S-mode

version 4.8

Doug Bates, Ed Kademan, Frank Ritter and David Smith

An GNU Emacs package
for interacting with the
S/Splus statistical software packages

Documentation by David Smith
(D.M.Smith@lancaster.ac.uk)
Department of Mathematic and Statistics
Lancaster University, UK

Documentation version: 1.7

1 Introduction to S-mode

The S and Splus packages provide sophisticated statistical and graphical routines for manipulating data. The S-mode package provides useful routines for making the use of these packages much easier.

A bit of notation before we begin. I will refer to both the 'new S' package (as described in Becker, Chambers and Wilks, *The New S Language: A programming environment for data analysis and graphics*) and 'Splus' (an enhanced version of new S from Statsci) simply by "S". The interface which is used to run S under Emacs (which this manual documents) will be referred to as "S-mode", which should not be confused with the GNU Emacs major mode `S-mode` which is used for editing S source.

For exclusively interactive users of S, S-mode provides a number of features to make life easier. There is an easy-to-use command history mechanism, including a quick prefix-search history. To reduce typing, command-line completion is provided for all S objects and "hot keys" are provided for common S function calls. Help files are easily accessible, and a paging mechanism is provided to view them. Finally, an incidental (but very useful) side-effect of S-mode is that a transcript of your session is kept for later saving or editing. No special knowledge of Emacs is necessary when using S interactively under S-mode.

For those that use S in the typical edit-test-revise cycle when programming S functions, S-mode provides for editing of S functions in Emacs edit buffers. Unlike the typical use of S where the editor is restarted every time an object is edited, S-mode uses the current Emacs session for editing. In practical terms, this means that you can edit more than one function at once, and that the S process is still available for use while editing. Error checking is performed on functions loaded back into S, and a mechanism to jump directly to the error is provided. S-mode also provides for maintaining text versions of your S functions in specified source directories.

1.1 Why should I use S-mode?

S is a powerful system for manipulating and analysing data, but its user interface — particularly on Unix platforms — leaves something to be desired. S-mode is designed as a package that makes S easier to use.

S-mode provides several features which make it easier to interact with the S process (i.e. enter commands and view the output). These include:

- **Command-line editing** for fixing mistakes in commands before they are entered. The '`-e`' flag for S-plus provides something similar to this, but here you have the full range of Emacs commands rather than a limited subset. See Section 3.1 [Command-line editing], page 6.
- **Searchable command history** for recalling previously-submitted commands. This provides all the features of the '`Splus -e`' history mechanism, plus added features such as history searching. See Section 3.5 [Command History], page 11.
- **Command-line completion** of object and file names for quick entry. This is similar to `tcsh`'s facility for filenames; here it also applies to object names and list components. See Section 3.2 [Completion], page 6.

- **Hot-keys** for quick entry of commonly-used commands such as `objects()` and `search()`. See Section 3.7 [Hot keys], page 13.
- **Transcript recording** for a complete record of all the actions in an S session. See Section 3.4 [Transcript], page 8.
- **Interface to the help system**, with a specialised mode for viewing S help files. See Chapter 5 [Help], page 22.

If, like me, you commonly create or modify S functions, you will have found the standard facilities for this (the '`fix()`' function, for example) severely limiting. Using S's standard features, you can only edit one function at a time, and you can't continue to use S while editing. S-mode corrects these problems by introducing the following features:

- **Object editing**. S-mode allows you to edit more than one function simultaneously in dedicated Emacs buffers. The S process may continue to be used while functions are being edited. See Section 4.1 [Edit buffer], page 15.
- **A specialised editing mode** for S code, which provides syntactic indentation and highlighting. See Section 4.5 [Indenting], page 17.
- **Facilities for loading and error-checking source files**, including a keystroke to jump straight to the position of an error in a source file. See Section 4.3 [Error Checking], page 16.
- **Source code revision maintenance**, which allows you to keep historic versions of S source files. See Section 4.7 [Source Files], page 19.
- **Facilities for evaluating S code** such as portions of source files, or line-by-line evaluation of files (useful for debugging). See Section 4.4 [Evaluating code], page 16.

Finally, S-mode provides features for re-submitting commands from saved transcript files, including:

*** Not yet written ***

1.2 Authors of and contributors to S-mode

S-mode is based on Olin Shivers' excellent comint package (which is supplied with version 19 of FSF GNU Emacs). The original version of S-mode was written by Doug Bates (`bates@stat.wisc.edu`) and Ed Kademan (`kademan@stat.wisc.edu`). Frank Ritter (`ritter@psy.cmu.edu`) then merged this version with his own S mode to form `S.el` version 2.1.

Version 2.1 of S.el was then updated and expanded by David Smith to form version 3.4. This was then updated for Emacs 19 to create version 4. Most bugs have now been fixed (and several new ones introduced) and many new features have been added. Thanks must go to the many people who have helped with the development of the present version of S-mode:

- The multiple process code, and the idea for `S-eval-line-and-next-line` are by Rod Ball.
- Thanks to Doug Bates for many useful suggestions, and an FTP home for the code.
- Thanks to Martin Maechler for reporting and fixing bugs, providing many useful comments and suggestions, and for maintining the S-mode mailing list.

- Thanks to Frank Ritter for updates from the previous version, the menu code, and invaluable comments on the manual.
- Thanks to Ken'ichi Shibayama for his excellent indenting code, and many comments and suggestions.
- Last but definitely not least, thanks to the many beta testers of the S-mode mailing list.

1.3 Getting the latest version of S-mode

The latest version is always available via WWW from:

- `http://www.maths.lancs.ac.uk:2080/~maa036/elisp/S-mode/`

Recent versions of S-mode are also available for anonymous FTP from the following sites:

- `/anonymous@wingra.stat.wisc.edu:pub/src/emacs-lisp`
- `/anonymous@attunga.stats.adelaide.edu.au:pub/S-mode`

Check the `README` file first to see which files you need. S-mode is also available from the Emacs-Lisp archive on `archive.cis.ohio-state-edu` — retrieve

‘`pub/gnu/emacs/elisp-archive/README`’

for information on the archive. An older version is also available from Statlib by sending a blank message with subject “send index from S” to `statlib@stat.cmu.edu`, and following the directions from there.

Note that all new user-visible features to versions of S-mode are documented in the ‘`NEWS`’ file; all changes are listed in the ‘`ChangeLog`’ file.

1.4 How to read this manual

If S-mode has already been installed on your system, the next chapter has details on how to get started using S under S-mode.

If you need to install S-mode, read Appendix A [Installation], page 28 for details on what needs to be done before proceeding to the next chapter.

Appendix B [Customisation], page 30 provides details of user variables you can change to customize S-mode to your taste, but it is recommended that you defer this section until you are more familiar with S-mode.

Don't forget that this manual is not the only source of information about S-mode. In particular, the mode-based online help (obtained by pressing *C-h m* when in the process buffer, edit buffer or help buffer) is quite useful. However the best source of information is, as always, experience — try it out!

2 Starting the S process

To start an S session, simply type *M-x S RET*, i.e. press ⟨ESC⟩, then ⟨x⟩, then capital ⟨S⟩ and then the ⟨RETURN⟩ key.

S will then (by default) ask the question

`S starting data directory?`

Enter the name of the directory you wish to start S from (that is, the directory you would have `cd`'d to before starting S from the shell). This directory should have a '`.Data`' subdirectory.

You will then be popped into a buffer with name '`*S*`' which will be used for interacting with the S process, and you can start entering commands.

2.1 Running more than one S process

S-mode allows you to run more than one S process simultaneously in the same session. Each process has a name and a number; the initial process (process 1) is simply named '`S`'. You may start a new process by passing a numeric argument to *M-x S*. For example, typing *ESC 2 M-x S* starts up an S process with name '`S2`', in a buffer whose name is initially '`*S2*`'. The name of the process is shown in the mode line in square brackets (for example, '`[S2]`'); this is useful if the process buffer is renamed. Without a prefix argument, *M-x S* starts a new S process, using the first available process number.

