Discussion 2

September 6

Logistics

- A1 released, due Friday, September 13 at 8:00 PM
  - All coding in Hazel
  - Autograder is just a sanity check!
- Preliminary survey also due September 13
  - 1% of your final grade
- Come to OH if you have questions!

Functions Recap

- Functions take a single argument and produce a single value. For multiple inputs, you can use a tuple

- Recursive functions can be used to implement nonlinear control flow, like loops

- Functions can take functions as arguments and return functions

Functions Recap

- Functions take a single argument and produce a single value. For multiple inputs, you can use a tuple

- Recursive functions can be used to implement nonlinear control flow like loops

- **Functions can take functions as arguments and return functions**

Q: *What do we call such functions?*

Higher-order Functions

Higher-order Functions

- A **higher-order function** takes one or more functions, or returns one or more functions

- Functions taking functions can be thought of as a kind of **functional design pattern**

Q: *Which higher-order functions have you seen before this class? You've probably seen at least one*

Higher-order Functions: Sort

```
// Not higher order... yet!

let ??? = fun xs ->
  let insert = fun x, xs ->
    case xs
    | [] => [x]
    | hd::tl =>
      if x < hd
      then hd::insert(x, tl)
      else x::hd::tl end in
  case xs
  | [] => []
  | hd::tl => insert(hd, ???(tl)) end in
…
```

Q: *What does this function do?*

Higher-order Functions: Sort

```
// Not higher order: descending sort on integers

let sort = fun xs ->
  let insert = fun x, xs ->
    case xs
    | [] => [x]
    | hd::tl =>
      if x < hd
      then hd::insert(x, tl)
      else x::hd::tl end in
  case xs
  | [] => []
  | hd::tl => insert(hd, sort(tl)) end in
…
```

Higher-order Functions: Sort

```
// Not higher order: ascending sort on integers

let sort = fun xs ->
  let insert = fun x, xs ->
    case xs
    | [] => [x]
    | hd::tl =>
      if x > hd
      then hd::insert(x, tl)
      else x::hd::tl end in
  case xs
  | [] => []
  | hd::tl => insert(hd, sort(tl)) end in
…
```

Higher-order Functions: Sort

```
// Not higher order: ascending lexical sort on integer pairs

let sort = fun xs ->
  let insert = fun (x0, x1), xs ->
    case xs
    | [] => [x]
    | (hd0, hd1)::tl =>
      if x0 > hd0 || x0 == hd0 && x1 > hd1
      then hd::insert(x, tl)
      else x::hd::tl end in
  case xs
  | [] => []
  | hd::tl => insert(hd, sort(tl)) end in
…
```

Q: *This one doesn't quite follow the previous pattern; how can we tweak it to do so?*

Higher-order Functions: Sort

```
// Not higher order

let sort = fun xs ->
  let insert = fun x, xs ->
    case xs
    | [] => [x]
    | hd::tl =>
      if let (x₀, x₁) = x in
         let (hd₀, hd₁) = hd in
         x₀ > hd₀ || x₀ == hd₀ && x₁ > hd₁
      then hd::insert(x, tl)
      else x::hd::tl end in
  case xs
  | [] => []
  | hd::tl => insert(hd, sort(tl)) end in
…
```

Higher-order Functions: Sort

```
// Not higher order

let sort = fun xs ->
  let insert = fun x, xs ->
    case xs
    | [] => [x]
    | hd::tl =>
      if let (x0, x1) = x in
         let (hd0, hd1) = hd in
         x0 > hd0 || x0 == hd0 && x1 > hd1
      then hd::insert(x, tl)
      else x::hd::tl end in
  case xs
  | [] => []
  | hd::tl => insert(hd, sort(tl)) end in
…
```

Q: *How do we avoid rewriting this function for each way we want to sort each data type?*

Higher-order Functions: Sort

```
let comp : Bool =
  let (x0, x1) = x in
  let (hd0, hd1) = hd in
  x0 > hd0 || x0 == hd0 && x1 > hd1 in

let sort = fun comp, xs ->
  let insert = fun x, xs ->
    case xs
    | [] => [x]
    | hd::tl =>
      if comp
      then hd::insert(x, tl)
      else x::hd::tl end in
  case xs
  | [] => []
  | hd::tl => insert(hd, sort(tl)) end in
```

Q: *How about this? Does this work?*

Higher-order Functions: Sort

```
// Not higher order

let sort = fun xs ->
  let insert = fun x, xs ->
    case xs
    | [] => [x]
    | hd::tl =>
      if let (x0, x1) = x in
         let (hd0, hd1) = hd in
         x0 > hd0 || x0 == hd0 && x1 > hd1
      then hd::insert(x, tl)
      else x::hd::tl end in
  case xs
  | [] => []
  | hd::tl => insert(hd, sort(tl)) end in
…
```

```
let sort = fun xs ->
  let insert = fun x, xs ->
    case xs
    | [] => [x]
    | hd::tl =>
      if x < hd
      then hd::insert(x, tl)
      else x::hd::tl end in
  case xs
  | [] => []
  | hd::tl => insert(hd, sort(tl)) end in
…
```

```
let sort = fun xs ->
  let insert = fun x, xs ->
    case xs
    | [] => [x]
    | hd::tl =>
      if x > hd
      then hd::insert(x, tl)
      else x::hd::tl end in
  case xs
  | [] => []
  | hd::tl => insert(hd, sort(tl)) end in
…
```

