CSE 110A: Winter 2020

Fundamentals of Compiler Design I

Datatypes and Higher-order functions

Owen Arden
UC Santa Cruz

Based on course materials developed by Nadia Polikarpova

Representing complex data

- **We've seen:**
  - *base* types: `Bool`, `Int`, `Integer`, `Float`
  - some ways to *build up* types: given types `T1, T2`
    - functions: `T1 -> T2`
    - tuples: `(T1, T2)`
    - lists: `[T1]`
- **Algebraic Data Types:** a single, powerful technique for building up types to represent complex data
  - lets you define your own data types
  - subsumes tuples and lists!

Product types

- Tuples can do the job but there are two problems...

```
deadlineDate :: (Int, Int, Int)
deadlineDate = (2, 4, 2019)

deadlineTime :: (Int, Int, Int)
deadlineTime = (11, 59, 59)

-- | Deadline date extended by one day
extension :: (Int, Int, Int) -> (Int, Int, Int)
extension = ...
```

- Can you spot them?

1. Verbose and unreadable

```haskell
type Date = (Int, Int, Int)
type Time = (Int, Int, Int)

deadlineDate :: Date
deadlineDate = (2, 4, 2019)

deadlineTime :: Time
deadlineTime = (11, 59, 59)

-- | Deadline date extended by one day
extension :: Date -> Date
extension = ...
```

A **type synonym** for T: a name that can be used interchangeably with T

2. Unsafe

- We want this to fail at compile time!!!

```haskell
extension deadlineTime
```

- *Solution:* construct two different **datatypes**

```haskell
data Date = Date Int Int Int
data Time = Time Int Int Int
-- constructor^   ^parameter types

deadlineDate :: Date
deadlineDate = Date 2 4 2019

deadlineTime :: Time
deadlineTime = Time 11 59 59
```


Record Syntax

- Haskell's **record syntax** allows you to *name* the constructor parameters:
- Instead of

```haskell
data Date = Date Int Int Int
```

- You can write:

```haskell
data Date = Date {
  month :: Int,
  day   :: Int,
  year  :: Int
}
deadlineDate = Date 2 4 2019
deadlineMonth = month deadlineDate
```

Use the *field name* as a function to access part of the data

Building data types

- Three key ways to build complex types/values:
  1. **Product types (each-of)**: a value of `T` contains a value of `T1` *and* a value of `T2` **[done]**
  2. **Sum types (one-of)**: a value of `T` contains a value of `T1` *or* a value of `T2`
  3. **Recursive types**: a value of `T` contains a *sub-value* of the same type `Ts`

Example: NanoMD

- Suppose I want to represent a *text document* with simple markup. Each paragraph is either:
  - plain text (`String`)
  - heading: level and text (`Int` and `String`)
  - list: ordered? and items (`Bool` and `[String]`)
- I want to store all paragraphs in a *list*

```
doc = [ (1, "Notes from 130")              -- Lvl 1 heading
      , "There are two types of languages:" -- Plain text
      , (True, ["purely functional", "purely evil"])
                                            --^^ Ordered list
      ] -- But this doesn't type check!!!
```


Sum Types

- Solution: construct a new type for paragraphs that is a *sum* (*one-of*) the three options!
  - plain text (`String`)
  - heading: level and text (`Int` and `String`)
  - list: ordered? and items (`Bool` and `[String]`)
- I want to store all paragraphs in a *list*

```
data Paragraph =
    Text String          -- 3 constructors,
  | Heading Int String   -- each with different
  | List Bool [String]   -- parameters
```

QUIZ

What would GHCi say? *

```
data Paragraph =
    Text String | Heading Int String | List Bool [String]
```

What would GHCi say to

```
>:t Text "Hey there!"
```

- A. Syntax error
- B. Type error
- C. Paragraph
- D. [Paragraph]
- E. [String]

Constructing datatypes

```
data T =
    C1 T11 .. T1k
  | C2 T21 .. T2l
  | ..
  | Cn Tn1 .. Tnm
```

`T` is the new **datatype**

`C1 .. Cn` are the **constructors** of `T`

A **value** of type `T` is

- *either* `C1 v1 .. vk` with `vi :: T1i`
- *or* `C2 v1 .. vl` with `vi :: T2i`
- *or* ...
- *or* `Cn v1 .. vm` with `vi :: Tni`

