Intro to Haskell Notes: Part 8

Adrian Brasoveanu*

October 8, 2013

Contents

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

1 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.

Having code split up into several modules has several advantages:

- if a module is generic enough, the functions it exports can be used for many different programming tasks
- 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 (some of) them separately later on
- splitting code into several parts, each of which has a specific purpose, makes writing code more manageable

The Haskell standard library is split into modules. Each of them contains functions and types that are somehow related and serve a common purpose.

There's a module for manipulating lists, one for manipulating sets, 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 set of notes and the following one, we're going to examine a few useful modules and the functions that they make available.

1.1 Importing modules

But first, we're going to see how to import modules. The syntax for importing modules in a Haskell script is:

(1) **import** < *moduleName*>

This must be done before defining any functions, so imports are usually done at the top of the file.

We can import several modules in one script, we just have to put each import statement on a separate line.

(2) **import** < *moduleName1*>
import < *moduleName2*>
import < *moduleName3*>
...

For example, let's import the *Data.List* module, which makes available many useful functions for working with lists.

ghci 1> :*show imports*
import Prelude – implicit

ghci 2> **import** *Data.List*

ghci 3> :*show imports*
import Prelude – implicit import Data.List

When we do **import** *Data.List*, all the functions that *Data.List* exports become available in the global namespace, meaning that we can call them from wherever in the script.

In `ghci`, we can also import modules by running $:m + Data.List$.

If we want to load up the names from several modules inside GHCI, we don't have to do $:m + ...$ several times, we can just load up several modules at once: $:m + Data.List\ Data.Map\ Data.Set$.

Note that if we've loaded a script in `ghci` that already imports a module, we don't need to use $:m + ...$ to get access to it.

1.2 Importing functions from modules

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

(3) **import** *Data.List* $(nub, sort)$

1.3 Importing modules without certain functions

We 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 we want to avoid name conflicts. For example, suppose 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. Then we do the following:

(4) **import** *Data.List hiding* (nub)

1.4 Qualified imports

Another way of dealing with name clashes is to do qualified imports. For example, 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:

(5) **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:

(6) **import** *qualified Data.Map as M*

To reference *Data.Map*'s *filter* function, we just use *M.filter*.

2 *Data.List*

We will now introduce several very useful functions in the *Data.List* module.

We've already met some of its functions (like *map* and *filter*) because the *Prelude* module imports some functions from *Data.List* for convenience.

You don't have to import *Data.List* via a qualified import because it doesn't clash with any *Prelude* names except for those that Prelude already takes from *Data.List*.

Let's take a look at some of the functions that we haven't seen before.

2.1 *nub*

nub tells us how many unique elements a list has, that is, it takes a list and weeds out duplicate elements.

```
ghci 4> nub [1,2,3,4,3,2,1,2,3,4,3,2,1]
        [1,2,3,4]
```

```
ghci 5> nub "Lots of words and stuff"
        "Lots fwrdanu"
```

Composing *length* and *nub* by doing $length \circ nub$ produces a function that's the equivalent of $\lambda xs \rightarrow length\ (nub\ xs)$.

```
ghci 6> let {numUniques :: (Eq a) ⇒ [a] → Int;
             numUniques = length ∘ nub}
```

```
ghci 7> nub [1,2,4,5,4,2,37,7,2]
        [1,2,4,5,37,7]
```

```
ghci 8> numUniques [1,2,4,5,4,2,37,7,2]
        6
```

2.2 *intersperse* **and** *intercalate*

intersperse takes an element and a list and then puts that element in between each pair of elements in the list.

```
ghci 9> intersperse '.' "MONKEY"
        "M.O.N.K.E.Y"
```

```
ghci 10> intersperse 0 [1,2,3,4,5,6]
         [1,0,2,0,3,0,4,0,5,0,6]
```

intercalate takes a list of lists and a list. It then inserts that list in between all those lists and then flattens the result.

```
ghci 11> intercalate " " ["hey","there","guys"]
         "hey there guys"
```

```
ghci 12> 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]
```

