William Harrell 09/14/21 LUG @ NC State

Functional Programming in

(What the hell is a monad?)

Procedural Programming

- Program is made of several steps the computer is to carry out
- You tell the computer **what** to do (imperative)
- What you're probably used to
  - Python, Java, C, JavaScript, Lua

Functional Programming

- Program is made of functions that map values onto other values
- You tell the computer **what you want** (declarative)
  - Lisp, Erlang, Elixir, OCaml, F#, Scala

$$f(x) = x^2$$

Haskell is purely functional

- You have to give up lots of things you're used to
  - Mutation (x = 1; x =2;)
  - Looping (use recursion instead)
  - Side effects (IO, globals and state)
    - (not really, but it's kind of funky)

Why?

- Being used more in mainstream languages
- Often results in code that is shorter and easy to read
- Allows you to do complex operations very easy
- Clout

Diving in

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

-- λ reverseList [4,3,2,1]
-- [1,2,3,4]
-- λ reverseList "abc"
-- "cba"
```

Types

- Haskell has a powerful type system that's clear to read
- `mult :: Int -> Int -> Int`
  - Takes two ints, returns an int
- `multBy :: Int -> (Int -> Int)`
  - Takes an int, returns a function which takes an int and returns an int

Types

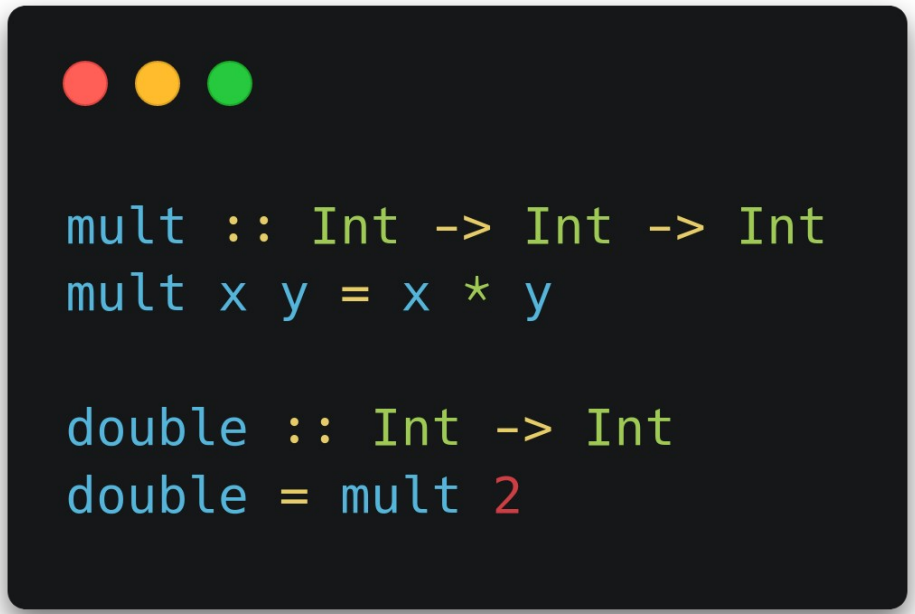
- `reverseList :: [a] -> [a]`
  - Takes a list of anything, and returns a list of anything
- `allSame :: Eq a => [a] -> Bool`
  - Takes a list of anything that is an instance of Eq, returns a boolean
  - Being an instance of Eq means two of a type can be equal (`==`) or not equal (`/=`)

Currying

- In Haskell, functions can only take on parameter
- But, functions can return functions
- Similar to idea of closures in procedural languages
- You can still use them as though they're taking multiple arguments, without having to worry about currying behind the scenes

Currying

- Allows for partial application



