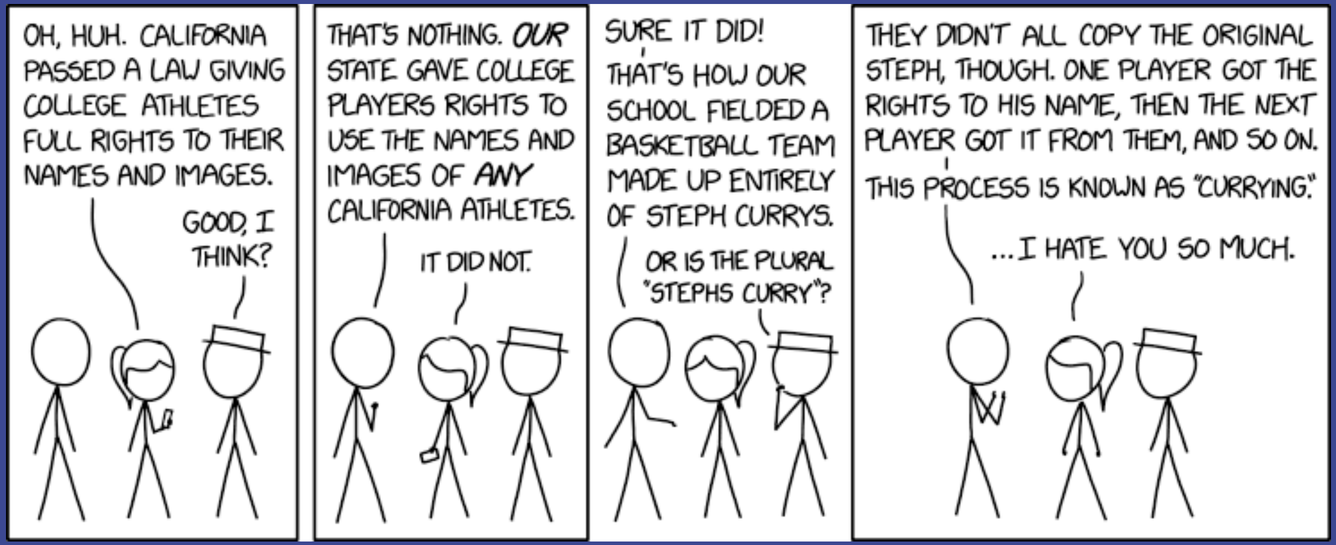
CS 152

Programming Paradigms

More Haskell



https://xkcd.com/2210/

Today

- User Defined Types
- Higher-Order Functions: zipping and folding
- Lazy Evaluation
- Currying
- Maybe?

User Defined Types

Syntax:
data TypeName = Constructor … deriving

Example:
data SJSUColor = Blue | Yellow deriving (Show, Eq)
> color = Blue
> :t color
color :: SJSUColor
> color
Blue
> anotherColor = Yellow
> color == anotherColor
False

User Defined Types

```
data Shape = Circle Float
           | Rectangle Float Float
           deriving (Show, Eq)

> box = Rectangle 5 8
> plate = Circle 2
> box
Rectangle 5.0 8.0
> box == plate
False
```

This is known as an algebraic sum data type.
Each variant has its own constructor, which takes a specified number of arguments with specified types.

User Defined Types and Pattern Matching

```
data Shape = Circle Float
           | Rectangle Float Float
           deriving (Show, Eq)
```

```
area :: Shape -> Float
area (Circle r) = pi * r ** 2
area (Rectangle width length) = width * length

> box = Rectangle 5 8
> area box
40.0
> plate = Circle 2
> area plate
12.566371
```

Record Data Types

```
data Car = Car {make :: String, model :: String, year :: Int}
    deriving Show
```

This is known as an algebraic product data type.

```
>myCar = Car{make="Honda", year=2010, model="Civic"}
>myCar
Car {make = "Honda", model = "Civic", year = 2010}
```

Recursive Data Types

```
data Tree = EmptyTree
          | Node Int Tree Tree
    deriving Show
> babytree = Node 5 EmptyTree EmptyTree
> :t babytree
babytree :: Tree
> biggertree = Node 10 EmptyTree babytree
> biggertree
Node 10 EmptyTree (Node 5 EmptyTree EmptyTree)
```

User defined Types with Type Parameters

```
data Tree = EmptyTree
            | Node Int Tree Tree
      deriving Show

data Tree = EmptyTree
            | Node String Tree Tree
      deriving Show
```

This is a tree where the underlying type is Int. What if we wanted a tree of strings?

