Standard ML for 15-212-X

August, 1996

This guide provides a brief introduction to Standard ML and the environment which is used for this year's ML-track of the 15-212 course at Carnegie Mellon University. This is *not* a reference manual for the Standard ML language, but it rather is intended to help you get started with the system in use for the 15-212 course. This guide was written by Iliano Cervesato (`iliano@cs.cmu.edu`), Carsten Schürmann (`carsten@cs.cmu.edu`) and Frank Pfenning (`fp@cs.cmu.edu`), on the basis of a previous version by Peter Lee (`petel@cs.cmu.edu`).

There are several sources of information about the language. For textbooks, we recommend either *ML for the Working Programmer, 2nd edition*, by L. Paulson (Cambridge University Press, 1996), or *Elements of ML Programming*, by J. Ullman (Prentice Hall, 1994). Brief introductions to the language may also be found on-line via the world-wide web at URL:

`http://foxnet.cs.cmu.edu/sml.html`

In particular, the *Introduction to SML* and the *Four Lectures on Standard ML* are useful sources of examples and explanations about Standard ML (SML).

SML is a practical programming language incorporating a large number of modern programming language concepts. As such, it is unique in having a complete formal definition, which is still subject of current research especially in the areas of mathematical semantics, type theory, language design, and compiler design. The definition of SML has recently been upgraded to reflect the experience acquired with the language and its theory. This new definition is known as *Standard ML'96.* We will adopt SML'96 for the course. More information about can be found at the homepage:

`http://foxnet.cs.cmu.edu/15-212-X/`

On this page you find basic documentation about SML'96. The new definition will shortly be published, superseding the "old" *The Definition of Standard ML*, by R. Milner, M. Tofte, and R. Harper (The MIT Press, 1990). Paulson's book above (note the *2nd edition!*) relies on SML'96.

SML at CMU

For 15-212, we use *MLWorks* by *Harlequin Incorporated.* Differently from previous SML implementations, such as *Standard ML of New Jersey* adopted in previous editions of this course, MLWorks is an integrated programming environment providing an interpreter, access to various external editors (in particular to *emacs*), a debugger and management tools for modular programming, such as a compilation manager and a browser. The major source of documentation about MLWorks is *MLWorks, User's Guide to the Environment*, available on the andrew file system as

`/afs/andrew/scs/cs/15-212-X/handouts/MLWorks-Users-Guide.ps`

MLWorks is only available for the Sun4 architecture (SPARCstations) so far. The binary can be found on the Andrew network under

`/afs/andrew/scs/cs/15-212-X/mlworks/bin/mlworks`

MLWorks provides an X-window user interface. If you run MlWorks on a machine without the X-windows environment, start it with the flag `-tty`.

Interacting with MLWorks

When you start MLWorks, an initial window called the *console* window will appear. The displayed version number should be

```
MLWorks(TM) Version1.0 (beta) Revision 10
```

The window also offers a number of pull-down menus which are described in the *User's Guide*.

The ML interpreter is launched from the menu **Tools>Listener**. A *listener* window will appear, prompting you for an SML expression: This command will then be compiled and executed, and the result displayed.

When prompted, you can type in a *top-level declaration*. There are many kinds of top-level declarations in SML, and you will learn about them in this course. For example, the following is the declaration of a function called `inc` that increments its integer argument. (In the following examples, "`MLWorks>`" represents the MLWorks prompt, and the text type-set in the `teletype` font is the user input. *Italic* is used for the output from the MLWorks listener. The symbol ↩ represents a carriage return on Unix-based systems.)

```
MLWorks> fun inc (x:int) = x + 1;↩
 val inc = fn : int -> int
```

The text "`fun inc (x:int) = x + 1`" is the declaration for the `inc` function. (Don't worry if you don't understand SML programs yet; you will learn about this during the course.) The semicolon ("`;`") is a marker that signalizes the MLWorks system that it should perform the following actions: *elaborate* (that is, perform typechecking), *compile* (to machine code), *execute*, and finally *print* the result of this declaration. After all of this, the system then prompts for new input and the whole process starts again. This is the so-called "top-level loop". To exit from the MLWorks listener, select **File>Close** or simply type the end-of-file character (Control-D) at the prompt.

