

CS 162 Programming Languages

# Lecture 4: OCaml Crash Course III

Yu Feng  
Winter 2023

# Outline for today

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

# Recursive types

```
type nat = Zero | Succ of nat
```

What are values of nat ?  
One nat contains another!

**nat = recursive type**



# plus: nat\*nat -> nat

*Base pattern*  
*Inductive pattern*

```
type nat =  
| Zero  
| Succ of nat
```

*Base pattern*  
*Inductive pattern*

```
let rec plus n m =  
match m with  
| Zero -> n  
| Succ m' -> Succ (plus n m')
```

*Base expression*

*Inductive expression*

# List datatype

```
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”

# List function: length

```
let rec len l =  
  match l with  
    Base pattern | Nil -> 0 Base expression  
    Inductive pattern | Cons(h,t) -> 1 + (len t)  
                                Inductive expression
```

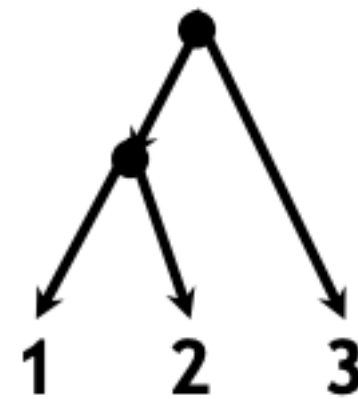
# List function: list\_max

```
let rec list_max l =  
  match l with  
    Base pattern | Nil -> 0 Base expression  
    Inductive pattern | Cons(h,t) -> max h (list_max t)  
                                     Inductive expression
```

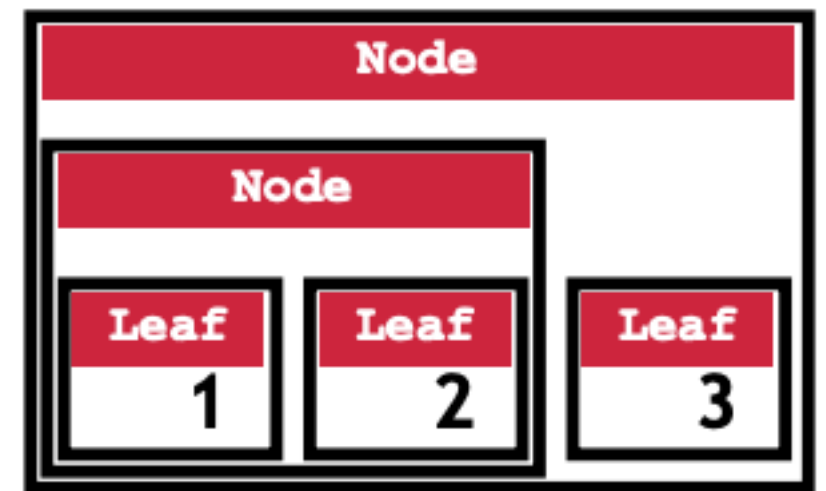
```
let max x y = if x > y then x else y;;
```

# Representing Trees

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



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





$\text{sum\_leaf: tree} \rightarrow \text{int}$

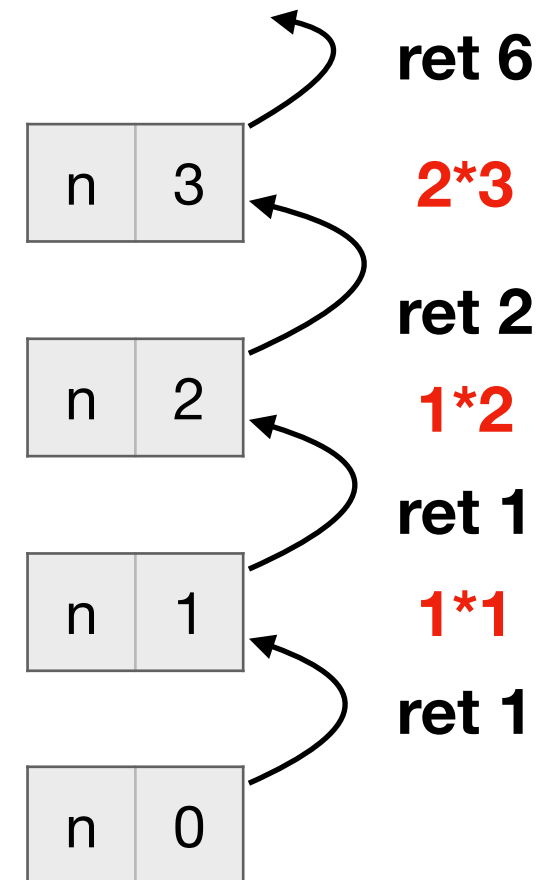
```
type tree =  
  Leaf of int  
| Node of tree*tree
```

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

# Factorial: $\text{int} \rightarrow \text{int}$