You can switch to any active S process with the command *C-c C-k* (`S-request-a-process`). Just enter the name of the process you require; completion is provided over the names of all running S processes. This is a good command to bind to a global key.

2.2 Changing the startup actions

If you do not wish S-mode to prompt for a starting directory when starting a new process, set the variable `S-ask-for-S-directory` to `nil`. In this case, the value of the variable `S-directory` is used as the starting directory. The default value for this variable is your home directory. If `S-ask-for-S-directory` has a non-`nil` value (as it does by default) then the value of `S-directory` provides the default when prompting for the starting directory. Incidentally, `S-directory` is an ideal variable to set in `S-pre-run-hook`.

If you like to keep a records of your S sessions, set the variable `S-ask-about-transfile` to `t`, and you will be asked for a filename for the transcript before the S process starts.

S-ask-about-transfile User Option

If non-`nil`, as for a file name in which to save the session transcript.

Enter the name of a file in which to save the transcript at the prompt. If the file doesn't exist it will be created (and you should give it a file name ending in '`.St`'; if the file already exists the transcript will be appended to the file. (Note: if you don't set this variable but you still want to save the transcript, you can still do it later — see Section 3.4.4 [Saving transcripts], page 10.)

Once these questions are answered (if they are asked at all) the S process itself is started by calling the program name specified in the variable `inferior-S-program`. If you need to pass any arguments to this program, they may be specified in the variable **inferior-***S`program`name***-args** (e.g. if `inferior-S-program` is `"S+"` then the variable to set is `inferior-S+-args`. It is not normally necessary to pass arguments to the S program; in particular do not pass the '`-e`' option to `Splus`, since S-mode provides its own command history mechanism.

3 Interacting with the S process

The primary function of the S-mode package is to provide an easy-to-use front end to the S interpreter. This is achieved by running the S process from within an Emacs buffer, so that the Emacs editing commands are available to correct mistakes in commands, etc. The features of Inferior S mode are similiar to those provided by the standard Emacs shell mode (see section "Shell Mode" in *The Gnu Emacs Reference Manual*). Command-line completion of S objects and a number of 'hot keys' for commonly-used S commands are also provided for ease of typing.

3.1 Entering commands and fixing mistakes

Sending a command to the S process is as simple as typing it in and pressing the ⟨RETURN⟩ key:

- *RET* (`inferior-S-send-input`)
  Send the command on the current line to the S process.

If you make a typing error before pressing *RET* all the usual Emacs editing commands are available to correct it (see section "Basic editing commands" in *The GNU Emacs Reference Manual*). Once the command has been corrected you can press ⟨RETURN⟩ (even if the cursor is not at the end of the line) to send the corrected command to the S process.

S-mode provides some other commands which are useful for fixing mistakes:

- *C-c C-w* (`backward-kill-word`)
  Deletes the previous word (such as an object name) on the command line.
- *C-c C-u* (`comint-kill-input`)
  Deletes everything from the prompt to point. Use this to abandon a command you have not yet sent to the S process.
- *C-c C-a* (`comint-bol`)
  Move to the beginning of the line, and then skip forwards past the prompt, if any.

See section "Shell Mode" in *The Gnu Emacs Reference Manual*, for other commands relevant to entering input.

3.2 Completion of object names

In the process buffer, the ⟨TAB⟩ key is for completion, similar to that provided by Shell Mode for filenames. In Inferior S mode, pressing the ⟨TAB⟩ key when the cursor is following the first few characters of an object name *completes* the object name; if the cursor is following a file name *TAB* completes the file name.

- *TAB* (`comint-dynamic-complete`)
  Complete the S object name or filename before point.

When the cursor is just after a partially-completed object name, pressing ⟨TAB⟩ provides completion in a similar fashion to `tcsh` except that completion is performed over all known S object names instead of file names. S-mode maintains a list of all objects known to S at any given time, which basically consists of all objects (functions and datasets) in every attached directory listed by the `search()` command along with the component objects of attached data frames (if your version of S supports them).

For example, consider the three functions (available in Splus version 3.0) called `binomplot()`, `binom.test()` and `binomial()`. Typing *bin TAB* after the S prompt will insert the characters '`om`', completing the longest prefix ('`binom`') which distinguishes these three commands. Pressing *TAB* once more provides a list of the three commands which have this prefix, allowing you to add more characters (say, '`.`') which specify the function you desire. After entering more characters pressing *TAB* yet again will complete the object name up to uniqueness, etc. If you just wish to see what completions exist without adding any extra characters, type *M-?*.

- *M-?* (`S-list-object-name-completions`)
  List all possible completions of the object name at point.

S-mode also provides completion over the components of named lists accessed using the '`$`' notation, to any level of nested lists. This feature is particularly useful for checking what components of a list object exist while partway through entering a command: simply type the object name and '`$`' and press *TAB* to see the names of existing list components for that object.

Completion is also provided over file names, which is particularly useful when using S functions such as `get()` or `scan()` which require fully expanded file names. Whenever the cursor is within an S string, pressing *TAB* completes the file name before point, and also expands any '`~`' or environment variable references.

If the cursor is not in a string and does not follow a (partial) object name, the ⟨TAB⟩ key has a third use: it expands history references. See Section 3.6 [History expansion], page 11.

3.3 Completion details

S-mode automatically keeps track of any objects added or deleted to the system (such as new objects created, or directories added to the search list) to make completion as accurate as possible. Whenever S-mode notices that search list has changed [1] when you attach a directory or data frame, the objects associated with it immediately become available for a completion; when it is detached completion is no longer available on those objects.

To maintain a list of accessible objects for completion, S-mode needs to determine which objects are contained in each directory or data frame on the search list. This is done at the start of each S session, by running the `objects()` command on every element of the search list. On some systems, however, this can be rather slow; it's doubly frustrating when you consider that most of the directories on the search list are the standard S libraries, which never change anyway! When S-mode was installed, a database of the standard object names should have been created which should speed up this process at the start of an S session; if it has not been created you will get a warning like 'S-namedb.el does not exist'. See Appendix A [Installation], page 28, for information on how to create this database.

---

[1] The variable `S-change-sp-regex` is a regular expression matching commands which change the search list. You will need to modify this variable if you have defined custom commands (other than `attach`, `detach`, `collection` or `library`) which modify the search list.

Efficiency in completion is gained by maintaining a cache of objects currently known to S; when a new object becomes available or is deleted, only one component of the cache corresponding to the associated directory needs to be refreshed. If S-mode ever becomes confused about what objects are available for completion (such as when if refuses to complete an object you **know** is there), the command *M-x S-resynch* forces the *entire* cache to be refreshed, which should fix the problem.

3.4 Manipulating the transcript

Most of the time, the cursor spends most of its time at the bottom of the S process buffer, entering commands. However all the input and output from the current (and previous) S sessions is stored in the process buffer (we call this the transcript) and often we want to move back up through the buffer, to look at the output from previous commands for example.

Within the process buffer, a paragraph is defined as the prompt, the command after the prompt, and the output from the command. Thus *M-{* and *M-}* move you backwards and forwards, respectively, through commands in the transcript. A particularly useful command is *M-h* (`mark-paragraph`) which will allow you to mark a command and its entire output (for deletion, perhaps). For more information about paragraph commands, see section "Paragraphs" in *The GNU Emacs Reference Manual*.

If an S process finishes and you restart it in the same process buffer, the output from the new S process appears after the output from the first S process separated by a form-feed ('`^L`') character. Thus pages in the S process buffer correspond to S sessions. Thus, for example, you may use *C-x [* and *C-x]* to move backward and forwards through S sessions in a single S process buffer. For more information about page commands, see section "Pages" in *The GNU Emacs Reference Manual*.

