

# Getting Started In Haskell

Wayne Chang (wayne@wycd.net)

Nov 4th, 2017

# Slides and Exercises

With Git installed, you can run this command to download the slides and exercises.

```
$ git clone https://github.com/wyc/haskell-intro.git
```

# Table of Contents

These slides have coverage of:

- ▶ Language Overview
- ▶ Installation
- ▶ Basic Data Types
- ▶ Pattern Matching
- ▶ Algebraic Data Types
- ▶ Type and Value Constructors
- ▶ Introduction to Functors
- ▶ References and Resources

glhf!

# Introduction

Haskell is *statically-typed*, *functional*, *immutable*, and *lazy*.

## Statically-Typed

- ▶ Types are checked at compile time, as opposed to runtime.

## Functional

- ▶ “Code is data.”

## Immutable

- ▶ Values do not get overwritten.

## Lazy

- ▶ Evaluation is deferred for as long as possible.

## Statically-Typed

Types are checked at compile time, as opposed to runtime.

```
int add(int x, int y) {  
    return x + y;  
}
```

```
int main() {  
    add(1, 2);  
    add(1, "two");  
    return 0;  
}
```

```
$ gcc -o s s.c
```

```
s.c: In function 'main':
```

```
s.c:9:20: warning: passing argument 2 of 'add'
```

```
    makes integer from pointer without a cast
```

```
        add(1, "two");
```

```
        ^
```

# Functional

“Code is data.”

Functions are passed around as easily as any other values, getting “first-class” treatment. This style allows us operate on whole data structures as opposed to just pieces of them at a time.

```
$ node  
> [1, 2, 3, 4, 5]  
  .map((elem) => elem * 2)  
  .reduce((acc, elem) => acc * elem, 1)  
3840
```

# Immutable

Values do not get overwritten, or else you get yelled at.

```
$ node
```

```
> const a = 5
```

```
undefined
```

```
> a = 6
```

```
TypeError: Assignment to constant variable.
```

# Immutable

## What's the difference between Caching and Memoization?

- ▶ Caching is hoping that the result happens to be in a fast store, and then fetching it if it isn't.
- ▶ Memoization is remembering the mapping of specific inputs to their results for a function.

A function can only be memoized if it is *referentially transparent*, otherwise known as *immutable* or *pure*; the same inputs must always produce the same result.



# Lazy

Evaluation is deferred for as long as possible.

“Don’t start cleaning the apartment until the doorbell rings, and then only clean the parts that they can see.”

```
Prelude> [1..]
```

```
[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,27,28,29,30,31,32,33,34,35,36,37,38,39,40,41,42,43,44,45,46,47,48,49,50,51,52,53,54,55,56,57,58,59,60,61,62,63,64,65,66,67,68,69,70,71,72,73,74,75,76,77,78,79,80,81,82,83,84,85,86,87,88,89,90,91,92,93,94,95,96,97,98,99,100]
```

```
Prelude> take 5 [1..]
```

```
[1,2,3,4,5]
```

# Goodbye to Pretense

You don't need category theory, abstract algebra, or graduate-level algorithms to use or understand Haskell.

# Hello, functional programming

To to remove the first three elements:

OOP-like:

```
[1,2,3,4,5].drop(3)  // [4,5]
```

Functional:

```
drop 3 [1,2,3,4,5]  -- [4,5]
```

In functional programming, we focus on the functions! When you focus on the objects themselves, you're "object-oriented".

# Functional programming

Focus on functions.

- ▶ We ask “what can this function operate on?” and not “what can this object do?”
- ▶ What if we didn't have countless POJOs, and instead some core functions that we can use over and over again?

Composability is key!

```
Prelude> map (drop 1) [[1, 2, 3], [4, 5, 6], [7, 8, 9]]  
  
[[2,3],[5,6],[8,9]]
```

## Getting Started (1/5)

1. Install Stack and the Glasgow Haskell Compiler (GHC) for GNU/Linux, Mac OSX, or Windows from <https://docs.haskellstack.org/en/stable/README/>
2. Setup Stack.

```
$ stack setup
```

3. Write hello.hs:

```
main :: IO ()  
main = do  
  putStrLn "Hello, World!"
```

## Getting Started (2/5)

4. Compile it and run!

```
$ stack ghc hello.hs
```

```
$ ./hello
```

```
Hello, World!
```

5. Alternatively, you can use runhaskell:

```
$ stack runhaskell hello.hs
```

```
Hello, World!
```

## Getting Started (3/5)

7. Write a function to use in GHC's interactive environment, `ghci. myfuncs.hs`:

```
salesTax :: Double -> Double
salesTax price = price * 0.089
```

8. Enter the `ghci` shell and load your source file.

```
$ stack ghci
GHCi, version 8.0.2: http://www.haskell.org/ghc/
Prelude> :load myfuncs
[1 of 1] Compiling Main ( myfuncs.hs, interpreted )
Ok, modules loaded: Main.
*Main> salesTax 100
8.9
```

## Getting Started (4/5)

### 7. Importing modules:

```
-- Haskell  
import Data.Char  
import Data.Char (ord)  
import qualified Data.Char as C
```

```
ord 'a'      -- 97  
C.ord 'a'    -- 97
```

```
# Python  
import data.char  
import ord from data.char  
import data.char as C
```

```
ord('a')     # 97  
C.ord('a')    # 97
```



## Getting Started (5/5)

8. Try importing from ghci.

```
$ stack ghci
```

```
GHCI, version 8.0.2: http://www.haskell.org/ghc/
```

```
Prelude> import Data.Char (ord)
```

```
Prelude Data.Char> ord 'a'
```

```
97
```

# Basic Types

Types start with a Capital letter, variables (including functions and type variables) start with lowercase.

- ▶ Int
  - ▶ Fixed-Precision Whole Number
- ▶ Integer
  - ▶ Unbounded Whole Number
- ▶ Float
  - ▶ Single-Precision Floating
- ▶ Double
  - ▶ Double-Precision Floating
- ▶ Bool
  - ▶ True or False
- ▶ Char
  - ▶ Single character, such as 'a'

## Lists (1/2)

### ► List

- For example, list of Char looks like [Char]
- The type resembles generic lists in Java such as List<T> (instantiated to List<Character>), but that's where the similarities end.
- Lisp-style construction (Horribly inefficient for, e.g., binary data? Yes. More on this later.) Illustration in JSON:

```
{  
  'value': 5,  
  'next': {  
    'value': 4,  
    'next': {  
      value: 3,  
      'next': null  
    }  
  }  
}
```

## Lists (2/2)

- ▶ List (continued...)

