Structural Recursion

- Functions on recursive datatypes (e.g., lists) tend to be recursive
- Recursion over recursive datatypes generally by structural recursion
  - Recursive calls made to components of structure of the same recursive type
  - Base cases of recursive types stop the recursion of the function

Forward Recursion

- In Structural Recursion, split input into components and (eventually) recurse on components
- Forward Recursion form of Structural Recursion
  - In forward recursion, first call the function recursively on all recursive components, and then build final result from partial results
  - Wait until whole structure has been traversed to start building answer

# let rec length list = match list with
| [] -> 0 (* Nil case *)
| x :: xs -> 1 + length xs;; (* Cons case *)
val length : 'a list -> int = <fun>

# length [5; 4; 3; 2];;
- : int = 4

- Nil case [] is base case
- Cons case recurses on component list xs

### Question

- How do you write length with forward recursion?

let rec length l =
Question

How do you write length with forward recursion?

```ml
let rec length l =
    match l with
    | [] -> 0
    | (a :: bs) -> 1 + length bs
```

Functions Over Lists

```ml
# let rec double_up list =
    match list with
    | [] -> []
    | (x :: xs) -> x :: x :: double_up xs;
val double_up : 'a list -> 'a list = <fun>

# let fib5_2 = double_up fib5;;
val fib5_2 : int list = [8; 8; 5; 5; 3; 3; 2; 2; 1; 1; 1; 1]
```

Functions Over Lists

```ml
# let silly = double_up ["hi"; "there"];;

# let rec poor_rev list =
    match list with
    | [] -> []
    | (x :: xs) -> poor_rev xs @ [x];;
val poor_rev : 'a list -> 'a list = <fun>

# poor_rev silly;;
- : string list = ["there"; "here"; "hi"; "hi"]
```

An Important Optimization

- When a function call is made, the return address needs to be saved to the stack so we know to where to return when the call is finished
- What if \( f \) calls \( g \) and \( g \) calls \( h \), but calling \( h \) is the last thing \( g \) does (a tail call)?
**An Important Optimization**

- When a function call is made, the return address needs to be saved to the stack so we know to where to return when the call is finished.
- What if \( f \) calls \( g \) and \( g \) calls \( h \), but calling \( h \) is the last thing \( g \) does (a tail call)?
- Then \( h \) can return directly to \( f \) instead of \( g \).

**Tail Recursion**

- A recursive program is tail recursive if all recursive calls are tail calls.
- Tail recursive programs may be optimized to be implemented as loops, thus removing the function call overhead for the recursive calls.
- Tail recursion generally requires extra “accumulator” arguments to pass partial results.
- May require an auxiliary function.

**Example of Tail Recursion**

```ocaml
# let rec prod l =  
  match l with []  
  | (x :: rem) -> x * prod rem;;  
val prod : int list -> int = <fun>
# let prod list =  
  let rec prod_aux l acc =  
    match l with [] -> acc  
    | (y :: rest) -> prod_aux rest (acc * y)  
    (* Uses associativity of multiplication *)  
    in prod_aux list 1;;  
val prod : int list -> int = <fun>
```

**Question**

- How do you write length with tail recursion?

```ocaml
let length l =  
  let rec length_aux list n =  
    in
```

**Question**

- How do you write length with tail recursion?

```ocaml
let length l =  
  let rec length_aux list n =  
    match list with [] ->  
    | (a :: bs) ->  
    in
```
Question

How do you write length with tail recursion?

```ml
let length l =
  let rec length_aux list n =
    match list with [] -> n
    | (a :: bs) ->
      in
```

Question

How do you write length with tail recursion?

```ml
let length l =
  let rec length_aux list n =
    match list with [] -> n
    | (a :: bs) -> length_aux bs (n + 1)
  in
```

Question

How do you write length with tail recursion?

```ml
let length l =
  let rec length_aux list n =
    match list with [] -> n
    | (a :: bs) -> length_aux bs (n + 1)
  in length_aux l 0
```

