∼ Lecture II ∼

Keywords:

mosml; sml; value declarations; static binding; basic types (integers, reals, truth values, characters, strings); function declarations; overloading; tuples; recursion; expression evaluation; call-by-value.

References:

- [MLWP, Chapter 2]
- SML/NJ ⟨`http://www.smlnj.org/`⟩
- Moscow ML ⟨`http://www.dina.dk/~sestof/mosml.html`⟩

Running ML

```
$ mosml
Moscow ML version 2.00 (June 2000)
Enter `quit();' to quit.
-
```

```
% sml
Standard ML of New Jersey v110.57
-
```


Most functional languages are interactive:

```
- val pi = 3.14159 ;
> val pi = 3.14159 : real
- val area = pi * 2.0 * 2.0 ;
> val area = 12.56636 : real
-
```

values expressions types

A *declaration* gives something a name, or *binds* something to a name. In ML many things can be named: *values*, *types*, . . .

Static binding

If a name is redeclared then the new meaning is adopted afterwards, but does not affect existing uses of the name.

```
val pi = 3.14159 ;
val radius = 2.0 ;
val area = pi * radius * radius ;
val pi = 0.0 ;
area ;
~
```

```
"p01"
```

```
-  use"p01";
```

Read a file into ML

```
[opening file "p01"]
> val pi = 3.14159 : real
> val radius = 2.0 : real
> val area = 12.56636 : real
> val pi = 0.0 : real
> val it = 12.56636 : real
[closing file "p01"]
```

The name `it` always has the value of the last expression typed at top level.

Arithmetic

Integers and Reals

- The type of integers: `int`.

  Constants: . . . , `~2` , `~1` , `0` , `1` , `2` , . . .

  Built-in operators and functions: Try the following in ML

  ```
  - load"Int"; (* needed in mosml, but not in sml *)
  - open Int;
  ```

- The type of reals: `real`.

  Constants:

  . . . , `~1E6` , `~1.41` , `~1E~10` , `1E~10` , `1.41` , `1E6` , . . .

  Built-in operators and functions: Try the following in ML

  ```
  - load"Real";open Real;
  - load"Math";open Math;
  ```


Truth values

The type of booleans: `bool`.

Constants: `false` `true`

Built-in operators and functions:

Try the following in ML `- load"Bool";open Bool;`

Characters and strings

- The type of characters: `char`.

  Constants: `#"A"` , `#"a"` , . . . , `#"1"` , . . . , `#" "` , . . . , `#"\n"`

  Built-in operators and functions:

  Try the following in ML `- load"Char"; open Char;`

- The type of strings: `string`.

  Constants:

  `""` , `" "` , `"A"` , `"z"` , `"0 a 1 A ... 0 z 1 Z "` `"Bye, bye ... \n"`

  Built-in operators and functions:

  Try the following in ML `- load"String"; open String;`

Declaring functions

```
val pi = 3.14159 ;
fun square (x:real) = x * x ;
fun area (radius) = pi * square(radius) ;
area (0.5) ;
val pi = 0.0 ;
area 0.5 ;
~
"p02"
> val pi = 3.14159 : real
> val square = fn : real -> real
> val area = fn : real -> real
> val it = 0.7853975 : real
> val pi = 0.0 : real
> val it = 0.7853975 : real
```

(* overloading *)

`fun`, name, formal parameters, body

functions are values (`fn`) function types

Overloading

Certain built-in operators are *overloaded*, having more than one meaning. For instance, + and * are defined both for integers and reals.

The type of an overloaded function must be determined from the context; occasionally types must be stated explicitly.

```
- fun int_square (x:int) = x * x ;
> val int_square = fn : int -> int
```

NB: SML'97 defines a notion of *default type*. The SML compiler will resolve the overloading in a predefined way; relying on this is *bad* programming style.

```
- fun default_square x = x * x ;
> val default_square = fn : int -> int
```


Declaring functions

Conditional expressions

To define a function by cases —where the result depends on the outcome of a test— we employ a *conditional expression*.

```
fun sign n
  = if n>0 then 1 else if n=0 then 0 else ~1 ;
fun absval x
  = if x >= 0.0  then x else ~x ;
~
"p03"
> val sign = fn : int -> int
> val absval = fn : real -> real
```


- The boolean infix operators `andalso` and `orelse` are not functions, but stand for conditional expressions:
  - `E1 andalso E2` ≡ `if E1 then E2 else false`
  - `E1 orelse E2` ≡ `if E1 then true else E2`

Tuples

A *tuple* is an ordered, possibly empty, collection of values.

The tuple whose components are $v_1, \ldots, v_n$ ($n \geq 0$) is written $(v_1, \ldots, v_n)$.

- A tuple is constructed by an expression of the form `(E1,...,En)`.

  In particular, the *empty tuple* is given by `()` which is of `unit` type:

  ```
  - ();
  > val it = () : unit
  ```

- The components of a non-empty tuple can be *selected* (or *projected*).

- With functions, tuples give the effect of multiple arguments and/or results.

