Introduction to ML

CSCI3180 Principles of Programming Languages

Spring 2018
Sunny Lai

Introduction of ML

- Features
- Data Types
  - primitive type, composite type
- Tuple
- List
- Function
  - Recursive Function
  - Anonymous function / Lambda
- Let
- Union Type

Standard ML of New Jersey

- Developed by various parties including Princeton University.
- Functional programs are made up of functions applied to data
- expressions rather than command

How to use SML

- To invoke SML, type
  - `> sml`
  - `CSE Unix platform`
- To load the code in the file “my.ml”
  - `- use "my.ml";`
- Remember end of statement (;)
- `Ctrl-D` to exit

Functional Programming Language

- ML as a programmable calculator

```
- length [1, 2, 3, 4, 5];
val it = 5 : int
- "house" ^ "cat";
val it = "housecat" : string
```

refers to last value computed

- Referential transparency
  - Values NOT changed
  - NO storages, NO reassignments

Functional Programming Language

- Example:
  - `val x=3;`
  - `fun addx(a)=a+x;`
  - `val x=10;`

- `addx(2);` ← what's the result?

12?   5?

```
val it = 5 : int
```

NO reassignments

Type

- ML is a strongly typed language.
  - primitive type, composite type

- Primitive Type

int	:	3, ~4 (~ minus/negative)
real	:	3.5, ~9.4
string	:	"Jimmy"
char	:	#"c"
bool	:	true, false

Type

- Composite Type
  - Tuple
  - List
  - Function
  - Union

Type checking/casting

In ML

```
- 2.5 + 1;
stdIn:34.1-34.8 Error: operator and
operand don't agree [literal]
  operator domain: real * real
  operand:         real * int
  in expression:
    2.5 + 1
```

Type checking

Type Casting

```
- 2.5 + real(1);
val it = 3.5 : real
```

In Python

```
>>> 2.5 + 1
3.5
```

Tuple

- Fixed number of components, possibly mixed typed
- Enclosed by parenthesis

```
(true,3.5,"x") : bool*real*string
((4,2),(7,3))  : (int*int)*(int*int)
```

List

- Sequence of identically typed components be of any length
- Enclosed by square brackets

```
["Andrew","Ben"]      : string list
[(2,3),(2,2),(9,1)] : (int*int) list
[[],[1],[1,2]]        : int list list
```

List

- nil is the empty list
- a::b = head item a + tail list b

nil	[]
1::nil	[1]
2::(1::nil)	[2,1]
3::2::1::nil	[3,2,1]
4::3::2::1	Error

List - built-in functions

- `a@b` = concatenation of 2 lists `a, b`

- `hd(L)` = 1st element (head) of `L`
- `tl(L)` = List without head of `L`
- `null(L)` is true if `L = nil`
- `length(L)` = number of elements in `L`
- `rev(L)` = reverse of `L`

Function

○ Function Type

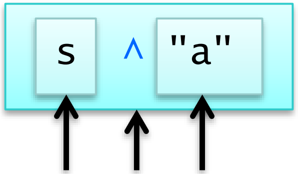
- parameter type -> return type

```
- fun adda s = s^"a";
val adda = fn : string -> string
```

how do we know?

○ Type Inference

- automatically deduce (partially or fully) the type of an expression at compile time
- e.g. ^ is a function on strings => adda type



all are strings

Function

explicitly define parameter type and return type

- Type declaration

```
- fun double(x:int):int = 2*x;
val double = fn : int -> int;
```

- Type inference

```
- fun double(x) = 2*x;
val double = fn : int -> int;
```

- Polymorphism

```
- length;
val it = fn : 'a list -> int
```

Function

- Single parameter function:

```
fun adda s = s^"a";
```

- Multiple parameter function:
  - Using tuple to include all parameters

```
- fun add(x,y):int = x+y;
val add = fn : int*int->int;
```

Recursive Function

required in assignment

- Pattern matching + mutual recursion

```
- fun length nil = 0
=   | length (h::t) = 1 + length(t);
```

- Other than pattern matching: if-then-else expression

```
- fun length list =
= if null(list)
= then 0
= else 1 + length (tl list);
```

