CS 162 Programming Languages

# Lecture 4: OCaml Crash Course III

Yu Feng

Winter 2026

# Outline for today

- Last lecture on OCaml
- Date types
- Higher-order functions

# One-of types

We've defined a "one-of" type named attrib

Elements are one of:

- string
- `int`
- `int*int*int`
- float
- bool

```
type attrib =
  Name of string
| Age of int
| DOB of int*int*int
| Address of string
| Height of real
| Alive of bool
| Phone of int*int
| Email of string;
```

# Each-of types

We've defined a "Each-of" type (i.e., product type) named "DOB" attrib is the composition of three ints:

- `int*int*int`

```
type attrib =
  Name of string
| DOB of int*int*int
```

# List data type

```
type int_list =
  Nil
| Cons of int * int_list
```

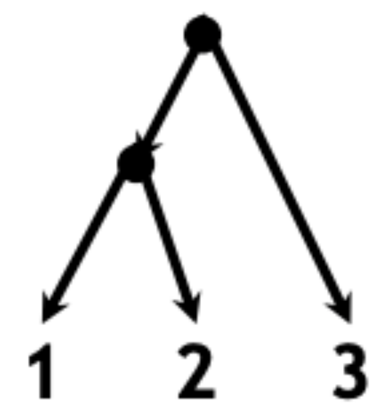
Lists are a derived type: built using elegant core!

1. Each-of
2. One-of
3. Recursive

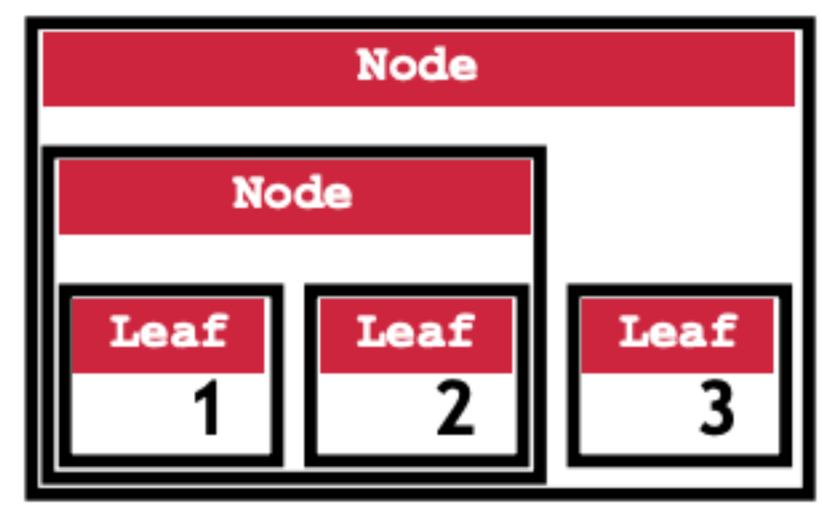
:: is just a syntactic sugar for "Cons"
[] is a syntactic sugar for "Nil"

# Representing Trees

```
type tree =
  Leaf of int
| Node of tree*tree
```




Node(Node(Leaf 1, Leaf 2), Leaf 3)



# sum_leaf: tree -> int

```
type tree =
  Leaf of int
| Node of tree*tree
```

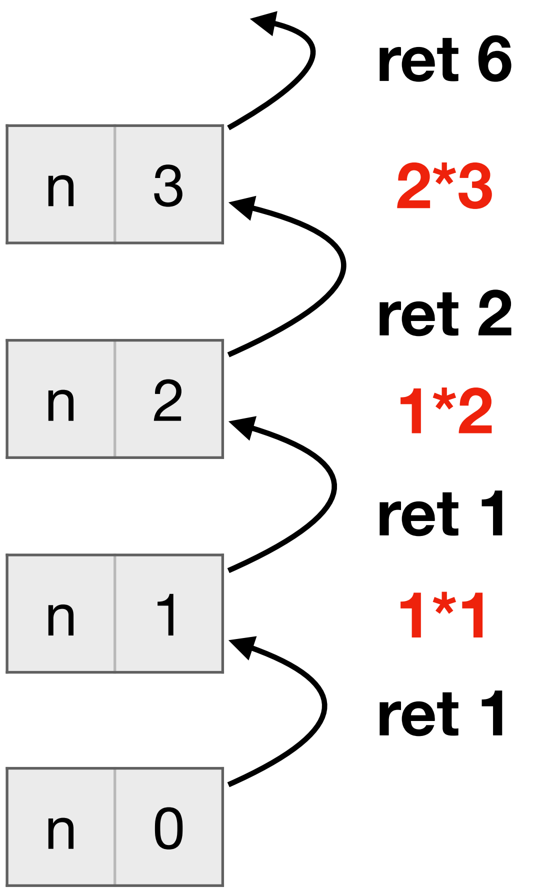
```
let rec sum_leaf t =
    match t with
    | Leaf n -> n
    | Node(t1,t2) -> (sum_leaf t1)
                    +(sum_leaf t2)
```

