wada

Modules

Introduction

To have software that is easy to maintain and scalable, it is necessary and important to organize the code. Haskell provides tools to achieve these goals: Modules.

Definition

A Haskell module is a collection of functions, types, and typeclasses put together to achieve a specific purpose.

Why Modules

Having code split up into several modules has quite a lot of advantages. 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.

Loading 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. In this part, we're going to examine a few useful modules and the functions that they have. But first, we're going to see how to import modules.

The syntax for importing modules in a Haskell script is `import <module name>`. This must be done before defining any functions, so imports are usually done at the top of the file. One script can, of course, import several modules. Just put each import statement into a separate line. Let's import the Data.List module, which has a bunch of useful functions for working with lists and use a function that it exports to create a function that tells us how many unique elements a list has.

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

When you do `import Data.List`, all the functions that Data.List exports become available in the global **namespace** (collection of types name, functions name, typeclasses name of a module), meaning that you can call them from wherever in the script. nub is a function defined in Data.List that takes a list and weeds out duplicate elements. Composing length and nub by doing length . nub produces a function that's the equivalent of \xs -> length (nub xs).

Selective Import

You can also put the functions of modules into the global namespace when using GHCI. If you're in GHCI and you want to be able to call the functions exported by Data.List, do this:

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

However, if you've loaded a script that already imports a module, you don't need to use :m + to get access to it.

If you just need a couple of functions from a module, you can selectively import just those functions. If we wanted to import only the nub and sort functions from Data.List, we'd do this:

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

You can also choose to import all of the functions of a module except a few select ones. That's often useful when several modules export functions with the same name and you want to get rid of the offending ones. Say we already have our own function that's called nub and we want to import all the functions from Data.List except the nub function:

```
import Data.List hiding (nub)
```

Another way of dealing with name clashes is to do qualified imports. The Data.Map module, which offers a data structure for looking up values by key, exports a bunch of functions with the same name as Prelude functions, like filter or null. So when we import Data.Map and then call filter, Haskell won't know which function to use. Here's how we solve this:

```
import qualified Data.Map
```

This makes it so that if we want to reference Data.Map's filter function, we have to do Data.Map.filter, whereas just filter still refers to the normal filter we all know and love. But typing out Data.Map in front of every function from that module is kind of tedious. That's why we can rename the qualified import to something shorter:

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

Now, to reference Data.Map's filter function, we just use M.filter.

wada

Some modules

Data.List

The Data.List module is all about lists, obviously. It provides some very useful functions for dealing with them. Data.List module exposes a lot of functions. To see the functions exposes by Data.List, in REPL, run the command:

```
:browse Data.List
```

The documentation of those functions can be found on Hoogle. It's a really awesome
Haskell search engine, you can search by name, module name or even type signature.

Data.Char

The Data.Char module does what its name suggests. It exports functions that deal with characters. It's also helpful when filtering and mapping over strings because they're just lists of characters.
To see the functions exposes by Data.Char, in REPL, run the command:

```
:browse Data.Char
```

Data.Map

Association lists (also called dictionaries) define a data structure internally implemented with trees that store key-value pairs where ordering doesn't matter.
For instance, we might use an association list to store phone numbers, where phone numbers would be the values and people's names would be the keys. We don't care
in which order they're stored, we just want to get the right phone number for the right person.

wada

The most obvious way to represent association lists in Haskell would be by having a list of pairs. The first component in the pair would be the key, the second component the value. Here's an example of a list representing an association list with phone numbers:

```
phoneBook = [("betty","555-2938")
,("bonnie","452-2928")
,("patsy","493-2928")
,("lucille","205-2928")
,("wendy","939-8282")
,("penny","853-2492")
]
```

The fromList function takes an association list (in the form of a list) and returns a map with the same associations.

```
ghci> Map.fromList
[("betty","555-2938"),("bonnie","452-2928"),("lucille","2
05-2928")]
fromList
[("betty","555-2938"),("bonnie","452-2928"),("lucille","2
05-2928")]
ghci> Map.fromList [(1,2),(3,4),(3,2),(5,5)]
fromList [(1,2),(3,2),(5,5)]
```

If there are duplicate keys in the original association list, the duplicates are just discarded. This is the type signature of fromList
Map.fromList :: (Ord k) => [(k, v)] -> Map.Map k v

To see the functions exposes by Data.Map, in REPL, run the command:

```
:browse Data.Map
```

The documentation of those functions can be found on Hoogle.

wada

Data.Set

The Data.Set module offers us, well, sets. Like sets from mathematics, sets are kind of like a cross between lists and maps. All the elements in a set data structure are unique. And because they're internally implemented with trees (much like maps in Data.Map), they're ordered. Checking for membership, inserting, deleting, etc. is much faster than doing the same thing with lists. The most common operation when dealing with sets are inserting into à set, checking for membership and converting a set to a list.

Because the names in Data.Set clash with a lot of Prelude and Data.List names, we do a qualified import. Put this import statement in a script:

```
import qualified Data.Set as Set
```

And then load the script via GHCI.