Constructing datatypes

You can think of a `T` value as a **box**:

- *either* a box labeled `C1` with values of types `T11 .. T1k` inside
- *or* a box labeled `C2` with values of types `T21 .. T2l` inside
- *or* ...
- *or* a box labeled `Cn` with values of types `Tn1 .. Tnm` inside

Apply a constructor = pack some values into a box (and label it)

- `Text "Hey there!"`
  - put `"Hey there!"` in a box labeled `Text`
- `Heading 1 "Introduction"`
  - put `1` and `"Introduction"` in a box labeled `Heading`
- Boxes have different labels but same type (`Paragraph`)

Example: NanoMD

```
data Paragraph =
    Text String | Heading Int String | List Bool [String]
```

Now I can create a document like so:

```
doc :: [Paragraph]
doc = [
    Heading 1 "Notes from 130"
  , Text "There are two types of languages:"
  , List True ["purely functional", "purely evil"]
  ]
```


Example: NanoMD

Now I want **convert documents in to HTML.**

I need to write a function:

```
html :: Paragraph -> String
html p = ??? -- depends on the kind of paragraph!
```

How to tell what's in the box?

- Look at the label!

Pattern Matching

Pattern matching = looking at the label and extracting values from the box

- we've seen it before
- but now for arbitrary datatypes

```
html :: Paragraph -> String
html (Text str)        = ...
  -- It's a plain text! Get string
html (Heading lvl str) = ...
  -- It's a heading! Get level and string
html (List ord items)  = ...
  -- It's a list! Get ordered and items
```

Dangers of pattern matching (1)

```
html :: Paragraph -> String
html (Text str) = ...
html (List ord items) = ...
```

What would GHCi say to:

```
html (Heading 1 "Introduction")
```

Answer: Runtime error (no matching pattern)

Dangers of pattern matching (1)

Beware of **missing** and **overlapped** patterns

- GHC warns you about *overlapped* patterns
- GHC warns you about *missing* patterns when called with `-W` (use `:set -W` in GHCi)

Pattern matching expression

We've seen: pattern matching in *equations*

You can also pattern-match *inside your program* using the **case** expression:

```
html :: Paragraph -> String
html p =
  case p of
    Text str -> unlines [open "p", str, close "p"]
    Heading lvl str -> ...
    List ord items -> ...
```

QUIZ

What is the type of *

```
let p = Text "Hey there!"
in case p of
    Text str -> str
    Heading lvl _ -> lvl
    List ord _ -> ord
```

- A. Syntax error
- B. Type error
- C. String
- D. Paragraph
- E. Paragraph -> String

Pattern matching expression: typing

The `case` expression

```
case e of
  pattern1 -> e1
  pattern2 -> e2
  ...
  patternN -> eN
```

has type `T` if

- each `e1`...`eN` has type `T`
- `e` has some type `D`
- each `pattern1`...`patternN` is a *valid pattern* for `D`
  - i.e. a variable or a constructor of `D` applied to other patterns

The expression `e` is called the *match scrutinee*

Building data types

- Three key ways to build complex types/values:
  1. **Product types** (**each-of**): a value of `T` contains a value of `T1` *and* a value of `T2` **[done]**
  2. **Sum types** (**one-of**): a value of `T` contains a value of `T1` *or* a value of `T2` **[done]**
  3. **Recursive types**: a value of `T` contains a *sub-value* of the same type `Ts`

Recursive types

Let's define **natural numbers** from scratch:

```
data Nat = ???
```

22

Recursive types

```
data Nat = Zero | Succ Nat
```

A `Nat` value is:

- either an *empty* box labeled `Zero`
- or a box labeled `Succ` with another `Nat` in it!