Higher-order Functions: Sort

```
// Not higher order

let sort = fun xs ->
  let insert = fun x, xs ->
    case xs
    | [] => [x]
    | hd::tl =>
      if (fun (a, b) ->
            let (a0, a1) = a in
            let (b0, b1) = b in
            a0 > b0 || a0 == b0 && a1 > b1)
          (x, hd)
      then hd::insert(x, tl)
      else x::hd::tl end in
  case xs
  | [] => []
  | hd::tl => insert(hd, sort(tl)) end in
…
```

```
let sort = fun xs ->
  let insert = fun x, xs ->
    case xs
    | [] => [x]
    | hd::tl =>
      if (fun (a, b) -> a < b)(x, hd)
      then hd::insert(x, tl)
      else x::hd::tl end in
  case xs
  | [] => []
  | hd::tl => insert(hd, sort(tl)) end in
…
```

```
let sort = fun xs ->
  let insert = fun x, xs ->
    case xs
    | [] => [x]
    | hd::tl =>
      if (fun (a, b) -> a > b)(x, hd)
      then hd::insert(x, tl)
      else x::hd::tl end in
  case xs
  | [] => []
  | hd::tl => insert(hd, sort(tl)) end in
…
```

Higher-order Functions: Sort

```
// Not higher order!

let sort = fun comp, xs ->
  let insert = fun x, xs ->
    case xs
    | [] => [x]
    | hd::tl =>
      if comp(x, hd)
      then hd::insert(x, tl)
      else x::hd::tl end in
  case xs
  | [] => []
  | hd::tl => insert(hd, sort(tl)) end in
…
```

```
let sort = fun comp, xs ->
  let insert = fun x, xs ->
    case xs
    | [] => [x]
    | hd::tl =>
      if comp(x, hd)
      then hd::insert(x, tl)
      else x::hd::tl end in
  case xs
  | [] => []
  | hd::tl => insert(hd, sort(tl)) end in
…
```


```
let sort = fun comp, xs ->
  let insert = fun x, xs ->
    case xs
    | [] => [x]
    | hd::tl =>
      if comp(x, hd)
      then hd::insert(x, tl)
      else x::hd::tl end in
  case xs
  | [] => []
  | hd::tl => insert(hd, sort(tl)) end in
…
```

Other Higher-order Functions

- Differentiation / Integration from calculus
- Callbacks (e.g. for database or API calls)
- Recursion schemes (e.g. fold/reduce, traversals, map, filter)

Map

Map: Motivation

```
let ???: [Int] -> [Int] =
  fun xs ->
    case xs
    | [] => []
    | hd::tl => (hd + 1) :: ???(tl) end in
…
```

Q: *What does this function return, given [1, 2, 3]?*

Map: Motivation

```
let incr_all: [Int] -> [Int] =
  fun xs ->
    case xs
    | [] => []
    | hd::tl => (hd + 1) :: incr_all(tl) end in
…

let ???: [Int] -> [Bool] =
  fun xs ->
    case xs
    | [] => []
    | hd::tl => (hd != 0) :: ???(tl) end in
…
```

Q: *What does this function return, given [0, 1, 2]?*

Map: Motivation

```
let incr_all: [Int] -> [Int] =
  fun xs ->
    case xs
    | [] => []
    | hd::tl => (hd + 1) :: incr_all(tl) end in
…

let to_bools: [Int] -> [Bool] =
  fun xs ->
    case xs
    | [] => []
    | hd::tl => (hd != 0) :: to_bools(tl) end in
…

let ???: [String] -> [Int] =
  fun xs ->
    case xs
    | [] => []
    | hd::tl => string_length(hd) :: ???(tl) end in
…
```

Q: *What does this function return, given [“aaa”, “bb”, “c”]*

Map: Motivation

```
let incr_all: [Int] -> [Int] =
  fun xs ->
    case xs
    | [] => []
    | hd::tl => (hd + 1) :: incr_all(tl) end in
…

let to_bools: [Int] -> [Bool] =
  fun xs ->
    case xs
    | [] => []
    | hd::tl => (hd != 0) :: to_bools(tl) end in
…

let lengths: [String] -> [Int] =
  fun xs ->
    case xs
    | [] => []
    | hd::tl => string_length(hd) :: lengths(tl) end in
…
```

Q: What do all these functions have in common? What are their differences?

Map: Motivation

```
let incr_all: [Int] -> [Int] =
  fun xs ->
    case xs
    | [] => []
    | hd::tl => (hd + 1) :: incr_all(tl) end in
…

let to_bools: [Int] -> [Bool] =
  fun xs ->
    case xs
    | [] => []
    | hd::tl => (hd != 0) :: to_bools(tl) end in
…

let lengths: [String] -> [Int] =
  fun xs ->
    case xs
    | [] => []
    | hd::tl => string_length(hd) :: lengths(tl) end in
…
```

Map: Mappish

```
let mappish =
  fun xs ->
    case xs
    | [] => []
    | hd::tl => (hd + 1) :: map_ish(tl) end in
…
```

```
let mappish =
  fun xs ->
    case xs
    | [] => []
    | hd::tl => (hd != 0) :: map_ish(tl) end in
…
```

```
let mappish =
  fun xs ->
    case xs
    | [] => []
    | hd::tl => string_length(hd) :: map_ish(tl) end in
…
```

