Evolution of the S Language

John M. Chambers
Bell Laboratories
Murray Hill NJ 07974
jmc@bell-labs.com

Abstract

The S language and its supporting programming environment provide rapid high-level prototyping for computations with data, featuring interaction, graphics, and universal, self-describing objects. Programming in the current version of S uses function objects, informal classes and methods, and interfaces to other languages and systems.

A major revision of S, described as "Version 4", has been underway for several years, designed to improve the usefulness of the language, for a wide variety of applications and at every stage of the process of programming with data. This paper describes the design goals of the new version, reviews changes to date, and outlines some plans for future work.

The overall goal for S is: *to turn ideas into software, quickly and faithfully.* Version 4 aims to get closer to that goal with a variety of new features:

- Classes, generic functions and methods use an explicit representation and implementation, more controllable and more general than many other OOP systems.
- Event management is provided, with S-level facilities for specifying events and actions.
- The language and environment treat large problems more effectively, while retaining the essential features that make the language useful.
- The *chapter* concept has been extended to automatically link S and other software, documentation and class/method information.
- The *connection* class of objects unifies input/output operations and extends them to non-file streams (e.g., pipes and FIFO's).
- On-line documentation has been integrated with the language via documentation objects; S functions are self-documenting.
- Some syntactic extensions have been introduced to make S look more familiar to new users and to improve readability.
- The interface to C and facilities for integrating C code with S have been extended.
- The underlying code has simpler organization and installation, and uses C and Posix standards to clarify portability.

At the same time, Version 4 maintains a large measure of compatibility with the current release. In particular, most software using the earlier approach to classes and methods, as described in *Statistical Models in S*, should continue to work.

1 Goals for Version 4.

The design of Version 4 of S has as its central goal a powerful environment for *programming with data.* Programming with data means creating or modifying software that, at some point, uses information from data to summarize, display, report, or investigate. This includes programming for data analysis, but is potentially much more general. The important characteristics are that we want to express some new computation (the programming part) and that this computation requires getting information out of data. The current version of S is already being used for programming with data, often but not only by statisticians. Users perceive the main strengths of S for this activity as rapid interactive prototyping, graphics, and high-level objects.

S has evolved continually over many years. The current release, called Version 3 here, [2], reflects a major redesign carried out about ten years ago. It emphasizes a dual style of functional and object-based programming. That is, nearly everything in S can be viewed as the evaluation of a function call, analogous to more strictly functional languages such as ML. Also, everything handled in S is an S object, including expressions and functions as well as data. All objects are dynamically generated and self-describing. The combination of these two styles leads to the ease of prototyping in S.

A release of S in 1991, revised in 1992, added an informal approach to classes and methods [1, Appendix A]. Informality meant that users could create methods as ordinary S functions, using a naming convention to identify the generic function and the class of objects intended. Objects carried their own class and inheritance information.

Version 4 represents a major revision designed to strengthen S as a programming environment at every stage, from early prototyping to development of an extensive collections of software. We want to support new aspects of programming not explicitly part of the previous design, notably the construction of user interfaces. At the same time, the new version should be compatible with the current version (as defined by [3] and [1]).

The following sections outline the concepts behind the new features, assuming the reader is at least slightly familiar with a current version of S or S-Plus.

2 Classes and Methods.

A major motivation for Version 4 of S is to provide a more powerful and explicit mechanism for classes and methods. In Version 4, classes can be specified explicitly, including their representation and their relations to other classes. Generic functions and methods can likewise be specified explicitly in a very general form. The combination of generality and explicit representation gives S a unique mechanism for programming with classes and methods.

