Why teaching functional programming to undergraduates at CUNY is important

Evan Misshula

2018-03-28

Plan for the talk

1. Why Functional Programming is intellectually interesting

Plan for the talk

1. Why Functional Programming is intellectually interesting

(particularly with Haskell)

Plan for the talk

1. Why Functional Programming is intellectually interesting

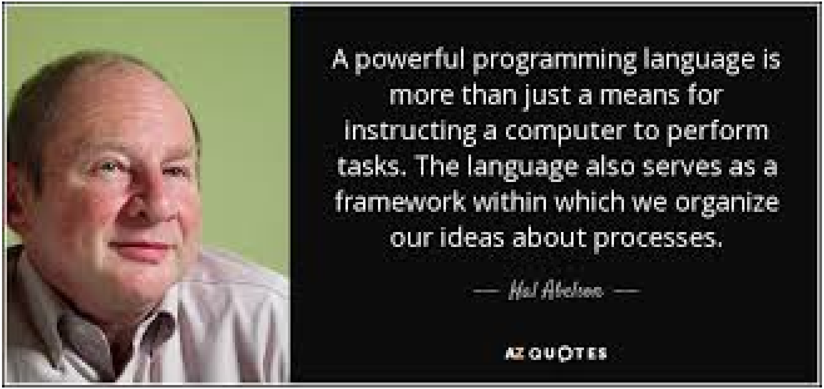
(particularly with Haskell)

1. The size and growth of the Tech sector in NYC
2. The size, growth and earnings of CUNY CS grads
3. The demographic biasis of the Tech Industry relative to NYC Population
4. My thoughts on how helping to close this gap can benefit you and your employer

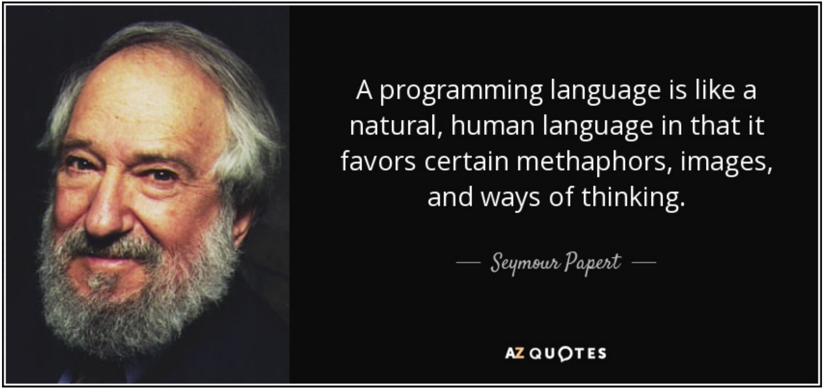
First computers were imperative by necessity

```
55 89 e5 53  83 ec 04 83  e4 f0 e8 31  00 00 00 89  c3 e8 2a 00
00 00 39 c3  74 10 8d b6  00 00 00 00  39 c3 7e 13  29 c3 39 c3
75 f6 89 1c  24 e8 6e 00  00 00 8b 5d  fc c9 c3 29  d8 eb eb 90
```

Programming languages help us think



Languages encourage patterns of thought

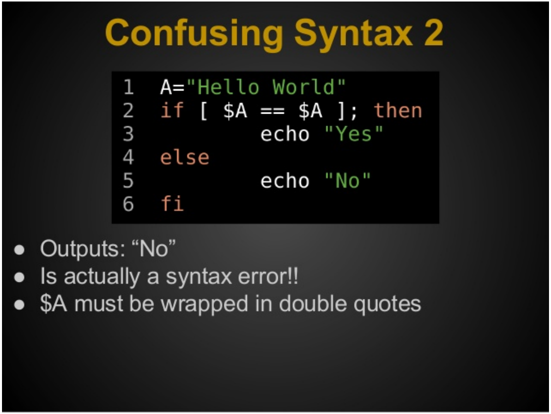


There are dissenting opinions

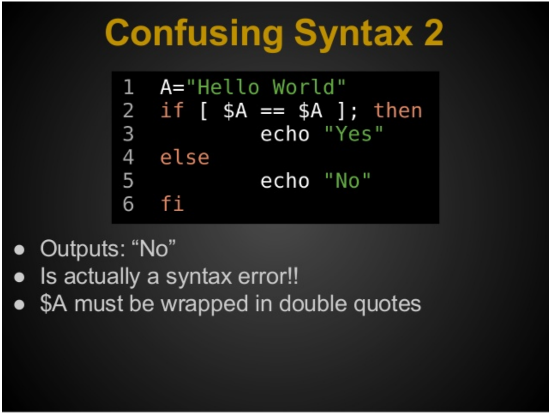
The Value of Programming Paradigms

- To be taught in universities
- To ignite flamewars
- To characterize programming languages
- To inspire memes

