Ocaml
Higher Order Functions
(fns which take fns as args)

Tail calls
- pop local frame before calling
- callee returns to fn's caller

```
let rec fac n = match n with
    | 0 -> 1
    | n when n > 0 -> n * fac (n-1)
    | _ -> raise (Invalid_arg "fac")
```

```
let fac n =
    let fac' n' m' =
        if n' = 0 then m'
                  else fac' (n'-1) (n' * m')
    in if n >= 0 then fac' n 1
                 else raise (Invalid_arg "fac"
```

- can use patter matching
  or if then else

Pattern Matching

Ocaml HOF (2)

```
#let hd l = match l with
          | [] -> raise (Invalid_arg "hd")
          | x :: _ -> x
  hd: 'a list -> 'a
#let tl l = match l with
          | [] -> raise (Invalid_arg "tl")
          | _ :: xs -> xs
  tl: 'a list -> 'a list
```

— 'a is parametric type
— _ is wildcard (ignored)

```
#let curry f (x,y) = f x y
('a * 'b) -> 'a -> 'b -> 'c
#let uncurry f x y = f (x, y)
'a -> 'b -> ('a * 'b) -> 'c
```

```
#let add = uncurry (+)
  add (3,4) ==> 7
```

~~('a -> 'a) -> int -> 'a~~
~~- fn computes n closure~~ $f^n x$
~~ex $f^3 x = f(f(f\ x))$~~
~~- arg & result both 'a~~

Apply fn n times
ex $f^3 x = f(f(f\,x))$

```
# let apply_n f n x =
    if n<0 then raise (Invalid_arg "apply_n")
    else if n = 0 then x
         else apply_n f (n-1) (f x)
('a -> 'a) -> int -> 'a -> 'a
# apply_n (fun x -> x+1) 4 ∅
  ⇒ 4
```

apply_n ((+)1) 4 ∅

```
# let foldl f u list = match list with
    | [] -> u
    | x::xs -> foldl f (f u x) xs
('a -> 'b -> 'a) -> 'a -> 'b list -> 'a

# let sum l = match l with
    | [] -> 0
    | x::xs -> x + sum xs
# let sum l =
    let sum' l t = match l with
        | [] -> t
        | x::xs -> sum' xs (t+x)
    in sum' l ∅
# let sum l = foldl (+) ∅ l
```

β reduction:

```
# let sum = foldl (+) ∅
```

- currying makes fns simpler

Ocaml HoF (4)

```
#let len l =
    let len' l m = match l with
        | [] -> m
        | _::xs -> len' xs (m+1)
    in len' l 0
```

```
#let len = fold l (fun a _ -> a+1) 0
```

use a lambda, ignore 2nd argument

NOTE: foldl is left associative with unit value at left of list
- is tail recursive

But what if no identity
use reduce instead.

```
#let reduce f l = match l with
    | [] -> raise (Invalid_arg "reduce")
    | x::xs -> foldl f x xs
```

don't want error? → return option

```
#let reduce f l = match l with
    | [] -> None
    | x::xs -> Some (foldl f x xs)
```

('a → 'b → 'a) → 'b list → 'a option.

Folding

Ocaml HOF 5

```
#let foldl f u l = match l with
        | [] -> u
        |x::xs -> foldl f (f u x) xs
#let foldr f l u = match l with
        | [] -> u
        |x::xs -> f x (foldr f xs u)
foldl : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a
foldr : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b
```

Use of foldr

```
#let map f l = match l with
        | [] -> []
        | x::xs -> f x ::map f xs
('a -> 'b) -> 'a list -> 'b list
#let map f l = foldr
                (fun h t -> f h :: t)
                l []
```

```
map (+1) [1;2;3]
= foldr @ [1;2;3] []
= (+)1 1 :: (foldr @ [2;3] [])
= (+)1 1 :: ((+)1) 2 :: (foldr @ [3] [])
= 2 :: 3 :: 4 :: []
= [2;3;4]
```

skipped steps.

Ocaml HOF 6

Reverse a list

- consider two stacks
- pop old push new until done

```
# let reverse l =
    let rev l m = match l with
      | [] -> m
      | x::xs -> rev xs (x::m)
    in rev l []
```

reverse [1;2;3] = rev [1;2;3] []
= rev [2;3] [1]
= rev [3] [2;1]
= rev [] [3;2;1]
= [3; 2; 1]

tail rec so use fold l

let reverse = foldl (fun t h -> h :: t) []

reverse [1;2;3] = foldl ⊙ [] [1;2;3]
= foldl ⊙ 1 [2;3]
⋮

Filter

$l \leftarrow [\forall x \in l \mid p\ x]$

Ocaml HOF (7)

```
# let filter p l = match l with
      | [] -> []
      | x::xs -> if p x then x::filter p xs
                 else filter p xs

# let filter p =
      foldr (fun h t -> if p h
                        then h::t
                        else t) []
```

```
# let rec copy l = match l with
      | [] -> []
      | x::xs -> x :: copy xs

# let copyl = foldr (fun t h -> h::t) l []
```

```
    copyl [1;2;3]
= foldr ◎ [1;2;3] []
=                  1 :: 2 :: (3::[])
```

```
#let rec unzip l = match l with
    | [] -> ([], [])
    | (a,b)::xs ->
        let (al,bl) = unzip xs
        in (a::al, b::bl)
```

// list of pairs → pair of lists
('a * 'b) list → 'a list * 'b list

unzip [1,2;3,4;5,6] → [1;3;5], [2;4;6]

---

zipwith
join pair of lists with operator

```
#let rec zipwith f l1 l2 =
    match l1, l2 with
    | [], [] -> []
    | [], _ -> raise (Invalid_arg "zipwith")
    | _, [] -> raise (Invalid_arg "zipwith")
    | h1::t1, h2::t2 ->
        f h1 h2 :: zipwith f t1 t2
```

('a → 'b → 'c) → 'a list → 'b list → 'c list

zipwith (+) [1;2;3] [4;5;6] ⟹ [5;7;9]
zipwith (fun a b → a,b) [1;2;3] [4;5;6]
⟹ [(1,4);(2,5);(3,6)]

```
# let rec apply'n f n x =
    match n with
    | n when n<0 -> invalid_arg "apply'n"
    | 0 -> x
    | n -> apply'n f (n-1) (f x)

('a -> 'a) -> int -> 'a -> 'a
# apply'n (fun x -> x+2) 5 0 ;; 10
                          f(f(f x))
```

How about trees?

```
# type 'a tree = Tree of 'a tree * 'a * 'a tree
               | Null
```

~~let postorder f tree = match tree with~~
~~| Leaf a -> f a~~
~~| Tree (l,a,r)~~

```
# let rec postorder f u tree = match tree with
    | Null -> u
    | Tree (l,a,r) -> f (postorder f u l) a (postorder f u r)

('a -> 'b -> 'a -> 'a) -> 'a -> 'b tree -> 'a
```