

STANDARD ML OF NEW JERSEY

The version of the ML programming language that we will be using in CS301 this semester is Standard ML of New Jersey (also know as SMLNJ or just SML). This handout contains notes on how to run SMLNJ on the Linux workstations.

In the SMLNJ programming environment, you will use Emacs to create program files and will load these files into an interactive SML top-level interpreter that executes in a special Emacs buffer. Type inference is performed on all SML programs and you cannot test your programs until they pass the type checker. Understanding error messages from the type checker and using them to pinpoint type errors in your program are important skills that you will need to hone. Collections of ML modules are best compiled using a *configuration manager* that serves the purposes of “makefile” or “projects” in many C and Java development environments. You will need to learn the rudiments of configuration management to compile your programs.

There are two ways to run SMLNJ on the Linux workstations: within a shell window and within Emacs. These approaches are described below in Sections 1 and 2. It is recommended that you use the Emacs interface, since it simplifies many interactions. Nevertheless, you should still read Section 1 as many of the notes still apply to the Emacs interface. The SMLNJ configuration manager (CM) is described in Section 3.

This handout only scratches the surface of SMLNJ. For more detailed documentation, browse the official SMLNJ web page:

<http://cm.bell-labs.com/cm/cs/what/smlnj/index.html>

1. Running SML in a Shell Window.

1.1 Launching the SML interpreter

The simplest way to run SML on the Linux workstations is within a Linux shell window. Within such a window, execute `sml`. This will print a herald on the screen and eventually the ``-`` prompt of SML.

```
[cs301@paxi ~]$ sml
Standard ML of New Jersey, Version 110.0.6, October 31, 1999 [CM; autoload
enabled]
-
```

You can now evaluate expressions by typing them in followed by a semi-colon and a line return. The semi-colon tells the interpreter that you are done with the expression. This allows you to have expressions with multiple lines; SML will show a ``=` at the beginning of every line of a multiple line expression. For example, here is the transcript of a session in which two single line expressions and one multiple line expressions are evaluated:

```

1. 1 + 2;
   val it = 3 : int
   - val lst = map (fn x => x * 2) [4,7,3];
   val lst = [8,14,6] : int list
   - let val a = 1 + 2
     =     val b = 4 * a
     =   in (a,b)
     = end;
   val it = (3,12) : int * int

```

If you forget the semi-colon at the end of an expression, SML will think you are continuing the expression onto the next line. In this case, you can just type a semi-colon followed by a line return to indicate that you are done. For example:

```

- 2 + 3
= ;
val it = 5 : int

```

The SML interpreter expects that the unit of evaluation will be a declaration, typically one of the form `val name = exp`, where `exp` is an expression. If you just enter an expression `E`, the SML interpreter treats it as a declaration of the form `val it = E`.

1.2 Loading Files

It is tedious to type all expressions directly at the SML interpreter. It is especially frustrating to type in a long definition only to notice that you made an error near the beginning and you have to type it in all over again. In order to reduce your frustration level, it is wise to use a text editor (e.g., Emacs) to type in all but the simplest SML definitions. This way, it is easy to correct bugs and to save your definitions between different sessions with the SML interpreter. (Note: the file extension that you should use for SML files is `".sml"`. Using this extension will enable various SML features in Emacs. See Section 2.2 for details.)

If *filename* is the name of a file containing SML expressions and definitions, evaluating

```
use("filename")
```

will evaluate all of the expressions in the file, one by one, as if you had typed them in by hand. The `use` function can be used within a file to load other files.

