Editing and running Standard ML under GNU Emacs

SML mode, Version 3.3
April 1997

Author: Matthew J. Morley

Copyright © (Anon)

GNU General Public License as published by the Free Software Foundation; either version 2, or (at your option) any later version.

SML mode is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with GNU Emacs; see the file COPYING. If not, write to the Free Software Foundation, 675 Mass Ave, Cambridge, MA 02139, USA.

1 Introduction

SML mode is a major mode for Emacs for editing Standard ML. It has some novel bugs, and some nice features:

- Automatic indentation of sml code—a number of variables to customise the indentation.
- Easy insertion for commonly used templates like let, local, signature, and structure declarations, with minibuffer prompting for types and expressions.
- Magic pipe insertion: `|` automatically determines if it is used in a case or fun construct, and indents the next line as appropriate, inserting `=>` or the name of the function.
- Inferior shell for running ML. There's no need to leave Emacs, just keep on editing while the compiler runs in another window.
- Automatic "use file" in the inferior shell—you can send files, buffers, or regions of code to the ML subprocess.
- Menus, and syntax and keyword highlighting supported for Emacs 19 and derivatives.
- Parsing errors from the inferior shell, and repositioning the source—much like the next-error function used in c-mode.
- SML mode can be easily configured to work with a number of Standard ML compilers, and other SML based tools.

1.1 The SML mode distribution

The distribution contains several Emacs Lisp files—this is for ease of maintenance, you can concatenate them if you're careful:

'`sml-mode.el`'
: Main file, and should work in any Emacs editor or version post 18.58—it only knows, or thinks it knows, about SML syntax and indentation.

'`sml-menus.el`'
: Menus to access user settable features of the mode, and for those who prefer menus over keys under Emacs 19 and derivatives.

'`sml-{hilite,font}.el`'
: Syntax highlighting functions to display keywords in a bold font, comments in italics, etc., using one of Emacs' two popular syntax colouring packages.

'`sml-proc.el`'
: Process interaction requires the '`comint`' package (normally distributed with Emacs 19 and derivatives).

'`sml-{poly-ml,mosml}.el`'
: Auxiliary library support for Poly/ML and Moscow ML compilers.

There is also the Texinfo generated `info` file:

'`sml-mode.{dvi,info}`'
: This file—rudimentary SML mode documentation, and

'`sml-site.el`'
: Configuration file for system-wide installation. Read and edit this file if you are installing SML mode for general use.

1.2 Getting started

With luck your system administrator will have installed SML mode somewhere convenient, so all you have to do is put the line

```
(require 'sml-site)
```

in your '`.emacs`' configuration file and all will be well—you can skip the rest of this getting started section. Otherwise you will need to tell Emacs where to find all the SML mode '`.el`' files, and when to use them. The where is addressed by locating the Lisp code on your Emacs Lisp load path—you may have to create a directory for this, say '`/home/mjm/elisp`', and then insert the following lines in your '`/home/mjm/.emacs`' file[1]:

```
(setq load-path (cons "/home/mjm/elisp" load-path))
(autoload 'sml-mode "sml-mode" "Major mode for editing SML." t)
```

The first line adjusts Emacs' internal search path so it can locate the Lisp source you have copied to that directory; the second line tells Emacs to load the code automatically when it is needed. You can then switch any Emacs buffer into SML mode by entering the command

```
M-x sml-mode
```

It is usually more convenient to have Emacs automatically place the buffer in SML mode whenever you visit a file containing ML programs. The simplest way of achieving this is to put something like

```
(setq auto-mode-alist
      (append '(("\\.sml$" . sml-mode)
                ("\\.sig$" . sml-mode)
                ("\\.ML$"  . sml-mode)) auto-mode-alist))
```

also in your '`.emacs`' file. Subsequently (after a restart), any files with these extensions will be placed in SML mode buffers when you visit them.

You may want to pre-compile the '`sml-*.el`' files (*`M-x byte-compile-file`*) for greater speed—byte compiled code loads and runs somewhat faster.

1.3 Help!

You're reading it. Apart from the on-line info tree (*`C-h i`* is the Emacs key to enter the `info` system—you should follow the brief tutorial if this is unfamiliar), there are further details on specific commands in their documentation strings. Only the most useful SML mode commands are documented in the info tree: to find out more use Emacs' help facilities.

Briefly, to get help on a specific function use *`C-h f`* and enter the command name. All (almost all, then) SML mode commands begin with `sml-`, so if you type this and press ⟨TAB⟩ (for completion) you will get a list of all commands. Another way is to use *`C-h a`* and enter the string `sml`. This is command apropos; it will list all commands with that sub-string in their names, and any key binding they may have in the current buffer. Command apropos gives a one-line synopsis of what each command does.