In the example above, the printed result shows that `inc` is a function that takes an integer argument and yields an integer result. Actually, it is important to note that, in SML, functions are "first-class" values, fundamentally no different than other values such as integers. So, to be more precise, it is better to say that the identifier `inc` has been *bound* to a value (which happens to be a function, as denoted by the `fn` keyword above) of type *int* `->` *int*.

If we had left out the semicolon, then the elaboration, compilation, execution, and printing would have been deferred and another top-level declaration would be expected. When a semicolon is finally entered (perhaps after several more top-level declarations), all of the declarations since the last semi-colon would be processed in sequence. For example:

```
MLWorks> fun inc (x:int) = x + 1↩
fun f (n:int) = (inc n) * 5;↩
 val inc = fn : int -> int
 val f = fn : int -> int
```

In this example, we have defined the `inc` function as well as a function `f` that uses `inc`. Notice, that no prompt was given for the second function.

In the interactive top-level loop, the simplest form of input is an expression. For example, after typing in the declarations for `inc` and `f` above, we can now call `f` by typing in:

```
MLWorks> f (2+4);↩
 val it = 35 : int
```

Notice that the interactive system has bound the identifier `it` to the result of compiling and executing the expression `f (2+4)`.

You may have experience with other languages whose implementations support a similar kind of interactive top-level loop. For example, most implementations of the Lisp, Scheme, and Basic languages support top-level loops. If so, then you might expect that re-defining a function will change the binding of the function name, as well as any other functions that call that function. However, in the MLWorks system, this is not the case. For example, suppose we wish to change the definition of the `inc` function, so that it increments by two instead of one:

```
MLWorks> fun inc (x:int) = x + 2;↩
 val inc = fn : int -> int
```

In typical Lisp and Scheme systems, such a re-definition would cause the function `f` to change as well, since `f` calls `inc`. But in the MLWorks system, `f`'s binding does not change, so in fact referring to `f` now still yields the original function:

```
MLWorks> f (2+4);↩
 val it = 35 : int
```

To understand why the MLWorks system behaves in this way, consider what would happen if we re-defined `inc` so that it had a type different than *int* `->` *int*, for example:

```
MLWorks> fun inc (x:int) = (x mod 2 = 0);↩
 val inc = fn : int -> bool
```

Here, `inc` has been changed to a function that returns `true` if and only if its integer argument is even. Now, if `f` should also be changed to reflect this re-definition (as it would be in Lisp and Scheme systems), it would fail to typecheck. This is not necessarily a bad thing, but at any rate the MLWorks system does not bother to go back to earlier top-level declarations and re-elaborate them; hence, `f`'s binding is left unchanged.

If you are already familiar with the SML language, then you can think of the sequence of top-level declarations typed into an MLWorks interactive top-level loop as being in nested `let`-bindings:

```
let fun inc (x:int) = x + 1 in
  let fun f (n:int) = (inc n) * 5 in
    let fun inc (x:int) = x + 2 in
      ⋮
```

(Again, don't worry if you don't understand this last example. It will be covered in class.)

Using files and the pervasive environment

Instead of typing your program into the interactive top-level, it is more productive to put your program into a file and then load it into the MLWorks system. To do so use the built-in function `use`. For example:

```
MLWorks> use "myprog.sml";↩
 [opening myprog.sml]
   ⋮
 val it = () : unit
```

The `use` function takes the name of the file (of type `string`) to load. If the file exists, it is opened and read, with each top-level declaration in the file processed in turn (and the results printed on the standard output). The "result" of the `use` function is the unit value ("`()`").

For those who prefer clicking than typing, the `use` function can also be invoked from the menu **File>Use file...**; a file dialog will appear and allow you to choose the file to `use`.

Editing SML programs using `emacs`

We recommend using a release of the `emacs` editor (such as `xemacs`) to edit your SML programs and also to manage interaction with the `sml` system. To do this, select **Emacs Server** from the menu **Preferences>Editor...**, include the following lines in your `.emacs` file

```
(setq load-path (cons "/usr/contributed/lib/mlworks/env/emacs" load-path))

(autoload 'mlworks-server "mlworks-server" "The MLWorks server" t)
(autoload 'sml-mode "sml-mode" "Major mode for editing Standard ML programs." t)

(setq auto-mode-alist (cons '("\\.sml$" . sml-mode) auto-mode-alist))
```

start `emacs` and type *Meta*-`x mlworks-server`. The error manager will then communicate with your `emacs` session and locate errors directly in the source file.