2.3 *concat* and *concatMap*

concat flattens a list of lists.

```
ghci 13> concat ["foo","bar","car"]
    "foobarcar"
```

```
ghci 14> concat [[3,4,5],[2,3,4],[2,1,1]]
    [3,4,5,2,3,4,2,1,1]
```

But it will just remove one level of nesting. So if we want to completely flatten $[[[2,3],[3,4,5],[2]],[[2,3],[3,4]]]$, which is a list of lists of lists, we have to concatenate it twice.

```
ghci 15> concat [[[2,3],[3,4,5],[2]],[[2,3],[3,4]]]
    [[2,3],[3,4,5],[2],[2,3],[3,4]]
```

```
ghci 16> concat ∘ concat $ [[[2,3],[3,4,5],[2]],[[2,3],[3,4]]]
    [2,3,3,4,5,2,2,3,3,4]
```

Doing *concatMap* is the same as:

- first mapping a function over a list,
- then concatenating the list with *concat*.

```
ghci 17> replicate 4 1
    [1,1,1,1]
```

```
ghci 18> map (replicate 4) [1..3]
    [[1,1,1,1],[2,2,2,2],[3,3,3,3]]
```

```
ghci 19> concatMap (replicate 4) [1..3]
    [1,1,1,1,2,2,2,2,3,3,3,3]
```

2.4 *and* and *or*

and takes a list of boolean values and returns *True* only if all the values in the list are *True*.

```
ghci 20> and $ map (>4) [5,6,7,8]
    True
```

```
ghci 21> and $ map (≡ 4) [4,4,4,3,4]
    False
```

or is like *and* except it returns *True* if any of the boolean values in a list is *True*.

```
ghci 22> or $ map (≡ 4) [2,3,4,5,6,1]
    True
```

```
ghci 23> or $ map (>4) [1,2,3]
    False
```

2.5 *any* and *all*

any and *all* take a predicate and then check if any or all the elements in a list satisfy the predicate, respectively.

We usually use these two functions instead of:

- mapping over a list first,
- then doing *and* or *or*.

```
ghci 24> any (≡ 4) [2,3,5,6,1,4]
    True
```

```
ghci 25> all (>4) [6,9,10]
    True
```

```
ghci 26> all (∈ ['A'..'Z']) "HEYGUYSwhatsup"
    False
```

```
ghci 27> any (∈ ['A'..'Z']) "HEYGUYSwhatsup"
    True
```

2.6 *iterate*

iterate takes a function and a starting value.

It applies the function to the starting value, then it applies that function to the result, then it applies the function to that result again etc.

It returns all the results in the form of an infinite list.

```
ghci 28> take 10 $ iterate (*2) 1
    [1,2,4,8,16,32,64,128,256,512]
```

```
ghci 29> take 3 $ iterate (++"hihi") "haha"
    ["haha","hahahihi","hahahihihihi"]
```

2.7 *splitAt*

splitAt takes a number and a list. It then splits the list at that many elements, returning the resulting two lists in a tuple.

```
ghci 30> splitAt 3 "heyman"
    ("hey","man")
```

```
ghci 31> splitAt 100 "heyman"
    ("heyman","")
```

```
ghci 32> splitAt (-3) "heyman"
    ("","heyman")
```

```
ghci 33> let (a,b) = splitAt 3 "foobar" in b ++ a
    "barfoo"
```

2.8 *takeWhile* and *dropWhile*

takeWhile (which we've alread met) takes elements from a list while the predicate holds and then when an element is encountered that doesn't satisfy the predicate, it stops.

```
ghci 34> takeWhile (>3) [6,5,4,3,2,1,2,3,4,5,4,3,2,1]
    [6,5,4]
```

```
ghci 35> takeWhile (/=' ') "This is a sentence"
    "This"
```

Say we wanted to know the sum of all third powers that are under 10000. We can't just:

- map (↑3) over [1..],
- then apply a filter,
- and then try to sum up the result,

because filtering an infinite list never finishes. We may know that all the elements here are ascending, but Haskell doesn't.