# Factorial: int –> int

```
let rec fact n =
    if n<=0
    then 1
    else n * fact (n-1);;

fact 3;;
```

**How does it execute?**



# Tail recursion

Tail recursion

- Recursion where all recursive calls are immediately followed by a return

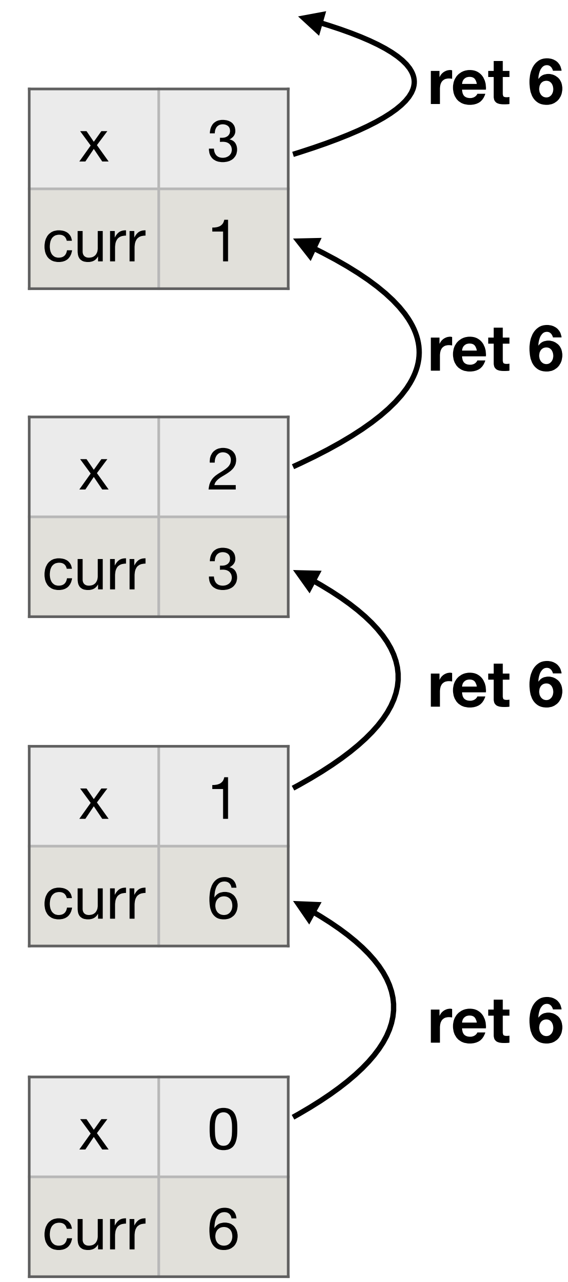
- In other words: not allowed to do anything between recursive call and return

# Tail recursive Factorial

```
let fact x =
  let rec helper x curr =
     if x <= 0
     then curr
     else helper (x - 1) (x * curr)
  in
     helper x 1;;

fact 3;;
```

**How does it execute?**



# Tail recursion

Tail recursion

- Recursion where all recursive calls are immediately followed by a return
- In other words: not allowed to do anything between recursive call and return

Why do we care about tail recursion?

- Tail recursion can be optimized into a simple loop

# Compiler optimization

```
let fact x =
  let rec helper x curr =
     if x <= 0
     then curr
     else helper (x - 1) (x * curr)
  in
     helper x 1;;
```

```
fact(x) {
  curr := 1;
  while (1) {
  if (x <= 0)
  then { return curr }
  else { x := x - 1;
         curr := (x * curr) }}
}
```

**Recursion**

**Loop**

# `max` function

```
let max x y = if x < y then y else x;;

(* return max element of list l *)
let list_max l =
   let rec l_max l =
      match l with
        [] -> 0
       | h::t -> max h (l_max t)
   in
      l_max l;;
```

# A better `max` function

```
let max x y = if x < y then y else x;;

(* return max element of list l *)
let list_max2 l =
   let rec helper cur l =
      match l with
        [] -> cur
       | h::t -> helper (max cur h) t
   in
      helper 0 l;;
```

**Tail recursion**

# concat function

```
(* concatenate all strings in a list *)
let concat l =
   let rec helper cur l =
      match l with
        [] -> cur
       | h::t -> helper (cur ^ h) t
   in
      helper "" l;;
```

