News

- Programming assignment #3 is up.
- Due Monday Feb 2
- Midterm: Thursday Feb 12

Recap from last time

Three key ways to build complex types/values

1. “Each-of” types
   Value of T contains value of T1 and a value of T2

2. “One-of” types
   Value of T contains value of T1 or a value of T2

3. “Recursive”
   Value of T contains (sub)-value of same type T

Today

- More on recursion
- Higher-order functions
  - taking and returning functions
Factorial

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

Factorial

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

How does it execute?

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

How does it execute?

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

- Tail recursion:
  - recursion where all recursive calls are followed by a return
  - in other words: not allowed to do any between recursive call and return

- Why do we care about tail recursion?
  - it turns out that tail recursion can be optimized into a simple loop

Tail recursive factorial

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

How does it execute?

```ocaml
fact 3;;
```
How does it execute?

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;;

Compiler can optimize!

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

Tail recursion summary

• Tail recursive calls can be optimized as a jump

• Part of the language specification of some languages (ie: you can count on the compiler to optimize tail recursive calls)

More recursion: interval

(* return a list that contains the integers i through j inclusive *)

let rec interval i j =
(* return a list that contains the integers i through j inclusive *)
let rec interval i j =  
  if i > j  
  then []  
  else i::(interval (i+1) j);;

(* return a list that contains the elements \(f(i), f(i+1), \ldots f(j)\) *)
let rec interval_init i j f =  
  if i > j  
  then []  
  else (f i)::(interval_init (i+1) j f);;

(* our regular interval function in terms of the one with the init function *)
let rec interval i j =
**interval function again**

(* our regular interval function in terms of the one with the init function *)

let rec interval i j =
  interval_init i j (fun x -> x);;

**Interval function yet again!**

(* let's change the order of parameters... *)

let rec interval_init f i j =
  if  i > j then []
  then (f i)::(interval_init f (i+1) j);

(* now can use currying to get interval function! *)

let interval = interval_init (fun x -> x);;

---

**Function Currying**

In general, these two are equivalent:

let f = fun x1 -> ... -> fun xn -> e

let f x1 ... xn = e

Multiple argument functions by returning a function that takes the next argument
- Named after a person (Haskell Curry)

---

**Function Currying vs tuples**

Tuple version:

let f (x1,...,xn) = e  f (x1,...,xn)

call

Curried version:

let f x1 ... xn = e  f x1 ... xn
Function Currying vs tuples

Consider the following:

```ocaml
let lt x y = x < y;
```

Could have done:

```ocaml
let lt (x,y) = x<y;
```

• But then no “testers” possible

In general: Currying allows you to set just the first n params (where n smaller than the total number of params)

---

filter

(* return a list containing all elements of l for which f returns true *)
let rec filter f l =

match l with
    [] -> []
| h::t -> let t' = (filter f t) in
    if (f h)
    then h::t'
    else t';;

map

(* return the list containing f(e) for each element e of l *)
let rec map f l =
**map**

(* return the list containing f(e) for each element e of l *)

```ml
let rec map f l =
  match l with
  | [] -> []
  | h::t -> (f h)::(map f t);;
```

**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*)

```ml
let compose f1 f2 =
  fun x -> (f1 (f2 x));;
```

(* another way of writing it *)

```ml
let compose f1 f2 x = f1 (f2 x);;
```

**Higher-order functions!**

```ml
let map_incr_2 = compose map_incr map_incr;;
map_incr_2 (interval (-10) 10);;

let map_incr_3 = compose map_incr map_incr_2;;
map_incr_3 (interval (-10) 10);;

let map_incr_3_pos = compose pos_filter map_incr_3;;
map_incr_3_pos (interval (-10) 10);;
(compose map_incr_3_pos_filter) (interval (-10) 10);;
```
Higher-order functions!

```
let map_incr_2 = compose map_incr map_incr;;
map_incr_2 (interval (-10) 10);;
let map_incr_3 = compose map_incr map_incr_2;;
map_incr_3 (interval (-10) 10);;
let map_incr_3_pos = compose pos_filter map_incr_3;;
map_incr_3_pos (interval (-10) 10);;
(compose map_incr_3 pos_filter) (interval (-10) 10);;
```

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

```
let max x y = if x < y then y else x;;
(* return max element of list l *)
let list_max l =
```

max function

```
let max x y = if x < y then y else x;;
(* return max element of list l *)
let list_max l =
```

concat function

```
let concat l =
```

```
concat function

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

fold

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

What’s the pattern?

let list_max l =
  let rec helper curr l =
    match l with
    [] -> curr
    | h::t -> helper (max h curr) t
  in helper 0 l;;

let concat l =
  let rec helper curr l =
    match l with
    [] -> curr
    | h::t -> helper (curr ^ h) t
  in
  helper "" l;;

let list_max l =
  let rec helper curr l =
    match l with
    [] -> curr
    | h::t -> helper (max h curr) t
  in helper 0 l;;

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

let rec fold f curr l =
  match l with
  | [] -> curr
  | h::t -> fold f (f h curr) t;;

Examples of fold

let list_max = fold max 0;;

let concat = fold (^) "";;

let multiplier = fold (*) 1;;

Notice how all the recursion is buried inside two functions: interval and fold!
Examples of fold

```ocaml
let cons x y = x :: y;;
let f = fold cons [];;
(* same as: 
  let f l = fold cons [] l *)
```

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

Funcs taking/returning funcs

Higher-order funcs enable modular code
- Each part only needs local information

Data Structure

Client

Uses list

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

Data Structure

Library

list

Provides meta-functions: map, fold, filter
to traverse, accumulate over lists, trees etc.
Meta-functions don’t need client info
Different way of thinking

“Free your mind”
-Morpheus

• Different way of thinking about computation
• Manipulate the manipulators