Counterexamples of good languages

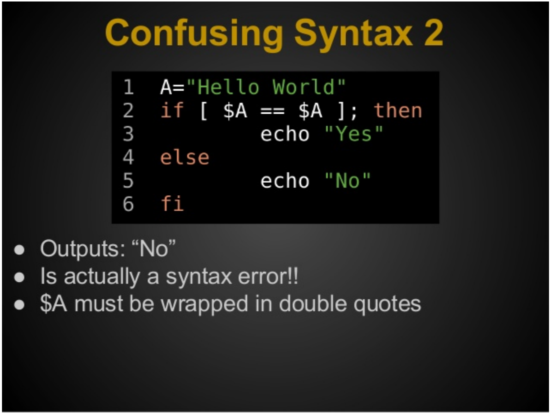


Counterexamples of good languages



- actually the slide is wrong

Counterexamples of good languages



- actually the slide is wrong
- comparison should be `[[`

You can't talk about poor language design and not mention JS

```
console.log(0.1 + 0.2);
console.log(0.1 + 0.2 == 0.3);
```

You can't talk about poor language design and not mention JS

```
console.log(0.1 + 0.2);
console.log(0.1 + 0.2 == 0.3);
```

- 0.30000000000000004
- false

Comparisons can fail

```
console.log(1 < 2 < 3);
console.log(3 > 2 > 1);
```

Comparisons can fail

```
console.log(1 < 2 < 3);
console.log(3 > 2 > 1);
```

- true (1<2) -> true is implicitly coerced to 1 and 1<3
- false (3>2) -> true coerced to 1 and and 1>1 is false

Even assignment is perilous

```
var a= [1,2,3];
a[10]=99;
console.log(a[10])
console.log(a[6])
```

Even assignment is perilous

```
var a= [1,2,3];
a[10]=99;
console.log(a[10])
console.log(a[6])
```

- 99
- [1, 2, 3, <7 empty items>, 99]

Yet Haskell allows us to

Yet Haskell allows us to

- Explore recursion both in functions and in data structures

Yet Haskell allows us to

- Explore recursion both in functions and in data structures
- Rewrite classic sort algorithms in breathtakingly simple form

Yet Haskell allows us to

- Explore recursion both in functions and in data structures
- Rewrite classic sort algorithms in breathtakingly simple form
- Introduce students to algebraic ideas on functions so that they can master abstraction

Right triangle problem

Let's find a problem that puts constraints on tuples

- Which right triangle that has integers for all sides and all sides equal to or smaller than 10 has a perimeter of 24?

Right triangle problem

Let's find a problem that puts constraints on tuples

- Which right triangle that has integers for all sides and all sides equal to or smaller than 10 has a perimeter of 24?
- crack the problem like an egg

Right triangle problem

Let's find a problem that puts constraints on tuples

- Which right triangle that has integers for all sides and all sides equal to or smaller than 10 has a perimeter of 24?
- crack the problem like an egg
- Opportunity to teach: solution by problem relaxation

Right triangle problem relax solution

Integer sides all < 10 and perimeter $= 24$

- generate all tuples of sides less than 10

Right triangle problem relax solution

Integer sides all < 10 and perimeter $= 24$

- generate all tuples of sides less than 10
- designate z as the hypotenuse (bigger than x and y)

Right triangle problem relax solution

Integer sides all < 10 and perimeter = 24

- generate all tuples of sides less than 10
- designate z as the hypotenuse (bigger than x and y)
- make $x^2 + y^2 = z^2$

```
:set +m
length([(x,y,z) | x<-[1..10],y<-[1..10],
z<-[1..10],y<z,x<z,
(x^2 + y^2 == z^2)])
i==i
```

```
Prelude Control.Applicative| Prelude Control.Applicative| 4
```

Adding the perimeter constraint

Let's add constraints

- the perimeter equal 24
- $a + b + c = 24$

```
:set +m

length([(x,y,z) | x<-[1..10],y<-[1..10],z<-[1..10],
y<z,
x+y+z==24,
(x^2 + y^2 == z^2)])
[(x,y,z) | x<-[1..10],y<-[1..10],z<-[1..10],y<z,
  x+y+z==24,
  (x^2 + y^2 == z^2)]
i==i


Prelude Control Applicative> Prelude Control Applicative| Prel
```

Type system

Haskell is statically typed

- Haskell allows students inquire about the type
  - We can see that type by using the ':t' command in the repl:

```
 :t 'a'
 :t True
 :t "HELLO!"
 :t (True, 'a')
 :t 4 == 5
1==1

'a' :: Char
True :: Bool
"HELLO!" :: [Char]
(True, 'a') :: (Bool, Char)
4 == 5 :: Bool
```