Instead, we can do this:

```
ghci 36> sum $ takeWhile (<10000) $ map (↑3) [1..]
    53361
```

- we apply $(\uparrow 3)$ to an infinite list;
- once an element that's over 10000 is encountered, the list is cut off;
- now we can sum up the resulting list.

dropWhile is similar to *takeWhile*, only it drops all the elements while the predicate is true.

```
ghci 37> dropWhile (≢ ' ') "This is a sentence"
    " is a sentence"
```

```
ghci 38> dropWhile (<3) [1,2,2,2,3,4,5,4,3,2,1]
    [3,4,5,4,3,2,1]
```

2.9 *span* and *break*

span is like *takeWhile*, only it returns a pair of lists:

- the first list contains everything in the list resulting from *takeWhile*;
- the second list contains the part of the list that would have been dropped by *takeWhile*.

```
ghci 39> span (≢ ' ') "This is a sentence"
    ("This"," is a sentence")
```

```
ghci 40> let
           (fw, rest) = span (≢ ' ') "This is a sentence"
         in
           "First word:" ++ fw ++ ", the rest:" ++ rest
    "First word:This, the rest: is a sentence"
```

Whereas *span* spans the list while the predicate is true, *break* breaks it when the predicate is first true:

- doing *break p* is the equivalent of doing *span* $(\neg \circ p)$.

```
ghci 41> break (≡ 4) [1,2,3,4,5,6,7]
    ([1,2,3],[4,5,6,7])
```

```
ghci 42> span (≢ 4) [1,2,3,4,5,6,7]
    ([1,2,3],[4,5,6,7])
```

When using *break*, the second list in the result will start with the first element that satisfies the predicate.

2.10 *sort*

sort simply sorts a list. The type of list elements has to be in the *Ord* typeclass so that the elements can be sorted.

```
ghci 43> sort [8,5,3,2,1,6,4,2]
    [1,2,2,3,4,5,6,8]
```

```
ghci 44> sort "This will be sorted soon"
    "    Tbdeehiillnooorssstw"
```

2.11 *group*

group takes a list and groups adjacent elements into sublists if they are equal.

```
ghci 45> group [1,1,1,1,2,2,2,2,3,3,2,2,2,5,6,7]
    [[1,1,1,1],[2,2,2,2],[3,3],[2,2,2],[5],[6],[7]]
```

If we sort a list before grouping it, we can find out how many times each element appears in the list.

```
ghci 46> map (λl@(x:xs) → (x, length l)) ∘ group ∘ sort $ [1,1,1,1,2,2,2,2,3,3,2,2,2,5,6,7]
    [(1,4),(2,7),(3,2),(5,1),(6,1),(7,1)]
```

2.12 *inits* and *tails*

inits and *tails* are like *init* and *tail*, only they recursively apply *init* / *tail* to a list until there's nothing left.

```
ghci 47> inits "w00t"
    ["","w","w0","w00","w00t"]
```

```
ghci 48> tails "w00t"
    ["w00t","00t","0t","t",""]
```

```
ghci 49> let w = "w00t" in zip (inits w) (tails w)
    [("","w00t"),("w","00t"),("w0","0t"),("w00","t"),("w00t","")]
```

2.12.1 Example: Searching through a list for a sublist with *foldl* and *tails*

Let's use a fold to implement searching through a list for a sublist.

```
ghci 50> let {search :: (Eq a) ⇒ [a] → [a] → Bool;
              search needle haystack =
                let nlen = length needle
                in foldl (λacc x → if take nlen x ≡ needle then True else acc)
                         False
                         (tails haystack)}
```

- first we call *tails* on the list through which we're searching;
- then we go over each tail and see if it starts with what we're looking for.

```
ghci 51> search "cat" "i'm a cat burglar"
    True
```