Some commands are also variables—such things are allowed in Lisp, if not in ML! See [Command Index], page 17, for a list of (info) documented functions. See [Variable Index], page 17, for a list of user settable variables to control the behaviour of SML mode.

[1] cf. commentary in the site initialisation file '`sml-site.el`'.

Before accessing this information on-line from within Emacs you may have to set the variable `sml-mode-info`. Put in your '`.emacs`' file something like:

```
(setq sml-mode-info "/home/mjm/info/sml-mode.info")
```

When different from the default this variable should be a string giving the absolute name of the '`.info`' file. Then *`C-c C-i`* in SML mode (i.e., the command *`M-x sml-mode-info`*) will bring up the manual. This help is also accessible from the menu. (Resetting this variable will not be necessary if your site administrator has been kind enough to install SML mode and its attendant documentation in the Emacs hierarchy.)

2 Editing with SML Mode

Now SML mode provides just a few additional editing commands. Most of the work (see [Credit & Blame], page 17) has gone into implementing the indentation algorithm which, if you think about it, has to be complicated for a language like ML. See Section 2.4 [Indentation Defaults], page 5, for details on how to control some of the behaviour of the indentation algorithm. Principal goodies are the 'electric pipe' feature, and the ability to insert common SML forms (macros or templates).

2.1 On entering SML mode

sml-mode Command

This switches a buffer into SML mode. This is a *major mode* in Emacs. To get out of SML mode the buffer's major mode must be set to something else, like `text-mode`. See Section 1.2 [Getting Started], page 2, for details on how to set this up automatically when visiting an SML file.

Emacs is all hooks of course. A hook is a variable: if the variable is non-nil it binds a list of Emacs Lisp functions to be run in some order (usually left to right). You can customise SML mode with these hooks:

sml-mode-hook Hook

Default: `nil`

This is run every time a new SML mode buffer is created (or if you type *`M-x sml-mode`*). This is one place to put your preferred key bindings. See Chapter 4 [Configuration], page 12, for some examples.

sml-load-hook Hook

Default: `'sml-mode-version`

Another, maybe better, place for key bindings. This hook is only run when SML mode is loaded into Emacs. See Chapter 4 [Configuration], page 12.

sml-mode-version Command

Prints the current version of SML mode in the mini-buffer, in case you need to know. I've put it on `sml-load-hook` so you can easily tell which version of SML mode you are running.

2.2 Automatic indentation

ML is a complicated language to parse, let alone compile. The indentation algorithm is a little wooden (for some tastes), and the best advice is not to fight it! There are several variables that can be adjusted to control the indentation algorithm (see Section 2.4 [Customising SML Mode], page 5, below).

sml-indent-line Command

Key: ⟨TAB⟩

This command indents the current line. If you set the indentation of the previous line by hand, `sml-indent-line` will indent relative to this setting.

sml-indent-region Command

Key: *C-M-*

Indent the current region. Be patient if the region is large (like the whole buffer).

sml-back-to-outer-indent Command

Key: *M-*⟨TAB⟩

Unindents the line to the next outer level of indentation.

Further indentation commands that Emacs provides (generically, for all modes) that you may like to recall:

– *M-x newline-and-indent*

  On ⟨LFD⟩ by default. Insert a newline, then indent according to the major mode. See section "Indentation for Programs" in *The Emacs Editor Manual*, for details.

– *M-x indent-rigidly*

  On *C-x* ⟨TAB⟩ by default. Moves all lines in the region right by its argument (left, for negative arguments). See section "Indentation" in *The Emacs Editor Manual*.

– *M-x indent-for-comment*

  On *M-;* by default. Indent this line's comment to comment column, or insert an empty comment. See section "Comment Commands" in *The Emacs Editor Manual*.

– *M-x indent-new-comment-line*

  On *M-*⟨LFD⟩ by default. Break line at point and indent, continuing comment if within one. See section "Multi-Line Comments" in *The Emacs Editor Manual*.

As with other language modes, *M-;* gives you a comment at the end of the current line. The column where the comment starts is determined by the variable `comment-column`—default is 40, but it can be changed with `set-comment-column` (on *C-x ;* by default).

