Programming In Haskell Chapter 6

CS 1JC3

Algebraic Datatypes Recap

How to define data structures? We need to have a way of defining new values and structure. ADT's in Haskell are constructed using:

Data Constructors - Defines the name of your new type, consider the data constructor TypeName

```
data TypeName = ...
someFunc :: Int -> TypeName
```

Value Constructors - Allow you to define new values and wrap other values. Consider the value constructors Type1,Type2

```
data TypeName = Type1 | Type2 Int
someFunc x = if x == 0 then Type1 else (Type2 x)
```

Algebraic Datatypes by Example

Product Types - used to group values together into a single value, for example

```
data StudentID = StudentID String String
macID (StudentID mID _) = mID
studentNum (StudentID _ sID) = sID
```

Algebraic Datatypes by Example

Product Types - used to group values together into a single value, for example

```haskell
data StudentID = StudentID String String
macID (StudentID mID _) = mID
studentNum (StudentID _ sID) = sID
```

Record Syntax - special syntax feature

```haskell
data StudentID = StudentID { macID :: String,
                             studentNum :: String }
```

provides the same functionality with less code and more descriptive

Algebraic Datatypes by Example

Sum Types - used to create separate, distinguishable values under the same type, for example

```
data UniversityID = StudentID {macID :: String,
                               studentNum :: String}
                  | FacultyID {macID :: String,
                               facultyNum :: String,
                               salary :: Float }
```

Algebraic Datatypes by Example

Sum Types - used to create separate, distinguishable values under the same type, for example

```
data UniversityID = StudentID {macID :: String,
                               studentNum :: String}
                  | FacultyID {macID :: String,
                               facultyNum :: String,
                               salary :: Float }
```

Makes extracting information simpler through pattern matching

```
debt :: UniversityID -> Float
debt (StudentID _ _) = 100000.0
debt (FacultyID _ _ sal) = 2.0 * sal
```

Algebraic Datatypes by Example

Recursive Types - allow you to construct types whose structure can expand infinitely, for example

```
data IntList = Cons Int IntList
             | Empty
```

Algebraic Datatypes by Example

Recursive Types - allow you to construct types whose structure can expand infinitely, for example

```
data IntList = Cons Int IntList
             | Empty
```

Is a way of encoding lists of Int. For example, you could encode the list [1,2,3] as

```
myList = Cons 1 (Cons 2 (Cons 3 Empty))
```

Algebraic Datatypes by Example

Polymorphic Data Types - allow you to construct a type that varies over another type, for example

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

Algebraic Datatypes by Example

Polymorphic Data Types - allow you to construct a type that varies over another type, for example

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

Note: the type that's varied is given as a parameter to the data constructor

```
myIntList :: List Int
myIntList = Cons 1 (Cons 2 (Cons 3 Empty))

myBoolList :: List Bool
myBoolList = Cons False (Cons True (Cons False Empty))
```

Important Detail

Put deriving Show under your data type definitions

```haskell
data MyDataType = Type1 | Type2
    deriving Show
```

if you plan on running your code ghci (So.. just always put deriving Show after your data type definitions).

Important Detail

Put deriving Show under your data type definitions

```
data MyDataType = Type1 | Type2
     deriving Show
```

if you plan on running your code ghci (So.. just always put deriving Show after your data type definitions).

The reason for this is ghci relies on the Show type class,

```
class Show a where
     show :: a -> String
```

which converts values to their "String form", to output values. Using the deriving keyword makes a generic instance of the class for you

Case Syntax

The case syntax provides another means of pattern matching, particularly useful when the value you want to pattern match isn't a parameter. For example

```
data Lights = Red | Green | Yellow

nextLight Red    = Green
nextLight Yellow = Red
nextLight Green  = Yellow
```

Case Syntax

The case syntax provides another means of pattern matching, particularly useful when the value you want to pattern match isn't a parameter. For example

```
data Lights = Red | Green | Yellow

nextLight Red    = Green
nextLight Yellow = Red
nextLight Green  = Yellow
```

can also be defined as

```
nextLight light = case light of
                    Red    -> Green
                    Yellow -> Red
                    Green  -> Yellow
```

List Pattern Matching

The list type is of the form

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

where Cons is (:) and Empty is [ ]

List Pattern Matching

The list type is of the form

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

where Cons is (:) and Empty is [ ]

Therefore, we pattern match on list like so

```
func (x:xs) = ... vs func (Cons x xs) = ...
func []     = ... vs func Empty       = ...
```

Iterating Through a List

Example, implementing the length function

```haskell
length (x:xs) = 1 + length xs    -- Recursive Case
length []     = 0                -- Base Case
```

Iterating Through a List

Example, implementing the length function

```
length (x:xs) = 1 + length xs    -- Recursive Case
length []     = 0                -- Base Case
```

Example Evaluation

```
    length [1,2]
=   length (1:[2])
=   1 + length [2]
=   1 + length (2:[])
=   1 + (1 + length [])
=   1 + (1 + 0)
=   2
```

Iterating Through a List

Example, implementing the map function

```
map :: (a -> b) -> [a] -> [b]
map f (x:xs) = (f x) : (map f xs) -- Recursive Case
map f []     = []                 -- Base Case
```

Example, implementing the map function

```
map :: (a -> b) -> [a] -> [b]
map f (x:xs) = (f x) : (map f xs) -- Recursive Case
map f []     = []                 -- Base Case
```

Example Evaluation

```
    map (\x -> x+1) [1,2]
=   map (\x -> x+1) (1:[2])
=   (1+1):(map (\x -> x+1) (2:[]))
=   (1+1):((1+2):(map (\x -> x+1) []))
=   (1+1):((1+2):[])
=   [1+1,1+2]
=   [2,3]
```

Polymorphic Lists

Whats the type of length?

```
length :: [Bool] -> Int ?
length :: [[Float] -> Int ?
```

Polymorphic Lists

Whats the type of length?

```
length :: [Bool] -> Int ?
length :: [[Float] -> Int ?
```

The length function works on any of those types, in fact it works on all types

```
length :: [a] -> Int
```

Note: this is the same as any other polymorphic data type, where [ ] is the data constructor with the type a as the parameter enclosed inside instead of proceeding

Installing QuickCheck

QuickCheck is a powerful tool for testing your programs. In order to use it, you first need to install it through cabal. Open a terminal and enter the following

```
cabal install quickcheck
```

Then to use QuickCheck in your Haskell file add the import to the top of the file

```
import Test.QuickCheck
```

What is QuickCheck?

