# Principles of Programming Languages
# Lecture 12


***Wael Aboulsaadat***

**wael@cs.toronto.edu**

http://portal.utoronto.ca/



Acknowledgment: parts of these slides are based on material by Diane Horton & Eric Joanis @ UoT


References: Scheme by Dybvig; PL Concepts and Constructs by Sethi; Concepts of PL by Sebesta; ML for the Working Prog. By Paulson; Prog. in Prolog by Clocksin and Mellish; PL Pragmatics by Scott

# ML: introduction

- **Developed at Edinburgh (early '80s) as Meta-Language for a program verification system**
  - Now a general purpose language
  - There are two basic dialects of ML
    - Standard ML (1991) & ML 2000
    - Caml (including Objective Caml, or OCaml)

- **A pure functional language**
  - Based on *typed lambda calculus*
  - Grew out of frustration with Scheme!
  - Serious programs can be written without using variables

| | |
|---|---|
| 1930's | Lambda calculus (Church) |
| 1950's | Lisp (McCarthy) |
| 1960's | semantics, deconstruction |
| 1970's | FP (Backus) |
| 1980's | Miranda (Turner), ML (Milner) |
| 1990's | Haskell |

- **Widely accepted**
  - reasonable performance (claimed)
  - can be compiled
  - syntax not as arcane as Scheme

# ML: main features

- **Strong, static typing**
  - Quite a fancy type system!
- **Parametric polymorphism**
  - Similar to OOP (*in fact, it influenced OO*)
- **Pattern matching**
  - Function as  a template
- **Exception handling**
  - Allow you to handle errors/exception
- **Type inference**
- **Recursive data type**

# ML: how far have PL advanced?

- **Writing a gcd implementation**

$$\gcd(m,n) = \begin{cases} n & m = 0 \\ \gcd(n \bmod m, m) & m > 0 \end{cases}$$

**Pascal**

```
function gcd(m,n: integer): integer;
var prevm: integer;
begin
  while m<>0 do begin
        prevm := m; m := n mod m; n := prevm
  end;
  gcd := n
end;
```

**Scheme**

```
(define gcd
 (lambda (m n)
      (if (zero? n) m
          (gcd n (remainder m n)))))
```

**ML**

```
fun gcd(m,n) =
    if m=0 then n  else
         gcd(n mod m, m);
```

# ML: types & expressions

- **Primitive types**
  - bool, int, real, string

- **Complex types**
  - list, tuple, array, record, function

- **Each ML expression has a type associated with it.**
  - Interpreter builds the type expression for each input
  - Cannot mix types in expressions

    2+3.0 ➔ error!

    - Must explicitly coerce/type-case e.g.

      real(2) + 3.0 : real

# ML: Primitive Types

- **int** e.g. x: int;
  - Negative sign uses ~
  - Operators: + - * div mod
- **real** e.g. x: real;
  - 3.45 or using e notation (3E7)
  - Operators: + - * /
  - Conversion functions: *real(integer), floor(real), abs(x)*
- **string** e.g. s: string;
  - Delimited by double quotes
  - Caret ^ is concatenation e.g. "house" ^ "cat"
  - Function size returns length of string
  - Special characters: \n \t \" \\
- **bool** e.g. b: bool;
  - **true (and false)**

# ML: operators

- **All operators are infix**

- **Logical operators:**
  - Short-circuit evaluation
  - if … then … else is an expression, not a control structure…

| | |
|---|---|
| not | *Negation* |
| andalso | *Conjunction* |
| orelse | *Disjunction* |
| if … then … else | *conditional selection* |

- **Numeric operators:**
  - The usual <, >, <=, >= and <> are available
  - For reals, = and <> are not available ( a <= b andalso a >= c )
  - For strings, these can be used for lexicographic ordering

- **Operator overloading:**
  - Same symbol could be used for operations that are internally dissimilar
  - *, +, - , <, <=, >, >= are all overloaded
  - Leftmost argument is inspected first to decide on type

# ML: assignment

- **Use val to assign value to variables**

  val <indentifier-name> = <expression>;

- **Examples:**

```
- val seconds = 60;
> val seconds = 60 : int

- val minutes = 60;
> val minutes = 60 : int

- val tm = seconds * minutes;
> val tm = 86400 : int

- val shout = "aaa" ^ "rgh" ^ "!!!! ";
> val shout = “aaargh!!!” : string
```

# ML: constructor types - lists

- **Syntax** **[ $obj_1$, $obj_2$, … ]**
  - Objects in a list must be homogenous (*same type*)
    - E.g.

| | | |
|---|---|---|
| [1,2,3] | : int list | -- a list of integers |
| ["dog", "cat", "moose"] | : string list | -- a list of strings |
| [1.0,2.0,3.0] | : real list | -- a list of reals |
| [[1,2],[3,4],[5,6]] | : int list list | -- a list of lists of integers |

  - The empty list is written [] or nil

- **Operations:**
  - @ operator is used to concatenate two lists of the same type
  - :: operator returns a new list with the first argument append to the front
    - E.g. 2 :: [3,4] returns [2,3,4]
      [1,2]::[ [3,4], [5,6]] returns ??
  - hd returns the first element of a list
    - E.g. hd[1,2,3] returns 1
  - tl returns the tail
    - E.g tl[1,2,3] returns [2,3]

# ML: constructor types - tuples

- **Syntax** **( $obj_1$, $obj_2$, … )**
  - Objects in a tuple can be heterogeneous (*different types*)
    - E.g.