```
let rec fact n =  
  if n <= 0  
  then 1  
  else n * fact (n-1);;  
  
fact 3;;
```

**How does it execute?**



# Tail recursion

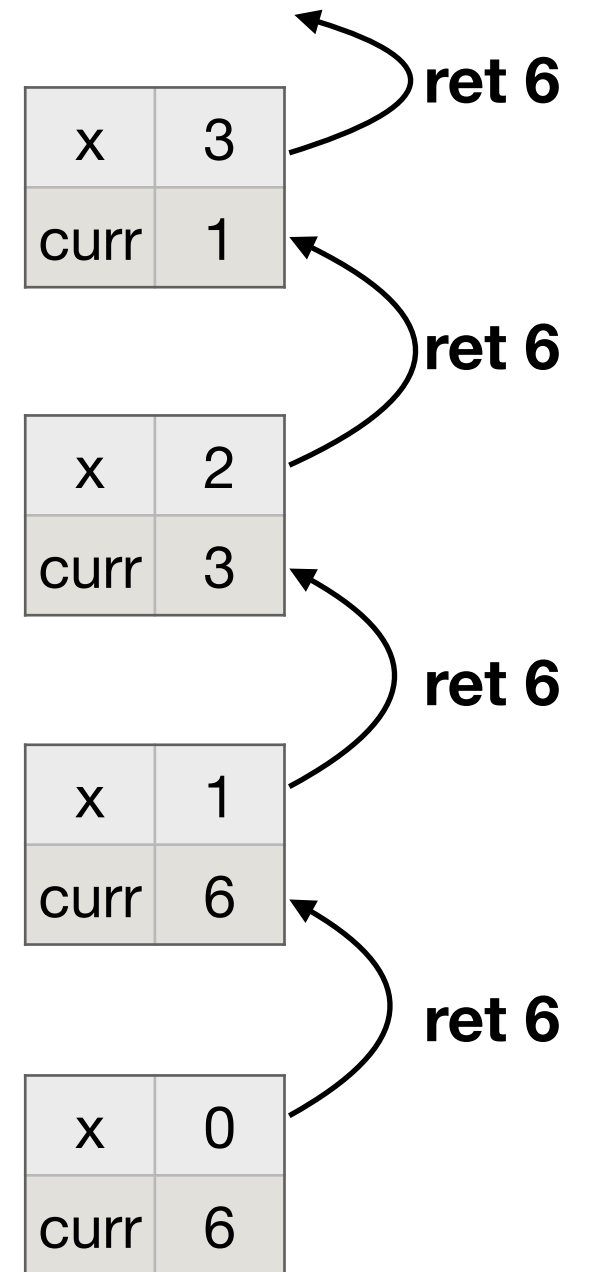
## Tail recursion

- 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;;
```

**Recursion**

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

**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?

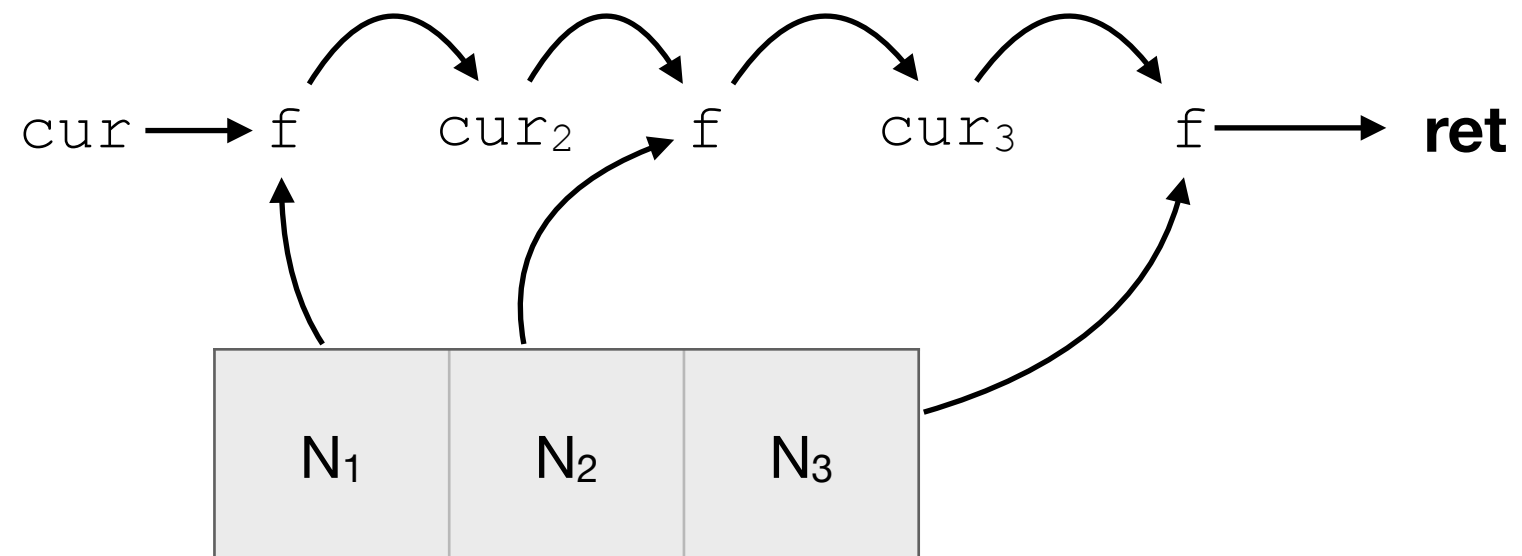
```
(* return max element of list l *)  
let list_max2 l =  
  let rec helper cur l =  
    match l with  
    [] -> cur  
    | h::t -> helper (max cur h) t  
  in  
    helper 0 l;;
```

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!**

# 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