

A Crash Course on Standard ML

Hongwei Xi

Version of December 19, 2002

1 Start

Type 'sml-cm' at a Unix window to get an interactive interpreter for Standard ML (SML). You can later quit the interpreter by typing ctrl-D.

2 Generalities

At the prompt, which is a dash symbol -, you can type an expression or a declaration. Here are some examples on evaluating expressions.

```
- 1+2;
val it = 3 : int
- "Hello" ^ " world!";
val it = "Hello world!" : string
- print "Hello, world!\n";
Hello, world!
val it = () : unit
- fn x:int => x + 1;
val it = fn : int -> int
```

Each time you type a semicolon followed by Enter, you get your expression evaluated. If you just press Enter without a semicolon preceding it, the interpreter gives a continuation prompt =, allowing you to type a new line without evaluating what you have just written.

You can also introduce declarations as follows, which bind n and f to 0 and the successor function on integers, respectively.

```
- val n = 0;
val n = 0 : int
- val f = fn x => x+1;
val f = fn : int -> int
```

The arithmetic operators \sim (the unary minus operator), $+$, $-$ and $*$ are overloaded; they can take either integer or real arguments. For instance, we have

```
- 1 + 1;
val it = 2 : int
- 1.0 + 1.0;
val it = 2.0 : real
- ~(2 - 1);
val it = ~1 : int
```

The division operators *div* and */* take integers and reals, respectively. For instance, we have

```
- 6 div 3;  
val it = 2 : int  
- 6.0 / 3.0;  
val it = 2.0 : real
```

Note that the expression `6 / 3` is ill-typed.

3 Expressions

We have seen examples of arithmetic expressions and string expressions. We now introduce more forms of expressions.

3.1 let-expressions

We can also form an expression involving some local declarations as follows.

```
let  
  val x = 1  
  fun f (x:int) = x + x  
in  
  f(x) * f(x)  
end
```

3.2 if-then-else-expressions

The syntax for *if-then-else* expressions in SML is as follows,

```
if <exp_0> then <exp_1> else <exp_2>
```

where the expression $\langle exp_0 \rangle$ needs to have the type *bool* and the expressions $\langle exp_1 \rangle$ and $\langle exp_2 \rangle$ must have the same type.

3.3 case-expressions

Another form of expression involves pattern matching,

```
case <exp_0> of <pat_1> => <exp_1> | ... | <pat_n> => <exp_n>
```

which evaluates $\langle exp_1 \rangle$ to a value v , chooses the first clause $\langle pat_i \rangle \Rightarrow \langle exp_i \rangle$ such that v matches $\langle pat_i \rangle$, and then evaluates $\langle exp_i \rangle$ after introducing some proper bindings for the variable in $\langle pat_i \rangle$. If v matches none of the patterns $\langle pat_1 \rangle, \dots, \langle pat_n \rangle$, a *Match* exception is raised. In particular, the above *if-then-else* expression is equivalent to the following expression.

```
case <exp_0> of true => <exp_1> | false => <exp_2>
```

We are to say more about pattern matching later.

3.4 Tuples and Records

You can readily form tuples and select from tuples. Here is an example.

```
- val x = (1, true, #"a", "a");
val x = (1,true,#"a","a") : int * bool * char * string
- #1 (x);
val it = 1 : int
- #2 (x);
val it = true : bool
- #3 (x);
val it = #"a" : char
- #4 (x);
val it = "a" : string
```

For a tuple of length n , there are n projections $\#1, \dots, \#n$ for selecting the components of the tuple.

A record is basically a tuple with field name. The ordering of the components in a record is irrelevant. For instance, $\{x = 0, y = false\}$ is equivalent to $\{y = false, x = 0\}$. Here is an example of forming and selecting from a record.

```
- val r = {a = 1, b = true, c = #"a", d = "a"};
val r = {a=1,b=true,c=#"a",d="a"} : {a:int, b:bool, c:char, d:string}
- #a(r);
val it = 1 : int
- #b(r);
val it = true : bool
- #c(r);
val it = #"a" : char
- #d(r);
val it = "a" : string
```

For a record with a field name n , $\#n$ is the selector for selecting the value associated with the field name n in the record. Note that a tuple is just syntactic sugar for a records with integer fields. For instance, $(0, false)$ is really for the record $\{1 = 0, 2 = false\}$.

