Your turn now

Try steps 1 - 3 from WA0-practice

Functions

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

First definition syntactic sugar for second

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

Note: in fun v -> exp(v), scope of variable is only the body exp(v)

Values fixed at declaration time

```
# let x = 12;;
val x : int = 12
# let plus_x y = y + x;;
val plus_x : int -> int = <fun>
# plus_x 3;;
```

What is the result?
Values fixed at declaration time

```ocaml
# let x = 7;; (* New declaration, not an update *)
val x : int = 7

# plus_x 3;;
```

What is the result this time?

---

Question

- Observation: Functions are first-class values in this language
- Question: What value does the environment record for a function variable?
- Answer: a closure

---

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:
  \[ f \rightarrow < (v_1, \ldots, v_n) \rightarrow \text{exp}, \rho_f > \]
- Where \( \rho_f \) is the environment in effect when \( f \) is defined (if \( f \) is a simple function)

---

Closure for plus_x

- When plus_x was defined, had environment:
  \[ \rho_{\text{plus}_x} = \{ ..., x \rightarrow 12, ... \} \]
- Recall: let plus_x y = y + x
  is really let plus_x = fun y -> y + x
- Closure for fun y -> y + x:
  \[ < y \rightarrow y + x, \rho_{\text{plus}_x} > \]
- Environment just after plus_x defined:
  \[ \{ \text{plus}_x \rightarrow < y \rightarrow y + x, \rho_{\text{plus}_x} > \} + \rho_{\text{plus}_x} \]
Now it’s your turn

You should be able to do remainder of WA0-practice

Recursion and Induction

```
# let rec nthsq n = match n with
| 0 -> 0
| n -> (2 * n - 1) + nthsq (n - 1)
```

- Base case is the last case; it stops the computation
- Recursive call must be to arguments that are somehow smaller - must progress to base case
- **if** or **match** must contain base case
- Failure of these may cause failure of termination

Lists

- First example of a recursive datatype (aka algebraic datatype)

- Unlike tuples, lists are homogeneous in type (all elements same type)
Lists

- List can take one of two forms:
  - Empty list, written \([\ ]\)
  - Non-empty list, written \(x :: xs\)
    - \(x\) is head element, \(xs\) is tail list, :: called "cons"
  - Syntactic sugar: \([x]\) == \(x :: [\ ]\)
  - \([x_1; x_2; \ldots; x_n]\) == \(x_1 :: x_2 :: \ldots :: x_n :: [\ ]\)

# Lists

```
# let fib5 = [8;5;3;2;1];;;
val fib5 : int list = [8; 5; 3; 2; 1]
# let fib6 = 13 :: fib5;;
val fib6 : int list = [13; 8; 5; 3; 2; 1]
# (8::5::3::2::1::[ ]) = fib5;;
- : bool = true
# fib5 @ fib6;;
- : int list = [8; 5; 3; 2; 1; 13; 8; 5; 3; 2; 1; 1]
```

# Lists are Homogeneous

```
# let bad_list = [1; 3.2; 7];;
Characters 19-22:
  let bad_list = [1; 3.2; 7];; 
                             ^^^ 
This expression has type float but is here 
used with type int
```

# Question

- Which one of these lists is invalid?
  1. \([2; 3; 4; 6]\)
  2. \([2,3; 4,5; 6,7]\)
  3. \([(2.3,4); (3.2,5); (6,7.2)]\)
  4. \(["hi"; "there"; ["wahcha"]]; [ ]; ["doin"]\)

- 3 is invalid because of last pair

# Answer

- Which one of these lists is invalid?
  1. \([2; 3; 4; 6]\)
  2. \([2,3; 4,5; 6,7]\)
  3. \([(2.3,4); (3.2,5); (6,7.2)]\)
  4. \(["hi"; "there"; ["wahcha"]]; [ ]; ["doin"]\)

- 3 is invalid because of last pair

# Functions Over Lists

```
# let rec double_up list = 
    match list 
    with 
      | [ ] -> [ ]  (* pattern before ->, 
expression after *)
      | (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; 3; 2; 1; 13; 8; 5; 3; 2; 1; 1]
```
Functions Over Lists

```ocaml
# let silly = double_up ['hi'; 'there'];;
val silly : string list = ['hi'; 'hi'; 'there'; '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'; 'there'; 'hi'; 'hi']
```

Question: Length of list

- Problem: write code for the length of the list
- How to start?

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

Question: Length of list

- Problem: write code for the length of the list
- What patterns should we match against?

```ocaml
let rec length l =
    match l with
        [] -> 0
    | (a :: bs) -> length bs + 1
```
**Question: Length of list**

- Problem: write code for the length of the list
  - What result do we give when \( l \) is not empty?