- ▶ Literal notation: `[1,2,3]`
- ▶ Syntactic sugar for: `1:2:3:[]`, where `:` prepends a single element to a list, and `[]` is an empty list:

```
Prelude> 1:[2,3]  
[1,2,3]
```

- ▶ List items must all be of the same type. Heterogenous lists are possible using advanced language features, but they are usually an anti-pattern. When have you ever been happy to see `List<Object>` in Java?

## Operations on Lists (1/4)

- Take first element with head:

```
Prelude> head [1,2,3]
```

```
1
```

```
Prelude> head []
```

```
*** Exception: Prelude.head: empty list
```

- Take  $n$ th element using the index (!!) operator:

```
Prelude> [10, 11, 12] !! 0
```

```
10
```

```
Prelude> [10, 11, 12] !! 2
```

```
12
```

```
Prelude> [10, 11, 12] !! 5
```

```
*** Exception: Prelude.!!: index too large
```

## Operations on Lists (2/4)

- Check for the presence of a value using `elem`:

```
Prelude> elem 99 [42, 52, 62]
```

```
False
```

```
Prelude> elem 42 [42, 52, 62]
```

```
True
```

- Reverse a list with `reverse`:

```
Prelude> reverse [1,2,3,4]
```

```
[4,3,2,1]
```

## Operations on Lists (3/4)

- Add two lists together with ++:

```
Prelude> [1,2] ++ [3,4]
```

```
[1,2,3,4]
```

```
Prelude> "type" ++ "writer"
```

```
"typewriter"
```

- Grab the first  $n$  elements with take:

```
Prelude> take 2 [1,2,3,4,5]
```

```
[1,2]
```

## Operations on Lists (4/4)

- ▶ Mapping a function across a list with map:

```
Prelude> let addFive x = x + 5
```

```
Prelude> map addFive [1,2,3]  
[6,7,8]
```

- ▶ Reducing a list with foldl:

```
Prelude> foldl (\acc elem -> acc + elem) 0 [1,2,3]  
6
```



# Tuples (1/2)

## ► Tuple

- Store several values into a single value. They can be of any type, but you must know how many values there will be.
- For example,

```
Prelude> let point = (3,4)
```

```
Prelude> fst point
```

```
3
```

```
Prelude> snd point
```

```
4
```

```
Prelude> let point3d = (3,4,5)
```

```
Prelude> let location = ("Germany", (51.1657, 10.4515))
```

```
Prelude> fst location
```

```
"Germany"
```

```
Prelude> snd location
```

```
(51.1657, 10.4515)
```

## Tuples (2/2)

- ▶ Tuple (continued...)

- ▶ An associative list is [(a, b)], for example, one of type [(String, Int)]:

```
Prelude> let eatingContestScores =  
          [("Jimmy", 5), ("Sally", 8)]
```

```
Prelude> lookup "Jimmy" eatingContestScores  
Just 5
```

- ▶ Tuples are of different types when they vary in size or constituent types:

```
Prelude> (5, 5) :: (Int, Int)
```

```
Prelude> (5, 5, 5) :: (Int, Int, Int)
```

```
Prelude> (5, "Five") :: (Int, String)
```

## Reading Types (1/5)

`::` can generally be read as “has type”, and it is a more of a declaration than an observation.

```
-- 5 ``has type'' Int
```

```
Prelude> 5 :: Int
```

```
-- 5 ``has type'' Integer
```

```
Prelude> 5 :: Integer
```

```
-- 5 ``has type'' String...uh oh, we can't do that!
```

```
Prelude> 5 :: String
```

```
<interactive>:1:1: error:
```

- No instance for (Num String) arising from the literal '5'
- In the expression: `5 :: String`  
In an equation for 'it': `it = 5 :: String`

## Reading Types (2/5)

The *function arrow* or *function operator* `->` denotes a function that takes an argument of the type on the left and returns a value of the type on the right.

```
-- addFive ``has type`` function that takes
--    an Int and returns an Int
addFive :: Int -> Int
addFive x = x + 5

-- mkAdder ``has type`` function that takes
--    an Int and returns a (function that takes
--    an Int and returns an Int)
mkAdder :: Int -> (Int -> Int)
mkAdder x = (\y -> y + x)
addFive = (mkAdder 5)
```

## Reading Types (3/5)

More practice reading types.

```
import Data.Char (toLower)
-- toLower ``has type`` function that takes a Char
--    and returns a Char
toLower :: Char -> Char

-- lowerString ``has type`` function that takes
--    a list of characters and returns a list of
--    characters
lowerString :: [Char] -> [Char]
lowerString = map toLower

Prelude> map toLower "ABCDE"
"abcde"
Prelude> lowerString "ABCDE"
"abcde"
```

## Reading Types (4/5)

You can use `:t` in GHCi to ask about an expression's type.

```
Prelude> :t "Hello, World!"  
"Hello, World!" :: [Char]  
-- String and [Char] are the same type
```

```
Prelude> let addFive x = x + 5  
Prelude> :t addFive  
addFive :: Num a => a -> a  
Prelude> addFive 1.2  
6.2
```

## Reading Types (5/5)

Locking down the type:

```
Prelude> :t (addFive :: Int -> Int)
(addFive :: Int -> Int) :: Int -> Int
```

```
Prelude> (addFive :: Int -> Int) 1.2 -- BOOM!
```

```
<interactive>:3:25: error:
```

- No instance for (Fractional Int) arising from the literal '1.2'
  - In the first argument of  
'addFive :: Int -> Int', namely '1.2'
- In the expression:
- ```
(addFive :: Int -> Int) 1.2
```
- In an equation for 'it':
- ```
it = (addFive :: Int -> Int) 1.2
```

## Currying (1/3)

Currying is the process of transforming a function that takes multiple arguments into a function that takes just a single argument and returns another function if any arguments are still needed.

*-- Uncurried*

```
add (x, y) = x + y
```

```
add (5, 2)
```

Curried forms are preferred because they allow for partial application:

*-- Curried*

```
add x y = x + y
```

```
addFive = add 5
```

```
addFive 2
```



## Currying (2/3)

Curried 1, 2, and 3 are equivalent.