4 Functions

We can declare a recursive function in SML as follows.

```
val rec f91: int -> int =
  fn n => if n <= 100 then f91 (f91 (n+11)) else n - 10
```

It is required that the right-hand side of a declaration following the keyword `val rec` be a function expression. The function *f91* can also be declared through the following syntax,

```
fun f91 (n:int): int =
  if n <= 100 then f91 (f91 (n+11)) else n-10
```

which should really be regarded as syntactic sugar for the function declaration given previously.

The following example declares two mutually recursive functions.

```

fun even (n:int): bool =
  if n = 0 then true else odd (n-1)

and odd (n: int): bool =
  if n = 0 then false else even (n-1)

```

When defining a function, we may often define some help functions or auxiliary functions. The following declaration is such an example.

```

fun fib (n: int): int =
  let
    fun aux (n: int): int * int =
      if n = 0 then (1, 0)
      else
        let
          val (x, y) = aux (n-1)
        in
          (x+y, x)
        end
    in
      #2 (aux (n))
    end

```

Note that if you want a function to return multiple values then you can make the function to return a tuple containing these values. The inner function *aux* in the above example is such a case.

5 Datatypes

You can declare datatypes to model various data structures. For instance, the following declaration forms a datatype for binary trees with integer labelled branching nodes.

```

datatype binary_tree = E | B of binary_tree * int * binary_tree

```

The function computing the height of a binary tree can be defined as follows through the use of pattern matching.

```

fun height E = 0
  | height (B (lt, _, rt)) = 1 + max (height lt, height rt)

```

For instance, $height(B(E, 0, B(E, 1, E)))$ evaluates to 2.

The following is a more interesting example, where a datatype is declared for representing untyped λ -terms using deBruijn indices.

```

datatype lambda_expr1 =
  One
| Shift of lambda_expr1
| Lam of lambda_expr1
| App of lambda_expr1 * lambda_expr1

```

For instance, the λ -term $\lambda x. \lambda y. x(y)$ can be represented as follows.

$$Lam(Lam(App(Shift(One), One)))$$

We can also use higher-order abstract syntax to represent closed λ -terms.

```
datatype lambda_expr2 =
  Lam of (lambda_expr2 -> lambda_expr2)
| App of lambda_expr2 * lambda_expr2
```

For instance, the λ -term $\lambda x.\lambda y.x(y)$ can be represented as follows.

$$Lam(fn\ x \Rightarrow (Lam(fn\ y \Rightarrow App(x, y))))$$

A name starting with a single quote stands for a type variable. The following declaration introduces a datatype constructor *rbtree* that forms the type $(\tau)rbtree$ when applied a type τ .

```
datatype 'a rbtree =
  E | R of 'a rbtree * 'a * 'a rbtree | B of 'a rbtree * 'a * 'a rbtree
```

For instance, you can now have:

```
- val x = B(E, 0, E);
val x = B (E,0,E) : int rbtree
- val y = B(E, true, E);
val y = B (E,true,E) : bool rbtree
```

6 Lists

In SML, *list* is a built-in type constructor. Given a type τ , $(\tau)list$ is the type for lists in which each element is of type τ . The constructor *nil* represents the empty list, and the infix constructor *::* (with right associativity) takes an element and a list to form another list. For instance, $1 :: 2 :: 3 :: nil$ is a list of type *int list*. Alternatively, you may write $[]$ for *nil* and $[1, 2, 3]$ for $1 :: 2 :: 3 :: nil$.

There are already various built-in functions on lists. As an example, the following program implements a function that reverses a given list.

```
fun reverse (xs: 'a list): 'a list =
  let
    fun aux ys = fn [] => ys | x :: xs => aux (x :: ys) xs
  in
    aux [] xs
  end
```

7 Arrays

There is a built-in type constructor *array* in SML, take a type τ to form the type $(\tau)array$ for arrays in which each element is of type τ . The functions *array*, *sub* and *update* are given the following types and their meaning should be obvious.

```
val array: int * 'a -> 'a array
val length: 'a array -> int
val sub: 'a array * int -> 'a
val update: 'a array * int * 'a -> unit
```

The following example implements the dot product function on two real vectors.

```

fun dotprod (v1: real array, v2: real array): real =
  let
    val n1 = length v1
    val n2 = length v2
    fun aux (i:int, s: real): real =
      if i < n1 then aux (i+1, s + sub (v1, i) * sub (v2, i))
      else s
  in
    if n1 = n2 then aux (0, 0.0)
    else raise Unequal_array_length
  end

```

Please see the next section for the use of exception in this example.