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

**Same Length**

- How can we efficiently answer if two lists have the same length?

\[
\text{let rec same_length } list1 \ list2 = \\
\text{match } list1 \text{ with } [] \rightarrow \\
\text{(match } list2 \text{ with } [] \rightarrow \text{true} \\
| (y :: ys) \rightarrow false \\
| (x :: xs) \rightarrow \\
\text{(match } list2 \text{ with } [] \rightarrow false \\
| (y :: ys) \rightarrow \text{same_length } xs \ ys) \\
\]

**Functions with more than one argument**

# let add_three \( x y z = x + y + z \);
val add_three : int \( \rightarrow \) int \( \rightarrow \) int \( \rightarrow \) int = \text{<fun>}

# let \( t = \text{add_three } 6 \ 3 \ 2 \);
val \( t \) : int = 11

# let \( \text{add_three } = \\
\text{fun } x \rightarrow (\text{fun } y \rightarrow (\text{fun } z \rightarrow x + y + z)); \\
\val add_three : int \( \rightarrow \) int \( \rightarrow \) int \( \rightarrow \) int = \text{<fun>}

Again, first syntactic sugar for second

**Partial application of functions**

let add_three \( x y z = x + y + z \);

# let \( h = \text{add_three } 5 \ 4 \);
val \( h \) : int \( \rightarrow \) int = \text{<fun>}

# \( h \ 3 \);
- : int = 12

# \( h \ 7 \);
- : int = 16
**Functions as arguments**

```ocaml
# let thrice f x = f (f (f x));;
val thrice : ('a -> 'a) -> 'a -> 'a = <fun>
# let g = thrice plus_two;;
val g : int -> int = <fun>
# g 4;;
- : int = 10
# thrice (fun s -> "Hi! " ^ s) "Good-bye!";;
- : string = "Hi! Hi! Hi! Good-bye!"
```

**Functions on tuples**

```ocaml
# let plus_pair (n,m) = n + m;;
val plus_pair : int * int -> int = <fun>
# plus_pair (3,4);;
- : int = 7
# let double x = (x,x);;
val double : 'a -> 'a * 'a = <fun>
# double 3;;
- : int * int = (3, 3)
# double "hi";;
- : string * string = ("hi", "hi")
```

**Closure for plus_pair**

- Assume $\rho_{plus\_pair}$ was the environment just before $plus\_pair$ defined
- Closure for $plus\_pair$:
  $$<(n,m) \rightarrow n + m, \rho_{plus\_pair}>$$
- Environment just after $plus\_pair$ defined:
  $$\{plus\_pair \rightarrow <(n,m) \rightarrow n + m, \rho_{plus\_pair}>, \rho_{plus\_pair}\}$$

**Recall: let plus_x = fun x => y + x**

- Let $x = 12$
- $X \rightarrow 12$
- $X \rightarrow 12$
- $\cdots$
- $plus\_x \rightarrow y \rightarrow y + x \rightarrow 12$
- $let\ plus\_x = fun\ y => y + x$
- $let\ x = 7$
- $y \rightarrow y + x \rightarrow 7$
- $\cdots$

**Closure for plus_x**

- When $plus\_x$ was defined, had environment:
  $$\rho_{plus\_x} = \{..., x \rightarrow 12, ...\}$$
- Recall: $let\ plus\_x\ y = y + x$
  is really $let\ plus\_x = fun\ y \rightarrow y + x$
- Closure for $fun\ y \rightarrow y + x$:
  $$<y \rightarrow y + x, \rho_{plus\_x}>$$
- Environment just after $plus\_x$ defined:
  $$\{plus\_x \rightarrow <y \rightarrow y + x, \rho_{plus\_x}>, \rho_{plus\_x}\}$$
Functions on tuples

# let plus_pair (n,m) = n + m;;
val plus_pair : int * int -> int = <fun>
# plus_pair (3,4);;
- : int = 7
# let double x = (x,x);;
val double : 'a -> 'a * 'a = <fun>
# double 3;;
- : int * int = (3, 3)
# double "hi";;
- : string * string = ("hi", "hi")

Closure for plus_pair

Assume ρ_{plus_pair} was the environment just before plus_pair defined

Closure for \( \text{fun (n,m) -> n + m} \):
\(<(n,m) \rightarrow n + m, \rho_{plus_pair} >\)

Environment just after plus_pair defined:
\( \{ plus_pair \rightarrow <(n,m) \rightarrow n + m, \rho_{plus_pair} > \} \)

+ ρ_{plus_pair}

Curried vs Uncurried

Recall
val add_three : int -> int -> int -> int = <fun>

How does it differ from
# let add_triple (u,v,w) = u + v + w;;
val add_triple : int * int * int -> int = <fun>

add_three is \textit{curried};
add_triple is \textit{uncurried}

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:
\(< (v_1, \ldots, v_n) \rightarrow \text{expr}, \rho >\)