Some `Nat` values:

```
Zero                      -- 0
Succ Zero                 -- 1
Succ (Succ Zero)          -- 2
Succ (Succ (Succ Zero))   -- 3
...
```

23

Functions on recursive types

Principle: Recursive code mirrors recursive data

24

1. Recursive type as a parameter

```
data Nat = Zero     -- base constructor
         | Succ Nat -- inductive constructor
```

Step 1: add a pattern per constructor

```
toInt :: Nat -> Int
toInt Zero     = ... -- base case
toInt (Succ n) = ... -- inductive case
                     -- (recursive call goes here)
```


1. Recursive type as a parameter

```
data Nat = Zero     -- base constructor
         | Succ Nat -- inductive constructor
```

Step 2: fill in base case

```
toInt :: Nat -> Int
toInt Zero     = 0   -- base case
toInt (Succ n) = ... -- inductive case
                     -- (recursive call goes here)
```


1. Recursive type as a parameter

```
data Nat = Zero     -- base constructor
         | Succ Nat -- inductive constructor
```

Step 3: fill in inductive case using a recursive call:

```
toInt :: Nat -> Int
toInt Zero     = 0           -- base case
toInt (Succ n) = 1 + toInt n -- inductive case
```

QUIZ

What does this evaluate to? *

```
let foo i = if i <= 0 then Zero else Succ (foo (i - 1))
in foo 2
```

- ◯ A. Syntax error
- ◯ B. Type error
- ◯ C. 2
- ◯ D. Succ Zero
- ◯ E. Succ (Succ Zero)

2. Recursive type as a result

```
data Nat = Zero     -- base constructor
         | Succ Nat -- inductive constructor

fromInt :: Int -> Nat
fromInt n
  | n <= 0    = Zero                  -- base case
  | otherwise = Succ (fromInt (n - 1)) -- inductive
                                       -- case
```


2. Putting the two together

```
data Nat = Zero     -- base constructor
         | Succ Nat -- inductive constructor

add :: Nat -> Nat -> Nat
add Zero     m = m                -- base case
add (Succ n) m = Succ (add n m)   -- inductive case

sub :: Nat -> Nat -> Nat
sub n        Zero     = n         -- base case 1
sub Zero     _        = Zero      -- base case 2
sub (Succ n) (Succ m) = sub n m   -- inductive case
```

2. Putting the two together

```
data Nat = Zero          -- base constructor

add
add
add

sub
sub
sub Zero     _        = Zero    -- base case 2
sub (Succ n) (Succ m) = sub n m -- inductive case
```

Lessons learned:

- **Recursive code mirrors recursive data**
- With **multiple** arguments of a recursive type, which one should I recurse on?
- The name of the game is to pick the right **inductive strategy**!

Lists

Lists aren't built-in! They are an *algebraic data type* like any other:

```
data List = Nil           -- base constructor
          | Cons Int List -- inductive constructor
```

- List [1, 2, 3] is *represented* as Cons 1 (Cons 2 (Cons 3 Nil))
- Built-in list constructors [] and (:) are just fancy syntax for Nil and Cons

Functions on lists follow the same general strategy:

```
length :: List -> Int
length Nil         = 0              -- base case
length (Cons _ xs) = 1 + length xs  -- inductive case
```


Lists

What is the right *inductive strategy* for appending two lists?

```
append :: List -> List -> List
append ??? ??? = ???
```

Lists

What is the right *inductive strategy* for appending two lists?

```
append :: List -> List -> List
append Nil ys = ys
append ??? ??? = ???
```


Lists

What is the right *inductive strategy* for appending two lists?

```
append :: List -> List -> List
append Nil ys = ys
append (Cons x xs) ys = Cons x (append xs ys)
```


Recursion is…

Building solutions for *big problems* from solutions for *sub-problems*

- **Base case:** what is the *simplest version* of this problem and how do I solve it?
- **Inductive strategy:** how do I *break down* this problem into sub-problems?
- **Inductive case:** how do I solve the problem *given* the solutions for subproblems?

- But it can get kinda repetitive!

