S from a file, and no provisions were made for accommodating Fortran-style formatted input/output.

From the beginning, S included interactive graphics. This was implemented by the GR-Z software. A separate overlay area was provided for the graphics device functions, so that the S user could perform appropriate graphics on a graphics terminal or at least line-printer style graphics on a character terminal.

Another early decision was that S should have documentation available online. A *detailed documentation* file was written for each function and described arguments, default values, usage, details of the algorithm, an example, and cross references to other documentation.

The S executive program was a very basic loop: read user input, parse it according to the S grammar, walk the parse tree evaluating the arguments and functions, and then print the result. The S executive used 20K 36-bit words, and the functions were executed in a 6K-word overlay area. This gave a reasonable time-sharing response time on GCOS for datasets of less than 10,000 data items.

Internal data structures were based on a 4-word header for S vectors, that gave a pointer to a character string name, a numeric code for the mode of the data, its length, and a pointer to a vector of appropriate values (Chambers, 1974). These headers are still around in recent implementations, although they no longer reflect the true data structure, since they make no explicit provision for attributes.

A Working Version of 'S'

By July, 1976, we decided to name the system. Acronyms were in abundance at Bell Laboratories, so it seemed sure that we would come up with one for our system, but no one seemed to be able to agree with any one else's suggestion: Interactive SCS (ISCS), Statistical Computing System (too confusing), Statistical Analysis System (already taken), etc. In the end, we decided that all of these names contained an 'S' and, with the C programming language as a precedent, decided to name the system 'S'. (We dropped the quotes by 1979).

By January, 1977, there was a manual describing Version 1.0 of S that included 30 pages of detailed documentation on the available functions. People who computed on the GCOS machine at Murray Hill were invited to use S, and a user community gradually began to form.

Work on S progressed quickly and by May, 1977, we had a revised implementation that used a new set of database management routines. Version 2.0 was introduced about a year later and included new features, such as missing values, looping, broader support for character data, and many more functions.

The earliest versions of S worked reasonably well and were used for statistical computing at Bell Labs. However, there were a number of problems that we hoped to address in the future. We saw the need for an increasing number of functions and, even with the interface macros, it was too hard to interface new subroutines to S. A goal was to enable users to interface their own subroutines. Our reliance on the QED editor to provide basic maintenance scripts for S was painful, because QED programs were difficult to write and almost impossible to read.

As S developed, the transfer vectors that pointed into the S executive would occasionally need updating. This necessitated a long process of relinking all of the S functions.

The basic parse-eval-print loop was successful and insulated users from most problems involving operating system crashes; when the computer went down, all computations carried out to that time were already saved. However, the master scatter storage file was sometimes corrupted by a crash that happened in the middle of updates, making the system somewhat fragile.

THE UNIX GENERATION (1981 BROWN BOOK)

Although the GCOS implementation of S proved its feasibility, it soon became apparent that growth of S required a different environment. Many of our colleagues in statistics research were using S for routine data analysis, but were still writing stand-alone Fortran programs for their research work. At the same time, word of S was getting to other researchers, both within the Bell System and at universities around the country. The time had come for S to expand its horizons.

Hardware Changes

In 1978, an opportunity came to perform a portability experiment with S. Portability was considered an important notion for S, so when the computer science research departments got an Interdata 8/32 computer to use in portability experiments for the UNIX operating system (Johnson and Ritchie, 1978), S naturally went along. The Interdata machine was a byte-oriented computer with 32-bit words. Prior to this, S work on UNIX was hampered by the 16-bit address space of machines that ran the UNIX system and by the lack of a good Fortran compiler. The latter problem was solved by a new Fortran 77 compiler (Feldman and Weinberger, 1978). As the UNIX research progressed on the Interdata machine, all sorts of useful tools and languages (C in particular) became available for use with S.

Gradually, we realized the advantages of using the UNIX system as the base for S. By the time that we had parallel implementations of S on GCOS and UNIX, it became apparent to us that porting S to a new operating system was going to be painful, no matter how much work was done to isolate system dependencies. That is because the utilities, the scaffolding needed to build S, would have to be rewritten for each new operating system. That was a big task. By writing those utilities once for the UNIX system and making use of the shell, the loader, *make*, and other tools, S could travel to any hardware by hitching a ride with UNIX. It was obvious even then that there would be far more interest in porting the UNIX operating system to new hardware than in porting S to a new system.

The link with the UNIX system provided other benefits as well. S could use the C language for implementation, since C was likely to be available and well debugged on any UNIX machine. Since the *f77* Fortran-77 compiler used the same code generator as C, inter-language calls were easy. S could rely on a consistent environment, with the shell command language (a real programming language to replace QED scripts), the *make* command for describing the system build process, known compilers and compiler options, a reasonably stable linker and libraries, and a hierarchical file system with very general file names.

The benefits of linking S and UNIX were obvious, but the Interdata environment was a one-time experiment and not hardware that would be widely used. The DEC PDP-11 machines were the natural home of the UNIX system at the time and we wanted to try S there. Unfortunately, these machines had a 16-bit architecture and the squeeze proved to be too much until the PDP-11/45 machine came out with 16-bit addressing for both instructions and data. S functions naturally ran as separate processes, but to fit on the PDP-11 we even made the parser into a separate process. Similarly, graphics was done by running the device function as a separate process linked by two-way pipes to the main S process. S put a rather severe strain on the resources of such a small machine, so it was obvious that the PDP-11 was not the ideal S vehicle, either.

Soon we had another opportunity to test our new model of piggyback portability. The UNIX system was ported to the DEC VAX computers, a new generation of 32-bit machines, where the size constraints of the PDP-11 wouldn't chafe so much. Freed from process size constraints, we tried a new implementation, where all of the functions were glued together into one large S process. That allowed the operating system to use demand paging to bring in from disk just the parts of S that were needed for execution. This worked well with S, because it is unlikely that a single S session will use more than a fraction of the S functions. The 32-bit architecture also allowed S to take on much bigger problems, with size limited only by the amount of memory the operating system would allow S to dynamically allocate. By eliminating all of the separately executable functions, disk usage was also much less, and the single S process could easily be relinked when basic S or system library routines changed.