The paradigm we have in mind is that programming with data should move continuously from casual interaction, perhaps not even considered as programming, through more explicit and carefully designed steps. The formality, the generality,and other qualities of the results should reflect a balance of the usefulness of the resulting software to its end users against the expertise and effort needed to create the software. As the software develops, S will provide many levels of "seriousness" at which the creators can place their efforts. For example, simple functions can be created quickly. As the functions are used and revised, they will become more general and careful. At some point, it will often be useful to see these as *generic* functions, specialized by methods to various classes of data. Similarly, the kinds of data relevant to a particular project can be organized quickly using existing S classes. As the important characteristics of the data become clearer, new classes of objects are often useful, extending existing classes.

The programmer can eventually be as explicit and detailed as makes sense, although initial specifications are often extremely simple. Because everything exists as an S object, the structure of classes, generic functions, and methods is accessible in S itself, allowing programming styles not possible in languages where this structure is hidden away at compile time or where some aspects of the language (e.g., functions) are not first-class objects.

Some specific features are as follows:

- Classes can be specified, with a representation that includes any combination of other classes and specific slots. In addition or instead, an explicit prototype can be specified, allowing unambiguous re-use of simple objects as new classes.
- Classes can extend other classes, automatically from the representation or explicitly by programmer control.
- Virtual classes can be defined to encapsulate properties shared by objects that may differ in their actual representation; e.g., class `vector` is a virtual class, which all actual vector classes extend.
- Methods are specified by a general "signature"; that is, by saying what class one or more arguments should have (or be equivalent to) in order for a particular method to apply. S assembles all this information into an efficient table-driven selection of methods. Method selection is more precise and more general than in the earlier version.
- Existing functions can be converted trivially into generic functions, just by specifying a method. The existing definition is retained as the default method (including the use of Version 3 methods).
- Functions can also be defined directly as generics, which is the preferred style when no "default" method makes sense.

One major change during the design of Version 4 was the attitude to backward compatibility. The initial notion was that essentially all the earlier S data organization would be re-defined in terms of the new classes, doing it "right" this time. As the work evolved, it became clear that such a revision involved much more effort and time than was available. Also, it would inevitably leave users with substantial incompatibilities. The chosen alternative has been to build the Version 4 mechanism on top of existing facilities, including the informal class/method mechanism of [1]. The S evaluator takes considerable trouble to retain compatibility with earlier classes and methods. Old-style methods will be detected and called when an argument's class matches the method's name according to the rules in [1]. Old-style classes will be detected when objects are read from a database. The implicit hierarchy implied by multiple strings in the old-style class attribute will be recorded and applied in finding methods. The only constraint is that class hierarchies have to be consistent at any particular time. In other words, the inheritance of a particular class is uniquely defined, and stored internally by S, and should not vary from one object of the class to another. This was an intended constraint in old-style classes but was not enforced.

In parallel with this compatibility, new implementations of existing S objects (for example, matrices, arrays and other structure objects) are being designed. These provide a modern alternative to the traditional implementation and will have many advantages for new projects. Their compatibility with all existing code is not guaranteed, but most computations should work compatibly. Back-compatibility is a curse for software designers but important for users. My approach, like most, is not ideal: the compatibility inevitably slows down computation and at times seems the source for most of the bugs encountered. There may be room in the future for a different approach, perhaps an attempt to distill a "standard" S language that could be implemented with less regard to back-compatibility.

The general philosophy of classes and methods in Version 4 is one of gradual evolution from casual, interactive definition of S functions towards substantial software projects.

3 Chapters; Dynamic Linking.

For purposes of programming, the fundamental unit in version 4 of S is the *chapter*, a collection of software for use with S. This can include S functions, methods, class definitions, documentation, and other S objects. It can also include subprograms in lower-level languages, typically C or Fortran. The concept of a chapter unifies these various elements. From the user's view, attaching a chapter makes accessible all the facilities in that chapter. S handles automatically the details of where and how data objects, class/method definitions, documentation and other information are stored. Where earlier versions of S viewed the `attach()` function as attaching just the data sets (the `.Data` subdirectory), the new view is that the entire chapter is attached — details such as the subdirectory names are inferred by S.

