Discussion on HW3

CSCE 314

Section 502

Anandi Dutta

Fold Function

- Previously, when we work with the recursive function and lists, it follows that:
- Have an edge case for the empty list.
- Introduce the **x:xs** pattern
- Do some action that involves a single element and the rest of the list.

It turns out this is a very common pattern, so a couple of very useful functions were introduced to encapsulate it. These functions are called folds. They're sort of like the **map** function, only they reduce the list to some single value.

Fold Function

- A fold takes a binary function, a starting value (let’s call it the accumulator) and a list to fold up.
- The binary function itself takes two parameters. The binary function is called with the accumulator and the first (or last) element and produces a new accumulator. Then, the binary function is called again with the new accumulator and the now new first (or last) element, and so on. Once we've walked over the whole list, only the accumulator remains, which is what we've reduced the list to.

foldl

- foldl can be called as the left fold
- It folds the list up from the left side. The binary function is applied between the starting value and the head of the list. That produces a new accumulator value and the binary function is called with that value and the next element, etc.

foldl

- `sum' :: (Num a) => [a] -> a`
- `sum' xs = foldl (\acc x -> acc + x) 0 xs`

\acc x -> acc + x is the binary function

`0` is the starting value and `xs` is the list to be folded up

foldl

0 + 3

[3,5,2,1]

3+ 5

[5,2,1]

8+2

[2,1]

10 + 1

[1]

11

- Now first, **0** is used as the **acc** parameter to the binary function and **3** is used as the **x** (or the current element) parameter. **0 + 3** produces a **3** and it becomes the new accumulator value, so to speak. Next up, **3** is used as the accumulator value and **5** as the current element and **8** becomes the new accumulator value. Moving forward, **8** is the accumulator value, **2** is the current element, the new accumulator value is **10**. Finally, that **10** is used as the accumulator value and **1** as the current element, producing an **11**

Fold Function

- The lambda function **(\acc x -> acc + x)** is the same as **(+)**. We can omit the **xs** as the parameter because calling **foldl (+) 0** will return a function that takes a list. Generally, if you have a function like **foo a = bar b a**, you can rewrite it as **foo = bar b**, because of currying.
- Reversing a list, what’s the difference between **foldl** and **foldr**
- **Folds can be used to implement any function where you traverse a list once, element by element, and then return something based on that. Whenever you want to traverse a list to return something, chances are you want a fold.** That's why folds are, along with maps and filters, one of the most useful types of functions in functional programming.

Fold Function

- The foldl1 and foldr1 functions work much like foldl and foldr, only you don't need to provide them with an explicit starting value. They assume the first (or last) element of the list to be the starting value and then start the fold with the element next to it. With that in mind, the sum function can be implemented like so: sum = foldl1 (+).
- How they act on empty lists?
- How they act on infinite lists?

Modules

- A Haskell module is a collection of related functions, types and typeclasses. A Haskell program is a collection of modules where the main module loads up the other modules and then uses the functions defined in them to do something.
- If a module is generic enough, the functions it exports can be used in a multitude of different programs. If your own code is separated into self-contained modules which don't rely on each other too much (we also say they are loosely coupled), you can reuse them later on. It makes the whole deal of writing code more manageable by having it split into several parts, each of which has some sort of purpose.

Modules

- The Haskell standard library is split into modules, each of them contains functions and types that are somehow related and serve some common purpose. There's a module for manipulating lists, a module for concurrent programming, a module for dealing with complex numbers, etc. All the functions, types and typeclasses that we've dealt with so far were part of the **Prelude** module, which is imported by default.

Modules

Folllow this : import <module name>

```
import Data.List

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

Modules

- `ghci> :m + Data.List`
- `ghci> :m + Data.List Data.Map Data.Set`
- `import Data.List (nub, sort)`   -- import only nub and sort
- `import Data.List hiding (nub) --import every function of Data.List except nub`
- `import qualified Data.Map --we do it to avoid name clash`

Modules

`Data.Map` contains some functions named as `filter, null` etc. `Prelude` also has some functions named as `filter, null` etc. But, they are different. How to avoid confusion?