By October, 1979, we had made a concerted effort to move all of our S functions to the UNIX version of S, making it our primary platform.

Changes to S

Perhaps the biggest change made possible by running S on the VAX machines was the ability to distribute it outside of Bell Labs research. Our first outside distribution was in 1980, but by 1981, source versions of S were made widely available. At the time, the Bell System was a regulated monopoly, and as such would often distribute software for a nominal fee for educational use. Source distribution not only allowed others access to S, but provided the statistics research departments a new vehicle for disseminating the results of our research. As we began wider distribution of S, we dropped version numbering in favor of

dates; for example an S release might be identified as ‘‘June, 1980’’. Major releases were accompanied by completely new books; updates had revised versions of the online documentation.

As S became more widely used, it also gained some new features. One major addition was that of a portable macro processor, implemented by Ed Zayas, based on the macro processor from *Software Tools* (Kernighan and Plaugher, 1976).† Macros allowed users to extend the S language by combining existing functions and macros into entities designed specifically to carry out their own tasks. Once a macro was completed, it would carry out the computation and hide the implementation details.

As users began to use S for larger problems, it became apparent that the implicit looping that S used (iterating over all elements of a vector), was unable to handle every problem. It was necessary to add explicit looping constructs to the language. We did so with a general `for` expression:

```
for(index in values) expression
```

where the `index` variable would take on the series of values in turn, each time executing the `expression`.

Since one very common use of for loops was to iterate over the rows or columns of a matrix, the `apply` function was introduced. This function takes a matrix and the name of a function and applies that function in turn to each of the rows or columns of the matrix (or to general slices of an array) and does not require an explicit loop. We were able to implement `apply` so that the implicit looping was done with much greater efficiency than could be achieved by a loop in the S language.

Graphics were enhanced considerably at this time. New devices were supported, including pen plotters from Hewlett-Packard and Tektronix. Graphic input was available, to allow interaction with a plot. The user would manipulate the cursor or pen of the plotting device to signal a position. Underlying *graphical parameters* allowed control over both graphical layout and rendering. These parameters originated with the GR-Z graphics library, but extended naturally to S use. Layout parameters allowed specification of the number of plots per page (or screen), the size and shape of the plot region; these parameters were all set by the `par` function. Rendering parameters, controlling such things as line width, character size, and color, were allowed as arguments to any graphics functions, and their values were reset following the call to the graphics function.

Another innovation was the `diary` function, that allowed the user to activate (or deactivate) a recording of the expressions that were typed from the terminal. This recognized the importance of keeping track of what analysis had been performed and of the unique position of the system to provide a machine-readable record. This facility also made S comments more important. Syntactically, the remainder of a line following a ‘‘#’’ was ignored, thus allowing comments, but now, the comments could also be recorded in the diary file, giving a way of making annotations. In the interests of conserving disk space, the diary file was off by default.

At this time, S provided 3 directories† for holding datasets: the *working* directory, the *save* directory, and the *shared* directory. Any datasets created by assignments were stored in the working directory. Although it provided storage that persisted from one S session to another, the thought was that the user would clean up the working directory from time to time, or perhaps even clear it out altogether. In order to preserve critical data (the raw data for an analysis and important computed results), the save directory was available. The function `save` was used to record datasets on the save directory.† Finally, the shared directory was the repository for some datasets that S provided — generally example datasets that were used in the manual.

A new data structure was introduced at this time: the category. It was designed to hold discrete (categorical) data and consisted of a vector of labels and an integer vector that indexed those labels. Computationally, the category acted like a vector of numbers, but it still retained the names of the original levels.

As S became more popular, demand for users to add their own algorithms began to grow, and we also

---

† A macro facility was described in documentation for the earlier version of S, but it was seldom used.

†The name ‘‘directory’’ replaced the earlier usage ‘‘database’’ to emphasize that the S datasets were stored in a UNIX system directory.

†We should have realized that people have a very hard time at cleaning up; the working data tended to be used for everything and the save directory was seldom used.

felt the need to expand the basic pool of algorithms available within S. This lead to the creation of the *interface language*. This language was designed to describe new S functions, specifying the S data structures for arguments, their default values, simple computations, and the output data structure. It was built from several components. First, there was a ‘‘compiler’’ for the language, that used a *lex* program to turn the input syntax into a set of *m4* macro calls. This compiler was written by Graham McRae, a summer employee of Bell Labs. The macro calls emitted by the compiler were run through *m4* with a large collection of macro definitions, the interface macros, whose job was to produce the code that dealt with argument and data structure processing for S functions, producing a file in the algorithm language (described above). The algorithm language was then processed by *m4*, *ratfor*, and the Fortran compiler, in sequence, in order to produce executable code. As might be expected, the interface language had a number of quirks, each inherited from one of the many tools required to process it.

The macro processor was very popular, yet it was also the source of confusion for many users. While simple macros could be used by rote, a real understanding of macro processing was required for more complex tasks. Because macro processing was carried out as a preliminary to the S parser, users often needed to understand the difference between operations that were carried out at macro time by the macro processor and those that were carried out during execution by the S functions. The macro processor built the expressions that were given to the S parser and evaluator, but in so doing, occasionally needed its own special operations. Thus, the macro processor had separate assignments (the `?DEFINE` macro), conditionals (the `?IFELSE` macro), and looping (the `?FOR` macro). Although the initial ‘‘?’’ and the all-capital-letter names were meant as a cue that these operations were different from ordinary S ‘‘functions’’, the notion confused many users. While in many ways macros were able to provide a convenient way for users to store sequences of S expressions, there were annoying problems: macros could not be used by the `apply` function, for example.

