BAHUG 101 - Lecture 3

7th October 2015

Outline of Today's Lecture

- Libraries
- Enumerations
- Algebraic data types
- Polymorphic data types
- Recursive data types

Quick notes on libraries

Base Haskell

Haskell comes with a lot of functions in a standard library called `base`.

Base Haskell

Haskell comes with a lot of functions in a standard library called `base`.

Libraries are structured as so.

```
libraries -> modules
base      -> Data.Char
```

Packages are used to distribute libraries.

Base Haskell

Haskell comes with a lot of functions in a standard library called **`base`**.

Libraries are structured as so.

```
libraries -> modules
base      -> Data.Char
```

Packages are used to distribute libraries.

When using an import statement, we are importing a **`module`** from a

```haskell
import Data.Char ( toUpper )
```

or

```haskell
import Data.Char -- Imports everything exported by Data.Char
```

Grabbing new packages from Hackage

Often we want to import code that someone else has written and distributed. For that we can use **cabal**.

```
cabal update
cabal install text-1.1.1.3
```

Sometimes this can lead to *cabal hell*. To alleviate this, use cabal sandboxes or Stackage.

Enumeration Types

Simple Enumerations

In Haskell we can create a type for an enumeration of things as so.

```
data Thing = Shoe
           | Ship
           | SealingWax
           | Cabbage
           | King
  deriving Show -- tells Haskell to make a function
                -- for `Thing -> String`
```

A value of type `Thing` can be `Shoe` or `Ship` or `SealingWax`, etc.

Using values of an enumeration

Some simple uses of `Thing` include an explicit value

```
shoe :: Thing
shoe = Shoe
```

Using values of an enumeration

Some simple uses of `Thing` include an explicit value

```
shoe :: Thing
shoe = Shoe
```

and a list of `Thing`

```
listO'Things :: [Thing]
listO'Things = [Shoe, SealingWax, King, Cabbage, King]
```

Pattern Match on Thing

We can pattern match on the data constructors when creating a function.

```
isSmall :: Thing → Bool
isSmall Shoe       = True
isSmall Ship       = False
isSmall SealingWax = True
isSmall Cabbage    = True
isSmall King       = False
```

Pattern Match on Thing

We can pattern match on the data constructors when creating a function.

```
isSmall :: Thing -> Bool
isSmall Shoe       = True
isSmall Ship       = False
isSmall SealingWax = True
isSmall Cabbage    = True
isSmall King       = False
```

or more simply as the following, using the order of cases to our advantage.

```
isSmall2 :: Thing -> Bool
isSmall2 Ship = False
isSmall2 King = False
isSmall2 _    = True
```

Data constructors can take arguments

Data constructors can also take any number of arguments.

```
data FailableDouble = Failure
                    | OK Double
  deriving Show
```

Data constructors can take arguments

Data constructors can also take any number of arguments.

```
data FailableDouble = Failure
                    | OK Double
  deriving Show
```

To create a value of type `FailableDouble`, we can do either

```
ex01 :: FailableDouble
ex01 = Failure
```

Data constructors can take arguments

Data constructors can also take any number of arguments.

```
data FailableDouble = Failure
                    | OK Double
  deriving Show
```

To create a value of type `FailableDouble`, we can do either

```
ex01 :: FailableDouble
ex01 = Failure
```

or

```
ex02 :: FailableDouble
ex02 = OK 3.4
```

Data constructors can take arguments

Data constructors can also take any number of arguments.

```
data FailableDouble = Failure
                    | OK Double
  deriving Show
```

To create a value of type `FailableDouble`, we can do either

```
ex01 :: FailableDouble
ex01 = Failure
```

or

```
ex02 :: FailableDouble
ex02 = OK 3.4
```

What is the type of `OK`?

Data constructors can take arguments

Data constructors can also take any number of arguments.

```
data FailableDouble = Failure
                    | OK Double
  deriving Show
```

To create a value of type `FailableDouble`, we can do either

```
ex01 :: FailableDouble
ex01 = Failure
```

or

```
ex02 :: FailableDouble
ex02 = OK 3.4
```

What is the type of `OK`?

```
Double -> FailureDouble
```

Examples functions using **FailableDouble**

Let's use our **FailableDouble** to make a safe version of division.

```
safeDiv :: Double -> Double -> FailableDouble
safeDiv _ 0 = Failure
safeDiv x y = OK (x / y)
```