```
Data.Map.filter -- it’s tedious to do in front of every uses.
import qualified Data.Map as M
--Now, to reference Data.Map's filter function, we just use
M.filter
```

Helpful Links:

1) **List of Haskell modules: https://downloads.haskell.org/~ghc/latest/docs/html/libraries/**
2) **Hoogle: https://www.haskell.org/hoogle/**

Read about: Data.List, Data.Char, Data.Set, Data.Map

Creating your own module

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


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


sphereArea :: Float -> Float
sphereArea radius = 4 * pi * (radius ^ 2)
  ----continues
--To use it write import Geometry
```

--Geometry is a module that has three sub-modules, one for each type of object.

--First, we'll make a folder called Geometry. Mind the capital G. In it, we'll place three files: Sphere.hs, Cuboid.hs, and Cube.hs. Here's what the files will contain:

```
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)
--To use it write import Geometry.Sphere
```

Creating your own Types and Typeclasses

```
data Bool = False | True
--use the data keyword to define a type

--The part before the = denotes the type, which is
Bool.The parts after the = are value constructors. They
specify the different values that this type can have. The
| is read as or. So we can read this as: the Bool type
can have a value of True or False. Both the type name and
the value constructors have to be capital cased.
Ex: data Int = -2147483648 | -2147483647 | ... | -1 | 0 |
1 | 2 | ... | 2147483647
```

Creating your own Types and Typeclasses

```
data Shape = Circle Float Float Float | Rectangle
Float Float Float Float

--Value constructors are actually functions that
ultimately return a value of a data type.

surface :: Shape -> Float
surface (Circle _ r) = pi * r ^ 2
surface (Rectangle (Point x1 y1) (Point x2 y2)) =
(abs $ x2 - x1) * (abs $ y2 - y1)
```

Creating your own Types and Typeclasses

--if we add deriving (Show) at the end of a data declaration, Haskell automagically makes that type part of the Show typeclass

```
data Shape = Circle Float Float Float | Rectangle Float Float Float Float
deriving (Show)

ghci> Circle 10 20 5
Circle 10.0 20.0 5.0
ghci> Rectangle 50 230 60 90
Rectangle 50.0 230.0 60.0 90.0

ghci> map (Circle 10 20) [4,5,6,6]
[Circle 10.0 20.0 4.0,Circle 10.0 20.0 5.0,Circle 10.0 20.0 6.0,Circle
10.0 20.0 6.0]
```

Creating your own Types and Typeclasses

```
data Tree a = EmptyTree | Node a (Tree a) (Tree a)
deriving (Show, Read, Eq)

--the type for our insertion function is going to be
something like a -> Tree a - > Tree a. It takes an
element and a tree and returns a new tree that has
that element inside.
```

Creating your own Types and Typeclasses

```
singleton :: a -> Tree a
singleton x = Node x EmptyTree EmptyTree

treeInsert :: (Ord a) => a -> Tree a -> Tree a
treeInsert x EmptyTree = singleton x
treeInsert x (Node a left right)
    | x == a = Node x left right
    | x < a  = Node a (treeInsert x left) right
    | x > a  = Node a left (treeInsert x right)
```

Creating your own Types and Typeclasses/ instances

```
data TrafficLight = Red | Yellow | Green

instance Eq TrafficLight where
    Red == Red = True
    Green == Green = True
    Yellow == Yellow = True
    _ == _ = False                          -- we're
going to write up some instances by hand, even
though we could derive them for types like Eq and
Show. Here's how we make it an instance of Eq.
```

Define a recursive function

```
merge :: [Int] → [Int] → [Int]
```

```
merge [] ys = ys
merge xs [] = xs
merge (x:xs) (y:ys) = if x <= y then
                             x: merge xs (y:ys)
                          else
                             y: merge (x:xs) ys
```

that merges two sorted lists of integers to give a single sorted list. For example:

```
> merge [2,5,6] [1,3,4]

[1,2,3,4,5,6]
```

Define a recursive function

```
msort :: [Int] → [Int]
```

that implements merge sort, which can be specified by the following two rules:

1. Lists of length ≤ 1 are already sorted;
2. Other lists can be sorted by sorting the two halves and merging the resulting lists.

```
halves xs = splitAt (length xs `div` 2) xs
msort [] = []
msort [x] = [x]
msort xs = merge (msort ys) (msort zs)
           where (ys,zs) = halves xs
```

References:

1) Programming in Haskell
2) Learn Your Haskell
3) Real World Haskell