Standard ML Tutorial[1]

January 6, 2009

[1] Adapted from slides and notes by John Reppy and Matthias Blume

Course Project

The project for the course is to implement a small functional programming language, called LangF. (Students who have taken CMSC 22100 should recognize the language as an enrichment of System F, the polymorphic λ-calculus.) The project will be divided into four parts, each requiring a significant programming effort. The implementation will be undertaken using the ***Standard ML*** *programming language and submission of the project milestones will be managed using the course GForge server. Programming projects will be individual efforts (no group submissions).*

There are lots of programming languages — why Standard ML?

Why Standard ML?

A language particularly suited to compiler implementation:

- Efficiency
- Safety
- Simplicity
- Higher-order functions
- Static type checking with type inference
- Polymorphism
- Algebraic datatypes and pattern matching
- Modularity
- Garbage collection
- Exceptions and exception handling
- Libraries and tools

What is Standard ML?

SML is a strongly typed, impure, strict, functional language:

- **Strongly typed:** Every expression in the language has a *type* (`int`, `real`, `bool`, etc.). The compiler rejects a program that does not confirm to the type system.
- **Functional:** Every expression evalutes to a *value*. One kind of value is a *function*. In fact, every function is a value. Like other values, functions can be bound to variabels, passed as arguments to function calls, returned as values from function calls, and stored in data structures.

What is Standard ML?

SML is a strongly typed, impure, strict, functional language:

- **Impure** The evaluation of expressions in SML can incur *side-effects*, e.g., assigment to locations in mutable data structures or I/O.
- **Strict** The arguments to SML functions are evaluated before the function call is performed. Thus, if one of the arguments loops forever, then so does the entire program — regardless of whether or not the function actually needed the argument. Similarly, all side-effects caused by the evalution of the argument occur before any side-effects caused by the evaluation of the function body.

Using the SML/NJ Compiler

- Type `sml` to run the SML/NJ interactive compiler.
  - Installed in `usr/local/bin` on CS dept. Linux machines.
- `Ctrl-d` exits the compiler; `Ctrl-c` interrupts execution.
- Four ways to run ML programs:
  1. type in code in the interactive read-eval-print loop

     ```
     - 1 + 1;
     ```

  2. load ML code from a file (e.g., `foo.sml`)

     ```
     - use "foo.sml";
     ```

  3. use Compilation Manager (CM)

     ```
     - CM.make "sources.cm";
     ```

  4. load/compile a program using one of the previous methods, then export a function to be run in a later session.
     - course project will demonstrate this method

Simple expressions

- Integers: `3`, `54`, `~3`, `~54`
- Reals[2]: `3.0`, `3.14159`, `~3.6E00`
- Booleans: `true`, `false`, `not`
- Strings: `"abc"`, `"hello world\n"`, `x ^ ".sml"`
- Chars: `\#"a"`, `\#"\n"`,
- Overloaded operators: `+`, `-`, `*`, `<`, `<=`
- Lists: `[]`, `[1,2,3]`, `["x","sml"]`, `1::2::nil`
- Tuples: `()`, `(1,true)`, `(3,"abc",false)`
- Records: `{a=1,b=true}`, `{name="bob",age=8}`
- conditionals, functions, function applications

[2]floating-point numbers

Value Declarations

Binding a value to a variable.

- syntax

$$\texttt{\textbf{val}}\ \mathit{var} = \mathit{exp}$$

- examples

```
val x = 3

val y = x + 1

val z = y - x
```

Thus, variables are identifiers that *name* values. Once a binding for a variable is established, the variable names the *same* value until it goes out of scope. Standard ML variables are *immutable*.

Function Declarations

Binding a function (which is a value) to a variable.

- syntax (simplified)

$$\textbf{fun}\ \mathit{var}_f\ \mathit{var}_a = \mathit{exp}$$

- examples

```
fun fact_loop (n, f) =
  if n = 0 then f
  else fact_loop (n - 1, n * f)

fun fact n = fact_loop (n, 1)
```

Let expressions

Limit the scope of variables from declarations.

