Learn Functional Programming with PureScript

(Or I'll buy you a coffee!)

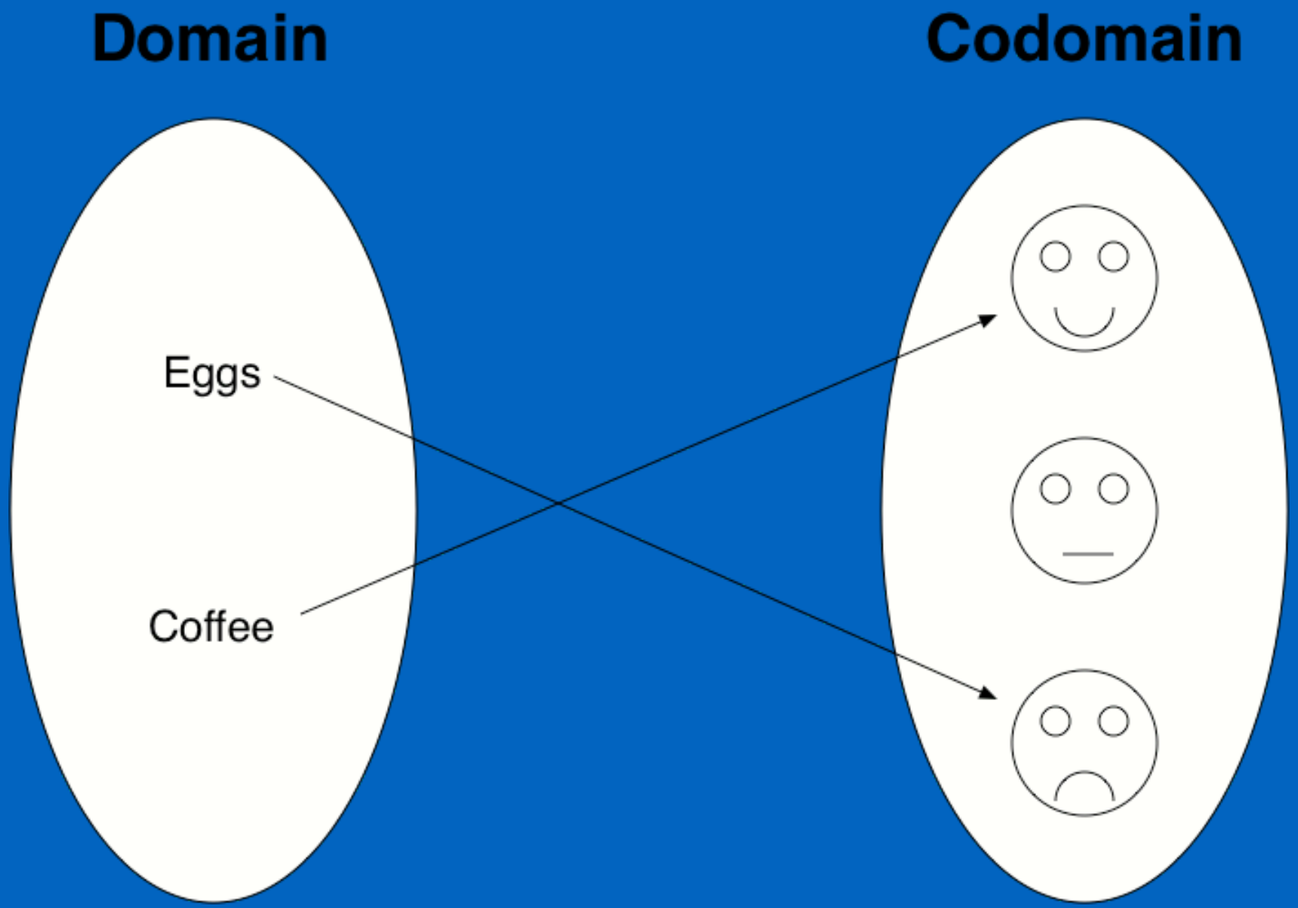
John A. De Goes — @jdegoes

Agenda

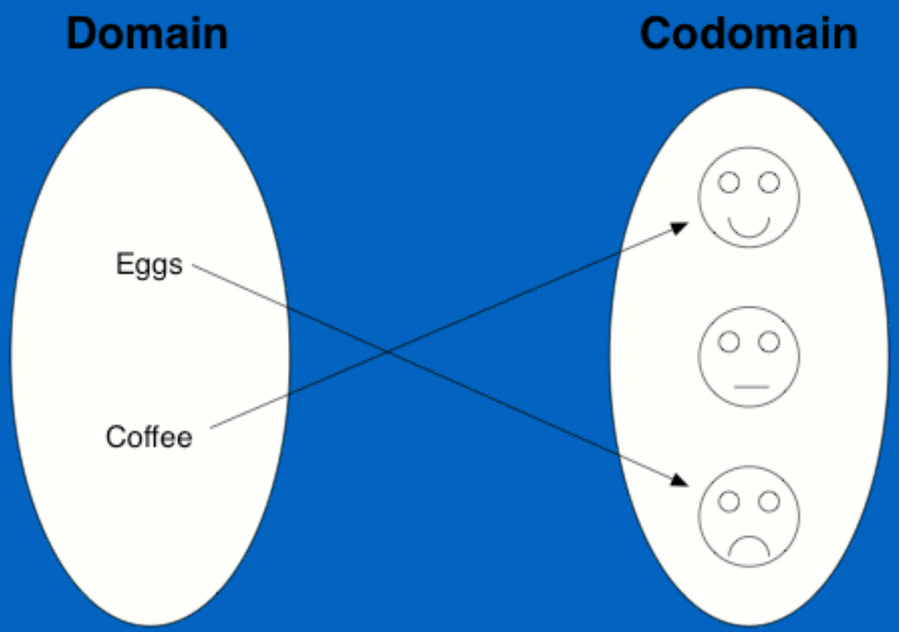
- Functions
- Types, Kinds, & More Functions
- FP Toolbox
- OMG COFFEE BREAK!!!
- Type Classes, Effects
- Scary Sounding Things
- Let's Build a Game!

Functional Programming

It's all about functions.


Domain
Codomain
Eggs
Coffee

john



Food Happiness

Function Definition

```
data Food = Eggs | Coffee

data Happiness = Happy | Neutral | Unhappy

john :: Food -> Happiness
john Eggs   = Unhappy
john Coffee = Happy
```

Function Application

```
> john Eggs

Unhappy

> john Coffee

Happy
```

The Real Deal

1. **Totality**. Every element in *domain* must be mapped to some element in *codomain*.
2. **Determinism**. Applying a function with the same value (in domain) results in same value (in codomain).

Exercises

```
superpower :: CharacterClass -> Superpower

weakness :: Superpower -> Kryptonite
```

1. Create a set called `CharacterClass`, which represents the different types of characters.
2. Create a set called `Superpower`, which represents different superpowers.
3. Create a set called `Kryptonite`, which represents different weaknesses for characters.
4. Create the above functions `superpower` and `weakness`, and apply them at various elements in their domain.

