A quick introduction to SML

CMSC 15300

April 9, 2004

1 Introduction

Standard ML (SML) is a *functional language* (or *higher-order language*) and we will use it in this course to illustrate some of the important concepts. We will only use a limited set of core features in this course, SML is also used in some of the advanced courses (*e.g.*, the compiler sequence 22610-22630), where the full language comes into play. The purpose of this note is to explain how to run SML/NJ to test the small code examples that you write.

2 Running SML/NJ

The department's Linux machines and the MacLab Macs have SML/NJ installed. You can also download the system, which is open source, from `http:smlnj.org`.

You run SML from a shell (the Terminal application on MacOS X). At the command prompt, type the command **sml**, which will start up the SML *read-eval-print* loop (also known as the *top-level loop*):

```
% sml
Standard ML of New Jersey v110.45 [FLINT v1.5], February 13, 2004
-
```

The "`-`" symbol is the SML prompt. In these examples, we follow the convention of typesetting the user's input in *`italics`*. To exit the

When we type an SML expression or definition at the SML prompt, it is compiled, evaluates, and its result is printed. For example:

```
- 5-3;
val it = 2 : int
- 3=4;
val it = false : bool
-
```

Note that the semicolon terminates the expression. We can also *bind* names to the result of expressions:

```
- val a = 5-3;
val a = 2 : int
- val b = a=2;
val b = true : bool
-
```

We can also define named functions at the command line:

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

To exit the top-level loop, type the *end-of-file* character (typically Control-D).

```
% sml
Standard ML of New Jersey v110.45 [FLINT v1.5], February 13, 2004
- ^D
%
```

2.1 Using files

You can load SML code from a file, by applying the "`use`" function to a string that specifies the file. For example, assume that the file `foo.sml` contains the following code:

```
val x = 1+2;
val y = 17;
```

The we can load the file as follows:

```
- use "foo.sml";
[opening foo.sml]
val x = 3 : int
val y = 17 : int
val it = () : unit
-
```

Note that loading a file this way has the same effect as if you had directly entered the contents of the file at the read-eval-print loop (with the exception that the variable `it` is bound to "`()`").

3 A tour of SML

In this section, we give a brief introduction to SML. SML is a *value oriented* language, by which we mean that variables name values (not storage locations).

3.1 Basic types and values

The basic types of SML include Booleans (`bool`), integers (`int`), and strings (`string`). The two values of type `bool` are `true` and `false`, and the primary operator is `not`.

```
- not true;
val it = false : bool
-
```

SML also has conditional operators `orelse` and `andalso`, which like C's `||` and `&&` operators, short-circuit evaluation.

Integers are written in decimal notation, with negative numbers are designated by `~`. For example:

```
- 3-5;
val it = ~2 : int
-
```

3.2 Tuples

SML also supports tuples as first-class values. We use parentheses to construct tuple values:

```
- val a = ();
val a = () : unit
- val b = (1);
val b = 1 : int
- val c = (false, true);
val c = (false,true) : bool * bool
- val d = (a, b, c);
val d = ((),1,(false,true)) : unit * int * (bool * bool)
-
```

Note that the type of empty tuples is called `unit` and that, unlike the treatment of tuples in the textbook, the tuple of a single element has the same type as the element itself. Tuple types are constructed using the "`*`" operator. Note also, that tuples can contain tuples as elements (*e.g.*, the definition of `d` above). SML defines a family of projection functions (`#1`, `#2`, ...) for extracting elements of tuples. Continuing the example from above:

```
- #2 d;
val it = 1 : int
- #3 d;
val it = (false,true) : bool * bool
- #1 it;
val it = false : bool
-
```

3.3 Functions

Functions in SML are defined using the syntax

```
fun f param = expression
```

where `f` is the name of the function, `param` is the function parameter, and `expression` is the body of the function. For example, a function that doubles its argument is written as:

```
- fun twice (x : int) = x+x;
val twice = fn : int -> int
-
```

The “`->`” symbol is the function type constructor. This operator associates to the right; for example,

```
int -> bool -> unit
```

and

```
int -> (bool -> unit)
```

are the same type (a function that takes an integer and returns a function from bool to unit). Function application is by juxtaposition, although one is free to add parentheses around the argument:

```
- twice 2;
val it = 4 : int
- twice (3);
val it = 6 : int
-
```

Function application associates to the left.

Functions can take tuples as arguments, which is one way of writing functions with multiple arguments:

```
- fun max (a : int, b : int) = if (a < b) then b else a;
val max = fn : int * int -> int
-
```

While `max` looks like a function of two arguments, SML treats it as a function of one argument that happens to have tuple type:

```
- val x = (1, 2);
val x = (1,2) : int * int
- max x;
val it = 2 : int
- max (3, 4);
val it = 4 : int
-
```

We can also write functions of multiple arguments as *curried* functions:

```
- fun min (a : int) (b : int) = if (a < b) then a else b;
val min = fn : int -> int -> int
-
```

We can also write functions that take functions as arguments:

```
- fun f (g : int -> int -> int) (x : int, y : int) = g x y;
val f = fn : (int -> int -> int) -> int * int -> int
- f min (3, 4);
val it = 3 : int
-
```