Anonymous function / Lambda

- define function that is used only once and won’t be referred later.
  - It makes perfect sense that the function need not to be named, i.e. anonymous.

```
In ML
- (fn (x,y) => (2*x,3*y)) (2,3);
val it = (4,9) : int * int
```

```
In Python
>>> (lambda (x, y): (2*x, 3*y))((2, 3))
(4, 9)
```

Function - example: Fibonacci

- define a function fib: int -> int
- given n, return the nth Fibonacci number

```
fun fib 0 = 0 (* Base case *)
  | fib 1 = 1 (* Base case *)
  | fib n = fib(n - 1) + fib(n - 2)(* Recursive case *)
```

Pattern matching
(Case matching)

1 , 1 , 2 , 3 , 5 , 8 , 13 , 21 , 34 , 55 , 89 , 144

Let

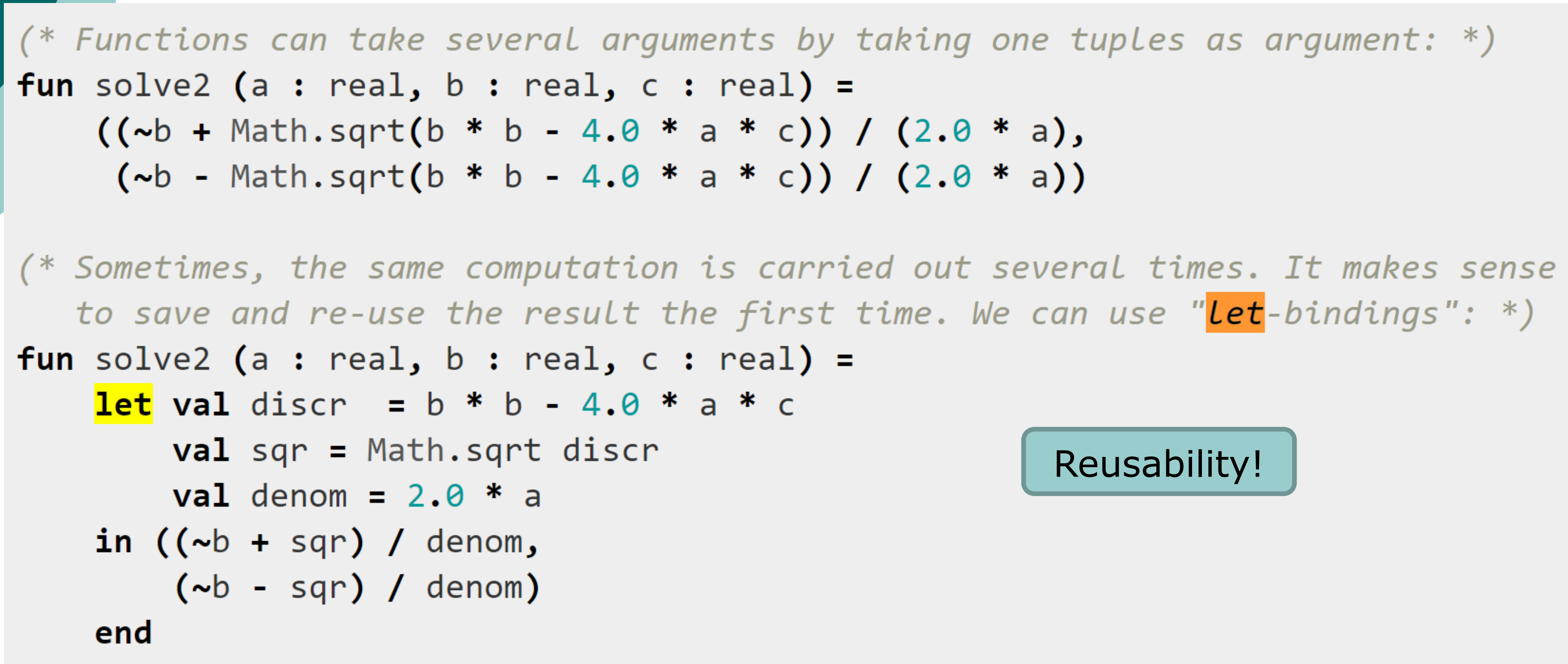
Variables are only bound within a certain scope. Outside of that region that binding does not apply.

```
(* x not bound here *)
  let
    val x = 1
  in
    x + 5
  end
(* x not bound here *)
```

```
let val x = 5
  in
    (let
        val x = 6
     in
        x + 7 (* this x = 6 *)
     end) + x (* this x = 5 *)
end
```