Types

Sets of values.

Literal Types

- `String` : The set that contains all strings; `"foo"` is an element of this set.
- `Number` : The set that contains all numbers;[0] `5.5` is an element of this set.
- `Boolean` : The set that contains the values `true` and `false`.

---

[0] Not really, $%#@&!!

Product Types[1]

```
data Loc = Loc Number Number
```

[1] They get their name from an easy way you can use to compute the size of these sets (hint: product = multiplication).

Product Types

```
  data Loc = Loc Number Number
--        |
--        |
--        |
--   The name of
--    the type.
```

Product Types

```
  data Loc = Loc Number Number
--           |
--           |
--           |
-- The name of a function
-- that will create values
-- of the type. AKA the
--     constructor!
```

Product Types

```
  data Loc = Loc Number Number
--                \     /
--                 \   /
--                  \ /
--       Constructor parameters (types).
```

Product Types

```
data Loc = Loc Number Number

whereAmI = Loc 1 2
```

Product Types

What's the opposite of *con*struction?[4]

```haskell
locX :: Loc -> Number
locX (Loc x _) = x

locY :: Loc -> Number
locY (Loc _ y) = y

locX (Loc 1 2) -- 1
locY (Loc 1 2) -- 2
```

[4] *De*construction, of course! AKA *pattern matching*.