8 Exceptions

The exception mechanism in SML offers an approach to alter the normal control flow in program execution. Declaring an exception is similar to declaring a datatype.

```

exception Fatal
fun fatal (msg: string) = (print msg; raise Fatal)

```

In the above code, a call to *fatal* with an argument *msg* prints out *msg* before raising the exception *Fatal*.

The following program implements a function that tests whether a given binary tree is perfectly balanced. Notice the interesting use of exception in this program.

```

fun is_perfect (t: binary_tree): bool =
  let
    exception Not_perfect
    fun aux (E) = 0
      | aux (B (lt, _, rt)) =
        let
          val lh = aux lt
          val rh = aux rt
        in
          if lh = rh then 1 + lh else raise Not_perfect
        end
  in
    let val _ = aux (t) in true end handle Not_perfect => false end
  end

```

9 References

References are similar to pointers in C. For instance, the following declaration

```

val x = ref 0

```

allocates some memory to store the integer value 0 and then binds the memory location to x . Note that the expression $ref\ 0$ itself is *not* a value; the reference returned from evaluating the expression is a value, but such a value cannot be expressed explicitly in SML.

After the declaration, the type of x is *int ref*. You cannot change the binding of x but you can update the value stored in the memory location to which x is bound. For instance, the value is updated to 1 after the expression $x := 1$ is evaluated. In general, an expression of the form $e_1 := e_2$ evaluates e_1 to a reference r and e_2 to a value v , and then store the value v in the memory location bound to the reference r .

The dereference operator is `!`. For instance, `!e` evaluates e to a reference r and then returns the value store in the memory location bound to r .

Equality on references is the usual pointer equality. Also, you may use *ref* to form patterns. What does the following function *inc* do?

```
fun inc ((r as ref n): int ref): unit = (r := n+1)
```

The following program in SML is an implementation of the factorial function involving the use of references.

```
fun factorial (n:int): int =
  let
    val result: int ref = ref 1
    fun loop (i: int): int =
      if i > n then !result
      else (result := i * !result; loop (i+1))
  in
    loop (1)
  end
```

10 Higher-Order Functions

In general, the type of a function f is of the form $\tau_1 \rightarrow \dots \rightarrow \tau_n \rightarrow \tau$, where $n \geq 1$ and $\tau_1, \dots, \tau_n, \tau$ are types. If τ_i contains an occurrence of \rightarrow for some $i \leq i \leq n$, the f is a higher-order function. For instance, the following function *zero_finder* is a higher-order function since its type is $(int \rightarrow int) \rightarrow int$.

```
fun zero_finder (f: int -> int): int =
  let
    fun aux (i: int): int =
      if f(i) = 0 then i
      else if i > 0 then aux (~i)
      else aux (~i+1)
  in
    aux (0)
  end
```

Given a function f of the type $int \rightarrow int$, *zero_finder*(f) finds through brute-force search the first n in the list $0, 1, -1, 2, -2, \dots$ of integers such that $f(n) = 0$; it loops forever if f has no zeros.

A commonly used function *map* can be defined as follows, which applies a given function f to each element of a given list and then collects the return results in a list.

```

fun map f ([]) = []
  | map f (x :: xs) = f (x) :: map f xs

```

The type of *map* is $(\text{'a} \rightarrow \text{'b}) \rightarrow \text{'a list} \rightarrow \text{'b list}$ written in the syntax of SML.

The following function is similar to *map*, which applies a given function *f* with *unit* as its return type to each member in a given list.

```

fun foreach (f: 'a -> unit) =
  fn [] => () | x :: xs => (f (x); foreach f xs)

```

The type of *foreach* is $(\text{'a} \rightarrow \text{unit}) \rightarrow \text{'a list} \rightarrow \text{unit}$ written in the syntax of SML.