2.13 *isPrefixOf* and *isSuffixOf*

With that, we actually just made a function that behaves like *isInfixOf*.

isInfixOf searches for a sublist within a list and returns *True* if the sublist we're looking for is somewhere inside the target list.

```
ghci 52> "cat" `isInfixOf` "i'm a cat burglar"
    True
```

```
ghci 53> "Cat" `isInfixOf` "i'm a cat burglar"
    False
```

```
ghci 54> "cats" `isInfixOf` "i'm a cat burglar"
    False
```

isPrefixOf and *isSuffixOf* search for a sublist at the beginning and at the end of a list, respectively.

```
ghci 55> "hey" `isPrefixOf` "hey there!"
    True
```

```
ghci 56> "hey" `isPrefixOf` "oh hey there!"
    False
```

```
ghci 57> "there!" `isSuffixOf` "oh hey there!"
    True
```

```
ghci 58> "there!" `isSuffixOf` "oh hey there"
    False
```

2.14 *partition*

partition takes a predicate and a list and returns a pair of lists:

- the first list in the result contains all the elements that satisfy the predicate;
- the second contains all the ones that don't.

```
ghci 59> partition (∈ ['A'..'Z']) "BOBsidneyMORGANeddy"
    ("BOBMORGAN","sidneyeddy")
```

```
ghci 60> partition (>3) [1,3,5,6,3,2,1,0,3,7]
    ([5,6,7],[1,3,3,2,1,0,3])
```

It's important to understand how this is different from *span* and *break*:

- *span* and *break* are done once they encounter the first element that doesn't / does satisfy the predicate;
- in contrast, *partition* goes through the whole list and splits it up according to the predicate.

```
ghci 61> span (∈ ['A'..'Z']) "BOBsidneyMORGANeddy"
    ("BOB","sidneyMORGANeddy")
```

2.15 *find*

find takes a predicate and a list and returns the first element that satisfies the predicate. But it returns that element wrapped in a *Maybe* value.

We'll be covering algebraic data types in depth soon but for now, this is what we need to know:

- a *Maybe* value can either be *Just* < *something*> or *Nothing*;
- much like a list can be either an empty list or a list with some elements, a *Maybe* value can be either 'no elements' or a single element;
- and just like the type of a list of integers is [*Int*], the type of maybe having an integer is *Maybe Int*.

```
ghci 62> find (>4) [1,2,3,4,5,6]
    Just 5
```

ghci 63> *find* (>9) [1,2,3,4,5,6]
Nothing

ghci 64> :t *find*
find :: (*a* → *Bool*) → [*a*] → *Maybe a*

ghci 65> :! *hoogle* -- info find
Data.List find :: (a -> Bool) -> [a] -> Maybe a
The find function takes a predicate and a list and returns the first element in the list matching the predicate, or Nothing if there is no such element.
From package base find :: (a -> Bool) -> [a] -> Maybe a

2.16 *elemIndex* and *elemIndices*

elemIndex is similar to *elem*, only it doesn't return a boolean value but a *Maybe* index:

- if the element we're looking for is in our list, it returns its index wrapped in a *Just*;
- if the element is not in our list, it returns a *Nothing*.

ghci 66> :t *elemIndex*
elemIndex :: *Eq a* ⇒ *a* → [*a*] → *Maybe Int*

ghci 67> :! *hoogle* -- info elemIndex
Data.List elemIndex :: Eq a => a -> [a] -> Maybe Int
The elemIndex function returns the index of the first element in the given list which is equal (by ==) to the query element, or Nothing if there is no such element.
From package base elemIndex :: Eq a => a -> [a] -> Maybe Int

ghci 68> 4 `*elemIndex*` [1,2,3,4,5,6]
Just 3

ghci 69> 10 `*elemIndex*` [1,2,3,4,5,6]
Nothing

ghci 70> **import** *Data.Maybe* (*fromJust*)

```
ghci 71> fromJust (4 `elemIndex` [1,2,3,4,5,6])
    3
```

```
ghci 72> [1,2,3,4,5,6] !! fromJust (4 `elemIndex` [1,2,3,4,5,6])
    4
```

elemIndices is like *elemIndex*, only it returns a list of indices – in case the element we're looking for appears several times in our list.