- syntax

  let *decl* **in** *exp* **end**

- example

```
let
  val x = let val y = 1
          in y + y
          end
  fun f z = (z, x * z)
in
  f (4 + x)
end
```

Function expressions

Introduce a function from one argument to one result.
Such an *anonymous* function has no name, but is a value, so it can be bound to a variable.

- syntax (simplified)

  fn *var* => *exp*

- example

  ```
  val double = fn z => 2.0 * z

  val inc = fn x => x + 1
  ```

  The last is equivalent to

  ```
  fun inc x = x + 1
  ```

Function expressions (cont.)

Because functions are *first-class*, one function can return another function as a result.

- example

```
val add = fn x => fn y => x + y
val inc = add 1   (* == fn y => 1 + y *)
val three = inc 2
```

The first is equivalent to

```
fun add x y = x + y
```

This is one "solution" to functions taking multiple arguments; such functions are called *curried* functions.
Another "solution" is to take a value that is a data structure containing multiple values.

Tuple and record expressions

Create (and take apart) collections of values.

- tuples, syntax

  (*exp*$_1$, . . . , *exp*$_n$)   # *digit exp*

- tuples, examples

```
val x = ("foo", 1.0 / 2.0, false)
val first = #1 x
val third = #3 x
```

- records, syntax

  { *lab*$_1$ = *exp*$_1$, . . . , *lab*$_n$ = *exp*$_n$ }   # *lab exp*

- records, examples

```
val car = {make = "Toyota", year = 2001}
val mk = #make car
val yr = #year car
```

List expressions

Finite sequences of values.

- syntax

$$\texttt{nil} \qquad \mathit{exp}_x \; \texttt{::} \; \mathit{exp}_l$$

$$\texttt{[} \; \mathit{exp}_1 \; \texttt{,} \; \ldots \; \texttt{,} \; \mathit{exp}_n \; \texttt{]}$$

- examples

```
val l0 = nil
val l1 = 1.0 :: 2.0 :: 3.0 :: nil
val l2 = [1.0, 2.0, 3.0]
val l3 = 1.0 :: 2.0 :: [3.0]
```

All of `l1`, `l2`, and `l3` are equivalent.

Patterns

Decompose compound values; commonly used in value bindings and function arguments.

- revized syntax for declarations and function expressions

  val *pat* = *exp* **fun** *var*$_f$ *pat*$_a$ = *exp*

  fn *pat* => *exp*

- variable patterns

```
val z = 3
val pair = (z, true)
```

  ⇒ z = 3, pair = (3, true)

- tuple and record patterns

```
val (x,y) = pair
```

  ⇒ x = 3, y = true

```
val {make=mk, year=yr} = car
```

  ⇒ mk = "Toyota", yr = 2001

Patterns (cont.)

- wildcard patterns

```
val _ = 4 * 3 * 2 * 1
```

$\Rightarrow$

- constant patterns

```
val 3 = 1 + 2
val true = 1 < 3
```

- constructor patterns

```
val l = [1,2,3]
val fst::rest = l
val [x,_,z] = l
```

$\Rightarrow \mathtt{fst} = 1, \mathtt{rest} = [2, 3], \mathtt{x} = 1, \mathtt{z} = 3$

Patterns (cont.)

- nested patterns

```
val ((x,y),z) = ((1,2),3)
val (a,b)::_ = [(3.0,true),(5.0,false)]
```

⇒ x = 1, y = 2, z = 3
⇒ a = 3.0, b = true

- as patterns

```
val l as (a,b)::_ = [(3.0,true),(5.0,false)]
val t as (p as (x,y),z) = ((1,2),3)
```

⇒ l = [(3.0,true),(5.0,false)],
⇒ a = 3.0, b = true,
⇒ t = ((1,2),3), p = (1,2), x = 1,
⇒ y = 2, z = 3

Pattern matching

What to do when there is more than one way to decompose a value? Use *pattern matching* to consider each possible way.