Another example of the problems with macros was the use of storage. Often, a computation encapsulated in a macro required intermediate datasets, but it was important that those datasets not conflict with ones that the user had created explicitly. A naming convention, that named these temporaries with an initial ‘‘T’’, was adopted, but even this was insufficient. Suppose one macro called another macro. If the first macro had a temporary named ‘‘Tx’’, it was important that the second macro not use the same name. Eventually, we developed a macro ‘‘?T’’ that would include a level number in the generated name, to avoid these problems. Thus, ‘‘?T(x)’’ would produce the name ‘‘T1x’’ in the first-level macro and ‘‘T2x’’ in the next level. Finally, some way was needed to clean up all of these temporary datasets at the end, so the function that removed objects was made to return a value so that the last step of a macro could remove temporaries and yet still return a value.

Overall, this first UNIX-based version of S worked well and its availability to a wider audience helped S grow. The next step was to polish it up, provide better documentation, and make it more available.

S TO THE WORLD (1984 BROWN BOOK)

In 1984, the book *S: An Interactive Environment for Data Analysis and Graphics* was published by Wadsworth (and later reviewed by Larntz, 1986). This book, along with an article in the *Communications of the ACM,* (Becker and Chambers, 1984b) described S as of 1984. The following year, Wadsworth published *Extending the S System*, describing the interface language and the process of building new S functions in detail.

Prior to 1984, S was distributed directly from the Computer Information Service at Bell Labs in New Jersey. Beginning in 1984, S source code was licensed, both educationally and commercially, by the AT&T Software Sales organization in Greensboro, North Carolina. S was always distributed in source form, in keeping with the small group of people working directly with it, i.e. Becker and Chambers (and, in 1984, Wilks). Especially after our experiences with the divergent GCOS and UNIX versions of S, we realized the importance of keeping only one copy of the source code. It was the only way that our small group could manage the software. We also knew that a source code distribution would allow people to make their own adjustments as they installed S on a variety of machines and we also knew that there was no way that we could provide binary distributions for even a handful of the hardware/software combinations that S users had. Our focus was on portability within the UNIX system (Becker, 1984). Also, remember that the S

work was being done by researchers; S was not viewed as a commercial venture with support by Bell Laboratories. Thus, sites that used S needed source code to give them a way to administer and and maintain the system.

While a number of S features were expanded and cleaned up in the 1984 release, perhaps the more important influences were happening outside of that version. John Chambers moved to a different organization at Bell Laboratories and began work on a system known as QPE, the Quantitative Programming Environment (Chambers, 1987). The QPE work was based on S, but was intended to make it into a more general programming language for applications involving data. Up to now, S was known as a ''statistical'' language, primarily because of its origins in the statistics research departments, but in reality, most of its strengths were in general data manipulation, graphics, and exploratory data analysis. While millions of people with personal computers began using spreadsheets for understanding their own data, few of them thought they were doing anything like ''statistics''. The actual statistical functions in S were primarily for regression. There were, of course, functions to compute probabilities and quantiles for various distributions, but these were left as building blocks, not integrated with statistical applications. QPE was a natural notion to move S away from a perception that it was best used by a statistician.

For some time, Chambers worked on QPE by himself, rewriting the executive in C and generalizing the language. In 1986 QPE was ready to be used by others, and the question was how should QPE and S relate to one another? At the time, S had thousands of users, while QPE was still a research project; S had hundreds of functions, while QPE had few. A marriage was proposed: we would integrate the functions of S with the executive of QPE, gaining the best of both worlds.

NEW S (1988 BLUE BOOK)

The combination of QPE and S produced a language that was called ''New S''. Because it was different from the 1984 version of S in many ways, there was a need for a new name.† In the Fall of 1984, just after publication of the Brown book, Allan Wilks joined Bell Laboratories and became a full partner in the S effort. The blue book, *The New S Language*, was authored by Becker, Chambers, and Wilks, as was the New S software. Major changes to S occurred during the 1984-1988 period.

Functions and Syntax

The transition to New S was perhaps the most radical that S users experienced. The macro processor was gone, replaced by a new and general way of defining functions in the S language itself. This made functions into truly first-class S objects. Functions were named by assignment statements†:

```
square <- function(x) x^2
```

S functions could now be passed to other functions (like `apply`), and could be the output of other computations. Because S data structures could contain functions, it no longer seemed appropriate to use the name ''dataset''; from this point on, we referred to S ''objects'' (a suggestion of David Lubinsky).

One nice characteristic of S functions is that they are able to handle arbitrary numbers of arguments by a special argument named " `...`". Any arguments not otherwise matched are matched by `...` and can be passed down to other functions.

Internally, S functions are stored as parse trees, and explicit functions `parse` and `deparse` are provided to produce parse trees from text and to turn those trees back into character form. Parse trees are S objects, and can be manipulated by S functions. For example, the blue book contains a function that takes symbolic derivatives of S expressions.

Many functions were rewritten in the S language when New S was introduced. Functions like `apply` became easy to express in the S language itself, although these implementations were somewhat less efficient than the versions they replaced. In general, as functions are implemented in S, the users have an

---

† Eventually, the new S language was called simply S; the 1984 version became known as Old S and now it is hardly mentioned.

†The initial idea of functions in QPE did not include this form of function as an S object. Instead, the syntax was more akin to the way we defined macros: `FUNCTION square(x) x^2`.

easier time of modifying and understanding them, because debugging is done in the interpreted S language (with the help of interactive browsing).

Unlike the transition from the 1981 to 1984 versions of S, the transition to New S was difficult for users. Their macros would no longer work. To help in this change, we provided a conversion utility, `MAC.to.FUN` that would attempt to convert straightforward macros to S functions.

New S provided a very uniform environment for functions. All S functions were represented by S objects, even those whose implementation was done internally in C code. An explicit notion of interfaces provided the key: there were functions to interface to internal code (`.Internal`), to algorithms in Fortran (`.Fortran`), to algorithms in C (`.C`), and even to old-S routines (`.S`). Along these same lines, the function `unix` provided an interface to programs that run in the UNIX system—the standard output of the process was returned as the value of the function. This was an outgrowth of successful early experiments in interfacing S and GLIM (Chambers and Schilling, 1981); we found it was quite feasible to have a stand-alone program carry out computations for S.

