Introduction To Standard ML

CS251 Programming Languages
Spring 2016, Lyn Turbak

Department of Computer Science
Wellesley College

The ML Programming Language

ML (Meta Language) was developed by Robin Milner in 1975 for specifying theorem provers. It since has evolved into a general purpose programming language.

Important features of ML:

- **static typing**: catches type errors at compile-time.
- **type reconstruction**: infers types so programmers don't have to write them explicitly
- **polymorphism**: functions and values can be parameterized over types (think Java generics, but much better).
- **function-oriented (functional)**: encourages a composition-based style of programming and first-class functions
- **sum-of-products dataypes with pattern-matching:** simplifies the manipulation of tree-structured data

These features make ML an excellent language for mathematical calculation, data structure implementation, and programming language implementation.

ML Dialects

There are several different dialects of ML. The two we use at Wellesley are:

- **Standard ML (SML)**: Version developed at AT&T Bell Labs and used by Paulson, Stoughton, and many others. We'll use this in CS235. The particular implementation we'll use is Standard ML of New Jersey (SMLNJ):

  http://www.smlnj.org/

- **Objective CAML**: Version developed at INRIA (France). We have sometimes used this in other Wellesley courses.

These dialects differ in minor ways (e.g., syntactic conventions, library functions). See the following for a comparison:

http://www.mpi-sws.mpg.de/~rossberg/sml-vs-ocaml.html

Learning SML by Interactive Examples

Try these in your wx appliance! (Note: many answers are missing in these slides so you can predict them.)

```
[wx@wx ~]$ which sml
/usr/local/smlnj/bin/sml

[wx@wx ~]$ sml
Standard ML of New Jersey v110.78 [built: Tue Aug 25 23:58:36 2015]

- 1 + 2;
val it =

- 3+4;
val it =

- 5+6
= ;
val it =

- 7
= +
= 8;
val it =
```

Naming Values

```
- val a = 2 + 3;
val a =  : int

- a * a;
val it =    : int

- it + a;
val it =    : int
```

Negative Quirks

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

- -17;
stdIn:60.1 Error: expression or pattern begins with infix
identifier "-"
stdIn:60.1-60.4 Error: operator and operand don't agree
[literal]
  operator domain: 'Z * 'Z
  operand:         int
  in expression:
    - 17

- ~17;
val it = ~17 : int

- 3 * ~1;
val it = ~3 : int
```

Division Quirks

```
- 7 / 2;
stdIn:1.1-1.6 Error: operator and operand don't agree
[literal]
  operator domain: real * real
  operand:         int * int
  in expression:
    7 / 2

- 7.0 / 2.0;
val it = 3.5 : real

- 7 div 2; (* integer division *)
val it = 3 : int

(* For a description of all top-level operators, see:
  http://www.standardml.org/Basis/top-level-chapter.html *)
```

Simple Functions

```
- val inc = fn x => x + 1;
val inc = fn : int -> int (* SML figures out type! *)

- inc a;
val it =   : int

- fun dbl y = y * 2;
  (* Syntactic sugar for val dbl = fn y => y * 2 *)
val dbl = fn : int -> int

- dbl 5;
val it =    : int

- (fn x => x * 3) 10; (* Don’t need to name function to use it *)
val it =    : int
```

When Parentheses Matter

```
- dbl(5); (* parens are optional here *)
val it = 10 : int

- (dbl 5); (* parens are optional here *)
val it = 10 : int

- inc (dbl 5); (* parens for argument subexpressions are required! *)
val it = 11 : int

- (inc dbl) 5;
stdIn:1.2-2.2 Error: operator and operand don't agree [tycon mismatch]
  operator domain: int
  operand:         int -> int
  in expression:
    inc dbl

- inc dbl 5; (* default left associativity for application *)
stdIn:22.1-22.10 Error: operator and operand don't agree [tycon
mismatch]
  operator domain: int
  operand:         int -> int
  in expression:
    inc dbl
```

Function Composition

```
- (inc o dbl) 10; (* SML builtin infix function composition *)
val it =     : int

- (dbl o inc) 10;
val it =     : int

- fun id x = x; (* we can define our own identity fcn *)
val id = fn : 'a -> 'a (* polymorphic type; compare to
  Java's public static <T> T id (T x) {return x;} *)

- (inc o id) 10;
val it =     : int

- (id o dbl) 10;
val it =     : int

- (inc o inc o inc o inc) 10;
val it =     : int
```

Functions as Arguments

```
- fun app5 f = f 5;
val app5 = fn : (int -> 'a) -> 'a

- app5 inc;
val it =   : int

- app5 dbl;
val it =    : int

- app5 (fn z => z - 2);
val it =   : int
```

We' ll see later that functions can also be returned as results from other functions and stored in data structures, so funtions are first-class in SML just as in Racket.

Scope of Top-Level Names

