UMEÅ UNIVERSITY
Department of Computing Science

7th November 2002

The Very Short Introduction To Standard ML

Written by
Marcus Bergner

Contents

1 Fundamental entities of Standard ML 2
1.1 The Moscow ML environment 2
1.2 Values and identifiers 2
1.3 Types and operators 3
1.4 Functions 3

2 Flow control, encapsulation and declarations 4
2.1 Controlling the flow 4
2.2 Encapsulation and declarations 4

3 Lists and higher order functions 5
3.1 Lists 5
3.2 Higher order functions 5

4 Creating datatypes 6
4.1 Basic creation of types 6
4.2 Using advanced declarations 6

A Libraries 7

B Further reading on SML 7

1 Fundamental entities of Standard ML

Standard ML (SML) is a functional language. A functional language is based upon the concept of functions. Unlike other programming languages, functional languages are more similar to mathematics. The entities covered in this section are the following.

- The Moscow ML environment
- Values and identifiers
- Types and operators
- Functions

1.1 The Moscow ML environment

This is the environment in which programs are executed. On the Windows NT machines it is found in the menu under *Programming*. On the Unix and Linux systems Moscow ML is located at `/pkg/mosml`. You can simply write `mosml` wherever you are in the file system to start it.

When `mosml` starts it will show you a *prompt* that looks like a dash (`-`). To exit `mosml` write `quit();`.

To do programming with `mosml` you will need to use an external editor. Usually text is written in the editor and then copied into the `mosml` window. In the following examples all lines starting with a dash (`-`) are input lines and all lines starting with a greater-than symbol (`>`) are output lines.

To get help on various topics the Moscow ML command `help` can be used. For example `help "list";` will give you lots of useful help on lists.

1.2 Values and identifiers

SML is built around values functions. We will see later that a function is infact a value. Values can be of different kinds, namely:

- Scalar
- Tuple
- Records
- List

A *scalar* value is simply a constant, i.e. `7`. A *tuple* is a composition of two or more scalars surrounded by `()`, i.e. `(3, 2.5)`. *Records* are similar to tuples but with extra identifiers bound to the values, i.e. `{name="Marcus", age=21}`.

To bind identifiers to values the reserved word `val` is used. Here follows a few examples.

```
- val number=17;
> val number = 17 : int
- val me={name="Marcus", age=21};
> val me = {age = 21, name = "Marcus"} : {age : int, name : string}
- #name me;
> val it = "Marcus" : string
```

The `it` identifier is *the last unnamed result*. Lists are one of the most important parts of SML so it will be given its own section later on. Later on we will also see a special identifier _ that is considered as a *discard-me* identifier.

1.3 Types and operators

Moscow ML is an implementation of Standard ML supporting the following standard types: `int`, `real`, `string`, `char` and `bool`. SML is a *strongly typed* language, which means that no implicit type conversion is performed, and it is for example illegal to say `2 + 2.5`, since this is an attempt to operate on an `int` and a `real`, which are distinct types.

What makes SML powerful is its ability to treat types a little arbitrarily. When declaring a function you are not forced to specify the types of the parameters or the return value. In some cases this leads to functions that can operate on arbitrary types.

For the `bool` type there are special operators, `not`, `andalso` and `orelse`. These work as you expect, with the addition that lazy evaluation is used. This means that *A* `andalso` *B* returns *false* without ever evaluating *B* if *A* is false. The same thing holds for `orelse` if *A* is true. This is the same as in C/C++ or Java.

There are of course some operators predefined by SML as well. These are fairly easy to understand. The only surprise is that unary negation uses the tilde (˜) as operator. The operators are `+`, `-`, `*`, `/`, `div`, `mod`, `^` and `@`. The last two of these denote *string concatenation* and *list concatenation*. The division operator `/` denotes floating point division. Integer division uses `div`.

1.4 Functions

In a functional language function are important! Because there is no such thing as a variable in a functional language, there are only values, the programmer has to use a different approach to problems than with imperative or object-oriented programming languages. The solution is called *recursion*. Recursion is a common thing in mathematics. The standard example is the *factorial function*, $n!$. It is defined as:

$$\begin{aligned} 0! &= 1 \\ n! &= n \times (n-1)! \end{aligned}$$

The function is separated into a *base case* and a *recursive case*. When defining a function i SML the reserved word `fun` is used. The definition of the factorial function, `fact` will look as follows.

```
fun fact 0 = 1
  | fact n = n * fact(n - 1);
```

The syntax is very similar to mathematics as you can see. We will return to the concepts of functions later on when we know more about other aspects of SML. Function calls has higher precedence than any operator so `fact n - 1` means `fact(n) - 1`. This also points out that parenthesis are not required to do function calls. They simply change the order of evaluation.

