

A Brief Sketch of the ML Programming Language

Comp 131

Dept. of Mathematics and Computer Science
Wesleyan University

Fall, 2008

1 Introduction

SML is an *interpreted, typed, functional, pattern-based, polymorphic* programming language. We will devote some time to explaining each of these concepts. Let's start with a first approximation to what an *interpreted programming language* is. It means a collection of symbolic statements (that can be formed using the symbols on a computer keyboard) that are typed and directly *evaluated* or somehow processed by a machine during a *session* or dialogue with a program called the **SML interpreter**.

It will be useful to think of **SML** as composed of two kinds of such statements: *expressions* or *terms*, and *definitions*. In all cases, when **SML** reads a statement, it *returns a value to the programmer*, but in the case of definitions, **SML** associates values to certain symbols (called identifiers) and remembers them for future use.

1.1 Expressions

Let's look at some examples. We will spend some time first just looking at basic *expressions* and the values they return when a machine running **SML** takes them as inputs.

To start up an **SML** session one clicks on an appropriate icon or types the command `sml` in some command window. Then, **SML** starts up the *interpreter* and displays a *prompt*, a dash, that indicates it is waiting for the programmer to input an expression or a definition.

```
Standard ML of New Jersey, Version 110.0.7, September 28, 2000 [CM; autoload enabled]
```

```
-
```

If the programmer does so, the interpreter responds with a value, and also a type.

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

Remember, what follows the prompt, i.e., the *dash* -, is what the programmer (you) has typed. The next line is what the **SML** interpreter has returned.

The interpreter has returned the value 5, of type **int**. This stands for the integers, which are whole numbers (positive, negative or 0). We will have more to say about types later.

The last value the interpreter has handled is called “it” by default. Let’s look at some more examples.

```
- 50 - 20;
val it = 30 : int
- 20*3;
val it = 60 : int
```

Now we’ll try expressions of a different type: the type of real numbers, which means floating point decimals.

```
- 2.0+3.0;
val it = 5.0 : real
- 50.0/2.0;
val it = 25.0 : real
- 50.0/3.0;
val it = 16.6666666667 : real
-
```

SML doesn’t always like what you give it.

```
- 50/2;
stdIn:19.3 Error: overloaded variable not defined at type
  symbol: /
  type: int
```

It turns out that / can only be carried out on reals, not ints. There is an integer division operation, called *div*, which can only be used with ints and will only return the integer quotient.

```
- 50 div 2;
val it = 25 : int
- 50 div 3;
val it = 16 : int
```

Expressions don’t have to be integers or real numbers. They can be words in quotes, known as strings.

```
- "Hello there";
val it = "Hello there" : string
```

There are operations that can be performed on strings, such as *concatenation* (gluing together) denoted by ^

```
- "Hi"^" there";
val it = "Hi there" : string
```

Expressions can contain various types of structures used as “containers” of data. You will learn about two in this course: *lists* and *tuples*.

Some examples of lists:

```
- [1,2,5,7];
val it = [1,2,5,7] : int list
- ["Hi","to","everyone"];
val it = ["Hi","to","everyone"] : string list
```

Lists must be homogeneous. That means they may not be of mixed type. If you try to make a list with both ints and strings in it you will get a noisy complaint from **SML**, that will be understood later in the course.

```
- [1,"hi",2];
stdIn:27.1-27.11 Error: operator and operand don't agree [literal]
operator domain: string * string list
operand:          string * int list
in expression:
  "hi" :: 2 :: nil
```

Lists of a given type can be of any length. Including length 0, for the empty list (which is of undetermined type).

```
- [];
val it = [] : 'a list
```

There are many operations that one can perform on lists, such as computing length, list-concatenation (gluing), or adding new members from the left hand side.

```
- length ["this","is","a","list"];
val it = 4 : int

- [1,2]@[3];
val it = [1,2,3] : int list

- "this"::["is","a","list"];
val it = ["this","is","a","list"] : string list
-
```