From the user's point of view, the most noticeable syntax change was in the way very simple expressions were interpreted. For example, with old-S, the expression

```
foo
```

would print the dataset named ''foo'' if one existed or else execute the function ''foo'' with no arguments. With new S, this expression would always print the object named ''foo'', even if it happened to be a function. To execute the function required parentheses:

```
foo()
```

In old-S, it had been possible to omit parentheses from even complicated expressions:

```
foo a,b
```

was once equivalent to

```
foo(a,b)
```

but it is now a syntax error.

Some functions familiar to old-S users had their names changed or were subsumed by new functions. These changes were described in the detailed documentation of the blue book under the name `Deprecated`. In some instances, the operations that the functions carried out were no longer needed; in others the operation changed enough that it was deemed reasonable to change the name to force the user to realize that something had changed. In other cases, fat was trimmed. For example, there was no need for the Fortran-style ''**'' operator for superscripts when ''^'' was available and more suggestive.

Various functions were introduced in order to help in producing and debugging S functions. Most important was the `browser`, that allowed the user interactively to examine S data structures. It became an invaluable tool for understanding the state of a computation and was often sprinkled liberally throughout code under development. The function `trace` allowed selected functions to be monitored, with user-defined actions when the functions were executed. There was also an option that specified the functions to be called when S encountered an error or interrupt. Implementation of `trace` is particularly elegant; a traced function is modified to do the user actions and is placed in the session frame (retained for the duration of an S session) where it will be earlier on the search path when the S executive looks for it. Both the `browser` and `trace` functions are written entirely in S.

Data and Data Management

New S regularized the treatment of directories, by having a list of them, rather than the fixed sequence of working, save, and shared directories of Old S. The functions `get` and `assign` were extended to allow retrieval and storage of S objects on any of the attached directories. An S object, `.Search.list` was used to record the names of the directories on the search list.

A major change in data structures was in the separation of the notions of hierarchical structures and attributes of vectors. A vector of mode "`list`" represented a vector of other S objects. Such vectors could be subscripted and manipulated like other, atomic mode, vectors. Now, any vectors could have attributes:

`names` to name the elements of the vector, `dim` to give dimensioning information for treating a vector as an array, etc. The notion of a vector structure was recognized as simply a vector with other attributes. Vectors were also allowed to have zero length, a convenience that regularizes some computations.†

A first hint at the object-oriented features of S came about with the 1988 release. The function `print` was made to recognize an attribute named `class`; if this attribute was present with a value, say, `xxx` and if there was a function named `print.xxx`, that function would be called to handle the printing.

Another feature of New S was the notion that S objects could be used to control the operation of S itself. In particular, there were four such objects: `.Program` expressed the basic parse/eval/print loop in S itself and could be replaced for special purposes; `.Search.list` gave a character vector describing the directories to be searched for S objects, `.Options` was a list controlling optional features such as the S prompt string, default sizes and limits for execution parameters, etc. `.Random.seed` recorded the current state of the random number generator. In addition, various functions named beginning with `sys.` contained information about the state of the S executive and its current evaluation. This allowed the `browser` to be applied to evaluation frames of running functions.

The diary file of old-S was replaced by an audit file. The audit file furnishes both an audit trail by which the entire set of expressions given to S can be recreated, and it also is the basis for a history function that allows re-execution of expressions and for an externally implemented auditing facility (Becker and Chambers, 1988). There have been occasional complaints about the presence of the audit file, about its size, and all of the information therein describing which S objects were read and written by each expression, but generally it is recognized that the file contains important information about the path of the analysis. The external audit file processor can be used to recreate analyses, to find the genesis of objects, or to decide what should be executed following changes to data objects.

S data directories inherited a feature from commercial databases: commitment and backout upon error. As in the past, each S expression typed by the user is evaluated as a whole and the system starts afresh with each new expression. With the database backout facilities, S objects that are created during an expression are not written to the database until the expression successfully completes. This provides several advantages. First, because objects are kept in memory during expression execution, multiple use or assignments to one object do not require multiple disk access. Second, if an error occurs, the old data that was present before the error is preserved.

Another feature is that of in-memory databases, called *frames*, which hold name/value pairs in memory. A new frame is opened for each function that executes, providing a place to store temporary data until that function exits. When a function exits, its frame and all that was stored there is deallocated. There is also a frame associated with each top-level expression as well as one that lasts as long as the entire S session.

Implementation

New S was implemented primarily in C, as was the QPE executive. Internal routines, too, such as arithmetic and subscripting, were generally rewritten in C. The rewrite was necessary to support double-precision arithmetic. Code to support complex numbers was added. Many routines that in old-S had been implemented in the interface language were re-implemented in the S language, although graphics functions were largely left in the interface language and invoked through the `.S` function. Most of the Fortran that was present in S was rewritten, although basic numerical algorithms, such as those from LINPACK (Dongerra, et. al, 1979) or EISPACK (Smith et. al, 1976) were left in Fortran.

The advent of New S also brought about a basic change in underlying algorithms and data. Now, everything is done in double precision, although the user is presented with a unified view of a data mode named `numeric`, which includes integer and floating point values. For special purposes (most notably for passing values down to C or Fortran algorithms), it is possible to specify the representation of numbers (integer, single, or double precision), but this is normally far from the user's mind.

S uses lazy evaluation. This means that an expression is not evaluated until S actually needs the

†The APL language is especially good at treating boundary cases such as zero-length vectors and we found that proper treatment of boundary cases often makes writing S functions easier.

value. For example, when a function is called, the values of its actual arguments are not computed until, somewhere in the function body, an expression uses the value.

Along with the ability to call C or Fortran algorithms, on some machines S is capable of reading object files (such as those produced by the C or Fortran compilers) and loading them directly into memory while S is executing. This means that new algorithms can be incorporated into S on the fly, with recompilation of only the changed program, and without relinking the S system itself. Dynamic loading like this is machine dependent, requiring a knowledge of object file formats and relocation procedures, so it only works on a select group of computers. It is always possible to link external subroutines with S by using the UNIX system loader.