11 Continuations

In SML/NJ, there are two functions *callcc* and *throw* in the module *SMLofNJ.Cont* for supporting programming with continuations.

```

val callcc : ('a cont -> 'a) -> 'a
val throw : 'a cont -> 'a -> 'b
val isolate: ('a -> unit) -> 'a cont

```

As an example, we implement a function *comp_fun_cont* that composes a function with a continuation.

```

fun comp_fun_cont (f: 'a -> b) (E: 'b cont): 'a cont =
  callcc (fn E0 => throw E (f (callcc (fn E1 => throw E0 E1))))

```

```

fun comp_fun_cont (f: 'a -> b) (E: 'b cont): 'a cont =
  isolate (fn x: 'a => throw E (f x))

```

For instance, the factorial function can be implemented as follows in the continuation-passing style.

```

fun factorial (n: int): int =
  let
    fun aux (n: int) (E: int cont): int =
      if n = 0 then throw E 1
      else aux (n-1) (comp_fun_cont (fn (res:int) => n * res) E)
  in
    callcc (fn E => aux n E)
  end

```

Also, the previously defined function *is_perfect* can now be implemented as follows.

```

fun is_perfect (t: binary_tree): bool =
  let
    fun aux (k: bool cont) (t: binary_tree): int =
      case t of
        E => 0
      | B (lt, _, rt) =>
          let
            val lh = aux k lt
            val rh = aux k rt
          in

```

```

        if lh = rh then 1 + lh else throw k false
    end
in
    callcc (fn k => let val _ = aux k t in true end)
end

```

12 Module System

There are three major ingredients in the module system of Standard ML.

- Structures: a structure in SML is basically a collection of declarations for types, values, exceptions and structures.
- Signatures: a signature is basically a structure interface; in some sense, it is the type of a structure.
- Functors: a functor is basically a structure parameterized over structures; in some sense, it is a function from structures to structures.

12.1 Structures

```

structure S = struct
  val x = 0
  fun f (x:int): bool = (x <> 0)
  type t = bool * int list
  datatype binary_tree = E | B of binary_tree * int * binary_tree
  exception Fatal
  structure T = ...
  ...
end

```

You may refer to a component in S by appending $S.$ to the front of the name of the component. For instance, $S.x$ evaluates to 3; $S.f(S.x)$ evaluates to 4; $S.t$ is equivalent to $bool * intlist$; $S.E$ and $S.B$ have the types $S.binary_tree$ and $S.binary_tree * int * S.binary_tree \rightarrow S.binary_tree$, respectively; $S.Fatal$ is an exception; $S.T$ is a structure; ...

12.2 Signatures

Signatures provide an approach to specifying an interface to a structure. For instance, the following signature is one that the above structure S can match. However, a structure may have additional components that are not specified in a signature that the structure matches.

```

signature SIG_FOR_S = sig
  val x: int
  fun f: int -> bool
  type t
  datatype binary_tree = E | B of binary_tree * int * binary_tree
  exception Fatal
  structure T: SIG_FOR_T
end

```

```

signature RATIONAL = sig
  type rat

  val make: int * int -> rat
  val toString: rat -> string

  val zero: rat
  val one: rat

  val op~: rat -> rat
  val op+: rat * rat -> rat
  val op-: rat * rat -> rat
  val op*: rat * rat -> rat
  val op/: rat * rat -> rat

  val isZero: rat -> bool
  val isPositive: rat -> bool
  val isNegative: rat -> bool

  val op>: rat * rat -> bool
  val op>=: rat * rat -> bool
  val op<: rat * rat -> bool
  val op<=: rat * rat -> bool
end

structure Rational :> RATIONAL = struct
  type t = int * int

  exception DenominatorIsZero

  fun gcd (p: int, q: int) =
    if p = 0 then q else gcd (q mod p, p)

  fun make0 (p: int, q: int) = (* q is assumed to be positive *)
    let
      val r =
        if p > 0 then gcd (p, q) else gcd (~p, q)
    in
      (p div r, q div r)
    end

  fun make (p: int, q: int): rat =
    if q = 0 then raise DenominatorIsZero
    else if q > 0 then make0 (p, q) else make0 (~p, ~q)

  ... ..
end

```

12.3 Functors

13 Compilation Manager

14 Libraries