Because we're using a list to represent the indices, we don't need a *Maybe* type: failure is represented as the empty list, which is very much synonymous to *Nothing*.

```
ghci 73> :t elemIndices
    elemIndices :: Eq a => a -> [a] -> [Int]
```

```
ghci 74> :! hoogle  -- info elemIndices
    Data.List elemIndices :: Eq a => a -> [a] -> [Int]
    The elemIndices function extends elemIndex, by returning the indices of all elements equal
    to the query element, in ascending order.
    From package base elemIndices :: Eq a => a -> [a] -> [Int]
```

```
ghci 75> ' ' `elemIndices` "Where are the spaces?"
    [5,9,13]
```

```
ghci 76> map ("Where are the spaces?"!!) (' ' `elemIndices` "Where are the spaces?")
    "   "
```

2.17 *findIndex* and *findIndices*

findIndex is like *find*, but it returns the index of the first element that satisfies the predicate wrapped in a *Maybe*.

findIndices returns the indices of all elements that satisfy the predicate in the form of a list.

```
ghci 77> findIndex (≡ 4) [5,3,2,1,6,4]
    Just 5
```

```
ghci 78> findIndex (≡ 7) [5,3,2,1,6,4]
    Nothing
```

```
ghci 79> [5,3,2,1,6,4] !! fromJust (findIndex (== 4) [5,3,2,1,6,4])
    4
```

```
ghci 80> findIndices (`elem` ['A'..'Z']) "Where Are The Caps?"
    [0,6,10,14]
```

```
ghci 81> map ("Where Are The Caps?" !!) (findIndices (`elem` ['A'..'Z']) "Where Are The Caps?")
    "WATC"
```

2.18 *zip3*, *zipWith3*, *zip4*, *zipWith4*, ...

We already covered *zip* and *zipWith*. We noted that they zip together two lists, either in a tuple or with a binary function (i.e., a function that takes two parameters).

But what if we want to zip together three lists? Or zip three lists with a function that takes three parameters? We have *zip3*, *zip4* etc. for that, and *zipWith3*, *zipWith4* etc. These variants go up to 7.

```
ghci 82> zipWith3 (\x y z -> x + y + z) [1,2,3] [4,5,2,2] [2,2,3]
    [7,9,8]
```

```
ghci 83> zip4 [2,3,3] [2,2,2] [5,5,3] [2,2,2]
    [(2,2,5,2),(3,2,5,2),(3,2,3,2)]
```

Just like with normal zipping, lists that are longer than the shortest list that's being zipped are cut down to size.

2.19 *lines* and *unlines*

lines is a useful function when dealing with files or input from somewhere. It takes a string and returns every line of that string in a separate list.

```
ghci 84> lines "first line\nsecond line\nthird line"
    ["first line","second line","third line"]
```

`'\n'` is the character for a unix newline. Backslashes have special meaning in Haskell strings and characters.

unlines is the inverse function of lines. It takes a list of strings and joins them together using a `'\n'`.

```
ghci 85> unlines ["first line","second line","third line"]
    "first line\nsecond line\nthird line\n"
```

2.20 *words* and *unwords*

words and *unwords* are for splitting a line of text into words or joining a list of words into a text.

```
ghci 86> words "hey these are the words in this sentence"
    ["hey","these","are","the","words","in","this","sentence"]
```

```
ghci 87> words "hey these           are    the words in this\nsentence"
    ["hey","these","are","the","words","in","this","sentence"]
```

```
ghci 88> unwords ["hey","there","mate"]
    "hey there mate"
```