*-- Uncurried*

`add :: (Int, Int) -> Int`

`add = \(x, y) -> x + y`

*-- Curried 1*

`add :: Int -> Int -> Int`

`add x y = x + y`

*-- Curried 2*

`add :: Int -> (Int -> Int)`

`add x = (\y -> x + y)`

*-- Curried 3*

`add :: (Int -> (Int -> Int))`

`add = (\x -> (\y -> x + y))`

## Currying (3/3)

The arrow operator is right-associative. Each value we apply returns a value conforming to “the rest” of the type signature.

$a \rightarrow a \rightarrow a \rightarrow a$

$a \rightarrow a \rightarrow (a \rightarrow a)$

$a \rightarrow (a \rightarrow (a \rightarrow a))$

Function application is left-associative due to currying. It “cancels out” because we apply values one at a time.

$f\ a\ b\ c$

$(f\ a)\ b\ c$

$((f\ a)\ b)\ c$

## Reading the Type of map (1/3)

```
Prelude> import Data.Char (toLower)
```

```
Prelude Data.Char> :t map  
map :: (a -> b) -> [a] -> [b]
```

```
-- `a' and `b' are known as type variables,  
-- because they can represent any type. They  
-- are uncapitalized to distinguish them  
-- from specific types such as `Int'.
```

```
Prelude Data.Char> :t toLower  
toLower :: Char -> Char
```

```
Prelude Data.Char> :t map toLower  
map toLower :: [Char] -> [Char]
```

## Reading the Type of map (2/3)

Hindley–Milner type inference!

1. Pass toLower to map.

```
map :: (a -> b) -> [a] -> [b]
toLower :: (Char -> Char)
```

2. See if there could be a fit.

```
(a -> b) -> [a] -> [b]
(Char -> Char)
```

3. There it is! Substitute let a = Char, b = Char in

```
(Char -> Char) -> [Char] -> [Char]
(Char -> Char)
```

4. “Subtract,” and then our result is

```
[Char] -> [Char]
map toLower :: [Char] -> [Char]
```

## Reading the Type of map (3/3)

We've just done partial function application!

```
map :: (a -> b) -> [a] -> [b]
```

```
lowerString = (map toLower :: [Char] -> [Char])
```

lowerString is the resulting function of a partially-applied map.

map has a and b so we can change the type of the list.

```
Prelude> :t even
```

```
even :: Int -> Bool
```

```
Prelude> :t [1,2,3]
```

```
[1,2,3] :: [Int]
```

```
Prelude> map even [1,2,3]
```

```
[False,True,False]
```

```
Prelude> :t [False,True,False]
```

```
[False,True,False] :: [Bool]
```

## Reading Typeclasses (1/5)

Num is a typeclass that defines some common numeric operations. Think interfaces in Java or Go.

```
Prelude> let addFive x = x + 5
```

```
Prelude> :t addFive
```

```
addFive :: Num a => a -> a
```

```
Prelude> addFive 1.2
```

```
6.2
```

## Reading Typeclasses (2/5)

List function examples with type variables:

```
fst :: (a,b) -> a
```

```
head :: [a] -> a
```

```
take :: Int -> [a] -> [a]
```

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

```
id :: a -> a
```

## Reading Typeclasses (3/5)

Abbreviated examples:

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

```
-- Num - Numeric Types
```

```
(+) :: Num a => a -> a -> a
```

```
-- Eq - Equality Types
```

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

```
-- Ord - Ordered Types
```

```
(<) :: Ord a => a -> a -> Bool
```



## Reading Typeclasses (4/5)

Nearly-complete definition from the GHC source code:

```
-- | Basic numeric class.
class Num a where
    (+), (-), (*)      :: a -> a -> a
    -- | Unary negation.
    negate             :: a -> a
    -- | Absolute value.
    abs                :: a -> a
    -- | Sign of a number.
    signum             :: a -> a
    -- | Conversion from an 'Integer'.
    fromInteger        :: Integer -> a

    x - y              = x + negate y
    negate x           = 0 - x
```

## Reading Typeclasses (5/5)

Typeclasses are useful because they allow us to define functions that can work across a variety of types sharing common properties.

```
class Show a where
  show      :: a    -> String
  -- ...more function requirements
```

```
instance Show Int where
  show = showSignedInt'  -- Some low-level function
```

```
prefixedShow :: (Show a) => String -> a -> String
prefixedShow prefix x = prefix ++ ": " ++ show x
```

```
prefixedShow "MyValue" 42      -- "MyValue: 42"
prefixedShow "MyPet"    "cat"  -- "MyPet: \"cat\""
prefixedShow "MyPoint" (4,5)   -- "MyPoint: (4,5)"
```

## Infix Functions (1/3)

Functions can be put in between arguments like this:

```
Prelude> 5 + 5  
10
```

They are of type  $a \rightarrow b \rightarrow c$ .

```
Prelude> :t (+)  
(+) :: Num a => a -> a -> a
```

When surrounded by parentheses, they are considered in *prefix notation*, meaning that they are called like any other non-infix functions.

```
Prelude> (+) 5 5  
10
```

## Infix Functions (2/3)

Likewise, non-infix functions matching type  $a \rightarrow b \rightarrow c$  may be used as infix functions.

```
Prelude> mod 12 5
```

```
2
```

```
Prelude> 12 `mod` 5
```

```
2
```

This is all just syntactic sugar.

## Infix Functions (3/3)

In other languages such as C, infix functions are built in. In Haskell, we can define our own.

```
infixr 0 ~ -- right associative  
--  $a \sim b \sim c$  becomes  $a \sim (b \sim c)$ 
```

```
infixl 0 ~ -- left associative  
--  $a \sim b \sim c$  becomes  $(a \sim b) \sim c$ 
```

```
infix 0 ~ -- non-associative  
--  $a \sim b \sim c$  becomes  $(a \sim b) \sim c$  OR  $a \sim (b \sim c)$ 
```

They have associativity and precedence (0 the weakest, 9 the strongest).

# Anonymous Functions

Anonymous functions help us not think about names. They can be defined as follows:

```
add :: Int -> Int -> Int
add = \x y -> x + y
```

```
addFive :: Int -> Int
addFive = \x -> add x 5
```

They are expressions, and can be used as inputs to functions that take functions as arguments or assigned to variables.

```
Prelude> map (\x -> x + 5) [1,2,3]
[6,7,8]
Prelude> let double = (\x -> x * 2) in double 5
10
```

# Evaluation Order

When in doubt, use parentheses.

```
Prelude> 5 - 3 * 2
```

```
-1
```

```
Prelude> (5 - 3) * 2
```

```
4
```

You can then look up the associativity and precedence per operator to reason about what the execution would be.