```
mult :: Int -> Int -> Int
mult x y = x * y

double :: Int -> Int
double = mult 2
```

Currying

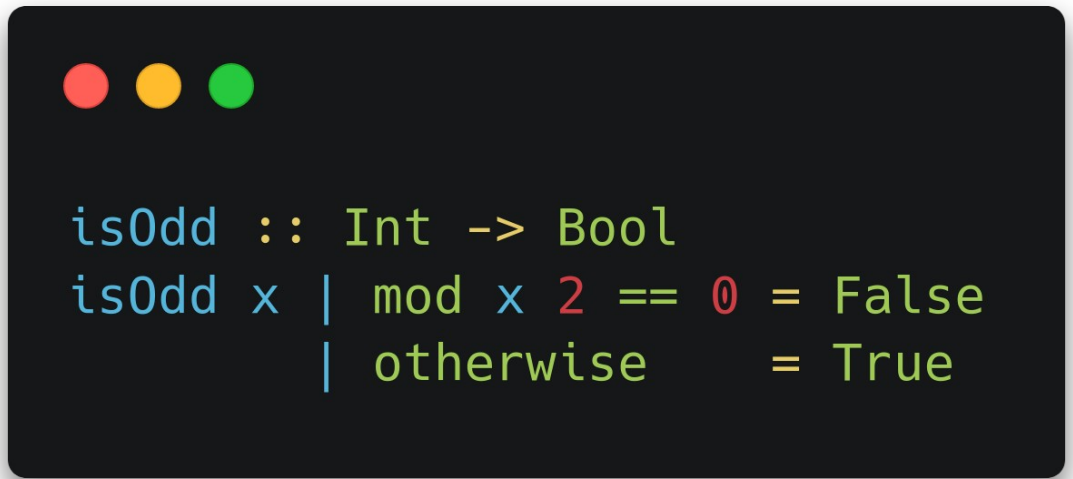
- To Haskell, parentheses in type definitions **do not matter** unless taking functions as parameters
  - `mult :: Int -> Int -> Int`
  - `multBy :: Int -> (Int -> Int)`
  - Both of these mean the exact same thing
- But they usually help a reader better understand what your function is supposed to mean

Function parameters

- Haskell offers many ways to give and accept parameters
- Regular
- Guards
- Pattern matching

Guarded parameters

- Think of splitting an if-else block into separate functions



```haskell
isOdd :: Int -> Bool
isOdd x | mod x 2 == 0 = False
        | otherwise    = True
```

Pattern matching

- Possibilities are attempted, going down
- If none match, error is thrown
  - A side effect (gross)

```
int2str :: Int -> String
int2str 0 = "Zero"
int2str 1 = "One"
int2str 2 = "Two"
int2str 3 = "Three"
int2str 4 = "Four"
```

Pattern matching, multiple arguments

- Underscore discards argument

```haskell
mult :: Int -> Int -> Int
mult 0 _ = 0
mult _ 0 = 0
mult = (*)
```

Pattern matching, lists

- (x:xs) pattern puts first element in x, rest in xs

```haskell
sum :: [Int] -> Int
sum [] = 0
sum (x:xs) = x + sum xs

length :: [a] -> Int
length [] = 0
length (_:xs) = 1 + length xs
```

Lists

- Lists and recursion are your replacement for loops
- All elements are of same type
- Lists are constructed one element at a time, with `:`
- Lists can be concatenated with `++`
- Random access with `[1,2,3,4] !! 0`

List functions


```
λ x = [1..4]
λ x
-- [1,2,3,4]

-- Get first element
λ head x
-- 1
-- Last elements
λ tail x
-- [2,3,4]

-- Take n elements
λ take 2 x
-- [1,2]
-- Drop n elements
λ drop 2 x
-- [3,4]
```

```
-- Get last element
λ last x
-- 4
-- Drop last element
λ init x
-- [1,2,3]

-- Check if list is empty
λ null x
-- False

-- Check if element in list
λ elem 3 x
-- True
-- P.S. You can use functions with two args
-- like operators by doing this:
λ 5 `elem` x
-- False
```

List comprehensions

- Generate a new list from an existing one very easily

```
λ [int2str i | i <- [1..5]]
-- ["One","Two","Three","Four","Five"]
λ [int2str i | i <- [1..10], i `mod` 2 == 0]
-- ["Two","Four","Six","Eight","Ten"]
λ [(int2str x, int2str y) | x <- [1..3], y <- [4..5]]
-- [("One","Four"),
--  ("One","Five"),
--  ("Two","Four"),
--  ("Two","Five"),
--  ("Three","Four"),
--  ("Three","Five")]
```

Advanced list functions


