CMCS 312: Programming Languages

An Overview of Standard ML

Umut A. Acar **Sept. 29, 2006**

Contents

1 What is SML

Standard ML is a strongly typed functional language.

Strongly typed: Every value, expression in the language has a *type* (int, real, bool *etc.*). The compiler rejects a program that does not conform to the type system of the language.

Functional: Each expression is either a value, or a function. Functions are *first-class* values—they can be bound a variable, passed as arguments, returned from functions, stored in data structures, *etc.*.

In this class, we will use the SML/NJ compiler. SML/NJ is not a high-performance compiler (it does not generate very fast code) but it has a very nice programming environment. MLton is perhaps the fastest compiler for SML. I program with SML/NJ but use MLton for performance evaluation.

2 Integers, reals, booleans

Built-in data types has the following types: `int, real, bool, 'a option, etc.` . Standard ML calls floating points numbers as `real`s. This is a misnomer; they are really finite precision numbers. Here are some examples.

```
[sinop:~] umut% sml
Standard ML of New Jersey v110.54 [built: Thu May 19 10:44:48 2005]
(* Integers *)
- 1;
val it = 1 : int
- 2;
val it = 2 : int
- 1+2;
val it = 3 : int

(* Reals *)
- 1.0;
val it = 1.0 : real
- 2.0;
val it = 2.0 : real
- 1.0+2.0;
val it = 3.0 : real

(* Booleans *)
- false;
val it = false : bool
- true;
val it = true : bool
- if true then 1 else 2;
```

```
val it = 2 : int
-
```

3 Option type

Standard ML has built in support for option types. The type `int option` has two possible values, `NONE` or `SOME i` where `i` some integer.

```
- NONE;
val it = NONE : 'a option
- SOME 1;
val it = SOME 1 : int option
- SOME 1.0;
val it = SOME 1.0 : real option
- SOME false;
val it = SOME false : bool option
```

4 Tuples

You can define fixed-ary tuples as usual.

```
- (1, 2);
val it = (1,2) : int * int
- (1.0, 2.0);
val it = (1.0,2.0) : real * real
- (1, 2.0, true, NONE);
val it = (1,2.0,true,NONE) : int * real * bool * 'a option
```

Projection out of tuples is easy but it is discouraged because pattern matching (see below) does the same thing with better compile-time safety checks.

```
- #1 (1, 2.0);
val it = 1 : int
- #2 (1, 2.0);
val it = 2.0 : real
```

5 Records

Records are a commonly used data type for grouping together other data. Record fields can contain anything including functions and other records. Projection syntax is similar to that of tuple projection except we index by field name instead of tuple position.

```
- val r = { a = 1.0, b = true, c = {a = 3.0}, d = fn x => x};
val r = {a=1.0,b=true,c={a=3.0},d=fn}
  : {a:real, b:bool, c:{a:real}, d:'a -> 'a}
- #a (#c r);
val it = 3.0 : real
- (#d r) 1;
val it = 1 : int
```

6 Data Types

You can define your own data types.

```
- datatype 'a myList = Nil | Cons of ('a * 'a myList);
datatype 'a myList = Cons of 'a * 'a myList | Nil

- Nil;
val it = Nil : 'a myList
- Cons (1, Nil);
val it = Cons (1,Nil) : int myList
- Cons (1, Cons (2, Nil));
val it = Cons (1,Cons (2,Nil)) : int myList
- Cons (1.0, Cons (2.0, Nil));
val it = Cons (1.0,Cons (2.0,Nil)) : real myList
- Cons (1.0, Cons (false, Nil));
stdIn:51.1-51.30 Error: operator and operand don't agree [tycon mismatch]
  operator domain: real * real myList
  operand:         real * bool myList
  in expression:
    Cons (1.0,Cons (false,Nil))
```

The option that we have seen is just a (built-in) data type.
You can do pattern matching on data type using `case`

```
- val l = Cons (1, Cons (2, Cons (3, Nil)));
val l = Cons (1,Cons (2,Cons #)) : int myList

case l of
=   Nil => true
= | Cons (h,t) => false;
val it = false : bool
```