Type system

Haskell is statically typed

- Haskell allows students inquire about the type
  - We can see that type by using the ':t' command in the repl:

```
 :t 'a'
 :t True
 :t "HELLO!"
 :t (True, 'a')
 :t 4 == 5
1==1

'a' :: Char
True :: Bool
"HELLO!" :: [Char]
(True, 'a') :: (Bool, Char)
4 == 5 :: Bool
```

Decompose the typeclass

(==) :: Eq a => a -> a -> Bool

- Typeclass constraint
  - The declaration we can read says:

Decompose the typeclass

(==) :: Eq a => a -> a -> Bool

- Typeclass constraint
  - The declaration we can read says:
- The equality function takes two variables of the same type and returns a Bool

Decompose the typeclass

(==) :: Eq a => a -> a -> Bool

- Typeclass constraint
  - The declaration we can read says:
- The equality function takes two variables of the same type and returns a Bool
- The new part 'Eq a =>' says:
- The type must be part of Eq typeclass
  - This is called the class constraint

Interface of Eq

The Eq typeclass provides an interface for testing for equality

- Eq is used for types that support equality testing
  - Its members implement both:
    - '=='
    - '/='

```
5==5
5/=5
'a' == 'a'
"Ho Ha" == "Ho Ha"
3.4 == 3.4
  1==1
```

```
True
False
True
True
```

Introducing Ord typeclass

Ord is for types that have an ordering

- We can see the type of '>' comparison
- We can see some functions which rely on being in the ord typeclass

```
:t (>)
"Abc"< "Zev"
compare "Abc" "Zev"
5 >= 2
compare 5 3
1==1

(>) :: Ord a => a -> a -> Bool
True
LT
True
GT
```

Ord has a connection with inference

Ord is important in statistics

- Ord can be used to explain: ordinal levels of measurement

Ord has a connection with inference

Ord is important in statistics

- Ord can be used to explain: ordinal levels of measurement
- Ord can also be used to introduce: utility curves

Introducing Show typeclass

Everything except function has been part of show

- It works like Java or Ruby's toString methods
- Mostly we use it to examine a value

```
show 3
show 5.334
show True
1==1
```

```
3
5.334
True
```

Introducing Read typeclass

Read is the inverse of show

- It works reads a string and returns a type which supports the interface Read

Introducing Read typeclass

Read is the inverse of show

- It works reads a string and returns a type which supports the interface Read
- You can use it to create Javascript like craziness

Introducing Read typeclass

Read is the inverse of show

- It works reads a string and returns a type which supports the interface Read
- You can use it to create Javascript like craziness
- But you have to work at it

```
read "True" || False
read "8.2" + 3.8
read "5" - 2
read "[1,2,3,4]" ++ [3]
1==1

True
12.0
3
```

Limits to the type inference system

Let's look at a type error

```
read 4
1==1

<interactive>:2327:6: error:
    • Could not deduce (Num String) arising from the literal '4
      from the context: Read a
        bound by the inferred type of it :: Read a => a
        at <interactive>:2327:1-6
    • In the first argument of 'read', namely '4'
      In the expression: read 4
      In an equation for 'it': it = read 4
```

- GHCI is saying it does not know what type to return
  - Do you want an Float or an Integer?

Type specification

We can specify a type

- We just add '::<Type>' and read will work

```
read "5" :: Int
read "5" :: Float
(read "5" :: Int) * 4
read "[1,2,3,4]" :: [Int]
read "(3,'a')" :: (Int, Char)
1==1

5
5.0
20
[1,2,3,4]
(3,'a')
```

Enum type class

Sequentially ordered types

- Being *sequentialy ordered* means that they can be counted in order
- This property is also called being *enumerable*
- We can use them in list ranges
  - they each have a predecessor which you can get with 'pred'
  - they each have a successor which you can get with 'succ'

```
['a'..'e']
[LT .. GT]
[3..7]
succ 'B'
1==1

abcde
[LT,EQ,GT]
[3,4,5,6,7]
'C'
```

Bounded Type class

Bounded type class has concrete types

- with maximum and minimum elements
  - minBound and maxBound are functions with polymorphic type
    - (Bounded a) => a

```
minBound :: Int
maxBound :: Char
maxBound :: Bool
minBound :: Bool
i==i
```

```
-9223372036854775808
'\1114111'
True
False
```

Numeric Types

Numeric types can be operated on mathematically

- Let's look at this type

```
:t (*)
(5 :: Int) * (6 :: Integer)
(5 :: Int) * 6
i==i
```

```
(*) :: Num a => a -> a -> a
<interactive>:2350:15: error:
    • Couldn't match expected type ‘Int’ with actual type ‘Inte
    • In the second argument of ‘(*)’, namely ‘(6 :: Integer)’
      In the expression: (5 :: Int) * (6 :: Integer)
      In an equation for ‘it’: it = (5 :: Int) * (6 :: Integer
30
```