Example: evens

Let's write a function evens:

```
-- evens []        ==> []
-- evens [1,2,3,4] ==> [2,4]
evens       :: [Int] -> [Int]
evens []     = ...
evens (x:xs) = ...
```


Example: four-letter words

Let's write a function `fourChars`:

```
-- fourChars [] ==> []
-- fourChars ["i","must","do","work"] ==> ["must","work"]
fourChars :: [String] -> [String]
fourChars []     = ...
fourChars (x:xs) = ...
```


Yikes, Most Code is the Same!

```
foor []             = []
foo (x:xs)
  | x mod 2 == 0  = x : foo xs
  | otherwise     =     foo xs

foo []            = []
foo (x:xs)
  | length x == 4 = x : foo xs
  | otherwise     =     foo xs
```

Only difference is **condition**

- `x mod 2 == 0` vs `length x == 4`

Moral of the day

D.R.Y. Don’t Repeat Yourself!

Can we

- *reuse* the general pattern and
- *substitute in* the custom condition?

HOFs to the rescue!

General **Pattern**

- expressed as a *higher-order function*
- takes customizable operations as *arguments*

Specific **Operation**

- passed in as an argument to the HOF

The “filter” pattern

```
evens []            = []
evens (x:xs)
  | x `mod` 2 == 0 = x : evens xs
  | otherwise      =     evens xs
```

```
fourChars []       = []
fourChars (x:xs)
  | length x == 4 = x : fourChars xs
  | otherwise     =     fourChars xs
```

```
filter f []       = []
filter f (x:xs)
  | f x           = x : filter f xs
  | otherwise     =     filter f xs
```

Use the `filter` pattern to avoid duplicating code!

The "filter" pattern

General **Pattern**

- HOF `filter`
- Recursively traverse list and pick out elements that satisfy a predicate

Specific **Operation**

- Predicates `isEven` and `isFour`

```
filter f []       = []
filter f (x:xs)
  | f x           = x : filter f xs
  | otherwise     =     filter f xs
```

```
evens         = filter isEven
  where
    isEven x = x `mod` 2 == 0
```

```
fourChars     = filter isFour
  where
    isFour x = length x == 4
```


Let's talk about types

```
-- evens [1,2,3,4] ==> [2,4]
evens :: [Int] -> [Int]
evens xs = filter isEven xs
  where
    isEven :: Int -> Bool
    isEven x  =  x `mod` 2 == 0
filter :: ???
```


Let's talk about types

```
-- evens [1,2,3,4] ==> [2,4]
evens :: [Int] -> [Int]
evens xs = filter isEven xs
  where
    isEven :: Int -> Bool
    isEven x  =  x `mod` 2 == 0
filter :: ???
```

Let's talk about types

```
-- fourChars ["i","must","do","work"] ==> ["must","work"]
fourChars :: [String] -> [String]
fourChars xs = filter isFour xs
  where
    isFour :: String -> Bool
    isFour x  =  length x == 4
filter :: ???
```


Let's talk about types

Uh oh! So what's the type of `filter`?

```
filter :: (Int -> Bool) -> [Int] -> [Int] -- ???
```

```
filter :: (String -> Bool) -> [String] -> [String] -- ???
```

- It *does not care* what the list elements are
  - as long as the predicate can handle them
- It's type is **polymorphic** (generic) in the type of list elements

```
-- For any type `a`
--   if you give me a predicate on `a`s
--   and a list of `a`s,
--   I'll give you back a list of `a`s
filter :: (a -> Bool) -> [a] -> [a]
```


Example: all caps

Lets write a function `shout`:

```
-- shout []                    ==> []
-- shout ['h','e','l','l','o'] ==> ['H','E','L','L','O']
shout :: [Char] -> [Char]
shout []     = ...
shout (x:xs) = ...
```

Example: squares

Lets write a function `squares`:

```
-- squares []        ==> []
-- squares [1,2,3,4] ==> [1,4,9,16]
squares :: [Int] -> [Int]
squares []     = ...
squares (x:xs) = ...
```


Yikes, Most Code is the Same!