```
-- Map a function over a list
λ map int2str [1..4]
-- ["One","Two","Three","Four"]

-- Type of [1..4] is [Int]
-- Type of int2str is Int -> String
-- Type of output is [String]
-- Type of map is (a -> b) -> [a] -> [b]
```

```
-- Filter a list with a predicate
λ filter isOdd [1..4]
-- [1..3]
-- Type of filter is (a -> Bool) -> [a] -> [a]

-- Combine two lists into tuples
λ zip [1..3] [4..6]
-- [(1,4),(2,5),(3,6)]
```

Lambda functions

- Quick and dirty functions, ideal for map and filter

```
λ (\a -> a + 2) 1
-- 3

λ (\a b -> a + b) 3 5
-- 8

λ filter (\a -> isOdd a && a > 4) [1..10]
-- [5,7,9]
```

Function composition

- (.) operator works similarly to function composition from math

```haskell
halve :: Int -> String
halve = int2str . (\x -> div x 2)

-- Type of (.) is (b -> c) -> (a -> b) -> (a -> c)
-- so it can't work on functions with more than one arg.
-- You can take advantage of currying and do this:

myDiv :: Int -> Int -> String
myDiv = (int2str . ) . div

-- But it's probably more readable to do it the more
-- obvious way:

myDiv x y = int2str (div x y)
```

Folding

- Many recursive functions follow a common pattern:
  - Base case
  - Operation that takes next element and accumulation

- When applied to [1..4]:
  - (1 + (2 + (3 + (4 + 0))))
  - (1 + (2 + (3 + (4))))
  - (1 + (2 + (7)))
  - (1 + (9))
  - (10) → 10

```haskell
sum :: [Int] -> Int
-- foldr takes a function, and a base case
sum = foldr (\x acc -> x + acc) 0
```

foldl1

- Order of ops reversed (folds left)
- 1 variant uses first element as base case

```
subList :: [Int] -> Int
subList = foldl1 (\acc x -> acc - x)

λ subList [10,7,2]
-- 1
```

Defining your data

- Structuring your own data is easy, and there are several tools for you
  - Types
  - Data
  - Typeclasses and Newtypes

Types

- Just an alias for an existing type, good for readability

```haskell
type Pair a = (a, a)
type Pos = Pair Int

-- Haskell will consider all of these to be
-- the same type
-- (5, 5)
-- (4, 3) :: Pair Int
-- (-1, 2) :: Pos
```

Data

- Similar to a struct in C, with some new tricks

```
data Person = PersonPhone String String Int | PersonEmail String String String
    deriving Show
-- deriving Show automatically makes it an instance of the Show typeclass, so it can
-- be printed to the console

λ PersonPhone "John" "Doe" 1234567890
-- PersonPhone "John" "Doe" 1234567890
-- It works, but you have to remember what goes where,
-- and to use it in functions you have to do this:
-- f (PersonPhone fstName lstName phone) = ...

data BetterPerson = BPersonPhone { bppFirstName :: String
                                 , bppLastName  :: String
                                 , bppPhone     :: Int }
                  | BPersonEmail { bpeFirstName :: String
                                 , bpeLastName  :: String
                                 , bpeEmail     :: String }
    deriving Show

-- Now you can do this
λ jdoe = BPersonEmail { bpeFirstName = "John", bpeLastName = "Doe", bpeEmail = "jdoe@gmail.com"}
λ bpeFirstName jdoe
-- "John"
```

Recursive data


```haskell
data NaturalNumber = Zero | Succ NaturalNumber
    deriving Show

nat2int :: NaturalNumber -> Int
nat2int Zero = 0
nat2int (Succ n) = 1 + nat2int (n-1)

int2nat :: Int -> NaturalNumber
int2nat 0 = Zero
int2nat n = Succ (int2nat (n-1))

λ int2nat 3
-- Succ (Succ (Succ Zero))
λ nat2int (Succ (Succ Zero))
-- 2
```

Typeclasses

- Defines a set of functions for a given type

