Fundamentals of Haskell Programming

Alpheus Madsen

Pyrofex Corporation

OpenWest Conference
Thursday, April 11, 2019

Outline

What is Haskell?

What Is Haskell?

What is Haskell?

- Purely Functional

What Is Haskell?

What is Haskell?

- Purely Functional
- Strongly Typed (Hindley-Milner Type Inference)

What Is Haskell?

What is Haskell?

- Purely Functional
- Strongly Typed (Hindley-Milner Type Inference)
- Lazy

Outline

Basic Data Types

- Char

Basic Data Types

- Char
- Int and Integer

Basic Data Types

- Char
- Int and Integer
- Float and Double

Basic Data Types

- Char
- Int and Integer
- Float and Double
- List and String

Basic Data Types

- Char
- Int and Integer
- Float and Double
- List and String
- Tuple

Basic Data Types

- Char
- Int and Integer
- Float and Double
- List and String
- Tuple
- Types classes, like Maybe and Either

Basic Data Types: Lists

Basic Data Types: Lists

```haskell
-- Haskell lists
infiniteOddsList = [1,3..]
firstTenOdds = take 10 infiniteOddsList
```

Basic Data Types: Lists

```haskell
-- Haskell lists
infiniteOddsList = [1,3..]
firstTenOdds = take 10 infiniteOddsList

-- List Comprehensions
infiniteEvensList = [x | x <- [1..], even x]
```

Basic Data Types: Lists

```haskell
-- Haskell lists
infiniteOddsList = [1,3..]
firstTenOdds = take 10 infiniteOddsList

-- List Comprehensions
infiniteEvensList = [x | x <- [1..], even x]
```

Mathematical Notation:
$\{x | x \in \mathbb{N} \text{ and } x \text{ is even}\}$

Basic Data Types: Lists

```haskell
-- Haskell lists
infiniteOddsList = [1,3..]
firstTenOdds = take 10 infiniteOddsList

-- List Comprehensions
infiniteEvensList = [x | x <- [1..], even x]
```

Mathematical Notation:
$\{x | x \in \mathbb{N} \text{ and } x \text{ is even}\}$

```python
# Python List Comprehension
pythonEvens = [x for x in range(1, 1000) if x % 2 == 0]
```

Basic Data Types: Tuples

Basic Data Types: Tuples

```haskell
-- Haskell tuples
(10, "test", ["hello", "world"])
```

Basic Data Types: Tuples

```haskell
-- Haskell tuples
(10, "test", ["hello", "world"])

fst (10, "ten")
-- 10
snd (20, "twenty")
-- "twenty"
```

Basic Data Types: Tuples

```
-- Haskell tuples
(10, "test", ["hello", "world"])

fst (10, "ten")
-- 10
snd (20, "twenty")
-- "twenty"

fst (10, "ten", "dix")
-- Error:  fst :: (a, b) -> a
fst3 (10, "ten" "dix")
-- 10
thd3 (10, "ten", "dix")
-- "dix"
```

Basic Data Types: Maybe and Either

Basic Data Types: Maybe and Either

```
type Maybe = Nothing | Just a
```

Basic Data Types: Maybe and Either

```haskell
type Maybe = Nothing | Just a

-- example:
value = lookup 10 myIntMap
```

Basic Data Types: Maybe and Either

```haskell
type Maybe = Nothing | Just a

-- example:
value = lookup 10 myIntMap

-- If myIntMap = fromList[(10, "ten"), ...] then
--     value ==> Just "ten"
-- else
--     value ==> Nothing
```

Basic Data Types: Maybe and Either

```haskell
type Maybe = Nothing | Just a

-- example:
value = lookup 10 myIntMap

-- If myIntMap = fromList[(10, "ten"), ...] then
--     value ==> Just "ten"
-- else
--     value ==> Nothing

type Either a b = Left a | Right b
-- Error example:    Left "Disk out of memory"
-- Success example:  Right handle
```

Outline

The Fun in Functional

- Everything is Static: No Side Effects

The Fun in Functional

- Everything is Static: No Side Effects
- if/then/else is a Statement

The Fun in Functional

- Everything is Static: No Side Effects
- if/then/else is a Statement

```
flightStatus = if height > 30000
                 then "We're above the clouds!"
                 else "We can see the ground!"
```