The filename given to `use` may be either an absolute pathname (e.g. `/usr/users/gdome/private/ml/test.sml`) or a pathname relative to the current working directory. By default, the current working directory for the SML interpreter is the current working directory of the shell in which it was invoked, and by default this is your Nike home directory (e.g. `/usr/users/gdome`). So rather than evaluating

```
use("/usr/users/gdome/private/ml/test.sml")
```

you could instead evaluate

```
use("private/ml/test.sml")
```

It is typical to load many files (or the same file many times) from the same directory. For this reason, it is useful to be able to change the current working directory within the SML interpreter. To do this, evaluate

```
Posix.FileSys.chdir("dirname")
```

where *dirname* is the name of the directory which you want to become the new current working directory. (This name can either be an absolute pathname, or relative to the current working directory.) For example:

```
Posix.FileSys.chdir("private/ml")
```

It's easy to lose track of what the current working directory is. The `Posix.FileSys.getcwd()` command can be used to tell you what the current working directory is (as an absolute pathname). For example:

```
- Posix.FileSys.getcwd();  
val it = "/usr/users/cs301/private/ml" : string
```

1.3 Exiting SML

To terminate your session with the SML interpreter, type C-d (i.e., control D). (It is common in Unix systems for C-d to represent the “end of input”.)

2. Running SML within Emacs

You could do all of your SML programming in CS301 using just the techniques outlined in Section 1 above. However, you will find yourself constantly swapping attention between the Emacs editor (where you write your code) and the SML interpreter (where you evaluate your code). In particular, whenever you make a change to your Emacs file, you will have to save the file and reload it into the SML interpreter.

To reduce the overhead of swapping between Emacs and SML, you can run SML within Emacs. The following subsections outline the key aspects of running SML within Emacs.

2.1 Configuring Your .emacs File

In order to run SML within Emacs, you must tell Emacs about the whereabouts of the SML libraries and must tell it to load these libraries. You can do this by adding the following lines to your `~/.emacs` file (which is loaded every time you start Emacs):

```
;; Emacs initialization for SML mode  
(setq load-path  
  (append '("/usr/share/emacs/20.3/lisp/sml-mode-3.3"  
            "/usr/share/emacs/site-lisp/sml-mode-3.3")  
          load-path))  
  
(require 'sml-site)  
(add-hook 'sml-load-hook '(lambda () (require 'sml-font)))
```

You only need to perform the above step once. Once your `~/.emacs` file is configured as above, then you should have access to the features of SML mode automatically thereafter.

2.2 SML Mode

An Emacs buffer can be in various modes. Each mode tells the buffer how keystrokes in that buffer should be interpreted. The most useful mode for editing SML code is SML mode. You can tell a buffer to enter SML mode by typing `M-x sml-mode .` (Notational convention: M-x, pronounced “Meta x”, is entered by typing the “Meta” key and “x” key at the same time. On PC keyboards, the key labeled “Alt” is usually treated as the “Meta” key. On keyboards without a true “Meta” key, you can instead press the ESCape key *followed by* pressing the “x” key.) You

can tell that a buffer is in SML mode by the appearance of the word "SML" in the status line at the bottom of Emacs.

Whenever you edit a file that ends in `.sml`, Emacs will automatically enter SML mode for you. For this reason, it is wise to use the `.sml` extension for all of your SML files.

SML mode helps you write SML code because it understands the formatting conventions for SML code. Like many Emacs modes, it helps you match parentheses by flashing the matching open parenthesis whenever you type a close parenthesis. Additionally, SML mode helps you put your code in pretty-printed format. Typing the TAB key will indent the code on that line according to the SML indentation conventions. You should get into the habit of hitting TAB after every return so that you start typing the next line at the appropriate indentation level. You can format an entire expression by typing ESC C-q when the cursor is at the first character of the expression. Keeping your SML expressions indented properly is important for reading and debugging code. Indenting the code will often highlight that the structure of the expression has a bug.

SML mode also understands what SML keywords are, and will color keywords, strings, and comments different from the rest of the text to highlight them.

2.3 Launching SML in Emacs

To start an SML session within Emacs, execute the Emacs command `M-x sml`. This will create a buffer named `*sml*` that will be the location of the SML interpreter. The `*sml*` buffer for SML is similar to the `*scheme*` buffer in Scheme.