These algebraic datatypes work especially well with records. Use them to "label" records. During pattern matching, since the label will fully specify a particular record type, we can elide mention of all uninteresting fields.

```
- datatype arec = A of {a:real, b:bool};
datatype arec = A of {a:real, b:bool}
```

```
- fun f(A{b, ...}) = b;
val f = fn : arec -> bool
```

7 Functions

Functions in SML have type `'a -> 'b`. Anonymous functions are written with the arrow notation `fn x => x+1`. Named function are written with a name `fun inc x = x + 1`

```
- fn x => x+1;
val it = fn : int -> int
- fun id x = x+1;
val id = fn : int -> int
```

In SML, functions can be passed multiple arguments by *currying* like this

```
- fun myPlus i j = i+j;
val myPlus = fn : int -> int -> int
- myPlus 2 5;
val it = 7 : int
(* Partial evaluation *)
- val myPlus2 = myPlus 2;
val myPlus2 = fn : int -> int
- myPlus2 5;
val it = 7 : int
- myPlus2 10;
val it = 12 : int
```

To disambiguate the type of a curried function, how do you parenthesize? To figure out what SML means by `int -> int -> int`, consider the application notation `myPlus 2 5`. How would you parenthesize `myPlus 2 5`? SML understands the expression to mean `(myPlus 2) 5`. If you actually wanted the other parenthesization, you have to explicitly parenthesize. Application is left-associative. In order to be faithful to the semantics of application, the arrow type must be right-associative. Do you see why? Thus, `int -> int -> int` is understood to be `int -> (int -> int)`.

Here is how actually myPlus is defined:

```
- val myPlus = fn i => fn j => i + j;
val myPlus = fn : int -> int -> int
```

It is a function that takes a value and returns another function that takes value and adds it to value `i`; As far as the inner function is concerned `i` is some global, or *free variable* that is defined outside. In this sense, this function is *open*.

The language ensures, however, that whenever an instance of the inner function is defined `i` is also defined in the *environment* or *context*. The inner function

constructed as such is a bit more than a function, it is a *closure* that contains a function and a variable (`i`). The term *closure* refers to the property of being "closed", all undefined variables of the function (*i.e.*, `i`) are actually provided.

For example, the term `myPlus 2` evaluates to a closure that can informally be written as follows `fn j => i+j, where i=2`.

8 Using Files

When you start SML you are at the *top level*. The top level allows you type code, open files, and invoke CM, the Compilation Manager of SML/NJ, *etc.*. So far, we used the top level to evaluate small pieces of code. When you want to write more than a few lines, however, you want to use files.

Let's write a function that computes the length of a `myList` in a file called "MyList.sml"

```
(* MyList.sml *)
fun myLength l =
  case l of
    Nil => 0
  | Cons(h,t) => 1 + myLength t
```

We can now "use" this file at the top level.

```
- use "MyList.sml";
[opening MyList.sml]
val myLength = fn : 'a myList -> int
val it = () : unit

(* The prompt tells us that myLength is now incorporated into the top level *)
- val l = Cons (1, Cons (2, Cons (3, Nil)));
val l = Cons (1,Cons (2,Cons #)) : int myList
- myLength l;
val it = 3 : int
-val l = Cons (1, Cons (2, Cons (3, Cons (4,Nil))));
val l = Cons (1,Cons (2,Cons #)) : int myList
- myLength l;
val it = 4 : int
- myLength Nil;
val it = 0 : int
```

Note the type of `myLength: 'a myList -> int`. Since the input is `'a myList`, it can work on a list consisting of any type of elements.

```
- val l = Cons (1.0, Cons (2.0, Cons (3.0, Cons (4.0,Nil))));
val l = Cons (1.0,Cons (2.0,Cons #)) : real myList
- myLength l;
myLength l;
val it = 4 : int
```

The file MyList.sml does not contain the type definition for MyList. We have defined this type at the top level. The problem with this is that when we exist the system, the definition is gone. If we want to use MyList again, we must also define the type `'a myList` explicitly. So let's put this definition into the file.