- Recursion instead of loops

```
loopFactorial :: Integer -> Integer
loopFactorial n = factAccum n 1
  where
    factAccum 0 acc = acc
    factAccum 1 acc = acc
    factAccum k acc = factAccum (k - 1) (k * acc)
```

The Fun in Functional

- First Class Functions

- First Class Functions

```
map maximum [[1,2,3], [4,5,6], [1,5,7]]
```

The Fun in Functional

- First Class Functions

```
map maximum [[1,2,3], [4,5,6], [1,5,7]]

filter even [1..100]
```

- First Class Functions

```haskell
map maximum [[1,2,3], [4,5,6], [1,5,7]]

filter even [1..100]

-- "lambda" \ creates anonymous functions
filter (\x -> x < 50) [1..100]
```

The Fun in Functional

- First Class Functions

```haskell
map maximum [[1,2,3], [4,5,6], [1,5,7]]

filter even [1..100]

-- "lambda" \ creates anonymous functions
filter (\x -> x < 50) [1..100]

foldl (+) 0 [1..100] -- sum: 5050
foldl (*) 1 [1..100] -- 100!
```

- First Class Functions

```
map maximum [[1,2,3], [4,5,6], [1,5,7]]

filter even [1..100]

-- "lambda" \ creates anonymous functions
filter (\x -> x < 50) [1..100]

foldl (+) 0 [1..100] -- sum: 5050
foldl (*) 1 [1..100] -- 100!

-- Type signature for foldl
foldl :: (a -> a -> a) -> a -> [a] -> a
```

The Fun in Functional

- Function composition

- Function composition

```haskell
-- Using lambda
map (\xs -> negate (sum (tail xs))) [[1,2,3], [6,5,4]]
```

- Function composition

```haskell
-- Using lambda
map (\xs -> negate (sum (tail xs))) [[1,2,3], [6,5,4]]

-- Using composition
map (negate . sum . tail) [[1,2,3], [6,5,4]]
```

The Fun in Functional

- Function composition

```haskell
-- Using lambda
map (\xs -> negate (sum (tail xs))) [[1,2,3], [6,5,4]]

-- Using composition
map (negate . sum . tail) [[1,2,3], [6,5,4]]
```

- Currying

The Fun in Functional

- Function composition

```haskell
-- Using lambda
map (\xs -> negate (sum (tail xs))) [[1,2,3], [6,5,4]]
```

```haskell
-- Using composition
map (negate . sum . tail) [[1,2,3], [6,5,4]]
```

- Currying

```haskell
map (5+) [1, 2, 3, 4, 5, 6]
```

The Fun in Functional

- Function composition

```haskell
-- Using lambda
map (\xs -> negate (sum (tail xs))) [[1,2,3], [6,5,4]]

-- Using composition
map (negate . sum . tail) [[1,2,3], [6,5,4]]
```

- Currying

```haskell
map (5+) [1, 2, 3, 4, 5, 6]

-- (+) :: Int -> Int -> Int
-- (5+) :: Int -> Int
```

The Fun in Functional

- Function composition

```
-- Using lambda
map (\xs -> negate (sum (tail xs))) [[1,2,3], [6,5,4]]

-- Using composition
map (negate . sum . tail) [[1,2,3], [6,5,4]]
```

- Currying

```
map (5+) [1, 2, 3, 4, 5, 6]

-- (+) :: Int -> Int -> Int
-- (5+) :: Int -> Int
```

- Point Free Style

The Fun in Functional

- Function composition

```haskell
-- Using lambda
map (\xs -> negate (sum (tail xs))) [[1,2,3], [6,5,4]]
```

```haskell
-- Using composition
map (negate . sum . tail) [[1,2,3], [6,5,4]]
```

- Currying

```haskell
map (5+) [1, 2, 3, 4, 5, 6]
```

```haskell
-- (+) :: Int -> Int -> Int
-- (5+) :: Int -> Int
```

- Point Free Style

```haskell
pointed var = tan (logBase 3 (abs (sin (min 30 var))))
```

The Fun in Functional

- Function composition

```haskell
-- Using lambda
map (\xs -> negate (sum (tail xs))) [[1,2,3], [6,5,4]]

-- Using composition
map (negate . sum . tail) [[1,2,3], [6,5,4]]
```

- Currying

```haskell
map (5+) [1, 2, 3, 4, 5, 6]

-- (+) :: Int -> Int -> Int
-- (5+) :: Int -> Int
```