2.4 Interacting with the `*sml*` buffer

The simplest way to interact with the `*sml*` buffer is to visit the buffer and enter expressions at the prompt. But there are a number of `M-x` commands that simplify interactions. For instance, `M-x sml-cd` will prompt for a directory name and change the current working directory to that directory; this is much more convenient than using `Posix.FileSys.chdir`. And a number of `M-x` commands will send text from another buffer in SML mode to the `*sml*` buffer. These are helpful when creating small programs, but for larger programs you probably want to use the configuration manager (see Section 3):

M-x command	Description
<code>sml-send-buffer</code>	Sends the contents of the entire buffer to the <code>*sml*</code> buffer as if you had typed it directly into the <code>*sml*</code> buffer.
<code>sml-send-function</code>	Sends the text of the current declaration to the <code>*sml*</code> buffer as if you had typed it directly into the <code>*sml*</code> buffer.
<code>sml-send-function-and-go</code>	Like <code>sml-send-function</code> except that it also displays the <code>*sml*</code> buffer and moves the cursor there.
<code>sml-send-region</code>	Sends the text of the current region to the <code>*sml*</code> buffer as if you had typed it directly into the <code>*sml*</code> buffer.
<code>sml-send-region-and-go</code>	Like <code>sml-send-region</code> except that it also displays the <code>*sml*</code> buffer and moves the cursor there.

2.5 Exiting SML in Emacs

You can exit SML in Emacs by killing the `*sml*` buffer (using `C-x C-k`).

3. The SML Configuration Manager

3.1 Dependency Management

Recall that SML is more strict about order of bindings than Scheme or Java. That is, the only way you can define something before you use it is to collect function definitions into a recursive scope by using the `and` keyword to glue together the recursive functions. (In contrast, `fun` starts a new scope.) This means that when you employ the `use` command to load files, you have to make sure that you load them in the correct order so that each name gets defined before it is used. If you don't do this, then it's easy to have some functions referring to an old definition rather than the new one.

For example, suppose we have interactively defined the following two functions:

```
- fun foo(x) = 2*x;
val foo = fn : int -> int

- fun bar(y) = foo(y + 1);
val foo = fn : int -> int
```

Next, suppose we undertake the following sequence of actions:

1. We put the `bar` function in a file `bar.sml` and the `foo` function in a file `foo.sml`.
2. We create a file `both.sml` that loads both files via the following code:

```
use("bar.sml")
use("foo.sml")
```

3. We edit `foo.sml` so that the definition of the `foo` function is now:

```
fun foo(x) = 10*x;
```

4. We go to the SML interpreter and evaluate:

```
use("both.sml");
```

If now we test the two functions we get somewhat surprising results:

```
- bar 5;
val it = 12 : int

- foo 6;
val it = 60 : int
```

But if `bar(y)` is defined to be `foo(y + 1)`, how can this be? Since `"bar.sml"` was loaded before `"foo.sml"`, it refers to the old definition of `foo` (the one that doubled its input). Only after `bar` is defined is the new version of `foo` defined (the one that multiplies its input by 10).

This example is rather contrived, but even in programs of modest size, managing the dependencies between multiple files by hand can be non-trivial. Additionally, in very large programs (ones that contain hundreds of files), it is desirable when making a change to one file to process (i.e., type check and compile) only those files that depend on it, not all the files. Again, this comes down to managing dependencies between files.

3.2 The SMLNJ Configuration Manager

Many program development environments (such as Symantec Café and Metrowerks Code Warrior for Java) support a notion of a *project* that is a collection of files in a program. The user

specifies which files are in the project and the development environment is responsible for tracking the dependencies between the files. When one rebuilds a project in Symantec Café after editing a single source file, only those other files that depend on it are recompiled. Other files not depending on the edited file do not have to be reprocessed. In Linux-based programming environments, such dependency information is usually expressed in a “makefile”.

SMLNJ supports similar notions in its *configuration manager*. You must specify to the configuration manager which files are in the program, and it will automatically track the dependencies between them.

The way you specify the files in a program are to list them in a configuration file, which should have a “.cm” extension and whose contents should have the following syntax:

```
Group is
  filename_1
  .
  .
  filename_n
```

where *filename_i* is one of the files in the program. The order of the files in the .cm file is immaterial; the configuration manager will deduce the dependencies between them regardless of the order.

