

# Mini-ML

---

CS 502

Lecture 2

8/28/08

# ML

---

- This course focuses on compilation techniques for functional languages
    - Programs expressed in Standard ML
    - Mini-ML (the source language) is an expressive core subset of SML.
    - Implementation will need to use modules and structures.
  - Functional programming
    - immutable data structures
    - recursion and functional call as the primary control structure
    - heavy use of higher-order functions
    - Contrast with imperative and object-oriented languages
-

# Expressions

---

- Programs are expressions. The meaning of a program is the value of the expression:

```
# 16 + 18 ;  
    val it = 34 : int
```

```
# 2 * 8 + 3 * 6 ;  
    val it = 34 : int
```

---

# Names

---

## Introduce bindings using let:

```
# let val v = 200 in v + 1 end;  
val it = 201 : int
```

```
# let val y = it * 3 in y end;  
val it = 603 : int
```

Bindings cannot refer to themselves in definitions:

```
# let val z = z + 1 in z end;; (* illegal *)
```

# Functions

---

```
# val double = fn (x:int) => x + x
```

```
val double = fn: int -> int
```

```
# double 3;
```

```
val it = 6: int
```

- $x$  is the parameter of the function. The expression  $x * x$  is the body.
  - The expression "double 3" is the application of the function
  - The type of the function is "int -> int"
-

# Functions

---

```
# val sumsquare = fn (x:int) => fn (y:int) => x*x + y*y
val sumsquare = fn: int -> int -> int
    (* type annotations required in Mini-ML, but not ML *)
# sumsquare 3 4;
val it = 25 : int
# sumsquare 3;
val it = fn: int -> int
# fun sumsquare x y = x * x + y * y
    (* in Mini-ML would write:
        fun sumsquare(x:int): int -> int =
            fn(y:int) => (x*x) + (y*y)  *)
val sumsquare = fn: int -> int -> int
# fun sumsquare (x,y) = x * x + y * y
    (* in Mini-ML would write:
        fun sumsquare(p: (int * int)):int =
            (case p of (x,y) => (x * x) + (y * y)) *)
val sumsquare = fn: int * int -> int
```

---

# Booleans and Conditionals

---

There are only two values of type Boolean: true and false

```
# 1 = 2;  
val it = false : bool  
# not (5 <= 10);  
val it = false : bool
```

In Mini-ML true and false represented as constructors of a bool datatype:

```
datatype bool = true | false
```

Conditional expressions defined as usual:

```
# if 3 < 4 then 7 else 100;  
val it = 7 : int  
# if false then (3 + 3) else 10;  
val it = 10 : int  
# if true then 10 else false;  
Error: types of rules don't agree
```

---

# Inductive and Recursive Definitions

---

```
# fun sum 0 = 0
  | sum n = n + sum(n-1)
- val sum = fn: int -> int
# sum 3
- val it = 6 : int      (* Can't write this in Mini-ML: instead write *)
```

```
fun sum (n:int):int =
  case n of
    0 => 0
  | n => n + sum(n-1)
```

Or write:

```
# fun sum(n) = if n = 0
  then 0
  else n + sum(n-1)
- val sum = fn: int -> int
# sum 3;
- val it = 6 : int
```

---

# Lists

---

- Lists store a collection of homogeneous data values.
- The empty list is written as `[]`; `hd` and `tl` used to deconstruct lists

```
# [1,2,3];  
val it = [1,2,3] : int list  
# let val l = [1,2,3] in [hd(l),hd(tl(l))] end;  
val it = [1,2] : int list  
# let val l = [1,2,3] in [hd(l),tl(l)] end;  
???
```

- Lists can be built from any of type
- Can build list of lists:

```
# [ [1,2], [1,2,3] ]  
- val it = [ [1,2], [1,2,3] ] : int list list
```

Mini-ML only supports integer lists; must construct lists constructively.

```
datatype intlist = Cons of int * intlist | Nil  
# Cons(1,Cons(2,Cons(3,Nil)))
```

---

# Lists

---

- In ML, can construct lists using cons:

```
# 1 :: [2,3]
val it = [1,2,3] : int list
```

- Functions to build lists:

```
# fun map(f, []: int list) = []
    | map(f:int->int, x::l) = f(x)::map(f,l)
val map = fn: (int -> int) * int list -> int list
# map(fn x => x + 1, [1,2,3])
val it = [2,3,4]: int list
```

- What would happen if we removed the type declarations on f and []?
-

# Example

---

Reverse a list:

```
# fun snoc(x:int,y) = if y = []
                        then [x]
                        else hd(y)::snoc(x,tl(y))
val snoc = fn: int * int list -> int list
# fun reverse(l: int list) =
    if l = []
    then []
    else snoc( hd(l), reverse(tl l))
```

Why is this inefficient?

---

# Example

---

```
# fun reverse l: int list =  
  let fun aux([], result) = result  
      | aux(l', result) =  
          aux(tl(l'), hd(l')::result)  
  in aux l []  
  end  
- val reverse = fn: int list -> int list
```

Why is this better?

---

# Tail Recursion

---

- Express loops using tail recursion: A tail recursive function is one in which the result of every control-path is defined in terms of another function call.

```
# fun fact 0 = 1
  | fact n = n * fact(n-1)
```

This is not in tail form.

```
# fun fact n =
  let fun aux(0,r) = r
        | aux(n,r) = aux(n-1,r*n)
  in aux(n,1)
  end
```

This is in tail form. Why is this better?