Examples functions using `FailableDouble`

Let's use our `FailableDouble` to make a safe version of division.

```
safeDiv :: Double → Double → FailableDouble
safeDiv _ 0 = Failure
safeDiv x y = OK (x / y)
```

We can also pattern match on `FailableDouble`.

```
failureToZero :: FailableDouble → Double
failureToZero Failure = 0
failureToZero (OK d)  = d
```

Data constructors can have more than one argument

Data constructors can take any number of inputs.

```
data Person = Person String Int Thing
  deriving Show

richard :: Person
richard = Person "Richard" 32 Ship

stan :: Person
stan  = Person "Stan" 94 Cabbage

getAge :: Person -> Int
getAge (Person _ a _) = a
```

Algebraic Data Types

The general structure for a data type is as follows.

```
data AlgDataType = Constr1 Type11 Type12
                 | Constr2 Type21
                 | Constr3 Type31 Type32 Type33
                 | Constr4
```

Type constructors and data constructors must start with a capital letter; values must start with a lower case (this includes functions).

Pattern Matching

Pattern matching is used to determine *which data constructor created a value.*

Pattern Matching

Pattern matching is used to determine *which data constructor created a value*.

For the prior **`AlgDataType`**, pattern matching looks like so.

```
foo (Constr1 a b)   = ...
foo (Constr2 a)     = ...
foo (Constr3 a b c) = ...
foo Constr4         =
```

Pattern Matching extra syntax

An underscore _ represent a “match anything”, but you cannot use the value.

Pattern Matching extra syntax

An underscore _ represent a "match anything", but you cannot use the value.

A pattern in the form **`x@pat`** can match the entire constructor and the values inside of it.

```
baz :: Person → String
baz p@(Person n _ _) = "The name of the (" ++ show p ++
                       ") is " ++ n
```

Pattern Matching extra syntax

An underscore _ represent a "match anything", but you cannot use the value.

A pattern in the form **x@pat** can match the entire constructor and the values inside of it.

```
baz :: Person -> String
baz p@(Person n _ _) = "The name of the (" ++ show p ++
                       ") is " ++ n
```

```
*Main> baz richard
"The name field of (Person "Richard" 32 Ship) is Richard""
```

Nested Patterns

Patterns can also be nested.

```
checkFav :: Person -> String
checkFav (Person n _ Ship) = n ++ ", you're my kind of person!"
checkFav (Person n _ _)    = n ++ ", your favorite thing is lame."
```

Nested Patterns

Patterns can also be nested.

```
checkFav :: Person -> String
checkFav (Person n _ Ship) = n ++ ", you're my kind of person!"
checkFav (Person n _ _)    = n ++ ", your favorite thing is lame."

*Main> checkFav richard
"Richard, you're my kind of person!"
```

Nested Patterns

Patterns can also be nested.

```
checkFav :: Person -> String
checkFav (Person n _ Ship) = n ++ ", you're my kind of person!"
checkFav (Person n _ _)    = n ++ ", your favorite thing is lame."

*Main> checkFav richard
"Richard, you're my kind of person!"

*Main> checkFav stan
"Stan, your favorite thing is lame."
```

General pattern matching grammar

The general grammar for pattern matching is as follows.

```
pat ::= _
      | var
      | var @ ( pat )
      | ( Constructor pat1 pat2 ... patn )
```

Case expressions

Case expressions are the general form of pattern matching, and can be used on any value.

```
case exp of
  pat1 -> exp1
  pat2 -> exp2
  ...
```

Case expressions

Case expressions are the general form of pattern matching, and can be used on any value.

```
case exp of
  pat1 -> exp1
  pat2 -> exp2
  ...
```

As an example:

```
ex03 = case "Hello" of
         []      -> 3
         ('H':s) -> length s
         _       -> 7
```

Regular pattern matching is syntactic sugar

We can write all of our functions with case statements instead of pattern matching.

```haskell
failureToZero' :: FailableDouble → Double
failureToZero' x = case x of
                     Failure → 0
                     OK d    → d
```

Polymorphic data types

Currently we have a redundancy

Here, as a preview, are some (similar) data structures in homework 3.

```
data LogMessage = LogMessage Int String

data MaybeLogMessage = ValidLM LogMessage
                     | InvalidLM

data MaybeInt = ValidInt Int
              | InvalidInt
```

Currently we have a redundancy

Here, as a preview, are some (similar) data structures in homework 3.

```
data LogMessage = LogMessage Int String

data MaybeLogMessage = ValidLM LogMessage
                     | InvalidLM

data MaybeInt = ValidInt Int
              | InvalidInt
```

Hmm, those last two are pretty similar.