For example, for the (contrived) example above, we could make a file `both.cm` with the following contents:

```
Group is
  bar.sml
  foo.sml
```

In addition to referring to .sml files within a .cm file, you can also refer to other .cm files. This effectively adds all the files (transitively) mentioned in the .cm file to the list of files to be checked for dependencies. For example, suppose we create a file `baz.sml` that uses the `foo` and `bar` functions from above. We can make a configuration file `baz.cm` whose contents are:

```
Group is
  both.cm
  baz.sml
```

We could later reference `baz.cm` in yet another configuration file for a program that used any or all of the functions `foo`, `bar`, and `baz`.

It is OK for the same file to be referenced by multiple paths through a configuration file. For instance, the following configuration file causes no trouble even though `foo.sml` is referenced directly and indirectly (both through `both.cm` and `baz.cm`):

```
Group is
  foo.sml
  both.cm
  baz.cm
```

Neither configuration manager (nor the underlying SMLNJ language) can handle mutual dependencies in declarations across file boundaries. For example, if you have mutually recursive functions `f` and `g`, you *must* define these in a single file – in fact, within a single mutually recursive declaration (i.e., `fun f ... and g ...`). If you attempt to split the two declarations across two files, the configuration manager will complain about cyclic dependencies. If you get such an error, you will need to rearrange the contents of your files to remove the cyclic

dependencies. That is, you must guarantee that the dependencies form a DAG (directed *acyclic* graph).

It is good hygiene to explicitly document the dependencies of *every* `.sml` file with an associated `.cm` file. That is, for every `.sml` file you should have a `.cm` file that explicitly lists the other files it depends on. An advantage of this is that it simplifies the creation of all the `.cm` files. Suppose that file `A.sml` exports structure `A`. If we have a file `A.cm`, then the configuration file for any other file using structure `A` can just refer to `A.cm` rather than all the individual files on which `A` depends.

3.3 Invoking the Configuration Manager in a Shell

To launch a version of SMLNJ supporting configuration management directly from a shell, you need to invoke `sml-cm` rather than `sml`. In the resulting read-eval-print loop, you can invoke the configuration manager (CM) on a `.cm` file via `CM.make'`. For example:

```
CM.make' ("both.cm");
```

This causes the configuration manager to compile all of the files in the dependency DAG denoted by `both.cm` and to load all the exported declarations of these files into the interpreter in the order specified by the dependencies. This will avoid the sort of ordering bug demonstrated in Section 3.1

Actually, the configuration manager is smarter than this. It will only compile files in the dependency DAG if the dependencies say it has to. For instance, if you have a dependency DAG with hundreds of files that you have already compiled via CM. If you only edit one of these files and invoke CM again, CM will only recompile those files that depend on declarations in the edited file.

Typically, invoking `CM.make'` will cause lots of error messages to be printed as the SMLNJ front end checker finds syntax and type bugs in your program. Each error message lists a file and a range of line numbers narrowing down where the error is. You have to fix all such errors before you can run your program. (In addition to errors there are sometimes “warnings” – you can usually ignore these, but you should read what they say anyway.)

3.4 Invoking the Configuration Manager In Emacs

To invoke the version of SMLNJ with CM in Emacs, first type `C-u M-x sml`. In the Emacs mini-buffer at the bottom of the screen, you will be prompted for “ML command:”. Enter `sml-cm` followed by `Return`. You will then be prompted for “Any args:”; just type `Return` for this. This will create a new `sml-cm` process and a buffer named `*sml-cm*` through which you can communicate with that process.

Now you can go to the `*sml-cm*` buffer and start evaluating SML expressions. You can invoke the configuration manager via `CM.make'`, as explained in Section 3.3.

3.5 CM Documentation

For more detailed information about the SMLNJ configuration manager, consult the CM user manual:

```
http://cm.bell-labs.com/cm/cs/what/smlnj/doc/CM/index.html
```