Tuples (of a given type) are collections of fixed length (unlike lists) which can have slots reserved for different types.

```
- (2,"hi");
val it = (2,"hi") : int * string
- (2,"hi",["this","is"],3,3.0);
val it = (2,"hi",["this","is"],3,3.0) : int * string * string list * int * real
-
```

1.2 Definitions

We could just sit here all day, feeding **SML** expressions to evaluate, like a calculator.

```
- 2.34 * (3.14159 + 2.0);
val it = 12.0313206 : real
- Math.sqrt(16.0);
val it = 4.0 : real
- Math.sqrt(2.0);
val it = 1.41421356237 : real
```

but maybe we expected more out of a programming language, such as the possibility of defining complex procedures, and later executing them. In **SML**, this is done by making *definitions*, in one of several ways.

1.2.1 val

Firstly, one can associate a value with a variable and keep it for later use, with the keyword **val**.

```
- val x = 32;
val x = 32 : int
- x - 30;
val it = 2 : int
```

The definition of *x* effected above is called *binding x to the value 32*. Before carrying it out, *x* was an unbound variable. After carrying it out, *x* has become a bound variable, bound to the integer value 32.

SML does not like unbound variables.

```
- 2 + y;
stdIn:34.5 Error: unbound variable or constructor: y
```

1.2.2 fun

We can also define new functions (operations), using the keyword **fun** (rather than **val**).

```
- fun foo x = x + 2*x*x;
val f = fn : int -> int
```

This defines a new operation on integers, called *foo*. Now, in addition to adding, squaring, subtracting, negating we can also try *foo*-ing an integer, which means adding twice its square to itself.

```
- foo 3;
val it = 21 : int
```

You can define operations on (two) strings:

```
- fun gluedot x y = x^"."^y;
val gluedot = fn : string -> string -> string
```

and try them out on inputs:

```
- gluedot "Hi" "there";
val it = "Hi.there" : string
```

```
- gluedot "." ".";
val it = "...": string
```

You can mix this with variable definitions:

```
-val dot = ".";
val dot = "." : string
- gluedot dot dot;
val it = "...": string
```

1.3 Conditionals, Recursion, Cases

1.3.1 Conditionals

There are several constructs available to programmers in **SML** to define functions. One of them is the *conditional* `if...then...else`.

```
- fun classify x = if x < 0 then "negative" else "nonnegative";
val classify = fn : int -> string
```

which behaves as expected

```
- classify 70;
val it = "nonnegative" : string
- classify ~4;
val it = "negative" : string
```

1.4 Recursion

As we have seen in class, when we do proofs by induction, we *assume that the statement of a theorem is true for n* and then show it is true for $n + 1$.

We do something similar when we *define functions inductively*, also called *definitions by recursion*. Consider the following ML definition of the factorial function which returns, on input x , the value $x! = x \cdot (x - 1) \cdots 3 \cdot 2 \cdot 1$. This function can be defined mathematically by

$$0! = 1 \quad x! = x * (x - 1)!$$

Here's the code in ML

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

Using *patterns* we can also define it as follows:

```
fun fact1 0 = 1
  | fact1 x = x*(fact1 (x-1));
```

Notice that *to define the function for input n we use the fact that it is already defined for $n - 1$.*

In a similar vein, here's the definition of a function that computes, on input n , the sum of all natural numbers from 1 to n .

```
fun sumto 0 = 0
  | sumto n = n+(sumto (n-1));
```

Exercise 1 Define a function `sumOdd:int-->int` which adds up the first n odd numbers.

Exercise 2 Define the fibonacci function in ML which must satisfy:

$$\begin{aligned} fib(0) &= 1 \\ fib(1) &= 1 \\ fib(x) &= fib(x - 1) + fib(x - 2) \end{aligned}$$

2 Datatypes

In ML we can define new types of data via the `datatype` command. We give an example of the definition of the data type of propositions. It will be discussed in lab.

```
datatype prop = True|False| Var of string| And of prop*prop| Or of prop*prop|
  Imp of prop*prop| Not of prop;
```