# FUNCTIONAL PROGRAMMING

## (LEARN YOU A HASKELL)

Created by Chris Foster / @chrisfosterelli

# WHAT IS IT?

- It's a *programming paradigm*
- Combinatory logic develops in 1920s
- Lambda calculus develops in 1930s
- Lisp developed in 1950s
- Expansion of many, many new languages

# WHY LEARN IT?

If we list all the natural numbers below 10 that are multiples of 3 or 5, we get 3, 5, 6 and 9. The sum of these multiples is 23. Find the sum of all the multiples of 3 or 5 below 1000.

# JAVA

```java
public class Multiple {
  public static void main(String args[])
  {
    int cont = 0;
    for (int i=0; i < 1000; i++)
    {
      if (i % 3 == 0 || i % 5 == 0)
      {
        cont = cont + i;
      }
    }
    System.out.println(cont);
  }
}
```

# HASKELL

```
sum $ [ a | a <- [1..999], 0 `elem` fmap (mod a) [3, 5] ]
```

# WHY LEARN IT?

Remember how many lines quicksort took you in Java?

# HASKELL

```haskell
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
```

# WHY LEARN IT?

Functional programming is better at solving many types of problems, and you can apply the concepts to imperative languages you use for everyday programming as well.

# WHY LEARN HASKELL?

- *Functional programming at its best!*
- Haskell is a purely functional language
- Widely used in academia and industry
- Excellently demonstrates functional concepts

# WHERE IS HASKELL USED?

- Spam filtering
- Semiconductor design
- Cryptographic algorithm design
- Web frameworks
- Military simulations
- Aerospace systems
- Education

# HASKELL IS HARD

- Functional programming is new and difficult for you
- Haskell is a very big language with a lot of components
- You won't need to fully understand Haskell today!
- At the end, I'll give you resources for learning Haskell

# HASKEL 101

Let's cover the basics of Haskell

# VARIABLE ASSIGNMENT

Variables are assigned with the 'let' keyword

```
let a = 10
```

# FUNCTIONS

Functions are defined by writing them in equation format

```
doubleMe x = x + x
```

To call a function, you simply write it's name

```
ghci> doubleMe 10
20
```

# FUNCTIONS

Your functions can call other functions, like you'd expect

```
doubleUs x y = doubleMe x + doubleMe y
```

# LISTS

Haskell has standard, intuitive lists

```
ghci> let lostNumbers = [4,8,15,16,23,42]
ghci> lostNumbers
[4,8,15,16,23,42]
ghci> [1,2,3,4] ++ [9,10,11,12]
[1,2,3,4,9,10,11,12]
```

In Haskell, strings are simply lists of characters

```
ghci> "hello" ++ " " ++ "world"
"hello world"
```

You can append to lists and get elements from them

```
ghci> 5:[1,2,3,4,5]
[5,1,2,3,4,5]
ghci> [9.4,33.2,96.2,11.2,23.25] !! 1
33.2
```

# LIST GENERATION

Haskell can smartly generate lists for you

```
ghci> [1..20]
[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20]
ghci> ['a'..'z']
"abcdefghijklmnopqrstuvwxyz"
ghci> [2,4..20]
[2,4,6,8,10,12,14,16,18,20]
```

Haskell also has list comprehensions

```
ghci> [x*2 | x <- [1..10]]
[2,4,6,8,10,12,14,16,18,20]
ghci> [ x | x <- [50..100], x `mod` 7 == 3]
[52,59,66,73,80,87,94]
```

# A SIMPLE FUNCTION

Can you guess what this does?

```
removeNonUppercase st = [ c | c <- st, c `elem` ['A'..'Z']]
```

# NEAT!

```
ghci> removeNonUppercase "Hahaha! Ahahaha!"
"HA"
ghci> removeNonUppercase "IdontLIKEFROGS"
"ILIKEFROGS"
```

# GUARDS

Here is something you've not seen before

```
bmiTell :: (RealFloat a) => a -> String
bmiTell bmi
    | bmi <= 18.5 = "You're underweight, you emo, you!"
    | bmi <= 25.0 = "You're supposedly normal."
    | bmi <= 30.0 = "You're fat! Lose some weight, fatty!"
    | otherwise   = "You're a whale, congratulations!"
```

```
ghci> bmiTell 25
"You're supposedly normal."
```

# INNER FUNCTIONS

Functions can be inline in Haskell

```
myCompare :: (Ord a) => a -> a -> Ordering
a `myCompare` b
    | a > b     = GT
    | a == b    = EQ
    | otherwise = LT
ghci> 3 `myCompare` 2
GT
```

# TUPLES

## Haskell has python-like tuples

```
ghci> zip [1,2,3,4,5] [5,5,5,5,5]
[(1,5),(2,5),(3,5),(4,5),(5,5)]
ghci> let triangles = [ (a,b,c) |
  c <- [1..10],
  b <- [1..10],
  a <- [1..10] ]
```

# LET ... IN

You can clean up your functions with let ... in

```
cylinder :: (RealFloat a) => a -> a -> a
cylinder r h =
    let sideArea = 2 * pi * r * h
        topArea = pi * r ^2
    in  sideArea + 2 * topArea
```

# IF STATEMENTS

Haskell has if statements!

```
ghci> if 5 > 3 then "Woo" else "Boo"
"Woo"
ghci> 4 * (if 10 > 5 then 10 else 0) + 2
42
```

# IMPERATIVE VS FUNCTIONAL

- There is no loops
- There is no side effects
- Functions are very different
- Classes are very different

# FUNCTIONAL MEANS

- Immutability
- Type systems
- Lazy Evaluation
- Referential Transparency
- Pattern Matching
- Higher Order Functions
- Recursion

