Functions

```
# let plus_two n = n + 2;;
val plus_two : int -> int = <fun>
# plus_two 17;;
- : int = 19
# let plus_two = fun n -> n + 2;;
val plus_two : int -> int = <fun>
# plus_two 14;;
- : int = 16
```

First definition syntactic sugar for second

Closure for \( \text{plus}_x \)

- When \( \text{plus}_x \) was defined, had environment:
  \[ \rho_{\text{plus}_x} = \{ \ldots, x \to 12, \ldots \} \]
- Recall: \( \text{let } \text{plus}_x \ y = y + x \)
  is really \( \text{let } \text{plus}_x = \text{fun } y \to y + x \)
- Closure for \( \text{fun } y \to y + x \):
  \( <y \to y + x, \rho_{\text{plus}_x}> \)
- Environment just after \( \text{plus}_x \) defined:
  \( \{ \text{plus}_x \to <y \to y + x, \rho_{\text{plus}_x}> \} + \rho_{\text{plus}_x} \)

Save the Environment!

- A closure is a pair of an environment and an association of a sequence of variables (the input variables) with an expression (the function body), written:
  \[ <(v_1, \ldots, v_n) \to \text{exp, } \rho> \]
- Where \( \rho \) is the environment in effect when the function is defined (for a simple function)
### Functions with more than one argument

Let's define a function that adds three numbers:

```ocaml
let add_three x y z = x + y + z;;
```

We can use it like this:

```ocaml
let t = add_three 6 3 2;;
```

### Partial application of functions

Let's define a function that adds two numbers:

```ocaml
let add_triple (u,v,w) = u + v + w;;
```

We can use it like this:

```ocaml
let add_triple (6,3,2);;
```

### Curried vs Uncurried

Recall:

```ocaml
val add_three : int -> int -> int -> int = <fun>
```

This is known as a **curried** function. In contrast,

```ocaml
val add_triple : int * int * int -> int = <fun>
```

This is known as an **uncurried** function.

### Functions as arguments

Let's define a function `thrice` that applies a given function three times:

```ocaml
let thrice f x = f (f (f x));;
```

We can use it like this:

```ocaml
let g = thrice plus_two;;
```

```ocaml
thrice (fun s -> "Hi! " ^ s) "Good-bye!";;
```
Higher Order Functions

- A function is **higher-order** if it takes a function as an argument or returns one as a result.
- Example:

  ```
  # let compose f g = fun x -> f (g x);
  val compose : ('a -> 'b) -> ('c -> 'a) -> 'c -> 'b = <fun>
  ``

  The type (`'a -> 'b`) and (`'c -> 'a`) are higher order types because they take functions as arguments.

  ```
  # let thrice f x = f (f (f x));
  val thrice : ('a -> 'a) -> 'a -> 'a = <fun>
  ``

  How do you write thrice with compose?

  ```
  # let thrice f = compose f (compose f f);
  val thrice : ('a -> 'a) -> 'a -> 'a = <fun>
  ``

  Partial Application

  ```
  # (+);
  - : int -> int -> int = <fun>
  # (+) 2 3;
  - : int = 5
  # let plus_two = (+) 2;;
  val plus_two : int -> int = <fun>
  # plus_two 7;;
  - : int = 9
  ```

  Partial Application also called sectioning

  Partial Application and “Unknown Types”

  ```
  # let f1 = compose plus_two;;
  val f1 : ('_a -> int) -> '_a -> int = <fun>
  ```

  Compare to lambda lifted version:

  ```
  # let f2 = fun g -> compose plus_two g;;
  val f2 : ('a -> int) -> 'a -> int = <fun>
  ```

  What is the difference?
Partial Application and “Unknown Types”

- A can be repeatedly instantiated

```ml
# f2 plus_two;;
- : int -> int = <fun>
# f2 List.length;;
- : '_a list -> int = <fun>
```

Lambda Lifting

- You must remember the rules for evaluation when you use partial application

```ml
# let add_two = (+) (print_string "test\n"; 2);;
test val add_two : int -> int = <fun>
# let add2 = (* lambda lifted *)
  fun x -> (+) (print_string "test\n"; 2) x;;
val add2 : int -> int = <fun>
```

Evaluating declarations

- Evaluation uses an environment \( \rho \)
- To evaluate a (simple) declaration \( \text{let } x = e \)
  - Evaluate expression \( e \) in \( \rho \) to value \( v \)
  - Update \( \rho \) with \( x \rightarrow v \)
  - Update: \( \rho_1 + \rho_2 \) has all the bindings in \( \rho_1 \) and all those in \( \rho_2 \) that are not rebound in \( \rho_1 \)
  \[
  \{x \rightarrow 2, y \rightarrow 3, a \rightarrow "hi"\} + \{y \rightarrow 100, b \rightarrow 6\} = \{x \rightarrow 2, y \rightarrow 3, a \rightarrow "hi", b \rightarrow 6\}
  \]

Evaluating expressions

- Evaluation uses an environment \( \rho \)
- A constant evaluates to itself
- To evaluate an variable, look it up in \( \rho (v) \)
- To evaluate uses of +, -, etc, eval args, then do operation
- Function expression evaluates to its closure
- To evaluate a local dec: \( \text{let } x = e_1 \text{ in } e_2 \)
  - Eval \( e_1 \) to \( v \), eval \( e_2 \) using \( \{x \rightarrow v\} + \rho \)

Evaluation of if-then-else

- Assume current environment \( \rho \)
- Evaluate \( \text{if } e_1 \text{ then } e_2 \text{ else } e_3 \) by
  - First evaluate \( e_1 \) to boolean \( v_1 \)
  - If \( v_1 \) is \( \text{true} \), evaluate \( e_2 \) to \( v_2 \); \( v_2 \) value of whole expression
    - Do not evaluate \( e_3 \)
  - If \( v_1 \) is \( \text{false} \), evaluate \( e_3 \) to \( v_3 \); \( v_3 \) value of whole expression
    - Do not evaluate \( e_2 \)
Eval of App $e_1 e_2$ with Closures in Ocaml

1. In environment $\rho_0$, evaluate right term $e_2$ to values $(v_1, ..., v_n)$
2. In environment $\rho_1$, evaluate left term $e_1$ to closure, $c = \langle (x_1, ..., x_n) \rightarrow b, \rho_1 \rangle$
3. Match $(x_1, ..., x_n)$ variables in (first) argument with value $(v_1, ..., v_n)$
4. Update environment $\rho_1$ to $\rho' = \{ x_1 \leftarrow v_1, ..., x_n \leftarrow v_n \} + \rho_1$
5. Evaluate body $b$ in environment $\rho'$

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

Structural Recursion: List Example