2.3 Electric features

Electric keys are generally pretty irritating, so those provided by SML mode are fairly muted. The only truly electric key is *;*, and this has to be enabled to take effect.

sml-electric-pipe Command

Key: *M-|*

When the point is in a 'case' statement this opens a new line, indents and inserts `| =>` leaving point just before the double arrow; if the enclosing construct is a 'fun' declaration, the newline is indented and the function name copied at the appropriate column. Generally, try it whenever a `|` is wanted—you'll like it!

sml-electric-semi Command

Key: *;*

Just inserts a semi-colon, usually. The behaviour of this command is governed by the variable `sml-electric-semi-mode`.

sml-electric-semi-mode Command, Variable

Default: `nil`

If this variable is `nil`, `sml-electric-semi` just inserts a semi-colon, otherwise it inserts a semi-colon and a newline, and indents the newline for SML. The command toggles the value of the variable; if you give the command a prefix argument (i.e., *C-u M-x sml-electric-semi-mode*) this always disables the electric effect of *;*.

sml-insert-form Command

Key: *C-c* ⟨RET⟩

Interactive short-cut to insert common ML forms (a.k.a. macros, or templates). Recognised forms are 'let', 'local', 'case', 'abstype', 'datatype', 'signature', 'structure', and 'functor'. Except for 'let' and 'local', these will prompt for appropriate parameters like functor name and signature, etc.. This command prompts in the mini-buffer, with completion.

By default *C-c* ⟨RET⟩ will insert at point, with the indentation of the current column; if you give a prefix argument (i.e., *C-u C-c* ⟨RET⟩) the command will insert a newline first, indent, and then insert the template.

`sml-insert-form` is also extensible: see Chapter 4 [Configuration], page 12 for further details.

2.4 Indentation defaults

Several variables try to control the indentation algorithm and other features of SML mode. For these user settable variables there is generally a function of the same name that does the job—look for them in the menu under *Format/Mode Variables*.

sml-indent-level Command, Variable

Default: `4`

This variable controls the block indentation level. The command prompts for a numeric value unless a numeric prefix is provided instead. For example *M-2 M-x sml-indent-level* will set the variable to 2 without prompting.

sml-pipe-indent Command, Variable

Default: `-2`

This variable adjusts the indentation level for lines that begin with a `|` (after any white space). The extra offset is usually negative. The command prompts for a numeric value unless a numeric prefix is provided instead.

sml-paren-lookback Variable

Default: `1000`

The number of characters the indentation algorithm searches for an opening parenthesis. 1000 characters is about 30-40 lines; larger values mean slower indentation. If the value of the variable is `nil` this means the indentation algorithm won't look back at all.

If the default values are not acceptable you can set these variables permanently in your '`.emacs`' file. See Chapter 4 [Configuration], page 12, for details and examples. Three further variables control the behaviour of indentation.

sml-case-indent Command, Variable

Default: `nil`

How to indent 'case' expressions:

If `t`:
```
case expr
  of exp1 => ...
   | exp2 => ...
```

If `nil`:
```
case expr of
    exp1 => ...
  | exp2 => ...
```

The first seems to be the standard in SML/NJ. The second is the (nicer?) default.

sml-nested-if-indent Command, Variable

Default: `nil`

Nested 'if-then-else' expressions have the following indentation depending on the value.

If `t`:
```
if exp1 then exp2
else if exp3 then exp4
else if exp5 then exp6
     else exp7
```

If `nil`:
```
if exp1 then exp2
else if exp3 then exp4
     else if exp5 then exp6
          else exp7
```

sml-type-of-indent Command, Variable

Default: `t`

Determines how to indent 'let', 'struct', etc..

If `t`:
```
fun foo bar = let
                  val p = 4
              in
                  bar + p
              end
```

If `nil`:
```
fun foo bar = let
    val p = 4
in
    bar + p
end
```

`sml-type-of-indent` will not have any effect if the starting keyword is the first word on the line.

3 Running ML under Emacs

The most useful feature of SML mode is that it provides a convenient interface to the compiler. How serious users of ML put up with a teletype interface to the compiler is beyond me... but perhaps there are other interfaces to compilers that require one to part with serious money. Such remarks can quickly become dated—in this case, let's hope so!

Anyway, SML mode provides an interaction mode, `inferior-sml-mode`, where the compiler runs in a separate buffer in a window or frame of its own. You can use this buffer just like a terminal, but it's usually more convenient to mark some text in the SML mode buffer and have Emacs communicate with the sub-process. The features discussed below are syntax-independent, so they should work with a wide range of ML-like tools and compilers. See Section 3.4 [Process Defaults], page 11, for some hints.

`inferior-sml-mode` is a specialisation of the '`comint`' package that comes with GNU Emacs and GNU XEmacs.