## Symbols Related to Evaluation Order

You will encounter two symbols that help dictate order of evaluation, \$ is known as *application*, and . is known as *function composition*.

- ▶ (\$) calls the function which is its left-hand argument on the value which is its right-hand argument.

```
map addFive (map addFive [1,2,3])  
-- Same as  
map addFive $ map addFive [1,2,3]
```

- ▶ (.) composes the function which is its left-hand argument on the function which is its right-hand argument.

```
addFive (double 5)  
-- Same as  
(addFive . double) 5
```

(Source: ellisbben, StackOverflow)



## PITFALL: Mismatched Arguments (1/3)

Watch out for what gets passed in as arguments!

```
Prelude> map addFive map addFive [1,2,3]
```

```
<interactive>:3:1: error:
```

- Couldn't match expected **type** `'(Integer -> Integer) -> [Integer] -> t'`  
with actual **type** `'[Integer]'`
- The function `'map'` is applied to four arguments, but its **type** `'(Integer -> Integer) -> [Integer] -> [Integer]'` has only two

In the expression:

```
map addFive map addFive [1, 2, 3]
```

In an equation for `'it'`:

```
it = map addFive map addFive [1, 2, 3]
```

- Relevant bindings include  
`it :: t` (bound at `<interactive>:3:1`)

## PITFALL: Mismatch Arguments (2/3)

```
-- ...continued
```

```
<interactive>:3:13: error:
```

- Couldn't match expected **type**

    '**[Integer]**'

with actual **type**

    '**(a0 -> b0) -> [a0] -> [b0]**'

- Probable cause: 'map' is applied to too few arguments

In the second argument of 'map', namely 'map'

In the expression:

```
map addFive map addFive [1, 2, 3]
```

In an equation for 'it':

```
it = map addFive map addFive [1, 2, 3]
```

## PITFALL: Mismatched Arguments (3/3)

```
map addFive map addFive [1,2,3]
```

```
|---^-----^---^-----^  
      1         2   3         4
```

Instead, do:

```
map addFive (map addFive [1,2,3])
```

```
|---^-----^  
      1         2
```

Or

```
map addFive $ map addFive [1,2,3]
```

```
|---^-----^  
      1         2
```

# Pattern Matching

Pattern matching is an extremely useful technique that is a staple in functional programming languages. It allows us to “search” for the data we want from an expression. We will cover:

- ▶ Pattern matching constants, lists, and tuples
- ▶ Using pattern matching in
  - ▶ Function Definitions (we've already been using them!)
  - ▶ Guards
  - ▶ where
  - ▶ let
  - ▶ case...of

**Pattern Matching in Haskell is top to bottom, left to right.**

## Pattern Matching: In JavaScript

Here is an example of code with and without destructuring syntax in ES6:

```
const user = {name: "Jones", age: 50,  
              email: "jones@jones.com"}  
const name = user.name  
const age  = user.age  
const email = user.email
```

With pattern matching:

```
const user = {name: "Jones", age: 50,  
              email: "jones@jones.com"}  
const {name, age, email} = user
```

While destructuring isn't pattern matching, it has a similar intent, and there are even proposals to add true pattern matching to the language (see references).

## Pattern Matching: Matching Constants (1/2)

Somewhere in test.hs:

```
guess :: Int -> Bool
```

```
guess 5 = True
```

```
(guess 5)  -- True
```

```
(guess 10) -- BOOM!
```

```
-- *** Exception: Non-exhaustive
```

```
--           patterns in function guess
```

GHC can warn us about this before it happens!

```
$ ghc -Wall test.hs
```

```
test.hs:83:1: warning: [-Wincomplete-patterns]
```

```
Pattern match(es) are non-exhaustive
```

```
In an equation for ‘guess’:
```

```
Patterns not matched:
```

```
  p where p is not one of {5}
```

## Pattern Matching: Matching Constants (2/2)

Fixed, in `test.hs`:

```
guess :: Int -> Bool
guess 5 = True
guess _ = False
```

- ▶ Underscore (`_`) means “throw this value out, we’re not going to use it.”
- ▶ Patterns are tried sequentially until there’s a match.
- ▶ If no match could be found, then an exception is thrown. Therefore, it’s good practice to have catch-all patterns at the end.

## Pattern Matching: Matching Lists

Lists can be matched in a way that grants you their first few elements, which is very useful in recursive functions. The form is `(x:xs)`.

```
map :: (a -> b) -> [a] -> [b]
map f (x:xs) = (f x):(map f xs)
map f []     = []    -- Cover all the cases
```

If necessary, we can also specify a handle for the whole list:

```
addListLengthToFirstElement :: [Int] -> Int
addListLengthToFirstElement lst@(x:_) = length lst + x
addListLengthToFirstElement [] = error "empty list"
```



## PITFALL: Matching Lists

Be wary of these syntax fails that can lead to confusing compiler errors:

```
sum [x:xs] = x + sum xs  -- NO
```

```
sum x:xs = x + sum xs    -- NO
```

```
sum (x:xs) = x + sum xs  -- ok
```

## Pattern Matching: Matching Tuples

We can pull the values right out of tuples using pattern matching.

```
dist :: (Double, Double) -> (Double, Double) -> Double
dist (x, y) (x', y') =
    sqrt $ (x - x') ** 2 + (y - y') ** 2
Prelude> dist (0.0, 0.0) (1.0, 1.0)
1.4142135623730951
```

Even with nesting:

```
showIdAndFullName :: (Int, (String, String)) -> String
showIdAndFullName (id', (firstName, lastName)) =
    show id' ++ ": " ++ firstName ++ " " ++ lastName
Prelude> showIdAndFullName (5, ("Rebecca", "Jones"))
"5: Rebecca Jones"
```

## Pattern Not-Matching: Guards

Guards are ways to test whether some properties are true or false. They don't actually do pattern matching. However, they can be used with the `where` clause to do pattern matching. They are evaluated sequentially.

```
scovilleComment :: Int -> String
scovilleComment score
  | score < 0 = "Huh?"
  | score <= 300 = "What spice?"
  | score <= 700 = "Eh...mild."      -- Frank's
  | score <= 3500 = "A bit of kick!" -- Tabasco
  | score <= 10000 = "Mmm, spicy."   -- Orange Kush
  | score <= 1500000 = "WOW THAT'S HOT" -- Pyro Diablo
  | otherwise = "OMG"
```

