CS 162 Programming languages

Lecture 7: Higher-order Functions

Yu Feng

Winter 2020

`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?

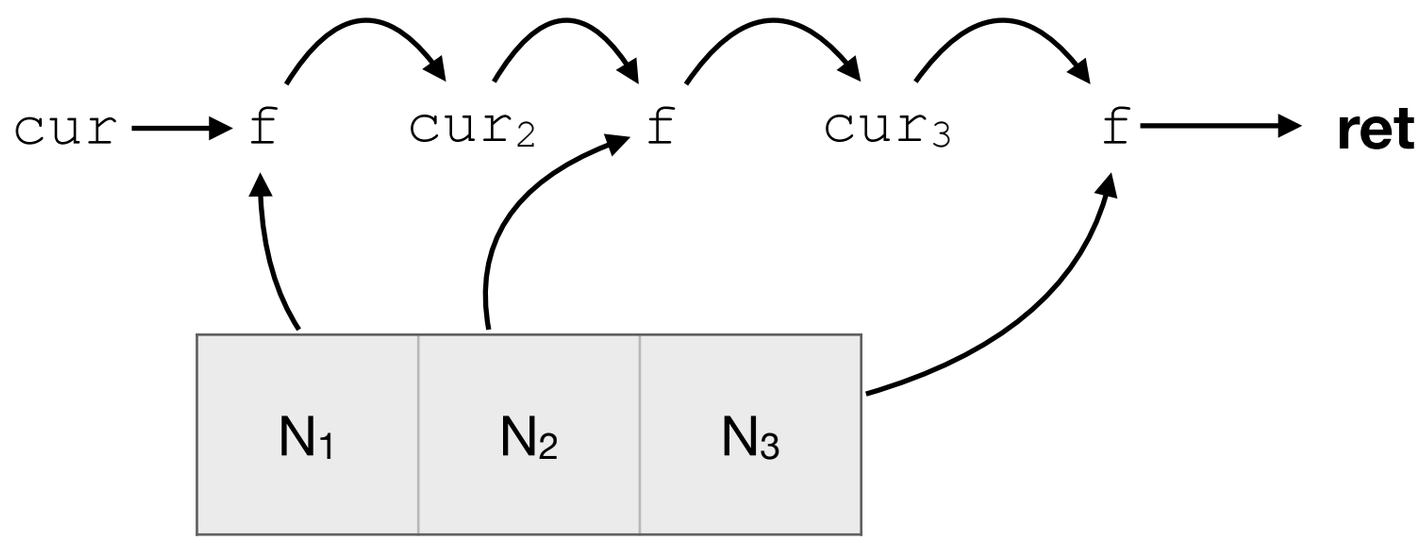
```
(* return max element of list l *)
let list_max2 l =
   let rec helper cur l =
      match l with
        [] -> cur
      | h::t -> helper (max cur h) t
   in
      helper 0 l;;
```

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;;
```

```
let concat = fold (^) "";;
```

```
let multiplier = fold (*) 1;;
```

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 ∘ 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

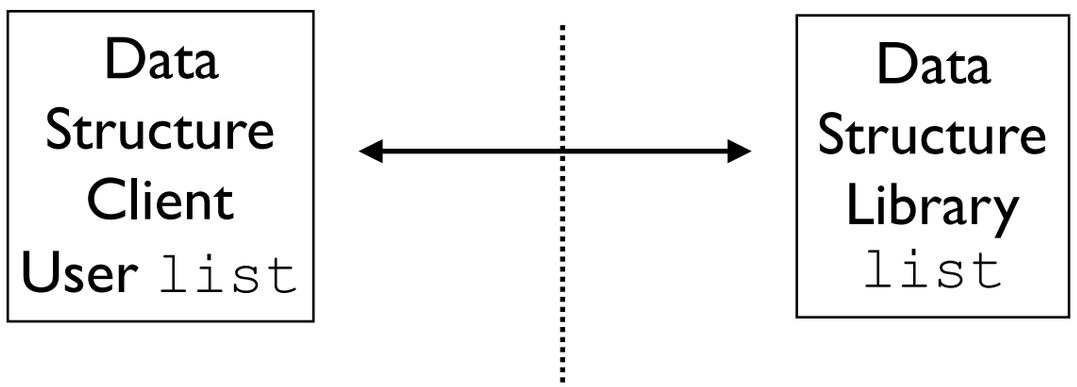
Pull out (factor) “common” code:

- Computation Patterns
- Re-use in many different situations

Functions as arguments/return

Higher-order functions enable modular code

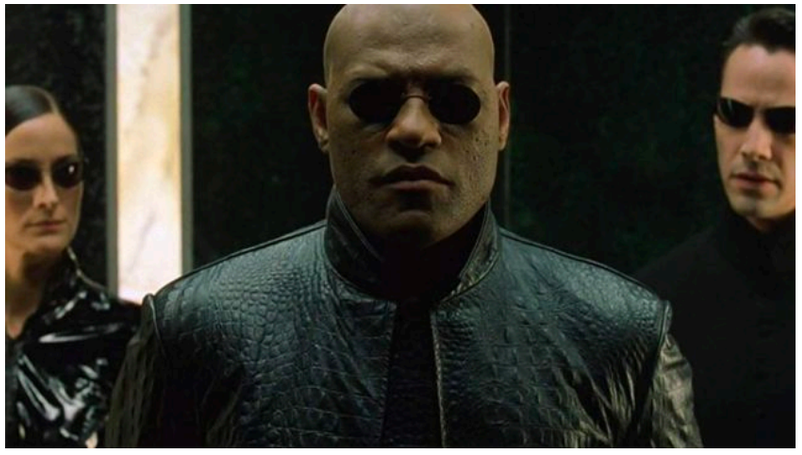
- Each part only needs local information



Uses meta-functions:
`map,fold,filter`
With locally-dependent funs (lt h), square etc.
Without requiring details of data structure

Provides meta-functions:
`map,fold,filter`
to traverse, accumulate over lists, trees etc.
Meta-functions don't need client info

Functions as arguments/return



"Free your mind"
- Morpheus

Different way of thinking about computation

- Manipulate the manipulators