Here is how MyList.sml looks like.

```
datatype 'a myList = Nil | Cons of ('a * 'a myList)

fun myLength l =
  case l of
    Nil => 0
  | Cons(h,t) => 1 + myLength t
```

Let's extend MyList.sml with a function that adds a given value say x to all elements of a list.

```
fun myAdd (x,l) =
  case l of
    Nil => Nil
  | Cons(h,t) => Cons(h+x, myAdd (x,t))
```

Here is what we get when we use the file. Note the type of myAdd. The input type is now restricted to integer lists.

```
- use "MyList.sml";
[opening MyList.sml]
datatype 'a myList = Cons of 'a * 'a myList | Nil
val myLength = fn : 'a myList -> int
val myAdd = fn : int * int myList -> int myList
val it = () : unit
-
```

Can we generalize myAdd to a function, say myMap, that applies a given function to every element in the list, and returns a list of resulting values?

```
fun myMap (f,l) =
  case l of
    Nil => Nil
  | Cons(h,t) => Cons(f h, myMap (f,t))
```

Note that the function `f` is just like any other value. This is one instance of what it means to being "first-class value" (*a.k.a.*, first-class citizen).

Can we know use myMap to write myAdd more succintly. How about something like this:

```
fun myAdd' (x,l) = myMap ((fn (h,x) => h+x), l)
```

It is sometimes more preferable to curry some arguments. For example, in `myMap`, we can curry the function `f`.

```
fun myMap f l =
  case l of
    Nil => Nil
  | Cons(h,t) => Cons(f h, myMap f t)
```

We can now define myAdd' as a value.

```
- use "MyList.sml";
[opening MyList.sml]
datatype 'a myList = Cons of 'a * 'a myList | Nil
val myLength = fn : 'a myList -> int
val myMap = fn : ('a -> 'b) -> 'a myList -> 'b myList
val it = () : unit

- val myAdd' = myMap (fn (i,h) => i+h);
val myAdd' = fn : (int * int) myList -> int myList
```

Since this is so easy, we just throw away myAdd and myAdd' from MyList.sml. You probably realized, we are constructing some kind of list library.

Suppose now we want to add up all the elements in a list? How can we write a function to do this?

Can we generalize this function to apply an arbitrary binary operator? What will we need to start with?

1. the operator or the function, and
2. an initial value.

Here is the code for this.

```
fun myFold f i l =
  case l of
      Nil => i
    | Cons(h,t) => myFold f (f(i,h)) t
```

We can use myFold to compute the sum of the values in a list.

```
- val l = Cons (1, Cons (2, Cons (3, Cons (4, Nil))));
val l = Cons (1,Cons (2,Cons #)) : int myList
- myFold (fn (i,h) => i+h) 0 l;
val it = 10 : int
```

9 Common Errors

9.1 Capturing during Nested Pattern Matching or Exception Handlers

ML is not sensitive to whitespace. Even though a nested pattern matching or exception handle looks offset, it might not be. The outer pattern matching cases following a nested pattern matching or exception handle may easily be "captured" by the nested cases. Add parentheses to disambiguate.

```
- fun f x = case x of [] => NONE
    | [y] => List.nth(y,1)
     handle Subscript => NONE
          | _ => SOME ~1
    | y::ys => f ys;
[autoloading]
[autoloading done]
stdIn:4.20-4.89 Error: types of rules don't agree [tycon mismatch]
  earlier rule(s): exn -> int option
  this rule: 'Z list -> 'Y
  in rule:
    y :: ys => f ys

- fun f x = case x of [] => NONE
       | [y] => (List.nth(y,1)
      handle Subscript => NONE
           | _ => SOME ~1)
    | y::ys => f ys;
val f = fn : int option list list -> int option
```

9.2 Curried versus Uncurried Function Application

Types are quite helpful to distinguish between functions that take tupled arguments or multiple arguments. Double-check what kind of arguments your function expects if you get something similar to the following errors:

```
- fun f a b = a + b;
val f = fn : int -> int -> int
- f (5,2);
stdIn:3.1-3.8 Error: operator and operand don't agree [tycon mismatch]
  operator domain: int
  operand:         int * int
  in expression:
    f (5,2)

- fun f (a,b) = a + b;
val f = fn : int * int -> int
```

```
- f 5 2;
stdIn:4.1-4.6 Error: operator and operand don't agree [literal]
  operator domain: int * int
  operand:         int
  in expression:
    f 5
```