# What is the pattern?

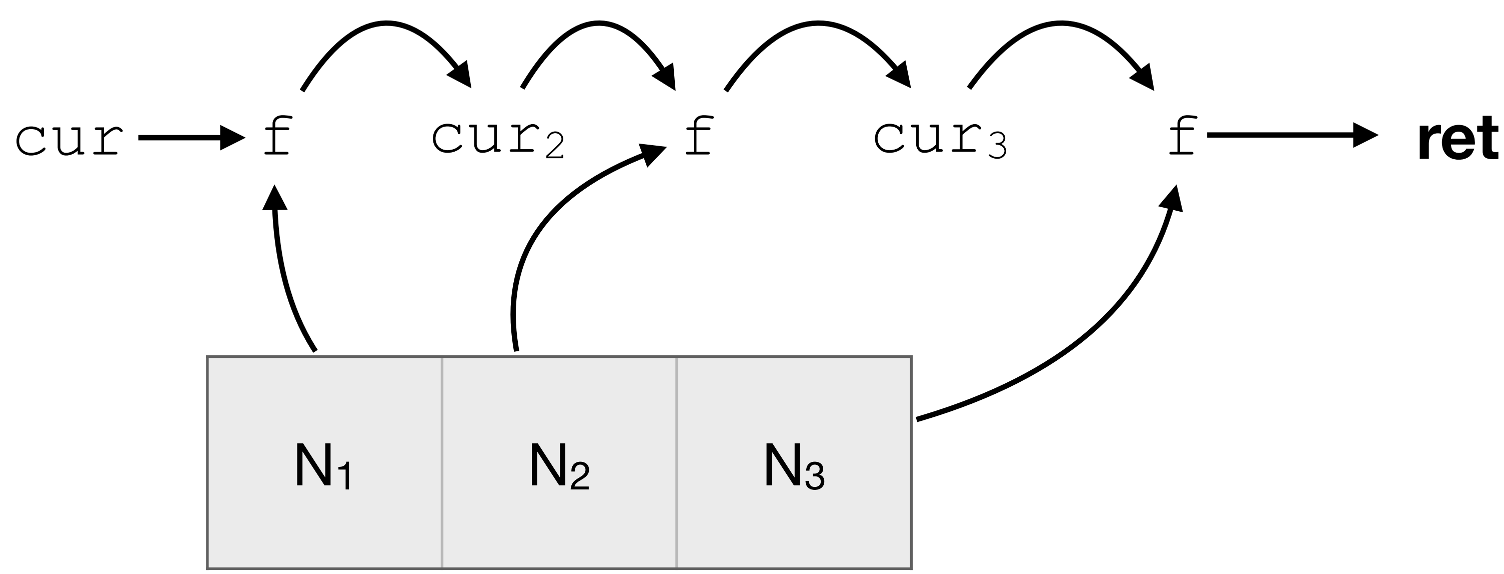
```
(* return max element of list l *)
let list_max2 l =
   let rec helper cur l =
      match l with
        [] -> cur
       | h::t -> helper (max cur h) t
   in
      helper 0 l;;
```

The two functions are sharing the same template!

```
(* concatenate all strings in a list *)
let concat l =
   let rec helper cur l =
      match l with
        [] -> cur
       | h::t -> helper (cur ^ h) t
   in
      helper "" l;;
```

# fold

```
(* fold, the coolest function! *)
let rec fold f cur l =
   match l with
      [] -> cur
      | h::t -> fold f (f cur h) t;;
```



# fold: examples

```
let list_max = fold max 0 l;;
```

```
let concat = fold (^) “" l;;
```

# map

```
# (* return the list containing f(e)
     for each element e of l *)
let rec map f l =
    match l with
    [] -> []
    | h::t -> (f h)::(map f t);;
```

```
let incr x = x+1;;

let map_incr = map incr;;

map_incr [1;2;3];;
```

# Composing functions

**$(f \circ g)(x) = f(g(x))$**

```
# (* return a function that given an argument x
applies f2 to x and then applies f1 to the result *)
let compose f1 f2 = fun x -> (f1 (f2 x));;

(* another way of writing it *)
let compose f1 f2 x = f1 (f2 x);;
```

# Higher-order functions

```
let map_incr_2 = compose map_incr map_incr;;
map_incr_2 [1;2;3];;

let map_incr_3 = compose map_incr map_incr_2;;
map_incr_3 [1;2;3];;

let map_incr_3_pos = compose pos_filer map_incr_3;;
```

**Instead of manipulating lists, we are manipulating the list manipulators!**

# Putting all together

Function ***choose*** that takes a list ***xs*** and a non-negative integer ***n***, and returns a list of all possible ways to choose n elements from xs.
For example, ***choose [1;2;3] 2*** should return ***[ [1;2]; [1;3]; [2;3] ]***

```
let rec choose (n: int) (xs: 1 ) : 2 =
  if n = 0 then 3
  else
    match xs with
    | [] -> 4
    | y::ys ->
      let r1 = choose 5 6 in
      let r2 = choose 7 8 in
      let r3 = map (fun (zs: 9 ) -> 10 11 12 ) r2 in
      r1 13 r3
```

**Candidate pool:**

1: `n` 2: `n-1` 3: `n-2`
4: `::` 5: `@` 6: `[]` 7: `[[]]`
8: `xs` 9: `y` 10: `ys` 11: `zs`
12: `[xs]` 13: `[y]` 14: `[ys]` 15: `[zs]`
16: `'a list` 17: `'a list list`

# Benefits of higher-order functions

Identify common computation patterns

- Iterate a function over a set, list, tree …
- Accumulate some value over a collection

Pull out (factor) “common” code:

- Computation Patterns
- Re-use in many different situations