The standard documentation for QuickCheck is located on Hackage at
https://hackage.haskell.org/package/QuickCheck-2.10.1/docs/Test-QuickCheck.html

and a manual at
http://www.cse.chalmers.se/~rjmh/QuickCheck/manual.html

A variety of functions are provided for constructing tests in Test.QuickCheck but the only one we'll concern ourself with is

```
quickCheck :: Testable prop => prop -> IO ()
```

What is QuickCheck?

QuickCheck takes an argument of the type class Testable and outputs results to the standard output (i.e IO)

```
quickCheck :: Testable prop => prop -> IO ()
```

What is QuickCheck?

QuickCheck takes an argument of the type class Testable and outputs results to the standard output (i.e IO)

```
quickCheck :: Testable prop => prop -> IO ()
```

The Testable typeclass has many instances, but the only one we'll concern ourselves with are boolean functions of the form

```
prop :: (Arbitrary a, Show a) => a -> Bool
```

Which works for almost any built-in type a (i.e defined in Prelude), including lists and tuples

Using QuickCheck By Example

We test our functions with QuickCheck by defining boolean properties that must hold to be correct. Consider

```
myAbs x = if x < 0 then -x else x
```

has the very obvious and simple property to check

Using QuickCheck By Example

We test our functions with QuickCheck by defining boolean properties that must hold to be correct. Consider

```haskell
myAbs x = if x < 0 then -x else x
```

has the very obvious and simple property to check

```haskell
absProp x = myAbs x > 0
```

check the property holds with quickCheck by running the function

```haskell
testMyAbs = quickCheck absProp
```

As another example, consider the function

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

has the perhaps less obvious property

Using QuickCheck By Example

As another example, consider the function

```
sum []     = 0
sum (x:xs) = x + sum xs
```

has the perhaps less obvious property

```
sumProp (xs,ys) = sum xs + sum ys == sum (xs ++ ys)
```

Note: sumProp takes a tuple, because it needs to be a function that takes one argument and returns a Bool to work with QuickCheck

Exercise 1

The zip function

```
zip :: [a] -> [b] -> [(a,b)]
```

takes two lists and combines them into one list of tuples of corresponding elements. Implement your own zip,

Note: zip already exists in the Prelude, so call it something like myZip

Solution 1

```
myZip :: [a] -> [b] -> [(a,b)]

myZip (x:xs) (y:ys) = (x,y) : myZip xs ys
myZip  _      _     = []
```

Exercise 2

Implement the function

```
mapWithIndex :: ((Int,a) -> b) -> [a] -> [b]
```

mapWithIndex is just like map, however it takes a function that takes a tuple with the corresponding index of the element in list

Solution 2

```
mapWithIndex :: ((Int,a) -> b) -> [a] -> [b]

mapWithIndex f xs = let
   mapWithIndex' f (x:xs) n =
               f (n,xs) : mapWithIndex' f xs (n+1)
   mapWithIndex' _ []      _ = []
  in mapWithIndex' f xs 0
```

Exercise 3

Define your own list type, and then implement your own versions of the following functions on it

- the sum function

```
mySum :: (Num a) => List a -> a
```

- the (++) function for combining lists

```
(+++) :: List a -> List a -> List a
```

- the reverse function

```
myReverse :: List a -> List a
```

Solution 3

```haskell
data List a = Cons a (List a) | Empty
    deriving Show

mySum Empty         = 0
mySum (Cons x xs) = 1 + mySum xs

Empty          +++ xs = xs
(Cons y ys) +++ xs = Cons y (ys +++ xs)

myReverse Empty = Empty
myReverse (Cons x xs) = (myReverse xs) +++ (Cons x Empty)
```

Exercise 4

Define a binary tree type, and then implement your own versions of the following functions on it

- the sum function

```
treeSum :: (Num a) => Tree a -> a
```

- the height function

```
treeHeight :: (Num a,Ord a) => Tree a -> a
```

Solution 4

```haskell
data Tree a = Node a (Tree a) (Tree a) | Leaf a

treeSum (Leaf x) = x
treeSum (Node x t1 t2) = x + treeSum t1 + treeSum t2

treeHeight (Leaf _) = 1
treeHeight (Node x t1 t2) = 1 + max (treeHeight t1)
                                    (treeHeight t2)
```

Exercise 5

Implement your own versions of the take and drop functions

```
take :: Int -> [a] -> [a]
drop :: Int -> [a] -> [a]
```

and write a quickCheck property to **test both simultaneously** (i.e check one in terms of the other)

Solution 5

```
myTake _ []     = []
myTake n (x:xs) = if n <= 0
                    then []
                    else x : myTake (n-1) xs

myDrop _ []     = []
myDrop n (x:xs) = if n <= 0
                    then (x:xs)
                    else myDrop (n-1) xs

takedropProp (xs,n) = myTake n xs == (reverse
                                   $ myDrop (length xs - n)
                                   $ reverse xs)
```