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 :: []
  - [x1; x2; ...; xn] == x1 :: x2 :: ... :: xn :: []

Lists are Homogeneous

# 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]

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”]]

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; 5; 3; 3; 2; 2; 1; 1; 1; 1; 1]  
```

# Functions Over Lists

```
# 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"]  
```

# Structural Recursion

- Functions on recursive datatypes (eg 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

# Question: Length of list

- Problem: write code for the length of the list
  - How to start?
  
```
let rec length l =  
  match l with  
```

# Question: Length of list

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

```
let rec length l =  
  match l with  
```
Question: Length of list

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

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

9/16/19

Question: Length of list

- Problem: write code for the length of the list
- What result do we give when \( l \) is empty?

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

9/16/19

Question: Length of list

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

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

9/16/19

---

Structural Recursion: List Example

```ml
# 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
```

9/16/19

---

Same Length

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

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

```ml
let rec same_length list1 list2 =  
  match list1 with [] ->  
  (match list2 with [] -> true  
   | (y::ys) -> false)  
  | (x::xs) ->  
    (match list2 with [] -> false  
     | (y::ys) -> same_length xs ys)
```

Higher-Order Functions Over Lists

```ml
# 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]
```

Iterating over lists

```ml
# 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"];;
  hithere- : unit = ()
```

Recursing over lists

```ml
# 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 = ()
```

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

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

Forward Recursion: Examples

```null
```
Forward Recursion: Examples

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

```
Base Case Operator Recursive Call
```

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

```
Base Case Operator Recursive Call
```

Encoding Forward Recursion with Fold

```
# 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
```

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

Mapping Recursion

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

```
# 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]
```

Mapping Recursion

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

```
# 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

```
# 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

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

```
# 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
```

- Computes (2 * (4 * (6 * 1)))
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
```

How long will it take?

- Remember the big-O notation from CS 225 and CS 374
- 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 ⇒ double time
- Quadratic time \( O(n^2) \)
  - double input ⇒ quadruple time
- Exponential time \( O(2^n) \)
  - increment input ⇒ double time

Linear Time

- Expect most list operations to take linear time \( O(n) \)
- Each step of the recursion can be done in constant time
- Each step makes only one recursive call
- List example: multList, append
- Integer example: factorial

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

- Poor worst-case 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
Exponential running time

```ocaml
# let rec slow n =  
  if n <= 1  
  then 1  
  else 1 + slow (n-1) + slow(n-2);;
val slow : int -> int = <fun>
# List.map slow [1;2;3;4;5;6;7;8;9];;
- : int list = [1; 3; 5; 9; 15; 25; 41; 67; 109]
```

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.

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 = <fun>
# let rev list = rev_aux list [ ];;
val 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]`

What is its running time?
Comparison

- \text{rev} [1,2,3] = 
- \text{rev\_aux} [1,2,3] [ ] = 
- \text{rev\_aux} [2,3] [1] = 
- \text{rev\_aux} [3] [2,1] = 
- \text{rev\_aux} [ ] [3,2,1] = [3,2,1]

Folding Functions over Lists

How are the following functions similar?

\begin{verbatim}
# 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 prodlst list = match list with
  | [] -> 1 | x::xs -> x * prodlst xs;;
val prodlst : int list -> int = <fun>
# prodlst [2;3;4];;
- : int = 24
\end{verbatim}

Folding

\begin{verbatim}
# 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)...))
\end{verbatim}

Folding - Forward Recursion

\begin{verbatim}
# let sumlist list = fold_right (+) list 0;;
val sumlist : int list -> int = <fun>
# sumlist [2;3;4];;
- : int = 9
# let prodlst list = fold_right ( *) list 1;;
val prodlst : int list -> int = <fun>
# prodlst [2;3;4];;
- : int = 24
\end{verbatim}

Folding - Tail Recursion

\begin{verbatim}
- # let rec rev list =
-         fold_left
-         (fun l -> fun x -> x :: l)     //comb op
-         []                         //accumulator cell
-         list
\end{verbatim}

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
Continuations

- 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

Continuation Passing Style

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

Why CPS?

- Makes order of evaluation explicitly clear
- Allocates variables (to become registers) for each step of computation
- Essentially converts functional programs into imperative ones
  - Major step for compiling to assembly or byte code
- Tail recursion easily identified
- Strict forward recursion converted to tail recursion
  - At the expense of building large closures in heap

Other Uses for Continuations

- CPS designed to preserve order of evaluation
- Continuations used to express order of evaluation
- Can be used to change order of evaluation
- Implements:
  - Exceptions and exception handling
  - Co-routines
  - (pseudo, aka green) threads
Example

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

Simple function using a continuation:
# let addk (a, b) k = k (a + b);
val addk : int * int -> (int -> 'a) -> 'a = <fun>
# addk (22, 20) report;
2
- : 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>

Nested Continuations

# let add_triple (x, y, z) = (x + y) + z;;
val add_triple : int * int * int -> int = <fun>
# let add_triple (x, y, z) = let p = x + y in p + z;;
val add_three : int * int -> int -> int = <fun>
# let add_triple_k (x, y, z) k =
  addk (x, y) (fun p -> addk (p, z) k );
val add_triple_k: int * int * int -> (int -> 'a) -> 
  'a = <fun>

Try Problem 7 on MP2
Try consk

add_three: a different order

# let add_triple (x, y, z) = x + (y + z);;
How do we write add_triple_k to use a different order?
let add_triple_k (x, y, z) k =