PROGRAMMING IN HASKELL

Chapter 7.2 – Modules in Haskell*

*From ‘Learn you a Haskell’ (http://learnyouahaskell.com/)

The Haskell prelude

So far, we've been using built-in functions provided in the Haskell prelude. This is a subset of a larger library that is provided with any installation of Haskell. (Google for Hoogle to see a handy search engine for these.)

Examples of other modules:

- lists
- concurrent programming
- complex numbers
- char
- sets
- …

Example: Data.List

To load a module, we need to import it:

```haskell
import Data.List
```

All the functions in this module are immediately available:

```haskell
numUniques :: (Eq a) => [a] -> Int
numUniques = length . nub
```

This is a function in Data.List that removes duplicates from a list.

You can also load modules from the command prompt:

```
ghci> :m + Data.List
```

Or several at once:

```
ghci> :m + Data.List Data.Map Data.Set
```

Or import only some, or all but some:

```
import Data.List (nub, sort)    OR
import Data.List hiding (nub)
```

If duplication of names is an issue, we can extend the namespace:

```
import qualified Data.Map
Data.Map.filter isUpper .......
```

When the Data.Map gets a bit long, we can provide an alias:

```
import qualified Data.Map as M
```

And now we can just type M.filter, and the normal list filter will just be filter.

Data.List has a lot more functionality than we've seen. A few examples:

```
ghci> intersperse '.' "MONKEY"
"M.O.N.K.E.Y"
ghci> intersperse 0 [1,2,3,4,5,6]
[1,0,2,0,3,0,4,0,5,0,6]
```

```
ghci> intercalate " " ["hey","there","guys"]
"hey there guys"
ghci> intercalate [0,0,0] [[1,2,3],[4,5,6],
                                              [7,8,9]]
[1,2,3,0,0,0,4,5,6,0,0,0,7,8,9]
```

And even more:

```
ghci> transpose [[1,2,3],[4,5,6],
                                 [7,8,9]]
[[1,4,7],[2,5,8],[3,6,9]]
ghci> transpose ["hey","there","guys"] ["
htg","ehu","yey","rs","e"]
```

```
ghci> concat ["foo","bar","car"]
"foobarcar"
ghci> concat [[3,4,5],[2,3,4],[2,1,1]]
[3,4,5,2,3,4,2,1,1]
```

And even more.. (you can come back to this later)

```
ghci> and $ map (>4) [5,6,7,8]
True
ghci> and $ map (==4) [4,4,4,3,4]
False
```

```
ghci> any (==4) [2,3,5,6,1,4]
True
ghci> all (>4) [6,9,10]
True
```

A nice example: adding functions

Functions are often represented as vectors: 8x^3 + 5x^2 + x - 1 is [8,5,1,-1].

So we can easily use List functions to add these vectors:

```
ghci> map sum ( transpose [[0,3,5,9],
          [10,0,0,9],[8,5,1,-1]]
[18,8,6,17])    -- we can use $ here ..
later
```

The Data.Char module: includes a lot of useful functions that may look familiar.

Examples: isAlpha, isLower, isSpace, isDigit, isPunctuation,…

```
ghci> all isAlphaNum "bobby283"
True
ghci> all isAlphaNum "eddy the fish!”
False
ghci> group By ((==) `on` isSpace)
          "hey guys its me"
["hey"," ","guys"," ","its"," ","me"]
```

The Data.Char module has a datatype that is a set of comparisons on characters. There is a function called generalCategory that returns the information. (This is a bit like the Ordering type for numbers, which returns LT, EQ, or GT.)

```
ghci> generalCategory ' '
Space
ghci> generalCategory 'A'
UppercaseLetter
ghci> generalCategory 'a'
LowercaseLetter
ghci> generalCategory '.'
OtherPunctuation
ghci> generalCategory '9'
DecimalNumber
ghci> map generalCategory " ¥t¥nA9?|"
[Space,Control,Control,UppercaseLetter,DecimalNumber,OtherPunctuation,M
athSymbol] ]
```

There are also functions that can convert between Ints and Chars:

```
ghci> map digitToInt "FF85AB"
[15,15,8,5,10,11]
ghci> intToDigit 15
'f'
ghci> intToDigit 5
'5'
ghci> chr 97
'a'
ghci> map ord "abcdefgh"
[97,98,99,100,101,102,103,104]
```

Neat application: Ceasar ciphers (more here)
A primitive encryption cipher which encodes messages by shifted them a fixed amount in the alphabet.

Example: hello with shift of 3

```
encode :: Int -> String -> String
encode shift msg =
   let ords = map ord msg
       shifted = map (+ shift) ords
   in  map chr shifted
```

Now to use it:

```
ghci> encode 3 "Heeeeey"
"Khhhhh|"
ghci> encode 4 "Heeeeey"
"Liiiii}"
ghci> encode 1 "abcd"
"bcde"
ghci> encode 5 "Merry Christmas! Ho ho ho!""Rf
ww~%Hmwnxyrfx&%Mt%mt%mt&"
```

Decoding just reverses the encoding:

```
decode :: Int -> String -> String
decode shift msg =
          encode (negate shift) msg
```

```
ghci> encode 3 "I'm a little teapot"
"Lp#d#olwwoh#whdsrw"
ghci> decode 3 "Lp#d#olwwoh#whdsrw"
"Im a little teapot"
ghci> decode 5 . encode 5 $ "This is a sentence
"
"This is a sentence"
```

Making our own modules

We specify our own modules at the beginning of a file. For example, if we had a set of geometry functions:

```
module Geometry
( sphereVolume
, sphereArea
, cubeVolume
, cubeArea
, cuboidArea
, cuboidVolume
) where
```

These functions are 'exposed' or available to users of the module

Then, we write the functions:

```
sphereVolume :: Float -> Float
sphereVolume radius = (4.0 / 3.0) * pi *
                        (radius ^ 3)

sphereArea :: Float -> Float
sphereArea radius = 4 * pi * (radius ^ 2)

cubeVolume :: Float -> Float
cubeVolume side = cuboidVolume side side side
…  etc.
```

Note that we can have “private” helper functions, also:

```
cuboidVolume :: Float -> Float -> Float
                    -> Float
cuboidVolume a b c = rectangleArea a b * c

cuboidArea :: Float -> Float ->
                           Float -> Float
cuboidArea a b c = rectangleArea a b * 2 + rectangl
eArea a c * 2 + rectangleArea c b * 2

rectangleArea :: Float -> Float -> Float
rectangleArea a b = a * b
```

Can also nest these. Make a folder called Geometry, with 3 files inside it, first up Sphere.hs

```
module Geometry.Sphere
( volume
, area
) where

volume :: Float -> Float
volume radius = (4.0 / 3.0) * pi * (radius ^
 3)
area :: Float -> Float
area radius = 4 * pi * (radius ^ 2)
```

Then Cuboid.hs:

```haskell
module Geometry.Cuboid
( volume
, area
) where

volume :: Float -> Float -> Float -> Float
volume a b c = rectangleArea a b * c


…
```

etc.


ANY
QUESTIONS?