

Intro to Haskell Notes: Part 9

Adrian Brasoveanu*

October 13, 2013

Contents

1	<i>Data.Char</i>	1
1.1	Predicates over characters	1
1.2	<i>GeneralCategory</i>	2
1.3	<i>toUpper, toLower</i>	3
1.4	The Caesar cipher	3
2	<i>Data.Set</i>	5
2.1	<i>fromList</i>	5
2.2	<i>intersection</i>	6
2.3	<i>difference</i>	6
2.4	<i>union</i>	6
2.5	<i>empty, null, size, member, singleton, insert, delete</i>	7
2.6	<i>isSubsetOf</i> and <i>isProperSubsetOf</i>	7
2.7	<i>map</i> and <i>filter</i>	8
2.8	<i>setNub</i>	8
3	Making our own modules	9
4	Hierarchical modules	10

1 *Data.Char*

```
ghci 1> import Data.Char
```

The *Data.Char* module does what its name suggests: it exports functions that deal with characters. But they're also helpful when filtering and mapping over strings because they're just lists of characters.

1.1 Predicates over characters

Here are some of the exported predicates over characters (functions of type *Char* → *Bool*):

- *isSpace* checks whether a character is a white-space character; that includes spaces, tab characters, newlines etc.
- *isLower* checks whether a character is a lower-case character

*Based primarily on *Learn You a Haskell for Great Good!* by Miran Lipovača, <http://learnyouahaskell.com/>.

- *isUpper* checks whether a character is an upper-case character
- *isAlpha* checks whether a character is a letter
- *isAlphaNum* checks whether a character is a letter or a number
- *isDigit* checks whether a character is a digit
- *isLetter* checks whether a character is a letter
- *isNumber* checks whether a character is numeric
- *isPunctuation* checks whether a character is punctuation
- *isSymbol* checks whether a character is a fancy mathematical or currency symbol

Most of the time, we use these to filter out strings.

For instance, let's say we're making a program that takes a username and the username can only be comprised of alphanumeric characters. We can use the *Data.List* function *all* in combination with the *Data.Char* predicates to determine if the username is OK.

```
ghci 2> all isAlphaNum "bobby283"
True
```

```
ghci 3> all isAlphaNum "eddy the fish!"
False
```

1.2 GeneralCategory

The *Data.Char* also exports a datatype *GeneralCategory* that's like *Ordering*.

The *Ordering* type can have a value of *LT*, *EQ* or *GT* – a sort of enumeration. It describes a few possible results that can arise from comparing two elements.

The *GeneralCategory* type is also an enumeration. It presents us with a few possible categories that a character can fall into. The main function for getting the general category of a character is *generalCategory*. Its type is listed below.

```
ghci 4> :t generalCategory
generalCategory :: Char → GeneralCategory
```

```
ghci 5> generalCategory ' '
Space
```

```
ghci 6> generalCategory 'A'
UppercaseLetter
```

```
ghci 7> generalCategory 'a'
LowercaseLetter
```

```
ghci 8> generalCategory '.'  
OtherPunctuation
```

```
ghci 9> generalCategory '9'  
DecimalNumber
```

1.3 *toUpper, toLower*

toUpper converts a character to upper-case. Spaces, numbers and the like remain unchanged.

```
ghci 10> map toUpper "wow that's great r2-d2"  
"WOW THAT'S GREAT R2-D2"
```

toLower converts a character to lower-case.

```
ghci 11> map toLower "WOW THAT'S GREAT R2-D2"  
"wow that's great r2-d2"
```

1.4 The Caesar cipher

The Caesar cipher is an old and fairly simple method of encoding messages by shifting each character in a message by a fixed number of positions in the alphabet. We can easily create a sort of Caesar cipher of our own, only we won't confine ourselves to the alphabet.

```
ghci 12> let {encode :: Int -> String -> String;  
            encode shift msg = map chr $ map (+shift) $ map ord msg }
```

Here, we first convert the string to a list of numbers with the function *ord*.

```
ghci 13> :! hoople -- info "ord :: Char -> Int"  
Data.Char ord :: Char -> Int  
The Prelude.fromEnum method restricted to the type Data.Char.Char.  
From package base ord :: Char -> Int
```

Here's an example:

```
ghci 14> map ord "Heeeey"  
[72, 101, 101, 101, 101, 101, 121]
```

Then we add the shift amount to each number:

```
ghci 15> map ((+3) . ord) "Heeeey"  
[75, 104, 104, 104, 104, 104, 124]
```

Then we finally convert the list of numbers back to characters with the function *chr*.

```
ghci 16> :! hoople -- info "chr :: Int -> Char"
Data.Char chr :: Int -> Char
The Prelude.toEnum method restricted to the type Data.Char.Char.
From package base chr :: Int -> Char
```