# PITFALL: Guards

Do not put an equal sign after the function name and variables:

```
-- NO
guess x =
  | x == 5      = "You got it!"
  | otherwise = "Wrong!"
```

```
-- ok
guess x
  | x == 5      = "You got it!"
  | otherwise = "Wrong!"
```

```
-- ok
guess x | x == 5      = "You got it!"
        | otherwise = "Wrong!"
```

## Pattern Matching: where

where can be used to help things read nicer and avoid repetition.

```
totalCompensation :: Int -> Int -> Int -> Int -> Int
totalCompensation base yearsEmployed rank performance =
    base + bonus
    where rankMultiplier = (yearsEmployed / rank) `mod` 2
          bonus = rankMultiplier * performance * 10000
```

We can see right away that totalCompensation simply returns base + bonus, but if we want more detail, then we can dig in to the where clause. They can also be used to pattern match:

```
showPoint :: (Double, Double) -> String
showPoint p = "In " ++ show p ++ " " ++ x' ++ "," ++ y'
    where (x, y) = p
          x' = "x=" ++ show x
          y' = "y=" ++ show y
Prelude> showPoint (1,2)
"In (1,2) x=1, y=2"
```

## Pattern Matching: let

let can be used to pattern match in the form of:

```
let <bindings> in <expression>
```

let itself is an expression and therefore produces a value, which is an important difference from where.

```
quadratic :: Double -> Double -> Double -> Int -> Double
quadratic a b c sign =
    let sign'      = if sign >= 0 then 1.0 else -1.0
        rootTerm   = sqrt $ (b**2) - 4*a*c
        numerator   = (- b) + (sign') * rootTerm
        denominator = 2 * a
    in
        numerator / denominator
```

## Pattern Matching: `case...of`

Case expressions can be thought as guards that do pattern matching instead of property testing. Like `let`, they are expressions.

```
case expression of pattern -> result
                    pattern -> result
                    pattern -> result
                    ...
```

(Source: Learn You a Haskell)

```
showPointType :: (Double, Double) -> String
showPointType p = "Point Type: " ++ case p of
    (0, 0)      -> "Origin"
    (_, 0)      -> "Y-Intercept"
    (0, _)      -> "X-Intercept"
    otherwise   -> "Normal Point"

Prelude> showPointType (0, 0)
"Point Type: Origin"
```

if...then...else

if...then...else constructs in Haskell are better thought of as *if expressions* than *if statements*. They evaluate to a value depending on what comes after if. They cannot be used to pattern match.

```
addIfEven :: Int -> Int -> Int
```

```
addIfEven x y = x + if even y then y else 0
```



## PITFALL: Sometimes spacing matters

```
zeroIfEven :: Int -> Int
```

```
zeroIfEven x =  -- NO  
    if even x then  
        0  
    else  
        x
```

```
zeroIfEven x =  -- ok  
    if even x  
--    ^ Code Block 1  
        then 0  
        else x  
--    ^ Code Block 2
```

## Sum Types (1/7)

If we have three types of people: employees, managers, and clients, how do we cleanly represent them? Let's say we're using Go.

```
type PersonType string

const (
    EmployeePersonType PersonType = "Employee"
    ManagerPersonType  PersonType = "Manager"
    ClientPersonType   PersonType = "Client"
)
```

## Sum Types (2/7)

```
type PersonType string
```

The primary benefit is that the compiler gives us some type checks for PersonType.

```
func AmIOkay(pt PersonType) bool {  
    switch pt {  
        case ManagerPersonType:  
            fallthrough  
        case ClientPersonType:  
            return true  
        case EmployeePersonType:  
            return false  
        default:  
            panic(fmt.Sprintf(  
                "Unknown PersonType: %s", pt))  
    }  
}
```

## Sum Types (3/7)

```
type PersonType string
```

Disadvantages:

- ▶ Strings are expensive to pass around
- ▶ Strings are OVERKILL.  $c^{strlen}$  possibilities!
- ▶ Someone can just decide to invent a new type via cast:  
    PersonType("Daughter")
- ▶ No exhaustiveness checks

We could consider using enum-like `iota`, but the last 3 points would still exist.

## Sum Types (4/7)

```
type PersonType string
```

What if we forget one of the checks?

```
-         case EmployeePersonType:  
-             return false
```

Runtime error!

```
$ go run test.go
```

```
panic: Unknown PersonType: Employee
```

```
goroutine 1 [running]:  
panic(0x48c5c0, 0xc42000a320)
```

## Sum Types (5/7)

Enter sum types!

```
-- Person is a sum type with zero-argument value
-- constructors of Employee, Manager, and Client.
data PersonType = Employee | Manager | Client

greet :: PersonType -> String
greet pt = case pt of
    Employee -> "G'day, employee!"
    Manager -> "Hello, manager."

main :: IO ()
main = do
    putStrLn $ greet Client
```

## Sum Types (6/7)

Run it:

```
$ ghc -Wall test.hs
```

```
test.hs:33:12: warning: [-Wincomplete-patterns]
```

```
    Pattern match(es) are non-exhaustive
```

```
    In a case alternative: Patterns not matched: Client
```

```
$ ./test
```

```
test: test.hs:(33,12)-(35,32):
```

```
    Non-exhaustive patterns in case
```

## Sum Types (7/7)

Fixed:

```
greet :: PersonType -> String
greet pt = case pt of
    Employee -> "G'day, employee!"
    Manager -> "Hello, manager."
    Client -> "Hey!"
```

```
$ ghc -Wall test.hs
```

```
$ ./test
```

```
Hey!
```



## Product Types (1/2)

Sum types can be combined with product types to store additional data:

```
data ApptTime = NotScheduled | WalkIn | At UTCTime
--
--           ^
--           |
--           | This product type combines At
--           | and UTCTime to form a new type.
--           | They're "glued" together.
```

  

```
apptStatus :: ApptTime -> String
apptStatus apptTime = case apptTime of
    NotScheduled -> "Not currently scheduled."
    WalkIn -> "Walk in during business hours."
    At time -> "Arrive by " ++ show time
```

The data can be accessed using pattern matching, as seen with time.