Graphics functions in New S are very similar to those in earlier versions (for the most part, they are still implemented by old-S interface routines), however they have been regularized in many ways. A single function `plot.xy` is used to determine the *x*- and *y*-coordinates for a scatterplot. Utilizing the parse-tree for the plotting expression, S can provide meaningful labels for the plot axes, based on the expressions used to produce the *x*- and *y*-coordinates. Graphics devices are implemented by C routines that fill a data structure with pointers to primitive routines: draw a line, plot a string, clear the screen, etc.

As pen plotters and graphics terminals became less prevalent, new graphics device functions were brought into S. In particular, the most commonly used devices are now one for the X window system (`x11`), and for Adobe's PostScript™ language (`postscript`).

A novel portion of the blue book describes the semantics of S execution in the S language itself. We found that it was often easier to implement new ideas using the S language in order to try them out, and only after deciding that they were worthwhile would we recode them more efficiently in C.

The advent of New S and the blue book gave S a solid foundation as a computational language. The time was ripe for a major thrust into statistics.

STATISTICAL SOFTWARE IN S (1991 WHITE BOOK)

Because of its relatively recent publication, it is harder to reflect on the book, *Statistical Models in S* (Chambers and Hastie, 1992). Certainly, a number of important enhancements to the core of S were made in that time, although, for the most part, the Statistical Models work was a very large implementation of software primarily in the S language itself. Notice that statistics is not properly part of S itself; S is a computational language and environment for data analysis and graphics. The Statistical Models work is implemented in S and is not S itself.

The models work was done by ten authors, coordinated by John Chambers and Trevor Hastie. In some ways, this was a departure from the small-team model of S development. On the other hand, the work was based upon a common core and was carried out by researchers with intense personal interests in their particular portions. In addition, formal code reviews were held for all of the models software, helping to ensure that it was well-implemented and fit in with the rest of S.

The most fundamental contribution of the Models work is probably the modeling language, and it was achieved with a minimal change to the S language — the addition of the "`~`" operator to create `formula` objects (Chambers, 1993a). The modeling language provided a way of unifying a wide range of statistical techniques, including linear models, analysis of variance, generalized linear models, generalized additive models, local regression models, tree-based models, and non-linear models.

The major change to S itself was the strong reliance on classes and methods. Generic functions were introduced; they call the function `UseMethod`, which tells the S executive to look for the `class` attribute of the generic function's argument and to dispatch the appropriate *method* function, the one whose name is made by concatenating the generic function name with the class name. By installing the object-oriented methods in this form much of S could remain as it was, with generics and methods being installed where they were needed. A form of inheritance was supplied by allowing the `class` attribute to be a character vector rather than just a single character string. If no method can be found to match the first class, S searches for a method that matches one of the later classes. Chambers (1993a) describes the ideas in much greater detail.

Another major addition to the language concerns data: the *data frame* class of objects provides a

matrix-like data structure for variables that are not necessarily of the same mode (matrices must have a single mode). This enables a variety of observations to be recorded on the subjects: for example, data on medical patients could contain, for each patient, the name (a character string), sex (a factor), age group (an ordered factor), height and weight (numeric).

Another interesting inversion happened with S data directories; we once again refer to them as databases. This is because the notion of a UNIX directory became too confining. Now, there are several sorts of S databases: 1) UNIX system directories; 2) S objects that are attached as part of the search list; 3) a compiled portion of S memory; 4) a user-defined database. Because of this, we now think of methods that operate on S databases. There are methods to attach databases to the search list, to read and write objects, to remove them, to generate a list of objects in a database, and to detach them from the search list. The search list itself outgrew its early character-string implementation; it is now a true list.

THE FUTURE

This paper has described S up to the present, but will soon be out of date, as S continues to evolve. What is likely to happen?

We intend to address efficiency issues. New ideas in internal data structures may allow significant savings at run time by allowing earlier binding of operations to data. A better memory management model may use reference counts or other techniques to avoid extraneous copies of data, improving memory usage.

We hope to eliminate another of our side effects: graphics. Instead of calling graphics functions to produce output, graphics functions would instead produce a graphics object. The (automatically called) print method for a graphics object would plot it. This concept would allow plots to be saved, passed to other functions, modified, composed with one another, etc.

One general area that we would like to integrate into S is the area of high-interaction graphics with linked windows. Difficult research issues remain with this, including the problem of determining when a set of points on one plot corresponds to a set on another plot. There seems to be a conflict with this application and the general independence of S expressions.

In the long run, for portability reasons, we would like to eliminate Fortran from S, relying on C or perhaps C++ as the implementation language. One less language would make it easier (and less expensive) for people to compile S. But without Fortran, how do we make use of broadly available numerical algorithms written in Fortran? The *f2c* tool (Feldman, et. al, 1990) holds some promise here, since it has been used successfully to install S on machines without Fortran.

When we built S, we intentionally did not provide a user interface with added features such as command editing or graphical input. We preferred instead to build a simple system that could support a variety of specialized user interfaces. The X window system (Scheifler and Gettys, 1986) may now provide a portable way to implement graphical user interfaces. Perhaps the time has come for serious experiments along these lines.

With the Statistical Models work, we did not press the object-oriented methodology to its logical conclusion. For example, we did not convert current time series, matrices, and categories into objects with `class` attributes, although we plan to do so in the future. This seems obvious in retrospect: many of the special cases in arithmetic computations and subscripting are caused by the need to deal with time-series, matrices, and categories. By using classes, we could handle these special cases in separate functions. Factors are one step in this direction; a factor is essentially a category with a class attribute and a variety of methods to provide appropriate operations on factors.

The future of S is increasingly influenced by people in the academic and business communities. The commercial success of the S-PLUS system from StatSci (1993) has brought the S language to thousands of people, including users of personal computers (reviewed by Therneau, 1990). Software related to S is distributed electronically via the Mike Meyer's Statlib archive, `statlib@lib.stat.cmu.edu`, and an electronic user group, `S-news@stat.toronto.edu`, actively discusses S issues. Härdle (1990) has written a book describing smoothing with an implementation in S, and Venables and Ripley (1994) have written a statistics text based on S.