For example:

```
ghci 17> map (chr ∘ (+3) ∘ ord) "Heeeey"
"Khhhh| "
```

If you're a composition cowboy, you could write the *encode* function more concisely following the code in the example above:

```
ghci 18> let {encode :: Int → String → String;
             encode shift msg = map (chr ∘ (+shift) ∘ ord) msg }
```

Let's try encoding a few messages.

```
ghci 19> encode 3 "Heeeey"
"Khhhh| "
```

```
ghci 20> encode 4 "Heeeey"
"Liinii}"
```

```
ghci 21> encode 1 "abcd"
"bcde"
```

```
ghci 22> encode 5 "Merry Christmas! Ho ho ho!"
"Rjww~%Hmwnxyrfx&%Mt%mt%mt&"
```

Decoding a message is just shifting it back by the same number of places:

```
ghci 23> let decode shift msg = encode (negate shift) msg
```

```
ghci 24> encode 3 "Im a little teapot"
"Lp#d#olwwoh#whdsrw"
```

```
ghci 25> decode 3 "Lp#d#olwwoh#whdsrw"  
"Im a little teapot"
```

```
ghci 26> decode 5 ◦ encode 5 $ "This is a sentence"  
"This is a sentence"
```

2 *Data.Set*

The *Data.Set* module offers us sets in the mathematical sense:

- all the elements in a set are unique;
- and because they're internally implemented with trees (for speed), they're ordered;
- checking for membership, inserting, deleting etc. is much faster than doing the same thing with lists;

Because the names in *Data.Set* clash with a lot of *Prelude* and *Data.List* names, we do a qualified import.

```
ghci 27> import qualified Data.Set as S
```

2.1 *fromList*

Besides inserting into a set and checking for membership, the most common operation when dealing with sets is probably converting a list into a set.

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

```
ghci 28> let text1 =  
        "I just had an anime dream. Anime... Reality... Are they so different?"
```

```
ghci 29> let 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 30> let set1 = S.fromList text1
```

```
ghci 31> let set2 = S.fromList text2
```

```
ghci 32> :t S.fromList  
S.fromList :: Ord a => [a] -> S.Set a
```

```
ghci 33> set1  
fromList " .?AIRadefhijlmnorstuy"
```

```
ghci 34> :t set1  
set1 :: S.Set Char
```

```
ghci 35> set2  
fromList " !Tabcdefghilmnorstuvw"
```

As you can see, the items are ordered and each element is unique.

2.2 intersection

Now let's use the *intersection* function to see which elements are in both sets.

```
ghci 36> S.intersection set1 set2  
fromList " adefhilmnorstuy"
```

2.3 difference

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 37> S.difference set1 set2  
fromList " .?AIRj"
```

```
ghci 38> S.difference set2 set1  
fromList " !Tbcgvw"
```

2.4 union

Or we can see all the unique letters used in both sentences by using *union*.

```
ghci 39> S.union set1 set2
fromList " !.?AIRTabcdefghijklmnorstuvwxy"
```

2.5 *empty, null, size, member, singleton, insert, delete*

The *empty*, *null*, *size*, *member*, *singleton*, *insert* and *delete* functions all work like you'd expect them to.

```
ghci 40> S.empty
fromList []
```

```
ghci 41> S.null S.empty
True
```

```
ghci 42> S.null $ S.fromList [3,4,5,5,4,3]
False
```

```
ghci 43> S.size $ S.fromList [3,4,5,3,4,5]
3
```

```
ghci 44> S.singleton 9
fromList [9]
```

```
ghci 45> S.insert 4 $ S.fromList [9,3,8,1]
fromList [1,3,4,8,9]
```

```
ghci 46> S.insert 8 $ S.fromList [5..10]
fromList [5,6,7,8,9,10]
```

```
ghci 47> S.delete 4 $ S.fromList [3,4,5,4,3,4,5]
fromList [3,5]
```

2.6 *isSubsetOf* and *isProperSubsetOf*

We can also check for subsets or proper subsets.

```
ghci 48> S.fromList [2,3,4] 'S.isSubsetOf' S.fromList [1,2,3,4,5]
True
```

```
ghci 49> S.fromList [1,2,3,4,5] 'S.isSubsetOf' S.fromList [1,2,3,4,5]
True
```

```
ghci 50> S.fromList [2,3,4,8] 'S.isSubsetOf' S.fromList [1,2,3,4,5]
False
```

```
ghci 51> S.fromList [1,2,3,4,5] 'S.isProperSubsetOf' S.fromList [1,2,3,4,5]
False
```

2.7 *map* and *filter*

We can also map over sets and filter them.

```
ghci 52> S.filter odd $ S.fromList [3,4,5,6,7,2,3,4]
fromList [3,5,7]
```

```
ghci 53> S.map (+1) $ S.fromList [3,4,5,6,7,2,3,4]
fromList [3,4,5,6,7,8]
```

2.8 *setNub*

Sets are often used to weed out duplicates in a list by first making the list into a set with *fromList* and then converting the set back to a list with *toList*.

The *Data.List* function *nub* already does this, but it is much faster to weed out duplicates for large lists if you convert them to sets and then convert them back to lists.

However, using *nub* only requires the type of the list elements to be part of the *Eq* typeclass, whereas if we want to convert the list to a set, the type of the list elements has to be in *Ord*.