Lets rename the functions to `foo`:

```
-- shout
foo []     = []
foo (x:xs) = toUpper x : foo xs

-- squares
foo []     = []
foo (x:xs) = (x * x)   : foo xs
```

Lets **refactor** into the **common pattern**

```
pattern = ...
```


The "map" pattern

```
shout []     = []
shout (x:xs) = toUpper x : shout xs
```

```
squares []     = []
squares (x:xs) = (x*x) : squares xs
```

```
map f []     = []
map f (x:xs) = f x : map f xs
```

The `map` Pattern

General Pattern

- HOF `map`
- Apply a transformation `f` to each element of a list

Specific Operations

- Transformations `toUpper` and `\x -> x * x`

The "map" pattern

```
map f []     = []
map f (x:xs) = f x : map f xs
```

Lets refactor `shout` and `squares`

```
shout   = map ...

squares = map ...
```

```
map f []     = []
map f (x:xs) = f x : map f xs
```

```
shout = map (\x -> toUpper x)
```

```
squares = map (\x -> x*x)
```


QUIZ

What is the type of map? *

```
map f []     = []
map f (x:xs) = f x : map f xs
```

(A) `(Char -> Char) -> [Char] -> [Char]`

(B) `(Int -> Int) -> [Int] -> [Int]`

(C) `(a -> a) -> [a] -> [a]`

(D) `(a -> b) -> [a] -> [b]`

(E) `(a -> b) -> [c] -> [d]`

The "map" pattern

```
-- For any types `a` and `b`
--   if you give me a transformation from `a` to `b`
--   and a list of `a`s,
--   I'll give you back a list of `b`s
map :: (a -> b) -> [a] -> [b]
```

Type says it all!

- The only meaningful thing a function of this type can do is apply its first argument to elements of the list (Hoogle it!)

Things to try at home:

- can you write a function `map' :: (a -> b) -> [a] -> [b]` whose behavior is different from `map`?
- can you write a function `map' :: (a -> b) -> [a] -> [b]` such that `map' f xs` returns a list whose elements are not in `map f xs`?

Don't Repeat Yourself

Benefits of **factoring** code with HOFs:

- Reuse iteration pattern
  - think in terms of standard patterns
  - less to write
  - easier to communicate
- Avoid bugs due to repetition

Recall: length of a list

```
-- len []      ==> 0
-- len ["carne","asada"] ==> 2
len :: [a] -> Int
len []     = 0
len (x:xs) = 1 + len xs
```


Recall: summing a list

```
-- sum []      ==> 0
-- sum [1,2,3] ==> 6
sum :: [Int] -> Int
sum []     = 0
sum (x:xs) = x + sum xs
```

Example: string concatenation

Let's write a function `cat`:

```
-- cat [] ==> ""
-- cat ["carne","asada","torta"] ==> "carneasadatorta"
cat :: [String] -> String
cat []     = ...
cat (x:xs) = ...
```


Can you spot the pattern?

```
-- len
foo []     = 0
foo (x:xs) = 1 + foo xs

-- sum
foo []     = 0
foo (x:xs) = x + foo xs

-- cat
foo []     = ""
foo (x:xs) = x ++ foo xs

pattern = ...
```


The "fold-right" pattern

```
len []     = 0
len (x:xs) = 1 + len xs
```

```
sum []     = 0
sum (x:xs) = x + sum xs
```

```
cat []     = “”
cat (x:xs) = x ++ sum xs
```

```
foldr f b []     = b
foldr f b (x:xs) = f x (foldr f b xs)
```

The `foldr` Pattern

General Pattern

- Recurse on tail
- Combine result with the head using some binary operation

The "fold-right" pattern

```
foldr f b []     = b
foldr f b (x:xs) = f x (foldr f b xs)
```

Let's refactor `sum`, `len` and `cat`:

```
sum = foldr ...  ...

cat = foldr ...  ...

len = foldr ...  ...
```

Factor the recursion out!

The "fold-right" pattern

```
foldr f b []     = b
foldr f b (x:xs) = f x (foldr f b xs)
```

```
len = foldr (\x n -> 1 + n) 0
```

```
sum = foldr (\x n -> x + n) 0
```

```
cat = foldr (\x s -> x ++ n) ""
```

You can write it more clearly as

```
sum = foldr (+) 0
cat = foldr (++) ""
```


The "fold-right" pattern

```
foldr f b []     = b
foldr f b (x:xs) = f x (foldr f b xs)
```

```
len = foldr (\x n -> 1 + n) 0
```

```
sum = foldr (\x n -> x + n) 0
```

```
cat = foldr (\x s -> x ++ n) ""
```