CONCLUSIONS

Early on, we hinted that a study of the history of S could be helpful in understanding the software development process. Now, the time has come to try to draw some of these conclusions.

Much of the work in getting S to its current state followed an evolutionary process. Couldn't we have gotten there more quickly, more easily, and with less user pain by designing the language better in the first place? I think the answer is ''No.'', we could not have avoided some of the excursions we made.

S in 1976 was remarkably like S of 1993 — many of the basic ideas were in place. However, even if we had a blueprint for the full version of S in 1976, it would have been overwhelming, both to us as implementors and to the computing hardware of the day. Aiming too high may have caused us to give up in advance. It was useful to keep the project at a level that could be sustained by two or three people, because in that way, S evolved in a way that was aesthetically pleasing to all of us. S was not designed by committee. There is nothing like individual motivation — wanting to produce something for your personal use — to get it done right.

S was also the result of an evolution with plenty of feedback from our own and others' real use. The iteration of design, use, and feedback was important. It seemed that each time we completed a new revision to S and had a little while to rest, we then felt new energy that enabled us to think of forging ahead again.

Each version of S had limited goals. When we started, our initial idea was to produce an interactive version of the SCS library. Because it was limited, the goal also seemed attainable. As Brian Kernighan has stated ''I never start on a project that I think will require more than a month.'' Of course, the projects do grow to require more time than that, but especially for research, it is good to feel like the time horizon is short and that there will be time for feedback and changes.

Another important concept is that we always emphasized the way the user would see the system. Our initial efforts gave only a vague thought toward machine efficiency and concentrated instead on human efficiency. It is a mistake to let hardware dominate the design of software; yes, it does have to be implemented and actually work, but it need not be totally controlled by current hardware. In the years that S has been around, machine speed has increased by several orders of magnitude, disk space is far more available, and costs have come down dramatically. It would have been unfortunate if we had allowed the performance of now-ancient hardware to affect the overall design of S. A good idea is to first make it work and then make it fast.

Efficiency in S has always been an elusive goal and has often been the part of the code that changes most from one version of S to another. Work on efficiency must be continuous because, as the system gets more efficient and runs on more powerful computers, our tendency and that of users is to implement more and more in the S language itself, causing efficiency to be a constant battle. We have also found more than once that it is not obvious which changes will improve efficiency. Our best advice is to implement and then measure. Chambers (1992) discusses some recent attempts to measure performance and Becker, Chambers, and Wilks (1993) describes a tool we devised to monitor memory usage.

Along these same lines, we tried from the beginning to make S as broad as possible, by eliminating constraints. S was built to use whatever cpu or memory resources the operating system would allow; nothing in S itself restricts the problem size. Similarly, S has no restrictions in its language — everything is an expression, built up from combinations of other expressions. Data structures are also general combinations of vectors, with no restrictions on how they can be put together.

A general principle is that the language should have minimal side effects. In S, the major side effects are assignments (that create persistent objects), printing, and graphics. Minimal side effects provide an important property: each expression is independent of the expressions that preceded it, aside from the data objects that it uses.

S also succeeded by its basic building-block approach. Within the broad context of the S language, we furnished a number of functions to carry out primitive computations, relying on the user to put them together in a way most appropriate to the problem at hand. Although this does not make for a system that can be used without thought, it does provide a powerful tool for those who know how to get what they want.

We have always *used* S ourselves. Although that may seem like a trite observation, it is very

important to the way S has evolved. Often, the impetus for change in S is the result of using it to address a new problem. Without new problems to solve, S wouldn't grow. If we hadn't considered S our primary research tool, we would probably not have kept it up-to-date with the latest algorithms and statistical methods. S has always had new methodology available quickly, at least in part because it was designed to be extended by *users* as well as its originators.

S has done well, too, because of the synergy that comes from integrating tools. When developing an algorithm, the S user can use the data management and graphical routines to generate test data, to plot results, and generally to support the activity.

One of the characteristics that made S into what it has become is that each part of S is ''owned'' by someone. There has been no ''market driven'' component to S — each function is there because one of us wanted it there for our own use or because someone else convinced us that a particular function was necessary. Decisions about S have generally been by consensus (a necessity with two authors and highly desirable with three). Thus, S has the imprint of a few individuals who cared deeply and is not the bland work of a committee.

Because S came from a research environment, we have felt free to admit our mistakes and to fix them rather than propagate them forever. There have been a few painful conversions, most notably the transition from old-S to new S, but the alternative is to carry the baggage of old implementations along forever. It is better to feel the momentary pain of a conversion than to be burdened forever by older, less informed decisions. Perhaps, had S been a commercial venture, we would not have had the luxury of doing it again and again until we got it right.

A tools approach was certainly beneficial to the development of S. The software available within the UNIX system allowed us to build S quickly and portably. For example, the interface language compiler was a fairly easy project because of *lex*, *m4*, and *ratfor*. On the other hand, a tool built from other tools tends to inherit the idiosyncrasies of each of those tools. It was difficult to describe to users of the interface language just why there were so many reserved words in the language (see Appendix A of Becker and Chambers, 1985).

One way of assessing the evolution of a language is to watch for concepts that have been dropped over time. These notions are important clues to what was done wrong and later improved. In S, the notion of a dataset was dropped in favor of the idea of an object. This recognized the much more general role that S objects had in the language, holding not only data but functions and expressions. Similarly, our idea of a vector structure was approximately right, but was clarified by two more precise concepts: the (vector) list of S objects and the notion that any object could have attributes.

Steve Johnson once said that programming languages should never incorporate an untried feature. While we didn't slavishly follow that advice, most of the features present in S were influenced by some other languages. We often found a good concept in an existing system and adapted it to our purposes.