- match rule, syntax

$$\mathtt{pat} \; \texttt{=>} \; \mathtt{exp}$$

- match, syntax

$$\mathtt{pat}_1 \; \texttt{=>} \; \mathtt{exp}_1 \; | \; \cdots \; | \; \mathtt{pat}_n \; \texttt{=>} \; \mathtt{exp}_n$$

  When a match is applied to a value `value`, we try the rules from left to right, looking for the first rule whose pattern matches `value`. We then bind the variables in the pattern and evaluate the expression.

Pattern matching (cont.)

Pattern matching is used in a number of expression and declaration forms.

- case expression, syntax

$$\texttt{case}\ \mathit{exp}\ \texttt{of}\ \mathit{match}$$

- function expression, syntax

$$\texttt{fn}\ \mathit{match}$$

- clausal function declaration, syntax

$$\texttt{fun}\ \mathit{var}_f\ \mathit{pat}_1 = \mathit{exp}_1 \mid \cdots \mid \mathit{var}_f\ \mathit{pat}_n = \mathit{exp}_n$$

  The function name (var_f) is the same in all branches.

Pattern matching examples

```
fun length l =
  case l of [] => 0
          | _ :: r => 1 + length r

fun length [] = 0
  | length (_ :: r) = 1 + length r


val isZero = fn 0 => true | _ => false


fun even 0 = true
  | even n = odd (n - 1)
and odd 0 = false
  | odd n = even (n - 1)
```

Types

Every expression has a *type*.

- primitive types: `int`, `string`, `bool`

  `3 : int`   `true : bool`   `"abc" : string`

- function types: ty_1 `->` ty_2

  `even : int -> bool`

- product types: $ty_1 * \cdots * ty_n$, `unit`

  `(3, true) : int * bool`   `() : unit`

- record types: `{` lab_1`:` ty_1 `,` $\cdots$ `,` lab_n`:` ty_n `}`

  `car : {make: string, year: int}`

- type operators: ty `list` (for example)

  `[1,2,3] : int list`

Type abbreviations

Introduce a new name for a type.

- syntax

$$\texttt{\textbf{type}}\ \mathit{tycon} = \mathit{ty}$$

- examples

```
type point = real * real
type line = point * point
type car = {make: string, year: int}
```

- syntax

$$\texttt{\textbf{type}}\ \mathit{tyvar}\ \mathit{tycon} = \mathit{ty}$$

- examples

```
type 'a pair = 'a * 'a
type point = real pair
```

Datatypes

Algebraic datatypes are one of the most useful and convenient features of Standard ML (and other functional programming languages). They introduce a (brand) new type that is a *tagged union* of some number of variant types.

- syntax

  $$\textbf{datatype}\ \mathit{tycon} = \mathit{con}_1\ \textbf{of}\ \mathit{ty}_1 \mid \cdots \mid \mathit{con}_n\ \textbf{of}\ \mathit{ty}_n$$

- example

```
datatype color = Red | Green | Blue
datatype shape =
    Circle of color * real
  | Rectangle of color * real * real
```

Datatypes (cont.)

The data constructors can be used in both expressions to create values of the new type and in pattrns to discriminate variants and to decompose values.

- example

```
fun area s =
  case s of
     Circle (_, r) = Math.pi * r * r
   | Rectangle (_, l1, l2) => l1 * l2

val c = Circle (Red, 2.0)

val a = area c
```

Datatypes can be *recursive*.