Refer to: http://www.cs.cornell.edu/courses/cs312/2004fa/lectures/rec05.txt

Let - examples



```
(* Functions can take several arguments by taking one tuples as argument: *)
fun solve2 (a : real, b : real, c : real) =
    ((~b + Math.sqrt(b * b - 4.0 * a * c)) / (2.0 * a),
     (~b - Math.sqrt(b * b - 4.0 * a * c)) / (2.0 * a))

(* Sometimes, the same computation is carried out several times. It makes sense
   to save and re-use the result the first time. We can use "let-bindings": *)
fun solve2 (a : real, b : real, c : real) =
    let val discr  = b * b - 4.0 * a * c
        val sqr = Math.sqrt discr
        val denom = 2.0 * a
    in ((~b + sqr) / denom,
        (~b - sqr) / denom)
    end
```

Reusability!

https://learnxinyminutes.com/docs/standard-ml/

Let - examples

```
fun fib 0 = 0 (* Base case *)
  | fib 1 = 1 (* Base case *)
  | fib n = fib(n - 1) + fib(n - 2)
(* Recursive case *)
```

```
fun fib n =
  let
      fun fibi (a,b,0) = a
        | fibi (a,b,n) = fibi (b,(a+b),(n-1))
  in
      fibi (1,1,n)
  end;
```

local declarations

1 , 1 , 2 , 3 , 5 , 8 , 13 , 21 , 34 , 55 , 89 , 144

Union Type

- Definition:

```
datatype money = cash of int | cheque of
string * real;
```

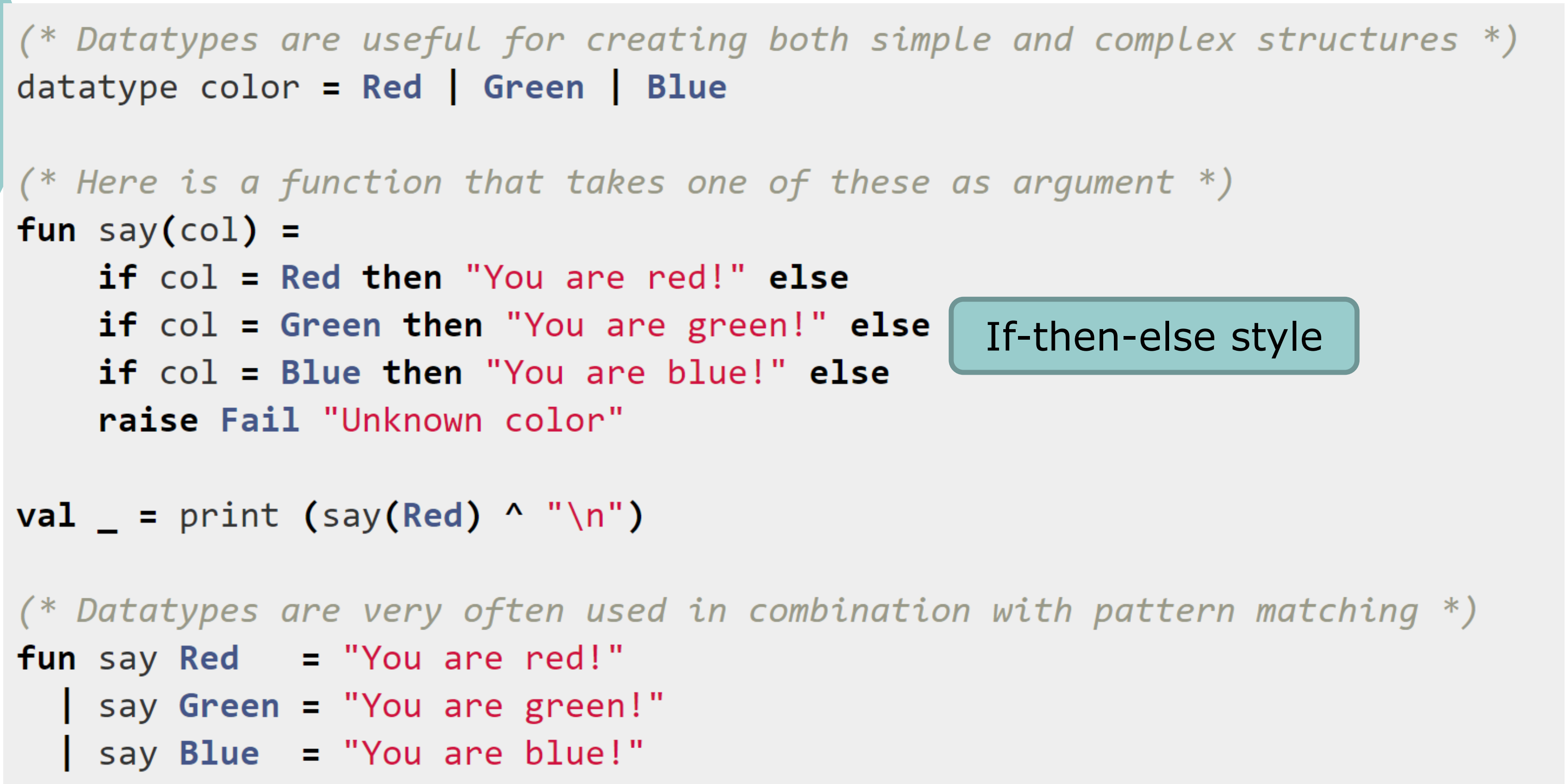
- Usage:

```
val lunch = cash 45;
val car   = cheque(“HSBC",36500.0);
```