```
ghci 54> let setNub xs = S.toList $ S.fromList xs
```

```
ghci 55> setNub "HEY WHATS CRACKALACKIN"
" ACEHIKLNIRSTWY"
```

```
ghci 56> import Data.List
```

```
ghci 57> nub "HEY WHATS CRACKALACKIN"
"HEY WATSCRKLIN"
```


As we already mentioned, *setNub* is generally faster than *nub* on big lists. But as you can see, *nub* preserves the ordering of the list's elements, while *setNub* does not.

3 Making our own modules

We've looked at some modules so far, but how do we make our own module?

Almost every programming language enables you to split your code up into several files and Haskell is no different. 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*.

We have repeatedly said that a module exports functions. What that means is that:

- when we import a module, we can use the functions that it *explicitly* exports;
- it can define functions that its functions call internally, i.e., inside the module, but we can only see and use the functions 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;
- after that, we can start writing the functions.

Here's the full code in the file:

```
module Geometry
(sphereVolume
, sphereArea
, cubeVolume
, cubeArea
, cuboidArea
, cuboidVolume
) where
sphereVolume :: Double → Double
sphereVolume radius = (4.0 / 3.0) * pi * (radius ↑ 3)
sphereArea :: Double → Double
sphereArea radius = 4 * pi * (radius ↑ 2)
cubeVolume :: Double → Double
cubeVolume side = cuboidVolume side side side
cubeArea :: Double → Double
cubeArea side = cuboidArea side side side
cuboidVolume :: Double → Double → Double → Double
cuboidVolume a b c = rectangleArea a b * c
cuboidArea :: Double → Double → Double → Double
cuboidArea a b c = rectangleArea a b * 2 + rectangleArea a c * 2 + rectangleArea c b * 2
rectangleArea :: Double → Double → Double
rectangleArea a b = a * b
```

We defined a helper function called *rectangleArea* that calculates the area of a rectangle based on the lengths of its sides. Notice that we used it in our functions in the module (namely *cuboidArea* and *cuboidVolume*) but we didn't export it. That's because we want our module to just present functions for dealing with 3D objects.

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.

```
ghci 58> :l Geometry
```

```
ghci 59> sphereArea 4
201.06192982974676
```

Geometry.hs has to be in the same folder as the program importing it.

4 Hierarchical modules

Modules can also be given a 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*.

Then we add code to these files. Let's start with *Geometry.Sphere*. Notice how we placed it in a folder called *Geometry* and then defined the module name as *Geometry.Sphere*.

```
module Geometry.Sphere
  (volume
  ,area
  ) where
volume :: Double → Double
volume radius = (4.0 / 3.0) * pi * (radius ↑ 3)
area :: Double → Double
area radius = 4 * pi * (radius ↑ 2)
```

Now we add the *Cuboid* and *Cube* submodules in the corresponding *hs* files.

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

```

module Geometry.Cube
  (volume
  ,area
  ) where
  import qualified Geometry.Cuboid as Cuboid
  volume :: Double → Double
  volume side = Cuboid.volume side side side
  area :: Double → Double
  area side = Cuboid.area side side side

```

Notice how in all three sub-modules, we defined functions with the same names:

- we can do this because they're separate modules;
- but if we want to use functions from *Geometry.Cuboid* in *Geometry.Cube*, we can't just import them with **import** *Geometry.Cuboid* because we would import functions with the same names as the one in *Geometry.Cube*;
- so we need to do a qualified import – namely, **import** *qualified Geometry.Cuboid as Cuboid* – to avoid the name clash.

In general, if we want to use two or more of these modules at the same time, we have to do qualified imports because they export functions with the same names. If we want to do this in *ghci*, a good way is to create a file *GeomAll.hs* with the following content:

```

module GeomAll where
  import qualified Geometry.Sphere as Sphere
  import qualified Geometry.Cuboid as Cuboid
  import qualified Geometry.Cube as Cube

```

and then load it in *ghci* as follows:

```
ghci 60> :l GeomAll
```

If we're not working in *ghci*, we can simply add the **import** commands directly at the top of the script in which we need the *Geometry* module.

We can now call various functions from the *Geometry* module, e.g., *Sphere.area*, *Sphere.volume*, *Cuboid.area* etc., in *ghci* or inside our script.

```
ghci 61> Sphere.area 4
201.06192982974676
```

```
ghci 62> Sphere.volume 4
268.082573106329
```

```
ghci 63> Cube.area 4
96.0
```

```
ghci 64> Cube.volume 4  
64.0
```

To summarize, if you have a file that's really big and has a lot of functions:

- try to see which functions serve a common purpose;
- put them in their own module;
- import the module next time you're writing a program that requires some of the same functionality.