The above commands also load the "sml mode", a special editing mode that will be invoked any time you edit a file with the extension "`.sml`." As in other special editing modes, using the Tab key or Control-j will cause emacs to attempt to indent your code in a pleasing way. Control-c followed by Tab will indent the current region. Since SML's syntax is rather complex, the sml mode indentation can be rather haphazard at times. Still, many people find it to be quite useful. A particularly useful key combination is "Meta" along with a vertical bar (`M-|`); this creates a template for an arm of a case expression or clause of a function.

Other editors can be used in conjunction with MLWorks. Consult the MLWorks' *User Guide* for details.

Making sense of error messages

As with most compilers, the MLWorks system can often produce error messages that are hard to decipher. The problem is compounded by the fact that SML has polymorphic typing, which makes it very difficult for the compiler to identify precisely the real source of type errors. On the other hand, once all of the compile-time type errors are removed, there is a wonderful tendency for SML programs to simply *work*!

MLWorks displays the error messages in a dedicated window (the *error browser*) with often intelligible messages. If the error was present in a file and the interaction with the editor has been set up propertly, clicking on the **action>edit** menu item will highlight the (approximate) location of the error in the source file. More about errors and error handling can be found in the *User's Guide*.

Type mismatches. The most common kind of error is the simple type mismatch. For example, suppose we have the following code in a file called `myprog.sml`:

```
fun inc x = x + 1
fun f n = inc true
```

Notice that a semi-colon is not needed here, since the end-of-file marker will serve the same purpose. Now, if we "use" this file, we get the following error message:

```
MLWorks> use "myprog.sml";↩
 Line 2: error: Function applied to argument of wrong type
   Near: inc true
   Required argument type: int
   Actual argument type: bool
     Type clash between int and bool
```

This message indicates that the expression `inc true`, on line 2, is guilty of a type mismatch. The function `inc` is being applied, and its domain (argument type) is `int` but the operand actually has type `bool`. Selecting **Action>Edit**, or double-clicking on the error in the upper part of the error browser window will locate the cursor at the right position in your file and highlight the faulty term.

Errors in derived forms. To see a simple example of how error messages aren't always so illuminating, consider the following code:

```
fun fact 0 = 1
  | fact n = n * fact true
```

Here, we have attempted to define the factorial function, but in the recursive call we have (stupidly) applied the `fact` function to the boolean value true instead of to the integer argument `n-1`. The error message given by the MLWorks system is as follows:

```
 Line 1 to 2: Type mismatch in recursive value binding for fact
   Near: fn 0 => ...
   Pattern type:    bool -> int
   Expression type: int -> int
     Type clash between int and bool
```

Despite the fact that the error is "clearly" in the recursive call to `fact`, the message indicates that the error is somewhere between line one and line two — this is the entire program! Another confusing aspect of this error message is that the function declaration is printed out in a form that does not closely resemble our original program. This is because many of SML's constructs are "derived forms," in other words, essentially macros that expand into a more basic "core" language. The MLWorks system always prints out code in terms of the core language, never the derived forms.

Unresolved overloading. Certain SML built-in functions have been assigned several types, for convenience. For example, the multiplication operator, "`*`", denotes both integer and real multiplications. This is called *overloading.* This issue of overloading of the arithmetic operators leads to another possible source of confusion for the novice SML programmer. Consider, for example, the following declaration of a function for squaring numbers:

```
fun square x = x * x
```

The resulting error message is:

```
 Line 1: error: Unresolved overloading for *
```

Because there is not enough information in this program to determine whether the * is for integers or for reals, the error message is given, complaining about the inability to "resolve" the overloading.

The simple fix for this kind of error is simply to declare the type of one of the arguments to (or the result of) the arithmetic operation, for example:

```
fun square x = x * x : int
```

It is not uncommon to spend quite a long time tracking down the source of a type error. (Actually, the time spent doing this is almost always much less than the time it takes to track down the same error without the benefit of static typechecking!) A common way to narrow down the possibilities, and also to improve the precision of the error messages produced by the compiler, is to annotate the program with explicit types, in the way that we have done above. It is particularly helpful to annotate the types of formal parameters, for example:

```
fun square (x : int) = x * x
```

This is similar to the declaration of parameter types in languages such as C and Pascal. Of course, in those languages the declarations are required; in SML they are optional.