User defined Types with Type Parameters

```
data Tree a = EmptyTree
        | Node a Tree Tree
        deriving Show

> babytree = Node "Hello" EmptyTree EmptyTree
> :t babytree
babytree :: Tree [Char]
```

Type Synonyms

```
type Grades = [Int]
average:: Grades -> Int
average [] = 0
average xs = sum xs `div` length xs

>:t average
average :: Grades -> Int
> average [100, 90, 80]
90
```

Type Summary

- We define data types with data
- Sum data types: Alternative1 | Alternative2 |…
- Product data types (record type)
- Important to add: deriving…
- Type parameters for more generic data types (Tree a)
- We define type synonym with type

Higher Order Functions in Haskell

- ✓ map
- ✓ filter
- zip
- zipWith
- foldl and foldr

zip

zip: takes two lists and combines them into a list of tuples

zip:: [a] -> [b] -> [(a,b)]

> zip [1, 2, 3, 4] ['A'..'Z']

[(1,'A'),(2,'B'),(3,'C'),(4,'D')]

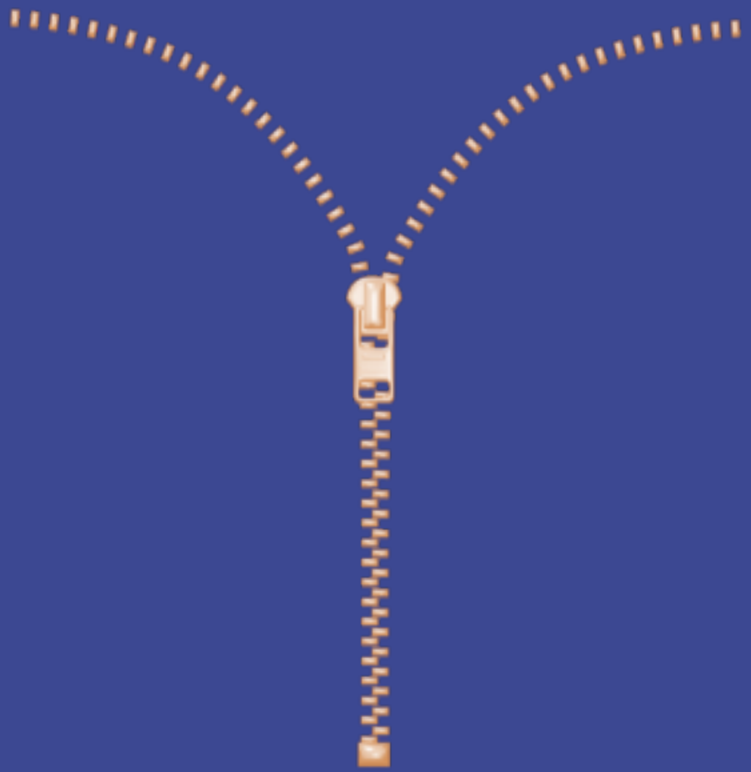
zipWith

zipWith: takes a function and two lists and applies the functions on the corresponding elements of the two lists

zipWith:: (a -> b -> c) -> [a] -> [b] -> [c]

```
>zipWith (+) [1..10] [10, 11, 12]
[11,13,15]
>zipWith (\x y -> x*x + y*y) [1, 2] [4..200]
```

iClicker: What is the length of the resulting list?



zipWith

zipWith: takes a function and two lists and applies the functions on the corresponding elements of the two lists

