Class 2: Algebraic Data Types

January 30

Q: difference between : operator and ++ operator?

A: the first is “cons”, the second is “append”

```
(:) :: Int -> [Int] -> [Int]
```

e.g. `1 : [2, 3]`

```
(++) :: [Int] -> [Int] -> [Int]
```

e.g. `[1, 2] ++ [3, 4]`

questions from homework

Q: can we pattern match on whether an input is negative?

A: yes, using *guards*

questions from homework

see week 1 reading for more details!

```
func :: Int -> Int
func n
  | n < 0 = something
  | otherwise = something else
```

questions from homework

Q: how does indentation work?

A: most of what we will write is single line, and thus left-aligned.
most **multiline things** need to be indented.

questions from homework

Q: why is it hard to pattern match on the end of the list?

A: because we pattern match on the *constructors* of a data type, and the list constructors are “nil” and “cons”

questions from homework

data types by example

note: the weather example is inspired by the video
“Algebraic Data Types (ADT) in Scala | Rock the JVM”

name of the type being defined

```
data Weather
  = Sunny
  | Cloudy
  | Windy
  | Rainy
  | Snowy
```

constructors for values of that type

enumeration types

name of the type being defined

```
data Weather
  = Sunny
  | Cloudy
  | Windy
  | Rainy
  | Snowy
```

the only constructors for values of that type

enumeration types

```
w :: Weather
w = Sunny

ws :: [Weather]
ws = [Sunny, Rainy, Snowy]
```

enumeration types

(example: deriving)

```
isGray :: Weather -> Bool
isGray Sunny  = True
isGray Cloudy = False
isGray Windy  = True
isGray Rainy  = False
isGray Snowy  = True
```

enumeration types

```
isGray :: Weather -> Bool
isGray Cloudy = True
isGray Rainy  = True
isGray w      = False
```

enumeration types

```
isGray :: Weather -> Bool
isGray Cloudy = True
isGray Rainy  = True
isGray _      = False
```

“wildcard” pattern

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

enumeration types

```
or :: Bool -> Bool -> Bool
or True  True  = True
or True  False = True
or False True  = True
or False False = False
```

enumeration types

```
or :: Bool -> Bool -> Bool
or False False = False
or _     _     = True
```

enumeration types

```
or :: Bool -> Bool -> Bool
or True  _  = True
or False b  = b
```

enumeration types

```
data WeatherRequest
  = ByLocation String
  | ByCoordinate Int Int
```

constructors that take arguments

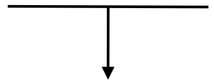
please follow along!

```
predict :: WeatherRequest -> Weather
predict (ByLocation “Philly”) = Sunny
predict (ByCoordinate 90 _)   = Snowy
```

pattern matching can use *literals*

please follow along!

```
predict :: WeatherRequest -> Weather
predict (ByLocation “Philly”) = Sunny
predict (ByCoordinate 90 _)   = Snowy
```



pattern matching can also be *nested*

```
data WeatherForecast
  = Forecast Weather
  | Unknown
```

more data types

please follow along!

```
predict :: WeatherRequest -> WeatherForecast
predict (ByLocation “Philly”) = Forecast Sunny
predict (ByCoordinate 90 _)   = Forecast Snowy
predict _                     = Unknown
```

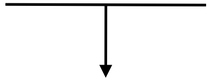
```
ByLocation :: String -> WeatherRequest

ByCoordinate :: Int -> Int -> WeatherRequest
```

more data types

(polls: types for WeatherForecast)

```
data Coord = Coord Int Int
```



idiom for single-constructor types

```
c :: Coord
c = Coord 90 0

reflect :: Coord -> Coord
reflect (Coord x y) = Coord y x
```

(exercises: add and incorporate)

important: see the reading for sections on "Pattern Matching" and "Case Expressions"

interlude: why “algebraic”?

```
data Bool = False | True

data Color = Red | Green | Blue
```

```
data Mix = Mix Bool Color Color
```

number of values of this type: 2 x 3 x 3

product types

```
data Unit = Unit
```

1

product types

```
data Mix

  = Black                          1
  | Single Color                   + 3
  | Double Color Color             + 9
  | White                          + 1
```

sum types

```
data Empty    0
```

sum types

```haskell
data Shape
  = Rectangle Int Int
  | Circle Int
```

```haskell
area :: Shape -> Double
area (Rectangle x y) = …
area (Circle r) = …
```

```java
interface Shape
  double area();
```

```java
class Rectangle implements Shape
  int x, y;
  double area() { … }
```

```java
class Circle implements Shape
  int r;
  double area() { … }
```

readiness for change

```
data Shape
  = Rectangle Int Int
  | Circle Int
```

```
area :: Shape -> Double
area (Rectangle x y) = …
area (Circle r) = …
```

```
perimeter :: Shape -> Int
```

```
interface Shape
  double area();
  int perimeter();
```

```
class Rectangle implements Shape
  int x, y;
  double area() { … }
```

```
class Circle implements Shape
  int r;
  double area() { … }
```

readiness for change

```
data Shape
  = Rectangle Int Int
  | Circle Int
  | Triangle Int Int Int
```

```
area :: Shape -> Double
area (Rectangle x y) = …
area (Circle r) = …
```

```
interface Shape
  double area();
```

```
class Rectangle implements Shape
  int x, y;
  double area() { … }
```

```
class Circle implements Shape
  int r;
  double area() { … }
```

```
class Triangle implements Shape
```

readiness for change

recursive data types

```
data IntList
  = Nil
  | Cons Int IntList
```

data types can defined in terms of themselves!

```
prod :: IntList -> Int
prod Nil = 1
prod (Cons x xs) = x * prod xs
```

```
data Tree
  = Leaf
  | Node Tree Int Tree
```

recursive types

```
tree :: Tree
tree = Node Leaf 1 (Node Leaf 2 Leaf)
```

recursive types

(exercises: size and add3)