csci54 – discrete math & functional programming
lambdas and folds

practice problem from last time

- 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.

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

Implement the `mapish` function. What is the type of the `mapish` function?

```
mapish :: [a->b] -> a -> [b]
mapish [] _ = []
mapish (f:fs) x = (f x) : (mapish fs x)

mapish' :: [(a->b)] -> a -> [b]
mapish' fs x = [f x | f <- fs]

mapish’’ fs x = map (\f → f x) fs
```

use `mapish` to implement a function `f` that takes a number x and computes:

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

$f2(x) = 4x - 10$

Higher order functions

- Let’s get practice with a few higher-order functions:

- dup :: a → (a → a → b) → b
- compose :: (a → b) → (b → c) → (a → c)
- rot :: (a → b → c → d) → (b → c → a → d)
  - Same as: (a → b → c → d) → b → c → a → d

Implement these functions. You may (but don’t have to) use lambdas.

Currying

- Remember that in partial application, we always eliminate the *outermost* (typically leftmost) arrow.
- dup :: a → (a → a → b) → b
  - i.e. (a → ((a → a → b) → b))
  - dup 7 :: (Num a) => (a → a → b) → b
- compose :: (a → b) → (b → c) → (a → c)
  - compose double isEven :: ????
- rot :: (a → b → c → d) → (b → c → a → d)
  - rot foldl :: … we'll get to this later

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
```

)

Currying practice

- foldr’: (a → b → b) → b → [a] → b
  - foldr’ (+) :: ...
- rot :: (a → b → c → d) → (b → c → a → d)
  - rot foldr’ :: …