All parameters to functions are evaluated before the call, which is the same as in languages such as C/C++ or Java. SML has some very interesting features regarding parameters to functions. Consider the following two implementations and usage of the `pow` function, which is $f(n,x) = x^n$.

```
fun pow (0, _) = 1.0                fun pow 0 _ = 1.0
  | pow (n, x) = x * pow(n-1,x);      | pow n x = x * pow (n - 1) x;

> val pow = fn : int*real->real     > val pow = fn : int->real->real
- pow(3, 4.0)                       - pow 3 4.0;
> val it = 64.0 : real              > val it = 64.0 : real
```

The difference between these two implementation is that the first uses a tuple and the second uses the concept of *partial functions*. A partial function is a function that can be *partially applied*. This means that it is possible to write `pow 2`. The result of such an expression is a function of type `real -> real`. We could write `val square = pow 2;`, and at once we have our very own squaring function.

2 Flow control, encapsulation and declarations

Flow control is important is most programming languages. SML has a few ways to control the flow. First we have the *pattern matching* then the `if-then-else` expressions and at last the `case` expression. Encapsulation simplifies usage by hiding unimportant things and some declarations simplify the programming. It is all covered here.

2.1 Controlling the flow

An example of pattern matching has already been shown by the factorial function. First the parameter to the function is matched against 0. If there is a match 1 is returned. If there is no match the argument is matched against `n`, which always matches. In that case `n` is multiplied by the recursive call to `fact(n - 1)` and that result is returned. It is possible to have arbitrarily many levels of pattern matching. It is very important to order the matchings correct though. If the factorial function would have been written with the `n`-matching first, the function would never terminate.

The second form of flow control is the `if-then-else` expression. It can be used to write an alternative factorial function

```
fun fact n = if n = 0 then
                1
             else
                n * fact(n - 1);
```

The third, and last, form of flow control is the `case` expression. It is similar to the pattern matching and the factorial function will look like this with the use of a `case` expression.

```
fun fact n = case n of
                 0   =>   1
               | n   =>   n * fact(n - 1);
```

2.2 Encapsulation and declarations

There are many ways to do encapsulation in SML. The two most common ones are the `let` and `local` statements. The `let` statement allows a *declaration to be used in an expression* while the `local` allows a *declaration to be used in a declaration*. Consider the following function for solving equations on the form $ax^2 + bx + c = 0$. The example also illustrates the use of an *exception* to indicate that something has gone wrong.

```
exception Solve;
local
   fun disc (a, b, c) = b * b - 4.0 * a * c;
in
   fun solve (a, b, c) = let
                            val d = disc(a, b, c)
                         in
                            if d < 0.0 orelse a = 0.0 then
                               raise Solve
                            else
                               ((~b + Math.sqrt d) / (2.0 * a)
                               ,(~b - Math.sqrt d) / (2.0 * a))
                         end
end;
```

This means that the function `disc` is hidden from the user, and the value `d` is defined only within a restricted context between the second `in` and the first `end`.

3 Lists and higher order functions

Lists are one of the most fundamental parts of any functional language. Here they will be covered in a fairly brief way, so for a more detailed discussion on lists you should look into a good book on SML. The concept of higher order function is also one of the most powerful features of functional programming. It is also a concept that should be covered more in detail than is done here, but this is a *short* introduction.

3.1 Lists

Lists are constructed using *list notation* i.e. `[2, 4, 3]` or by using the *cons* operator `::` i.e. `2::4::3::[]`. This can be written as `2::[4, 3]` or `2::4::[3]` as well. The empty list `[]` has an alternative name `nil`.

Functions that operate on lists are usually divided into two or more parts by pattern matching. To calculate the length of *any* list the following function can be used.

```
fun length []      = 0
  | length (x::xs) = 1 + length xs;
```

This also illustrates the concept of *polymorphism*. This means that arbitrary types can be used in the lists, since the type of the elements has no meaning in the calculation.

One very useful function on lists is the `filter` function. It takes a function and a list as parameters. The function is of type `'a -> bool` which means that it is any function with one parameter returning a boolean value. The list has type `'a list`, which means that the elements are of the same type as the parameter to the function. The return value from `filter` is a list of all elements from the original list that makes the function parameter return *true*. Here follows an implementation and an example on how to use it.

```
fun filter p []      = []
  | filter p (x::xs) = if p x then
                          x::filter p xs
                       else
                          filter p xs;

fun even x = (x mod 2 = 0);

- filter even [1, 2, 3, 4, 5, 6, 7, 8, 9];
> val it = [2, 4, 6, 8] : int list
```

As you can see this function can be used to create powerful list processing. More on powerful processing in the following section. For more information on lists try `help "list"`.