## Product Types (2/2)

Running our latest example:

```
-- ...
```

```
main :: IO ()  
main = do  
    currentTime <- getCurrentTime  
    putStrLn $ apptStatus (At currentTime)
```

```
$ ghc -Wall test.hs
```

```
$ ./test
```

```
Arrive by 2017-05-08 10:04:09.234585398 UTC
```

# Algebraic Data Types

Sum and product types together make up the *algebraic data types* (ADTs) in Haskell. These are not to be confused with *abstract data types*. There are also *generic algebraic data types* (GADTs), but those are out of scope for this presentation.

```
-- Sum Types are combined with '|',  
--   behaving like an "OR" operator  
data MySumType = A | B  
-- Product Types are combined with ' ',  
--   behaving like an "AND" operator  
data MyProductType = C D
```

# Namespaces and Constructors (1/12)

Haskell has two namespaces:

- ▶ One for *values*.
- ▶ One for *types*.

Constructors:

- ▶ A data/value constructor is a “function” that takes 0 or more values and gives you back a new value.
- ▶ A type constructor is a “function” that takes 0 or more types and gives you back a new type.

(Source: kqr, StackOverflow)

## Namespaces and Constructors (2/12)

The data operator constructs a new type.

```
data Color = Red | Green | Blue
```

```
Red :: Color
```

```
Green :: Color
```

```
Blue :: Color
```

```
data Color = RGB Int Int Int
```

```
RGB :: Int -> Int -> Int -> Color
```

For all practical purposes you can just think of data constructors as constants belonging to a type.

# Namespaces and Constructors (3/12)

## Same-named constructors

```
data Person = Person String Int
--      ^           ^
--      |           |
--      |   Here we declared and defined a
--      |   value constructor named ``Person``.
--      |   To construct the value, we need a String value
--      |   followed by an Int value. It takes two value
--      |   arguments, so it is ``binary``.
--      |
--      |   Here we declared and defined a type constructor
--      |   named ``Person``. It takes no type arguments and is
--      |   therefore considered ``nullary``.
```

## Namespaces and Constructors (4/12)

```
data Point a = Point a a
--      ^           ^
--      |           |
--      |   Created a binary value constructor named
--      |   ``Point'' which takes two value arguments
--      |   of type a.
--      |
--      Created a type constructor named ``Point''
--      that takes one type argument such as Int or
--      Double. It is ``unary''.
```

```
Prelude> :t Point (3.0 :: Double) (4.0 :: Double)
--          ^ value constructor
Point (3.0 :: Double) (4.0 :: Double) :: Point Double
--          type constructor ^
```

# PITFALL: Constructor Naming

```
-- NO
data Person = String Int
--      ^           ^
--      |           |
--      |   Created a value constructor named
--      |   ``String''. Possible because it is
--      |   currently available in the value
--      |   namespace (although not in the type
--      |   namespace). Probably not what was intended.
--      |
--      Created a nullary type constructor ``Person''

-- ok
data Person = Person String Int
```



## Namespaces and Constructors (5/12)

Pattern matching is the primary way we access the data wrapped in value constructors.

```
data Person = Person String Int
showNameAndAge :: Person -> String
showNameAndAge (Person name age) =
    name ++ " age=" ++ show age
```

```
Prelude> showNameAndAge (Person "Fred" 2)
"Fred age=2"
```

## Namespace and Constructors (6/12)

Next example:

```
-- Cats have a certain number of lives
data Creature = Dog String | Cat String Int
showCreature :: Creature -> String
showCreature c = case c of
    (Dog name)      -> name
    (Cat name lives) -> name ++ " lives=" ++ show lives
```

```
Prelude> showCreature (Dog "Fido")
"Fido"
```

```
Prelude> showCreature (Cat "Mittens" 9)
"Mittens lives-left=9"
```

## Namespaces and Constructors (7/12)

Another pattern matching example for constructed values:

```
-- Function from the test-taking belt of New Jersey  
data Grade = A | B | C | D | F  
data TestResult = Complete Grade | Missed  
showTestResult :: TestResult -> String  
showTestResult Missed = "The test was missed."  
showTestResult (Complete g) = case g of  
    A          -> "Top score!"  
    B          -> "Horrible score, but you'll live!"  
    otherwise -> "Why even bother?"
```

```
Prelude> showTestResult (Complete A)  
"Top score!"
```

## Namespaces and Constructors (8/12)

Here is an example of a recursive type definition:

```
data List a = Nil | Cons a (List a)
```

- ▶ List is a unary type constructor, meaning it takes one type argument. This can be anything from Int, to Char, to [Char], etc.

```
List :: * -> *
```

- ▶ Nil is a nullary value constructor, taking no value arguments.

```
Nil :: List a
```

- ▶ Cons is a binary value constructor, taking two value arguments.

```
Cons :: a -> List a -> List a
```

# Namespaces and Constructors (9/12)

Here is an example of a recursive type definition:

```
data List a = Nil | Cons a (List a)
```

## Kinds

List can be viewed as an even higher-order function with the *kind* of  $* \rightarrow *$ , in the same fashion as  $\text{Int} \rightarrow \text{Int}$ .

*Kinds are higher than types.*

$*$  represents a type argument, and in this case, the *type variable*  $a$ , which contains a type.

# Namespaces and Constructors (10/12)

You can ask GHCi to print out some kinds:

```
-- Int is a nullary type constructor
```

```
Prelude> :k Int
```

```
Int :: *
```

```
-- Int is a unary type constructor
```

```
Prelude> :k []
```

```
[] :: * -> *
```

```
-- We feed [] :: (* -> *) an Int :: *
```

```
Prelude> :k [Int]
```

```
[Int] :: *
```

## PITFALL: Value Constructor Argument Mismatch

Value constructors only understand nullary type constructors (of kind `*`). Watch out for argument mismatches. In this case, the RHS `List` does not get passed the type argument `a`. Instead, `List` gets passed to `Cons` as its second argument and `a` gets passed as its third:

```
-- NO
data List a = Nil | Cons a List a
--           ^  ^      ^
--           1 2      3
```

Instead, wrap the type constructor and its type argument in parentheses.

```
-- ok
data List a = Nil | Cons a (List a)
--           ^  ^
--           1 2
```

# Namespaces and Constructors (11/12)

Let's re-examine our recursive type definition:

```
data List a = Nil | Cons a (List a)
```

Value constructors explained (again):

- ▶ Nil is a nullary value constructor: it takes no values to make a new constructed value. This is just like a previous example of three of them:

```
data = Employee | Manager | Client
```

- ▶ Cons is a binary value constructor. It takes a first argument of type a and a second argument of type List a. This is okay because List is of kind  $* \rightarrow *$ , so when it is passed the type variable a in the expression List a, the resulting kind is \*, just like Int.