- Point Free Style

```haskell
pointed var = tan (logBase 3 (abs (sin (min 30 var))))

pointFree = tan . logBase 3 . abs . sin . min 30
```

Outline

Pattern Matching: whole@(element:remainder)

Pattern Matching

Pattern Matching: whole@(element:remainder)

```
parseMyString :: String -> String
parseMyString "" = "String is empty, ha!"
parseMyString whole@(x:xs) = whole ++ " starts with " ++
   [x] ++ " and ends with " ++ xs
```

Pattern Matching: whole@(element:remainder)

```haskell
parseMyString :: String -> String
parseMyString "" = "String is empty, ha!"
parseMyString whole@(x:xs) = whole ++ " starts with " ++
    [x] ++ " and ends with " ++ xs
```

```haskell
extractMaybeInt :: Maybe Int -> Int
extractMaybeInt Nothing = 0
extractMaybeInt (Just x) = x
```

Pattern Matching

Pattern Matching: whole@(element:remainder)

```haskell
parseMyString :: String -> String
parseMyString "" = "String is empty, ha!"
parseMyString whole@(x:xs) = whole ++ " starts with " ++
    [x] ++ " and ends with " ++ xs

extractMaybeInt :: Maybe Int -> Int
extractMaybeInt Nothing = 0
extractMaybeInt (Just x) = x

translation = let (a, b, c) = (10, "ten", "dix")
              in "French for " ++ b ++ " is " ++ c
```

Pattern Matching: Guards

Pattern Matching: Guards

```
weatherReport :: Integer -> String
weatherReport degF
  | degF <  0 = "Holy cow it's cold!  Stay inside!"
  | degF < 30 = "Watch out for ice!"
  | degF < 60 = "Bundle up!"
  | otherwise = "T-shirt and shorts!"
```

Pattern Matching

Helper Definitions: where

Helper Definitions: where

```
eggSize :: Float -> Float -> String
eggSize height radius
  | volume < hummingbird = "Just humming a bird along."
  | volume < ostrich     = "What are you...chicken?"
  | otherwise            = "Talk about being ostrich-sized
  where
    volume = 2 * pi * height * radius ^ 2
    hummingbird = 1000
    ostrich     = 5000
```

Helper Definitions: let … in

Helper Definitions: let ... in

```
boundingSurface :: Float -> Float -> Float
boundingSurface height radius =
  let
    sideArea = 2 * pi * radius * height
    topArea = pi * radius ^ 2
  in
    sideArea + (2 * topArea)
```

Helper Definitions: let ... in

```haskell
boundingSurface :: Float -> Float -> Float
boundingSurface height radius =
  let
    sideArea = 2 * pi * radius * height
    topArea = pi * radius ^ 2
  in
    sideArea + (2 * topArea)

-- let is a statement, much like if/then/else
myCubes = [let cube x = x ^ 3 in (cube 1, cube 3, cube 7)]
```

Helper Definitions: case … of

Helper Definitions: case … of

```
factorial :: Integer -> Integer
factorial n = case n of
  0 -> 1
  1 -> 1
  n -> n * (factorial (n - 1))
```

Outline

Haskell's Type System

- Everything has a type.

Haskell's Type System

- Everything has a type.

```
ghci> :t True
True :: Bool
```

Haskell's Type System

- Everything has a type.

```
ghci> :t True
True :: Bool

ghci> :t "Hello, world"
"Hello, world" :: [Char]
```

Haskell's Type System

- Everything has a type.

```
ghci> :t True
True :: Bool

ghci> :t "Hello, world"
"Hello, world" :: [Char]

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

Haskell's Type System

- Everything has a type.

```
ghci> :t True
True :: Bool
```

```
ghci> :t "Hello, world"
"Hello, world" :: [Char]
```

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

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

Haskell's Type System

- Everything has a type.

```
ghci> :t True
True :: Bool

ghci> :t "Hello, world"
"Hello, world" :: [Char]

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

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

Common type classes: Eq, Ord, Read, Show, Num, Integer, Float

Haskell's Type System

- Type variables.

Haskell's Type System

- Type variables.

```
ghci> :t maximum
maximum :: (Foldable t, Ord a) => t a -> a
```

Haskell's Type System

- Type variables.

```
ghci> :t maximum
maximum :: (Foldable t, Ord a) => t a -> a
```

To use "maximum", we need something foldable ("t") that contains values of something that can be ordered ("a"), and returns a value of the same type that is contained in "t".