```haskell
class Eq a where
    (==) :: a -> a -> Bool
    (/=) :: a -> a -> Bool
    x == y = not (x /= y)
    x /= y = not (x == y)

instance Eq NaturalNumber where
    Zero == Zero = True
    Zero == _ = False
    _ == Zero = False
    (Succ n1) == (Succ n2) = nat2int n1 == nat2int n2
```

Newtypes

- Similar to data, but only allows for one constructor and field
- Used to wipe typeclass definitions for an existing type
- Also comes with some performance benefits

```haskell
newtype MyList a = MyList { getList :: [a] }

-- Can't use !!, :, or ++ with MyLists
-- We're free to define them ourselves now
```

The hard stuff

- Functors 😀
- Applicatives 😐

The hard stuff

- Monads

The Maybe type

- Easy way to model a computation that may fail

```haskell
data Maybe a = Just a | Nothing
    deriving (Eq, Ord)

safeDiv :: Int -> Int -> Maybe Int
safeDiv _ 0 = Nothing
safeDiv x y = Just (x `div` y)
```

Maybe as a functor

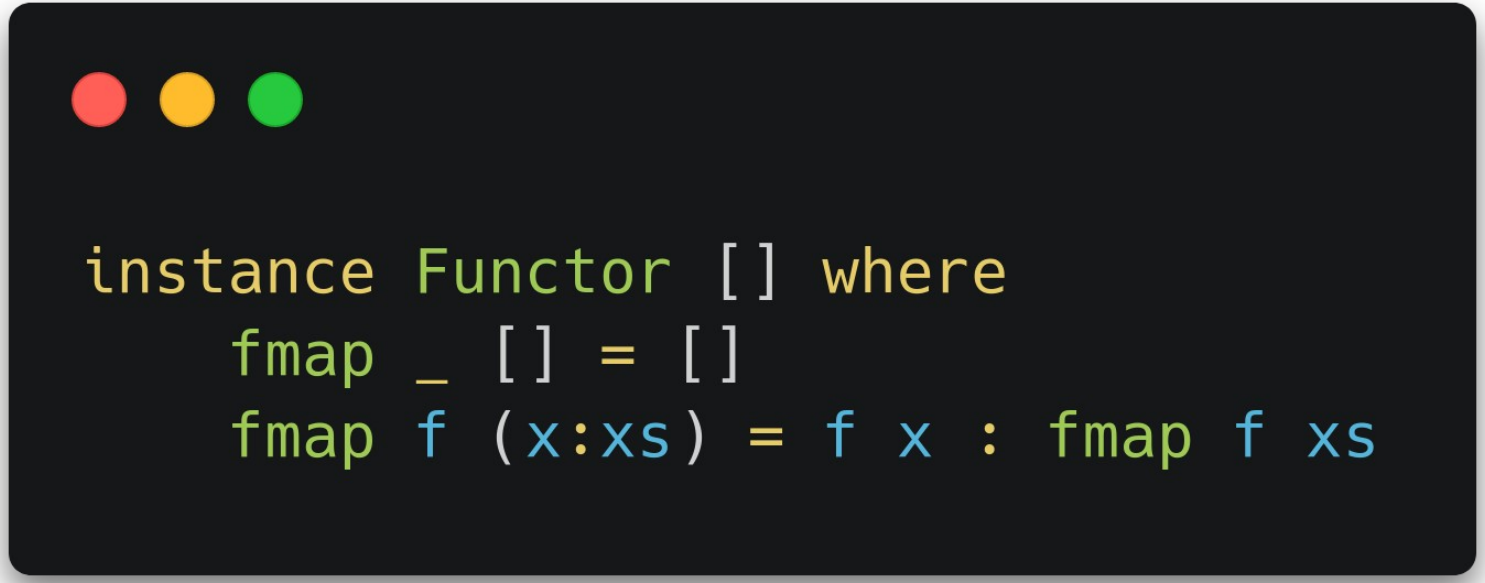
- With a functor, we can map a function onto its inner value(s)

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

λ fmap (+2) (safediv 9 3)
-- Just 5
λ fmap (+2) (safediv 9 0)
-- Nothing
```