9.3 Value Restriction

This topic is somewhat more advanced than what we will be covering in the course. However, if you ever encounter this error message in your code, you can add type annotations to make this error go away.

```
- val f = (fn x => x) (fn x => x);
stdIn:17.5-17.32 Warning: type vars not generalized because of
   value restriction are instantiated to dummy types (X1,X2,...)
val f = fn : ?.X1 -> ?.X1
- val f = (fn x : (int -> int) => x) (fn x : int => x);
val f = fn : int -> int
```

10 Tips for Bug Hunting

1. Use type annotations in critical locations to zero in on where type inference and your expectations do not match
2. Use exceptions to zero in on runtime errors, to get unimplemented parts to type check so that you can test the rest of the program

11 The Module System

Our file MyList.sml now looks like this. So far, we accessed this code by using the "use" function at the top level. The "use" function incorporates the contents of this file into the top level, as though the code was all written at the top level in the first place.

```
datatype 'a myList = Nil | Cons of ('a * 'a myList)

fun myLength l =
  case l of
    Nil => 0
  | Cons(h,t) => 1 + myLength t


fun myMap f l =
  case l of
    Nil => Nil
```

```
  | Cons(h,t) => Cons(f h, myMap f t)

fun myFold f i l =
  case l of
      Nil => i
    | Cons(h,t) => myFold f (f(i,h)) t
```

This is problematic. As the program grow, we will start having name space problem. Note that there is some structure to our file MyList. It defines a type and some operations on this type. SML allows us to group such definitions into *structures*.

Here is how MyList.sml looks like:

```
structure MyList =
struct

datatype 'a myList = Nil | Cons of ('a * 'a myList)

fun myLength l =
  case l of
    Nil => 0
  | Cons(h,t) => 1 + myLength t


fun myMap f l =
  case l of
    Nil => Nil
  | Cons(h,t) => Cons(f h, myMap f t)

fun myFold f i l =
  case l of
      Nil => i
    | Cons(h,t) => myFold f (f(i,h)) t
end
```

Let's now quit sml to start a new and use "MyList.sml" again. Here is what we get when use MyList.sml. We cannot anymore access the contents of MyList.sml directly. They are under the structure MyList. To access the contents we must explicitly specify the name of the structure.

```
- use "MyList.sml";
[opening MyList.sml]
structure MyList :
  sig
    val myFold : ('a * 'b -> 'a) -> 'a -> 'b myList -> 'a
    val myLength : 'a myList -> int
    val myMap : ('a -> 'b) -> 'a myList -> 'b myList
```

```
    datatype 'a myList = Cons of 'a * 'a myList | Nil
  end
val it = () : unit

- Nil;
stdIn:2.1-2.4 Error: unbound variable or constructor: Nil

- MyList.Nil;
val it = Nil : 'a MyList.myList

- MyList.Cons (1, MyList.Cons (2, MyList.Nil));
val it = Cons (1,Cons (2,Nil)) : int MyList.myList
```

You can rename structures or open them.

```
- structure L = MyList;
structure L :
  sig
    val myFold : ('a * 'b -> 'a) -> 'a -> 'b myList -> 'a
    val myLength : 'a myList -> int
    val myMap : ('a -> 'b) -> 'a myList -> 'b myList
    datatype 'a myList = Cons of 'a * 'a myList | Nil
  end
- L.Cons (1, L.Cons (2, L.Nil));
- open MyList;
opening MyList
  val myFold : ('a * 'b -> 'a) -> 'a -> 'b myList -> 'a
  val myLength : 'a myList -> int
  val myMap : ('a -> 'b) -> 'a myList -> 'b myList
  datatype 'a myList = Cons of 'a * 'a myList | Nil
```

When you open a structure all its contents is incorporated into the current context (in this case the top level). I would strongly discourage the use of *open*.