Map: Mappier

```
let mappier =
  fun xs ->
    case xs
    | [] => []
    | hd::tl => (fun x -> x + 1)(hd) :: mappier(tl) end in
…
```

```
let mappier =
  fun xs ->
    case xs
    | [] => []
    | hd::tl => (fun x -> x != 0)(hd) :: mappier(tl) end in
…
```

```
let mappier =
  fun xs ->
    case xs
    | [] => []
    | hd::tl => string_length(hd) :: mappier(tl) end in
…
```

Map: Mappiest

```
let map =
  fun f, xs ->
    case xs
    | [] => []
    | hd::tl => f(hd) :: map(f, tl) end in
…
```

```
let map =
  fun f, xs ->
    case xs
    | [] => []
    | hd::tl => f(hd) :: map(f, tl) end in
…
```

```
let map =
  fun f, xs ->
    case xs
    | [] => []
    | hd::tl => f(hd) :: map(f, tl) end in
…
```

Map

```
let map =
  fun f, xs ->
    case xs
    | [] => []
    | hd::tl => f(hd) :: map(f, tl) end in
…

let incr_all = ?? in
let to_bools = ?? in
let lengths = ?? in
…
```

Q: *So how can we write our original functions using map?*

Map

```
let map =
  fun f, xs ->
    case xs
    | [] => []
    | hd::tl => f(hd) :: map(f, tl) end in
…


let incr_all = fun xs -> ?? in
let to_bools = fun xs -> ?? in
let lengths = fun xs -> ?? in
…
```

Map

```
let map =
  fun f, xs ->
    case xs
    | [] => []
    | hd::tl => f(hd) :: map(f, tl) end in
…

let incr_all = fun xs -> map(??, xs) in
let to_bools = fun xs -> map(??, xs) in
let lengths = fun xs -> map(??, xs) in
…
```

Q: *What function do we want to pass for incr_all?*

Map

```
let map =
  fun f, xs ->
    case xs
    | [] => []
    | hd::tl => f(hd) :: map(f, tl) end in
…


let incr_all = fun xs -> map(fun x -> x + 1, xs) in
let to_bools = fun xs -> map(??, xs) in
let lengths = fun xs -> map(??, xs) in
…
```

Q: *How about for to_bools?*

Map

```
let map =
  fun f, xs ->
    case xs
    | [] => []
    | hd::tl => f(hd) :: map(f, tl) end in
…


let incr_all = fun xs -> map(fun x -> x + 1, xs) in
let to_bools = fun xs -> map(fun x -> x == 1, xs) in
let lengths = fun xs -> map(??, xs) in
…
```

Q: *And for string_lengths?*

Map

```
let map =
  fun f, xs ->
    case xs
    | [] => []
    | hd::tl => f(hd) :: map(f, tl) end in
…


let incr_all = fun xs -> map(fun x -> x + 1, xs) in
let to_bools = fun xs -> map(fun x -> x != 0, xs) in
let lengths = fun xs -> map(string_length, xs) in
…
```

fold_right

fold_right

```
let fold_right: ((A, B) -> B, [A], B) -> B =
  fun f, xs, b ->
    case xs
      | [] => b
      | hd::tl =>
        f(hd, fold_right(f, tl, b)) end in
```

- Cyrus already showed in class how fold_right can be abstracted from examples as we did with sort and map

- Instead, let’s break from our previous pattern and instead try re-writing some recursive functions as right folds

fold_right: ??

```
let fold_right: ((A, B) -> B, [A], B) -> B =
  fun f, xs, b ->
    case xs
      | [] => b
      | hd::tl =>
        f(hd, fold_right(f, tl, b)) end in

let ??: [String] -> String =
  fun xs ->
    case xs
      | [] => ""
      | hd::tl =>
        hd ++ ??(tl) end in
```

Q: *What does this function return given [“hello”, “ “, “world”]?*

fold_right: concat

```
let fold_right: ((A, B) -> B, [A], B) -> B =
  fun f, xs, b ->
    case xs
      | [] => b
      | hd::tl =>
        f(hd, fold_right(f, tl, b)) end in

let concat: [String] -> String =
  fun xs ->
    case xs
      | [] => ""
      | hd::tl =>
        hd ++ concat(tl) end in

let concat: [String] -> String =
  fun xs -> fold_right(
    ??,
    ??,
    ??) in
```

Q: *Let's start off easy: what are the second two arguments?*

fold_right: concat

```
let fold_right: ((A, B) -> B, [A], B) -> B =
  fun f, xs, b ->
    case xs
      | [] => b
      | hd::tl =>
        f(hd, fold_right(f, tl, b)) end in

let concat: [String] -> String =
  fun xs ->
    case xs
      | [] => “”
      | hd::tl =>
        hd ++ concat(tl) end in

let concat: [String] -> String =
  fun xs -> fold_right(
    ??,
    xs,
    “”) in
```

Q: *Now what goes here? Remember to check the type*

fold_right: concat

```
let fold_right: ((A, B) -> B, [A], B) -> B =
  fun f, xs, b ->
    case xs
      | [] => b
      | hd::tl =>
        f(hd, fold_right(f, tl, b)) end in

let concat: [String] -> String =
  fun xs ->
    case xs
      | [] => “”
      | hd::tl =>
        hd ++ concat(tl) end in

let concat: [String] -> String =
  fun xs -> fold_right(
    fun x, ?? -> ??,
    xs,
    “”) in
```