Haskell's Type System

- Type variables.

```
ghci> :t maximum
maximum :: (Foldable t, Ord a) => t a -> a
```

To use "maximum", we need something foldable ("t") that contains values of something that can be ordered ("a"), and returns a value of the same type that is contained in "t".

```
maximum [1, 2, 3]                   -- "3"
maximum ["a", "ab", "c", "def"] -- "def"
```

```
ghci> :t foldl
foldl :: Foldable t => (b -> a -> b) -> b -> t a -> b
```

```
ghci> :t foldl
foldl :: Foldable t => (b -> a -> b) -> b -> t a -> b
```

To use "foldl", we need something foldable ("t"), a function that takes a type "b" and a type "a", an initial value of type "b", a "t" container that contains values "a" to be folded, and returns a final "b" value.

```
ghci> :t foldl
foldl :: Foldable t => (b -> a -> b) -> b -> t a -> b
```

To use "foldl", we need something foldable ("t"), a function that takes a type "b" and a type "a", an initial value of type "b", a "t" container that contains values "a" to be folded, and returns a final "b" value.

```
foldl (+) 0 [1..100]                    -- 5050
foldl sumSqrt 1.5 ([1..5] :: [Integer])  -- 9.882333
  where
    sumSqrt :: Float -> Integer -> Float
    sumSqrt f i = f + sqrt (fromInteger i)
```

Haskell's Type System

- Defining new types: The "data" keyword

Haskell's Type System

- Defining new types: The "data" keyword

```
data Bool = False | True
```

Haskell's Type System

- Defining new types: The "data" keyword

```
data Bool = False | True

data Tank = Tank String Float Float Float Int
```

Haskell's Type System

- Defining new types: The "data" keyword

```
data Bool = False | True

data Tank = Tank String Float Float Float Int

data Tank = Tank
  { name        :: String
  , speed       :: Float
  , direction   :: Float
  , turretAngle :: Float
  , shells      :: Int
  } deriving (Show, Eq)
```

Haskell's Type System

- Defining new types: The "data" keyword

```
data Bool = False | True

data Tank = Tank String Float Float Float Int

data Tank = Tank
  { name        :: String
  , speed       :: Float
  , direction   :: Float
  , turretAngle :: Float
  , shells      :: Int
  } deriving (Show, Eq)
```

The keyword "deriving" assigns type classes to our new data type

Haskell's Type System

- Type synonyms

Haskell's Type System

- Type synonyms

```haskell
-- A simple definition
prettify :: [Char] -> [Char]
prettify str = str ++ " :-)"
```

Haskell's Type System

- Type synonyms

```
-- A simple definition
prettify :: [Char] -> [Char]
prettify str = str ++ " :-)"

-- Defines nothing new, but clarifies our intention
type String = [Char]
```

Haskell's Type System

- Type synonyms

```haskell
-- A simple definition
prettify :: [Char] -> [Char]
prettify str = str ++ " :-)"

-- Defines nothing new, but clarifies our intention
type String = [Char]

-- Equivalent to the previous example,
-- but easier to understand
prettify :: String -> String
prettify str = str ++ " :-)"
```

Haskell's Type System

- Type synonyms

```haskell
-- A simple definition
prettify :: [Char] -> [Char]
prettify str = str ++ " :-)"

-- Defines nothing new, but clarifies our intention
type String = [Char]

-- Equivalent to the previous example,
-- but easier to understand
prettify :: String -> String
prettify str = str ++ " :-)"

-- We can create new types with type variables
type AssociativeMap k v = [(k, v)]
```

Haskell's Type System

- Recursive type definitions

Haskell's Type System

- Recursive type definitions

```
-- We can define a list as something empty,
-- or something that has a value, and is followed
-- by the remainder of the list
infixr :-: 5
data MyList a = Empty | a :-: (MyList a)
    deriving (Show, Read, Eq, Ord)
```

Haskell's Type System

- Recursive type definitions

```haskell
-- We can define a list as something empty,
-- or something that has a value, and is followed
-- by the remainder of the list
infixr :-: 5
data MyList a = Empty | a :-: (MyList a)
    deriving (Show, Read, Eq, Ord)

-- This enables us to build lists like this:
myList = 1 :-: 2 :-: 3 :-: 4 :-: Empty
```