Let's say we have two pieces of text. We want to find out which characters were used in both of them.

```
text1 = "I just had an anime dream. Anime... Reality...
Are they so different?"
text2 = "The old man left his garbage can out and now his
trash is all over my lawn!"
```

The fromList function works much like you would expect. It takes a list and converts it into a set.

```
ghci> let set1 = Set.fromList text1
ghci> let set2 = Set.fromList text2
ghci> set1
fromList " .?AIRadefhijlmnorstuy"
ghci> set2
```

```
fromList " !Tabcdefghilmnorstuvwy"
```

As you can see, the items are ordered and each element is unique. Now let's use the intersection function to see which elements they both share.

```
ghci> Set.intersection set1 set2
fromList " adefhilmnorstuy"
```

We can use the difference function to see which letters are in the first set but aren't in the second one and vice versa.

```
ghci> Set.difference set1 set2
fromList ".?AIRj"
ghci> Set.difference set2 set1
fromList "!Tbcgvw"
```

etc.
To see the functions exposes by Data.Set, in REPL, run the command:

```
:browse Data.Set
```

The documentation of those functions can be found on Hoogle.

Making our own modules

When making programs, it's good practice to take functions and types that work towards a similar purpose and put them in a module. That way, you can easily reuse those functions in other programs by just importing your module.
Let's see how we can make our own modules by making a little module that provides some functions for calculating the volume and area of a few geometrical objects. We'll start by creating a file called Geometry.hs.

wada

Defining Module and Exporting function

We say that a module exports functions. What that means is that when I import a module, I can use the functions that it exports. It can define functions that its functions call internally, but we can only see and use the ones that it exports.
At the beginning of a module, we specify the module name. If we have a file called Geometry.hs, then we should name our module Geometry. Then, we specify the functions that it exports and after that, we can start writing the functions. So we'll start with this.

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

As you can see, we'll be doing areas and volumes for spheres, cubes and cuboids. Let's go ahead and define our functions then:

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

```

wada

```
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

cubeArea :: Float -> Float
cubeArea side = cuboidArea side side side

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 + rectangleArea
a c * 2 +
rectangleArea c b * 2

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

Pretty standard geometry right here. There are a few things to take note of though. Because a cube is only a special case of a cuboid, we defined its area and volume by treating it as a cuboid whose sides are all of the same length. We also defined a helper function called rectangleArea, which calculates a rectangle's area based on the lengths of its sides. It's rather trivial because it's just multiplication. Notice that we used it in our functions in the module (namely cuboidArea and cuboidVolume) but we didn't export it! Because we want our module to just present functions for dealing with three dimensional objects, we used rectangleArea but we didn't export it. When making a module, we usually export only those

functions that act as a sort of interface to our module so that the implementation is hidden. If someone is using our Geometry module, they don't have to concern themselves with functions that we don't export. We can decide to change those functions completely or delete them in a newer version (we could delete rectangleArea and just use * instead) and no one will mind because we weren't exporting them in the first place.
To use our module, we just do:

```
import Geometry
```

Geometry.hs has to be in the same folder that the program that's importing it is in, though.

Hierarchical Structured Modules

Modules can also be given hierarchical structures. Each module can have a number of sub-modules and they can have sub-modules of their own. Let's section these functions off so that 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
```

wada

```
area radius = 4 * pi * (radius ^ 2)
```

Cuboid.hs

```
module Geometry.Cuboid
( volume
, area
) where
volume :: Float -> Float -> Float -> Float
volume a b c = rectangleArea a b * c
area :: Float -> Float -> Float -> Float
area a b c = rectangleArea a b * 2 + rectangleArea a c *
2 +
rectangleArea c b * 2
rectangleArea :: Float -> Float -> Float
rectangleArea a b = a * b
```

Cube.hs

```
module Geometry.Cube
( volume
, area
) where
import qualified Geometry.Cuboid as Cuboid
volume :: Float -> Float
volume side = Cuboid.volume side side side
area :: Float -> Float
area side = Cuboid.area side side side
```

So first is Geometry.Sphere. Notice how we placed it in a folder called Geometry and

wada

then defined the module name as Geometry.Sphere. We did the same for the cuboid. Also notice how in all three sub-modules, we defined functions with the same names. We can do this because they're separate modules. We want to use functions from Geometry.Cuboid in Geometry.Cube but we can't just straight up do import Geometry.Cuboid because it exports functions with the same name as Geometry.Cube. That's why we do a qualified import and all is well. So now if we're in a file that's on the same level as the Geometry folder, we can do, say:

```
import Geometry.Sphere
```

And then we can call area and volume and they'll give us the area and volume for a sphere. And if we want to juggle two or more of these modules, we have to do qualified imports because they export functions with the same names. So we just do something like:

```
import qualified Geometry.Sphere as Sphere
import qualified Geometry.Cuboid as Cuboid
import qualified Geometry.Cube as Cube
```

And then we can call Sphere.area, Sphere.volume, Cuboid.area, etc. and each will calculate the area or volume for their corresponding object. The next time you find yourself writing a file that's really big and has a lot of functions, try to
see which functions serve some common purpose and then see if you can put them in their own module. You'll be able to just import your module the next time you're writing a program that requires some of the same functionality.