"Open", however, is very useful when exploring the SML/NJ libraries. For example, a structure called `List` is already supplied by the SML/NJ library.

```
- open List;
[autoloading]
[autoloading done]
opening List
  datatype 'a list = :: of 'a * 'a list | nil
  exception Empty
  val null : 'a list -> bool
  val hd : 'a list -> 'a
  val tl : 'a list -> 'a list
  val last : 'a list -> 'a
  val getItem : 'a list -> ('a * 'a list) option
```

```
  val nth : 'a list * int -> 'a
  val take : 'a list * int -> 'a list
  val drop : 'a list * int -> 'a list
  val length : 'a list -> int
  val rev : 'a list -> 'a list
  val @ : 'a list * 'a list -> 'a list
  val concat : 'a list list -> 'a list
  val revAppend : 'a list * 'a list -> 'a list
  val app : ('a -> unit) -> 'a list -> unit
  val map : ('a -> 'b) -> 'a list -> 'b list
  val mapPartial : ('a -> 'b option) -> 'a list -> 'b list
  val find : ('a -> bool) -> 'a list -> 'a option
  val filter : ('a -> bool) -> 'a list -> 'a list
  val partition : ('a -> bool) -> 'a list -> 'a list * 'a list
  val foldr : ('a * 'b -> 'b) -> 'b -> 'a list -> 'b
  val foldl : ('a * 'b -> 'b) -> 'b -> 'a list -> 'b
  val exists : ('a -> bool) -> 'a list -> bool
  val all : ('a -> bool) -> 'a list -> bool
  val tabulate : int * (int -> 'a) -> 'a list
  val collate : ('a * 'a -> order) -> 'a list * 'a list -> order
-
```

It is much more extensive than our program but the basic idea is the same.

I encourage you to explore SML libraries and play with them. In my opinion, types are the best documentation that there exists. So I use open quite often to figure out how various functions are used, and to see what is provided by the library.

12 Standard Basis

12.1 List

Lists are fundamental in functional programming. The List library contains many critical commonly used list-oriented functions.

You have already seen List.map. The library also provides List.app that is similar to map except it applies a function purely for side-effect.

There are two flavors of fold: List.foldr and List.foldl. They accumulate over lists in opposite directions:

```
- foldr (fn (x,y) => x^y) "" ["hi", "hello", "howdy"];
val it = "hihellohowdy" : string
- foldl (fn (x,y) => x^y) "" ["hi", "hello", "howdy"];
val it = "howdyhellohi" : string
```

List.filter is a very useful function for selecting elements from a list meeting certain criteria encoded in a predicate function.

```
- filter (fn x => x mod 2 = 0) [0,1,2,3,4,5];
val it = [0,2,4] : int list
```

List offers hd and tl functions, but use of these functions are discouraged when pattern matching is possible.

12.2 ListPair

You might find yourself working with ListPairs every now and then when you encounter what is essentially an association list (i.e., a list of pairs). ListPair has app, map, foldl, and foldr functions adjusted for pairs of lists. Often you just want to decompose a list of pairs to a pair of lists or vice versa. The relevant functions are List.zip and List.unzip.

```
- ListPair.zip (["Sam", "Sally", "Steve"],[26,24,18]);
val it = [("Sam",26),("Sally",24),("Steve",18)] : (string * int) list

- ListPair.unzip [("Fred",21),("Fanny",22),("Flora",23)];
val it = (["Fred","Fanny","Flora"],[21,22,23]) : string list * int list
```

12.3 String

List of strings can be easily concatenated:

```
- String.concat ["hi", " ", "this", " ", "list"];
val it = "hi this list" : string
```

12.4 Arrays

Arrays and two-dimensional arrays are available as libraries Array and Array2.

```
- val a = Array.array (5,0);
val a = [|0,0,0,0,0|] : int array
- Array.update(a,3,Array.sub(a,3) + 1);
val it = () : unit
- a;
val it = [|0,0,0,1,0|] : int array
```

Array.app, Array.foldl, etc. are also available.

12.5 Option

There are also a few functions for working with Option data:

```
- Option.valOf(SOME 5);
val it = 5 : int
```

Generally speaking, pattern matching is preferable to these utility functions.

13 SML/NJ Library

In addition to the Standard Basis which is common to all Standard ML implementations, SML/NJ provides another library of useful advanced data structures. Most of these modules are "functorized" over order establishing modules that match the signature ORD_KEY. This way, SML/NJ can provide one set of functions that work over data with any kind of order.

The library provides various map, set, and even a splay tree data structure. All the implementations match ORD_MAP and ORD_SET signatures. Check the documentation on those signatures for more details.

For all these data structures, we have to specify the key type beforehand. However, the value type specification can be delayed until we actually start inserting values into the data structure.