3.4.1 Manipulating the output from the last command

Viewing the output of the command you have just entered is a common occurence and S-mode provides a number of facilities for doing this. Whenever a command produces a longish output, it is possible that the window will scroll, leaving the next prompt near the middle of the window. The first part of the command output may have scrolled off the top of the window, even though the entire output would fit in the window if the prompt were near the bottom of the window. If this happens, you can use the command

- *C-c C-e* (`comint-show-maximum-output`)
  Move to the end of the buffer, and place cursor on bottom line of window.

to make more of the last output visible. (To make this happen automatically for all inputs, set the variable `comint-scroll-to-bottom-on-input` to `t`; for information on this and other options for handling process input and output see section "Shell Mode Options" in *The GNU Emacs Reference Manual*.)

If the first part of the output is still obscured, use

- *C-c C-r* (`comint-show-output`)
  Moves cursor to the previous command line and and places it at the top of the window.

to view it. Finally, if you want to discard the last command output altogether, use

- *C-c C-o* (`comint-kill-output`)
  Deletes everything from the last command to the current prompt.

to delete it. Use this command judiciously to keep your transcript to a more manageable size.

3.4.2 Viewing more historic commands

If you want to view the output from more historic commands than the previous command, commands are also provided to move backwards and forwards through previously entered commands in the process buffer:

- *C-c C-p* (`comint-previous-input`)
  Moves point to the preceding command in the process buffer.
- *C-c C-n* (`comint-next-input`)
  Moves point to the next command in the process buffer.

Note that these two commands are analagous to *C-p* and *C-n* but apply to command lines rather than text lines. And just like *C-p* and *C-n*, passing a prefix arg to these commands means to move to the *ARG*'th next (or previous) command. (These commands are also discussed in section "Shell History Copying" in *The GNU Emacs Reference Manual*.)

There are also two similar commands (not bound to any keys by default) which move to preceding or succeeding commands, but which first prompt for a regular expression (see section "Syntax of Regular Expression" in *The GNU Emacs Reference Manual*), and then moves to the next (previous) command matching the pattern.

comint-backward-matching-input *regexp arg* Command
comint-forward-matching-input *regexp arg* Command

Search backward (forward) through the transcript buffer for the *arg*'th previous (next) command matching *regexp*. *arg* is the prefix argument; *regexp* is prompted for in the minibuffer.

3.4.3 Re-submitting commands from the transcript

When moving through the transcript, you may wish to re-execute some of the commands you find there. S-mode provides three commands to do this; these commands may be used whenever the cursor is within a command line in the transcript (if the cursor is within some command *output*, an error is signalled). Note all three commands involve the ⟨RETURN⟩ key.

- *RET* (`inferior-S-send-input`)
  Copy the command under the cursor to the current command line, and execute it.
- *C-c RET* (`comint-copy-old-input`)
  Copy the command under the cursor to the current command line, but don't execute it. Leaves the cursor on the command line so that the copied command may be edited.
- *M-RET* (`S-transcript-send-command-and-move`)
  Copy the command under the cursor to the current command line, and execute it. Moves the cursor to the following command.

When the cursor is not after the current prompt, the ⟨RETURN⟩ key has a slightly different behaviour than usual. Pressing *RET* on any line containing a command that you entered (i.e. a line beginning with a prompt) sends that command to the S process once again. If you wish to edit the command before executing it, use *C-c RET* instead; it copies the command to the current propt but does not execute it, allowing you to edit it before submitting it.

These two commands leave the cursor at the new command line, allowing you to continue with interactive use of S. If you wish to resubmit a series of commands from the transcript, consider using *M-RET* instead, which leaves the cursor at the command line following the one you re-submitted. Thus by using *M-RET* repeatedly, you can re-submit a whole series of commands.

These commands work even if if the current line is a continuation line (i.e. the prompt is '`+`' instead of '`>`') — in this case all the lines that form the multi-line command are concatenated together and the resulting command is sent to the S process (currently this is the only way to resubmit a multi-line command to the S process in one go). If the current line does not begin with a prompt, an error is signalled. This feature, coupled with the command-based motion commands described above, could be used as a primitive history mechanism. S-mode provides a more sophisticated mechanism, however, which is described in Section 3.5 [Command History], page 11.

3.4.4 Keeping a record of your S session

To keep a record of your S session in a disk file, use the Emacs command *C-x C-w* (`write-file`) to attach a file to the S process buffer. The name of the process buffer will (probably) change to the name of the file, but this is not a problem. You can still use S as usual; just remember to save the file before you quit Emacs with *C-x C-s*. You can make S-mode prompt you for a filename in which to save the transcript every time you start S by setting the variable `S-ask-about-transfile` to `t`; see Section 2.2 [Customizing startup], page 4. We recommend you save your transcripts with filenames that end in '`.St`'. There is a special mode (S transcript mode — see Chapter 6 [Transcript Mode], page 24) for editing transcript files which is automatically selected for files with this suffix.

S transcripts can get very large, so some judicious editing is appropriate if you are saving it in a file. Use *C-c C-o* whenever a command produces excessively long output (printing large arrays, for example). Delete erroneous commands (and the resulting error messages or other output) by moving to the command (or its output) and typing *M-h C-w*. Also, remember that *C-c C-e* (and other hot keys) may be used for commands whose output you do not wish to appear in the transcript. These suggestions are appropriate even if you are not saving your transcript to disk, since the larger the transcript, the more memory your Emacs process will use on the host machine.

Finally, if it is your intention to produce S source code (suitable for using with `source()` or inclusion in an S function) from a transcript, then the command *M-x S-clean-region* may be of use. This command works in any Emacs buffer, and removes all prompts and command output from an S transcript within the current region, leaving only the commands. Don't forget to remove any erroneous commands first!

3.5 Command History

S-mode provides easy-to-use facilities for re-executing or editing previous commands. An input history of the last few commands is maintained (by default the last 50 commands are stored, although this can be changed by setting the variable `comint-input-ring-size` in `inferior-S-mode-hook`.) The simplest history commands simply select the next and previous commands in the input history:

- *M-p* (`comint-previous-input`)
  Select the previous command in the input history.
- *M-n* (`comint-next-input`)
  Select the next command in the input history.

For example, pressing *M-p* once will re-enter the last command into the process buffer after the prompt but does not send it to the S process, thus allowing editing or correction of the command before the S process sees it. Once corrections have been made, press *RET* to send the edited command to the S process.

If you want to select a particular command from the history by matching it against a regular expression (see section "Syntax of Regular Expression" in *The GNU Emacs Reference Manual*), to search for a particular variable name for example, these commands are also available:

- *M-r* (`comint-previous-matching-input`)
  Prompt for a regular expression, and search backwards through the input history for a command matching the expression.
- *M-s* (`comint-next-matching-input`)
  Prompt for a regular expression, and search backwards through the input history for a command matching the expression.

A common type of search is to find the last command that began with a particular sequence of characters; the following two commands provide an easy way to do this:

- *A-M-r* (`comint-previous-matching-input-from-input`)
  Select the previous command in the history which matches the string typed so far.
- *A-M-s* (`comint-next-matching-input-from-input`)
  Select the next command in the history which matches the string typed so far.

Instead of prompting for a regular expression to match against, as they instead select commands starting with those characters already entered. For instance, if you wanted to re-execute the last `attach()` command, you may only need to type `att` and then *A-M-r* and *RET*. (Note: you may not have an ⟨ALT⟩ key on your keyboard, in which case it may be a good idea to bind these commands to some other keys.)

See section "Shell History Ring" in *The GNU Emacs Reference Manual*, for a more detailed discussion of the history mechanism.