# IMMUTABILITY

- A variable that is set cannot be changed
- An array's contents cannot be changed
- Any objects cannot be altered or changed
- Instead, immutability requires you to copy data
- *Everything* in Haskell is immutable

# LAZY EVALUATION

- Code that you write doesn't necessarily run
- Haskell doesn't do anything until it has to
- This behaviour is also called *non-strict evaluation*
- This allows for things like infinitely long data structures

```
Prelude> let a = [1..]
Prelude> let b = take 5 a
Prelude> print b
[1,2,3,4,5]
Prelude>
```

# TYPE SYSTEMS

- Haskell is a *statically typed language*
- Haskell relies heavily on *type inference*

```
Prelude> let a = [1..]
Prelude> :t a
a :: (Enum t, Num t) => [t]
Prelude>
```

- Types in Haskell are totally different than what you know
- Functional programming using *algebriac types*
- Type instances, type classes, data types, oh my!
- *We won't get into these*

# ELIMINATING SIDE EFFECTS

## (REFERENTIAL TRANSPARENCY)

Given the *same parameters*,
a function produces the *same result* every single time.

- Functions cannot have *side effects*
- A function takes parameters, and produces a result

# PATTERN MATCHING

- Pattern matching allows flexiblity in functions and code
- You can match conditions or extract values with this

```
lucky :: (Integral a) => a -> String
lucky 7 = "LUCKY NUMBER SEVEN!"
lucky x = "Sorry, you're out of luck, pal!"
```

```
Prelude> lucky 7
"LUCKY NUMBER SEVEN!"
Prelude> lucky 5
"Sorry, you're out of luck pal!"
```

# RECURSION

- *Well, we have recursion in Java...*
- Haskell is *built* on recursion instead of loops

```
maximum' :: (Ord a) => [a] -> a
maximum' [] = error "maximum of empty list"
maximum' [x] = x
maximum' (x:xs)
    | x > maxTail = x
    | otherwise = maxTail
    where maxTail = maximum' xs
```

# HIGHER ORDER FUNCTIONS

- Lambda functions
- Curried functions
- Function composition
- Function application

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

```
ghci> sum (takeWhile (<10000) (filter odd (map (^2) [1..])))
166650
```

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

# PURE FUNCTIONAL CODE

- So, how does IO work purely?
- The answer: it doesn't!
- Haskell has *pure code*
- Haskell has *impure code (IO)*

```
main = do
    line <- getLine
    if null line
        then return ()
        else do
            putStrLn $ reverseWords line
            main

reverseWords :: String -> String
reverseWords = unwords . map reverse . words
```

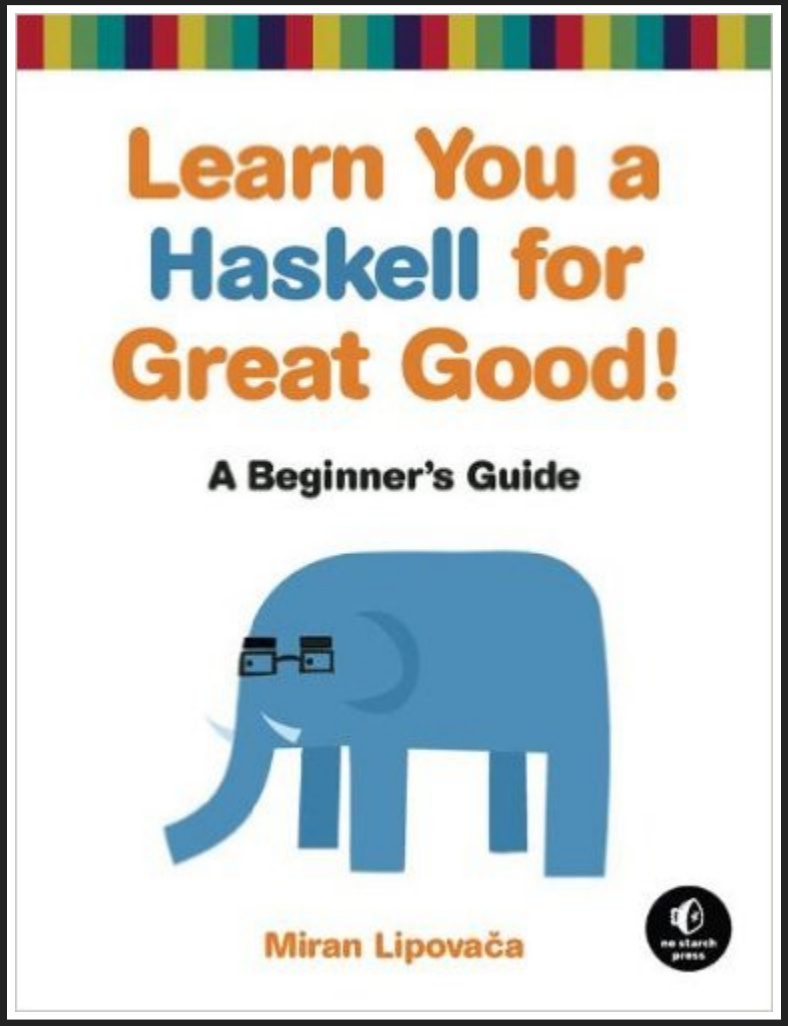
# NON-FUNCTIONAL LANGUAGES

- Javascript - underscore, lodash, and ES6
- Java 8 - Lambda expressions

# AND MORE!

- Monads
- Monoids
- Functors
- Seriously advanced standard lib
- Advanced typesystem
- Applicative Functors

# KEEP GOING!


Learn You a Haskell for Great Good!
A Beginner's Guide
Miran Lipovača
no starch press

# THE END

- @chrisfosterelli
- https://fosterelli.co