Editing and Running Standard ML under GNU Emacs

SML mode, Version v4_0
August 1999

Authors: Matthew J. Morley and Stefan Monnier

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 with next-error—just like 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 Contributors to the SML mode

Contributions to the package are welcome. I have limited time to work on this project, but I will gladly add any code that you contribute to me to this package.

Although the history of sml-mode is obscure, it seems that the following persons have made contributions to sml-mode:

- Lars Bo Nielsen wrote the original version of the code, providing the sml editing mode and the inferior-sml support.
- Olin Shivers (‘`shivers@ai.mit.edu`’) hacked the inferior-sml support to use comint and call the whole thing ml-mode.
- Steven Gilmore supposedly provided some early attempt at menubar support.
- Matthew J. Morley (‘`matthew@verisity.com`’) was maintainer for a long time (until version 3.4) and provided many additions and fixes in all areas.
- Frederick Knabe (‘`knabe@ecrc.de`’) provided the original code for font-lock and hilite support as well as for proper handling of nested comments and of all the string escape sequences.
- Matthias Blume (‘`blume@kurims.kyoto-u.ac.jp`’) provided a sml-make which was replaced by sml-compile.
- Monnier Stefan (‘`monnier@cs.yale.edu`’) completely reworked the indentation engine as well as most of the rest of the code and is the current maintainer since after version 3.4.

1.2 Getting started

With luck your system administrator will have installed SML mode somewhere convenient, so it will just magically all work—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:

```
(add-to-list 'load-path "/home/mjm/elisp")
(autoload 'sml-mode "sml-mode" "Major mode for editing SML." t)
(autoload 'run-sml "sml-proc" "Run an inferior SML process." t)
```

The first line adjusts Emacs' internal search path so it can locate the Lisp source you have copied to that directory; the second and third lines tell 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

```
(add-to-list 'auto-mode-alist '("\\.\\(sml\\|sig\\)\\'" . sml-mode))
```

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 11, for a list of (info) documented functions. See [Variable Index], page 11, for a list of user settable variables to control the behaviour of SML mode.

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 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 8, for some examples.

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).

`indent-for-tab-command` [Command]
: Key: TAB

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

`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-space` [Command]

Key: *M-SPC*

When the point is after a keyword like 'let', this inserts the corresponding predefined skeleton if one exists. Else it just inserts a space. Another way to insert those skeletons is to use `sml-insert-form`, described below.

`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` [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.

`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 8 for further details.

2.4 Indentation defaults

Several variables try to control the indentation algorithm and other features of SML mode. Most of them are still in flux so they are not described here yet. If the default values are not acceptable you can set these variables permanently in your '`.emacs`' file. See Chapter 4 [Configuration], page 8, for details and examples.

`sml-indent-level` [Variable]
Default: `4`

This variable controls the block indentation level.

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 7, for some hints.

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

3.1 Starting the compiler

Start your favourite ML compiler with the command

M-x run-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 7).

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 run-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")
```

`run-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 run-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.

`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-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 7).

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 7). 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-send-buffer` [Command]

Key: *C-c C-b*

Send the contents of the current buffer to ML.

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 7.

`next-error` [Command]

Key: *C-x `*

Jump to the source location of the next error reported by the compiler. All the usual error-navigation commands are available, see see section "Compilation Mode" in *The Emacs Editor Manual*.

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.

`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 `next-error` can be useful:

`sml-error-regexp-alist` [Variable]
Alist that specifies how to match errors in compiler output. Each elt has the form (REGEXP FILE-IDX LINE-IDX [COLUMN-IDX FILE-FORMAT...]) If REGEXP matches, the FILE-IDX'th subexpression gives the file name, and the LINE-IDX'th subexpression gives the line number. If COLUMN-IDX is given, the COLUMN-IDX'th subexpression gives the column number on that line. If any FILE-FORMAT is given, each is a format string to produce a file name to try; %s in the string is replaced by the text matching the FILE-IDX'th subexpression.

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.

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 assignments. 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. 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.

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 assignments. 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-mode-hook () "Global defaults for SML mode"
  (define-key sml-mode-map "\C-cd" 'sml-cd))
(add-hook 'sml-mode-hook 'my-sml-mode-hook)
```

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

```
(defun my-inf-sml-mode-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-mode-hook 'my-inf-sml-mode-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 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.

The indentation scheme (strangely enough) also relies on the highlighting code to properly handle nested comments, which is yet another reason to turn on highlighting. To turn on highlighting, use either of:

```
M-x font-lock-mode
(add-hook 'sml-mode-hook 'turn-on-font-lock)
(global-font-lock-mode 1)
```

The first will turn it on in the current buffer. The second will turn it on in all sml-mode buffers. The last will turn it on everywhere. This is valid for Emacs but maybe not for XEmacs. Check font-lock documentation if you encounter problems.

4.4 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:

```
(defun my-sml-mode-hook () "Global defaults for SML mode"
  ;; whatever else you do
  (add-to-list 'sml-forms-alist '("NAME" . FUNCTION)))
```

If you want to create templates like 'case' that prompt for parameters you'll have to do some Lisp programming. The `skeleton` package is a good stating point. Better yet, you can reuse the wrappers used by sml-mode itself in your sml-mode-hook:

```
(add-hook 'sml-mode-hook
  (lambda ()
    (sml-def-skeleton "case" "Case expr: "
      str " of" \n _ " => ")))
```

This will redefine 'case' in order to leave the 'of' on the first line. See the documentation of `skeleton-insert` to get a better understanding of how this works.

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

Ah, yes, of course, but this manual will not tell you how.

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

Sure, just rename the '`*sml*`' buffer and then use `run-sml` as usual.

What needs to be done to support other ML compilers?

Not much really. Just add the right regular expressions to `sml-error-regexp-alist` and that should be all.

Command Index

I

N

R

S

Variable Index

I

S

Key Index

;

C

L

M

T

Table of Contents