Mapping Recursion

One common form of structural recursion applies a function to each element in the structure

```ml
# let rec doubleList list = match list with
  [ ] -> [ ]
  | x::xs -> 2 * x :: doubleList xs;
val doubleList : int list -> int list = <fun>

# doubleList [2;3;4];;
- : int list = [4; 6; 8]
```
Mapping Functions Over Lists

```ocaml
# let rec map f list =  
    match list with  
    | [] -> []  
    | (h::t) -> (f h) :: (map f t);;  
val map : ('a -> 'b) -> 'a list -> 'b list = <fun>
```

```ocaml
# map plus_two fib5;;  
- : int list = [10; 7; 5; 4; 3; 3]
```

```ocaml
# map (fun x -> x - 1) fib6;;  
: int list = [12; 7; 4; 2; 1; 0; 0]
```

Your turn now

Write a function

```ocaml
make_app : (('a -> 'b) * 'a) list -> 'b list
```

that takes a list of function – input pairs and gives the result of applying each function to its argument. Use map, no explicit recursion.

```ocaml
let make_app l =
```

Folding Functions over Lists

How are the following functions similar?

```ocaml
# let rec sumlist list = match list with  
    [ ] -> 0  
    | x::xs -> x + sumlist xs;;
val sumlist : int list -> int = <fun>
```

```ocaml
# sumlist [2;3;4];;  
- : int = 9
```

```ocaml
# let rec prodlist list = match list with  
    [ ] -> 1  
    | x::xs -> x * prodlist xs;;
val prodlist : int list -> int = <fun>
```

```ocaml
# prodlist [2;3;4];;  
- : int = 24
```

Folding Recursion

```ocaml
# let rec multList list = match list with  
    [ ] -> 1  
    | x::xs -> x * multList xs;;
val multList : int list -> int = <fun>
```

```ocaml
# multList [2;4;6];;  
- : int = 48
```

How are the following functions similar?

```ocaml
# let rec sumlist list = match list with  
    [ ] -> 0  
    | x::xs -> x + sumlist xs;;
```

```ocaml
# sumlist [2;3;4];;  
- : int = 9
```

```ocaml
# let rec prodlist list = match list with  
    [ ] -> 1  
    | x::xs -> x * prodlist xs;;
```

```ocaml
# prodlist [2;3;4];;  
- : int = 24
```
Folding Functions over Lists

How are the following functions similar?

```ocaml
# let rec sumlist list = match list with
   | [ ] -> 0
   | x::xs -> x + sumlist xs;;

# sumlist [2;3;4];;
- : int = 9

# let rec prodlist list = match list with
   | [ ] -> 1
   | x::xs -> x * prodlist xs;;

# prodlist [2;3;4];;
- : int = 24
```

Folding Functions over Lists

How are the following functions similar?

```ocaml
# let rec sumlist list = match list with
   | [ ] -> 0
   | x::xs -> x + sumlist xs;;

# sumlist [2;3;4];;
- : int = 9

# let rec prodlist list = match list with
   | [ ] -> 1
   | x::xs -> x * prodlist xs;;

# prodlist [2;3;4];;
- : int = 24
```

Folding Functions over Lists

How are the following functions similar?

```ocaml
# let rec fold_right f list b = match list with
   | [] -> b
   | (x :: xs) -> f x (fold_right f xs b);;

# fold_right (fun s -> fun () -> print_string s) ["hi"; "there"]();
therehi- : unit = ()
```

Folding Functions over Lists

How are the following functions similar?

```ocaml
# let rec sumlist list = match list with
   | [ ] -> 0
   | x::xs -> x + sumlist xs;;

# sumlist [2;3;4];;
- : int = 9

# let rec prodlist list = match list with
   | [ ] -> 1
   | x::xs -> x * prodlist xs;;

# prodlist [2;3;4];;
- : int = 24
```

Recurring over lists

```ocaml
# let rec fold_right f list b = match list with
   | [] -> b
   | (x :: xs) -> f x (fold_right f xs b);;

# fold_right (fun val init -> val + init) [1; 2; 3]
   0;;
- : int = 6
```

