FizzBuzz in Haskell

Owen Lynch

April 10, 2018

Introducing Haskell

```haskell
fact :: Int -> Int
fact 0 = 1
fact n = n * fact (n-1)
```

Introducing Haskell

```haskell
fact :: Int -> Int
fact 0 = 1
fact n = n * fact (n-1)
```

1. Types

Introducing Haskell

```
fact :: Int -> Int
fact 0 = 1
fact n = n * fact (n-1)
```

1. Types
2. Functions

Introducing Haskell

```haskell
fact :: Int -> Int
fact 0 = 1
fact n = n * fact (n-1)
```

1. Types
2. Functions
3. Pattern Matching

Introducing Haskell

```haskell
fact :: Int -> Int
fact 0 = 1
fact n = n * fact (n-1)
```

1. Types
2. Functions
3. Pattern Matching
4. Recursion

Myths about Haskell

Myths about Haskell

1. Haskell is hard

Myths about Haskell

1. Haskell is hard
   1.1 *Learning* Haskell is hard, *programming* Haskell is easy!

Myths about Haskell

1. Haskell is hard
   1.1 *Learning* Haskell is hard, *programming* Haskell is easy!
   1.2 Haskell lets you choose the level of abstraction that you are comfortable with, and you can slowly increase this while being productive the whole time

Myths about Haskell

1. Haskell is hard
    1.1 *Learning* Haskell is hard, *programming* Haskell is easy!
    1.2 Haskell lets you choose the level of abstraction that you are comfortable with, and you can slowly increase this while being productive the whole time
2. Haskell is for techbro superprogrammers

Myths about Haskell

1. Haskell is hard
   1.1 *Learning* Haskell is hard, *programming* Haskell is easy!
   1.2 Haskell lets you choose the level of abstraction that you are comfortable with, and you can slowly increase this while being productive the whole time
2. Haskell is for techbro superprogrammers
   2.1 Haskell is for people who can't keep lots of stuff in their head at once and want the compiler to make sure they aren't messing things up

Myths about Haskell

1. Haskell is hard
   1.1 *Learning* Haskell is hard, *programming* Haskell is easy!
   1.2 Haskell lets you choose the level of abstraction that you are comfortable with, and you can slowly increase this while being productive the whole time
2. Haskell is for techbro superprogrammers
   2.1 Haskell is for people who can't keep lots of stuff in their head at once and want the compiler to make sure they aren't messing things up
   2.2 The Haskell community is one of the nicest and friendliest online communities I've been a part of, in part because there is so much to learn that everyone is comparatively a noob

Myths about Haskell

1. Haskell is hard
   1.1 *Learning* Haskell is hard, *programming* Haskell is easy!
   1.2 Haskell lets you choose the level of abstraction that you are comfortable with, and you can slowly increase this while being productive the whole time
2. Haskell is for techbro superprogrammers
   2.1 Haskell is for people who can't keep lots of stuff in their head at once and want the compiler to make sure they aren't messing things up
   2.2 The Haskell community is one of the nicest and friendliest online communities I've been a part of, in part because there is so much to learn that everyone is comparatively a noob
3. Haskell is for academic ivory-towerists who do too much category theory for their own good

Myths about Haskell

1. Haskell is hard
    1.1 *Learning* Haskell is hard, *programming* Haskell is easy!
    1.2 Haskell lets you choose the level of abstraction that you are comfortable with, and you can slowly increase this while being productive the whole time
2. Haskell is for techbro superprogrammers
    2.1 Haskell is for people who can't keep lots of stuff in their head at once and want the compiler to make sure they aren't messing things up
    2.2 The Haskell community is one of the nicest and friendliest online communities I've been a part of, in part because there is so much to learn that everyone is comparatively a noob
3. Haskell is for academic ivory-towerists who do too much category theory for their own good
    3.1 Haskell is actually a very practical language for all sorts of tasks, and it has been battletested in industry for decades

Introducing FizzBuzz

Make a list of the numbers from 1 to 100, except...

Introducing FizzBuzz

Make a list of the numbers from 1 to 100, except...

1. For every number divisible by 3, put "Fizz" instead

Introducing FizzBuzz

Make a list of the numbers from 1 to 100, except...

1. For every number divisible by 3, put "Fizz" instead
2. For every number divisible by 5, put "Buzz" instead

Introducing FizzBuzz

Make a list of the numbers from 1 to 100, except...

1. For every number divisible by 3, put "Fizz" instead
2. For every number divisible by 5, put "Buzz" instead
3. For every number divisible by 5 and 3, put "FizzBuzz" instead

Introducing FizzBuzz

Make a list of the numbers from 1 to 100, except...

1. For every number divisible by 3, put "Fizz" instead
2. For every number divisible by 5, put "Buzz" instead
3. For every number divisible by 5 and 3, put "FizzBuzz" instead

How would you solve FizzBuzz?