```ocaml
# 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
```

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

Forward Recursion: Examples

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

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

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
```

Your turn now

Try Problem 2 on ML2

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 [ ] -> 1 
  | (x :: rem) -> x * prod rem;;
val prod : int list -> int = <fun>
```

```ocaml
# 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 = 
    match list with [ ] -> n 
    | (a :: bs) -> n 
    in
```

Question

- How do you write length with tail recursion?

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

How do you write length with tail recursion?

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

Your turn now

Try Problem 4 on MP2

Mapping Recursion

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

```ocaml
# 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>
# map plus_two fib5;;
- : int list = [10; 7; 5; 4; 3; 3]
# map (fun x -> x - 1) fib6;;
: int list = [12; 7; 4; 2; 1; 0; 0]
```

Mapping Recursion

- Can use the higher-order recursive map function instead of direct recursion

```ocaml
# let doubleList list =    
  List.map (fun x -> 2 * x) list;;
val doubleList : int list -> int list = <fun>
# doubleList [2;3;4];;
- : int list = [4; 6; 8]
```

Folding Functions over Lists

- Another common form “folds” an operation over the elements of the structure

```ocaml
# let rec multList list =    
  match list with    
  | [] -> 1    
  | x::xs -> x * multList xs;;
val multList : int list -> int = <fun>
# multList [2;4;6];;
- : int = 48
```

Folding Recursion

- 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>
# sumList [2;3;4];;
- : int = 9
# let rec prodList list =    
  match list with    
  | [] -> 1    
  | x::xs -> x * prodList xs;;
val prodList : int list -> int = <fun>
# prodList [2;3;4];;
- : int = 24
```

Iterating over lists

```ocaml
# let rec fold_right f list b =    
  match list with    
  | [] -> b    
  | x::xs -> f x (fold_right f xs b);;
val fold_right : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b = <fun>
# fold_right    
  (fun s -> fun () -> print_string s)    
  (["hi"; "there"]    
  ();;
therehi- : unit = ()
```
Encoding Recursion with Fold

\[
\text{let rec append list1 list2 = match list1 with} \\
\text{[ ] -> list2 | x::xs -> x :: append xs list2}; \\
\text{val append : 'a list -> 'a list -> 'a list = <fun>} \\
\]  

Base Case | Operation | Recursive Call
--- | --- | ---

\[
\text{let append list1 list2 =} \\
\text{fold_right (fun x y -> x :: y) list1 list2}; \\
\text{val append : 'a list -> 'a list -> 'a list = <fun>} \\
\]  

\[
\text{append [1;2;3] [4;5;6];} \\
\text{- : int list = [1; 2; 3; 4; 5; 6]} \\
\]

Question

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

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

\[
\text{let length list =} \\
\text{List.fold_right (fun x -> fun n -> n + 1) list 0} \\
\]

Map from Fold

\[
\text{let map f list =} \\
\text{fold_right (fun x y -> f x :: y) list} \\
\]  

\[
\text{map ((+)1) [1;2;3];} \\
\text{- : int list = [2; 3; 4]} \\
\]

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

Iterating over lists

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

\[
\text{fold_left} \\
\text{(fun () -> print_string) ()} \\
\text{["hi"; "there"];} \\
\text{hithere- : unit = ()} \\
\]

Encoding Tail Recursion with fold_left

\[
\text{let rec prod list = let rec prod_aux l acc =} \\
\text{match l with [ ] -> acc} \\
\text{| (y :: rest) -> prod_aux rest (acc * y) in prod_aux list 1}; \\
\text{val prod : int list -> int = <fun>} \\
\]

\[
\text{prod [4;5;6];} \\
\text{- : int =120} \\
\]
Question

```ocaml
class 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?

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

Recall

```ocaml
# 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 f a [x_1; x_2;...;x_n] = f(...(f (f a x_1) x_2)...x_n)

# let rec fold_right f list b = match list
  with [] -> b |
    (x :: xs) -> f x (fold_right f xs b);;
val fold_right : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b = <fun>
fold_right f [x_1; x_2;...;x_n] b = f x_1(f x_2(...(f x_n b)...))
```

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:

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

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 list = rev_aux list [ ];;
val list : 'a list = <fun>
```

What is its running time?
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])) = [3, 2, 1]

Folding - Tail Recursion

- # let rev list =
  - fold_left
  - (fun i -> fun x -> x :: l) //comb op
    - [] //accumulator cell
    - 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

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
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

- 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 of Tail Recursion & CSP

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

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.

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 (eg 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.

Example

- Simple reporting continuation:
  ```ocaml
  # let report x = (print_int x; print_newline( ));;  
val report : int -> unit = <fun>
  # plusk 20 22 report;;  
- : unit = ()
  ```

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

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

Examples:

```ocaml
# 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

```ocaml
# 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>
```