Folding Recursion

- multList folds to the right
- Same as:

```ocaml
# let multList list = List.fold_right (fun x -> fun p -> x * p) list 1;
  val multList : int list -> int = <fun>

# multList [2;4;6];;
- : int = 48
```
Encoding Recursion with Fold

```ocaml
# let rec append list1 list2 = match list1 with
    | [] -> list2 | x::xs -> x :: append xs list2;;
val append : 'a list -> 'a list -> 'a list = <fun>

Base Case  Operation  Recursive call
```

```ocaml
# let append list1 list2 =
    fold_right (fun x y -> x :: y) list1 list2;;
val append : 'a list -> 'a list -> 'a list = <fun>
```

```ocaml
# append [1;2;3] [4;5;6];;
: int list = [1; 2; 3; 4; 5; 6]
```

Question

```
let rec length l =
    match l with
    | [] -> 0
    | (a :: bs) -> 1 + length bs

- How do you write length with fold_right, but no explicit recursion?
```

```
let length list =
    List.fold_right
        (fun x
            fun n
            -> n + 1)
    list 0
```

Can you write fold_right (or fold_left) with just map? How, or why not?

Question

```
let prod list =
    let rec prod_aux l acc =
        match l with
        | [] -> acc
        | (y :: rest) -> prod_aux rest (acc * y)
        in prod_aux list 1;;

# prod [4;5;6];; 
: int = 120
```

Encoding Tail Recursion with fold_left

```
# let prod list = let rec prod_aux l acc =
    match l with
    | [] -> acc
    | (y :: rest) -> prod_aux rest (acc * y)
    in prod_aux list 1;;
```

```
# let prod list =
    List.fold_left
        (fun acc y
            -> acc * y)
    1 list;;
```

```
# prod [4;5;6];; 
: int = 120
```

Iterating over lists

```
# let rec fold_left f a list =
    match list with
    | [] -> a
    | (x :: xs) -> fold_left f (f a x) xs;;
val fold_left : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a = <fun>

# fold_left
    (fun () -> print_string)
    ()
    [
        "hi"; "there"];

hither: unit = ()
```

9/14/2017
Question

let length l =
    let rec length_aux list n =
        match list with
        | [] -> n
        | (a :: bs) -> length_aux bs (n + 1)
    in
    length_aux l 0

How do you write length with fold_left, but no explicit recursion?

let length list =
    List.fold_left (
        fun n -> fun x -> n + 1
    ) 0 list

Folding

# let rec fold_left f a list = match list with
  | [] -> a
  | (x :: xs) -> fold_left f (f a x) xs;;
fold_left f a [x1; x2;...;xn] = f(…(f (f a x1) x2)…)xn

# let rec fold_right f list b = match list with
  | [] -> b
  | (x :: xs) -> f x (fold_right f xs b);;
fold_right f [x1; x2;...;xn] b = f x1(f x2(…(f xn b)))

Recall

# let rec poor_rev list = match list with
  | [] -> []
  | (x :: xs) -> poor_rev xs @ [x];;
val poor_rev : 'a list -> 'a list = <fun>

What is its running time?

Quadratic Time

- Each step of the recursion takes time proportional to input
- Each step of the recursion makes only one recursive call.
- List example:

# let rec poor_rev list = match list with
  | [] -> []
  | (x :: xs) -> poor_rev xs @ [x];;
val poor_rev : 'a list -> 'a list = <fun>

Comparison

- poor_rev [1,2,3] =
- (poor_rev [2,3]) @ [1] =
- ((poor_rev [3]) @ [2]) @ [1] =
- (((poor_rev [ ] ) @ [3]) @ [2]) @ [1] =
- ([ ] @ [3]) @ [2] @ [1] =
- (3 :: ([ ] @ [2])) @ [1] =
- [3,2] @ [1] =
- 3 :: (2 :: [1]) =
- 3 :: (2 :: ([ ] @ [1])) = [3, 2, 1]
Tail Recursion - Example

```ocaml
# let rec rev_aux list revlist = 
  match list with
  [ ] -> revlist 
  | x :: xs -> rev_aux xs (x::revlist);
val rev_aux : 'a list -> 'a list -> 'a list = <fun>

# let rev list = rev_aux list [ ];;
val rev : 'a list -> 'a list = <fun>
```

What is its running time?