Lists in Haskell

```haskell
data [a] = [] | a : [a]
-- data List a = Empty | Cons a (List a)
-- (:) x xs == x : xs
-- ([]) a == [a]
```

Lists in Haskell

```haskell
data [a] = [] | a : [a]
-- data List a = Empty | Cons a (List a)
-- (:) x xs == x : xs
-- ([]) a == [a]
```

1. Data declarations

Lists in Haskell

```
data [a] = [] | a : [a]
-- data List a = Empty | Cons a (List a)
-- (:) x xs == x : xs
-- ([]) a == [a]
```

1. Data declarations
2. Parametric data declarations

Lists in Haskell

```haskell
data [a] = [] | a : [a]
-- data List a = Empty | Cons a (List a)
-- (:) x xs == x : xs
-- ([]) a == [a]
```

1. Data declarations
2. Parametric data declarations
3. Special syntax for stuff

What does it look like?

```haskell
firstFourPrimes :: [Int]
firstFourPrimes = 2:(3:(5:(7:[])))

everFlavoredBeanFlavors :: [String]
everFlavoredBeanFlavors =
  ["earwax", "vomit", "marmalade", "spinach"]
```

(put drawing of a linked list here)

A useful function

```haskell
range :: Int -> Int -> [Int]
range n m
  | n == m    = m:[]
  | otherwise = m:(range (n+1) m)
-- range n m == [n..m]
```

A useful function

```haskell
range :: Int -> Int -> [Int]
range n m
  | n == m    = m:[]
  | otherwise = m:(range (n+1) m)
-- range n m == [n..m]
```

1. Currying

A useful function

```
range :: Int -> Int -> [Int]
range n m
   | n == m     = m:[]
   | otherwise = m:(range (n+1) m)
-- range n m == [n..m]
```

1. Currying
2. Guard Notation

Functors! (Scary? No!)

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

instance Functor ([]) where
  fmap g [] = []
  fmap g (x:xs) = (g x):(fmap g xs)
  -- fmap :: (a -> b) -> ([]) a -> ([]) b
  -- fmap :: (a -> b) -> [a] -> [b]
```

What does it look like?

(put drawing of a functor diagram here)

Luke, use the fmap!

```haskell
-- String == [Char]

fizzbuzz1 :: Int -> String
fizzbuzz1 n
  | rem n 15 == 0 = "FizzBuzz"
  | rem n  5 == 0 = "Buzz"
  | rem n  3 == 0 = "Fizz"
  | otherwise     = show n

sol1 :: [String]
sol1 = fmap fizzbuzz1 [1..100]
```

Discuss first solution

1. What did we like?

Discuss first solution

1. What did we like?
    1.1 Types and Typeclasses!

Discuss first solution

1. What did we like?
    1.1 Types and Typeclasses!
    1.2 Recursion!

Discuss first solution

1. What did we like?
   1.1 Types and Typeclasses!
   1.2 Recursion!
   1.3 Combinators that allowed us to not do recursion!

Discuss first solution

1. What did we like?
    1.1 Types and Typeclasses!
    1.2 Recursion!
    1.3 Combinators that allowed us to not do recursion!
    1.4 Case syntax!

Discuss first solution

1. What did we like?
   1.1 Types and Typeclasses!
   1.2 Recursion!
   1.3 Combinators that allowed us to not do recursion!
   1.4 Case syntax!
2. What did we not like?

Discuss first solution

1. What did we like?
   1.1 Types and Typeclasses!
   1.2 Recursion!
   1.3 Combinators that allowed us to not do recursion!
   1.4 Case syntax!
2. What did we not like?
   2.1 Not extensible enough! (What about Bazz??)

FizzBuzzBazz

```haskell
fizzbuzzbazz1 :: Int -> String
fizzbuzzbazz1 n
  | rem n 105 == 0 = "FizzBuzzBazz"
  | rem n  35 == 0 = "BuzzBazz"
  | rem n  21 == 0 = "FizzBazz"
  | rem n  15 == 0 = "FizzBuzz"
  | rem n   7 == 0 = "Bazz"
  | rem n   5 == 0 = "Buzz"
  | rem n   3 == 0 = "Fizz"
  | otherwise      = show n
```

FizzBuzzBazz

```
fizzbuzzbazz1 :: Int -> String
fizzbuzzbazz1 n
   | rem n 105 == 0 = "FizzBuzzBazz"
   | rem n  35 == 0 = "BuzzBazz"
   | rem n  21 == 0 = "FizzBazz"
   | rem n  15 == 0 = "FizzBuzz"
   | rem n   7 == 0 = "Bazz"
   | rem n   5 == 0 = "Buzz"
   | rem n   3 == 0 = "Fizz"
   | otherwise      = show n
```

This is terrible!

And now for something completely different

```
class Monoid m where
  mempty :: m
  mappend :: m -> m -> m
  -- (<>) == mappend
```

And now for something completely different

```haskell
class Monoid m where
  mempty :: m
  mappend :: m -> m -> m
  -- (<>) == mappend
```

We require a couple laws

And now for something completely different

```haskell
class Monoid m where
  mempty :: m
  mappend :: m -> m -> m
  -- (<>) == mappend
```

We require a couple laws

1. Identity

And now for something completely different