Product Types

Another way to deconstruct.

```
locX :: Loc -> Number
locX l = case l of
           (Loc x _) -> x
```

Exercises

1. Create a `CharacterStats` product type to model some character statistics in an role-playing game (e.g. health, strength, etc.).
2. Create some values of that type to understand how to use data constructors.
3. Use pattern matching to extract individual components out of the data type.

Coproduct Types

(AKA 'Sum' Types)[2]

```
data NPC =
  Ogre String Loc Number |
  Wolf String Loc Number
```

[2] They get their name from an easy way you can use to compute the size of these sets (hint: sum = addition).

Coproduct Types

```
-- The name of
--  the type
--      |
--      |
  data NPC =
    Ogre String Loc Number |
    Wolf String Loc Number
```

Coproduct Types

```
data NPC =
  Ogre String Loc Number |
  Wolf String Loc Number
--   |
--   |
-- Data constructor.
```

Coproduct Types

```
data NPC =
  Ogre String Loc Number |
  Wolf String Loc Number
--          |    |    |
--           \   |   /
--            \ | /
--   Constructor parameters (types).
```

Coproduct Types

Destruction / pattern matching.

```
nameOf :: NPC -> String
nameOf (Ogre name _ _) = name
nameOf (Wolf name _ _) = name
```

Coproduct Types

Deconstruction / pattern matching.

```
data NPC =
  Ogre String Loc Number |
  Wolf String Loc Number

nameOf :: NPC -> String
nameOf npc = case npc of
               (Ogre name _ _) -> name
               (Wolf name _ _) -> name
```

Exercises

1. Create a `Monster` sum type to represent different types of monsters in a game. Make sure they share at least one common piece of information (e.g. `health` or `name`).
2. Create a few monsters of varying types.
3. Create a function to extract out a piece of information common to all constructors.

Record Types[5]

```
data NPC =
  Ogre {name :: String, loc :: Loc, health :: Number} |
  Wolf {name :: String, loc :: Loc, health :: Number}
```

[5] Record types are represented using native Javascript objects.

Record Types

```
  data NPC =
    Ogre {name :: String, loc :: Loc, health :: Number} |
    Wolf {name :: String, loc :: Loc, health :: Number}
--        |                                          |
--        \------------------------|-----------------/
--                               |
--                          Record type.
```

Record Types

```
  data NPC =
    Ogre {name :: String, loc :: Loc, health :: Number} |
    Wolf {name :: String, loc :: Loc, health :: Number}
--         |                                    |
--          \-------------------|---------------------/
--                              |
--                        A 'row' of types.
```

Record Types

```
data NPC =
  Ogre {name :: String, loc :: Loc, health :: Number} |
  Wolf {name :: String, loc :: Loc, health :: Number}
--          |
--       A label.
```

Record Types

```
  data NPC =
    Ogre {name :: String, loc :: Loc, health :: Number} |
    Wolf {name :: String, loc :: Loc, health :: Number}
--                      |
--          The type of the label.
```

Record Types

Construction / deconstruction.

```
makeWolf :: String -> Loc -> Number -> NPC
makeWolf name loc health = Wolf {name: name, loc: loc, health: health}

nameOf :: NPC -> String
nameOf (Ogre { name : n }) = n
nameOf (Wolf { name : n }) = n
```

Record Types

The dot operator.

```
nameOf :: NPC -> String
nameOf (Ogre record) = record.name
nameOf (Wolf record) = record.name
```

Record Types

'Updating' records.

```
changeName :: NPC -> NPC
changeName (Ogre r) = Ogre r { name = "Shrek" }
changeName (Wolf r) = Wolf r { name = "Big Bad" }
```

Record Types

Magic record syntax stuff.

```
(_ { name = "Shrek" }) // Function from record to updated record

record { name = _ }    // Function from string to updated `record`

_ { name = _ }         // Guess? :-)
```

Exercises

1. Rework some of your early product types to use records.
2. Create another class called `InventoryItem` whose constructor takes a record that has fields relevant to items that a player can carry with her.