Where ρ is the environment in effect when the function is defined (for a simple function)

Functions with more than one argument

# let add_three x y z = x + y + z;;
val add_three : int -> int -> int -> int = <fun>
# let t = add_three 6 3 2;;
val t : int = 11
# let add_three =
fun x -> (fun y -> (fun z -> x + y + z));;
val add_three : int -> int -> int -> int = <fun>

Again, first syntactic sugar for second

Curried vs Uncurried

# add_triple (6,3,2);;
- : int = 11
# add_triple 5 4;;
Characters 0-10:
add_triple 5 4;;
^~~~~~~~~~~~~~~

This function is applied to too many arguments, maybe you forgot a `;`
# fun x -> add_triple (5,4,x);;
: int -> int = <fun>
Partial application of functions

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

```ml
# let h = add_three 5 4;;
val h : int -> int = <fun>
# h 3;;
- : int = 12
# h 7;;
- : int = 16
```

Your turn now

Try later parts from WA1

Caution!

Know what the argument is and what the body is

Functions as arguments

```ml
# let thrice f x = f (f (f x));;
val thrice : ('a -> 'a) -> 'a -> 'a = <fun>
# let g = thrice plus_two;;
val g : int -> int = <fun>
# g 4;;
- : int = 10
# thrice (fun s -> "Hi! " ^ s) "Good-bye!";;
- : string = "Hi! Hi! Hi! Good-bye!"
```

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 \( x \) \( \rightarrow v \):
    \( \{ x \rightarrow v \} + \rho \)

- 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 \( \rho \) (\( \rho(v) \))
- To evaluate uses of \( +, \_ \), etc, eval args, then do operation
- Function expression evaluates to its closure
- To evaluate a local dec: \( \text{let } x = e_1 \) in \( e_2 \)
  - Eval \( e_1 \) to \( v \), eval \( e_2 \) using \( \{ x \rightarrow v \} + \rho \)

Evaluation of Application with Closures

- In environment \( \rho \), evaluate left term to closure, \( c = <(x_1,...,x_n) \rightarrow b, \rho> \)
- \( (x_1,...,x_n) \) variables in (first) argument
- Evaluate the right term to value \( (v_{1,2}...,v_n) \)
- Update the environment \( \rho \) to \( \rho' = \{ x_1 \rightarrow v_{1,2}..., x_n \rightarrow v_n \} + \rho \)
- Evaluate body \( b \) in environment \( \rho' \)
Evaluation of Application of plus_x;;

- Have environment:
  \( \rho = \{ \text{plus}_x \rightarrow \langle y \rightarrow y + x, \rho_{\text{plus}_x} \rangle, \ldots, y \rightarrow 3, \ldots \} \)
  
  where \( \rho_{\text{plus}_x} = \{ \text{x} \rightarrow 12, \ldots \} \)

- \( \text{Eval} (\text{plus}_x \text{ y}, \rho) \) rewrites to
- \( \text{App} (\langle y \rightarrow y + x, \rho_{\text{plus}_x} \rangle, 3) \) rewrites to
- \( \text{Eval} (y + x, \{y \rightarrow 3\} + \rho_{\text{plus}_x}) \) rewrites to
- \( \text{Eval} (3 + 12, \rho_{\text{plus}_x}) = 15 \)

Closure question

- If we start in an empty environment, and we execute:
  
  let \( f = \text{fun n} \rightarrow n + 5;; \)
  
  (* 0 *)
  
  let \( \text{pair}_\text{map} \text{ g (n,m)} = (g \text{ n, g m});; \)
  
  let \( f = \text{pair}_\text{map} f;; \)
  
  let \( a = f \ (4,6);;; \)
  
  What is the environment at (* 0 *)?

Answer

\( \rho_0 = \{ f \rightarrow \langle n \rightarrow n + 5, \{ \} \rangle \} \)

Closure question

- If we start in an empty environment, and we execute:
  
  let \( f = \text{fun => n} \rightarrow n + 5;; \)
  
  (* 1 *)
  
  let \( \text{pair}_\text{map} \text{ g (n,m)} = (g \text{ n, g m});; \)
  
  (* 1 *)
  
  let \( f = \text{pair}_\text{map} f;; \)
  
  let \( a = f \ (4,6);;; \)
  
  What is the environment at (* 1 *)?

