

CS421 Lecture 2

- ▶ Reminder: Office Hours now posted on web page
- ▶ Midterm dates: *TBD*

- ▶ Today's class: More 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:** $\tau_1 \rightarrow \tau_2 \rightarrow \dots \rightarrow \tau_n \rightarrow \tau$
 - ▶ e.g. int \rightarrow int \rightarrow 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) ;;
```

```
# let x = 3 ;;
```

```
# let g y = y + 2 ;;
```



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;;
```



Further examples

let $z = \dots$

and $t = \dots$

in $\dots z \dots t \dots$

let $f \ x =$

let $f' \ y = \dots$

in let $f'' \ z = \dots$

in $\dots f' \dots f'' \dots$

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 = ... ;;

scope of a

scope of x is rest of session, except where x is hidden by a new definition of x

scope of f is rest of session, except where hidden

▶ e: let x = e₁ in e₂

scope of e (i.e. all names defined up to this point) scope of e + x

Rules of scope in OCaml

▶ $e : \text{let } f \times = e_1 \text{ in } e_2$

scope of e
+ x

scope of $e + f$

▶ $e : \text{let rec } f \times = e_1 \text{ in } e_2$

scope of e
+ $x + f$

scope of $e + f$

▶

To understand letrec, consider this definition:

let $f\ x = \dots f\ (x-1) \dots$ in \dots

It is legal if the entire let expression is in the scope of a definition of f (with the right type). In that case, the expression $f\ (x-1)$ refers to this prior definition - probably not what was intended.

Mutual Recursion

► Does this work?

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

Note that this definition exports definitions of both even and odd,

```
let rec even n =  
  let rec odd n =  
    if n=0 then ...  
  in if n=0 then ... ;;
```

while this one produces only a definition of even

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 do this:

```
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".
- ▶ Pairs 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`

`let b = (3, ('a', 4)) : int * (char * int)`
`fst b ⇒ 3` `snd b ⇒ ('a', 4)`
`snd (snd b) ⇒ 4`

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



Pattern matching

- ▶ Two ways to define the same function:
 - ▶ $\text{let sum } p = (\text{fst } p) + (\text{snd } p)$
 - ▶ $\text{let sum } (a,b) = a+b$
 - ▶ Both define the same function of type $\text{int} * \text{int} \rightarrow \text{int}$
- ▶ Examples:
 - ▶ $\text{let fst_of_3 } (x,y,z) = x;;$
 $\text{let } t = (1, "ab", 9);;$
 $\text{fst_of_3 } t;;$
 - ▶ $\text{let incr_fst_of_3 } (x,y,z) = x+1;;$

▶

“Polymorphic” types

▶ let fst_of_3 (x,y,z) = x;;

'a * 'b * 'c → 'a

▶ let incr_fst_of_3 (x,y,z) = x+1;;

int * 'a * 'b → int

▶

Curried vs. Uncurried functions

▶ let $f \times y = \dots \times \dots y \dots$

curried form

▶ let $g(x,y) = \dots \times \dots y \dots$

uncurried form

$f(1,2)$ X

$g\ 1\ 2$ X



“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)

```
List Append (List x, List y) {  
    if (x == null)  
        return y;  
    else return  
        List.Cons(x.head,  
                  Append(x.tail, 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] = ...

Built-in append: @

let g (x::xs) = ...

List module:

let h (x::y::xs) = ...

open List;;
(contains hd and tl)

let f x = match x with [] ->
| y::ys -> ...

Examples: let rec sum x =

match x with
[] -> 0

| y::ys -> y + sum ys ;;



Append

let rec append x y =

match x with [] → y

| z :: zs → z :: (append zs y);;

(Compare these OCaml functions to the Java functions.)

Tuples vs. lists

- ▶ Tuples are fixed-size, heterogenous collections

- ▶ Lists are extendable, homogeneous collections