The value restriction. In compliance with the definition of Standard ML'96, MLWorks implements something called the *value restriction.* Essentially, this restricts polymorphism to expressions that clearly are values, specifically single identifiers and functions. When this restriction is violated, the error message, "non-generalizable type variable," is given. For example, the following program results in this error:

```
fun id x = x

fun map f nil    = nil
    | map f (h::t) = (f h) :: (map f t)

val f = map id
```

The message given is

Line 6: error: Free imperative type variable at top level

which indicates that the expression `map id` is polymorphic, but not syntactically a value (that is, not an identifier or lambda expression), and hence this violates the value restriction. If time permits, the reasons for this restriction will be covered later in this course.

In some cases the compiler can determine from context that an expression like `map id` that appears polymorphic can be given a non-polymorphic type. In this case the compiler does not report an error. For example,

```
let
    val x = ref []
in
    x := [3]
end
```

is accepted as correct program.

Syntax errors. Because the syntax of SML is rather complex, there are several common errors that novices tend to make. One of the most common has to do with the syntax of patterns in clausal-form function declarations and case expressions. Consider the following code:

```
datatype 'a btree = Leaf of 'a
                  | Node of 'a btree * 'a btree

fun preorder Leaf(v)   = [v]
  | preorder Node(l,r) = preorder l @ preorder r
```

The MLWorks system complains vigorously over this:

> *Line 4: error: Value constructor Leaf used without argument in pattern*
> *Line 5: error: Value constructor Node used without argument in pattern*
> *Line 4 to 5: error: type mismatch in recursive value binding for*
> *preorder*
> *Near: fn _id216 => ...*
> *Pattern type: 'a -> ('a * 'a) list*
> *Expression type: 'a -> 'a * 'a -> ('a * 'a) list*
> *Type clash between*
> *('a * 'a) list*
> *and*
> *'a * 'a -> ('a * 'a) list*

The problem here is that `Leaf` and `Node` are patterns that are syntactically separate from, respectively, the `(v)` and `(l,r)` patterns. The (admittedly strange) syntax of SML requires extra parenthesization:

```
fun preorder (Leaf v)     = [v]
  | preorder (Node(l,r)) = preorder l @ preorder r
```

This is true in all contexts where patterns are used, including clausal-form function declarations, case expressions, and exception handlers.

Another rather confusing part of the syntax has to do with the interaction between case expressions, exception handlers, and clausal-form function declarations. Consider the following function, taken in slightly modified form from the MLWorks library (which is described below):

```
datatype 'a option = NONE | SOME of 'a

fun filter pred l =
  let fun filterP (x::r, l) =
            case (pred x) of
              SOME y => filterP(r, y::l)
            | NONE   => filterP(r, l)
        | filterP ([], l) = rev l
  in
    filterP (l, [])
  end
```

In this example, the local function `filterP` is defined in two clauses, the first handling the case of a non-empty list argument, and the second handling the empty list. In the first clause, a case expression is used. The syntactic ambiguity arises from the fact that it takes too much "lookahead" to figure out whether or not the second clause of `filterP` is actually the third arm of the case expression. This leads to the following rather cryptic error message:

Line 8: error: Non-constructor filterP used in pattern
Line 8: error: Unexpected '=', inserting '=>'
Line 8: error: Non-constructor filterP used in pattern
Line 8: error: Reserved word 'op' required before infix identifier '='

As before, parenthesization fixes the problem:

```
fun filter pred l =
  let fun filterP (x::r, l) =
            (case (pred x) of
               SOME y => filterP(r, y::l)
             | NONE   => filterP(r, l))
        | filterP ([], l) = rev l
  in
    filterP (l, [])
  end
```

Alternatively, in this example we can also exchange the two clauses of `filterP`:

```
fun filter pred l =
  let fun filterP ([], l) = rev l
        | filterP (x::r, l) =
            case (pred x) of
              SOME y => filterP(r, y::l)
            | NONE   => filterP(r, l)
  in
    filterP (l, [])
  end
```

As with many programming languages, the basic advice to follow is:

When in doubt, parenthesize.

Further issues

MLWorks contains numerous commands and options beyond the scope of this document. We suggest a careful reading of the *User's Guide* for getting acquainted with issues such as debugging, tracing, etc.