2.21 *delete*, \\, *union* and *intersect*

delete takes an element and a list and deletes the first occurence of that element in the list.

```
ghci 89> delete 'h' "hey there ghang!"
    "ey there ghang!"
```

```
ghci 90> delete 'h' ∘ delete 'h' $ "hey there ghang!"
    "ey tere ghang!"
```

```
ghci 91> delete 'h' ∘ delete 'h' ∘ delete 'h' $ "hey there ghang!"
    "ey tere gang!"
```

\\ is the list difference function. It acts like set difference: for every element in the right-hand list, it removes the first matching element in the left one.

```
ghci 92> [1..10] \\ [2,5,9]
    [1,3,4,6,7,8,10]
```

```
ghci 93> "He is a big baby" \\ "big"
    "He s a i baby"
```

```
ghci 94> "He is a big baby" \\ " big"
    "Hes a i baby"
```

```
ghci 95> "He is a big baby" \\ "big "
    "Hes a i baby"
```

```
ghci 96> "He is a big baby" \\ "bbig  "
    "Hesa i aby"
```

Here's another example that makes the behavior of \\ very clear: doing $[1..10] \setminus\setminus [2,5,9]$ is like doing:

```
ghci 97> delete 2 ∘ delete 5 ∘ delete 9 $ [1..10]
    [1,3,4,6,7,8,10]
```

union also acts like a function on sets: it returns the union of two lists.

In particular, it goes over every element in the second list and appends it to the first one if it isn't already in yet.

Duplicates are removed from the second list (only).

```
ghci 98> "hey man" `union` "man what's up"
    "hey manwt'sup"
```

```
ghci 99> [1..7] `union` [5..10]
    [1,2,3,4,5,6,7,8,9,10]
```

intersect works like set intersection. It returns only the elements that are found in both lists.

```
ghci 100> [1..7] `intersect` [5..10]
    [5,6,7]
```

2.22 *insert*

insert takes an element and a list of elements that can be sorted and inserts it into a specific position in the list.

The position is determined as follows: *insert* starts at the beginning of the list, keeps going until it finds an element that's equal to or greater than the element that we're inserting, and it does the insertion right before that element.

```
ghci 101> insert 4 [3,5,1,2,8,2]
    [3,4,5,1,2,8,2]
```

```
ghci 102> insert 4 [1,3,4,4,1]
    [1,3,4,4,4,1]
```

The 4 is inserted right after the 3 and before the 5 in the first example and in between the 3 and 4 in the second example.

If we use *insert* to insert into a sorted list, the resulting list will still be sorted.

```
ghci 103> sort [3,5,1,2,8,2]
    [1,2,2,3,5,8]
```

```
ghci 104> insert 4 $ sort [3,5,1,2,8,2]
    [1,2,2,3,4,5,8]
```

```
ghci 105> insert 'g' $ ['a'..'f'] ++ ['h'..'z']
    "abcdefghijklmnopqrstuvwxyz"
```

2.23 *genericLength*, *genericTake*, *genericDrop*, *genericSplitAt*, *genericIndex*, *genericReplicate*

What *length*, *take*, *drop*, *splitAt*, !! and *replicate* have in common is that they take an *Int* as one of their parameters or return an *Int*.

But they could be more generic if they just took any type that's part of the *Integral* or *Num* typeclasses (depending on the functions).

That's why *Data.List* provides their more generic equivalents *genericLength*, *genericTake*, *genericDrop*, *genericSplitAt*, *genericIndex* and *genericReplicate*, respectively.

For instance, *length* has this type signature:

```
ghci 106> :t length
    length :: [a] -> Int
```

If we try to get the average of a list of numbers by doing **let** $xs = [1..6]$ **in** *sum xs / length xs*, we get a type error, because we can't use / with an *Int*.

```
ghci 107> let xs = [1..6] in sum xs / length xs
```

genericLength, on the other hand, has this type signature:

```
ghci 108> :t genericLength
    genericLength :: Num i => [b] -> i
```

Because a *Num* can act like a floating point number, getting the average like this works out just fine:

```
ghci 109> let xs = [1..6] in sum xs / genericLength xs
    3.5
```