Functional programming

Making own data types

Data declarations

- One introduces, or declares, a type in Haskell via the data statement.
- Use the keyword **data**

```
data [context =>] type tv1 ... tvi = con1  c1t1 c1t2... c1tn |
                                ... | conm cmt1 ... cmtq
                              [deriving]
```

- Supply an optional context
- Give the type name and a variable number of type variables.
- This is then followed by a variable number of constructors, each of which has a list of type variables or type constants.
- At the end, there is an optional deriving.

Data declarations

- data Bool = False | True
- data Maybe a = Just a | Nothing
- data Tree a = Branch (Tree a) (Tree a) | Leaf a
- data Shape = Circle Float Float Float |
  Rectangle Float Float Float Float deriving (Show)

Making a record


```haskell
data Person = Person String String Int Float deriving (Show)

firstName :: Person -> String
firstName (Person firstname _ _ _) = firstname

lastName :: Person -> String
lastName (Person _ lastname _ _) = lastname

age :: Person -> Int
age (Person _ _ age _) = age

height :: Person -> Float
height (Person _ _ _ height) = height
```

```
λ: let me = Person "Roman" "Str" 28 1.81
λ: firstName me
=> "Roman"
λ: age me
=> 28
```

Looks nice, doesn't it?

Making a record

```haskell
data Person = Person { firstName :: String
                     , lastName :: String
                     , age :: Int
                     , height :: Float
                     } deriving (Show)
```

```
λ: :t Person
Person :: String -> String -> Int -> Float -> Person
```

```
λ: let me = Person "Roman" "Str" 28 1.81
λ: me
=> Person {firstName = "Roman", lastName = "Str", age = 28, height = 1.81}
```

```
λ: let me = Person {firstName = "Roman", lastName = "Str", age = 28, height = 1.81}
λ: me
=> Person {firstName = "Roman", lastName = "Str", age = 28, height = 1.81}
```

And now?

Type synonyms

- A type synonym is a new name for an existing type. Values of different synonyms of the same type are entirely compatible. In Haskell you can define a type synonym using type:
  - type String = [Char]
  - type MyChar = Char

```
type Name = String
type LastName = String
type Age = Int
type Height = Float

data Person = Person { firstName :: Name
                     , lastName :: LastName
                     , age :: Age
                     , height :: Height
                     } deriving (Show)
```

```
λ> :t Person
Person :: Name -> LastName -> Age -> Height -> Person
```

New types


```haskell
newtype Name = Name String deriving (Show)
newtype LastName = LastName String deriving (Show)
newtype Age = Age Int deriving (Show)
newtype Height = Height Float deriving (Show)


data Person = Person { firstName :: Name
                     , lastName :: LastName
                     , age :: Age
                     , height :: Height
                     } deriving (Show)
```

`data` can only be replaced with **`newtype`** if the type has *exactly one constructor* with *exactly one field* inside it

```
λ: let me = Person (Name "Roman") (LastName "Str") (Age 28) (Height 1.81)
λ: me
=> Person {firstName = Name "Roman", lastName = LastName "Str", age = Age 28, height =
Height 1.81}
```

Deriving

- Read - parsing of strings, producing values
- Show - conversion of values to readable strings
- Eq - defines equality (==) and inequality (/=)
- Ord - used for totally ordered datatypes
- Enum - defines operations on sequentially ordered types

A deck of cards


```
data Suit = Club | Diamond | Heart | Spade
   deriving (Read, Show, Enum, Eq, Ord)

data CardValue = Two | Three | Four
   | Five | Six | Seven | Eight | Nine | Ten
   | Jack | Queen | King | Ace
  deriving (Read,  Show, Enum, Eq, Ord)

data Card = Card {value :: CardValue,
                  suit :: Suit}
  deriving (Read, Show, Eq)

type Deck = [Card]
deck::Deck
deck = [Card val su | val <- [Two .. Ace], su <- [Club .. Spade]]
```

```
λ: take 3 deck
=> [Card {value = Two, suit = Club},Card {value = Two, suit = Diamond},Card
{value = Two, suit = Heart}]
```