---

# Type Inference & Polymorphism

---

```
# fun fact 0 = 1
  | fact n = n * fact (n - 1)
val fact = fn: int -> int
```

Compiler deduces that  $n$  must be an integer, and that `fact` is a function over integers

```
# fun map(f, []) = []
  | map(f, x::l) = f(x)::map(f, l)
val map = fn: ('a -> 'b) * 'a list -> 'b list
```

No polymorphic types (for now) in Mini-ML. 'a and 'b are type variables

The inferred type tells us that `f` operates over something of type 'a and returns something of type 'b, and that `l` is a list of type 'a and `map` returns a list of type 'b

---

# Pattern matching and type checking

---

```
fun reverse (l, r) = if l = nil
                    then r
                    else reverse(tl(l), hd(l) :: r)
```

```
reverse([1,2,3],[[]]) → [3,2,1]
```

What are the implications of comparing `l` with `nil`?

What is the type of `reverse`?

```
val reverse: fn: 'a list * 'b list -> 'b list
```

Assumes that `'a` is an equality type. What happens on the following call?

```
reverse([floor,trunc,ceil])
```

---

# Pattern matching and type checking

---

Now consider:

```
fun reverse(nil) = nil
  | reverse(x::xs) = reverse(xs) @ [x]
```

What is the type of reverse?

➤ val reverse: fn : 'a list -> 'a list

What happens if we evaluate:

➤ reverse([floor, trunc, ceil])

What about:

➤ reverse(fn a => a, fn b => b+1)

The type checker will attempt to instantiate polymorphic types to satisfy the context in which polymorphic functions are used. The type of this expression is therefore  
(int -> int) list

---

Or:

```
fun reverse(l,r) = case l of
  [] => r
  | x::y => reverse(y,[x]@r)
```

val reverse = fn : 'a list \* 'a list => 'a list

# Data Types

---

- Can create new inductively defined, recursive datatypes
- A powerful programming tool that is useful in preventing errors
  - Example: we have circles and squares, both represented in terms of coordinates:
    - circles: center and radius
    - square: bottom left corner and width
    - both represented as a triple of floats
    - how do we prevent accidentally mistaking a square for a circle?
- Mini-ML provides three built-in datatypes:
  - datatype bool = true | false
  - datatype intoption = NONE | SOME of int
  - datatype intlist = Nil | Cons of int \* intlist
- Mini-ML doesn't support the creation of new datatypes

# Datatypes

---

```
# datatype shape =  
    Circle of real * real * real  
  | Square of real * real * real
```

Creates two constructors of type shape  
named *Circle* and *Square*

```
# Square(1.0,2.0,3.0);  
- val it = Square(1.0,2.0,3.0) : shape
```

---

# Datatypes

---

Use pattern matching to extract elements of a datatype:

```
# fun areaOfSquare s =  
    case s of  
        Square(_,_,z) => z * z  
    | _ => raise Fail "not a square"  
# fun areaOfSquare(Square(_,_,z)) = z * z
```

Constructors behave as both patterns and functions. Identified by being capitalized.

---

# Datatypes

---

```
# fun areaOf s =  
    (case s of  
        Square(_,_,z) => z * z  
        | Circle(x,y,r) = 3.14 * r*r)  
- val areaOf = fn: shape -> real
```

---

# Recursive types

---

Consider the language of arithmetic expressions:

$exp :: \text{number}$   
|  $(exp + exp)$   
|  $(exp - exp)$   
|  $(exp * exp)$

---

# Recursive types

---

- We can translate this grammar directly into a datatype definition

```
datatype ast = Num of int
             | Plus of ast * ast
             | Minus of ast * ast
             | Times of ast * ast
```

Like the grammar, this datatype is recursive

Surface details about the syntax (e.g., brackets) have been omitted.

---

# Using abstraction

---

```
fun ('elem, 'state)
  fold (lst : 'elem list,
        istate: 'state,
        folder: 'elem * 'state ->
        'state : 'state =
    let fun loop (lst, state) =
        case lst of
            nil => state
          | h::t = loop(t, folder(h, state))
        in loop (lst, istate)
        end
    # fold([1,2,3],0,fn(i,s) => i+s)
  val it = 6:int
```

The type variables 'elem and 'state indicate that this function is polymorphic over its list and internal state.

They can be omitted.

---

# Using fold

---

```
fun ('a, 'b)
  map (lst: 'a list, f: 'a -> 'b): 'b list =
    fold(lst,
          [],
          fn (e, ac) => ac @ [f (e)])
```

```
# map([1,2,3], fn a => a+1)
[2,3,4]
```

---

# Value restriction

---

Value Restriction: prevents references from holding values of different type.

```
val r: 'a option ref = ref NONE
val r1: string option ref = r
val r2: int option ref = r
val () = r1 := SOME "foo"
val v: int = valOf (!r2)
```

First line violates the restriction because `ref NONE` is not a value. What about:

```
val f: unit -> 'a option ref = fn () => ref NONE
val r: 'a option ref = f ()
b
```

# Value restriction

---

Can be unnecessarily restrictive:

```
val id: 'a -> 'a = fn x => x
val f = id id
```

However, providing enough context would succeed:

```
val id: 'a -> 'a = fn x => x
val f = id id
val _ = f 13
```

But, cannot use *f* polymorphically:

```
val id: 'a -> 'a = fn x => x
val f = id id
val _ = f 13
val _ = f "foo"
```

# Value restriction

---

What happens in the following:

```
val f: 'a -> 'a =
  let
    val r: 'a option ref = ref NONE
  in
    fn x =>
      let
        val y = !r
        val () = r := SOME x
      in
        case y of
          NONE => x
        | SOME y => y
      end
    end
end
val _ = f 13
val _ = f "foo"
```