Basic Function Types

```
data Monster = Giant | Alien
data FavoriteFood = Humans | Kittens

fave :: Monster -> FavoriteFood
fave Giant = Humans
fave Alien = Kittens
```

Basic Function Types

Lambdas AKA closures AKA anonymous functions AKA arrow functions AKA...

```
fave :: Monster -> FavoriteFood
fave = \monster -> ...

var fave = function(monster) {
  ...
}

// ECMAScript 6
var fave = monster => ...
```

Exercises

1. Create a function from monster to total hit points (how much damage they can take before dying).
2. Express the same function as a lambda.
3. Apply the function at various inputs.

Type Aliases

What's in a name?

```
type CharData =
  {name :: String, loc :: Loc, health :: Number}

data NPC = Ogre CharData | Wolf CharData
```

Newtypes

Wrappers without the overhead.

```
newtype Health = Health Number

dead = Health 0
```

Newtypes

Deconstruction / pattern matching.

```
newtype Health = Health Number

isAlive :: Health -> Boolean
isAlive (Health v) = v > 0

isAlive h = case h of
              Health v -> v > 0
```

Exercises

1. Create a type alias for a record called `MagicalItemRec` which has several fields.
2. Use the type alias to define a `newtype` called `MagicalItem`, whose constructor is called `MagicalItem`.
3. Create some values of type `MagicalItem`.
4. Create a few functions to extract out the fields of `MagicalItem`.

Higher-Order Functions

Or, OMG sets can hold functions!!!

Higher-Order Functions

Functions that accept functions.

```
likesEmptyString :: (String -> Boolean) -> Boolean
likesEmptyString f = f ""
```

Higher-Order Functions

Functions that return functions.

```
matches :: String -> (String -> Boolean)
matches v = \text -> text == v

matchesEvil = matches "evil"

matchesEvil "john" -- false
matchesEvil "evil" -- true
```

Higher-Order Functions

"Multi-parameter" functions.[6]

```
damageNpc :: Number -> (NPC -> NPC)
damageNpc damage = \npc -> ...
```

[6] Not *really*, of course: functions in PureScript are always functions *from* one set *to* another set.

Higher-Order Functions

Making sense of "multi-parameter" functions: values.

```
f a b c d e

-- (((((f a) b) c) d) e)
```

Higher-Order Functions

Making sense of "multi-parameter" functions: types.

```
f :: a -> b -> c -> d -> e

-- f :: (a -> (b -> (c -> (d -> e))))
```

Higher-Order Functions

MORE functions that return functions.

```
damageNpc :: Number -> (NPC -> NPC)
damageNpc = \damage -> \npc -> ...

damageNpc :: Number -> (NPC -> NPC)
damageNpc = \damage npc -> ...

damageNpc :: Number -> (NPC -> NPC)
damageNpc damage = \npc -> ...

damageNpc :: Number -> (NPC -> NPC)
damageNpc damage npc = ...
```

Exercises

```
damagerOf :: String -> (NPC -> NPC)
```

```
type Damager = Number -> NPC -> NPC
```

1. Create a function `damagerOf` that takes a name (`String`), and returns another function that damages an `NPC` but only if its name is equal to the specified name.

2. Create a function `boostDamage` which takes a `Damager` (defined above) and returns another `Damager` that boosts the damage done by the passed in damager by 10%.

Parametric Polymorphism

Para..what?

Polymorphic Data

Type constructors: data with "holes".

```
data Map4x4 a = Map4x4 a a a a
                       a a a a
                       a a a a
                       a a a a

boolMap4x4 = Map4x4 true  true  false true
                    false true  true  true
                    false false false true
                    true  false false true
```

Polymorphic Data

Type-level functions.

```
-- invalid :: Map4x4

valid :: Map4x4 Boolean
```

The type constructor `Map4x4` is a function whose domain is the set of all types, and whose codomain is a family of `Map4x4 a` types.

Polymorphic Functions

Or, OMG sets can hold sets!!!

Polymorphic Functions

The heart of functional abstraction.

```
upperLeft :: forall a. Map4x4 a -> a
upperLeft v _ _ _
           _ _ _ _
           _ _ _ _
           _ _ _ _ = v
```

Polymorphic Functions

How to read these crazy signatures.

```
upperLeft :: forall a. Map4x4 a -> a

-- (a :: Type) -> Map4x4 a -> a
```

Exercises

```
data TreasureChest a = ???

isEmpty :: ???
```

1. Create a polymorphic `TreasureChest` sum type that can either contain any type of thing, or be empty.
2. Create a polymorphic function that determines whether or not any treasure chest is empty.