Q: *What’s a good name for our second argument?*

fold_right: concat

```
let fold_right: ((A, B) -> B, [A], B) -> B =
  fun f, xs, b ->
    case xs
      | [] => b
      | hd::tl =>
        f(hd, fold_right(f, tl, b)) end in

let concat: [String] -> String =
  fun xs ->
    case xs
      | [] => “”
      | hd::tl =>
        hd ++ concat(tl) end in

let concat: [String] -> String =
  fun xs -> fold_right(
    fun x, concat_of_rest -> ??,
    xs,
    “”) in
```

fold_right: concat

```
let fold_right: ((A, B) -> B, [A], B) -> B =
  fun f, xs, b ->
    case xs
      | [] => b
      | hd::tl =>
        f(hd, fold_right(f, tl, b)) end in

let concat: [String] -> String =
  fun xs ->
    case xs
      | [] => “”
      | hd::tl =>
        hd ++ concat(tl) end in

let concat: [String] -> String =
  fun xs -> fold_right(
    fun x, concat_of_rest -> x ++ concat_of_rest,
    xs,
    “”) in
```

fold_right: ??

```
let fold_right: ((A, B) -> B, [A], B) -> B =
  fun f, xs, b ->
    case xs
      | [] => b
      | hd::tl =>
        f(hd, fold_right(f, tl, b)) end in

let ??: [Int] -> [Int] =
  fun xs ->
    case xs
      | [] => []
      | hd::tl =>
        if hd < 2 then hd::hd::??(tl) else hd::??(tl) end in
```

Q: *What does this function return given [1, 2, 3, 0, 5]?*

fold_right: repeat_under_2s

```
let fold_right: ((A, B) -> B, [A], B) -> B =
  fun f, xs, b ->
    case xs
      | [] => b
      | hd::tl =>
        f(hd, fold_right(f, tl, b)) end in

let repeat_under_2s: [Int] -> [Int] =
  fun xs ->
    case xs
      | [] => []
      | hd::tl =>
        if hd < 2 then hd::hd::repeat_under_2s(tl) else hd::repeat_under_2s(tl) end in

let repeat_under_2s: [Int] -> [Int] =
  fun xs -> fold_right(
    ??,
    ??,
    ??) in
```

Q: *What are the second two arguments? Careful, as this time they have the same type.*

fold_right: repeat_under_2s

```
let fold_right: ((A, B) -> B, [A], B) -> B =
  fun f, xs, b ->
    case xs
      | [] => b
      | hd::tl =>
        f(hd, fold_right(f, tl, b)) end in

let repeat_under_2s: [Int] -> [Int] =
  fun xs ->
    case xs
      | [] => []
      | hd::tl =>
        if hd < 2 then hd::hd::repeat_under_2s(tl) else hd::repeat_under_2s(tl) end in

let repeat_under_2s: [Int] -> [Int] =
  fun xs -> fold_right(
    ??,
    xs,
    []) in
```

fold_right: repeat_under_2s

```
let fold_right: ((A, B) -> B, [A], B) -> B =
  fun f, xs, b ->
    case xs
      | [] => b
      | hd::tl =>
        f(hd, fold_right(f, tl, b)) end in

let repeat_under_2s: [Int] -> [Int] =
  fun xs ->
    case xs
      | [] => []
      | hd::tl =>
        if hd < 2 then hd::hd::repeat_under_2s(tl) else hd::repeat_under_2s(tl) end in

let repeat_under_2s: [Int] -> [Int] =
  fun xs -> fold_right(
    fun x, repeat_of_rest -> ??,
    xs,
    []) in
```

fold_right: repeat_under_2s

```
let fold_right: ((A, B) -> B, [A], B) -> B =
  fun f, xs, b ->
    case xs
      | [] => b
      | hd::tl =>
        f(hd, fold_right(f, tl, b)) end in

let repeat_under_2s: [Int] -> [Int] =
  fun xs ->
    case xs
      | [] => []
      | hd::tl =>
        if hd < 2 then hd::hd::repeat_under_2s(tl) else hd::repeat_under_2s(tl) end in

let repeat_under_2s: [Int] -> [Int] =
  fun xs -> fold_right(
    fun x, repeat_of_rest -> if x < 2 then x::x::repeat_of_rest else x::repeat_of_rest,
    xs,
    []) in
```

fold_right: ??

```
let fold_right: ((A, B) -> B, [A], B) -> B =
  fun f, xs, b ->
    case xs
      | [] => b
      | hd::tl =>
        f(hd, fold_right(f, tl, b)) end in

let ??: [Int] -> Int  =
  fun xs ->
    case xs
      | [] => max_int
      | hd::tl =>
        if hd < ??(tl) then hd else ??(tl) end in
```