3.6 References to historical commands

Instead of searching through the command history using the command described in the previous section, you can alternatively refer to a historical command directly using a notation very similar to that used in `csh`. History references are introduced by a '`!`' or '`^`' character and have meanings as follows:

'`!!`' The immediately previous command

'`!-`*N*' The *N*th previous command

'`!text`' The last command beginning with the string '`text`'

'`!?text`' The last command containing the string '`text`'

In addition, you may follow the reference with a *word designator* to select particular *words* of the input. A word is defined as a sequence of characters separated by whitespace. (You can modify this definition by setting the value of `comint-delimiter-argument-list` to a list of characters that are allowed to separate words and themselves form words.) Words are numbered beginning with zero. The word designator usually begins with a '`:`' (colon) character; however it may be omitted if the word reference begins with a '`^`', '`$`', '`*`' or '`-`'. If the word is to be selected from the previous command, the second '`!`' character can be omitted from the event specification. For instance, '`!!:1`' and '`!:1`' both refer to the first word of the previous command, while '`!!$`' and '`!$`' both refer to the last word in the previous command. The format of word designators is as follows:

'`0`' The zeroth word (i.e. the first one on the command line)

'*n*' The *n*th word, where *n* is a number

'`^`' The first word (i.e. the second one on the command line)

'`$`' The last word

'*x*`-`*y*' A range of words; '`-`*y*' abbreviates '`0-`*y*'

'`*`' All the words except the zeroth word, or nothing if the command had just one word (the zeroth)

'*x*`*`' Abbreviates *x*-$

'*x*`-`' Like '*x*`*`', but omitting the last word

In addition, you may surround the entire reference except for the first '`!`' by braces to allow it to be followed by other (non-whitespace) characters (which will be appended to the expanded reference).

Finally, S-mode also provides quick substitution; a reference like '`^old^new^`' means "the last command, but with the first occurrence of the string '`old`' replaced with the string '`new`'" (the last '`^`' is optional). Similarly, '`^old^`' means "the last command, with the first occurrence of the string '`old`' deleted" (again, the last '`^`' is optional).

To convert a history reference as described above to an input suitable for S, you need to *expand* the history reference, using the ⟨TAB⟩ key. For this to work, the cursor must be preceeded by a space (otherwise it would try to complete an object name) and not be within a string (otherwise it would try to complete a filename). So to expand the history reference, type *SPC TAB*. This will convert the history reference into an S command from the history, which you can then edit or press ⟨RET⟩ to execute.

For example, to execute the last command that referenced the variable `data`, type *!?data SPC TAB RET*.

3.7 Hot keys for common commands

S-mode provides a number of commands for executing the commonly used functions. These commands below are basically information-gaining commands (such as `objects()` or `search()`) which tend to clutter up your transcript and for this reason some of the hot keys display their output in a temporary buffer instead of the process buffer by default. This behaviour is controlled by the variable `S-execute-in-process-buffer` which, if non-`nil`, means that these commands will produce their output in the process buffer instead. In any case, passing a prefix argument to the commands (with *C-u*) will reverse the meaning of `S-execute-in-process-buffer` for that command, i.e. the output will be displayed in the process buffer if it usually goes to a temporary buffer, and vice-versa. These are the hot keys that behave in this way:

- *C-c C-x* (`S-execute-objects`)
  Sends the `objects()` command to the S process. A prefix argument specifies the position on the search list (use a negative argument to toggle `S-execute-in-process-buffer` as well). A quick way to see what objects are in your working directory.

- *C-c C-s* (`S-execute-search`)
  Sends the `search()` command to the S process.
- *C-c C-e* (`S-execute`)
  Prompt for an S expression, and evaluate it.

`S-execute` may seem pointless when you could just type the command in anyway, but it proves useful for 'spot' calculations which would otherwise clutter your transcript, or for evaluating an expression while partway through entering a command. You can also use this command to generate new hot keys using the Emacs keyboard macro facilities; see section "Keyboard Macros" in *The GNU Emacs Reference Manual*.

The following hot keys do not use `S-execute-in-process-buffer` to decide where to display the output — they either always display in the process buffer or in a separate buffer, as indicated:

- *C-c C-a* (`S-execute-attach`)
  Prompts for a directory to attach to the S process with the `attach()` command. If a numeric prefix argument is given it is used as the position on the search list to attach the directory; otherwise the S default of 2 is used. The `attach()` command actually executed appears in the process buffer.
- *C-c C-l* (`S-load-file`)
  Prompts for a file to load into the S process using `source()`. If there is an error during loading, you can jump to the error in the file with *C-x `* (`S-parse-errors`). See Section 4.3 [Error Checking], page 16 for more details.
- *C-c C-v* (`S-display-help-on-object`)
  Pops up a help buffer for an S object or function. See Chapter 5 [Help], page 22 for more details.
- *C-c C-q* (`S-quit`)
  Sends the `q()` command to the S process, and cleans up any temporary buffers (such as help buffers or edit buffers) you may have created along the way. Use this command

when you have finished your S session instead of simply typing `q()` yourself, otherwise you will need to issue the command *M-x S-cleanup* command explicitly to make sure that all the files that need to be saved have been saved, and that all the temporary buffers have been killed.

3.8 Other commands provided by inferior-S-mode

The following commands are also provided in the process buffer:

- *C-c C-c* (`comint-interrupt-subjob`)
  Sends a Control-C signal to the S process. This has the effect of aborting the current command.
- *C-c C-z* (`S-abort`)
  Sends a STOP signal to the S process, killing it immediately. It's not a good idea to use this, in general: Neither `q()` nor `.Last` will be executed and device drivers will not finish cleanly. This command is provided as a safety to `comint-stop-subjob`, which is usually bound to *C-c C-z*. If you want to quit from S, use *C-c C-q* (`S-quit`) instead.
- *C-c C-d* (`S-dump-object-into-edit-buffer`)
  Prompts for an object to be edited in an edit buffer. See Chapter 4 [Editing], page 15.

Other commands available is Inferior S mode are discussed in section "Shell Mode" in *The Gnu Emacs Reference Manual*.

4 Editing S functions

S-mode provides facilities for editing S objects within your Emacs session. Most editing is performed on S functions, although in theory you may edit datasets as well. Edit buffers are always associated with files, although you may choose to make these files temporary if you wish. Alternatively, you may make use of a simple yet powerful mechanism for maintaining backups of text representations of S functions. Error-checking is performed when S code is loaded into the S process.

4.1 Creating or modifying S objects

To edit an S object, type

- *C-c C-d* (`S-dump-object-into-edit-buffer`)
  Edit an S object in its own edit buffer.

from within the S process buffer (`*S*`). You will then be prompted for an object to edit: you may either type in the name of an existing object (for which completion is available using the *TAB* key), or you may enter the name of a new object. A buffer will be created containing the text representation of the requested object or, if you entered the name of a non-existent object at the prompt and the variable `S-insert-function-templates` is non-`nil`, you will be presented with a template defined by `S-function-template` which defaults to a skeleton function construct.

You may then edit the function as required. The edit buffer generated by `S-dump-object-into-edit-buffer` is placed in the `S-mode` major mode which provides a number of commands to facilitate editing S source code. Commands are provided to intelligently indent S code, evaluate portions of S code and to move around S code constructs.

Note: when you dump a file with *C-c C-d*, S-mode first checks to see whether there already exists an edit buffer containing that object and, if so, pops you directly to that buffer. If not, S-mode next checks whether there is a file in the appropriate place with the appropriate name (See Section 4.7 [Source Files], page 19) and if so, reads in that file. You can use this facility to return to an object you were editing in a previous session (and which possibly was never loaded to the S session). Finally, if both these tests fail, the S process is consulted and a `dump()` command issued. If you want to force S-mode to ask the S process for the object's definition (say, to reformat an unmodified buffer or to revert back to S's idea of the object's definition) pass a prefix argument to `S-dump-object-into-edit-buffer` by typing *C-u C-c C-d*.

4.2 Loading source files into the S process

The best way to get information — particularly function definitions — into S is to load them in as source file, using S's `source` function. You have already seen how to create source files using *C-c C-d*; S-mode provides a complementary command for loading source files (even files not created with S-mode!) into the S process:

- *C-c C-l* (`S-load-file`)
  Loads a file into the S process using `source()`.