In Version 4, the compiled version of C or Fortran routines are made available automatically by dynamic linking to the S process. When the chapter is attached, the S data manager looks for a specially named shared object file parallel to the directory of S objects (the `.Data` directory). If it finds such a file, it links the routines dynamically, making them accessible through the various interfaces. In Version 4, these include `.C()` and `.Fortran()` and a new form of interface designed to allow C programming with S objects, for advanced applications.

From the programmer's view, the procedure is to initialize a chapter directory with the

```
S CHAPTER
```

command. The command takes optional arguments which specify files of source or object code that should be linked. The `CHAPTER` command initializes a `makefile` in the directory. Subsequent `make` shell directives update the shared object. It is also possible (though not often necessary) to link a shared object explicitly by

```
dyn.open(file)
```

where `file` is the path name of the shared object.

4 Connections: Input/Output.

Version 4 of S provides a uniform approach to connections between the S process and input/output (e.g., files or pipes). The `connection` class of objects represents all such connections, with specific derived classes for `file`, `pipe`, `fifo`, S text objects, and potentially other kinds of connection. All the basic S functions doing input and output of character data take a connection argument. Input functions such as `parse()`, `scan()`, or `read.table()` and output functions such as `print()`, `cat()`, or `dump()` can be used much more flexibly as a result, while remaining compatible with earlier versions of S.

The general strategy for connection arguments is to check whether the connection is currently open. If it is not, the function opens it and arranges to close it on exit. If the connection is already open (and allows the reading or writing the function wants to do), it is left open on exit. With this assumption, programmers have a great deal of flexibility in dealing with connections. In addition to basic input/output operations, S programming can make use of functions to manipulate the connection object and to query it in various ways.

The connection concept is quite simple, but it has surprisingly high leverage, particularly in combination with some of the other new features of Version 4. It provides a wide variety of facilities that were either unavailable previously, or required unfortunate kludges such as spawning another S process. Much of the power comes from the class/method mechanism in Version 4. For example, the `connection` class itself is a virtual class. All actual connections come from specific classes such as `file` or `fifo` that extend `connection`. This allows many methods to be designed once for all connections while still leaving the programmer flexibility to make use of the properties of particular connection classes.

Some further extensions to input and output are related to connections. Standard input, output and message connections are built in. The `sink()` mechanism in S is greatly extended to allow diversion of any of these standard connections, including nested diversions. The argument to `sink()` is again a connection, providing substantial new flexibility.

5 Event Management: Readers and Monitors.

S evolved in the "glass teletype" era of user interaction, with an implicit model in which the user types expressions to S and S evaluates the expressions and prints back output, prompts, and other information. A characteristic of this model is that, when idle, the S process waits for user input, inside its own reader. This is not an adequate approach if S is to be more flexible in the design of its user interface.

A goal of Version 4 of S is to support user interface programming; that is, the design of interfaces to S, perhaps for specialized groups of users, that replace or supplement the standard user interaction. Such interfaces will often make use of graphical user interface tools (pop-up dialogues, list boxes, etc.). To support such programming, Version 4 of S is *event-driven*. The standard user interface waits in a loop for events. Typed user input is one such event, but a wide variety of other events can be specified.

The programmer defines the relevant events in S, specifying *readers*, which look for input from an S connection, or *monitors*, to be called after a specified timeout. Readers and monitors carry *actions*, in the form of S functions. The reader action takes the corresponding connection as an argument, and often responds by reading data from the connection. Programming events in S provides a very high-level interface, with many tools and the S evaluation model available.

Because connections are very general, readers can be programmed to carry out many useful tasks, in addition to those related to user interfaces. For example, readers can detect data written to a fifo and respond by reading the data into S. This provides an automatic input mechanism. Existing and new S functions provide a variety of mechanisms for handling the reader connection (`parse()`, `scan()`, `read.table()`, `readLines()`). If these are not enough, C-language tools to deal with connections are provided as well.