Q: *What does this function return given [7, 2, 1, 3]?*

fold_right: min

```
let fold_right: ((A, B) -> B, [A], B) -> B =
  fun f, xs, b ->
    case xs
      | [] => b
      | hd::tl =>
        f(hd, fold_right(f, tl, b)) end in

let min: [Int] -> Int  =
  fun xs ->
    case xs
      | [] => max_int
      | hd::tl =>
        if hd < min(tl) then hd else min(tl) end in

let min: [Int] -> Int =
  fun xs -> fold_right(
    ??,
    ??,
    ??) in
```

fold_right: min

```
let fold_right: ((A, B) -> B, [A], B) -> B =
  fun f, xs, b ->
    case xs
      | [] => b
      | hd::tl =>
        f(hd, fold_right(f, tl, b)) end in

let min: [Int] -> Int  =
  fun xs ->
    case xs
      | [] => max_int
      | hd::tl =>
        if hd < min(tl) then hd else min(tl) end in

let min: [Int] -> Int =
  fun xs -> fold_right(
    fun x, min_of_rest -> if x < min_of_rest then x else min_of_rest end
    max_int,
    b) in
```

fold_left

fold_right revisited

```
let fold_right: ((A, B) -> B, [A], B) -> B =
  fun f, xs, b ->
    case xs
      | [] => b
      | hd::tl =>
        f(hd, fold_right(f, tl, b)) end in
```

$$\texttt{fold\_right(f, } x_1\texttt{:: … ::}x_n\texttt{::[], base)}$$

$$\equiv$$

$$\texttt{f(}x_1\texttt{, f(}x_2\texttt{, … f(}x_n\texttt{, base) …))}$$

- Since we evaluate arguments before calling, f isn't called on the leftmost element until the recursive call is complete; f 'sees' the elements in opposite order
- This means that, although the leftmost cons cell is the first to begin being processed, it is the last one to finish
- In this sense, the fold proceeds 'from the right', or maybe more precisely: **from the bottom, up** (regarding the structure of cons cells as a tree)
- In imperativeland, think while loop + stack

Can we fold another way?

- What if instead we called f before making the recursive call? This means that the leftmost element is the first to begin **and** finish, and so this fold proceeds ‘from the left’ or **from the top of the cons tree, down** (imperativeland: while loop + queue)

- From the cons tree perspective, a right fold is a post-order traversal, and a left-fold is a pre-order traversal

```
let fold_left: ((B, A) -> B, B, [A]) -> B =
  fun f, acc, xs ->
    case xs
      | [] => acc
      | hd::tl =>
        fold_left(f, f(acc, hd), tl) in
```

$$\texttt{fold\_left(f, init, } x_1\texttt{:: … ::} x_n\texttt{::[])}$$

$$=$$

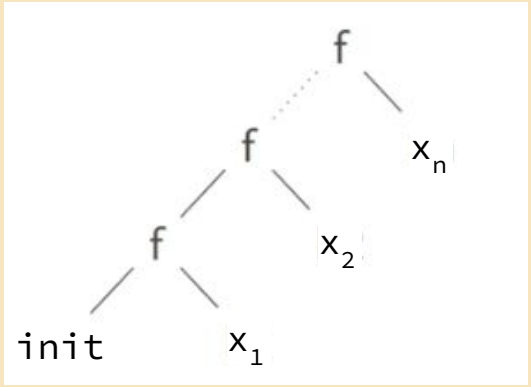
$$\texttt{f(… f(f(init, } x_1\texttt{), } x_2\texttt{) …, } x_n\texttt{)}$$

fold_left versus fold_right

fold_left(f, init, x_1:: … :: x_n::[])

≡

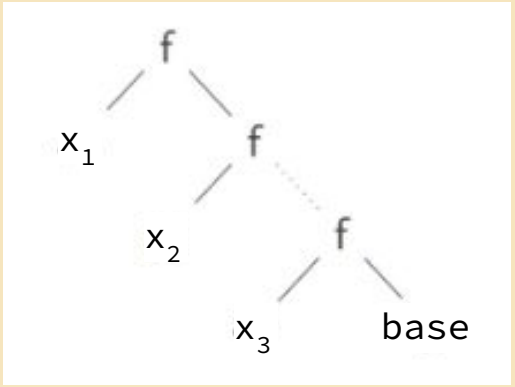
f(… f(f(init, x_1), x_2) …, x_n)



fold_right(f, x_1:: … ::x_n::[], base)

≡

f(x_1, f(x_2, … f(x_n, base) …))



A gotcha: Argument ordering

`fold_left(f, init, x1:: … :: xn::[])`

=

`f(… f(f(init, x1), x2) …, xn)`

`fold_right(f, x1:: … ::xn::[], base)`

=

`f(x1, f(x2, … f(xn, base) …))`

- Note that some languages (including Ocaml, and these slides) reverse the order of the base and function arguments to fold_left to reflect the order in which the accumulator argument is used: first for fold_left, last for fold_right

- Similarly, we reverse the order of the arguments to the function f

Does it matter which fold we use?

```
#Summing a list: [1, 2, 3, 4]#
```

```
fold_left(add, [1, 2, 3, 4])
= ?
```

```
fold_right(add, [1, 2, 3, 4])
= ?
```

Q: *Does it matter?*

Sometimes it doesn’t matter

```
# Summing a list: [1, 2, 3, 4]#
```

```
fold_left(add, [1, 2, 3, 4])
= (((1 + 2) + 3) + 4)
= 10
```

```
fold_right(add, [1, 2, 3, 4])
= 1 + (2 + (3 + 4))
= 10
```

But not always

```
# Subtracting a list: [1, 2, 3, 4]#
```

```
fold_left(sub, [1, 2, 3, 4])
= ?
```

```
fold_right(sub, [1, 2, 3, 4])
= ?
```

Q: *How about now? What's the difference?*

It matters for nonassociative functions

```
# Subtracting a list: [1, 2, 3, 4]#
```

```
fold_left(sub, [1, 2, 3, 4])
= (((1 - 2) - 3) - 4)
= -8
```

```
fold_right(sub, [1, 2, 3, 4])
= 1 - (2 - (3 - 4))
= -2
```

Sometimes it matters a lot

```
# Iteratively exponentiating a list: (non-associative!) #
```

```
fold_left(pow, [2, 3, 4])
  (2 ^ 3) ^ 4
  = 4096
```

```
fold_right(pow, [2, 3, 4])
= 2 ^ (3 ^ 4)
= 241785163922925834941235 2
```

$$\left(2^3\right)^4$$

$$2^{3^4}$$

So which should I actually use???