3.1 Starting the compiler

Start your favourite ML compiler with the command

M-x sml

This creates a process interaction buffer that inherits some key bindings from SML mode and from '`comint`' (see section "Shell Mode" in *The Emacs Editor Manual*). Starting the ML compiler adds some functions to SML mode buffers so that program text can be communicated between editor and compiler (see Section 3.2 [ML Interaction], page 9).

The name of the ML compiler is the first thing you should know how to specify:

sml-program-name Variable

Default: `"sml"`

The program to run as ML. You might need to specify the full path name of the program.

sml-default-arg Variable

Default: `""`

Useful for Poly/ML users who may supply a database file, or others who have wrappers for setting various options around the command to run the compiler. Moscow ML people might set this to `"-P full"`, etc..

The variable `sml-program-name` is a string holding the name of the program *as you would type it at the shell*. You can always choose a program different to the default by invoking

C-u M-x sml

With the prefix argument Emacs will prompt for the command name and any command line arguments to pass to the compiler. Thereafter Emacs will use this new name as the default, but for a permanent change you should set this in your '`.emacs`' with, e.g.:

```
(setq sml-program-name "nj-sml")
```

You probably shouldn't set this in `sml-mode-hook` because that will interfere if you occasionally run a different compiler (e.g., `poly` or `hol90`).

sml Command

Launches ML as an inferior process in another buffer; if an ML process already exists, just switch to the process buffer. A prefix argument allows you to edit the command line to specify the program, and any command line options.

inferior-sml-mode-hook Hook

Default: `nil`

`M-x sml` runs `comint-mode-hook` and `inferior-sml-mode-hook` hooks in that order, but *after* the compiler is started. Use `inferior-sml-mode-hook` to set any `comint` buffer-local configurations for SML mode you like.

inferior-sml-load-hook Hook

Default: `nil`

This hook is analogous to `sml-load-hook` and is run just after the code for `inferior-sml-mode` is loaded into Emacs. Use this to set process defaults, and preferred key bindings for the interaction buffer.

switch-to-sml Command

Key: *`C-c C-s`*

Switch from the SML buffer to the interaction buffer. By default point will be placed at the end of the process buffer, but a prefix argument will leave point wherever it was before. If you try *`C-c C-s`* before an ML process has been started, you'll just get an error message to the effect that there's no current process buffer.

sml-dedicated-frame Variable

Default: `(if window-system t nil)`

If `t` this indicates to `switch-to-sml` and other functions that the interaction buffer where ML is running will be displayed on its own, dedicated frame; otherwise the interaction buffer will appear on the current frame, splitting the window if necessary. The default means SML mode will try and use a dedicated frame if you are running Emacs under X Windows (say), but not otherwise. The variable `sml-display-frame-alist` configures the dedicated frame's appearance (*`C-h v sml-display-frame-alist`* for details).

sml-cd Command

When started, the ML compiler's default working directory is the current buffer's default directory. This command allows the working directory to be changed, if the compiler can do this. The variable `sml-cd-command` specifies the compiler command to invoke (see Section 3.4 [Process Defaults], page 11).

3.2 Speaking to the compiler

Several commands are defined for sending program fragments to the running compiler. Each of the following commands takes a prefix argument that will switch the input focus to the process buffer afterwards (leaving point at the end of the buffer):

sml-load-file Command

Key: *C-c C-l*

Send a 'use file' command to the current ML process. The variable `sml-use-command` is used to define the correct template for the command to invoke (see Section 3.4 [Process Defaults], page 11). The default file is the file associated with the current buffer, or the last file loaded if you are in the interaction buffer.

sml-send-region Command

Key: *C-c C-r*

Send the current region of text in the SML buffer. `sml-send-region-and-go` is a similar command for you to bind in SML mode if you wish: it'll send the region and then switch-to-sml.

sml-drag-region Command

Key: *M-S-down-mouse-1*

It's sometimes irritating to do all that *C-@* and *C-c C-r* stuff to send regions to the ML process, so if you are running Emacs under X Windows (say) you can do the same job by holding down both the ⟨SHIFT⟩ and ⟨META⟩ keys, and dragging with mouse button one over the region. This will temporarily highlight the region as you move the mouse, like `mouse-drag-region` (i.e., *down-mouse-1*), and send the highlighted text straight into the jaws of the ML compiler.

If you only click the mouse button, instead of dragging, the region of text sent to the compiler is delimited by the current position of point and the place where you click the mouse. In neither case will the command set the region.

sml-send-buffer Command

Key: *C-c C-b*

Send the contents of the current buffer to ML.