Events can be detected during an S task, as well as between tasks. The S evaluator can be instructed to check periodically for a waiting event. Using this mechanism, individual S functions can program user interface events using readers and monitors.

6 Online Documentation.

Two main enhancements have been made here. All S function and class definitions are now self-documenting and S on-line help is now driven entirely from S objects, rather than text files.

If the user requests documentation for a function and no explicit documentation exists, S will now construct documentation of the function call and argument defaults from the function itself. If there are initial comments in the function definition, these will be used as a description of the function. Provided programmers just remember to add a few lines of comments, functions can be usefully documented from their first writing.

Detailed documentation in S has previously been stored on separate text files, in a stylized form of the `nroff/troff` language. Shell commands were used to print the documentation. In Version 4, documentation is stored in S objects. Additional objects relate the documentation to topics and control printing and other operations, all through functions in S itself. There is no longer a tie to `nroff/troff`: the documentation uses commands whose meaning can be defined in S style objects. The intent is that the documentation be largely independent of the external word-processing tools, so that interfaces to new methods for viewing and processing the documentation can be written. In so far as there is a bias, the documentation commands follow a TEX/LATEXstyle.

The change to documentation objects and related S code is aimed at both specific and general goals. Generally, Version 4 works at unifying various aspects of S and bringing them into the central paradigm of S functions and methods operating on S objects. Online documentation was a conspicuous and ugly exception to this unity. The dependence on a number of external shell programs and commands linked S documentation to the existence and the limitations of these commands. The philosophy of internal uniformity — everything is an S object — has been a powerful one throughout the language, so applying it to documentation is desirable.

Specific benefits from the change include smoother interfaces for printing and other output. Style objects allow extensible control over the appearance of the documentation. The data objects are organized into S databases, using the same meta-data concepts and tools as for storing class/method information. Because documentation is in the form of S objects, specialized access can be programmed easily (for example, extracting the documentation of a particular argument to an S function is simple). As work proceeds on user-interface programming in S, flexible access to documentation will become increasingly useful.

Because the structure of S documentation objects can be related to the objects they describe, a number of automatic tools are possible or improved. Functions and class representations can provide some information directly; if no human-generated documentation is available, this information is supplied. (For example, the arguments and default values for a function will be printed.) The functions that produce initial outlines for documentation and that dump existing documentation for editing produce more readable text files for editing.

7 Databases and Data Communication.

S differs from many programming languages in explicitly including a form of database management for persistent objects. A database for S is a conceptual pairing of names and objects. There is a key duality between a named S object and a database: in both cases, individual character strings (names) identify objects, components in the first case and database objects in the second.

S began with the database concept tightly bound to an implementation using file system directories, with an object stored in a file of the same name. Increasingly this coupling has been loosened to avoid limitations and allow programmers to apply the S database idea more freely.

In Version 4, large libraries of S objects can be organized as single files, with their own on-file table, saving on disk space and avoiding implementation limitations. Earlier work on user-defined database classes has been retained and extended, based on a virtual `database` class.

While S databases, by default, store objects in binary format and so are not portable, S provides a mechanism to write and read objects in a symbolic format. Any S object can be unambiguously represented in this format and communicated without loss of accuracy. (Internally, the accuracy of numeric data is limited by the hardware, but the symbolic format does not enforce such limitations in contrast, e.g., to Java's virtual machine approach.) The symbolic format is explicitly defined in Version 4, so programmers can use it. For example, in some recent work by Mark Hansen and the author, this proved to be a nice basis for communication between S and Java.

8 Portability and Implementation.

Portability of a large, varied, interactive system such as S is inevitably a complicated issue, across different hardware configurations and even more across different operating systems and variants. The 1992 version of S was designed to run on a number of variants of the Unix operating system, using a set of files for each hardware and software version plus in some cases some hand-tuning. The overall goal for Version 4 is to be portable over a more clearly defined operating system model, with specialization (e.g., for particular hardware) centralized and much reduced. There are three key points in the new strategy.