Lists as a functor

- map is fmap for lists



```haskell
instance Functor [] where
    fmap _ [] = []
    fmap f (x:xs) = f x : fmap f xs
```

Applicatives

- Functors are cool, but they only work on functions with one argument
- Applicatives solve this problem
- They allow you to wrap a function within a type (pure), and then feed it parameters

Maybe as an applicative


```
instance Applicative Maybe where
    pure = Just
    Nothing <*> _ = Nothing
    (Just f) <*> something = fmap f something

λ pure (+) <*> Just 2 <*> Just 3
-- Just 5

-- A little bit of sugar
λ (*) <$> Just 3 <*> Just 4
-- Just 12
λ (\a b c -> a * b + c) <$> Just 10 <*> Just 5 <*> Just 25
-- Just 75

λ (+) <$> Nothing <*> Just 4
-- Nothing
λ (*) <$> Just 3 <*> Nothing
-- Nothing
```

Modeling a chain of computation

- We have a process that could fail at any step
- Let's use Maybe
- Each step needs to be able to pass on failure (Nothing) from the step before

Modeling a chain of computation

- Could do it all in one function like this, but I'd rather not

```
divFour :: Int -> Int -> Int -> Int -> Maybe Int
divFour a b c d = if a == 0
                  then Nothing
                  else
                    if b == 0
                    then Nothing
                    else
                      if c == 0
                      then Nothing
                      else
                        if d == 0
                        then Nothing
                        else Just (div (div (div a b) c) d)
```

Modeling a chain of computation

- Can we use functors?
  - No, because that only applies the function to the inner value. If the function fails, we'd still get an exception, or at best, `Just Nothing`
- What about applicatives?
  - No, because we have to start with a function and feed values into it. We can't chain the output of one function to the output of another

Modeling a chain of computation

- We need to be able to:
  - Chain together as many functions as we want
  - Be able to propate failure to the end of the chain, from anywhere in the chain
    - Each function needs to be able to return Nothing
    - But we also want our functions to be as generic as possible, so their parameters shouldn't be wrapped in Maybe
    - Their types should be `a -> Just a`
    - But now, how do we pass output (Just a) to the next function (which expects an a)?

Modeling a chain of computation

- Let's make an intermediary function to handle it

```haskell
bind :: Maybe a -> (a -> Maybe a) -> Maybe a
bind Nothing _ = Nothing
bind (Just x) f = f x

safeDiv :: Int -> Int -> Maybe Int
safeDiv _ 0 = Nothing
safeDiv x y = Just (x `div` y)

divBy :: Int -> Int -> Maybe Int
divBy x y = safeDiv y x

λ safeDiv 24 2 `bind` divBy 2 `bind` divBy 3
-- Just 2
λ safeDiv 24 0 `bind` divBy 2 `bind` divBy 3
-- Nothing
λ safeDiv 24 2 `bind` divBy 0 `bind` divBy 3
-- Nothing
λ safeDiv 24 2 `bind` divBy 2 `bind` divBy 0
-- Nothing
```

Monads

- This is really handy, someone should name this!
- `monad`
- Oh
- But hey, that wasn't so bad, was it?

Monads

- In practice, `bind` is replaced with >>=
- And there are many different monads, Maybe is just one of them
- But they all follow a similar idea, of being able to pass failure, or really any context, down a pipeline

Doin' stuff

- Haskell functions are pure, they have no side effects
- … but every way we could interact with a program is a side effect
- So is Haskell actually pure?

Doin' stuff

- *Every* Haskell function *is* pure
- So to get around it, we can make a type called IO, which is a side effect
- Functions can now return side effects, **but the function itself is not what causes a side effect to happen**

Doin' stuff

- To execute a side effect, set `main` equal to it

```
main = putStrLn "Hello, World!"

λ :type main
-- IO ()

-- IO types can also wrap a value
λ :type getLine
-- IO String

main = getLine >>= (\s -> putStrLn ("Hello, " ++ s))
```

Doin' stuff

- The flow of your program can be represented purely, so you're not really breaking any "purely functional" rules
- Haskell just executes the IO action you've given it, and uses the rules you've defined in your functions