```haskell
class Monoid m where
  mempty :: m
  mappend :: m -> m -> m
  -- (<>) == mappend
```

We require a couple laws

1. Identity

```haskell
a <> mempty == a == mempty <> a
```

And now for something completely different

```haskell
class Monoid m where
  mempty :: m
  mappend :: m -> m -> m
  -- (<>) == mappend
```

We require a couple laws

1. Identity

```haskell
a <> mempty == a == mempty <> a
```

2. Associativity

And now for something completely different

```haskell
class Monoid m where
  mempty :: m
  mappend :: m -> m -> m
  -- (<>) == mappend
```

We require a couple laws

1. Identity

```haskell
a <> mempty == a == mempty <> a
```

2. Associativity

```haskell
(a <> b) <> c == a <> (b <> c)
```

You already know lots of monoids!

Take a couple minutes and think of monoids.

List Monoid

```haskell
(++) :: [a] -> [a] -> [a]
[] ++ ys = ys
(x:xs) ++ ys = x:(xs ++ ys)

instance Monoid [a] where
  mempty = []
  mappend = (++)
```

Maybe Monoid

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

instance (Monoid a) => Monoid (Maybe a) where
  mempty = Nothing
  mappend Nothing Nothing = Nothing
  mappend (Just a) Nothing = Just a
  mappend Nothing (Just b) = Just b
  mappend (Just a) (Just b) = Just (a <> b)
```

Maybe Monoid

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

instance (Monoid a) => Monoid (Maybe a) where
  mempty = Nothing
  mappend Nothing Nothing = Nothing
  mappend (Just a) Nothing = Just a
  mappend Nothing (Just b) = Just b
  mappend (Just a) (Just b) = Just (a <> b)
```

Note the "type constraint" in the instance for the Maybe monoid; we need this in order to deal with the last case.

Two useful functions

```haskell
foldl :: (a -> b -> a) -> a -> [b] -> a
foldl f acc [] = acc
foldl f acc (x:xs) = foldl f (f acc x) xs
-- sum == foldl (+) 0
-- length == foldl (\n _ -> n + 1) 0

mconcat :: (Monoid m) => [m] -> m
mconcat = foldl mappend mempty
-- mconcat ["tweedle", "dee"] == "tweedledee"
```

Working with Maybes

```haskell
fromMaybe :: a -> Maybe a -> a
fromMaybe def Nothing = def
fromMaybe _ (Just y) = y

makeMaybe :: Bool -> a -> Maybe a
makeMaybe True x = Just x
makeMaybe False _ = Nothing
```

FizzBuzz revisited

```haskell
zzer1 :: [(Int, String)] -> Int -> String
zzer1 zzConds n = fromMaybe (show n) zzs
  where
    zzs = mconcat (fmap maybeZz zzConds)
    maybeZz (k, zz) = makeMaybe (rem n k == 0) zz

sol2 :: [String]
sol2 = fmap (zzer1 fizzbuzz) [1..100]
  where fizzbuzz = [(3, "Fizz"), (5, "Buzz")]
```

Reflections on take 2

What do we like?

Reflections on take 2

What do we like?

1. Much more extensible!

Reflections on take 2

What do we like?

1. Much more extensible!
2. Shorter!

Reflections on take 2

What do we like?

1. Much more extensible!
2. Shorter!

What do we not like?

Reflections on take 2

What do we like?

1. Much more extensible!
2. Shorter!

What do we not like?

1. Division

Infinity and Beyond!

```
onesForever :: [Int]
onesForever = 1:onesForever
-- take 3 onesForever == [1,1,1]

primes :: [Int]
primes = from 2
  where
    from k = k:(filter (ndiv k) (from (k+1)))
    ndiv k n = rem n k /= 0
-- take 4 primes == [2,3,5,7]
```

Infinity and Beyond!

```haskell
onesForever :: [Int]
onesForever = 1:onesForever
-- take 3 onesForever == [1,1,1]

primes :: [Int]
primes = from 2
  where
    from k = k:(filter (ndiv k) (from (k+1)))
    ndiv k n = rem n k /= 0
-- take 4 primes == [2,3,5,7]
```

1. Haskell is lazy – it only computes values when you ask for them

Loops and Repeats

```
looper :: Int -> a -> [Maybe a]
looper n s = loop (n-1)
  where
    loop 0 = (Just s):(loop (n-1))
    loop k = Nothing:(loop (k-1))

repeat :: a -> [a]
repeat x = x:(repeat x)
```

Zipping Lists

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

fibs :: [Int]
fibs = 0:1:(zipWith (+) fibs (tail fibs))
-- This one's a braintwister!
```

FizzBuzz: The Final Showdown

```haskell
zzer2 :: [(Int, String)] -> [Maybe String]
zzer2 zzConds = foldl (zipWith (<>)) init zzLists
  where
    zzLists = map (uncurry looper) zzConds
    init = repeat Nothing

sol3 :: [String]
sol3 = zipWith fromMaybe (fmap show [1..100]) zzs
  where zzs = zzer2 [(3,"Fizz"), (5,"Buzz")]
```