After typing *C-c C-l* you will prompted for the name of the file to load into S; usually this is the current buffer's file which is the default value (selected by simply pressing *RET* at the prompt). You will be asked to save the buffer first if it has been modified (this happens automatically if the buffer was generated with *C-c C-d*). The file will then be loaded, and if it loads successfully you will be returned to the S process.

4.3 Detecting errors in source files

If any errors occur when loading a file with `C-c C-l`, S-mode will inform you of this fact. In this case, you can jump directly to the line in the source file which caused the error by typing *C-x `* (`S-parse-errors`). You will be returned to the offending file (loading it into a buffer if necessary) with point at the line S reported as containing the error. You may then correct the error, and reload the file. Note that none of the commands in an S source file will take effect if any part of the file contains errors.

Sometimes the error is not caused by a syntax error (loading a non-existent file for example). In this case typing *C-x `* will simply display a buffer containing S's error message. You can force this behaviour (and avoid jumping to the file when there *is* a syntax error) by passing a prefix argument to `S-parse-errors` with *C-u C-x `*.

4.4 Sending code to the S process

Other commands are also available for evaluating portions of code in the S process. These commands cause the selected code to be evaluated directly by the S process as if you had typed them in at the command line; the `source()` function is not used. You may choose whether both the commands and their output appear in the process buffer (as if you had typed in the commands yourself) or if the output alone is echoed. The behaviour is controlled by the variable `S-eval-visibly-p` whose default is `nil` (display output only). Passing a prefix argument (*C-u*) to any of the following commands, however, reverses the meaning of `S-eval-visibly-p` for that command only — for example *C-u C-c C-j* echoes the current line of S-code in the S process buffer, followed by its output. This method of evaluation is an alternative to S's `source()` function when you want the input as well as the output to be displayed. (You can sort of do this with `source()` when the option `echo=T` is set, except that prompts do not get displayed. S-mode puts prompts in the right places.) The commands for evaluating code are:

- *C-c C-j* (`S-eval-line`)
  Send the line containing point to the S process.
- *C-c M-j* (`S-eval-line-and-go`)
  As above, but returns you to the S process buffer as well.
- *C-c C-f* or *ESC C-x* (`S-eval-function`)
  Send the S function containing point to the S process.
- *C-c M-f* (`S-eval-function-and-go`)
  As above, but returns you to the S process buffer as well.
- *C-c C-r* (`S-eval-region`)
  Send the text between point and mark to the S process.

- *C-c M-r* (`S-eval-region-and-go`)
  As above, but returns you to the S process buffer as well.
- *C-c C-b* (`S-eval-buffer`)
  Send the contents of the edit buffer to the S process.
- *C-c M-b* (`S-eval-buffer-and-go`)
  As above, but returns you to the S process buffer as well.
- *C-c C-n* (`S-eval-line-and-next-line`)
  Sends the current line to the S process, echoing it in the process buffer, and moves point to the next line. Useful when debugging for stepping through your code.

It should be stressed once again that these `S-eval-` commands should only be used for evaluating small portions of code for debugging purposes, or for generating transcripts from source files. When editing S functions, *C-c C-l* is the command to use to update the function's value. In particular, `S-eval-buffer` is now largely obsolete.

One final command is provided for spot-evaluations of S code:

C-c C-e (`S-execute-in-tb`)
Prompt for an S expression and evaluate it. Displays result in a temporary buffer.

This is useful for quick calculations, etc.

All the above commands are useful for evaluating small amounts of code and observing the results in the process buffer. A useful way to work is to divide the frame into two windows; one containing the source code and the other containing the process buffer. If you wish to make the process buffer scroll automatically when the output reaches the bottom of the window, you will need to set the variable `comint-scroll-to-bottom-on-output` to `others` or `t`.

*** Maybe a link to customisation section here ***

4.5 Indenting and formatting S code

S-mode now provides a sophisticated mechanism for indenting S source code (thanks to Ken'ichi Shibayama). Compound statements (delimited by '`{`' and '`}`') are indented relative to their enclosing block. In addition, the braces have been electrified to automatically indent to the correct position when inserted, and optionally insert a newline at the appropriate place as well. Lines which continue an incomplete expression are indented relative to the first line of the expression. Function definitions, `if` statements, calls to `expression()` and loop constructs are all recognised and indented appropriately. User variables are provided to control the amount if indentation in each case, and there are also a number of predefined indentation styles to choose from. See Section B.1.3 [Indentation variables], page 31.

Comments are also handled specially by S-mode, using an idea borrowed from the Emacs-Lisp indentation style. Comments beginning with '`###`' are aligned to the beginning of the line. Comments beginning with '`##`' are aligned to the current level of indentation for the block containing the comment. Finally, comments beginning with '`#`' are aligned to a column on the right (the 40th column by default, but this value is controlled by the variable `comment-column`,) or just after the expression on the line containing the comment if it extends beyond the indentation column.

The indentation commands provided by S-mode are:

- *TAB* (`S-indent-command`)
  Indents the current line as S code. If a prefix argument is given, all following lines which are part of the same (compound) expression are indented by the same amount (but relative indents are preserved).
- *LFD* (`newline-and-indent`)
  Insert a newline, and indent the next line. (Note: if your keyboard does not have a ⟨LINEFEED⟩ key, you can use *C-j* instead.) Some people prefer to bind ⟨RET⟩ to this command.
- *ESC C-q* (`S-indent-exp`)
  Indents each line in the S (compound) expression which follows point. Very useful for beautifying your S code.
- `{` and `}` (`S-electric-brace`)
  The braces automatically indent to the correct position when typed.
- *M-;* (`indent-for-comment`)
  Indents a comment line appropriately, or inserts an empty (single-'`#`') comment.
- *M-x S-set-style*
  Set the formatting style in this buffer to be one of the predefined styles: `GNU`, `BSD`, `K&R` and `C++`. The `DEFAULT` style uses the default values for the indenting variables (unless they have been modified in your '`.emacs`' file.) This command causes all of the formatting variables to be buffer-local.

4.6 Commands for motion, completion and more

A number of commands are provided to move across function definitions in the edit buffer:

- *ESC C-e* (`S-beginning-of-function`)
  Moves point to the beginning of the function containing point.
- *ESC C-a* (`S-end-of-function`)
  Moves point to the end of the function containing point.
- *ESC C-h* (`S-mark-function`)
  Places point at the beginning of the S function containing point, and mark at the end.

Don't forget the usual Emacs commands for moving over balanced expressions and parentheses: See section "Lists and Sexps" in *The GNU Emacs Reference Manual*.

Completion is provided in the edit buffer in a similar fashion to the process buffer: *M-TAB* completes file names and *M-?* lists file completions. Since ⟨TAB⟩ is used for indentation in the edit buffer, object completion is now performed with *C-c TAB*. Note however that completion is only provided over globally known S objects (such as system functions) — it will *not* work for arguments to functions or other variables local to the function you are editing.

Finally, two commands are provided for returning to the S process buffer:

- *C-c C-z* (`S-switch-to-end-of-S`)
  Returns you to the S process buffer, placing point at the end of the buffer.
- *C-c C-y* (`S-switch-to-S`)
  Also returns to to the S process buffer, but leaves point where it is.

In addition some commands available in the process buffer are also available in the edit buffer. You can still read help files with *C-c C-v*, edit another function with *C-c C-d* and of course *C-c C-l* can be used to load a source file into S. See Section 3.8 [Other], page 14 for more details on these commands.

4.7 Maintaining S source files

Every edit buffer in S-mode is associated with a *dump file* on disk. Dump files are created whenever you type *C-c C-d* (`S-dump-object-into-edit-buffer`), and may either be deleted after use, or kept as a backup file or as a means of keeping several versions of an S function.