You can write it more clearly as

```
sum = foldr (+) 0
cat = foldr (++) ""
```

QUIZ

What does this evaluate to? *

```
foldr f b []     = b
foldr f b (x:xs) = f x (foldr f b xs)

quiz = foldr (:) [] [1,2,3]
```

- (A) Type error
- (B) [1,2,3]
- (C) [3,2,1]
- (D) [[3],[2],[1]]
- (E) [[1],[2],[3]]

The "fold-right" pattern

```
foldr f b []     = b
foldr f b (x:xs) = f x (foldr f b xs)

foldr (:) [] [1,2,3]
  ==> (:) 1 (foldr (:) [] [2, 3])
  ==> (:) 1 ((:) 2 (foldr (:) [] [3]))
  ==> (:) 1 ((:) 2 ((:) 3 (foldr (:) [] [])))
  ==> (:) 1 ((:) 2 ((:) 3 []))
  ==  1 : (2 : (3 : []))
  ==  [1,2,3]
```


The "fold-right" pattern

```
foldr f b [x1, x2, x3, x4]
  ==> f x1 (foldr f b [x2, x3, x4])
  ==> f x1 (f x2 (foldr f b [x3, x4]))
  ==> f x1 (f x2 (f x3 (foldr f b [x4])))
  ==> f x1 (f x2 (f x3 (f x4 (foldr f b []))))
  ==> f x1 (f x2 (f x3 (f x4 b)))
```

Accumulate the values from the **right**

For example:

```
foldr (+) 0 [1, 2, 3, 4]
  ==> 1 + (foldr (+) 1 [2, 3, 4])
  ==> 1 + (2 + (foldr (+) 0 [3, 4]))
  ==> 1 + (2 + (3 + (foldr (+) 0 [4])))
  ==> 1 + (2 + (3 + (4 + (foldr (+) 0 []))))
  ==> 1 + (2 + (3 + (4 + 0)))
```

Tail recursion

Recursive call is the *top-most* sub-expression in the function body

- i.e. no computations allowed on recursively returned value
- i.e. value returned by the recursive call == value returned by function

The “fold-right” pattern

Is `foldr` **tail recursive?**

Answer: No! It calls the binary operations on the results of the recursive call

What about tail-recursive versions?

Let’s write tail-recursive `sum`!

```
sumTR :: [Int] -> Int
sumTR = ...
```

What about tail-recursive versions?

Let's write tail-recursive `sum`!

```
sumTR :: [Int] -> Int
sumTR xs = helper 0 xs
  where
    helper acc []     = acc
    helper acc (x:xs) = helper (acc + x) xs
```


What about tail-recursive versions?

Lets run `sumTR` to see how it works

```
sumTR [1,2,3]
  ==> helper 0 [1,2,3]
  ==> helper 1   [2,3]    -- 0 + 1 ==> 1
  ==> helper 3     [3]    -- 1 + 2 ==> 3
  ==> helper 6      []    -- 3 + 3 ==> 6
  ==> 6
```

Note: `helper` directly returns the result of recursive call!

What about tail-recursive versions?

Let's write tail-recursive `cat`!

```
catTR :: [String] -> String
catTR = ...
```

What about tail-recursive versions?

Let's write tail-recursive `cat`!

```
catTR :: [String] -> String
catTR xs = helper "" xs
  where
    helper acc []     = acc
    helper acc (x:xs) = helper (acc ++ x) xs
```


What about tail-recursive versions?

Lets run `catTR` to see how it works

```
catTR                 ["carne", "asada", "torta"]
  ==> helper ""       ["carne", "asada", "torta"]
  ==> helper "carne"           ["asada", "torta"]
  ==> helper "carneasada"               ["torta"]
  ==> helper "carneasadatorta"                 []
  ==> "carneasadatorta"
```

Note: `helper` directly returns the result of recursive call!