Extensible Rows

Like duck typing only better.

```
type Point r = { x :: Number, y :: Number | r }
```

Extensible Rows

Like duck typing only better.

```
type Point r = { x :: Number, y :: Number | r }
--             |                           |
--             |                           |
--     'remainder'          syntax that means "the rest of the row"

gimmeX :: forall r. Point r -> Number
gimmeX p = p.x

gimmeX {x: 1, y: 2, z: 3} -- 1 - works!

-- gimmeX {x: 1, z: 3}     -- Invalid, no x!
```

Exercises

```
type NonPlayerCharacterRec = ???
type ItemRec = ???
type PlayerCharacterRec = ???

getName :: ???
getName r = r.name
```

1. Create records for `NonPlayerCharacter`, `Item`, and `PlayerCharacter` that all share at least one field (`name`?).

2. Create a function that extracts a name from any record which has *at least* a `name` field of type `String`.

Kinds

Categories of sets.

*

The name for the category of *sets of values*.

(AKA *Type*)

Includes things like:

- `CharacterClass`
- `Superpower`
- `String`

* -> *

The name for the category of *type-level functions*.

(AKA Higher-Kinded Type / Type Constructor)

* -> *

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

* -> *

Type constructors are just (math) functions!

```
addOne :: Number -> Number
addOne n = n + 1

List :: * -> *
data List a = Nil | Cons a (List a)
```

* -> * -> *

Turtles all the way down.

```
Map :: * -> * -> *
data Map k v = ...
```

`(* -> *) -> *`

More turtles.

```
Container :: (* -> *) -> *
data Container f = {create :: forall a. a -> f a}

list :: Container List
list = Container {create: \a -> Cons a Nil}
```

```
* -> * -> * -> * -> * -> *
```

Reading type constructors.

```
foo :: f a b c d e

-- (((((f a) b) c) d) e)
```

!

The name for the category of *sets of effects*.

```
foreign import data DOM :: !
```

`# !`

The name for the category of *rows of effects*.

```
-- Supply a row of effects and a type,
-- and get back another type:
foreign import data Eff :: # ! -> * -> *

trace :: forall r. String -> Eff (trace :: Trace | r) Unit
```

*

The name for the category of *rows of types*.

```
-- Supply a row of types, get back another type:
foreign import data Object :: # * -> *
```

Foreign Types[7]

```
foreign import data jQuery :: *
```

[7] THERE BE DRAGONZ HERE!!!

FP Toolbox

Stuff you couldn't escape even if you wanted to.

FP Toolbox

Maybe it's there, maybe it's not?[8]

```
data Maybe a = Nothing | Just a

type Player =
  { armour :: Maybe Armor }
```

[8] AKA `null`, the FP way.

FP Toolbox

List: the ultimate FP data structure.

```
data List a = Nil | Cons a (List a)
--                          |    |
--                        head   |
--                               tail

oneTwoThree = Cons 1 (Cons 2 (Cons 3 Nil))
```

FP Toolbox

`Either` it's this or it's that.

```
data Either a b = Left a | Right b

type Player =
  { rightHand :: Either Weapon Shield }
```

FP Toolbox

`Tuple`, the opposite of `Either`.[9]

```
data Tuple a b = Tuple a b
--                     | |
--                 first second
I
type Player =
  { wrists :: Tuple (Maybe Bracelet) (Maybe Bracelet) }
```

[9] AKA sometimes it's just too damn hard to name stuff!

FP Toolbox

Native Javascript arrays.

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

Exercises

1. Use *all* the data structures you've learned about (`Maybe`, `Either`, `Tuple`, and `[]`) to build a representation of character state called `CharacterState`.

2. Define a few functions to extract some information out of the data structure.

Type Classes

Generic interfaces, the FP way.