As an example, the basic interactive operation of S, the parse/eval/print loop, was a well-explored concept, occurring in APL, Troll, LISP, and PPL, among others. From APL we borrowed the concept of the multi-way array (although we did not make it our basic data structure), and the overall consistency of operations. The notion of a typeless language (with no declarations) was also present in APL and PPL. As the PPL manual (Taft, 1972) stated, ''*type* is an attribute, not of *variables*, but rather of *values*.''

We wanted the basic syntax of the language to be familiar, so we used standard arithmetic expressions such as those found in Fortran or C. We wanted to emphasize that the assignment operation was different from an equality test so, like Algol, we used a special operator. The specific "`<-`" operator came from PPL (and was a single character on some old terminals).

The basic notion that everything in S is an expression fits in very nicely with the formal grammar that describes S — a language statement is just an expression. Certainly C and Ratfor contributed the notion that a braced expression was an expression, thus allowing S to get by without need for special syntax like BEGIN/END or IF/FI or CASE/ESAC.

Some of the more innovative ideas in data structuring came from LISP: the lambda calculus form of function declarations, the storage of functions as objects in the language, the notion of functions as first-class objects, property lists attached to data.

Some pieces of syntax come from languages that are pretty far removed from the applications we foresaw with S. In particular, the IBM 360 Assembler and Job Control (JCL) languages helped to contribute the notions of arguments given by either positional or keyword notation, along with defaults for the arguments. The common thread between S and these languages was the need for functions that had many arguments, although a user would only specify some of them for any particular application.

More recently, we borrowed ideas about classes and inheritance from languages like C++, CLOS, and Smalltalk.

The Troll system contributed in many ways: the need for a time-series data type and special operations on time-series; interactive graphics for data analysis; persistent storage of datasets; the power of functions as a basic way of expressing computations.

The UNIX operating system also contributed a few ideas (as well as an excellent development platform): the use of a leading "`!`" to indicate that a line should be executed by the operating system and the notion of having simple, well defined tools as the basis for computing. The UNIX shell also contributed to the notion of continuation lines. Like S, it knows when an expression is incomplete and responds with a special prompt to indicate this.

The origins of a few concepts are more mysterious. Perhaps they are unique to S in their precise implementation, but they were probably the result of some external influence. In particular, the syntax for the `for` loop: `for(i in x)` that allows looping over the elements of various data structures and doesn't force the creation of an integer subscript. Other programming languages have functions that can cope with arbitrary numbers of arguments, but the use of `...` in S has its own unique qualities.

One important non-technical concept was very important to the evolution of S. Almost from the beginning, S was made readily available in source form to AT&T organizations, to educational institutions, and to outside companies. This distribution of source allowed an S user community to grow and also gave the research community a vehicle for distributing software. At present there is an active S mailing list, consisting of hundreds of users around the world, asking questions, providing software, and contributing to the evolution of S. We hope that evolution continues for a long time.

ACKNOWLEDGEMENTS

I would like to thank John Chambers and Allan Wilks for comments on this manuscript. Over the years many people have been contributed to S in one way or another. Some of them have been mentioned here, but there are many others whose often considerable efforts could not be mentioned for lack of space. Among them are Doug Bates, Ron Baxter, Ray Brownrigg, Linda Clark, Bill Cleveland, Dick De Veaux, Bill Dunlap, Guy Fayet, Nick Fisher, Anne Freeny, Colin Goodall, Rich Heiberger, Ross Ihaka, Jean Louis Charpenteau, John Macdonald, Doug Martin, Bob McGill, Allen Mcintosh, Mike Meyer, Wayne Oldford, Daryl Pregibon, Matt Schiltz, Del Scott, Ritei Shibata, Bill Shugard, Ming Shyu, Masaai Sibuya, Kishore Singhal, Terry Therneau, Rob Tibshirani, Luke Tierney, and Alan Zaslavsky. Thank you. Unfortunately, I'm sure that other names that should appear here have been omitted. My apologies.

REFERENCES

Baker, Brenda S. (1977), ''An Algorithm for Structuring Flowgraphs'', *J. Assoc. for Comp. Machinery*, Vol 24, No. 1, pp. 98-120.

Becker, Richard A. (1978), ''Portable Graphical Software for Data Analysis'', *Proc. Computer Science and Statistics: 11th Annual Symposium on the Interface*, pp 92-95.

Becker, Richard A. (1984), ''Experiences with a Large Mixed-Language System Running Under the UNIX Operating System'', *Proc. USENIX Conference.*

Becker, Richard A. and John M. Chambers (1976), ''On Structure and Portability in Graphics for Data

Analysis'', *Proc. 9th Interface Symp. Computer Science and Statistics.*

Becker, Richard A. and John M. Chambers (1977), ''GR-Z: A System of Graphical Subroutines for Data Analysis'', *Proc. 10th Interface Symp. on Statistics and Computing*, 409-415.

Becker, Richard A. and John M. Chambers (1978), *'S': A Language and System for Data Analysis*, Bell Laboratories, September, 1978. (described ''Version 2'' of S).

Becker, Richard A. and John M. Chambers (1981), *S: A Language and System for Data Analysis*, Bell Laboratories, January, 1981.

Becker, Richard A. and John M. Chambers (1984), *S: An Interactive Environment for Data Analysis and Graphics*, Wadsworth Advanced Books Program, Belmont CA.

Becker, Richard A and John M. Chambers (1984b), ''Design of the S System for Data Analysis'', *Communications of the ACM*, Vol. 27, No. 5, pp. 486-495, May 1984.

Becker, Richard A. and John M. Chambers (1985), *Extending the S System*, Wadsworth Advanced Books Program, Belmont CA.

Becker, Richard A. and John M. Chambers (1988), ''Auditing of Data Analyses'', *SIAM J. Sci. and Stat. Comp.*, pp. 747-760, Vol 9, No 4.

Becker, Richard A. and John M. Chambers (1989), ''Statistical Computing Environments: Past, Present and Future,'' *Proc. Am. Stat. Assn. Sesquicentennial Invited Paper Sessions*, pp. 245-258.

Becker, Richard A., John M. Chambers and Allan R. Wilks (1988), *The New S Language*, Chapman and Hall, New York.