      (2, "abc") : int * string
      (2,3.0, "abc") : int * real * string
      (2,(3.0, "ab"), "cd") : int * (real * string ) * string
      [(1, "a"),(3, "bc"),(7, "efg")] : (int * string) list
  - The empty tuple is written () and often called unit
  - Composite format of a tuple can be used on left-hand side of val
    - Eg. - val (day, month, year ) = (13,"March",1066);

- **Operations:**
  - # operator to extract the ith field of a tuple

    - #2(6,7,"abc") returns 7
    - #3(6,7,"abc") returns abc
  - = and <> operators for equality/in-equality

    - val x = ~3; // -3
    - (3,"a",true) = (abs x, "a", (3 > 2));

# ML: constructor types - functions

- **Syntax** **fun <func-name> <input-param> = <expression>;**
  - Keyword fun starts the function declaration
  - Function arguments don’t always need parentheses, doesn’t hurt to use them

- **Examples:**
  - fun fahrToCelsius f =
    ```
    (f -freezingFahr) * 5 div 9;
    ```
  - fun foo L =
    ```
    (1 + hd L) :: (tl L);
    ```

# ML: constructor types - functions

- **Functions have types too. ML interpreter will infer the type.**
  - E.g. fun square x =
    x * x;
    - The function square takes an integer as input and returns an integer as output.
    - This is written as square: int→int (→ *indicates this a function*)

# ML: constructor types - functions

- **ML figures out the input and/or output types for simple expressions, constant declarations, and function declarations**
  - Type checking requires that type expression of functions and their arguments match, and that type expression of context match result of function
  - If the default isn't what you want, you can specify the input and output types

```
fun divideBy2 (y : real) =
        y / 2.0;
```

- **What is this doing?**

```
fun foo (m, n) =
    if m > n then
        [ ]
    else m :: foo(m+1, n);
```

```
> foo(1,6);
```

# ML: constructor types - functions

- **Examples:**
  - Factorial (n!)

```
fun fact n =
    if n = 0 then
        1
     else
        n * fact(n-1);
```

fun 4 → 4 * (fact 3)
→ 4 * 3 * (fact 2)
→ 4 * 3 * 2 (fact 1)
→ 4 * 3 * 2 * 1 * (fact 0)
→ 4 * 3 * 2 * 1 * 1
→ 24

  - List reverse

```
fun reverse L =
    if L = nil then
        nil
    else
        reverse(tl L) @ [hd L];
```

reverse [1,2,3] → reverse[2,3] @ 1
→ reverse[3] @ 2 @ 1
→ reverse[] @ 3 @ 2 @ 1
→ [] @ 3 @ 2 @ 1
→ [3,2,1]

# ML: local environment using let

- **Syntax**

```
let
   val <variable_1> = <expression_1>;
    …..
   val <variable_n> = <expression_n>;
in
   <expression>
end;
```

- **Let allows declarations to be used in expressions**
- **Similar to Let* in Scheme**
- **Example:**
  - Compute hundredth power of a number

```
fun hundredthPower(x : real ) =
    let
       val four    = x * x * x * x;
       val twenty = four * four * four * four * four;
    in
       twenty* twenty* twenty* twenty* twenty
    end;
```

# ML: pattern matching

- **Syntax**

```
fun  <func> <pattern_1>  = <expression_1>
 |   <func> <pattern_2>  = <expression_2>
 …….
 |   <func> <pattern_n>  = <expression_n>
```

- **Define a function by a series of equations, LHS is a pattern.**
  - Always put the most specific pattern first
  - ML interpreter will use the first equation whose LHS matches

- **Example:**
  - Fibonacci function ($a_n = a_{n-1} + a_{n-2}$ :- 0 , 1, 1, 2, 3, 5, 8, 13, 21,…)

```
fun fib n =
      if n = 0 then 0
         else if n = 1 then 1
               else fib(n-1) + fib(n-2);
```

```
fun fib(0)= 0
|   fib(1)= 1
|   fib(N)= fib(N-1) + fib(N-2);
```

- **Pattern matching is powerful:**
  - Allows the programmer to see the arguments. No more hd’s and tl’s.

# ML: pattern matching – cont'd

- **Examples:**
  - Sum all the elements in a list of integers

```
fun listsum L =
    if (null L) then 0
      else (hd L) + listsum(tl L);
```

listsum[1,2,3,4]➔ 1+ listsum[2,3,4]
1+2+listsum[3,4]
1+2+3+listsum[4]
1+2+3+4+listsum[]
1+2+3+4+0
10

**Better Version**

```
fun listsum [] = 0
|     listsum L = (hd L) + listsum(tl L);
```

**Even Better!**

```
fun listsum []      = 0
|     listsum(h::t) = h + listsum(t);
```

# ML: pattern matching – cont'd

- **Examples:**
  - Reversing a list

```
fun reverse L =
    if L = nil then nil
      else reverse(tl L) @ [hd L];


 fun reverse(nil) = nil
 |     reverse(h::t) = reverse(t) @ [h];
```

reverse [1,2,3] ➔ reverse[2,3] @ 1
➔ reverse[3] @ 2 @ 1
➔ reverse[] @ 3 @ 2 @ 1
➔ [] @ 3 @ 2 @ 1
➔ [3,2,1]

# ML: pattern matching – cont'd

- **Examples:**
  - Return first n elements of a list

```
fun take ([ ], Index)     = [ ]
|     take (h::tl, Index)  =
          if Index > 0 then
               h::take(tl, Index - 1)
          else
               [ ];
```

take( [1,2,3] , 2) → 1 :: take ([2,3], 1)
→ 1 :: 2 take([3], 0)
→ 1 :: 2 :: []
→ [1, 2]