By and large, Emacs can nowadays quite happily send large chunks of text to its subprocesses ('`comint`' does input splitting). However, it is still probably safest[2] to send larger program fragments to ML via the temporary file mechanism. This, for `sml-send-region` and other SML mode commands that use it in some way, takes advantage of the ML compiler's ability to open a file and compile the contents by making a temporary file of the indicated text. Two variables of interest are:

sml-temp-threshold Variable

Default: `0`

Determines what constitutes a large program fragment. A value of 512, say, will declare half a kilobyte a suitable threshold and larger fragments will be sent via a temporary file. A value of 0 means *all* text is sent via a temporary file; the value `nil` inhibits the temporary file mechanism altogether.

---

[2] XEmacs 19.11 users are warned that changing the default `sml-temp-threshold` may well cause XEmacs to hang; they seem to have fixed the problem in 19.12 and above.

sml-temp-file Variable

Default: `(make-temp-name "/tmp/ml")`

A string that gives the name of the temporary file to use. This default ensures Emacs will invent a unique name for this purpose for use throughout the rest of the editing session. Only one temporary file is used.

Another reason, you might well say *the reason*, for using the temporary file mechanism is that error messages reported by the ML compiler (see Section 3.3 [Tracking Errors], page 10) are generally useless to SML mode unless a real file is associated with the input (an embedded *use file* will count as a real file). Of course, this all rather depends on the compiler producing sensible error messages, and on SML mode being able to parse them.

3.3 Finding errors

SML mode provides one customisable function for locating the source position of errors reported by the compiler. This should work whether you type `use "puzzle.sml";` into the interaction buffer, or use one of the mechanisms provided for sending programs directly to the compiler—see Section 3.2 [ML Interaction], page 9.

sml-next-error Command

Key: *`C-c``*

Jump to the source location of the next error reported by the compiler. If the function bound to `sml-error-parser` returns a range of character positions for the location of the error in the source file, `sml-next-error` will put the mark at the end of the range with point at the beginning; it may also highlight the region specified; it will also echo the one-line text of the error message if the error parser returns one.[3]

If you enter *`C-u C-c``* instead, the command (a.k.a. `sml-skip-errors`) skips past all the remaining error messages and removes any error overlay in the current buffer. Note that *`C-c``* also works in the ML interaction buffer (by default).

sml-error-overlay Variable, Command

Default: `t`

Legal default values for this buffer-local variable are `t` and `nil`. The variable attains a local value in each SML mode buffer when the default is `t`; in this case the local value is an overlay (or *extent* in XEmacs speak), and this means `sml-next-error` will highlight errors in the buffer when it can. If the default is `nil` it stays that way and `sml-next-error` will not highlight anything, ever.

The command *`M-x sml-error-overlay`* will set the overlay around the current region, or remove the overlay if a prefix argument is given (i.e., *`C-u M-x sml-error-overlay`* removes the overlay, but this functionality can be accessed from the menu to save typing).

[3] Does `sml-error-parser` return these nice things? The answer is complicated! See Section 4.5 [Advanced Topics], page 15, and the docstring *`C-h v sml-error-parser`*.

Note that SML mode will usually locate errors relative to the start of the last major program fragment sent to the compiler (via `sml-load-file`, etc.), but if you don't use the temporary file mechanism to communicate text to the ML process (see Section 3.4 [Process Defaults], page 11), errors will generally not be located at all.

3.4 Process defaults

The process interaction code is independent of the compiler used, deliberately, so SML mode will work with a variety of ML compilers and ML-based tools. There are therefore a number of variables that may need to be set correctly before SML mode can speak to the compiler. Things are by default set up for Standard ML of New Jersey, but switching to a new system is quite easy—very easy if you are using Poly/ML or Moscow ML as these are supported by libraries bundled with SML mode.

sml-use-command Variable

Default: `"use \"%s\""`

Use file command template. Emacs will replace the `%s` with a file name. Note that Emacs requires double quote characters inside strings to be quoted with a backslash.

sml-cd-command Variable

Default: `"OS.FileSys.chDir \"%s\""`

Compiler command to change the working directory. Not all ML systems support this feature (well, Edinburgh (core) ML didn't), but they should.

sml-prompt-regexp Variable

Default: `"^[\-=] *"`

Matches the ML compiler's prompt: '`comint`' uses this for various purposes.

To customise error reportage for different ML compilers you need to set two further variables before `sml-next-error` can be useful:

sml-error-regexp Variable

Default: `sml-smlnj-error-regexp`

This is the regular expression for matching the start of an error message. The default matches the Standard ML of New Jersey compiler's Error and Warning messages. If you don't want stop at Warnings try, for example:

```
"^[-= ]*.+:[0-9]+\\.[0-9]+.+Error:"
```

If you're using Edinburgh (core) ML try `"^Parse error:"`.

sml-error-parser Variable

Default: `'sml-smlnj-error-parser`

The function that actually parses the error message. Again, the default is for SML/NJ. If you need to change this you may have to do a little Emacs Lisp programming.