  In particular, the `unit` type is often used with procedural programming in ML. A *procedure* is typically a 'function' whose result type is `unit`. The procedure is called for its effect; not for its value, which is always `()`. For instance,

  ```
  - use;
  > val it = fn : string -> unit
  - load;  (*** in mosml ***)
  > val it = fn : string -> unit
  ```


Complex numbers

```
load"Math" ;  (* needed in mosml, but not in sml *)
type complex = real * real ;
val origin = ( 0.0 , 0.0 ) : complex ;
fun X( (x,y):complex ) = x ;
fun Y( (x,y):complex ) = y ;
fun norm v = Math.sqrt( X(v)*X(v) + Y(v)*Y(v) ) ;
fun scalevec( r, v ) = ( r*X(v) , r*Y(v) ) ;
fun normal v = scalevec( 1.0/(norm v) , v ) ;
~
"p04"
```

A type declaration


```
> val it = () : unit
> type complex = real * real
> val origin = (0.0, 0.0) : real * real
> val X = fn : real * real -> real
> val Y = fn : real * real -> real
> val norm = fn : real * real -> real
> val scalevec
    = fn : real * (real * real) -> real * real
> val normal = fn : real * real -> real * real
```

the result of evaluating `load`

Declaring functions
Infix operators

An *infix operator* is a function that is written between its two arguments.

```
infix xor ;   (* exclusive or *)
fun (p xor q)
  = ( p orelse q ) andalso not( p andalso q ) ;
true xor false xor true ;
~
"p05"
```

```
> infix 0 xor
> val xor = fn : bool * bool -> bool
> val it = false : bool
```

default *precedence* 0

In ML the keyword `op` overrides infix status:

```
- op xor;
> val it = fn : bool * bool -> bool
- op xor ( true , false ) ;
> val it = true : bool
```


Declaring functions
Recursion

Examples

- Factorial

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

- Greatest Common Divisor

```
fun gcd( m , n )
  = if m = 0 then n
    else gcd( n mod m , m ) ;
> val gcd = fn : int * int -> int
```


- Fibonacci numbers

$$F_0 = 0\ ,\quad F_1 = 1\ ,\quad F_n = F_{n-2} + F_{n-1}\ (n \geq 2)$$

```
fun nextfib( Fn , Fsuccn ) : int * int
  = ( Fsuccn , Fn+Fsuccn ) ;
fun fibpair n
  = if n = 1 then (0,1)
    else nextfib( fibpair(n-1) ) ;
> val nextfib = fn : int * int -> int * int
> val fibpair = fn : int -> int * int
```

- Power-of-two test

```
fun powoftwo n
  = (n=1) orelse
    ( (n mod 2 = 0) andalso powoftwo( n div 2 ) ) ;
> val powoftwo = fn : int -> bool
```


Mutual recursion

Examples

- π

$$\frac{\pi}{4} = 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \cdots + \frac{1}{4k+1} - \frac{1}{4k+3} + \cdots$$

```
fun pos k
  = if k < 0 then 0.0
    else ( if k = 0 then 0.0 else neg(k-1) )
           + 1.0/real(4*k+1)
and neg k
  = if k < 0 then 0.0
    else pos(k) - 1.0/real(4*k+3) ;
> val pos = fn : int -> real
  val neg = fn : int -> real
```


- Parity test

```
fun even n
  = n = 0 orelse odd( n-1 )
and odd n
  = n<>0 andalso ( n = 1 orelse even( n-1 ) ) ;
> val even = fn : int -> bool
  val odd = fn : int -> bool
```


Evaluation of expressions

Execution is the *evaluation* (or *reduction*) of an expression to its value, replacing equals by equals.

Evaluation of conditionals

To compute the value of the conditional expression `if E then E1 else E2`, first compute the value of the expression `E`. If the value so obtained is `true` then return the value of the computation of the expression `E1`; otherwise, return the value of the computation of the expression `E2`.

The evaluation rule in ML is *call-by-value* (or *strict* evaluation).

Call-by-value evaluation

> To compute the value of `F(E)`, first compute the value of the expression `F` to obtain a function value, say `f`. Then compute the value of the expression `E`, say v, to obtain an actual argument for `f`. Finally compute the value of the expression obtained by substituting the value v for the formal parameter of the function `f` into its body.

NB: Most purely functional languages adopt *call-by-need* (or *lazy* evaluation).

The manual evaluation of expressions is helpful when understanding and/or debugging programs.

Examples

1.
```
fun minORmax b
  = (if b then Int.min else Int.max)( 1+3,2 ) ;
```

```
minORmax true
 ⇝ (if true then Int.min else Int.max)( 1+3,2 )
      if true then Int.min else Int.max
      ⇝ Int.min

      (1+3,2)
          1+3 ⇝ 4
      ⇝ (4,2)
 ⇝ Int.min(4,2)
 ⇝ 2
```


2.
```
fact(1-1)
    1 - 1 ⇝ 0

    if 0 = 0 then 1 else 0 * fact(0-1)
        0 = 0 ⇝ true
    ⇝ 1
⇝ 1
```

For succinctness, the above is typically abbreviated as follows

```
fact(1-1)
⇝ fact 0
⇝ if 0 = 0 then 1 else 0 * fact(0-1)
⇝ 1
```


In this vein, thus

```
fact(3)
⇝ if 3 = 0 then 1 else 3 * fact(3-1)
⇝ 3 * fact(3-1)
⇝ 3 * fact(2)
⇝ 3 * ( if 2 = 0 then 1 else 2 * fact(2-1) )
⇝ 3 * ( 2 * fact(2-1) )
⇝ 3 * ( 2 * fact(1) )
⇝ 3 * ( 2 * ( if 1 = 0 then 1 else 1 * fact(1-1) ) )
⇝ 3 * ( 2 * ( 1 * fact(1-1) ) )
```