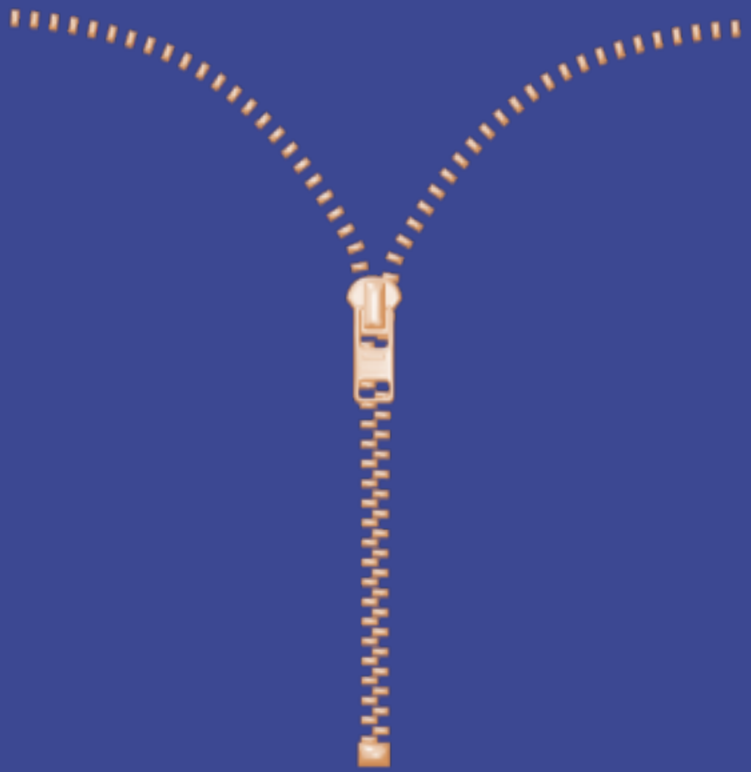
zipWith:: (a -> b -> c) -> [a] -> [b] -> [c]

>zipWith (+) [1..10] [10, 11, 12]

[11,13,15]

>zipWith (\x y -> x*x + y*y) [1, 2] [4..200]

iClicker: What is the resulting list?



zipWith

zipWith: takes a function and two lists and applies the functions on the corresponding elements of the two lists

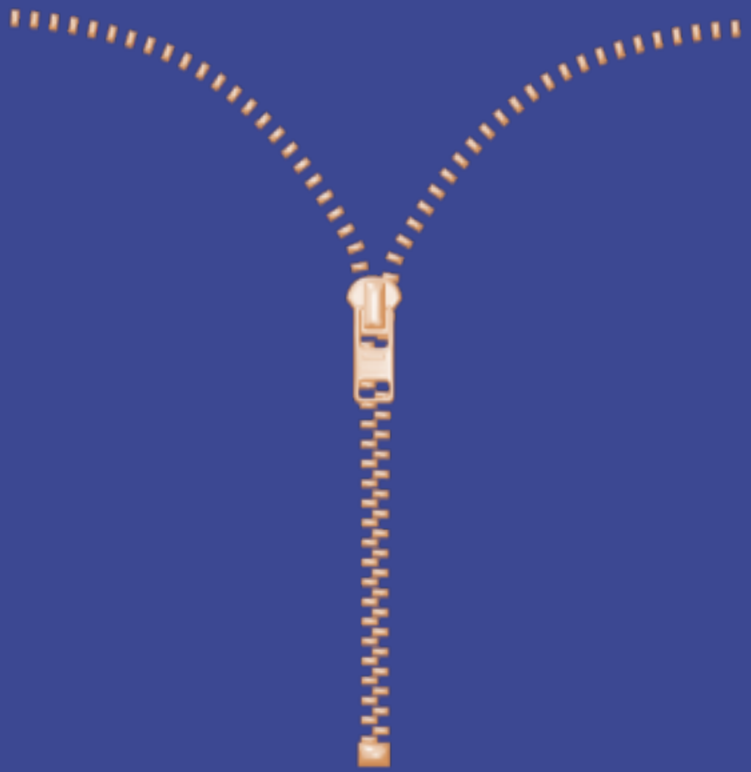
zipWith:: (a -> b -> c) -> [a] -> [b] -> [c]

>zipWith (+) [1..10] [10, 11, 12]

[11,13,15]

>zipWith (\x y -> x*x + y*y) [1, 2] [4..200]

[17,29]



Folding

foldl: takes a function, an initial value and a list and applies the function to the initial value and the first item of the list then to the result and the second item of the list and so on…