## Namespaces and Constructors (12/12)

Our List in action:

```
(Cons 5 Nil)                :: Num a => List a
(Cons 5 (Cons 4 Nil))       :: Num a => List a
(Cons 5 (Cons 4 (Cons 3 Nil))) :: Num a => List a
```

```
car                :: List a -> a
car (Cons x _)     = x
car Nil            = error "car called on Nil List"
```

```
-- Compare to the GHC source code:
-- | Extract the first element of a list...
head                :: [a] -> a
head (x:_)          = x
head []              = badHead
```

## Records (1/2)

Records are a way to generate accessor functions for a constructed value.

Without records:

```
data Person = Person String Int deriving Show
Prelude> :t Person "Fred" 5
Person "Fred" 5 :: Person
Prelude> let (Person _ age) = (Person "Fred" 5) in age
5
```

With records:

```
data Person = Person { name :: String, age :: Int } deriving Show
Prelude> Person {name="Fred", age=5}
Person {name = "Fred", age = 5}
Prelude> age Person {name="Fred", age=5}
5
```

## Records (2/2)

Trouble in paradise. There is a serious namespace issue:

```
data Person = Person { name :: String, age :: Int }  
data Company = Company { name :: String, revenue :: Int }
```

When we compile:

```
$ ghc -Wall ./test.hs  
[1 of 1] Compiling Main                ( test.hs, test.o )
```

```
test.hs:93:26: error:  
    Multiple declarations of 'name'  
    Declared at: test.hs:92:24  
                test.hs:93:26
```

Google “haskell record problem” to learn more about the workarounds.

# Typeclasses Revisited (1/6)

- ▶ Typeclasses were created to express “ad-hoc polymorphism,” AKA, overloaded functions.
- ▶ They turned out to be nicer in many ways than those in Java or C++.
- ▶ I stole some great examples from Philip Wadler’s talk at Microsoft. He implemented typeclasses for Haskell and generics for Java.

## Typeclasses Revisited (2/6)

We want a generic function `max` so that we don't have to define it over and over again for different types. Let's write it naively in C++ using templates.

```
#include <iostream>

template<class T>
T max(T x, T y) {
    return x < y ? y : x;
}
```

## Typeclasses Revisited (3/6)

```
#include <iostream>
```

```
template<class T>  
T max(T x, T y) {  
    return x < y ? y : x;  
}
```

```
void main() {  
    // 2  
    std::cout << max<int>(1, 2) << std::endl; // 2  
    // 'b' (on most machines)  
    std::cout << max<char>('a', 'b')) << std::endl;  
    const char* a = "zzz"; const char* b = "aaa";  
    // ???  
    std::cout << max<const char*>(a, b)) << std::endl;  
    // It's "aaa"  
}
```

## Typeclasses Revisited (4/6)

Instead of C++ templates, let's try it using Haskell typeclasses.

```
class Ord a where  
    (<) :: a -> a -> Bool
```

```
instance Ord Int where  
    (<) = primitiveLessInt
```

```
instance Ord Char where  
    (<) = primitiveLessChar
```

```
max :: Ord a => a -> a -> a  
max x y = if x < y then y else x
```

## Typeclasses Revisited (5/6)

List of Char? No problem. Just add:

```
instance Ord String where
  []    < []           = False
  []    < blst         = True
  alst < []           = False
  (a:as) < (b:bs)
    | a < b           = True
    | b < a           = False
    | otherwise       = as < bs
```



## Typeclasses Revisited (6/6)

But we can go deeper. String is just [Char]. Char already implements Ord. What if we abstracted this?

```
-- Instead of this tautology
Ord Char => [Char]
-- Let's say
Ord a => [a]

instance Ord a => Ord [a] where
    [] < []           = False
    [] < y:ys         = True
    x:xs < []         = False
    x:xs < y:ys | x < y   = True
                | y < x   = False
                | otherwise = xs < ys
```

Now we can handle comparisons with any level of list nesting.

# The Maybe Type

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

```
Just (5 :: Int) :: Maybe Int
```

```
Nothing :: Maybe a
```

```
addMaybe :: Int -> Maybe Int -> Int
```

```
addMaybe (x (Just y)) = x + y
```

```
addMaybe (x Nothing) = x
```

```
addMaybe 5 (Just 3) -- 8
```

```
addMaybe 5 (Nothing) -- 5
```

# Functors Introduction (1/2)

Functors are essentially types that define `fmap`, as follows:

```
class Functor f where
    fmap :: (a -> b) -> f a -> f b
```

`fmap` takes a function that maps from `a` to `b`. It lifts the function so that it can operate on the *argument* of a value constructor `f`. That is, `fmap` gives us a function that can take `f a` and return `f b`!

## Functors Introduction (2/2)

Additionally, functors must also satisfy some rules known as the functor laws. Namely,

```
-- functor identity law
```

```
fmap id = id
```

```
-- functor composition law
```

```
fmap (f . g) = fmap f . fmap g
```

We don't have time to go over these in detail, but the motivation behind supporting these laws is that they're necessary to reduce complexity by defining specific but still “small” behavior.

# The Maybe Functor

The Maybe functor is defined as follows:

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

Usage example:

```
Prelude> fmap (+1) $ Just 5
Just 6
Prelude> fmap (+1) $ Nothing
Nothing
Prelude> fmap (\x -> x^x) $ Just 10
Just 10000000000
Prelude> fmap (\x -> x^x) $ Nothing
Nothing
```

## Interview Questions: Select Permutations (1/2)

```
-- Given a string and list of case-switching characters,
-- return all versions of the string that can be formed
-- by toggling the case-switching characters.
-- Inputs will be lowercase.
--
-- "test", ['s']      -> ["test", "teSt"]
-- "test", ['e', 's'] -> ["test", "teSt", "tEst", "tEst",
import Data.Char (toLower, toUpper)

data Token = Normal Char | Switching Char deriving Show

tokenize :: String -> [Char] -> [Token]
tokenize str lst = map toToken str
  where toToken x = if elem x lst
                      then Switching x
                      else Normal x
```