S-delete-dump-files User Option

If non-`nil`, dump files created with C-c C-d are deleted immediately after they are created by the S-process.

Since immediately after S dumps an object's definition to a disk file the source code on disk corresponds exactly to S's idea of the object's definition, the disk file isn't really needed; deleting it now has the advantage that if you *don't* modify the file (say, because you just wanted to look at the definition of one of the standard S functions) the source dump file won't be left around when you kill the buffer. Note that this variable only applies to files generated with S's `dump` function; it doesn't apply to source files which already exist. The default value is `t`.

S-keep-dump-files User Option

Option controlling what to do with the dump file after an object has been successfully loaded into S. Valid values are `nil` (always delete), `ask` (always ask whether to delete), `check` (delete files generated with *C-c C-d* in this Emacs session, otherwise ask — this is the default) and `t` (never delete). This variable is buffer-local.

After an object has been successfully (i.e. without error) been loaded back into S with *C-c C-l*, the disk file again corresponds exactly (well, almost — see below) to S's record of the object's definition, and so some people prefer to delete the disk file rather than unnecessarily use up space. This option allows you to do just that.

If the value of `S-keep-dump-files` is `t`, dump files are never deleted after they are loaded. Thus you can maintain a complete text record of the functions you have edited within S-mode. Backup files kept as usual, and so by using the Emacs numbered backup facility — see section "Single or Numbered Backups" in *The Gnu Emacs Reference Manual*, you can keep a historic record of function definitions. Another possibility is to maintain the files with a version-control system such as RCS See section "Version Control" in *The Gnu Emacs Reference Manual*. As long as a dump file exists in the appropriate place for a particular object, editing that object with *C-c C-d* finds that file for editing (unless a prefix argument is given) — the S process is not consulted. Thus you can keep comments *outside* the function definition as a means of documentation that does not clutter the S object itself. Another useful feature is that you may format the code in any fashion you please without S re-indenting the code every time you edit it. These features are particularly useful for project-based work.

If the value of `S-keep-dump-files` is nil, the dump file is always silently deleted after a successful load with *C-c C-l*. While this is useful for files that were created with *C-c C-d* it also applies to any other file you load (say, a source file of function definitions), and so can be dangerous to use unless you are careful. Note that since `S-keep-dump-files` is buffer-local, you can make sure particular files are not deleted by setting it to `t` in the Local Variables section of the file See section "Local Variables in Files" in *The Gnu Emacs Reference Manual*.

A safer option is to set `S-keep-dump-files` to `ask`; this means that S-mode will always ask for confirmation before deleting the file. Since this can get annoying if you always want to delete dump files created with `C-c C-d`, but not any other files, setting `S-keep-dump-files` to `check` (the default value) will silently delete dump files created with *C-c C-d* in thie current Emacs session, but query for any other file. Note that in any case you will only be asked for confirmation once per file, and your answer is remembered for the rest of the Emacs session.

Note that in all cases, if an error (such as a syntax error) is detected while loading the file with *C-c C-l*, the dump file is *never* deleted. This is so that you can edit the file in a new Emacs session if you happen to quit Emacs before correcting the error.

Dump buffers are always autosaved, regardless of the value of `S-keep-dump-files`.

4.8 Names and locations of dump files

Every dump file should be given a unique file name, usually the dumped object name with some additions.

S-dump-filename-template User Option

Template for filenames of dumped objects. `%s` is replaced by the object name.

By default, dump file names are the user name, followed by '`.`' and the object and ending with '`.S`'. Thus if user `joe` dumps the object `myfun` the dump file will have name '`joe.myfun.S`'. The username part is included to avoid clashes when dumping into a publicly-writable directory, such as '`/tmp`'; you may wish to remove this part if you are dumping into a directory owned by you.

You may also specify the directory in which dump files are written:

S-source-directory User Option

Directory name (ending in a slash) where S dump files are to be written.

By default, dump files are always written to '`/tmp`', which is fine when `S-keep-dump-files` is `nil`. If you are keeping dump files, then you will probably want to keep them somewhere in your home directory, say '`~/S-source`'. This could be achieved by including the following line in your '`.emacs`' file:

```
(setq S-source-directory (expand-file-name "~/S-source/"))
```

If you would prefer to keep your dump files in separate directories depending on the value of some variable, S-mode provides a facility for this also. By setting `S-source-directory` to a lambda expression which evaluates to a directory name, you have a great deal of flexibility in selecting the directory for a particular source file to appear in. The lambda expression is evaluated with the process buffer as the current buffer and so you can use the variables local to that buffer to make your choice. For example, the following expression causes source files to be saved in the subdirectory 'Src' of the directory the S process was run in.

```
(setq S-source-directory
      (lambda ()
 (concat S-directory "Src/")))
```

(`S-directory` is a buffer-local variable in process buffers which records the directory the S process was run from.) This is useful if you keep your dump files and you often edit objects with the same name in different S processes. Alternatively, if you often change your S working directory during an S session, you may like to keep dump files in some subdirectory of the directory pointed to by the first element of the current search list. This way you can edit objects of the same name in different directories during the one S session:

```
(setq S-source-directory
   (lambda ()
       (file-name-as-directory
        (expand-file-name (concat
                           (car S-search-list)
                           "/.Src")))))
```

If the directory generated by the lambda function does not exist but can be created, you will be asked whether you wish to create the directory. If you choose not to, or the directory cannot be created, you will not be able to edit functions.

5 Reading help files in S-mode

S-mode provides an easy-to-use facility for reading S help files from within Emacs. From within the S process buffer or any S-mode edit buffer, typing *C-c C-v* (`S-display-help-on-object`) will prompt you for the name of an object for which you would like documentation. Completion is provided over all objects which have help files.

If the requested object has documentation, you will be popped into a buffer (named `*help(`*obj-name*`)*`) containing the help file. This buffer is placed in a special 'S Help' mode which disables the usual editing commands but which provides a number of keys for paging through the help file:

Help commands:

- *?* (`S-describe-help-mode`)
  Pops up a help buffer with a list of the commands available in S help mode.
- *h* (`S-display-help-on-object`)
  Pop up a help buffer for a different object

Paging commands:

- *b* or *DEL* (`scroll-down`)
  Move one page backwards through the help file.
- *SPC* (`scroll-up`)
  Move one page forwards through the help file.
- `>` (`beginning-of-buffer`) and `<` (`end-of-buffer`)
  Move to the beginning and end of the help file, respectively.

Section-based motion commands:

- *n* (`S-skip-to-next-section`) and *p* (`S-skip-to-previous-section`)
  Move to the next and previous section header in the help file, respectively. A section header consists of a number of capitalised words, followed by a colon.

In addition, the *s* key followed by one of the following letters will jump to a particular section in the help file:

'a' ARGUMENTS:

'b' BACKGROUND:

'B' BUGS:

'd' DETAILS:

'D' DESCRIPTION:

'e' EXAMPLES:

'n' NOTE:

'o' OPTIONAL ARGUMENTS:

'r' REQUIRED ARGUMENTS:

'R' REFERENCES:

's' SIDE EFFECTS:

`s` SEE ALSO:

`u` USAGE:

`v` VALUE:

`<` Jumps to beginning of file

`>` Jumps to end of file

`?` Pops up a help buffer with a list of the defined section motion keys.

Miscellaneous:

- *r* (`S-eval-region`)
  Send the contents of the current region to the S process. Useful for running examples in help files.
- */* (`isearch-forward`)
  Same as *C-s*.

Quit commands:

- *q* (`S-switch-to-end-of-S`)
  Returns to the S process buffer in another window, leaving the help window visible.
- *k* (`kill-buffer`)
  Kills the help buffer.
- *x* (`S-kill-buffer-and-go`)
  Return to the S process, killing this help buffer.

In addition, all of the S-mode commands available in the edit buffers are also available in S help mode (See Section 4.1 [Edit buffer], page 15). Of course, the usual (non-editing) Emacs commands are available, and for convenience the digits and - act as prefix arguments.