Note that bundled libraries supply an `sml-mosml-error-parser` and an `sml-poly-ml-error-parser`, and set all the attendant compiler variables. See Section 4.5 [Advanced Topics], page 15, for tips on how to program your own compiler extension to SML mode.

4 Configuration Summary

This (sort of pedagogic) section gives more information on how to configure SML mode: menus, key bindings, hooks and highlighting are discussed, along with a few other random topics. First, though, the auxiliary files 'sml-poly-ml.el' and 'sml-mosml.el' define defaults for these popular (?) ML compilers—Poly/ML and Moscow ML, respectively. One way to setup SML mode to use Moscow ML is to add to your '.emacs':

```
(defun my-mosml-setup () "Initialise inferior SML mode for Moscow ML."
  (load-library "sml-mosml.el")
  (setq sml-program-name "/home/mjm/mosml/bin/mosml"))
(add-hook 'inferior-sml-load-hook 'my-mosml-setup)
```

which creates a hook function `my-mosml-setup` and adds it to `inferior-sml-load-hook` so that the defaults for `sml-error-regexp` and its ilk (see Section 3.4 [Process Defaults], page 11) are correctly initialised; I have to set `sml-program-name` explicitly here because that directory isn't on my (Unix) PATH. The story is similar if you use Poly/ML. Note, by the way, that order matters here: the `load-library` call comes first because the default for `sml-program-name` in 'sml-mosml.el' is just `"mosml"`.

The auxiliary libraries bundled with SML mode define commands `sml-mosml` and `sml-poly-ml` (there's also an `sml-smlnj` for uniformity); these commands prompt for suitable values for `sml-program-name` and `sml-default-arg` before starting the compiler and setting the other process defaults. A prefix argument will give you the builtin defaults with no questions asked.

4.1 Hooks

One way to set SML mode variables (see Section 2.4 [Indentation Defaults], page 5), and other defaults, is through the `sml-mode-hook` in your '.emacs'. A simple example:

```
(defun my-sml-mode-hook () "Local defaults for SML mode"
  (setq sml-indent-level 2)        ; conserve on horizontal space
  (setq words-include-escape t)    ; \ loses word break status
  (setq indent-tabs-mode nil))     ; never ever indent with tabs
(add-hook 'sml-mode-hook 'my-sml-mode-hook)
```

The body of `my-sml-mode-hook` is a sequence of bindings. In this case it is not really necessary to set `sml-indent-level` in a hook because this variable is global (most SML mode variables are). With similar effect:

```
(setq sml-indent-level 2)
```

anywhere in your '.emacs' file (but probably on `sml-load-hook`). The variable `indent-tabs-mode` is automatically made local to the current buffer whenever it is set explicitly, so it *must* be set in a hook if you always want SML mode to behave like this. The same goes for the buffer-local `sml-error-overlay`; since this is globally `t` by default, set it globally `nil` if you never want errors highlighted:

```
(setq-default sml-error-overlay nil)
```

Again, on `sml-load-hook` would probably be the best place.

Another hook is `inferior-sml-mode-hook`. This can be used to control the behaviour of the interaction buffer through various variables meaningful to 'comint'-based packages:

```
(defun my-inf-sml-mode-hook () "Local defaults for inferior SML mode"
  (add-hook 'comint-output-filter-functions 'comint-truncate-buffer)
  (setq      comint-scroll-show-maximum-output t)
  (setq      comint-input-autoexpand nil))
(add-hook 'inferior-sml-mode-hook 'my-inf-sml-mode-hook)
```

Again, the body is a sequence of bindings. Unless you run several ML compilers simultaneously under one Emacs, this hook will normally only get run once. You might want to look up the documentation (*C-h v* and *C-h f*) for these buffer-local `comint` things.

4.2 Key bindings

Customisation (in Emacs) usually entails putting favourite commands on easily remembered keys. Two 'keymaps' are defined in SML mode: one is effective in program text buffers (`sml-mode-map`) and the other is effective in interaction buffers (`inferior-sml-mode-map`). The initial design ensures that (many of) the default key bindings from the former keymap will also be available in the latter (e.g., *C-c '*).

