CS411 Recitation 1
Introduction to Standard ML

R. Pucella

5 Feb 2001

For the programming assignments in this course, we will be using the programming language Standard ML. If you took CS312 last semester (Fall 2000), you should already be familiar with the language. Otherwise, we will expect you to pick it up in the coming weeks. We will only be using the basic features of the language, so this should not be a problem. The course web page has references to tutorial material on Standard ML. You should read them. We will be using a particular implementation of Standard ML, called *Standard ML of New Jersey* (SML/NJ). Again, the web page contains links to the software and tutorial information.

Before we start our overview of the language, a bit of propaganda is in order. Why are we using ML instead of a language that you are more likely to be familiar with? Although one is tempted to answer that learning new languages is a great way to expand your horizons and learn new ways of thinking about problems, the real reason is more pragmatic. In this course, we will discuss formal semantics of programming languages. Often times, the programming assignments will take the form of interpreters for various toy languages that we will use to experiment with features we will have seen in class. The fact is that ML was originally designed to be a language in which it would be easy to do just that sort of thing: ML stands for *Meta-Language*, which is just Greek shorthand for a language used to talk about other languages.

On to the language. ML is fundamentally an interactive language: the user is faced with a prompt, types an ML expression, the ML compiler evaluates the expression, and spits out an answer. Sometimes, there are side-effects to evaluating the expression, such as printing something on the screen, or popping up a user interface. But the basic mechanism is the same. Expressions can be simple (a line of code), or huge (a 100000+ lines compiler). ML features are meant to ease the burden of managing such large expressions (a.k.a. programs).

1 Basic expressions and types

The language is defined by specifying what expressions can be written and how they evaluate to produce a result. The simplest kind of expression is just a value, such as *0*, *1*, *2*, *∼1*, *true*, *false*, *"hi"*. An important point is that every value in ML has a unique *type*. Thus, *0*, *1*, *∼1* have type *int* (for integers), *true* and *false* have type *bool* (for booleans), *"hi"* has type *string* (for strings). There is a special value *()*, which is the only value of type *unit*. Values simply evaluate to themselves. You can try it out from the SML/NJ compiler prompt:

```
- 1;
val it = 1 : int
- true;
val it = true : bool
```

Notice that the compiler returns both the evaluated value and the type of the result. Also, notice the use of the semicolon: *a semicolon is used to indicate to the compiler that it should evaluate the result.* It is not formally part of the expression to evaluate.

More complex expressions can be written. By extension, every expression gets a type, which is the type of the result. The expression *if* $\langle e_1 \rangle$ *then* $\langle e_2 \rangle$ *else* $\langle e_3 \rangle$ first evaluates the expression $\langle e_1 \rangle$; if it evaluates to *true*, then $\langle e_2 \rangle$ is evaluated as the result, otherwise $\langle e_3 \rangle$ is evaluated as the result. Note that $\langle e_1 \rangle$ should evaluate to a boolean values, that is should have type *bool*. Similarly, both $\langle e_2 \rangle$ and $\langle e_3 \rangle$ should have the same type, and the type of the whole *if* expression has the type of $\langle e_2 \rangle$ and $\langle e_3 \rangle$. A *type error* occurs if these constraints are not respected. For example:

```
- if 1 then "a" else "b";
stdIn:20.1-20.23 Error: case object and rules don’t agree [literal]
  rule domain: bool
  object: int
  in expression:
    (case 1
      of true => "a"
       | false => "b")
```

This indicates (if you decode the error message...) that the compile was expecting a boolean, but found an integer.

Operations are available on booleans, integers and strings. The operation *not* takes a boolean and returns the negated boolean. The operations +, -, *, $>$, $>=$, $<$, $<=$, $=$ are the standard infix operations on integers. Notice that the arithmetic operations return integers, while the comparison operations return booleans. For instance:

```
- 42 + 42;
val it = 84 : int
- 42 > 42;
val it = false : bool
```

The operations *size* and ˆ are available for strings, and respectively return the size of a string and the concatenation of two strings.

```
- size "foo";
val it = 3 : int
- "foo"^"bar";
val it = "foobar" : string
```

2 Declarations

Now that we know how to evaluate simple expressions, it makes sense to assign name to the results, so they are easily reused. A *declaration* associates (or binds) a name with a value. A global declaration is made at the prompt, and is valid for as long as the compiler is running, or until another global declaration with the same name is made. A global declaration uses the keyword *val*:

```
- val a = 42 * 42;
val a = 1764 : int
- a + 42;
val it = 1806 : int
```

Sometimes, you need a binding to hold only for the duration of the evaluation of an expression. You can use a local binding for that (these are akin to local variables in traditional language). The syntax for a local declaration is *let* $\langle d \rangle$ *in* $\langle e_1 \rangle$ *end*. This is an expression. To evaluate this expression, you first evaluate the declaration, which is of the form *val* $\langle id \rangle$ = $\langle e_2 \rangle$, bind the result of evaluating $\langle e_2 \rangle$ to $\langle id \rangle$, and then evaluate $\langle e_1 \rangle$. The binding for $\langle id \rangle$ is forgotten after the result is returned.

```
- let val n = size "this is a string" in n + 10 end;
val it = 26 : int
```