Integral and Floating types

Integral and Floating types

- The Integral typeclass only includes Integer and Int
- The Floating typeclass only includes floats and double

```
:t fromIntegral
fromIntegral (length [1,2,3,4]) + 3.2
i==i

fromIntegral :: (Num b, Integral a) => a -> b
7.2
```

Curried Functions

Every function in haskell only takes one argument

- But what about 'max' or min?
- We actually apply parameters to functions one at time
  - These are called "curried" functions
    - This is after Haskell Curry

    max (Ord a) => a -> a -> a

    max (Ord a) => a -> (a -> a)
- If we call a function with to few parameters we get back a partially applied function

```
:set +m

-- multThree :: (Num a) => a -> a -> a -> a
multThree x y z = x * y * z
multThree 3 5 9 ==  ((multThree 3) 5) 9
i==1
```

Curried comparison

Here is a curried comparison

- These are the same because 'x' is on both sides of the equation

```
-- compareWithHundred :: (Num a, Ord a, Show a) => a -> Orderi
compareWithHundred x = compare 100 x

-- compareWithHundred1 :: (Num a, Ord a, Show a) => a -> Orderi
compareWithHundred1 = compare 100
```

Example partial application

Let's look at an infix function

- simply surround the function with parentheses and only supply one of the parameters
- this is called 'sectioning'

```
-- divideByTen :: (Floating a) => a -> a
divideByTen = (/10)
```

partial application of a string function

String functions can be partially applied too

- this is written in point free style
- it is also sectioned

```
-- isUpperAlphanum :: Char -> Bool
isUpperAlphanum = (`elem` ['A'..'Z'])
```

Returned functions

Functions can return functions

- take a function and apply it twice

```
-- applyTwice :: (a -> a) -> a -> a
applyTwice f x = f (f x)
```

ZipWith

We are going to implement ZipWith

- It joins two lists and performs a function on the corresponding elements

```
-- zipWith' :: (a -> b -> c) -> [a] -> [b] -> [c]
zipWith' _ [] _ = []
zipWith' _ _ [] = []
zipWith' f (x:xs) (y:ys) = f x y : zipWith' f xs ys
```

flip

flip changes the order of the arguements

```
-- flip' :: (a -> b -> c) -> (b -> a -> c)
flip' f = g
    where g x y = f y x

-- flip'' :: (a -> b -> c) -> b -> a -> c
flip'' f y x = f x y
```

Maps and Filters

Map

- map takes a function applies the function to each element of a list

```
-- map :: (a -> b) -> [a] -> [b]
map _ [] = []
map f (x:xs) = f x : map f xs
```

Filter

Filter

- 'filter' take a function called a predicate and a list of any type
- the predicate takes an element of the list and returns a Bool
    - the filter returns elements for which the predicate is True

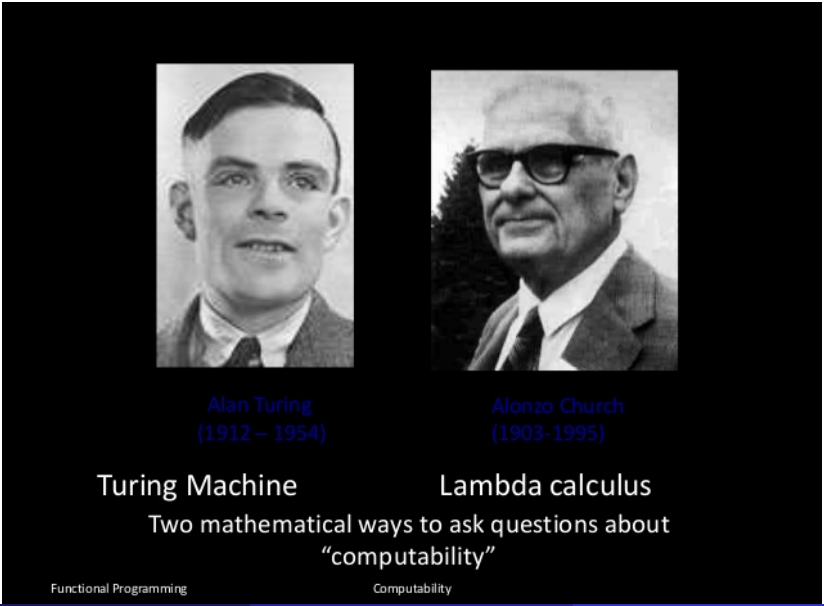
```
filter :: (a -> Bool) -> [a] -> [a]
filter _ [] = []
filter p (x:xs)
    | p x       = x : filter p xs
    | otherwise = filter p xs
```