Can you spot the pattern?

```
-- sumTR
foo xs                = helper 0 xs
  where
    helper acc []     = acc
    helper acc (x:xs) = helper (acc + x) xs


-- catTR
foo xs                = helper "" xs
  where
    helper acc []     = acc
    helper acc (x:xs) = helper (acc ++ x) xs

pattern = ...
```

The "fold-left" pattern

```
sum xs                = helper 0 xs
  where
    helper acc []     = acc
    helper acc (x:xs) = helper (acc + x) xs
```

```
cat xs                = helper "" xs
  where
    helper acc []     = acc
    helper acc (x:xs) = helper (acc ++ x) xs
```

```
foldl f b xs          = helper b xs
  where
    helper acc []     = acc
    helper acc (x:xs) = helper (f acc x) xs
```

The `foldl` Pattern

General Pattern

- Use a helper function with an extra accumulator argument
- To compute new accumulator, combine current accumulator with the head using some binary operation

The "fold-left" pattern

```
foldl f b xs          = helper b xs
  where
    helper acc []     = acc
    helper acc (x:xs) = helper (f acc x) xs
```

Let's refactor `sumTR` and `catTR`:

```
sumTR = foldl ...  ...

catTR = foldl ...  ...
```

Factor the tail-recursion out!

QUIZ

What does this evaluate to? *

```
foldl f b xs          = helper b xs
  where
    helper acc []     = acc
    helper acc (x:xs) = helper (f acc x) xs

quiz = foldl (\xs x -> x : xs) [] [1,2,3]
```

- (A) Type error
- (B) [1,2,3]
- (C) [3,2,1]
- (D) [[3],[2],[1]]
- (E) [[1],[2],[3]]

The “fold-left” pattern

```
foldl f b                     [x1, x2, x3, x4]
  ==> helper b                [x1, x2, x3, x4]
  ==> helper (f b x1)             [x2, x3, x4]
  ==> helper (f (f b x1) x2)          [x3, x4]
  ==> helper (f (f (f b x1) x2) x3)       [x4]
  ==> helper (f (f (f (f b x1) x2) x3) x4)  []
  ==> (f (f (f (f b x1) x2) x3) x4)
```

Accumulate the values from the **left**

For example:

```
foldl (+) 0                   [1, 2, 3, 4]
  ==> helper 0                [1, 2, 3, 4]
  ==> helper (0 + 1)             [2, 3, 4]
  ==> helper ((0 + 1) + 2)          [3, 4]
  ==> helper (((0 + 1) + 2) + 3)       [4]
  ==> helper ((((0 + 1) + 2) + 3) + 4)  []
  ==> ((((0 + 1) + 2) + 3) + 4)
```


Left vs. Right

```
foldl f b [x1, x2, x3]  ==> f (f (f b x1) x2) x3 -- Left

foldr f b [x1, x2, x3]  ==> f x1 (f x2 (f x3 b)) -- Right
```

For example:

```
foldl (+) 0 [1, 2, 3]  ==> ((0 + 1) + 2) + 3  -- Left

foldr (+) 0 [1, 2, 3]  ==> 1 + (2 + (3 + 0))  -- Right
```

Different types!

```
foldl :: (b -> a -> b) -> b -> [a] -> b  -- Left

foldr :: (a -> b -> b) -> b -> [a] -> b  -- Right
```


Useful HOF: flip

```
-- you can write
foldl (\xs x -> x : xs) [] [1,2,3]

-- more concisely like so:
foldl (flip (:))        [] [1,2,3]
```

What is the type of `flip`?

```
flip :: (a -> b -> c) -> b -> a -> c
```

Useful HOF: compose

```
-- you can write
map (\x -> f (g x)) ys

-- more concisely like so:
map (f . g) ys
```

What is the type of `(.)`?

```
(.) :: (b -> c) -> (a -> b) -> a -> c
```


Higher Order Functions

Iteration patterns over collections:

- **Filter** values in a collection given a *predicate*
- **Map** (iterate) a given *transformation* over a collection
- **Fold** (reduce) a collection into a value, given a *binary operation* to combine results

Useful helper HOFs:

- **Flip** the order of function's (first two) arguments
- **Compose** two functions

Higher Order Functions

HOFs can be put into libraries to enable modularity

- Data structure **library** implements `map`, `filter`, `fold` for its collections
  - generic efficient implementation
  - generic optimizations: `map f (map g xs) --> map (f.g) xs`
- Data structure **clients** use HOFs with specific operations
  - no need to know the implementation of the collection

Enabled the "big data" revolution e.g. *MapReduce*, *Spark*

That’s all folks!