Comparison

- rev [1,2,3] =
- rev_aux [1,2,3] [] =
- rev_aux [2,3] [1] =
- rev_aux [3] [2,1] =
- rev_aux [ ] [3,2,1] = [3,2,1]

Folding - Tail Recursion

```ocaml
# let rec rev_aux list revlist = 
  match list with
  [ ] -> revlist 
  | x :: xs -> rev_aux xs (x::revlist);
# let rev list = rev_aux list [ ];;
```

Folding

- Can replace recursion by fold_right in any
  forward primitive recursive definition
  - Primitive recursive means it only recurses on immediate subcomponents of recursive data structure
- Can replace recursion by fold_left in any
  tail primitive recursive definition

Example of Tail Recursion

```ocaml
# let rec app fl x = 
  match fl with [ ] -> x
  | (f :: rem_fs) -> f (app rem_fs x);
val app : ('a -> 'a) list -> 'a -> 'a = <fun>

# let app fs x =
  let rec app_aux fl acc = 
    match fl with [ ] -> acc
    | (f :: rem_fs) -> app_aux rem_fs
      (fun z -> acc (f z))
  in app_aux fs (fun y -> y) x;;
val app : ('a -> 'a) list -> 'a -> 'a = <fun>
```

Continuation Passing Style

- A programming technique for all forms of “non-local” control flow:
  - non-local jumps
  - exceptions
  - general conversion of non-tail calls to tail calls
- Essentially it’s a higher-order function version of GOTO
Continuations

- Idea: Use functions to represent the control flow of a program
- Method: Each procedure takes a function as an argument to which to pass its result; outer procedure “returns” no result
- Function receiving the result called a continuation
- Continuation acts as “accumulator” for work still to be done

Continuation Passing Style

- Writing procedures so that they take a continuation to which to give (pass) the result, and return no result, is called continuation passing style (CPS)

Example

- Simple reporting continuation:
  ```
  # let report x = (print_int x; print_newline( ));
  val report : int -> unit = <fun>
  ```

- Simple function using a continuation:
  ```
  # let plusk a b k = k (a + b)
  val plusk : int -> int -> (int -> 'a) -> 'a = <fun>
  # plusk 20 22 report;
  42
  ```

Example of Tail Recursion & CSP

- A compilation technique to implement non-local control flow, especially useful in interpreters.
- A formalization of non-local control flow in denotational semantics.
- Possible intermediate state in compiling functional code.
Terms

- A function is in **Direct Style** when it returns its result back to the caller.
- A **Tail Call** occurs when a function returns the result of another function call without any more computations (e.g. tail recursion)
- A function is in **Continuation Passing Style** when it passes its result to another function.
  - Instead of returning the result to the caller, we pass it forward to another function.

Continuation Passing Style

- A compilation technique to implement non-local control flow, especially useful in interpreters.
- A formalization of non-local control flow in denotational semantics
- Possible intermediate state in compiling functional code

Example

- Simple reporting continuation:
  
  ```
  # let report x = (print_int x; print_newline( ));
  val report : int -> unit = <fun>
  ```

- Simple function using a continuation:
  
  ```
  # let plusk a b k = k (a + b)
  val plusk : int -> int -> (int -> 'a) -> 'a = <fun>
  ```

  ```
  # plusk 20 22 report;;
  42
  ```

- :: unit = ()

Simple Functions Taking Continuations

- Given a primitive operation, can convert it to pass its result forward to a continuation

  Examples:

  ```
  # let subk x y k = k(x + y);;
  val subk : int -> int -> (int -> 'a) -> 'a = <fun>
  ```

  ```
  # let eqk x y k = k(x = y);;
  val eqk : 'a -> 'a -> (bool -> 'b) -> 'b = <fun>
  ```

  ```
  # let timesk x y k = k(x * y);;
  val timesk : int -> int -> (int -> 'a) -> 'a = <fun>
  ```

Nesting Continuations

  ```
  # let add_three x y z = x + y + z;;
  val add_three : int -> int -> int -> int = <fun>
  ```

  ```
  # let add_three x y z = let p = x + y in p + z;;
  val add_three : int -> int -> int -> int = <fun>
  ```

  ```
  # let add_three_k x y z k =
  addk x y [fun p -> addk p z k];;
  val add_three_k : int -> int -> int -> (int -> 'a -> 'a) = <fun>
  ```