Type Classes

Generic interfaces in Java.

```
public interface Appendable<A> {
  public A append(A a1, A a2);
}
class AppendableNumber extends Appendable<Float> {
  public Float append(Float a1, Float a2) {
    return a1 + a2;
  }
}
Appendable<Float> appendableNumber = new AppendableNumber();
appendableNumber.append(1, 2); // 3!
```

Type Classes

Generic 'interfaces' in Javascript.

```
function makeAppendable(append) {
  return {
    append: append
  };
}

var boolAppendable = makeAppendable(
  function(v1, v2) {
    return v1 && v2;
  }
);

boolAppendable.append(true, false); // false!
```

Type Classes

Generic interfaces in PureScript.

```
class Appendable a where
  append :: a -> a -> a

instance appendableNumber :: Appendable Number where
  append a1 a2 = a1 + a2

append 1 2 -- 3!
```

Type Classes

Turbocharged polymorphism.

```
repeat :: forall a. (Appendable a) => Number -> a -> a
repeat 0 a = a
repeat n a = append (repeat (n - 1) a) a

sort :: forall a. (Ord a) => [a] -> [a]

-- etc.
```

Type Classes

Hierarchies: like OO inheritance, but not.

```
class Eq a where
  equals :: a -> a -> Boolean

data Ordering = LT | GT | EQ

class (Eq a) <= Ord a where
  compare :: a -> a -> Ordering
```

Type Classes

Hierarchies: like OO inheritance, but not.

```
class (Eq a) <= Ord a where
--        |
--        |
-- The superclass.
--
-- Read: "Ord a implies Eq a"
```

Exercises

```
class Describable a where
  describe :: a -> String

data Weapon = Sword | Spear

instance describableWeapon :: ???
```

1. Create an instance of `Describable` for `Weapon`.
2. Create instances of `Eq` (the equal type class) for some of the data types you created.

Effects

Or, how to get in trouble *fast*.

```
import Debug.Trace

main = trace "Hello World!"
```

```
import Debug.Trace

main = do
  trace "Hello World!"

  trace "Bye World!"
```

Exercises

1. Import `Debug.Trace` and make your very own 'Hello World' program.

Scary Sounding Things

Monadic zygohistomorphic prepromorphisms...

WTF?!?!!

Scary Sounding Things

Let's play a game: give your friend a birthday present that she'll adore.

Scary Sounding Things

The rules of the game.

Rule 1: If something is inside a box, you may change it to anything else and the result will still be inside the box.

Rule 2: If something is not inside a box, you can pack it into a box.

Rule 3: If something is packed inside a box which is packed inside another box, you can replace that with a single box containing that thing.

Scary Sounding Things

Your inventory.

Item 1: You have Ripley, a Chihuaha mutt who can magically change a lump of coal into a beautiful present that your friend will like.

Item 2: You have a box containing a box containing a lump of coal.

Which rules should you apply to create a birthday present your friend will adore???

Scary Sounding Things

The rules of the game, redux.

Rule 1: If something is inside a box, you may change it to anything else and the result will still be inside the box.

```
(a -> b) -> f a -> f b
```

Rule 2: If something is not inside a box, you can pack it into a box.

```
a -> f a
```

Rule 3: If something is packed inside a box which is packed inside another box, you can replace that with a single box containing that thing.

```
f (f a) -> f a
```

Scary Sounding Things

The rules of the game, redux redux.

```
fmap :: (a -> b) -> f a -> f b -- AKA (<$>)

pure :: a -> f a                -- AKA return

join :: f (f a) -> f a

-- bind AKA (>>=) = \fa f -> join (fmap f fa)
```

OMG a **monad**, run in terror!!!!!

Nah, just kidding

Scary sounding things give you **rewrite rules** you can use to **manipulate the types** into the form you require.

The scary sounding names **don't matter** at all

Exercises

```
class Evitacilppa f where
  erup :: forall a. a -> f a

  pa :: forall a b. f (a -> b) -> f a -> f b
```

1. You are given `f Number` and `Number`, for some `Evitacilppa f`. If you have a function:

   ```
   add :: Number -> Number -> Number
   ```

   which "rewrite rules" do you need to use so that you can apply the `add` function to the two numbers?

Let's Build a Game!

Enough math already plz!!!

The Soul of an RPG

Or the types, anyway.

```
type Game s i = {
  initial  :: s,
  describe :: s -> String,
  parse    :: String -> Either String i,
  update   :: s -> i -> Either String s }

runGame :: forall s i. Game s i -> Eff (game :: GAME) Unit
runGame g = ...
```

On Your Marks, Get Set, Go!

THANK YOU!

John A. De Goes — @jdegoes

(Do I owe you a coffee?)