Answer

\( \rho_0 = \{ f \rightarrow \langle n \rightarrow n + 5, \{ \} \rangle \} \)

let \( \text{pair}_\text{map} \text{ g (n,m)} = (g \text{ n, g m});; \)

\( \rho_1 = \{ \text{pair}_\text{map} \rightarrow \langle g \rightarrow \text{fun (n,m)} \rightarrow (g \text{ n, g m}), \}

\{ f \rightarrow \langle n \rightarrow n + 5, \{ \} \rangle \rangle, \}

\{ f \rightarrow \langle n \rightarrow n + 5, \{ \} \rangle \rangle \} \)
Closure question

If we start in an empty environment, and we execute:

```plaintext
let f = fun => n + 5;;
let pair_map g (n,m) = (g n, g m);;
let f = pair_map f;;

(* 2 *)
let a = f (4,6);;
```

What is the environment at (* 2 *)?
Closure question

If we start in an empty environment, and we execute:

```ml
let f = fun => n => n + 5;;
let pair_map g (n,m) = (g n, g m);;
let f = pair_map f;;
let a = f (4,6);;
```

What is the environment at (* 3 *)?
Higher Order Functions

- A function is **higher-order** if it takes a function as an argument or returns one as a result.
- Example:
  ```haskell
  let compose f g = fun x -> f (g x);;
  val compose : ('a -> 'b) -> ('c -> 'a) -> 'c -> 'b = <fun>
  ``
- The type `(a -> 'b) -> ('c -> 'a) -> 'c -> 'b` is a higher order type because of `(a -> 'b)` and `(c -> 'a)` and `-> 'c -> 'b`.

Thrice

- Recall:
  ```haskell
  let thrice f x = f (f (f x));;
  val thrice : ('a -> 'a) -> 'a -> 'a = <fun>
  ``
- How do you write thrice with compose?
  ```haskell
  let thrice f = compose f (compose f f);;
  val thrice : ('a -> 'a) -> 'a -> 'a = <fun>
  ``
- Is this the only way?

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**

Lambda Lifting

- You must remember the rules for evaluation when you use partial application.
- Example:
  ```haskell
  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>
  ``
- Lambda lifting delayed the evaluation of the argument to `(+)` until the second argument was supplied.
Partial Application and “Unknown Types”

- Recall: compose plus_two:
  ```
  # 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?

- '_a can only be instantiated once for an expression
  ```
  # f1 plus_two;;
  - : int -> int = <fun>
  # f1 List.length;;
  Characters 3-14:
  f1 List.length;;
  ^^^^^^^^^^^^^^^
  This expression has type 'a list -> int but is here used with type int -> int
  ```

- 'a can be repeatedly instantiated
  ```
  # f2 plus_two;;
  - : int -> int = <fun>
  # f2 List.length;;
  - : '_a list -> int = <fun>
  ```

Functions Over Lists

- 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 list -> 'a = <fun>
  # fold_left
    (fun () -> print_string)
    ()
    ["hi"; "there"];;
  hithere- : unit = ()
  ```

Iterating over lists

- Iterating over lists:
  ```
  # let rec fold_right f b list =
    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 = ()
  ```
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.
- 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>
# double_up [2;3;4];;
- : int list = [4; 6; 8]
```

Encoding Recursion with Fold

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

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

```ocaml
# let rec append list1 list2 = fold_right (fun x y -> x :: y) list1 list2;;
val append : 'a list -> 'a list -> 'a list = <fun>
# append [1;2;3] [4;5;6];;
val result = [1; 2; 3; 4; 5; 6]
```

Mapping Recursion

- One common form of structural recursion applies a function to each element in the structure.
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 Recursion
- 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
```

How long will it take?
- Remember the big-O notation from CS 225 and CS 273
- Question: given input of size \( n \), how long to generate output?
- Express output time in terms of input size, omit constants and take biggest power

Common big-O times:
- Constant time \( O(1) \)
  - input size doesn’t matter
- Linear time \( O(n) \)
  - double input \( \Rightarrow \) double time
- Quadratic time \( O(n^2) \)
  - double input \( \Rightarrow \) quadruple time
- Exponential time \( O(2^n) \)
  - increment input \( \Rightarrow \) double 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>
```

Exponential running time

- Hideous running times on input of any size
- Each step of recursion takes constant time
- Each recursion makes two recursive calls
- Easy to write naïve code that is exponential for functions that can be linear

```ocaml
# let rec naiveFib n = match n
   with 0 -> 0
        | 1 -> 1
        | _ -> naiveFib (n-1) + naiveFib (n-2);;
val naiveFib : int -> int = <fun>
```

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)?

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

```ocaml
Tail call

h
f
...
```
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

```
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]
```

```
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]
```

Folding Functions over Lists

```
# 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
```

Folding - Forward Recursion

```
# let sumlist list = fold_right (+) list 0;;  
val sumlist : int list -> int = <fun>

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

# let prodlist list = fold_right ( * ) list 1;;  
val prodlist : int list -> int = <fun>

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

- # let rev list =
-   fold_left
-   (fun l -> 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