3.2 Higher order functions

Higher order functions means *functions taking function(s) as parameter(s)*. An example on this is the `filter` function. The `filter` function is also a partial function which means that we could do nice things like this.

```
- val evens = filter even;
> val evens = fn : int list -> int list
- evens [1, 2, 3, 4, 5];
> val it = [2, 4] : int list
```

The function `evens` is a function which is constructed by specifying that the first argument to `filter` has to be the function `even`. Given this information SML knows that the function is working with lists of integers.

4 Creating datatypes

There are several ways to create datatypes in SML. In this section the most important ones will be covered. The main part of the section is dedicated to structures and the `abstype` construct.

4.1 Basic creation of types

The most primitive way to create a type is to use the `type` declaration. It only *puts a new name to an existing type*, similar to the `typedef` statement in C/C++. This makes it possible to do things like these.

```
type pipe = int * int;
type mathfunction = real -> real;
type ’a pair = ’a * ’a;
type intpair = int pair;
```

The problem with these declarations is that they *don’t create distinct types*. This means that the types `intpair` and `pipe` are the same type!

To solve the problems of distinction between types created with the `type` declaration SML provides the programmer with the `datatype` declaration. It *creates distinct types* and has some more powerful features than the `type` declaration. This type of declaration uses a tag to reference a value within the datatype. This tag is used heavily in pattern matching as we will soon see. For example the following declarations can be done using the `datatype` declaration.

```
datatype descriptor = STDIN
                    | STDOUT
                    | STDERR
                    | OTHER of int;
datatype ’a tree = EMPTY
                 | NODE of ’a * ’a tree * ’a tree;
```

To see how these declarations can be used an implementation of the *inorder traversal algorithm* follows here. It uses the declaration of a `tree` above. The return value is a list of all elements of the tree. This example also illustrates the `@` (list-append) operator. It takes two lists and return a list consisting of the lists concatenated together.

```
fun inorder EMPTY = []
  | inorder (NODE (x, left, right)) = inorder left @ [x] @ inorder right;
```

4.2 Using advanced declarations

The `datatype` does some nice things, but it does no really solve the fact that the types `pipe` and `intpair` where the same. We could create a type such as `datatype pipe = PIPE of int * int` and `datatype intpair = PAIR of int * int`, that is making the tags different.

Often, one wants a set of data to be encapsulated with only a few functions to access it. Typical examples are *abstract data types* and a very familiar one is a *stack*. A stack is a well known datatype consisting of a few well defined functions and a set of data on which the functions operate. The implementation of a generic stack could look like this.

```
exception EmptyStack;
abstype ’a Stack = STACK of ’a list
with
   val newStack          = STACK([])
   fun push (STACK s) x   = STACK(x::s)
   fun pop (STACK [])     = raise EmptyStack
     | pop (STACK (_::xs)) = STACK(xs)
   fun top (STACK [])     = raise EmptyStack
     | top (STACK (x::_)) = x
```

```
    fun fromList xs         = STACK(xs)
    fun toList (STACK xs)   = xs
  end;
```

This demonstrates one of the nicer aspects of SML, namely the fact that we have created a generic and dynamic stack with only 12 lines of code. This is quite tricky to accomplish in languages such as C++ or Java.

A Libraries

There are many libraries available with Moscow ML. Some of them requires special command line arguments to be loaded. This is accomplished using the `-P` flag to the `mosml` executable. Here is a list of some of the available libraries with short descriptions. For a complete list of libraries use `help "lib";`. Libraries are loaded into a program by writing `load "`*library name*`";`.

Library	**Description**	**Library**	**Description**
BinIO	Binary I/O functions	Path	Manipulating path names
Binarymap	Dictionary	Process	Process management
Binaryset	Set	Random	Random numbers
CommandLine	Command line arguments	Regex	Regular expressions
Date	Date management	Signal	System signals
FileSys	File system interaction	Socket	Socket functions
Help	Help utilities	TextIO	Text I/O functions
List	List functions	Time	Time management
Math	Mathematical functions	Timer	Measuring time
Mosmlcgi	CGI support	Unix	IPC with Unix processes

B Further reading on SML

There are several books available on SML and functional programming. Here is a few of them. You will probably need one of them to look into the details of some things, but hopefully this document has given you a little idea on what to look for.

Michael R. Hansen, Hans Rischel *Introduction to Programming using SML* (1999) Addison-Wesley

Åke Wikström *Functional Programming using Standard ML* (1987) Prentice Hall

Greg Michaelson *Elementary Standard ML* (1996) UCL Press

Moscow ML documentation, see `/pkg/mosml/doc`

Moscow ML libraries, see `help "lib";` and `help "`*library name*`";` or if that doesn't work `/pkg/mosml/2.0/lib/*.sig`