```
- val b = a * 2; (* recall a is 5 from before *)
val b =    : int

- fun adda x = x + a; (* a is still 5 from before *)
val adda = fn : int -> int

- adda 7;
val it =    : int

- adda b;
val it =    : int

- val a = 42; (* this is a different a from the previous one *)
val a =    : int

- b; (* ML values are immutable; nothing can change b’s value *)
val it =    : int

- adda 7;
val it =    : int (* still uses the a where adda was defined *)
```

Booleans

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

- 1 > 2;
val it =       : bool

- (1 = 1) andalso (1 > 2);
val it =       : bool

- (1 = 1) orelse (1 = 2);
val it =      : bool

- (3 = 4) andalso (5 = (6 div 0)); (* short-circuit evaluation *)
val it =       : bool

- fun isEven n = (n mod 2) = 0;
val isEven = fn : int -> bool (* SML figures out type! *)

- isEven 17;
val it =       : bool

- isEven 6;
val it =      : bool
```

Conditionals

```
- fun f n = if n > 10 then 2 * n else n * n;
val f = fn : int -> int

- f 20;
val it =    : int

- f 5;
val it =    : int
```

Recursion

```
- fun fact n =
=   if n = 0 then
=     1
=   else
=     n * (fact (n - 1)); (* fun names have recursive scope *)
val fact = fn : int -> int
  (* simpler than Java definition b/c no explicit types! *)

- fact 5;
val it =     : int

- fact 12;
val it =           : int

- fact 13;
uncaught exception Overflow [overflow]
  raised at: <file stdIn>
  (* SML ints have limited size ☹ *)
```

Local Naming via `let`

`let` is used to define local names. Any such names “shadow” existing definitions from the surrounding scope.

```
- let val a = 27     (* 1st let binding *)
=     val b = 3   (* 2nd  binding *)
=     fun fact x = x + 2 (* 3rd binding *)
=  in fact (a div b) (* let body *)
= end; (* end terminates the let *)
val it =     : int
```

`let`-bound names are only visible in the body of the `let`.

```
- fact (a div b); (* these are global names *)
val it =     : int
```

Easier to Put Your Code in a File

```
(* This is the contents of the file mydefns.sml.
   (* By the way, comments nest properly in SML! *)
   It defines integers A and B the fact function. *)

val a = 2 + 3

val b = 2 * a

fun fact n = (* a recursive factorial function *)
  if n = 0 then
    1
  else
    n * (fact (n - 1))
```

- File is a sequence of value/function definitions.
- Definitions are **not** followed by semi-colons in files!
- There are **no equal signs** for multiple-line definitions.

Using Code From a File

```
- Posix.FileSys.getcwd(); (* current working directory *)
val it = "/home/fturbak" : string

- Posix.FileSys.chdir("/home/wx/cs251/sml");
  (* change working directory *)
val it = () : unit

- Posix.FileSys.getcwd();
val it = "/home/wx/cs251/sml" : string

- use "mydefns.sml";  (* load defns from file as if *)
[opening mydefns.sml] (* they were typed manually *)
val a = 5 : int
val b = 10 : int
val fact = fn : int -> int
val it = () : unit

- fact a
val it = 120 : int
```

Another File Example

```
(* This is the contents of the file test-fact.sml *)

val fact_3 = fact 3

val fact_a = fact a
```

```
- use "test-fact.sml";
[opening test-fact.sml]
val fact_3 = 6 : int
val fact_a = 120 : int
val it = () : unit
```

Nested File Uses

```
(* The contents of the file load-fact.sml *)
use ”mydefns.sml”; (* semi-colons are required here *)
use ”test-fact.sml”;
```

```
- use "load-fact.sml";
[opening load-fact.sml]
[opening mydefns.sml]
val a = 5 : int
val b = 10 : int
val fact = fn : int -> int
val it = () : unit
[opening test-fact.sml]
val fact_3 = 6 : int
val fact_a = 120 : int
val it = () : unit
val it = () : unit
```

Tuples

```
- val tpl = (1 + 2, 3 < 4, 5 * 6, 7 = 8);
val tpl = ( ,    ,  ,     ) : int * bool * int * bool

- #1 tpl;
val it =  : int

- #2 tpl;
val it =      : bool

(* In practice, always use pattern matching (below)
    rather than #1, #2, etc. *)
- ((#1 tpl) + (#3 tpl), (#2 tpl) orelse (#4 tpl));
val it = (  ,    ) : int * bool

(* Can “deconstruct” tuples via pattern matching *)
- let val (i1, b1, i2, b2) = tpl
=  in (i1 + i2, b1 orelse b2)
= end;
val it = (  ,    ) : int * bool
```

Strings

```
- "foobar";
val it =          : string

- "foo" ^ "bar" ^ "baz";
val it =             : string

- print ("baz" ^ "quux");
bazquuxval it = () : unit

- print ("baz" ^ "quux\n"); (* parens are essential here! *)
bazquux
val it = () : unit

- print "baz" ^ "quux\n";
stdIn:1.1-1.23 Error: operator and operand don't agree
[tycon mismatch]
  operator domain: string * string
  operand:         unit * string
  in expression:
    print "baz" ^ "quux\n"
```