Lambdas

Lamdas are anonymous functions

- These are unnamed functions
- They are passed as parameters to other functions
- They work like composition in math
- They are called 'lambdas' because of the 'lambda calculus'

Church and Turing



Lambda Calculus

Lambda Calculus is a formal system for computation

- it is equivelent to calculation by Turing Machine
- invented by Alonzo Church in the 1930's
- Church was Turing's thesis advisor
  - a function is denoted by the greek letter λ
  - a function f(x) that maps x -> f(x) is:
    - $\lambda x.y$

Example of a lambda

We can pass a lambda to ZipWith

- a lambda function in Haskell starts with '\'
- can't define several parameters for one para,eters

```
zipWith (\a b -> (a * 30 + 3) / b) [5,4,3,2,1] [1,2,3,4,5]
1==1
```

Quicksort

specification

```
-- quicksort :: (Ord a) => [a] -> [a]
quicksort [] = []
quicksort (x:xs) =
    let smallerSorted = quicksort [a | a <- xs, a <= x]
biggerSorted = quicksort [a | a <- xs, a > x]
    in  smallerSorted ++ [x] ++ biggerSorted
1==1
```

Folds

Folds encapsulate several functions with (x:xs) patterns

- they reduce a list to a single value
- 'foldl' is the left fold function

```
sum' :: (Num a) => [a] -> a
sum' xs = foldl (\acc x -> acc + x) 0 xs

sum'' :: (Num a) => [a] -> a
sum'' = foldl (+) 0
1==1
```

Function application

Function application with $

- '$' is called the *function application*
- changes to right association
- keeps us from writing parentheses

```
map ($ 3) [(4+), (10*), (^2), sqrt]
1==1

[7.0,30.0,9.0,1.7320508075688772*** Exception: <interactive>:23
```

Function composition

Function composition is just like math

- In math $f \cdot g(x) = f(g(x))$
- Let's look at Haskell function
- g takes a -> b
- f takes b -> c

Function composition

Function composition is just like math

- In math $f \cdot g(x) = f(g(x))$
- Let's look at Haskell function
- g takes a -> b
- f takes b -> c
- so the composition take f . g takes a -> c

```
(.) :: (b -> c) -> (a -> b) -> a -> c
f . g = \x -> f (g x)
```

Function composition examples

Function composition examples

- with a λ
- with point free notation

```
map (\x -> negate (abs x)) [5,-3,-6,7,-3,2,-19,24]
map (negate . abs) [5,-3,-6,7,-3,2,-19,24]

[-5,-3,-6,-7,-3,-2,-19,-24*** Exception: <interactive>:2394:1-
```

Polymorphism on a higher level

- Types are not part of a hierarchy
- We can think about how they should act
  - then connect them with typeclasses

Functors defined

Definition (definition)

A functor is a typeclass for all the things that can be mapped over

Functors defined

Definition (definition)

A functor is a typeclass for all the things that can be mapped over

Definition (Haskell syntax definition)

- class Functor f where

  fmap (a -> b) -> f a -> f b

Analogy with other typeclasses

Typeclasses define functions

- Eq *defines* concrete types that are equatable
  - functions ('=') and ('/')
- Ord *defines* concrete types that 'orderabe'
  - implements the 'compare' function
- Enum *defines* concrete types that enumerable
  - defines '..' a range

List Functor examples

Example (List Functor Examples)

- map:: (a -> b) -> [a] -> [b]
- instance Functor [] where
  - fmap = map

Functor code in the repl

List Functor in the repl

```
  :t map
  fmap (*2) [1..3]
  map (*2) [1..3]
i==i

map :: (t -> a) -> [t] -> [a]
[2,4,6]
[2,4,6*** Exception: <interactive>:2394:1-29: Non-exhaustive pa
```

Maybe Functor examples

Example (Maybe Functor Examples)

```
type myMaybe a = Nothing | Just a

instance Functor myMaybe where
  fmap f (Just x) = Just (f x)
  fmap f Nothing = Nothing
```

Maybe Functor code in the repl

Maybe Functor in the repl

```
:t fmap
fmap (++ " HEY GUYS IM INSIDE THE JUST") (Just "Something seri
fmap (++ " HEY GUYS IM INSIDE THE JUST") Nothing
fmap (*2) (Just 200)
fmap (*2) Nothing
i==i

fmap :: Functor f => (a -> b) -> f a -> f b
Just "Something serious. HEY GUYS IM INSIDE THE JUST"
Nothing
Just 400
Nothing
```

Functor Law intuition

If functors mean that something can be mapped over...

- then calling 'fmap' on a functor should
  - map a function over the functor

Functor Law intuition

If functors mean that something can be mapped over. . .