Another kind of declaration is a *function declaration*. A function is a function like in any other language. A function declaration has the form: *fun f (x1:t1,x2:t2,...):t* = $\langle e \rangle$, where *f* is the name of the function, *x1*, *x2*, ... are the parameters to the function, *t1*, *t2*, ... are the types of the parameters, *t* is the type of the result, and the expression $\langle e \rangle$ is the body of the function. For example, consider the standard factorial function (which is recursive):

```
- fun fact (n:int):int = if (n=0) then 1 else n * fact (n-1);
val fact = fn : int -> int
```

Notice the type of the result, which states that *fact* is a function of type $int - > int$, a function expecting an integer and returning an integer. To use a function, you need to apply it to arguments. A function application (or function call) has the form *f* ($\langle e_1 \rangle$,$\langle e_2 \rangle$,...). The arguments are first evaluated to values, and then the function *f* is called with those values.

```
- fact (10);
val it = 3628800 : int
- fact (5+5);
val it = 3628800 : int
```

Type-checking ensures that the type of the arguments corresponding to the types expected by the function (given in the declaration), and that the result of the function is used appropriately. Can you spot the type errors in the following fragments?

```
- fact ("hi");
stdIn:40.1-40.12 Error: operator and operand don't agree [tycon mismatch]
  operator domain: int
  operand:         string
  in expression:
    fact "hi"
- if (fact (10)) then 0 else 1;
stdIn:1.1-38.26 Error: case object and rules don't agree [tycon mismatch]
  rule domain: bool
  object: int
  in expression:
    (case (fact 10)
      of true => 0
       | false => 1)
```

Note that since function declarations are declarations, they can appear in the declaration part of a *let* expressions.

3 Datatypes

ML allows you to define your own types. Such types are called *datatypes*, and you define them by specifying how to construct values of that type. For example, suppose we wanted to define a type for Peano integers. Recall that a Peano integer is either a zero or a successor of a Peano integer. We could define such a type as follows:

```
- datatype peano = Z  |  S of peano;
datatype peano = S of peano | Z
```

(The compiler simply echoes back the definition, modulo some reorder). This defines a new type *peano*, and two *constructors*, *Z* and *S*, to construct values of that type. The constructor *Z* is a nullary constructor, meaning it doesn't expect an argument. We can verify that *Z* is a value of the appropriate type:

```
- Z;
val it = Z : peano
```

The constructor *S* is a constructor of one argument, and construct a *peano* value given another *peano* value. For instance:

```
- S (Z);
val it = S Z : peano
- S (S (Z));
val it = S (S Z) : peano
```

So we can construct values of our new type. How do we use them however? To be able to use a value of type *peano*, we need to be able to deconstruct it. A *case* expression is used to determine which constructor was used to build a value, and proceed accordingly. Here is a simple function that takes a *peano* value and returns a string stating what kind of value it is:

```
- fun whatKind (p:peano):string =
    case (p)
      of Z => "zero"
       | S (p’) => "successor";
val whatKind = fn : peano -> string
- whatKind (Z);
val it = "zero" : string
- whatKind (S (S (Z)));
val it = "successor" : string
```

Notice that each branch of a *case* expression contains one of the constructors of the datatype. For constructors with an argument, we can specify an identifier which will get bound to the argument of the constructor. In the above example, that identifier (*p’*) is not used. In the following function, which takes a *peano* value and converts it to an integer, it is:

```
- fun toInt (p:peano):int =
    case (p)
      of Z => 0
       | S (p’) => 1 + toInt (p’);
val toInt = fn : peano -> int
- toInt (Z);
val it = 0 : int
- toInt (S (S (S (S (Z)))));
val it = 4 : int
```

We will use datatypes extensively in this course, to model abstract syntax.

4 Modules

We will not really use the prompt to define our programs. Instead, we will put our declarations in files, and load them appropriately. A good programming principle is to package related things together. ML provides a state-of-the-art module system to manage such packaging. A module (also called a structure) wraps together related declarations. A module is declared as follows:

```
structure Foo = struct
  val bar = 42
  fun talkToMe ():int = bar
end
```

This declares a simple module named *Foo*, with declarations *bar* and *talkToMe*. To access the values or functions in that module, we use the dot-notation:

```
- Foo.bar;
val it = 42 : int
- Foo.talkToMe ();
val it = 42 : int
```

It is possible to associate a *signature* with a module, which specify which declarations should be *exported* from a module. For example, the signature

```
signature FOO = sig
   val talkToMe : unit -> int
end
```

can be ascribed to the module *Foo* as follows:

```
structure Foo2 : FOO = struct
  val bar = 42
  fun talkToMe ():int = bar
end
```

The net result is that *Foo2* only exports *talkToMe* to the outside world. The value *bar* is now only available inside the module:

```
- Foo2.talkToMe ();
val it = 42 : int
- Foo2.bar;
stdIn:72.1-72.9 Error: unbound variable or constructor: bar in
  path Foo2.bar
```

We will use modules extensively to structure our code.[1] SML/NJ provides a *Compilation Manager* (CM) to manage code written using modules. Information will be provided in programming assignments, and pointers to documentation are available from the course web page.

5 And the rest...

Standard ML provides a many more features than has been described today. The goal today was to review the basic, and to provide a feel for the language. ML provides tuples, records, lists, supports parametric polymorphism, higher-order functions, and a host of buzzwords that you will be learning about in this course.

[1]Pun fully intended.