- example

```
datatype intlist = Nil | Cons of int * intlist
```

Datatype example

```
datatype btree = LEAF
               | NODE of int * btree * btree

fun depth LEAF = 0
  | depth (NODE (_, t1, t2)) =
      max (depth t1, depth t2)

fun insert (LEAF, k) = NODE (k, LEAF, LEAF)
  | insert (NODE (i, t1, t2), k) =
      if k > i then NODE (i, t1, insert (t2, k))
      else if k < i then NODE (i, insert (t1, k), t2)
      else NODE (i, t1, t2)

(* in-order traversal of btrees *)
fun btreeToList LEAF = []
  | btreeToList (NODE (i, t1, t2)) =
      (btreeToList t1) @ (i :: (btreeToList t2))
```

Representing programs as datatypes

```
type id = string

datatype binop = PLUS | MINUS | TIMES | DIV

datatype stm = SEQ of stm * stm       (* s1 ; s2 *)
             | ASSIGN of id * exp     (* x := e *)
             | PRINT of exp list      (* print (e1,...) *)
and exp = VAR of id                      (* x *)
        | CONST of int                   (* 3 *)
        | BINOP of binop * exp * exp     (* e1 + e2 *)
        | ESEQ of stm * exp              (* s ; e *)

val prog =
  SEQ (ASSIGN ("a", BINOP (PLUS, CONST 5, CONST 3)),
       PRINT (VAR "a"))
(* a := 5 + 5 ; print (a) *)
```

Computing properties of programs: size

```
fun sizeS (SEQ(s1,s2)) = sizeS s1 +  sizeS s2
  | sizeS (ASSIGN(_,e)) = 2 + sizeE e
  | sizeS (PRINT es) = 1 + sizeEL es

and sizeE (BINOP(_,e1,e2)) = sizeE e1 + sizeE e2 + 2
  | sizeE (EQSEQ (s, e)) = sizeS s + sizeE e
  | sizeE _ = 1

and sizeEL [] = 0
  | sizeEL (e::es) = sizeE e + sizeEL es
```

sizeS prog ⇒ 8

Type inference

When defining values (including functions), types do not need to be declared — they will be *inferred* by the compiler:

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

- fun isPos n = n > 0
val isPos = fn : int -> bool
```

Any inconsistencies will be detected as type errors.

```
- if 1 < 2 then 3 else 4.0;
stdIn:1.1-1.25 Error: types of if branches do not agre
  then branch: int
  else branch: real
  in expression:
    if 1 < 2 then 3 else 4.0
```

Some error messages are better than others....

Type inference (cont.)

Type inference works with *all* types in the language.

```
- fun area (Circle (_,r)) = Math.pi * r * r
=    | area (Rectangle (_,l1,l2)) = l1 * l2;
val area = fn : shape -> real
```

Overloaded operators default to `int`;
use type annotations (called *ascriptions*) to be explicit.

```
- fun add (x, y) = x + y;
val add = fn : int * int -> int
- fun addR (x: real, y) = x + y;
val addR = fn : real * real -> real
```

Type inference (cont.)

Tuple and record selectors need to know the type of the argument.

```
- fun first p = #1 p
stdIn:1.1-1.19 Error: unresolved flex record
   (can't tell what fields there are besides #1)
- fun first (p: int * int) = #1 p;
val first = fn : int * int -> int
- val getMake = fn {make=mk, ...} => mk;
stdIn:2.5-2.38 Error: unresolved flex record
   (can't tell what fields there are besides #mak
- val getMake = fn ({make=mk, ...}: car) => mk;
val getMake = fn : car -> string
```

Polymorphic type inference

Type inference produces the *most general* type, which may be *polymorphic*.

```
- fun ident x = x;
val ident = fn : 'a -> 'a
- fun pair x = (x, x);
val pair = fn : 'a -> 'a * 'a
- val fst = fn (x, y) => x
val fst = fn : 'a * 'b -> 'a
- val foo = pair 4.0;
val foo = (4.0,4.0) : real * real
```

`pair` was used at the type `real -> real * real`.