- Pattern matching:

```
- fun worth(cash x) = real x
=  |  worth(cheque(“HSBC”,amt)) = 0.9*amt
=  |  worth(cheque(_,_)) = 0.0;
```

Union Type



```
(* Datatypes are useful for creating both simple and complex structures *)
datatype color = Red | Green | Blue

(* Here is a function that takes one of these as argument *)
fun say(col) =
    if col = Red then "You are red!" else
    if col = Green then "You are green!" else
    if col = Blue then "You are blue!" else
    raise Fail "Unknown color"

val _ = print (say(Red) ^ "\n")

(* Datatypes are very often used in combination with pattern matching *)
fun say Red   = "You are red!"
  | say Green = "You are green!"
  | say Blue  = "You are blue!"
```

If-then-else style

Pattern matching

https://learnxinyminutes.com/docs/standard-ml/

Union Type - bTree

- Syntax

```
datatype 'a bTree = nil | bt of 'a bTree*'a*'a bTree
```

polymorphism: type

tuple

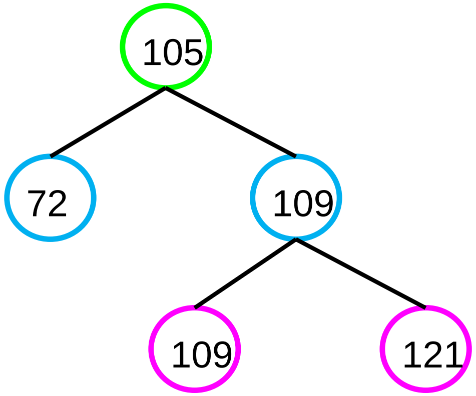
- Semantic

nil: null leave

bt(*'a bTree*, *'a*, *'a bTree***)**: left subtree, content, right subtree

- Example ('a now is int)

```
bt(bt(nil, 72, nil),
         105,
         bt(bt(nil,109,nil),
                  109,
             bt(nil,121,nil)
         )
)
```



Useful learning material

- A gentle introduction to ML
  - http://www.soc.napier.ac.uk/course-notes/sml/
- Notes on programming SML/NJ
  - http://www.cs.cornell.edu/riccardo/prog-smlnj/notes-011001.pdf
  - https://learnxinyminutes.com/docs/standard-ml/ (cheat sheet/examples)

Good luck for your final exam!