Other String Operations

```
- String.size ("foo" ^ "bar");
val it =   : int

- String.substring ("abcdefg", 2, 3); (* string, start index, len *)
val it =       : string

("bar" < "foo", "bar" <= "foo", "bar" = "foo", "bar" > "foo");
val it = (    ,    ,     ,     ) : bool * bool * bool * bool

-(String.compare("bar", "foo"), String.compare("foo", "foo"),
= String.compare("foo", "bar"));
val it = (    ,     ,       ) : order * order * order

- String.size;
val it = fn : string -> int

- String.substring;
val it = fn : string * int * int -> string

- String.compare;
val it = fn : string * string -> order

(* An API for all SMLNJ String operations can be found at:
http://www.standardml.org/Basis/string.html *)
```

Characters

```
- #"a";
val it = #"a" : char

- String.sub ("foobar",0);
val it =      : char

- String.sub ("foobar",5);
val it =      : char

- String.sub ("foobar",6);
uncaught exception Subscript [subscript out of bounds]
  raised at: stdIn:17.1-17.11

- String.str #"a"; (* convert a char to a string *)
val it = "a" : string

- (String.str (String.sub ("ABCD",2))) ^ "S"
= ^ (Int.toString (112 + 123));
val it =         : string

- (1+2, 3=4, "foo" ^ "bar", String.sub("baz",2));
val it = ( ,     ,         ,      ) : int * bool * string * char
```

Pattern-matching Function Arguments

```
- fun swap (x,y) = (y, x);
val swap = fn : 'a * 'b -> 'b * 'a (* infers polymorphic type *)

- swap (1+2, 3=4);
val it = (false,3) : bool * int

- swap (swap (1+2, 3=4));
val it = (3,false) : int * bool

- swap ((1+2, 3=4), ("foo" ^ "bar", String.sub("baz",2)));
val it = (("foobar",#"z"),(3,false)) : (string * char) * (int *
bool)
```

How to Pass Multiple Arguments

```
- fun avg1 (x, y) = (x + y) div 2; (* Approach 1: use pairs *)
val avg1 = fn : int * int -> int

- avg1 (10,20);
val it =    : int

- fun avg2 x = (fn y => (x + y) div 2); (* Approach 2: currying *)
val avg2 = fn : int -> int -> int

- avg2 10 20;
val it =    : int

- fun avg3 x y = (x + y) div 2; (* Syntactic sugar for currying *)
val avg3 = fn : int -> int -> int

- avg3 10 20;
val it =    : int

- app5 (avg3 15);
val it =    : int

- app5 (fn i => avg1(15,i));
val it =    : int
```

A Sample Iteration

```
(* This is the contents of the file step.sml *)

fun step (a,b) = (a+b, a*b)

fun stepUntil ((a,b), limit) = (* no looping constructs in ML; *)
  if a >= limit then           (* use tail recursion instead! *)
    (a,b)
  else
    stepUntil (step(a,b), limit)
```

```
- use ("step.sml");
[opening step.sml]
val step = fn : int * int -> int * int
val stepUntil = fn : (int * int) * int -> int * int
val it = () : unit

- step (1,2);
val it = (3,2) : int * int

- step (step (1,2));
val it = (5,6) : int * int

- let val (x,y) = step (step (1,2)) in x*y end;
val it = 30 : int

- stepUntil ((1,2), 100);
val it = (371,13530) : int * int
```

Adding print statements

```
(* This is the contents of the file step-more.sml *)

fun printPair (a,b) =
  print ("(" ^ (Int.toString a) ^ ","
          ^ (Int.toString b) ^ ")\n")

fun stepUntilPrint ((a,b), limit) =
  if a >= limit then
    (a,b)
  else
    (printPair (a,b); (* here, semicolon sequences expressions *)
     stepUntilPrint (step(a,b), limit))
```

```
- use ("step-more.sml");
[opening step-more.sml]
val printPair = fn : int * int -> unit
val stepUntilPrint = fn : (int * int) * int -> int * int
val it = () : unit

- stepUntilPrint ((1,2),100);
(1,2)
(3,2)
(5,6)
(11,30)
(41,330)
val it = (371,13530) : int * int
```

Counting Chars

Want to count the number of times a given char c appears in a string. E.g.:

```
- countChar (“abracadabra“, #”a”);
5 : int

- countChar (“abracadabra“, #”b”);
2 : int

- countChar (“abracadabra“, #”e”);
0 : int
```

Write recursive and iterative definitions of countChar.
You may use the following helper functions:

```
fun first s = String.sub (s,0)

fun butFirst s = String.substring (s, 1, (String.size s) - 1)
```