CS421 Lecture 2

- Reminder: Office Hours now posted on web page
- Midterm dates: Feb. 24, April 5 (Wednesday nights)

- Today's class: Ocaml:
  - Types
  - let expressions
  - Scope rules
  - Tuples & pattern-matching
  - Lists & pattern-matching

Ocaml

- Functional language – rely on expression evaluation rather than statement execution
  - Heavy use of recursion
  - Type inference
  - Dynamic memory allocation
  - "Higher-order functions" (will cover in second half of the course)

Types

- Basic: int, string, …
- Function: $\tau_1 \rightarrow \tau_2 \rightarrow \ldots \rightarrow \tau_n \rightarrow \tau$
  - e.g. int → int → int
- Later in this class: tuples, lists

Let expressions

- At "top level," use let to define variables and functions
- Use "let rec" for recursive definitions, e.g.:

```
let rec sumsqrs m =
    if m=0 then 0 else m*m + sumsqrs (m-1) ;;
```

Nested let definitions

```
let f x y = let z = sqrt(x+y)
            in x * z;;

let f x y = let f' a = a ^ "\n"
            in f' (x^y)

let sumsqrs n =
    let rec aux m =
        if m>n then 0
        else m*m + aux (m+1)
    in aux 1;;
```

Scope

- Set of variables accessible at a given point. We look at Java first. Basic rule: *closest enclosing declaration*.

```
class A {
    int x=3;
    void foo (int x) {
        System.out.println(x);
        for(int i=0; i<5; i++) {
                System.out.println(i);
        }
        System.out.println(i);
    }
}
```

Scope in OCaml

- Basic rule is the same, e.g.

```
let x = 5;;
let f x = let x = 7
          in print_int x;;
```


Rules of scope in OCaml

- Top level:

  let x = … ;;

  let f a = … ;;

- e : let x = e_1 in e_2

Rules of scope in OCaml

- e : let f x = e_1 in e_2

- e : let rec f x = e_1 in e_2

Mutual Recursion

- Does this work?

```
let even n = if n=0 then true
             else odd(n-1);;
let odd n = if n=0 then false
            else even(n-1);;
```

Tuples in OCaml

- Consider structs in C, or classes with public fields and no methods (and just one constructor).
  - Java Example:

```
class Pr {  public int x;
            public string s;
            public Pr(int x, int s) {
                this.x = x;   this.s = s;
            }
         }
```

- Purpose: Put several values together into a single object that can be passed to, or returned from, methods.

Tuples

- In Java, clients of class Pr access elements using dot notation:

```
Pr p = new Pr(3,"tim");
… p.x … p.s …
```

- OCaml: Create pair with no class definition needed:

```
let p = (3,"tim")
…  fst p  … .  snd p
```

- Type of p is "int * string".
- Tuples in OCaml serve same purpose as structs in C, Java.

Tuples

- Can have as many values as you wish in a tuple:

  (3, "rick", 4.0) : int * string * float

  ("ted", "bill") : string * string

  (3, ('a', 4)) : int * (char * int)

However, functions fst and snd work *only on pairs*. To define functions on other tuples, you need…

Pattern matching

- Three ways to define the same function:
  - let sum p = (fst p) + (snd p)
  - let sum (a,b) = a+b
  - let sum p = let (a,b) = p in a+b
  - All define the same function of type int * int → int
- Examples:
- let fst_of_3 (x,y,z) = x;;
- let incr_fst_of_3 (x,y,z) = x+1;;

"Polymorphic" types

- let fst_of_3 (x,y,z) = x;;
- let incr_fst_of_3 (x,y,z) = x+1;;

Curried vs. Uncurried functions

- let f x y = … x … y … curried form
- let g (x,y) = … x … y … uncurried form

"match" expressions

- Another way to use pattern-matching to define functions:

```
let fst_of_3  x =
    match x with
       (a,b,c) -> a;;
```

- But match expressions allow *alternates*:

```
let rec fib n =
    match n with 0 -> 1
                 | 1 -> 1
                 | _ -> fib(n-2) + fib(n-1);;
```

Lists

- Linked-lists in Java:

```
class List {
  int head;
  List tail;
  static List cons (int x, List y) {
    List lst = new List();
    lst.head = x;
    lst.tail = y;
    return lst;
  }
}
```

```
List lst1 = List.cons(3, null);
lst1.head = 3;
List lst2 = List.cons(4, lst1);
List lst3 = List.cons(5, lst2);
```

Recursive functions in Java

```
List lst1 = List.cons(3, null);
lst1.head = 3;
List lst2 = List.cons(4, lst1);

int sum (List L) {
    if (L==null)
    then return 0
    else return L.head + sum(L.tail);
}
```

or

```
int sum (List L) {
    return L==null ? 0 : L.head+sum(L.tail);
}
```

Recursive functions in Java

Exercise: define Append(List x, List y)

Lists in OCaml

- Built-in data type
- Syntax:

  [] - empty list

  [a; b; … ; c] - list with elements a, b, …, c

  a :: x - list obtained by putting a on the front of list x ("consing")
- Examples:

```
let lst1 = [];;
let lst2 = [3];;
lst1 = lst2;;
let lst3 = 5::4::lst2;;
lst3 = [5;4;3];;
```

Pattern-matchings on lists

```
let f [a;b] = …
let g (x::xs) = …
let h (x::y::xs) = …
let f x = match x with [] ->  ….
                     |  y::ys  -> …
```

Examples:

Lists of lists

Lists can contain anything, even other lists.

But, lists must be homogeneous – if a list contains an int, then all its elements must be ints; if it contains an int list, then all its elements must be int lists.

Which of the following are legal?

[1; 2; 3]	[[1]; [2;3]]	[1; [2;3]]	1 :: [2;3]
1 :: [[2;3]]	[1] :: [2;3]	[1] @ [2;3]	[1;2] :: 3
[1;2] @ 3	[1;2] @ [3]	[1] :: [[2; [3]]]	

Tuples vs. lists

- Tuples are fixed-size, heterogenous collections
- Lists are extendable, homogeneous collections