1. We will take advantage of evolving standards (e.g., Posix) but will go cautiously beyond them for important gains.
2. The contents of the S source has been much simplified compared to earlier versions, both in organization and in the languages and tools used.
3. Our immediate target system is Unix, in what we predict to be a fairly standard version for the immediate future. In addition, there is a hope and intent (not quite a promise) that the system will port also to a version of Microsoft Windows NT.

So far, the beta-test code for Version 4 runs on Sun (both OS4 and Solaris), SGI, Linux, and HP-UX. A partial port of an earlier stage was made to Windows NT.

9 Large Objects.

Research has been continuing, off and on, into techniques for allowing large(r) objects to be used in S. A balance has to be struck between allowing special techniques for such objects, on the one hand, and preserving the semantics and the generality of S on the other. That is, one could easily invent a few techniques that would work for some large objects by doing low-level calculations using the traditional one-row-at-a-time input. More helpful in the long run will be modifications that allow reasonable current computations to work on substantially larger objects, and provide more graceful slowdown of performance as the objects get larger or the computations more complicated. There is no likely magic technique here, but a number of recent and current modifications are promising.

The 1992 release included a number of such changes, notably a form of *copy-on-modify* for entire S objects. That is, if `mydata` is an object on my working data supplied as an argument to `myfun()`, then no copy is made of `mydata` inside the function unless the corresponding local object is modified. Version 4 extends this to the concept of *data sharing*: any object and any slot or component of an object may be shared during evaluation. So, for example, inserting a vector as a slot in an object does not require copying the vector. Only when elements within a vector change is copying required. (That this was not done with earlier versions of S results again from back-compatibility: much code was written using the original S model for data, and this code had to be extensively revised to allow data sharing.) In addition, Version 4 uses memory mapping to reduce the cost of working with large objects, to the extent that this mechanism delays reading all of an atomic vector into memory, carries out the access more efficiently, and, to some extent, provides more efficient "copying".

Some more specialized techniques are also supported; for example, non-S binary files can also use memory mapping, which may possibly help in sharing large amounts of data with other software.

10 Semantic Analysis.

Version 4 of S provides the operation of *semantic analysis*. Explicit input to the analysis is an S object, typically a function or a method definition. Semantic analysis returns another S object, which can be used in place of the first. The returned object is a *semantic method*. The method has a `body` slot containing the analyzed version (i.e., the "compiled" version) of the original definition. The rest of the object's slots contain various information useful for further semantic analysis, but also for debugging and other purposes. The analysis uses all current information about methods, classes, and function definitions to precompute information in the function.

The goal is to produce equivalent S code that can be evaluated more efficiently than the original. Just how central a role semantic analysis will play remains uncertain. While "compilation" is very much in vogue these days for interactive systems, its effectiveness depends partly on the underlying computational model. The lower the level (the closer to the machine) and the less general the model, the more one can expect from compilation. S has a model that is relatively far from the machine. We can still hope to get significant gains from semantic analysis; a number of the ideas have, in fact, been taken back into the standard evaluator and made automatic. The issues of machine efficiency are complicated, and although important, should not be allowed to harm the prime goal of turning ideas into software.

11 The Parser; Syntactic Extensions.

A number of changes have been made to the parser, mostly to extend the language itself (that is, the input the parser accepts). The purposes are to smooth over initial conceptual barriers, particularly for those accustomed to languages in the C/C++ family, to increase readability, and to allow for future work on other styles of user interface. Of course, syntactic extensions do not increase what the language can fundamentally do, but they can be important for users all the same.

Re-entrant Parsing. The S parser can be asked to return an incompletely parsed expression. The returned object can then be handed back to the parser, which will resume parsing as if the user had re-typed this portion of the expression. This is useful for user interface design.

