Lecture 36:

- OCaml Wrap Up

Announcements:

- HW-8 out

OCaml User-Defined Types

Binary Trees

```
type 'a tree = Node of 'a * 'a tree * 'a tree
             | Empty
```

- Another recursive data structure!

Example trees:

```
let t1 = Node (2, Node (1, Empty, Empty), Node (3, Empty, Empty)) ;;
let t2 = Node ("b", Node ("a", Empty, Empty), Node ("c", Empty, Empty)) ;;
let t3 = Node ("d", t2, Node ("e", Empty, Empty)) ;;
```

Height function:

```
let rec height t =
  match t with
  | Empty -> 0
  | Node (_, l, r) -> 1 + max (height l) (height r)
```

OCaml User-Defined Types

```
let rec insert x t =
  match t with
  | Empty -> Node (x, Empty, Empty)
  | Node (y, l, r) when x < y -> Node (y, insert x l, r)
  | Node (y, l, r) -> Node (y, l, insert x r)


let rec find x t =
  match t with
  | Empty -> false
  | Node (y, _, _) when x == y -> true
  | Node (y, l, _) when x < y -> find x l
  | Node (_, _, r) -> find x r
```


OCaml User-Defined Types

```
let rec delete x t =
  match t with
  | Empty -> Empty
  | Node (y, l, r) when x < y -> Node (y, delete x l, r)
  | Node (y, l, r) when x > y -> Node (y, l, delete x r)
  | Node (y, l, r) -> (* x = y *)
    let rec delete_root t =
      match t with
      | Node (_, l, Empty) -> l
      | Node (_, Empty, r) -> r
      | Node (_, l, r) -> let s = inorder_succ r in Node (s, l, delete s r)
    and inorder_succ t =
      match t with
      | Node (x, Empty, _) -> x
      | Node (_, l, _) -> inorder_succ l
    in delete_root (Node (y, l, r))
```

Note that non-exhaustive patterns ...

- to fix, add `_ -> failwith "..."` to delete root and inorder succ

OCaml Lambda Functions

Can be useful for defining simple functions "on the fly"

```
# (fun x -> x * 2) 4 ;;
- : int = 8

# (fun x -> (fun y -> x + y)) 3 4 ;;
- : int = 7

# (fun x y -> x + y) 3 4 ;;
- : int = 7

# List.filter (fun x -> 5 > x) [1; 7; 3; 10] ;;
- : int list = [1; 3]

# List.filter ;;
= : ('a -> bool) -> 'a list -> 'a list = <fun>
```

- Note `filter` is a higher-order function
- Denoted by `()`'s in the type ... i.e., `('a -> bool)`

OCaml Higher-Order Functions

Defining Higher-Order functions

The "classic" `map` function ...

```
let rec map f xs =
  match xs with
  | [] -> []
  | x::t -> f x :: map f t

# map ((+) 1) [1; 2; 3] ;;
- : int list = [2; 3; 4]

# map (fun x - > 10 * x) [1; 2; 3] ;;
- : int list = [10; 20; 30]

# map ((>=) 0.5) [0.2; 0.6; 0.5; 0.3] ;;
- : bool list = [true; false; true; true]

# map ;;
- : ('a -> 'b) -> 'a list -> 'b list = <fun>
```

OCaml Higher-Order Functions

The "classic" `combine_with` function ... aka "zip with"

```
let rec combine_with f xs ys =
  match (xs, ys) with
    | ([], _) -> []
    | (_, []) -> []
    | (x::xs', y::ys') -> f x y :: combine_with f xs' ys'

# combine_with (+) [10; 20] [100; 200] ;;
- : int list = [110; 220]

# combine_with (<) ["ab"; "cd"] ["aa"; "ce"] ;;
- : bool list = [false; true]

# combine_with (fun x y -> if x < y then x else y) ["ab"; "cd"] ["aa"; "ce"] ;;
- : string list = ["aa"; "cd"]

# combine_with ;;
- : ('a -> 'b -> 'c) -> 'a list -> 'b list -> 'c list = <fun>
```


OCaml Higher-Order Functions

The "super powerful" `fold` function ... in this case, fold left

```
let rec foldr f a xs =
  match xs with
    | [] -> a
    | x::t -> foldl (f a x) t

# foldl (+) 0 [1; 2; 3] ;;
- : int = 6
# foldl (^) "" ["Hello!"; ", "; "World!"] ;;
- : string = "Hello, World!"
# foldl (fun x y -> x > y then x else y) 0 [1; 3; 2; 4; 6] ;;
- : int = 6

# let all p xs = foldl (&&) true (map p xs) ;;
val all : ('a -> bool) -> 'a list -> bool = <fun>

# all ((<) 0) [1; 4; 2; 3] ;;
- : bool = true
# all ((<) 0) [1; 3; -2] ;;
- : bool = false
```