- then calling 'fmap' on a functor should
  - map a function over the functor
- Nothng else

The First Functor Laws

Definition (The First Functor Law)

states that if we map the identity (id) function over a functor, we get the functor

- fmap id = id

Identity in the Repl

Identity functions in the repl

```
fmap id (Just 3)
id (Just 3)
fmap id [1..5]
id [1..5]
fmap id []
fmap id Nothing
1==1

Just 3
Just 3
[1,2,3,4,5]
[1,2,3,4,5]
[]
Nothing
```

The Second Functor Law

Definition (The Second Functor Law says)

The Second Functor Law says that composing two functions and then mapping the composed function over a functor is the same as first mapping one function over the functor and then mapping the other one.

- fmap (f.g) = fmap f . fmap g
- fmap (f.g) F = fmap f (fmap g F)

Composition in the Repl

Composition functions in the repl

```
fmap ((+1).(*2)) (Just 3)
fmap (+1) (fmap (*2) (Just 3))
fmap  ((+1).(*2)) [1..5]
fmap (+1) (fmap (*2) [1..5])
1==1

Just 7
Just 7
[3,5,7,9,11]
[3,5,7,9,11]
```

What if we map a multi-parameter function over a functor?

- Look at the type signature

```
a = fmap (*) [1..4]
:t a
fmap (\f -> f 9) a
1==1
```

```
a :: (Num a, Enum a) => [a -> a]
[9,18,27,36]
```

What if we want to take a function out of a Just

Let's take a Just (3 *) and map

and map it over Just 5

```
:set +m
:{
class (Functor f) => Applicative f where
    pure :: a -> f a;
    (<*>) :: f (a -> b) -> f a -> f b
:}
```

Maybe Applicative

Let's look at the Applicative for Maybe

```
:set +m
:{
instance Applicative MyMaybe where
    pure = Just
    Nothing <*> _ = Nothing
    (Just f) <*> something = fmap f something
:}
```

Maybe Applicative inside the repl

Using the Maybe Applicative

```
-- :add Control.Applicative
Just (+3) <*> Just 9
pure (*2) <*> Just 10
pure (+3) <*> Just 9
Just (++"!!") <*> Just "Go now"
Nothing <*> Just "woot"
1==1


<interactive>:2476:1: error:
    • Could not deduce (Applicative Maybe) arising from a use o
      from the context: Num b
        bound by the inferred type of it :: Num b => Maybe b
        at <interactive>:2476:1-20
    • In the expression: Just (+ 3) <*> Just 9
      In an equation for ‘it’: it = Just (+ 3) <*> Just 9
```

Fmap as an infix operator

Control.Applicative exports a function called <$>

which is fmap as an infix operator

```
(<$>) :: (Functor f) => (a->b) -> f a -> f b
f <$> x = fmap f x
```

Compare Applicatives in the repl

Infix fmap in the repl

```
(++) <$> Just "John " <*> Just "Travolta"
(++) "John " "Travolta"
1==1

<interactive>:2486:1: error:
    • No instance for (Applicative Maybe) arising from a use of
    • In the expression: (++) <$> Just "John " <*> Just "Travol
      In an equation for ‘it’:
          it = (++) <$> Just "John " <*> Just "Travolta"
John Travolta
```

Lists are Applicative Functors

Definition (Definition of the Applicative for a list)

- Literally a Cartesian product of functions and list values

```
:set +m
:{
instance Applicative [] where
    pure x = [x]
    fs <*> xs = [f x | f <- fs, x<- xs]
:}
```

Applicative Functors of lists in the repl

Applicative Functors of lists in the repl

```
[(*0),(+100),(^2)] <*> [1..4]
[(+),(*)] <*>[1,2]<*>[3,4]
(++) <$> ["ha","heh","hmm"] <*> ["?","!","."]
1==1

[0,0,0,0,101,102,103,104,1,4,9,16]
[4,5,5,6,3,4,6,8]
["ha?","ha!","ha.","heh?","heh!","heh.","hmm?","hmm!","hmm."]
```

IO is an Applicative

Let's see how the IO Applicative is implemented:

```
:set +m
:{
instance Applicative IO where
    pure = return
    a <*> b = do
f <- a
x <- b
return (f x)
:}
```

Concatenating IO strings

Two ways to concatenate two lines of user input string

- Imperative code

```
:set +m
:{
myAction :: IO String
myAction = do
    a <- getLine
    b <- getLine
    return $ a ++ b
:}
```

Applicative way to concatenate two lines of user input string

- Applicative code

```
:set +m
:{
myAction :: IO String
myAction = (++)
    <$> getLine
    <*> getLine
:}
```

The first Applicative Functor Law

Theorem (The first Applicative Functor Law)

- *pure f <*> x = fmap f x*

Some lessons we've skipped

Defining types

- *data* will define a new algebraic type
- *type* creates a type synonym
- *newtype* creates new types from old types