Blanks in Names. S names now consist of one or more *words*. Previously, only one word was allowed in a name, but the Version 4 parser will recognize a sequence of words, separated by blanks. The parser keeps one blank between successive words. This is just a change to help readability, but we will try to use words meaningfully (for example, names starting with `set` are used for functions that set information, such as class or method definitions).

C-style Assignments. The parser now accepts `"="` as an assignment symbol, in addition to `"<-"`. The two should be semantically equivalent, with the one exception (required for back compatibility) that named arguments in a function call work the same way they always did. The goal here is simply to make the language look more familiar to the initial user, who may very likely be familiar with this assignment operator from the C-style family of languages or even from Fortran.

It's a fair question whether the extension muddies the distinction from named arguments in calls, which have always used `"="`. I think in fact the situation is simpler, particularly for beginners. The distinction between assignments and named arguments is actually subtler than it might seem, since the latter really are assignments but of an *unevaluated* expression and in the frame of the called function, not that of calling function. Such distinctions are best left until fairly advanced discussion, however. For initial use, the notion of associating a name with an expression is specific enough for both purposes.

Un-reserving Words. The lexical analyzer and parser try harder to allow some of the reserved words (`function`, `for`, `in`, etc.) in contexts where they ought to be unambiguous. The most important such is before `"="` in either the call to or the definition of a function. This frees such words to be function arguments. A second context is as following words in multi-word names. (This context can't be totally disambiguated: consider the parser's dilemma with

```
for(love in bloom in May) { ... }
```

It turns out `in` and `else` can't be allowed in names.)

A related extension is to allow the names of formal arguments to be quoted. This allows any string to be used as an argument name (quoting was always allowed in naming arguments in calls).

12 Miscellaneous.

A number of smaller-scale changes have been made in Version 4 as well. Here are some of them.

Debugging. The default error handler in Version 4 is the function `recover()`. This gives the user the option to debug immediately, without having to specify previously an option and then to invoke a debugger separately. If the user chooses to debug, a browser is entered in the nearest interesting evaluation frame to where the error occurred. Simple navigational and other commands are provided to examine data, move around in the evaluation structure or dump information for future use. The result is a simpler interface that gives the programmer access right at the time of the error, not from a dump. Some simple commands have been added to `recover()` and `browser()`: `up` and `down` navigate the traceback list of calling functions, and `q` quits from the debugging session.

Initialization. The initialization procedure has been extended to look for the file `".S.init"` in the user's login directory. If this file exists, it is parsed and evaluated at the start of the session (before looking for a `.First` object). The goal is to support user customization independent of the current location when S is invoked. Errors in initialization no longer terminate the S session; initialization is viewed as a task, so an error just terminates the task.

Graphics Devices. Duncan Temple Lang and I have implemented devices as classes of S objects (using virtual classes to organize simlar classes of device). Among other things, this provides multiple graphics devices and facilities for copying graphics between devices. There is also a general facility for defining the default graphics device.

Evaluation. S has an explicit model for evaluation. In version 4, evaluation has been revised to make the model and reality match better, for example by using a single technique throughout for name-matching in databases and during evaluation. The overall goal is cleaner and usually more efficient code.

Testing. The function `do.test()` has been extended. It now provides a general facility for running regression-test files, including a facility for stopping (for debugging) if a test fails. There is also a new function `psdiff()` (in library `test`) that attempts a semi-intelligent analysis of two postscript graphics output files that claim to be identical. In the past, only Unix tools were used for this and such tools are hopeless in the face of any minor differences.

References

[1] John M. Chambers and Trevor Hastie (eds). *Statistical Models in S.* Chapman and Hall, 1993.

[2] J. M. Chambers. Classes and methods in S. I: Recent developments. *Computatonal Statistics*, 8(3):167–184, March 1993.

[3] R. A. Becker, J. M. Chambers, and A. R. Wilks. *The New S Language.* Chapman and Hall, 1988.