Doin' lots of stuff

- You'll probably want to do more than one or two things in your main action
- This sucks

```
main = putStrLn "First name?" >>
    getLine >>= (\firstName ->
    putStrLn "Last name?" >>
    getLine >>= (\lastName ->
    putStrLn "Email?" >>
    getLine >>= (\email ->
    putStrLn (lastName ++ ", " ++ firstName ++ " (" ++ email ++ ")"))))
```

(do)in' stuff

- Haskell offers some more sugar
- This translates straight to what was seen earlier, just easier to read
- In fact, do syntax works on any monad, not just IO

```haskell
contactStr :: String -> String -> String -> String
contactStr f l e = l ++ ", " ++ f ++ " (" ++ e ++ ")"

main = do
    putStrLn "First name?"
    -- bind result of IO to a variable
    firstName <- getLine
    putStrLn "Last name?"
    lastName <- getLine
    putStrLn "Email?"
    email <- getLine
    let result = contactStr firstName lastName email
    -- could also do:
    -- result <- return (contactStr firstName lastName email)
    -- which wraps the String from contactStr in IO
    -- but there's no point to it since we're binding it right after
    putStrLn result
```

Laziness

- Since mutability is not supported, a function will always return the same output for the same input
  - Don't worry about IO for now
- Because of this, an expression will always evaluate the same
  - Also makes parallelization/concurrency very easy
- Haskell is lazy: it doesn't evaluate an expression until it absolutely needs to

Laziness


```
hardMathProblem :: Int
hardMathProblem = (code to actually calculate it)

λ let x = hardMathProblem
-- Interpreter sets x equal to hardMathProblem. No calculation
-- is actually performed because Haskell doesn't have to show
-- you the result yet.
λ x
-- Now, Haskell needs to figure out what x really is. This is
-- where computation starts, and where the interpreter would pause.
λ 3.14159.......
```

Laziness

- This has performance benefits (sometimes), since unnecessary computation isn't performed
  - Also makes debugging harder :(
- But it also allows for one very cool trick that is impossible (or at least much less natural) in a non-lazy language

Laziness: Infinite data


```
λ x = [1..]
-- x is now the list of all integers, starting at 1
λ take 5 x
-- [1,2,3,4,5]

-- Of course, this doesn't work, but Haskell will still try
-- sum :: [Int] -> Int
λ sum x

-- You can also make your own infinite data structures.
-- This hurts my head, but Haskell doesn't blink.
primes = filterPrime[2..]
    where filterPrime (p:xs) =
        p : filterPrime [x | x <- xs, x `mod` p /= 0]

λ take 5 primes
-- [2,3,5,7,11]
```

Laziness and IO


```
-- Let's say you have a config file on your drive, and you have a function
-- to open and parse it into a type Config:
let conf = parseConfigFile "config.json"
-- conf's type is IO Config
-- Now let's say you try to use it, but since typing the above line, you've
-- deleted the config file.
λ conf
-- Error! File not found
-- Haskell didn't try to open the file until it really needed to

-- However, by instead binding the result of parseConfigFile:
conf <- parseConfigFile "config.json"
-- IO Config is unwrapped into just Config, and Haskell is forced to get its
-- true value even if you don't necessarily need it yet.
```

The Haskell Ecosystem

- Glasgow Haskell Compiler (GHC)
  - The GCC of Haskell
  - Can also interpret, run a REPL
- Hackage: Package index
- Stack: Build system, package manager
- Cabal: Older build system
  - Still updated, but falling out of favor for Stack
- Haddock: Documentation generator
  - Like JavaDoc, Doxygen

Further resources

- Learn You a Haskell For Great Good!
  - Miran Lipovaca
  - Available for free online
- Programming in Haskell
  - Graham Hutton (of Computerphile fame)
- Real World Haskell
  - O'Reilly
- Tsoding on YouTube

Acknowledgements

Some examples were taken from LYAHFGG, Programming in Haskell, and the Haskell website.

carbon.now.sh was used to generate code snippets.


senior devs, professors,
embedded developers,
computer scientists, linus