## Interview Questions: Select Permutations (2/2)

```
generate :: String -> [Char] -> [String]
generate s switchList = generate' reversedTokens []
  where reversedTokens = reverse $ tokenize s switchList

generate' :: [Token] -> [Char] -> [String]
generate' [] current = [current]
generate' (r:rs) current = case r of
  Switching (ch) -> (generate' rs $ (toLower ch):current)
                  ++ (generate' rs $ (toUpper ch):current)
  Normal      (ch) -> (generate' rs $ ch:current)

main :: IO ()
main = do
  putStrLn $ show $ generate "test" ['s', 'e', 't']
  -- Output:
  -- ["test","Test","tEst","TEst","teSt","TeSt","tEST",
  --  "TEST","tesT","TesT","tEsT","TEsT","teST","TeST",
  --  "tEST","TEST"]
```

## Interview Questions: Check Leading Ones (1/3)

```
-- Write a program that takes a byte array message
-- and integer count as inputs, and returns true if
-- there are at least that many leading ones in the
-- message, false otherwise.

-- Messages beginning with the two bytes 1 1 1 1 1 1 1 1
-- / 1 0 0 0 0 0 0 0 have nine leading ones.
--
-- Messages beginning with the byte 1 1 1 0 1 0 0 0
-- have three leading ones.
```

```
import Data.List
import Data.Bits
import Data.Word
import qualified Data.ByteString as BS
```



## Interview Questions: Check Leading Ones (2/3)

```
-- Create a list: [(255,8), (254,7), (252,6), (248,5), ...]
byteAList :: [(Word8, Int)]
byteAList = map makeMaskPair [0..7]
    where makeMaskPair x = ((shift 0xFF x) .&. 0xFF, 8-x)

-- Count how many leading ones the Word8 b has
countByteLeadingOnes :: Word8 -> Int
countByteLeadingOnes b = case find match byteAList of
    Nothing -> 0
    Just (_, y) -> y
    where match (x, _) = x .&. b == x
```

## Interview Questions: Check Leading Ones (3/3)

```
-- Determines if the bytestring bs has
-- at least n leading ones
hasLeadingOnes :: Int -> BS.ByteString -> Bool
hasLeadingOnes n bs = hasLeadingOnes' n bs 0

hasLeadingOnes' :: Int -> BS.ByteString -> Int -> Bool
hasLeadingOnes' n bs count = case BS.uncons bs of
    Nothing -> count >= n
    Just (b, bs') -> hasLeadingOnes' n bs' newCount
        where currentCount = countByteLeadingOnes b
              newCount = currentCount + newCount
```

## Where to go from here

- ▶ Haskell Book by Chris Allen <http://haskellbook.com/>
- ▶ Learn You a Haskell by Miran Lipovača,  
<http://learnyouahaskell.com>
- ▶ Gentle Introduction To Haskell, version 98 by Paul Hudak,  
John Peterson, and Joseph Fasel,  
<https://www.haskell.org/tutorial/>
- ▶ Real World Haskell by Bryan O'Sullivan,  
<http://book.realworldhaskell.org/>
- ▶ An emporium of resources on the Haskell wiki:  
[https://wiki.haskell.org/Learning\\_Haskell](https://wiki.haskell.org/Learning_Haskell)

## Video learning materials

- ▶ Functional Programming Fundamentals by Dr. Erik Meijer (Video Lectures) <https://channel9.msdn.com/Series/C9-Lectures-Erik-Meijer-Functional-Programming-Fundamentals>
- ▶ Haskell Course Taught by Philip Wadler, <https://www.youtube.com/watch?v=A0l2y5uW0mA&list=PLtRG9GLtNcHBv4cuh2w1cz5VsgY6adoc3>
- ▶ Adventure with Types in Haskell by Simon Peyton Jones, <https://www.youtube.com/watch?v=6C0vD8oynmI>
- ▶ Haskell is Not For Production and Other Tales by Katie Miller, <https://www.youtube.com/watch?v=m1T0510z078>

## Resources to learn about monads (1/2)

There are so many monad tutorials online, so I'll only list a few that I felt really helped me. Your learning style might be different, so please check out many of them! However, here's some ground advice that could really help your venture:

- ▶ Know how to define typeclasses
- ▶ Be comfortable with type and value constructors and how they interact with functions and typeclasses
- ▶ Learn functors, applicative functors, and monoids first. LYAH will go over these beforehand.
- ▶ If you just need to *use* (as opposed to construct) a monad, you don't need to fully understand it.

## Resources to learn about monads (2/2)

- ▶ Read “What is a Monad?” asked by Peter Mortensen on StackOverflow for a variety of good answers, <https://stackoverflow.com/questions/44965/what-is-a-monad>
- ▶ Start with Learn You a Haskell by Miran Lipovača, <http://learnyouahaskell.com>
- ▶ My personal favorite explanation after somewhat grasping the concept was You Could Have Invented Monads! by sigfpe. The practice problems were immensely helpful in consolidating my knowledge, <http://blog.sigfpe.com/2006/08/you-could-have-invented-monads-and.html>
- ▶ There are so many monad tutorials that Haskell Wiki gave it a whole section called “Monad tutorials timeline” with dozens of them from over the years, [https://wiki.haskell.org/Monad\\_tutorials\\_timeline](https://wiki.haskell.org/Monad_tutorials_timeline)

## References (1/2)

These information sources were heavily leaned on for amazing examples, explanations, and more found in these slides.

- ▶ Learn You a Haskell, Miran Lipovača,  
<http://learnyouahaskell.com>
- ▶ Glasgow Haskell Compiler, <https://www.haskell.org/ghc/>
- ▶ “Haskell Type vs Data Constructor”, kqr on StackOverflow,  
<https://stackoverflow.com/questions/18204308/haskell-type-vs-data-constructor>
- ▶ “Haskell function composition (.) and function application (\$) idioms: correct use”, ellisbben on StackOverflow,  
<https://stackoverflow.com/questions/3030675/haskell-function-composition-and-function-application->

## References (2/2)

(continued)

- ▶ “Faith, Evolution, and Programming Languages: from Haskell to Java”, Philip Wadler,  
<https://www.youtube.com/watch?v=NZeDRs6snm0>
- ▶ “How to Learn Haskell in Less Than 5 Years”, Chris Allen,  
<https://www.youtube.com/watch?v=Bg9ccYzMbxc>
- ▶ “Gentle Introduction To Haskell, version 98”, Paul Hudak, John Peterson, Joseph Fasel,  
<https://www.haskell.org/tutorial/>
- ▶ “Algebraic data type”, Haskell Wiki,  
[https://wiki.haskell.org/Algebraic\\_data\\_type](https://wiki.haskell.org/Algebraic_data_type)
- ▶ ECMAScript Pattern Matching Proposal,  
<https://github.com/tc39/proposal-pattern-matching>



EOF