csci54 – discrete math & functional programming
lambdas and folds

map and filter (from last time)

- map :: (a -> b) -> [a] -> [b]
  - takes a function that maps elements of type a to type b
  - applies the function to every element in a list of type a and returns a list of the results (which have type b)

```
ghci> map length ["ab", "aaaaa", "b"]
ghci> map (^3) [1,3,6]
```

- filter :: (a -> Bool) -> [a] -> [a]
  - takes a function that maps elements of type a to True/False (a predicate)
  - applies the function to every element in a list of type a and returns only those elements for which the function returns True

```
ghci> headA x = (head x) == 'a'
ghci> filter headA ["ab", "aaaaa", "b"]
```

Curried functions

- Every function in Haskell only takes one parameter (!!)
- What does that mean?

```
ghci> mult x y z = x * y * z
```

```
ghci> mult x y z = x * y * z
ghci> let mult10 = mult 2 5 in map mult10 [1,2,3]
```

map and filter

- `map :: (a -> b) -> [a] -> [b]`
  - takes a function that maps elements of type a to type b
  - applies the function to every element in a list of type a and returns a list of the results (which have type b)

- `filter :: (a -> Bool) -> [a] -> [a]`
  - takes a function that maps elements of type a to True/False (a predicate)
  - applies the function to every element in a list of type a and returns only those elements for which the function returns True

- how would you implement `map`? `filter`?

practice problem

- The `mapish` function takes a list of functions and a single element x. It then returns a list of the results of applying each function to x. Implement the `mapish` function.

```
ghci> mapish [(+1), (*3)] 10
[11, 30]
```

- what is the type of the `mapish` function?

What if you wanted to mapish:

$f1(x) = x^2 +1$

$f2(x) = 4x-10$

lambdas (aka anonymous functions)

- functions that don't have names
- functions that you use once in the context of some other function

```
ghci> headA x = (head x) == 'a'
ghci> filter headA ["ab", "aaaaa", "b"]
```

```
ghci> filter (\y -> (head y) == 'a') ["ab", "aaaaa", "b"]
```

- syntax:

```
\a b -> (a * b + 10)
```

  - starts with \ (meant to resemble λ).
  - -> separates parameters from what the function evaluates to

lambdas (aka anonymous functions)

- note that if we wanted a function `headA` such that it would take out the elements that started with the character 'A', we could define it as follows:

```
ghci> headA = filter (\y -> (head y) == 'A')
```

- practice: what is the type of the function `foo`? what does it do?

```
foo y zs = map (\x -> x^y) zs
```

One more built-in higher order function

- map, filter, reduce

- How would you write a function `sumList` that returned the sum of a list of integers? `prodList` the returned the product of a list of integers?

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

```
prodList [] = 1
prodList (x:xs) = x * (prodList xs)
```

  - what is similar?
  - what is different?

- in Haskell "reduce" is referred to as "fold"

```
foldr' :: (b -> b -> b) -> b -> [b] -> b
```

Right fold (foldr)

```
foldr' :: (b -> b -> b) -> b -> [b] -> b
```

- `foldr (+) 0 [3,2,6]`
  - very, very informally can think:
    - [3,2,6] is really 3:2:6:[].
    - Replace [] with the base case 0 (sometimes called “seed” value)
    - Replace : with the operator (+)
  - associate to the right
  - 3 + (2 + (6 + 0))
- how would you write `sumList` and `prodList` using foldr?

foldr and foldl

```
foldr' :: (b -> b -> b) -> b -> [b] -> b
```

- `foldr (+) 0 [3,2,6]`
  - informally can think of as: [3,2,6] is really 3:2:6:[]. Replace [] with the base case and the : with the operator
  - associate to the right
  - 3 + (2 + (6 + 0))
- `foldl` - same idea but associates to the left
  - So the seed value also goes in at the leftmost position

foldr and foldl

```
foldr' :: (a -> a -> a) -> a -> [a] -> a
```

- foldr f x [y1, y2, ... yk] = f y1 (f y2 (... (f yk x) ...))

```
foldl' :: (a -> a -> a) -> a -> [a] -> a
```

- foldl f x [y1, y2, ... yk] = f (... (f (f x y1) y2) ...) yk

- foldr (+) 0 [3,2,6]
- foldl (+) 0 [3,2,6]

practice with folds

```
foldr f x [y1, y2, ... yk] = f y1 (f y2 (... (f yk x) ... ))

foldl f x [y1, y2, ... yk] =  f (... (f (f x y1) y2) ...) yk
```

- The following evaluate to two different values:
  - `foldr (^) 1 [2,3]`
  - `foldl (^) 1 [2,3]`

- What do they evaluate to and why?

and a hint of something more . . .

- `foldr f x [y1, y2, ... yk] = f y1 (f y2 (... (f yk x) ... ))`

- what does the following do?

```
foldr (\_ s -> 1 + s) 0 "abcde"
```

- what does this tell you about the type signature?

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

- (but really it's this:

```
foldr :: Foldable t => (a -> b -> b) -> b -> t a -> b
```

)