Applicative Functor in two ways

function left, each argument right

```
:m Control.Applicative
[(+1),(*100),(*5)] <*> [1..3]
1==1


[2,3,4,100,200,300,5,10,15]
```

function left, every argument right

```
:set +m
:{
instance Applicative ZipList whe
pure x = ZipList (repeat x)
ZipList fs <*> ZipList xs = Zip
:}
  getZipList $ ZipList [(+1),(*
  -- getZipList $
  -- ZipList [(+1),(*100),(*5)]
  --  <*> ZipList [1,2,3]
1==1


Prelude Control.Applicative| Pr
```

The newtype keyword

'newtype' takes one type and wrap it

- to present it as another type

```
newtype ZipList a = ZipList {getZipList ::  [a]}
```

- data can have multiple value contstructors

type vs. newtype vs. data examples

'data' to make new types

- Here are additive and multiplicative types with multiple constructors

```
data Profession = Fighter | Archer | Wizard
data Species = Human | Elf | Orc | Goblin
data PlayerCharacter = PlayerCharacter Species Profession
```

Using newtype to drive typeclass properties

newtype

```
newtype CharList = CharList {getCharList :: [Char]} deriving(Ec
CharList "this will be shown!"
CharList "benny" == CharList "benny"
CharList "benny" == CharList "oisters"
1==1


CharList {getCharList = "this will be shown!"}
True
False
```

Monoid Definition

Definition (Monoid definition)

A data type, category or set is a monoid if it has a binary operation $\bullet$ which is associative and has an identity.

- $\forall a, b, c \in S, (a \bullet b) \bullet c = a \bullet (b \bullet c)$
- $e \bullet a = a \bullet e = a$

```
:set +m
:{
class Monoid m where
    mempty :: m
    mappend :: m -> m -> m
    mconcat :: [m] -> m
    mconcat = foldr mappend mempty
:}
```

Monoid functions defined

Defining the monoid functions

- 'mempty' is just the identity function
- mappend is the binary function
  - it doesn't just append
- mconcat reduces a list of monoid values and reduces them to one by applying mappend

Monoid Laws

Theorem (The Monoid Laws are just the definition in Haskell)

- $mappend\ mempty\ x = x$
- $mappend\ x\ mempty = x$
- $mappend\ (mappend\ x\ y)\ z = mappend\ x\ (mappend\ y\ z)$

Monoid examples

Example (List is a monoid)

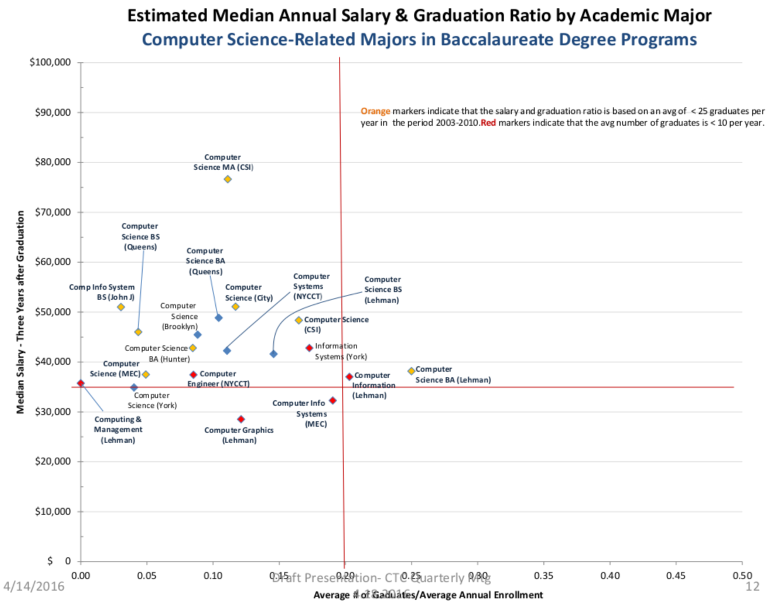
- [] with (++) is a monoid
  - id = ""
- Natural numbers with (*) is a monoid
  - id = 1
- Natural numbers with (+) is a monoid
  - id = 0

Why is all of this important to you

- BLS Statistics
- 2015 median salary is \$100,690
- Number of jobs: 1,114,000
- Job growth: 17% (much faster than average)

From NY State Comptroller's office

- The Technology Sector in New York City 4/2018
- New York State had the third-largest tech sector in the nation in 2016.
- Employment in NYC's tech sector increased by 57% between 2010 and 2016 (46,900 jobs), 3x faster than the rest of the private sector
- The average salary increased 3x faster than the rest of the City's private sector to reach a record $147,300 by 2016