- Any* problem that can be solved with one can be solved with the other, possibly with some pre- or post-processing
- But in some cases it might be **much** easier to use one of them
- So if you get stuck trying one, try the other, or try to write a solution as a recursion directly and then try to abstract it

** There are some performance/stack/laziness considerations here, but we're not very worried about these in this course*

Some vague intuition

- I tend to use fold_left when I'm thinking about starting with some **initial** accumulator value, and then **iterating** over a list, considering in turn each element and the previous accumulator in order to build a new accumulator which will give me what I need to tackle the next element

- I tend to use fold_right when I'm thinking more traditionally recursively: When I have a non-empty list, I assume I've **recursively** solved the problem for the tail, and then figure out how to combine that solution with the head element to solve the problem for the whole list. When the list is empty, I provide an appropriate base case as the **final** value

Exercise

fold_left: exercise

- Let’s write a function called cut which, given a list of numbers, returns the shortest initial segment of the list containing 3 positive numbers. If there is no such initial segment, it returns the whole list.

fold_left: exercise

- Let’s write a function called cut which, given a list of numbers, returns the shortest initial segment of the list containing 3 positive numbers. If there is no such initial segment, it returns the whole list.

```
cut: ??
```

Q: *What’s the type of cut?*

fold_left: exercise

- Let's write a function called cut which, given a list of numbers, returns the shortest initial segment of the list containing 3 positive numbers. If there is no such initial segment, it returns the whole list.

```
cut: [Int] -> [Int]

cut([1, -4, 5, -2, 6, 7, 2])
= ??
```

Q: *What should cut return here?*

fold_left: exercise

- Let's write a function called cut which, given a list of numbers, returns the shortest initial segment of the list containing 3 positive numbers. If there is no such initial segment, it returns the whole list.

```
cut: [Int] -> [Int]

cut([1, -4, 5, -2, 6, 7, 2])
= [1, -4, 5, -2, 6]
```

fold_left: exercise

- Cut can be implemented as a single fold

- But since this is more complex than the examples we've seen so far, let's first just try to implement a function called count which counts the number of positive integers in a list

```
count: [Int] -> Int
```

fold_left: count positives

```
let helper?: type? =
  ?? in

#count number of positives#
let count: [Int] -> Int =
  fun xs -> fold_left(helper?, 0, xs) in
```

Q: *Given that we're folding over helper, what MUST be true about it's type?*

fold_left: count positives

```
let helper?: (acc_type?, element_type?) -> acc_type? =
  ?? in

#count number of positives#
let count: [Int] -> Int =
  fun xs -> fold_left(helper?, 0, xs) in
```

fold_left: count positives

```
let helper?: (Int, Int) -> Int =
  ?? in

#count number of positives#
let count: [Int] -> Int =
  fun xs -> fold_left(helper?, 0, xs) in
```

Q: *Now that we know the types of the parameters, what are some reasonable names?*

fold_left: count positives

```
let helper?: (Int, Int) -> Int =
  fun count_so_far, x ->
    ?? in

#count number of positives#
let count: [Int] -> Int =
  fun xs -> fold_left(helper?, 0, xs) in
```

Q: *Now what might we call this helper? What do we need it to do?*

fold_left: count positives

```
let maybe_incr: (Int, Int) -> Int =
  fun count_so_far, x ->
    ?? in

#count number of positives#
let count: [Int] -> Int =
  fun xs -> fold_left(maybe_incr, 0, xs) in
```

fold_left: count positives

```
let maybe_incr: (Int, Int) -> Int =
  fun count_so_far, x ->
    if x > 0 then count_so_far + 1 else count_so_far in

#count number of positives#
let count: [Int] -> Int =
  fun xs -> fold_left(maybe_incr, 0, xs) in
```

fold_left: back to cut

```
let maybe_incr: (Int, Int) -> Int =
  fun count_so_far, x ->
    if x > 0 then count_so_far + 1 else count_so_far in

let helper? = ?? in
let init? = ?? in

#shortest initial segment including 3 positives#
let cut: [Int] -> [Int] =
  fun xs ->
    let ?? = fold_left(helper?, init?, xs) in
    ?? in
```

- Gameplan: Like for count, we need to keep track of the number of positives we've seen so far, but we're also going to need to figure out a way to build up the return list as we go

fold_left: cut

```
let maybe_incr: (Int, Int) -> Int =
  fun count_so_far, x ->
    if x > 0 then count_so_far + 1 else count_so_far in

let helper?: (??, ??) -> ?? = ?? in
let init?: ?? = ?? in

#shortest initial segment including 3 positives#
let cut: [Int] -> [Int] =
  fun xs ->
    let ?? = fold_left(helper?, init?, xs) in
    ?? in
```

