

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, *automatic deallocation*
 - ▶ “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 + y ;;  
g : int → int
```



Nested let definitions

let $f \times y = \text{let } z = \text{sqrt}(x+y)$
in $x * z$;

let var = expr
in expr

let $f \times y = \text{let } f' a = a ^ "\backslash n"$
in $f' (x^y)$

$x + (\text{let } y = z + 1$
in $y + w)$

let sumsqrs n =
let rec aux m =
if $m > n$ then 0
else $m * m + \text{aux } (m + 1)$
in aux 1;

let $z = \dots$
and $t = \dots$ in $z + t$



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); - - - this.x - - -  
        for(int i=0; i<5; i++) {  
            System.out.println(i);  
        }  
        System.out.println(i);  
    }  
}
```

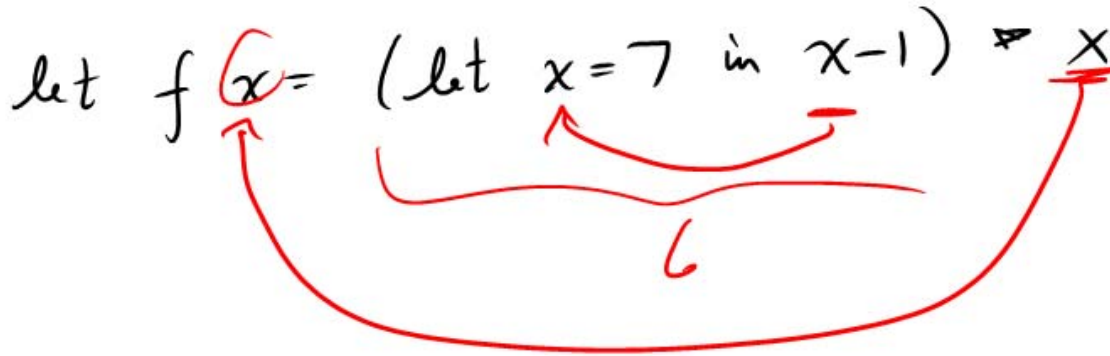
▶ }

Scope in OCaml

- ▶ Basic rule is the same, e.g.

let x = 5;;

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



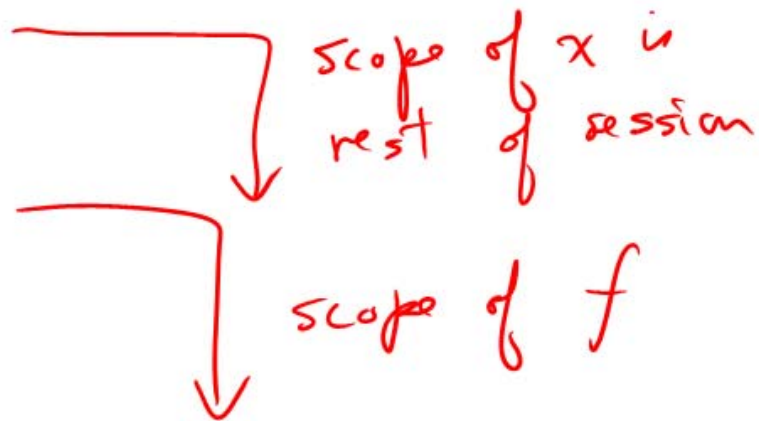
Rules of scope in OCaml

▶ Top level:

let $x = \dots$;;

let f a = ... ;;

scope of a



▶ e : let $x = e_1$ in e_2

scope of x



Rules of scope in OCaml

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

scope of x *scope of f*

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

scope of x
scope of f

▶

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);;
```

scope of even
and odd

```
let rec even n =  
  let rec odd n =  
    if --- even( )  
  in if n=0 --- odd( ) ---;
```

scope 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 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 -- fst 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:

let z =

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

~~fst z~~

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

let t =

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

~~int * char * int~~

fst (snd t) ✓

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



“Polymorphic” types

▶ `let fst_of_3 (x,y,z) = x;;`

Type: $\alpha * \beta * \gamma \rightarrow \alpha$
 $'a * 'b * 'c \rightarrow 'a$

▶ `let incr_fst_of_3 (x,y,z) = x+1;;`

Type: $\text{int} * 'a * 'b \rightarrow \text{int}$

▶

Curried vs. Uncurried functions

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

curried form

$int \rightarrow int \rightarrow int$

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

uncurried form

$int * int \rightarrow int$



“match” expressions

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

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

\equiv *let fst_of_3 (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;  
    List (int x, List y) {  
        head = x;  
        tail = y;  
    }  
    static List cons (int x, List y) {  
        return new List(x, y);  
    }  
}
```

```
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

```
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];;
```

```
let lst3 = lst2;;
```

```
let lst3 = 5::(4::lst2);;
```

```
lst3 = [5;4;3];;
```

~~(5::4)::lst2~~

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:

let rec append x y =
 match x with [] -> y
 | (a::as) -> a::(append as y)

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]]~~X~~ 1 :: [2;3]

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

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



Tuples vs. lists

- ▶ Tuples are fixed-size, heterogenous collections

- ▶ Lists are extendable, homogeneous collections