```
- val z = fst foo;
val z = 4.0 : real
```

`fst` was used at the type `real * real -> real`.

Polymorphic datatypes

```
datatype 'a btree = LEAF
                  | NODE of 'a * 'a btree * 'a btree

fun depth LEAF = 0
  | depth (NODE (_, t1, t2)) =
      max (depth t1, depth t2)
val depth = fn : 'a btree -> int

fun btreeToList LEAF = []
  | btreeToList (NODE (x, t1, t2)) =
      (btreeToList t1) @ (x :: (btreeToList t2))
val btreeToList = fn : 'a btree -> 'a list

fun btreeMap f LEAF = LEAF
  | btreeMap f (NODE (i, t1, t2)) =
      NODE (f x, btreeMap f t1, btreeMap f t2)
val btreeMap = fn : ('a -> 'b) -> 'a btree -> 'b btree
```

Exceptions

```
- 5 div 0;   (* primitive failure *)
uncaught exception Div

exception NotFound of string  (* declare new exception *)
type 'a dict = (string * 'a) list
fun lookup (s, nil) = raise (NotFound s)
  | lookup (s, (k,v)::rest) =
      if s = k then v else lookup (s, rest)
val lookup : string * 'a dict -> 'a

val d = [("foo",2), ("bar",~1)]
val d : (string * int) list  (* == int dict *)

val x = lookup ("foo", d)
val x = 2 : int

val y = lookup ("baz", d)
uncaught exception NotFound

val y = lookup ("baz", d) handle NotFound s =>
        (print ("NotFound: " ^ s ^ "\n") ; 0)
NotFound: baz
val y = 0 : int
```

References and Assignments

Although SML variables are immutable, SML provides a type of mutable cells.

```
type 'a ref
val ref : 'a -> 'a ref
val ! : 'a ref -> 'a
val := : 'a ref * 'a -> unit
```

```
- val lineNum = ref 0;   (* create mutable cell *)
val lineNum = ref 0 : int ref

- fun lineCount () = !lineNum;   (* access mutable cell *)
fun lineCount = fn : unit -> int

- fun newLine () = lineNum := !lineNum + 1;   (* increment the cell *)
fun newLine = fn : unit -> unit

- val lineNum = ref 0;   (* create mutable cell *)
val lineNum = ref 0 : int ref
```

References and Assignments (cont.)

SML variables are immutable:

```
local
  val x = 1
in
  fun new1 () = let val x = x + 1 in x end
end
```

`new1` always returns `2`.

SML references are mutable:

```
local
  val x = ref 1
in
  fun new2 () = (x := !x + 1; !x)
end
```

`new2` returns `2`, `3`, `4`, . . . on successive calls.

Modules – Structures

A *structure* is an encapsulated, named, collection of declarations.

```
structure Ford =
struct
  type car = {make: string, built: int}
  val first = {make = "Ford", built: 1904}
  fun mutate ({make,built}: car) year =
    {make = make, built = year}
  fun built ({built, ...}: car) = built
  fun show (c) = if built c < built first then " - "
                 else "(generic Ford)"
end

structure Year =
struct
  type year = int
  val first = 1900
  val second = 2000
  fun newYear (y: year) = y + 1
  fun show (y: year) = Int.toString y
end

structure MutableCar =
struct
  structure C = Ford
  structure Y = Year
end
```

Modules – Signatures

A *signature* is an encapsulated, named, collection of specifications.

```
signature MANUFACTURER =
sig
  type car
  val first : car
  val built : car -> int
  val mutate : car -> int -> car
  val show : car -> string
end

signature YEAR =
sig
  type year
  val first: year
  val second: year
  val newYear: year -> year
  val show: year -> string
end

signature MCSIG =
struct
  structure C : MANUFACTURER
  structure Y : YEAR
end
```

Modules – Signature Matching

A structure `S` *matches* signature `SIG` if every specification in `SIG` is satisfied by a component of `S`.