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

>foldl (+) 10 [1, 2, 3]

16

10 + 1 = 11
11 + 2 = 13
13 + 3 = 16

>foldl (-) 10 [1, 2, 3]

4

10 - 1 = 9
9 - 2 = 7
7 - 3 = 4

Folding

foldr: takes a function, an initial value and a list and applies the function to the last item of the list and the initial value and then to the next to last item so on…

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

>foldr (+) 10 [1, 2, 3]

16

foldr (-) 10 [1, 2, 3]

-8

3 + 10 = 13
    2 + 13 = 15
        1 + 15 = 16

3 - 10 = -7
    2 - -7 = 9
        1 - 9 = -8

Lazy Evaluation

- Haskell implements lazy evaluation.
- Expressions are only evaluated if and when they are actually needed.
- Arguments are not evaluated before they are passed to a function (call by value), but only when their values are actually used.
- Function calls are treated as promises (thunks).

Eager Evaluation

```
def test(first, second):
    return second

result = test(2/0, 5)
print(result)
```

What gets printed?

A. 5
B. An error

Lazy Evaluation

```
testLazy:: a -> a -> a
testLazy _ x = x

>testLazy  (2/0) 5
```

What is displayed?

A. 5
B. An error

Lists are also lazy

- That is why we can have infinite lists!

```
> xs = [1..]
> take 3 [1..]
[1,2,3]
```

- We can also create infinite lists recursively:

```
> ones = 1:ones
> take 5 ones
[1,1,1,1,1]
```

Lazy Evaluation

```
> sum [1..7]
28
> sum [1..]
```

Don't try this!

```
testLazy:: a -> a -> a
testLazy _ x = x

> testLazy (sum [1..])  5
```

What do we get?

Lazy Evaluation

```
> sum [1..7]
28
> sum [1..]
```

Don't try this!

```
testLazy:: a -> a -> a
testLazy _ x = x

> testLazy (sum [1..])  5
5
```

Currying

- Most programming languages allow functions to have any number of arguments.
- Haskell restricts all functions to have just one argument, without losing any expressiveness.
- This process is called *Currying*, after Haskell Curry.

Currying

```
sos :: Int -> Int -> Int
sos x y = x*x + y*y
```

The function takes its arguments one at a time

sos 2 4 = (sos 2) 4

(sos 2) is a function that takes in a single argument y.

Let's call (sos 2) g:

```
g :: Int -> Int
g  y = 2*2 + y*y
```

sos 2 4 = g 4 = 20

Partial Application

- Partial application means that we don't need to provide all arguments to a function.
- We can therefore create a specialized function by partial application.

```
> g = sos 2
> g 4
20
> f = min 3
> f 5
3
> f 2
2
```

Partial Application

```
> :t (*)
(*) :: Num a => a -> a -> a
> :t (* 2)
(* 2) :: Num a => a -> a
```

This is why we can write :

```
doubles = map (*2) [1 ..5 ]
```

instead of:

```
doubles = map (\x -> 2 * x) [1..5]
```

Why Curry?

- More abstraction
- Partial function applications
- Simpler language
- Based on mathematical functions

Maybe

- Haskell provides Maybe values, which allow us to denote missing results with Nothing. This is similar to Option in Scala
- Maybe is useful when computations fail to generate results. (head of an empty list, lowest element of an empty list, etc…)

Maybe

```
data Maybe a = Nothing
             | Just a
```

Maybe Example

```
lowest :: Ord a => [a] -> Maybe a
lowest (x:xs) =  Just (lowestHelper x xs)
lowest [] = Nothing


lowestHelper :: Ord a => a -> [a] ->  a
lowestHelper x [] = x
lowestHelper x (y:ys) = if x <= y
  then lowestHelper x ys
  else lowestHelper y ys
```

fmap

double (lowest [1, 2, 3])

<interactive>:23:1: error:

- Non type-variable argument in the constraint: Num (Maybe a)
  (Use FlexibleContexts to permit this)
- When checking the inferred type
  it :: forall a. (Ord a, Num a, Num (Maybe a)) => Maybe a

> fmap double (lowest [1, 2, 3])

Just 2

> fmap double (lowest [])

Nothing