If a help buffer already exists for an object for which help is requested, that buffer is popped to immediately; the S process is not consulted at all. If the contents of the help file have changed, you either need to kill the help buffer first, or pass a prefix argument (with *C-u*) to `S-display-help-on-object`.

Help buffers are marked as temporary buffers in S-mode, and are deleted when `S-quit` or `S-cleanup` are called.

6 Manipulating saved transcript files

Inferior S mode records the transcript (the list of all commands executed, and their output) in the process buffer, which can be saved as a *transcript file*, which should normally have the suffix '`.St`'. The most obvious use for a transcript file is as a static record of the actions you have performed in a particular S session. Sometimes, however, you may wish to re-execute commands recorded in the transcript file by submitting them to a running S process. This is what Transcript Mode is for.

If you load file a with the suffix '`.St`' into Emacs, it is placed in S Transcript Mode. Transcript Mode is similar to Inferior S mode (see Chapter 3 [Entering commands], page 6): paragraphs are defined as a command and its output, and you can move though commands either with the paragraph commands or with *C-c C-p* and *C-c C-n*.

6.1 Resubmitting commands from the transcript file

Three commands are provided to re-submit command lines from the transcript file to a running S process. They are:

- *RET* (`S-transcript-send-command`)
  Send the current command line to the S process, and execute it.
- *C-c RET* (`S-transcript-copy-command`)
  Copy the current command to the S process, and switch to the S process buffer (ready to edit the copied command).
- *M-RET* (`S-transcript-send-command-and-move`)
  Send the current command to the S process, and move to the next command line. This command is useful for submitting a series of commands.

Note that these commands are similar to those on the same keys in Inferior S Mode. In all three cases, the commands should be executed when the cursor is on a command line in the transcript; the prompt is automatically removed before the command is submitted.

6.2 Cleaning transcript files

Yet another use for transcript files is to exatract the command lines for inclusion in an S source file or function. Transcript mode provides one command which does just this:

- *C-c C-w* (`S-transcript-clean-region`)
  Deletes all prompts and command output in the region, leaving only the commands themselves.

The remaining command lines may then be copied to a source file or edit buffer for inclusion in a function definition, or may be evaluated directly (see Section 4.4 [Evaluating code], page 16) using the code evaluation commands from S mode, also available in S Transcript Mode.

7 Other features of S-mode

S-mode has a few miscellaneous features, which didn't fit anywhere else.

7.1 Syntactic hightlighting of buffers

S-mode provides Font-Lock (see section "Using Multiple Typefaces" in *The Gnu Emacs Reference Manual*) patterns for Inferior S Mode, S Mode, and S Transcript Mode buffers.

To activate the highlighting, you need to turn on Font Lock mode in the appropriate buffers. This can be done on a per-buffer basis with *M-x font-lock-mode*, or may be done by adding `turn-on-font-lock` to `inferior-S-mode-hook`, `S-mode-hook` and `S-transcript-mode-hook` (see Section B.2 [Hooks], page 33). Your systems administrator may have done this for you in '`S-site.el`' (see Appendix B [Customisation], page 30).

The font-lock patterns are defined in three variables, which you may modify if desired:

S-inf-font-lock-keywords Variable

Font-lock patterns for Inferior S Mode. The default value highlights prompts, inputs, assignments, output messages, vector and matrix labels, and literals such as '`NA`' and `TRUE`.

S-mode-font-lock-keywords Variable

Font-lock patterns for S Mode. The default value highlights function names, literals, assignments, source functions and reserved words.

S-trans-font-lock-keywords Variable

Font-lock patterns for S Transcript Mode. The default value highlights the same stuff as in Inferior S Mode.

7.2 Using graphics with S-mode

One of the main features of the `S` package is its ability to generate high-resolution graphics plots, and S-mode provides a number of features for dealing with such plots.

7.2.1 Using S-mode with the `printer()` driver

This is the simplest (and least desirable) method of using graphics within S-mode. S's `printer()` device driver produces crude character based plots which can be contained within the S process buffer itself. To start using character graphics, issue the S command

```
printer(width=79)
```

(the `width=79` argument prevents Emacs line-wrapping at column 80 on an 80-column terminal. Use a different value for a terminal with a different number of columns.) Plotting commands do not generate graphics immediately, but are stored until the `show()` command is issued, which displays the current figure.

7.2.2 Using S-mode with windowing devices

Of course, the ideal way to use graphics with S-mode is to use a windowing system. Under X windows, this requires that the DISPLAY environment variable be appropriately set, which may not always be the case within your Emacs process. S-mode provides a facility for setting the value of DISPLAY before the S process is started if the variable `S-ask-about-display` is non-`nil`. See Appendix B [Customisation], page 30 for details of this variable, and see Chapter 2 [Starting Up], page 4 for information on how to set the value of DISPLAY when beginning an S session.

8 Known bugs in S-mode

- Commands like `S-display-help-on-object` and list completion cannot be used while the user is entering a multi-line command. The only real fix in this situation is to use another S process.
- The `S-eval-` commands can leave point in the S process buffer in the wrong place when point is at the same position as the last process output. This proves difficult to fix, in general, as we need to consider all *windows* with `window-point` at the right place.
- It's possible to clear the modification flag (say, by saving the buffer) with the edit buffer not having been loaded into S.
- Backup files can sometimes be left behind, even when `S-keep-dump-files` is `nil`.
- Passing an incomplete S expression to `S-execute` causes S-mode to hang.
- The function-based commands don't always work as expected on functions whose body is not a parenthesised or compound expression, and don't even recognise anonymous functions (i.e. functions not assigned to any variable).
- Multi-line commands could be handled better by the command history mechanism.

You can send a Bug report by typing *M-x S-submit-bug-report*, or by sending E-mail to `D.M.Smith@lancaster.ac.uk`. Comments, suggestions, words of praise and large cash donations are also more than welcome.

Appendix A Installing S-mode on your system

The following section details those steps necessary to get S-mode running on your system.

First of all, you need to create a directory (say, '~/elisp') to place the Emacs-Lisp files. Copy 'S.el', 'S-tek.el', 'comint.el', 'comint-isearch.el' and 'comint-extra.el' to that directory, and add the lines

```
(setq load-path (cons (expand-file-name "~/elisp") load-path))
(autoload 'S "S" "Run an inferior S process" t)
(autoload 's-mode "S" "Mode for editing S source" t)
```

to your '.emacs' file.

This will be enough to get S-mode running on most systems — see Chapter 2 [Starting Up], page 4 for details on starting S-mode. If it does not work, see Section A.1 [System dependent], page 28 for other variables you may need to change. See Appendix B [Customisation], page 30 for other variables you may wish to set in your '.emacs' file, but it is suggested you defer this section until you are more familiar with S-mode.

It is recommended that the `.el` files all be byte-compiled with *M-x byte-compile-file* for efficiency.

A.1 Other variables you may need to change

If you run the S program (from the shell) with a command other than 'Splus' you will need to set the variable `inferior-S-program` to the name of the appropriate program by including a line such as

```
(setq inferior-S-program "S+")
```

in your '.emacs' file (substituting 'S+' for the name of your S program.)

If you need to call this program with any arguments, the variable you need to set is dependent on the value of `inferior-S-program`; for example if it is `"Splus"`, set the variable `inferior-Splus-args` to a string of arguments to the `Splus` program. If `inferior-S-program` has some other value, substitute the `Splus` part of `inferior-Splus-args` with the appropriate program name. There aren't many instances where you need to call S with arguments, however: in particular do not call the S program with the '-e' command-line editor argument since S-mode provides this feature for you.

If you are running an older version of S, you may need to set the variable `S-version-running` to reflect this fact. The default is `"3.0"` which indicates the August '91 revision; any other value indicates an older version. This variable is effective only when S-mode is *loaded*; setting it during an S session has no effect.

