Standard ML

Declarations
Declarations

- Re-declaration of names in ML
- The reserved words "val rec"
- Pattern Matching
- Type Abbreviation
- Inner scope declarations
- Simultaneous declarations
Declarations

- Re-declaration of names in ML
  - The reserved words "val rec"
  - Pattern Matching
  - Type Abbreviation
  - Inner scope declarations
  - Simultaneous declarations
Example - Circle Area

\[ area = \pi r^2 \]

- Area of a circle:
  - `val pi = 3.14159;`
  - `val pi = 3.14159 : real`
  - `fun area (r) = pi*r*r;`
  - `val area = fn : real -> real`
  - `area 2.0;`
  - `val it = 12.56636 : real`
Identifiers in ML

- **val** declaration binds a name to a value.
- A name can not be used to change its value!
  - Actually a constant
- A name can be reused for another purpose
  - ```
  val pi = "pi";
  val pi = "pi" : string
  ```
- If a name is declared again the new meaning is adopted afterwards
  - ```
  pi;
  val it = "pi" : string
  ```
  but does not affect existing uses of the name
  - ```
  area(1.0)
  val it = 3.14159 : real
  ```
Is permanence of names a good feature?

- **LUCKY**: redefining a value cannot damage the system or your program.
- **BUT**: redefining a value referred by your program may have no visible effect.
- **NOTE**: when modifying a program, be sure to recompile the entire file.
Declarations

✓ Re-declaration of names in ML

➢ The reserved words "val rec"
  o Pattern Matching
  o Type Abbreviation
  o Inner scope declarations
  o Simultaneous declarations
'val' and 'val rec'

◆ We can define function using val
  - `val sq = fn x => x*x;`

◆ What about recursive functions?
  - `fun f(n) = if n=0 then 1 else n * f(n-1);`
  - `val f = fn (n) => if n=0 then 1 else n * ??;`
  - `val rec f = fn (n) => if n=0 then 1 else n * f(n-1);`

◆ 'val rec' stands for recursive definition and it is just like 'fun'
Declarations

✓ Re-declaration of names in ML
✓ The reserved words "val rec"

➢ Pattern Matching
  o Type Abbreviation
  o Inner scope declarations
  o Simultaneous declarations
Pattern Matching

- Patterns can be used to simplify function definition
  - fun factorial 0 = 1
  - factorial n = n * factorial(n-1);
  - val factorial = fn : int -> int

- When the function is called, the first pattern to match the actual parameter determines which expression on the right-hand-side will be evaluated.

- Patterns can consist
  - Constants - int, real, string, etc ...
  - Constructs - tuples, datatype constructs
  - Variables - all the rest
  - Underscore - a wildcard

Later ...
Pattern Matching

- When matching a pattern P to a value X, the matching is done recursively - "from outside to inside".
- If matching succeeded, any variable in the pattern is binded with the corresponding value in X.
- There is no binding where the wildcard is used.
- Example

```plaintext
- fun foo (x,1) = x
  | foo (1,_) = 0
  | foo _ = ~1;
val foo = fn : int * int -> int
- foo(3,1);
val it = 3 : int
- foo(1,3);
val it = 0 : int
- foo(2,2);
val it = ~1 : int
```

foo(1,1) = 1
Since matching is done in the order of definition
Patterns in Conditional Expression

Patterns can be used in a `case` conditional expression

- `case E of P1 => E1 | ... | Pn => En`
- `case p-q of
  0 => "zero"
| 1 => "one"
| 2 => "two"
| n => if n<10 then "lots"
    else "lots & lots";

- If $P_i$ is the first to match then the result is the value of $E_i$
- Equivalent to an expression that defines a function by cases and applies it to $E$
- Scope of `case`: No symbol terminates the case expression!
  - Enclose in parentheses to eliminate ambiguity
Declarations

✓ Re-declaration of names in ML
✓ The reserved words "val rec"
✓ Pattern Matching
   ➢ Type Abbreviation
     o Inner scope declarations
     o Simultaneous declarations
You can give new name to existing type:

- `type vec = real*real;
  type vec = real * real`
- `infix ++;

- `fun (x1,y1) ++ (x2,y2) : vec = (x1+x2,y1+y2);`
- `val ++ = fn: (real * real) * (real * real) => vec`
- `(3.6,0.9) ++ (0.1,0.2) ++ (20.0,30.0);`
  `(23.7,31.1) : vec`

The new name is only an alias - it is acceptable where ever the original name is acceptable and vice versa.
Declarations