```
structure YearX : YEAR =
struct
  type year = int
  type century = string
  val first = 1900
  val second = 2000
  fun newYear (y: year) = y + 1
  fun leapYear (y: year) = y mod 4 = 0
  fun show (y: year) = Int.toString y
end

structure MCar : MCSIG = MutableCar

(* Use 'dot notation' to access components of structures. *)
val classic = YearX.show 1968

val antique = MCar.Y.show 1930

(* Can't access components not specified in signature. *)
val x = YearX.leapYear(YearX.first)   (* ERROR *)
```

Modules – Functors

A *functor* is a “function” from structures to structures; create new structure *parameterized* by a signature.

```
signature ORD =
sig
  type t
  val lt : t * t -> bool
end

functor Sort(X: ORD) =
struct
  fun insert(x, nil) = [x]
    | insert(x, l as y::ys) =
        if X.lt (x, y) then x::l
        else y::(insert (x, ys))
  fun sort (l: X.t list) =
    foldl insert nil l
end

structure IntOrd =
struct
  type t = int
  val lt = fn (x, y) => x < y
end

structure IntSort = Sort(IntOrd)

val one_two_three_four = IntSort.sort [2,4,3,1]
```

Modules – Type Abstraction

Sometimes we don't want clients of a structure to know how a type is implemented.
Consider the problem of providing *unique* identifiers:

```
signature UID =
sig
  type uid
  val compare : uid * uid -> order
  val gensym : unit -> uid
end

structure Uid : UID =
struct
  type uid = int
  val compare = Int.compare
  val count = ref 0   (* hidden *)
  fun gensym () = let val id = !count
                  in count := id + 1; id
                  end
end

val a = Uid.gensym ()
val b = Uid.gensym ()
val _ (* LESS *) = Uid.compare (a, b)

val c: Uid.uid = 1
val _ (* EQUAL *) = Uid.compare (b, c)
```

But, two unique identifiers should be equal iff they came from the same `gensym`.

Modules – Type Abstraction

Sometimes we don't want clients of a structure to know how a type is implemented.
Consider the problem of providing *unique* identifiers:

```
structure Uid :> UID =
struct
  type uid = int   (* abstract *)
  val compare = Int.compare
  val count = ref 0   (* hidden *)
  fun gensym () = let val id = !count
                  in count := id + 1; id
                  end
end

val a = Uid.gensym ()
val b = Uid.gensym ()
val _ (* LESS *) = Uid.compare (a, b)

(* Don't know that Uid.uid == int. *)
val c: Uid.uid = 1   (* ERROR *)
```

Readers

The `StringCvt` module defines the reader type, which defines a *pattern* of functional input.

```
type ('item, 'strm) reader =
  'strm -> ('item, 'strm) option
```

```
val Int.scan : (char, 'strm) reader -> (int, 'strm) reader
val Bool.scan : (char, 'strm) reader -> (bool, 'strm) reader
val List.scan : ((char, 'strm) reader -> ('a, 'strm) reader) ->
                (char, 'strm) reader -> ('a list, 'strm) reader
```

Readers – Examples

```
fun scanBool charRdr strm =
  let
    fun chkStrm (strm,[]) = SOME strm
      | chkStrm (strm,x::xs) =
          case charRdr strm of
             NONE => NONE
           | SOME (c, strm') =>
               if c = x
                  then chkStrm (strm',xs)
                  else NONE
  in
    case chkStrm (strm, explode "false") of
       SOME strm' => SOME (false,strm')
     | NONE => (case chkStrm (strm, explode "true") of
                  SOME strm' => SOME (true, strm')
                | NONE => NONE)
  end

val scanBool : (char, 'strm) reader -> (bool, 'strm) reader
```

Readers – Examples

```
fun skipWS charRdr =
  let
    fun skip strm =
      case charRdr strm of
         NONE => strm
       | SOME (c, strm') =>
           if Char.isSpace c
              then skip strm'
              else strm
  in
    skip
  end

val skipWS : (char, 'strm) reader -> 'strm -> 'strm
```