If you are running Splus (the enhanced version of S from Statsci) you may also need to set the variable `S-plus` to `t`. If your value of `inferior-S-program` is `"S+"` or `Splus` this will not be necessary, however; `S-plus` defaults to `t` in this case.

Finally, if you use a non-standard prompt within S, you will need to set the variable `inferior-S-prompt` to a regular expression which will match both the primary prompt (`"> "` by default) and the continuing prompt (default of `"+ "`.) The default value of this variable matches S's default prompts. For example, if you use (`"$ "`) as your primary prompt (you have `options(prompt="$ ")` in your `.First` function), add the following line to your '.emacs':

```
(setq inferior-S-prompt "^\\(\\+\\|[^\\$]*\\$\\) *")
```

You will also need to set the variable `inferior-S-primary-prompt` to a regular expression which matches the primary prompt only. Do not anchor the regexp to the beginning of the line with '^'. Once again, the default value matches S's default prompt; in the example above the appropriate value would be `"[^\\$]*\\$ *"`.

Once these variables are set appropriately, S-mode should work on any system.

Appendix B Customizing S-mode

S-mode can be easily customised to your taste simply by including the appropriate lines in your '.emacs' file. There are numerous variables which affect the behaviour of S-mode in certain situations which can be modified to your liking. Keybindings may be set or changed to your preferences, and for per-buffer customisations hooks are also available.

B.1 Variables for customisation

S-mode is easily customisable by means of setting variables in your '.emacs' file. In most cases, you can change defaults by including lines of the form

(setq *variable-name value*)

in your '.emacs'.

In what follows, variable names will be listed along with their descriptions and default values. Just substitute the variable name and the new value into the template above.

B.1.1 Variables for starting S

S-ask-for-S-directory User Option

Default: `t`

If this variable has a non-`nil` value, then every time S-mode is run with *M-x S* you will be prompted for a directory to use as the working directory for your S session; this directory should have a '.Data' subdirectory. If the value of `S-ask-for-S-directory` is `nil`, the value of `S-directory` is used as the working directory.

S-directory User Option

Default: Your home directory

The working directory for your S session if `S-ask-for-S-directory` is `nil`, and the default when prompting for a directory if it is not. For example, you may wish to set this to the value of the current buffer's working directory before starting S by adding the following line to your '.emacs' file (See Section B.2 [Hooks], page 33)

```
(setq S-pre-run-hook
   '((lambda () (setq S-directory default-directory))))
```

S-ask-about-display User Option

Default: `nil`

If this variable has a non-`nil` value, then every time S-mode is run with *M-x S* you will be asked for a value for the `DISPLAY` environment variable to be used in the current S session. If this variable is not set correctly, S will not be able to create any windows under the X windowing environment. Completion is provided over the `X-displays-list` variable; the default is the current value of `DISPLAY`. This feature is useful is you often run S on a different display than that of the machine you are running S from. If `S-ask-about-display` is `nil`, the current value of `DISPLAY` is used.

X-displays-list User Option

Default: `'(":0.0")`

List of possible values for the `DISPLAY` environment variable, provided for completion when prompting for such a value.

B.1.2 Variables for dump files

S-insert-function-templates User Option

Default: `t`

If this variable has a non-`nil` value, then dumping a non-existent object will result in the edit buffer containing a skeleton function definition, ready for editing.

S-source-directory User Option

Default: `"/tmp/"`

Directory name (ending in '/') in which dump files are placed. This should always be a writable directory.

S-source-directory-generator User Option

Default: `nil`

Alternative, dynamic method of specifying the directory for dump files.

S-dump-filename-template User Option

Default: *user_name*`.`*object_name*`.S`

Naming system to use for dumped object files. See Section 4.8 [Source Directories], page 20 for details of this and the previous two variables.

S-keep-dump-files User Option

Default: `nil`

Boolean flag signifying whether to keep dump files or to delete them after each use. See Section 4.7 [Source Files], page 19 for more details.

B.1.3 Variables controlling indentation

S-tab-always-indent User Option

Default: `t`

If non-`nil`, then *TAB* in the edit buffer always indents the current line, regardless of the position of point in the line. Otherwise, indentation is only performed if point is in the lines indentation, and a tab character is inserted is point is after the first nonblank character.

S-auto-newline User Option

Default: `nil`

Non-`nil` means automatically newline before and after braces inserted in S code.

The following variables control amounts of indentation. These variables automatically become buffer-local in any S-mode buffer, and so setting any of these variables has effect in the current buffer only.

S-indent-level User Option
Default: 2
Extra indentation of S statement sub-block with respect to enclosing braces.

S-brace-imaginary-offset User Option
Default: 0
Extra indentation (over sub-block indentation) for block following an open brace which follows on the same line as a statement.

S-brace-offset User Option
Default: 0
Extra indentation for braces, compared with other text in same context.

S-continued-statement-offset User Option
Default: 0
Extra indent for lines not starting new statements.

S-continued-brace-offset User Option
Default: 0
Extra indent for substatements that start with open-braces. This is in addition to `S-continued-statement-offset`.

S-arg-function-offset User Option
Default: 2
Extra indent for arguments of function `foo` when it is called as the value of an argument to another function in `arg=foo(...)` form. If not number, the statements are indented at open-parenthesis following `foo`.

S-expression-offset User Option
Default: 4
Extra indent for internal substatements of the call to `expression()` specified in

```
obj <- expression(...)
```

form. If not a number, the statements are indented at open-parenthesis following '`expression`'.

S-else-offset User Option
Default: 2
Extra indentation of the `else` clause with respect to the corresponding `if`.

In addition, a number of default styles are defined for you (in `S-style-alist`):

S-default-style User Option
Default: `DEFAULT`
The default formatting style to use in edit buffers. The DEFAULT style uses the values of the above indentation variables at load-time, so that changing these variables in your '`.emacs`' file will affect your buffer defaults. See Section 4.1 [Edit buffer], page 15 for more details.

B.1.4 Variables controlling interaction with the S process

comint-input-ring-size User Option

Default: 50

Number of commands to store in the command history.

S-execute-in-process-buffer User Option

Default: `nil`

If this is `nil`, then the `S-execute-` commands (see Section 3.8 [Other], page 14) output to a temporary buffer. Otherwise, the output goes to the S process.

S-eval-visibly-p User Option

Default: `nil`

If non-`nil`, then the `S-eval-` commands (see Section 4.1 [Edit buffer], page 15) echo the S commands in the process buffer by default. In any case, passing a prefix argument to the eval command reverses the meaning of this variable.

B.2 Customizing S-mode with hooks

S-mode provides five hooks, as follows:

S-mode-hook Hook

Called every time `S-mode` is run, i.e. every time an edit buffer is generated.

S-pre-run-hook Hook

Called before the S process is started with *M-x S*.

S-post-run-hook Hook

Called just after the S process is started with *M-x S*. This can be used to evaluate S code at the start of a session, with `S-eval-visibly`, say.

S-mode-load-hook Hook

Called just after the file '`S.el`' is loaded. Useful for setting up your keybindings, etc.

inferior-S-mode-hook Hook

Called just after the S process starts up, when the S process buffer is initialised.

S-help-mode-hook Hook

Called every time an S help buffer is generated.

B.3 Changing the default S-mode keybindings

S-mode provides a separate keymap variable for the S process buffer, for edit buffers and for help buffers.

inferior-S-mode-map Keymap

Keymap used in the S process buffer. The bindings from `comint-mode-map` are automatically inherited.

S-mode-map Keymap

Keymap used within edit buffers.

S-help-mode-map Keymap

Keymap used within help buffers. In addition, `S-help-sec-map` is the keymap for the '`s`' prefix key. Keys defined in `S-help-sec-keys-alist` are automatically inserted into this keymap when S-mode is loaded.

Concept Index

M

N

O

P

Q

R

S

T

U

V

W

X

Variable and command index

X

Table of Contents