The SML/NJ library also contains Red-Black tree implementations of Map and Set. These implementations are frequently used in practice.

`http://www.smlnj.org/doc/smlnj-lib/Manual/` gives a complete list of library functions.

13.1 Maps

```
- structure Map = RedBlackMapFn (struct type ord_key = int val compare = Int.compare end);
[autoloading]
[autoloading done]
structure Map : ORD_MAP?
- val m1 = Map.insert(Map.empty, 5, "good");
val m1 = - : string Map.map
- Map.find(m1,5);
val it = SOME "good" : string option
- val m2 = Map.insert(m1, 6, true);
stdIn:4.5-4.33 Error: operator and operand don't agree [tycon mismatch]
  operator domain: string Map.map * int * string
  operand:         string Map.map * int * bool
  in expression:
    Map.insert (m1,6,true)
```

13.2 Hash Tables

The type annotations in this example are necessary. Omitting them will lead to a hash table that does not work properly.

```
- structure StrTbl = HashTableFn(struct
   type hash_key = string
   val hashVal = HashString.hashString
   val sameKey = (op=) : string * string -> bool
 end);
structure StrTbl : MONO_HASH_TABLE
```

```
- val t1 : int StrTbl.hash_table = StrTbl.mkTable(10, Fail "Not found");
val t1 =
  HT
    {n_items=ref 0,not_found=Fail(-),
     table=ref [|NIL,NIL,NIL,NIL,NIL,NIL,NIL,NIL,NIL,NIL,NIL,NIL,...|]}
  : int StrTbl.hash_table
- StrTbl.insert t1 ("Alpha", 1);
val it = () : unit
- StrTbl.find t1 "Alpha";
val it = SOME 1 : int option
- StrTbl.find t1 "Beta";
val it = NONE : int option
```

14 CM: SML/NJ's Compilation Manager

CM is a tool that allow you to specify various files to be compiler. CM files end with .cm extension. Here is the .cm file from your homework.

```
Group is

  $/basis.cm
  $/smlnj-lib.cm
  Ast.sml
  TestFill.sml
```

The `$`'s are directives to the CM system. The rest are a listing of files that forms a group.

Alternatively, we could have specified these files as a library. The difference is that the library allows us to specify what is "visible" to the outside (in this case, they are the structure Ast, and structure Test).

```
Library
  structure Ast
  structure Test
is
  $/basis.cm
  $/smlnj-lib.cm

  Ast.sml
  TestFill.sml
```

When you have a `.cm` file, you can use `CM.make` function to make that file. `CM.make` is just a function defined by `structure CM`.

```
- CM.make ;
```

```
val it = fn : string -> bool

- CM.make "sources.cm";
[scanning sources.cm]
[loading (sources.cm):Ast.sml]
[loading (sources.cm):TestFill.sml]
[New bindings added.]
val it = true : bool
```

One you have make the sources, they will now be at the top level, as though you have applied "use" on all files.

```
open Test;
opening Test
  val DEBUG : bool
  val N_CHOICES : int
  val mkTree : int * int -> int -> int
  val randomTree : 'a * 'b -> 'c -> Ast.exp
  val selfCheck : int * int -> unit
  val test : 'a * 'b -> 'c -> int
  exception BadConfiguration
  exception N_CHOICESMustBeFour
- mkTree (10,10);
val it = fn : int -> int
- it 10;
tree = Num 1
eval(tree) = 1
val it = 1 : int
```

One word of caution about `open`. Once you `open` a structure, it is incorporated into the top level. If you change the code for that structure, and remake, the top level code will remain the same. To update you have to open again. For this reason, I avoid the open statement and instead type in the full path.

```
- Test.mkTree (10,10);
val it = fn : int -> int
- it 10;
tree = Num 1
eval(tree) = 1
val it = 1 : int
```

15 Further Information

The SML/NJ type errors may get very frustrating at times. Try not lose your patience and you will be rewarded. `http://www.smlnj.org/doc/` has very useful documentation about SML/NJ.

For tutorials see, `http://www.smlnj.org/doc/literature.html#tutorials` I strongly recommend that you look over Peter Lee's tutorial. For a detailed description of SML/NJ see Ricardo Pucello's notes.