Q: *Which of these type holes are going to be the same? Which one can we immediately fill in?*

fold_left: cut

```
let maybe_incr: (Int, Int) -> Int =
  fun count_so_far, x ->
    if x > 0 then count_so_far + 1 else count_so_far in

let helper?: (acc_type?, Int) -> acc_type? = ?? in
let init?: acc_type? = ?? in

#shortest initial segment including 3 positives#
let cut: [Int] -> [Int] =
  fun xs ->
    let ?? = fold_left(helper?, init?, xs) in
    ?? in
```

Q: We need to return a list, but also keep track of a count: What kind of type might help?

fold_left: cut

```
let maybe_incr: (Int, Int) -> Int =
  fun count_so_far, x ->
    if x > 0 then count_so_far + 1 else count_so_far in

let helper?: ((??, [Int]), Int) -> (??, [Int]) = ?? in
let init?: (??, [Int]) = ?? in

#shortest initial segment including 3 positives#
let cut: [Int] -> [Int] =
  fun xs ->
    let ?? = fold_left(helper?, init?, xs) in
    ?? in
```

fold_left: cut

```
let maybe_incr: (Int, Int) -> Int =
  fun count_so_far, x ->
    if x > 0 then count_so_far + 1 else count_so_far in

let helper?: ((??, [Int]), Int) -> (??, [Int]) = ?? in
let init?: (??, [Int]) = ?? in

#shortest initial segment including 3 positives#
let cut: [Int] -> [Int] =
  fun xs ->
    let _, acc_list = fold_left(helper?, init?, xs) in
    acc_list in
```

fold_left: cut

```
let maybe_incr: (Int, Int) -> Int =
  fun count_so_far, x ->
    if x > 0 then count_so_far + 1 else count_so_far in

let helper?: ((Int, [Int]), Int) -> (Int, [Int]) = ?? in
let init?: (Int, [Int]) = ?? in

#shortest initial segment including 3 positives#
let cut: [Int] -> [Int] =
  fun xs ->
    let _, acc_list = fold_left(helper?, init?, xs) in
    acc_list in
```

Q: *What might we call the initial accumulator?*

fold_left: cut

```
let maybe_incr: (Int, Int) -> Int =
  fun count_so_far, x ->
    if x > 0 then count_so_far + 1 else count_so_far in

let helper?: ((Int, [Int]), Int) -> (Int, [Int]) = ?? in
let init_count_and_list: (Int, [Int]) = ?? in

#shortest initial segment including 3 positives#
let cut: [Int] -> [Int] =
  fun xs ->
    let _, acc_list = fold_left(helper?, init_count_and_list, xs) in
    acc_list in
```

Q: *And it's value?*

fold_left: cut

```
let maybe_incr: (Int, Int) -> Int =
  fun count_so_far, x ->
    if x > 0 then count_so_far + 1 else count_so_far in

let helper?: ((Int, [Int]), Int) -> (Int, [Int]) = ?? in
let init_count_and_list: (Int, [Int]) = (0, []) in

#shortest initial segment including 3 positives#
let cut: [Int] -> [Int] =
  fun xs ->
    let _, acc_list = fold_left(helper?, init_count_and_list, xs) in
    acc_list in
```

Q: *Given the helper's type, how should it begin?*

fold_left: cut

```
let maybe_incr: (Int, Int) -> Int =
  fun count_so_far, x ->
    if x > 0 then count_so_far + 1 else count_so_far in

let helper?: ((Int, [Int]), Int) -> (Int, [Int]) =
  fun (??, ??), x ->
    ?? in
let init_count_and_list: (Int, [Int]) = (0, []) in

#shortest initial segment including 3 positives#
let cut: [Int] -> [Int] =
  fun xs ->
    let _, acc_list = fold_left(helper?, init_count_and_list, xs) in
    acc_list in
```

fold_left: cut

```
let maybe_incr: (Int, Int) -> Int =
  fun count_so_far, x ->
    if x > 0 then count_so_far + 1 else count_so_far in

let helper?: ((Int, [Int]), Int) -> (Int, [Int]) =
  fun (count_so_far, cut_so_far), x ->
    ?? in
let init_count_and_list: (Int, [Int]) = (0, []) in

#shortest initial segment including 3 positives#
let cut: [Int] -> [Int] =
  fun xs ->
    let _, acc_list = fold_left(helper?, init_count_and_list, xs) in
    acc_list in
```

fold_left: cut

```
let maybe_incr: (Int, Int) -> Int =
  fun count_so_far, x ->
    if x > 0 then count_so_far + 1 else count_so_far in

let maybe_append: ((Int, [Int]), Int) -> (Int, [Int]) =
  fun (count_so_far, cut_so_far), x ->
    ?? in
let init_count_and_list: (Int, [Int]) = (0, []) in

#shortest initial segment including 3 positives#
let cut: [Int] -> [Int] =
  fun xs ->
    let _, acc_list = fold_left(maybe_append, init_count_and_list, xs) in
    acc_list in
```

fold_left: cut

```
let maybe_incr: (Int, Int) -> Int =
  fun count_so_far, x ->
    if x > 0 then count_so_far + 1 else count_so_far in

let maybe_append: ((Int, [Int]), Int) -> (Int, [Int]) =
  fun (count_so_far, cut_so_far), x ->
    (??, ??) in
let init_count_and_list: (Int, [Int]) = (0, []) in

#shortest initial segment including 3 positives#
let cut: [Int] -> [Int] =
  fun xs ->
    let _, acc_list = fold_left(maybe_append, init_count_and_list, xs) in
    acc_list in
```