Estimated Median Annual Salary & Graduation Ratio by Academic Major
Computer Science-Related Majors in Baccalaureate Degree Programs
Orange markers indicate that the salary and graduation ratio is based on an avg of < 25 graduates per year in the period 2003-2010. Red markers indicate that the avg number of graduates is < 10 per year.
Median Salary - Three Years after Graduation
Average # of Gaduates/Average Annual Enrollment
Computer Science MA (CSI)
Computer Science BS (Queens)
Computer Science BA (Queens)
Comp Info System BS (John J)
Computer Science (Brooklyn)
Computer Science (City)
Computer Systems (NYCCT)
Computer Science BS (Lehman)
Computer Science (CSI)
Information Systems (York)
Computer Science BA (Hunter)
Computer Science (MEC)
Computer Engineer (NYCCT)
Computer Information (Lehman)
Computer Science BA (Lehman)
Computer Science (York)
Computing & Management (Lehman)
Computer Graphics (Lehman)
Computer Info Systems (MEC)
Draft Presentation- CTC Quarterly Mtg
4/14/2016
12

NYC Demographics

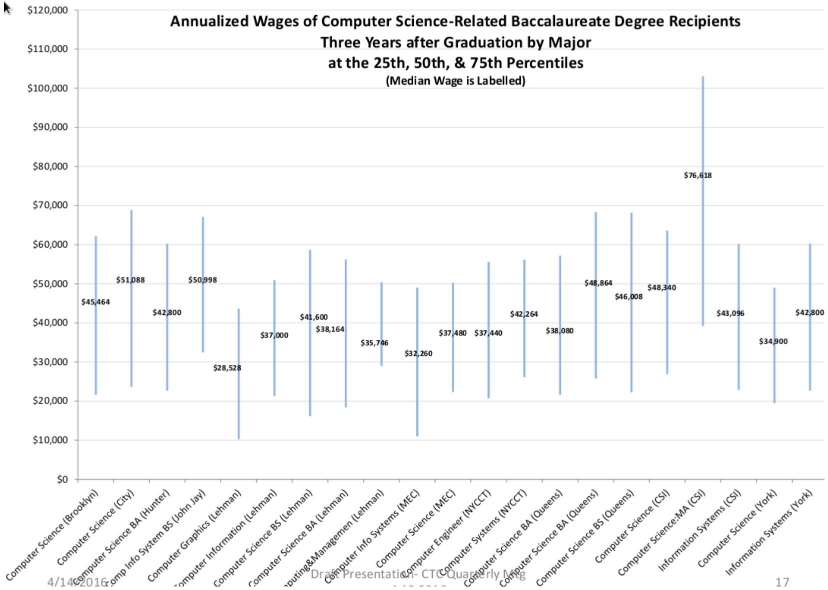
In New York City, 44.6% of the population is white, 25.1% is black, and 11.8% are of Asian descent. Hispanics of any race represent about 27.5% percent of New York City's population

- US Census 2018

NYC Tech Sector does not reflect our diversity

[B]lacks and Latinos constitute 25.1 percent and 27.5 percent of the population, respectively, but only 9 percent and 11 percent, respectively, are employed in the tech sector.

- City Limits: Why is NYC Tech so White?



Annualized Wages of Computer Science-Related Baccalaureate Degree Recipients
Three Years after Graduation by Major
at the 25th, 50th, & 75th Percentiles
(Median Wage is Labelled)
$120,000
$110,000
$100,000
$90,000
$80,000
$70,000
$60,000
$50,000
$40,000
$30,000
$20,000
$10,000
$0
$45,464
$51,088
$42,800
$50,998
$28,528
$37,000
$41,600
$38,164
$35,746
$32,260
$37,480
$37,440
$42,264
$38,080
$48,864
$46,008
$48,340
$76,618
$43,096
$34,900
$42,800
Computer Science (Brooklyn)
Computer Science (City)
Computer Science BA (Hunter)
Comp Info System BS (John Jay)
Computer Graphics (Lehman)
Computer Information (Lehman)
Computer Science BS (Lehman)
Computer Science BA (Lehman)
Computing&Managemen (Lehman)
Computer Info Systems (MEC)
Computer Science (MEC)
Computer Engineer (NYCCT)
Computer Systems (NYCCT)
Computer Science BA (Queens)
Computer Science BA (Queens)
Computer Science BS (Queens)
Computer Science (CSI)
Computer Science:MA (CSI)
Information Systems (CSI)
Computer Science (York)
Information Systems (York)
4/14/2016
Draft Presentation- CTC Quarterly Mtg
17

References

- CUNY Student Experience 2016
- NYC Tech is 62% White, 60% male
- NYC Tech Profile
- Why NYC's Growing Tech Sector is so White
- Numbers say New York's tech boom is real
- Will Silicon Alley Be the Next Silicon Valley?
- The Technology Sector in New York City
- NYC Population
- CUNY 2x Initiative
- Peter Drake's Prog Lang Course Materials
- `http://learnyouahaskell.com/`