Type *C-h m* in an SML mode buffer to find the default key bindings (and similarly in an ML interaction buffer), and use the hooks provided to install your preferred key bindings. Given that the keymaps are global (variables):

```
(defun my-sml-load-hook () "Global defaults for SML mode"
  (define-key   sml-mode-map "\C-cd" 'sml-cd)
  (define-key   sml-mode-map "\C-co" 'sml-error-overlay))
(add-hook 'sml-load-hook 'my-sml-load-hook)
```

This has the effect of binding `sml-cd` to the key *C-c d*, and the command `sml-error-overlay` to the key *C-c o*. If you want the same behaviour from *C-c d* in the ML buffer:

```
(defun my-inf-sml-load-hook () "Global defaults for inferior SML mode"
  (define-key inferior-sml-mode-map "\C-cd" 'sml-cd)
  ;; NB. for SML/NJ '96
  (setq sml-cd-command "OS.FileSys.chDir \"%s\""))
(add-hook 'inferior-sml-load-hook 'my-inf-sml-load-hook)
```

There is nothing to stop you rebuilding the entire keymap for SML mode and the ML interaction buffer in your '`.emacs`' of course: SML mode won't define `sml-mode-map` or `inferior-sml-mode-map` if you have already done so.

4.3 Menus

Menus are useful for fiddling with mode defaults and finding out what keys commands are on if you are forgetful (not all commands are listed in the menu). For menus to appear in the menu bar under GNU Emacs or GNU XEmacs, the editor must be able to find one of two packages—i.e., one or both must be on your `load-path`. The first option is '`easymenu`' which is distributed with GNU Emacs. Easy!

The second option is '`auc-menu`' which was written by Per Abrahamsen and distributed with AUCTeX, but it is independently available from the IESD lisp archive[4] at Aalborg.

[4] '`ftp://sunsite.auc.dk/packages/auctex/`'

You'll also find 'auc-menu' is available from the LCD archive[5], the main repository for all Emacs Lisp. The advantage of 'auc-menu' is that it works with XEmacs too.

Notice that certain menu entries are not illuminated at first—these are generally functions that depend on there being an ML process running with which to communicate.

4.4 Syntax colouring

Highlighting is very handy for picking out keywords in the program text, spotting misspelled kewyords, and, if you have Emacs' 'ps-print' package installed (you usually do these days), obtaining pretty, even colourful code listings—quite properly for your colourful ML programs.

Various highlight (hilite, if you spell real bad!) packages are available for GNU Emacs 19, and GNU XEmacs. SML mode can use either 'hilit19' which only comes with Emacs, or 'font-lock' which is the package of choice with XEmacs. If you are not familiar with these highlight packages you'll have to check their sources for installation guidelines, etc..

Use `sml-load-hook` to tell Emacs which scheme you prefer for SML mode. For example:

```
(add-hook 'sml-load-hook '(lambda () (require 'sml-font)))
```

This ensures the SML extensions to 'font-lock' will be available once SML mode loads (from 'sml-font.el'—if you prefer 'hilit19' you should `(require 'sml-hilite)` instead.

The variable `sml-font-lock-extra-keywords` is for further customising 'font-lock' for SML mode. The value of the variable should be a list of strings, each of which is a regular expression that should match the desired keyword exactly. Here's an example:

```
(setq sml-font-lock-extra-keywords
      '("\\babstraction\\b" "\\bfunsig\\b" "=>" "::"))))
```

The `\b` marks a word boundary, according to the syntax table defined for SML mode. Backslash must be quoted inside a string. See section "Regexps" in *The Emacs Editor Manual*, for a summary of Emacs' regular expression syntax.

Finally, the variable `sml-font-lock-auto-on` can be used to control whether or not 'font-lock' should be enabled by default in SML mode buffers; it is enabled by default. The `sml-hilite` package is customisable, but only with regard to colour changes.

4.5 Advanced Topics

These forms are bloody useless; can't we have better ones?

You can indeed. `sml-insert-form` is extensible so all you need to do is create the macros yourself. Define a *keybord macro* (*C-x (* <something> *C-x)*) and give it a suitable name: `sml-addto-forms-alist` prompts for a name, say `NAME`, and binds the macro `sml-form-NAME`. Thereafter *C-c* ⟨RET⟩ *NAME* will insert the macro at point, and *C-u C-c* ⟨RET⟩ *NAME* will insert the macro after a `newline-and-indent`. If you want to keep your macros from one editing session to the next, go to your '.emacs' file and call `insert-kbd-macro`; you'll need to add `NAME` to `sml-forms-alist` permanently yourself:

[5] 'ftp://archive.cis.ohio-state.edu/pub/gnu/emacs/elisp-archive/misc/'

```
(defun my-sml-load-hook () "Global defaults for SML mode"
  ;; whatever else you do
  (setq sml-forms-alist (cons '("NAME") sml-forms-alist)))
```

If you want to create templates like 'case' that prompt for parameters you'll have to do some Lisp programming. The `tempo` package looks like a good stating point. You can always overwrite your own macros, but the builtin forms for 'let', etc., can't be overwritten.