fold_left: cut

```
let maybe_incr: (Int, Int) -> Int =
  fun count_so_far, x ->
    if x > 0 then count_so_far + 1 else count_so_far in

let maybe_append: ((Int, [Int]), Int) -> (Int, [Int]) =
  fun (count_so_far, cut_so_far), x ->
    let new_count: Int = ?? in
    let new_cut: [Int] = ?? in
    (new_count, new_cut) in
let init_count_and_list: (Int, [Int]) = (0, []) in

#shortest initial segment including 3 positives#
let cut: [Int] -> [Int] =
  fun xs ->
    let _, acc_list = fold_left(maybe_append, init_count_and_list, xs) in
    acc_list in
```

fold_left: cut

```
let maybe_incr: (Int, Int) -> Int =
  fun count_so_far, x ->
    if x > 0 then count_so_far + 1 else count_so_far in

let maybe_append: ((Int, [Int]), Int) -> (Int, [Int]) =
  fun (count_so_far, cut_so_far), x ->
    let new_count: Int = maybe_incr(count_so_far, x) in
    let new_cut: [Int] = ?? in
    (new_count, new_cut) in
let init_count_and_list: (Int, [Int]) = (0, []) in

#shortest initial segment including 3 positives#
let cut: [Int] -> [Int] =
  fun xs ->
    let _, acc_list = fold_left(maybe_append, init_count_and_list, xs) in
    acc_list in
```

fold_left: cut

```
let maybe_incr: (Int, Int) -> Int =
  fun count_so_far, x ->
    if x > 0 then count_so_far + 1 else count_so_far in

let maybe_append: ((Int, [Int]), Int) -> (Int, [Int]) =
  fun (count_so_far, cut_so_far), x ->
    let new_count: Int = maybe_incr(count_so_far, x) in
    let new_cut: [Int] =
      if ??
      then ??
      else ?? in
    (new_count, new_cut) in
let init_count_and_list: (Int, [Int]) = (0, []) in

#shortest initial segment including 3 positives#
let cut: [Int] -> [Int] =
  fun xs ->
    let _, acc_list = fold_left(maybe_append, init_count_and_list, xs) in
    acc_list in
```

fold_left: cut

```
let maybe_incr: (Int, Int) -> Int =
  fun count_so_far, x ->
    if x > 0 then count_so_far + 1 else count_so_far in

let maybe_append: ((Int, [Int]), Int) -> (Int, [Int]) =
  fun (count_so_far, cut_so_far), x ->
    let new_count: Int = maybe_incr(count_so_far, x) in
    let new_cut: [Int] =
      if count_so_far >= 3
      then ??
      else ?? in
    (new_count, new_cut) in
let init_count_and_list: (Int, [Int]) = (0, []) in

#shortest initial segment including 3 positives#
let cut: [Int] -> [Int] =
  fun xs ->
    let _, acc_list = fold_left(maybe_append, init_count_and_list, xs) in
    acc_list in
```

fold_left: cut

```
let maybe_incr: (Int, Int) -> Int =
  fun count_so_far, x ->
    if x > 0 then count_so_far + 1 else count_so_far in

let maybe_append: ((Int, [Int]), Int) -> (Int, [Int]) =
  fun (count_so_far, cut_so_far), x ->
    let new_count: Int = maybe_incr(count_so_far, x) in
    let new_cut: [Int] =
      if count_so_far >= 3
      then cut_so_far
      else ?? in
    (new_count, new_cut) in
let init_count_and_list: (Int, [Int]) = (0, []) in

#shortest initial segment including 3 positives#
let cut: [Int] -> [Int] =
  fun xs ->
    let _, acc_list = fold_left(maybe_append, init_count_and_list, xs) in
    acc_list in
```

fold_left: cut

```
let maybe_incr: (Int, Int) -> Int =
  fun count_so_far, x ->
    if x > 0 then count_so_far + 1 else count_so_far in

let maybe_append: ((Int, [Int]), Int) -> (Int, [Int]) =
  fun (count_so_far, cut_so_far), x ->
    let new_count: Int = maybe_incr(count_so_far, x) in
    let new_cut: [Int] =
      if count_so_far >= 3
      then cut_so_far
      else cut_so_far @ [x] in
    (new_count, new_cut) in
let init_count_and_list: (Int, [Int]) = (0, []) in

#shortest initial segment including 3 positives#
let cut: [Int] -> [Int] =
  fun xs ->
    let _, acc_list = fold_left(maybe_append, init_count_and_list, xs) in
    acc_list in
```

fold_left: What have we learned?

- We can use tuples as accumulators to track intermediary values, and use destructuring to take them apart at the end: the return type of our fold doesn't have to be the type of our final answer

- Similarly, we can deconstruct the accumulator in our fold helper, helping us make decisions based on what we've seen so far

- When in doubt, follow the types

Etc

fold: Advanced topics

- Exercise: Try to implement map using fold
- Exercise (Challenging): Try to implement fold_left using fold_right, or vice-versa
- Question: All else being equal, are their performance implications for choosing fold_left versus fold_right? What about stack use?
- Question: In lazy languages like Haskell, one of the folds *might* terminate even if it's called on an infinite list. Which one?