Haskell's Type System

- Recursive type definitions

```haskell
-- We can define a list as something empty,
-- or something that has a value, and is followed
-- by the remainder of the list
infixr :-: 5
data MyList a = Empty | a :-: (MyList a)
    deriving (Show, Read, Eq, Ord)

-- This enables us to build lists like this:
myList = 1 :-: 2 :-: 3 :-: 4 :-: Empty

-- This is essentially how Haskell defines lists:
haskList = 1 : 2 : 3 : 4 : []
```

```haskell
-- This enables us to build lists like this:
myList = 1 :-: 2 :-: 3 :-: 4 :-: Empty

-- This is essentially how Haskell defines lists:
haskList = 1 : 2 : 3 : 4 : []
```

```haskell
-- A recursive function on a recursive data type
processMyList Empty       = 0
processMyList (x :-: xs) = x + processMyList xs

-- The equivalent definition on a Haskell list:
processHaskList []     = 0
processHaskList (x:xs) = x + processHaskList xs
```

Haskell's Type System

- Defining new type classes

Haskell's Type System

- Defining new type classes

```
-- Define typeclass Eq on type a
class Eq a where
  (==) :: a -> a -> Bool
  (/=) :: a -> a -> Bool
  x == y = not (x /= y)
  x /= y = not (x == y)
```

Haskell's Type System

- Defining new type classes

```
-- Define typeclass Eq on type a
class Eq a where
  (==) :: a -> a -> Bool
  (/=) :: a -> a -> Bool
  x == y = not (x /= y)
  x /= y = not (x == y)

-- create new type
data Ternary = TriTrue | TriUnknown | TriFalse
```

Haskell's Type System

- Defining new type classes

```
-- Define typeclass Eq on type a
class Eq a where
  (==) :: a -> a -> Bool
  (/=) :: a -> a -> Bool
  x == y = not (x /= y)
  x /= y = not (x == y)

-- create new type
data Ternary = TriTrue | TriUnknown | TriFalse

-- make the new type a member of the "Eq" class
instance Eq Ternary where
  TriFalse == TriFalse     = True
  TriUnknown == TriUnknown = True
  TriTrue == TriTrue       = True
  _ == _                   = False
```

```haskell
-- We can use class constraints on new typeclasses
class (Eq a) => Num a where
   ...
-- (This requires Num a to satisfy Eq.)
```

```haskell
-- We can use class constraints on new typeclasses
class (Eq a) => Num a where
   ...
-- (This requires Num a to satisfy Eq.)

-- We can also define optional constraints on
-- a new typeclass.
instance (Eq m) => Eq (Maybe m) where
  Just x == Just y    = x == y
  Nothing == Nothing = True
  _ == _             = False
-- Maybe is now Eq if and only if m is Eq too.
```

```haskell
-- We can use class constraints on new typeclasses
class (Eq a) => Num a where
    ...
-- (This requires Num a to satisfy Eq.)

-- We can also define optional constraints on
-- a new typeclass.
instance (Eq m) => Eq (Maybe m) where
  Just x == Just y    = x == y
  Nothing == Nothing = True
  _ == _             = False
-- Maybe is now Eq if and only if m is Eq too.
```

Outline

The dreaded Monad and Input/Output

- The limits of Analogy.

The dreaded Monad and Input/Output

- The limits of Analogy.

James Iry, One Div Zero
"A Brief, Incomplete, and Mostly Wrong History of Programming Languages"

1990 - A committee formed by Simon Peyton-Jones, Paul Hudak, Philip Wadler, Ashton Kutcher, and People for the Ethical Treatment of Animals creates Haskell, a pure, non-strict, functional language. Haskell gets some resistance due to the complexity of using monads to control side effects. Wadler tries to appease critics by explaining that "a monad is a monoid in the category of endofunctors, what's the problem?"