I hate that indentation algorithm; can't I suppress it?

Ah, yes, a common complaint. It's actually very easy to use SML mode without the troublesome `sml-indent-line`:

```
(defun my-sml-load-hook () "Global defaults for SML mode"
  ;; whatever else you do
  (fset 'sml-indent-line 'ignore))
```

though `indent-relative-maybe` may conceivable be more useful than `ignore`.

The dedicated frame for ML is too huge; can it be made smaller?

Of course, you just have to modify the frame parameters. The variable `sml-display-frame-alist` can be defined explicitly in your '`.emacs`'; the default is a frame of 80 columns by 24 lines, and the icon name will be the same as the ML interaction buffer's name—something like `*mosml*`. I like a small, tidy font for this frame so I

```
(setq sml-display-frame-alist
      (cons '(font . "7x14") sml-display-frame-alist))
```

in my `inferior-sml-load-hook`. If you want fewer lines, try:

```
(setcdr (assoc 'height sml-display-frame-alist) 15)
```

or something.

Can SML mode handle more than one compiler running at once?

The question is whether you can! See the `sml-buffer` variable's on-line help (*`C-h v sml-buffer`*). Note that the SML mode compiler variables (see Section 3.4 [Process Defaults], page 11) are all buffer-local, so you can even switch between different ML compilers, not just different invocations of the same one. Well, you *can*.

What needs to be done to support other ML compilers?

Not that much really, at least not to create minimal support. The interface between SML mode and the compiler is determined by the variables `sml-use-command`, `sml-cd-command`, `sml-prompt-regexp` (which are easy to get right), and `sml-error-regexp`, and `sml-error-parser` (which are more tricky). The general template to follow in setting this up is in the files '`sml-{poly-ml,mosml}.el`'. These rules will not change, I hope:

- `sml-next-error` uses `sml-error-regexp` to locate the start of the next error report in the ML interaction buffer (P)

- `sml-next-error` calls `sml-error-parser`, passing P, and expects up to five return values in this order:
  1. file name in which the error occurs (F)
  2. start line of the error ($L > 0$)
  3. start column of the error (C)
  4. an Emacs Lisp expression to be `eval`'d at (L,C) in F (*EOE*)
  5. the actual text of the one-line error report (*MSG*)
- `sml-error-parser` can assume that P is the start of the next error message that the user is interested in—since she defines this point by defining `sml-error-regexp`.
- What `sml-error-parser` returns is a list. In the event of problems, I foresee the following needs:
  - if the file is the standard input, return `("std_in"` L C`)`
  - if the file cannot be inferred, return `(nil` L C`)`
  - if L=0, or the start cannot be inferred, return `(`F `nil` C`)`
  - if the start column cannot be inferred, return `(`F L `1)`

There's no need to return anything else. However, if you do want the errorful text in F highlighted you should return a simple Lisp expression in the fourth argument that'll compute the region. *EOE* will be called with point at character (L,C) in F, and should move point to the end of the errorful text. In fact, *EOE* can actually do anything you wish, but in the simplest cases it'll just `(forward-char 45)`, or

```
(progn (forward-line 4) (forward-char 37))
```

etc.. If it does more, make sure it leaves point at the end of the region in F—use `save-excursion` if switching buffers. *MSG*, if returned, will be echoed in the minibuffer.

Credit & Blame

SML Mode was written originally by Lars Bo Nielsen for Emacs 18.5n; later hacked for comint by Olin Shivers (who called it `ml-mode`); much later hacked by myself because it didn't seem to work... Fritz Knabe brilliantly posted the `hilit19` and `font-lock` functions on the net. Lars probably would recognise much of what remains, yet now there're menus, syntax highlighting, support for various ML compilers, Texinfo (hey!), and more than a little hope it'll work with a variety of Emacs 19s. But there are still things to do. Lars wrote:

> *The indentation algorithm still can be fooled. I don't know if it will ever be 100% right, as this means it will have to actually parse all of the buffer up to the actual line [...].*

This is still the main cause of grief; SML's syntax is a nightmare for Emacs modes, and of course opinions vary about proper indentation. But there may be something we can do...

Command Index

Variable Index

Key Index

Table of Contents