✓ Re-declaration of names in ML
✓ The reserved words "val rec"
✓ Pattern Matching
✓ Type Abbreviation
   ➢ Inner scope declarations
   ▪ Simultaneous declarations
Declaration(s) inside an Expression

- let D in E end

  - fun fraction(n,d)=
    (n div gcd(n,d), d div gcd(n,d));

  val fraction = fn: int*int -> int*int

- fun fraction(n,d)=let
  val com = gcd(n,d)
  in
  (n div com, d div com)
  end;

val fraction = fn: int*int -> int*int

- D may be a compound declaration
  - D1;D2;...;Dn
- The semicolons are optional

- "let D in E end" Can be simulated using anonymous functions
  - fun fraction(n,d)=
    (fn com => (n div com, d div com))(gcd(n,d));
Nested Scopes

ML allows nested function definitions

```ml
fun sqroot a = 
  let
    val acc = 1.0e-10
    fun findroot x = 
      let
        val nextx = (a/x + x)/2.0
      in
        if abs (x - nextx) < acc * x
          then nextx else findroot nextx
      end
    in
      findroot 1.0
    end;
val sqroot = fn: real -> real
```
Declaration(s) inside a Declaration

- **local** \( D_1 \text{ in } D_2 \text{ end} \)
  - Behaves like the list \( D_1; D_2 \text{ in } \text{let} \)
  - \( D_1 \) is visible only within \( D_2 \)

- **Used to hide a declaration**
  
  ```
  local
  fun itfib (n,prev,curr):int = 
  if n=1 then curr 
  else itfib (n-1,curr,prev+curr) 
  in 
  fun fib (n) = itfib(n,0,1) 
  end;

  val fib = fn : int -> int
  ```
Comparing `let` and `local`

- `fun fib (n) = let
  
  fun itfib (p,prev,curr):int=
    if p=1 then curr
    else itfib (p-1,curr,prev+curr)
  
  in
  itfib(n,0,1)
  end;

  `val fib = fn : int -> int`

- `local

  fun itfib (p,prev,curr):int=
    if p=1 then curr
    else itfib (p-1,curr,prev+curr)
  
  in
  fun fib (n) = itfib(n,0,1)
  end;

  `val fib = fn : int -> int`
Declarations

- Re-declaration of names in ML
- The reserved words "val rec"
- Pattern Matching
- Type Abbreviation
- Inner scope declarations
  - Simultaneous declarations
Simultaneous Declarations (collateral)

- `val Id1 = E1 and ... and Idn = En`
  - evaluates `E1,...,En`
  - and only then declares the identifiers `Id1,...,Idn`

- Example: Swapping the values of names
  - `val x = y and y = x`
  - `val (x,y) = (y,x)`

- Note the last declaration. Actually the allowed format is
  - `val P = E;`

- So it can be used to disassemble tuples
  - `val a = (1,2,3);`
  - `val a = (1,2,3) : int * int * int`
  - `val (_,x,_) = a;`
  - `val x = 2 : int`
Mutually Recursive functions

Example:

\[
\pi/4 = \sum_{k=0}^{\infty} \frac{1}{4k + 1} - \frac{1}{4k + 3} = 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - \frac{1}{11} \ldots
\]

```haskell
fun pos d = neg(d-2.0) + 1.0/d
and neg d = if d>0.0 then pos(d-2.0)-1.0/d
            else 0.0;

fun sum(d,one)=
   if d>0.0 then sum(d-2.0,~one)+one/d
     else 0.0;
```
Translating an imperative code to mutually recursive functions

- Emulating `goto` statements...

```markdown
var x:=0; y:=0; z:=0;
F: x:=x+1; goto G
G: if y<z then goto F else (y:=x+y; goto H)
H: if z>0 then (z:=z-x; goto F) else stop
```

```ml
- fun F(x,y,z)=G(x+1,y,z)
  = and G(x,y,z)=if y<z then F(x,y,z) else H(x,x+y,z)
  = and H(x,y,z)=if z>0 then F(x,y,z-x) else (x,y,z);
val F = fn : int * int * int -> int * int * int
val G = fn : int * int * int -> int * int * int
val H = fn : int * int * int -> int * int * int
  - F(0,0,0);
val it = (1,1,0) : int * int * int
```
Declarations

✓ Re-declaration of names in ML
✓ The reserved words "val rec"
✓ Pattern Matching
✓ Type Abbreviation
✓ Inner scope declarations
✓ Simultaneous declarations