Maybe we can do better!

Haskell allows you to add **type variables** to stand in for a type.

```
data Maybe a = Just a
             | Nothing
```

Maybe we can do better!

Haskell allows you to add **`type variables`** to stand in for a type.

```
data Maybe a = Just a
             | Nothing
```

Note that **`Maybe`** is a *type constructor*, but **`Maybe Int`** is a type.

Brief intro to kinds

When we talk about the “type of a type”, we call this a *kind*. We can probe the kind of a type/type constructor by using `:k` in `ghci`.

```
*Main> :k Maybe
Maybe :: * -> *

*Main> :k Maybe Int
Maybe Int :: *
```

All type annotations must have kind `*`.

Maybe examples

As an example, we can take an "undefined" value and turn it into some default.

```
example_a :: Maybe Int -> Int
example_a (Just n) = n
example_a Nothing  = (-1)
```

Maybe examples

As an example, we can take an "undefined" value and turn it into some default.

```
example_a :: Maybe Int → Int
example_a (Just n) = n
example_a Nothing  = (-1)
```

Or we can check a log message to see if the severity is high enough to warrent a message.

```
example_b :: LogMessage → Maybe String
example_b (LogMessage severity s) | severity ⩾ 50 = Just s
example_b _                                       = Nothing
```

Recursive data types

Data types can be recursive, meaning that they are defined in terms of themselves. We have seen this already with `List` (also written as `[]`).

```
data List t = Empty | Cons t (List t)
  deriving Show
```

Recursive data types

Data types can be recursive, meaning that they are defined in terms of themselves. We have seen this already with `List` (also written as `[]`).

```
data List t = Empty | Cons t (List t)
 deriving Show
```

Here are some example creating lists using our `List t` type.

```
lst1 :: List Int
lst1 = Cons 3 (Cons 5 (Cons 2 Empty))
```

Recursive data types

Data types can be recursive, meaning that they are defined in terms of themselves. We have seen this already with **List** (also written as **[]**).

```
data List t = Empty | Cons t (List t)
 deriving Show
```

Here are some example creating lists using our **List t** type.

```
lst1 :: List Int
lst1 = Cons 3 (Cons 5 (Cons 2 Empty))

lst1' :: [] Int
lst1' = (:) 3 ((:) 5 ((:) 2 [])) — normally written [3, 5, 2]
```

Recursive data types

Data types can be recursive, meaning that they are defined in terms of themselves. We have seen this already with **List** (also written as **[]**).

```
data List t = Empty | Cons t (List t)
 deriving Show
```

Here are some example creating lists using our **List t** type.

```
lst1 :: List Int
lst1 = Cons 3 (Cons 5 (Cons 2 Empty))

lst1' :: [] Int
lst1' = (:) 3 ((:) 5 ((:) 2 [])) — normally written [3, 5, 2]

lst2 :: List Char
lst2 = Cons 'x' (Cons 'y' (Cons 'z' Empty))
```

Recursive data types

Data types can be recursive, meaning that they are defined in terms of themselves. We have seen this already with `List` (also written as `[]`).

```
data List t = Empty | Cons t (List t)
 deriving Show
```

Here are some example creating lists using our `List t` type.

```
lst1 :: List Int
lst1 = Cons 3 (Cons 5 (Cons 2 Empty))

lst1’ :: [] Int
lst1’ = (:) 3 ((:) 5 ((:) 2 [])) — normally written [3, 5, 2]

lst2 :: List Char
lst2 = Cons ’x’ (Cons ’y’ (Cons ’z’ Empty))

lst3 :: List Bool
lst3 = Cons True (Cons False Empty)
```

We can use recursive functions to process recursive types

Often when we want to process a recursive data type, we will use a recursive function.

```
intListProd :: List Int → Int
intListProd Empty      = 1
intListProd (Cons x l) = x * intListProd l
```

Tree Example

One way to make a binary tree data structure would be like so.

```
data Tree = Leaf Char
          | Node Tree Int Tree
  deriving Show
```

Tree Example

One way to make a binary tree data structure would be like so.

```haskell
data Tree = Leaf Char
          | Node Tree Int Tree
  deriving Show
```

And then we could make some random tree.

```haskell
tree :: Tree
tree = Node (Leaf 'x') 1 (Node (Leaf 'y') 2 (Leaf 'z'))
```