Becker, Richard A., John M. Chambers and Allan R. Wilks (1993), ''Dynamic Graphics as a Diagnostic Monitor for S'', AT&T Bell Laboratories Statistics Research Report.

Chambers, John M. (1974), ''Exploratory Data Base Management Programs'', Bell Laboratories Internal Memorandum, July 15, 1974.

Chambers, John M. (1975), ''Structured Computational Graphics for Data Analysis'', Intl. Stat. Inst. Invited Paper, 1-9 IX 1975, Warszawa.

Chambers, John M. (1977), *Computational Methods for Data Analysis*, Wiley, New York.

Chambers, John M. (1978), ''The Impact of General Data Structures on Statistical Computing'', Bell Laboratories Internal Memorandum, May 20, 1978.

Chambers, John M. (1980), ''Statistical Computing: History and Trends'', *The American Statistician*, Vol 34, pp 238-243.

Chambers, John M. (1987), ''Interfaces for a Quantitative Programming Environment'', *Comp. Sci. and Stat, 19th Symp. on the Interface*, pp. 280-286.

Chambers, John M. (1992), ''Testing Software for (and with) Data Analysis'', AT&T Bell Laboratories Statistics Research Report.

Chambers, John M. (1993a), ''Classes and Methods in S. I: Recent Developments'', *Computational Statis-*

tics, to appear.

Chambers, John M. (1993b), ‘‘Classes and Methods in S. II: Future Directions’’, *Computational Statistics,* to appear.

Chambers, John M. and Trevor Hastie, eds. (1992), *Statistical Models in S,* Chapman and Hall, New York.

Chambers, John M. and Judith M. Schilling (1981), ‘‘S and GLIM: An Experiment in Interfacing Statistical Systems’’, Bell Laboratories Internal Memorandum, January 5, 1981.

Dongerra, J. J., J. R. Bunch, C. B. Moler, and G. W. Stewart (1979), *LINPACK Users Guide,* Society for Industrial and Applied Mathematics, Philadelphia.

Feldman, S. I., David M. Gay, Mark W. Maimone, N. L. Schryer (1990), ‘‘A Fortran-to-C Converter, Computing Science Technical Report No. 149, AT&T Bell Laboratories.

Feldman, S. I. and P. J. Weinberger (1978), ‘‘A Portable Fortran 77 Compiler’’, Bell Laboratories Technical Memorandum.

Härdle, Wolfgang (1990) *Smoothing Techniques With Implementation in S,* Springer Verlag, New York.

Iverson, K. E. (1991), ‘‘A Personal View of APL’’, *IBM Systems Journal,* Vol 30, No. 4, pp 582-593. (The entire issue of this journal is devoted to APL.)

Johnson, S. C. and M. E. Lesk (1978), ‘‘Language Development Tools’’, *The Bell System Technical Journal,* Vol. 57, No. 6, Part 2, pp. 2155-2175, July-August 1978.

Johnson, S. C. and D. M. Ritchie (1978), ‘‘Portability of C Programs and the UNIX System’’, *The Bell System Technical Journal,* Vol. 57, No. 6, Part 2, pp. 2021-2048, July-August 1978.

Kernighan, Brian W. (1975), ‘‘RATFOR—A Preprocessor for a Rational Fortran’’, *Software—Practice and Experience,* Vol. 5, pp 395-406.

Kernighan, Brian W. and P. J. Plauger (1976), *Software Tools,* Addison Wesley.

Kernighan, Brian W. and Dennis M. Ritchie (1977), ‘‘The M4 Macro Processor’’, Bell Laboratories Technical Memorandum, April, 1977.

Larntz, Kinley (1986), ‘‘Review of S: An Interactive Environment for Data Analysis and Graphics’’, *J. of the American Statistical Association,* Vol. 81, pp. 251-252.

National Bureau of Economic Research (NBER, 1974), *Troll Reference Manual,* Installment 5, August 1974.

Ritchie, D. M. and K. Thompson (1970), ‘‘QED Text Editor’’, Bell Laboratories Technical Memorandum.

Ritchie, D. M, and K. Thompson (1978), ‘‘The UNIX Time-Sharing System’’, *The Bell System Technical Journal,* Vol. 57, No. 6, Part 2, pp. 1905-1929, July-August 1978.

Ritchie, D. M, S. C. Johnson, M. E. Lesk, and B. W. Kernighan (1978), ‘‘The C Programming Language’’, *The Bell System Technical Journal,* Vol. 57, No. 6, Part 2, pp. 1991-2019, July-August 1978.

Ryder, B. G. (1974), ‘‘The PFORT Verifier’’, *Software — Practice and Experience*, Vol. 4, pp. 359-377.

Scheifler, R. W. and J. Gettys (1986), ‘‘The X Window System’’, *ACM Transactions on Graphics*, Vol. 5, No. 2, pp. 79-109.

Smith, B. T., J. M. Boyle, J. J. Dongerra, B. S. Garbow, Y. Ikebe, V. C. Klema, and C. B. Moler (1976), *Matrix Eigensystem Routines—EISPACK Guide*, Second Edition, Lecture Notes in Computer Science 6, Springer-Verlag, Berlin.

Standish, T. A. (1969), ‘‘Some Features of PPL — A Polymorphic Programming Language’’, *Proc. of Extensible Language Symposium*, Christensen and Shaw, eds., *SIGPLAN Notices, Vol 4, Aug. 1969.*

StatSci (1993), *S-PLUS Programmer’s Manual, Version 3.1*, Statistical Sciences, Seattle.

Taft, Edward A. (1972), *PPL User’s Manual* Center for Research in Computing Technology, Harvard University, September, 1972.

Therneau, Terry M. (1990), ‘‘Review of S-PLUS’’, *The American Statistician*, Vol. 44, No. 3, pp. 239-241.

Tukey, John W. (1971), *Exploratory Data Analysis*, Limited Preliminary Edition. Published by Addison-Wesley in 1977.

Venables, W. N., and B. D. Ripley (1994), *Statistics with S*, Springer-Verlag.