The dreaded Monad and Input/Output

```haskell
getElem :: String -> String
getElem x = x
  -- ^ some day we'll convert this to a number, and
  --   look up the associated element

main :: IO ()
main = do
  putStrLn "Enter a number from 1 to 118:"
  protons <- getLine
  putStrLn $ "I don't know " ++
     (getElem protons) ++ " from Atoms!"
  return ()
    -- ^ A gratuitous return that doesn't do what
    --   we expect "return" to do!
```

The Monad typeclass

```
class Monad m where
  return :: a -> m a

  -- bind
  (>>=) :: m a -> (a -> m b) -> m b

  -- advance (?)
  (>>) :: m a -> m b -> m b
  x >> y = x >>= \_ -> y

  fail :: String -> m a
  fail msg = error msg
```

The Maybe Monad

The Maybe Monad

```
instance Monad Maybe where
  return x = Just x

  Nothing >>= f = Nothing
  Just x >>= f  = f x

  fail _ = Nothing
```

The Maybe Monad

```haskell
instance Monad Maybe where
  return x = Just x

  Nothing >>= f = Nothing
  Just x >>= f  = f x

  fail _ = Nothing
```

```haskell
Just 3 >>= (\x -> Just "!" >>= (\y -> Just (show x ++ y)))
```

The Maybe Monad

```
instance Monad Maybe where
  return x = Just x

  Nothing >>= f = Nothing
  Just x >>= f  = f x

  fail _ = Nothing
```

```
Just 3 >>= (\x -> Just "!" >>= (\y -> Just (show x ++ y)))
```

```
foo :: Maybe String
foo = Just 3   >>= (\x ->
      Just "!" >>= (\y ->
      Just (show x ++ y)))
```

The Maybe Monad

```haskell
instance Monad Maybe where
  return x = Just x

  Nothing >>= f = Nothing
  Just x >>= f  = f x

  fail _ = Nothing
```

```haskell
Just 3 >>= (\x -> Just "!" >>= (\y -> Just (show x ++ y)))
```

```haskell
foo :: Maybe String
foo = Just 3   >>= (\x ->
      Just "!" >>= (\y ->
      Just (show x ++ y)))
```

```haskell
foo :: Maybe String
foo = do
    x <- Just 3
    y <- Just "!"
    Just (show x ++ y)  -- "Just" here is literally "return"
```

The List Monad

The List Monad

```
instance Monad [] where
  return x = [x]

  xs >>= f = concat (map f xs)

  fail _ = []
```

The List Monad

```
instance Monad [] where
  return x = [x]

  xs >>= f = concat (map f xs)

  fail _ = []
```

```
listOfTuples :: [(Int,Char)]
listOfTuples = do
    n <- [1,2]
    ch <- ['a','b']
    return (n,ch)
```

The List Monad

```haskell
instance Monad [] where
  return x = [x]

  xs >>= f = concat (map f xs)

  fail _ = []
```

```haskell
listOfTuples :: [(Int,Char)]
listOfTuples = do
    n <- [1,2]
    ch <- ['a','b']
    return (n,ch)
```

```haskell
[(n,ch) | n <- [1,2], ch <- ['a','b']]
```

The Axioms of Monad

The Axioms of Monad

- Left identity:

```haskell
return x >>= f      -- is equivalent to "f x"
```

The Axioms of Monad

- Left identity:
  ```
  return x >>= f     -- is equivalent to "f x"
  ```
- Right identity:
  ```
  m >>= return       -- is eqivalent to "m"
  ```

The Axioms of Monad

- Left identity:

```
return x >>= f      -- is equivalent to "f x"
```

- Right identity:

```
m >>= return        -- is eqivalent to "m"
```

- Associativity:

```
(m >>= f) >>= g     -- is equivalent to
                    -- "m >>= (\x -> f x >>= g)"
```

Outline

Slides:

`https://github.com/snowfarthing/slides/blob/master/2019-openwest-fundamentals-of-haskell.pdf`

Any Questions?

Outline

References and Resources

http://learnyouahaskell.com (Learn You a Haskell for Great Good)
https://www.haskell.org/tutorial/index.html (A Gentle Introduction)
http://book.realworldhaskell.org (Real World Haskell)
https://hackage.haskell.org
https://hoogle.haskell.org
https://vaibhavsagar.com/blog/2016/10/12/monad-anti-tutorial
https://unknownparallel.wordpress.com/zero-analogy-monad-tutorial
https://byorgey.wordpress.com/2009/01/12/abstraction-intuition-and-the-monad-tutorial-fallacy
https://ncatlab.org/nlab/files/WadlerMonads.pdf
http://web.cecs.pdx.edu/~mpj/pubs/RR-1004.pdf (Composing Monads)
http://math.mit.edu/~dspivak/teaching/sp18/7Sketches.pdf
(Seven Sketches of Compositionality)
http://adit.io/posts/2013-04-17-functors,_applicatives,_and_monads_in_pictures.html