Weekly Assignments 1

To be submitted: Friday, 19 January 2018

Note that some tasks may deliberately ask you to look at concepts or libraries that we have not yet discussed in detail. But if you are in doubt about the scope of a task, by all means ask.

Please try to write high-quality code at all times! This means in particular that you should add comments to all parts that are not immediately obvious.

W1.1 Packaging

Prepare a Cabal package to contain all your solutions so that it can easily be built using cabal-install or stack.

Note that one package can contain one library (with arbitrarily many modules) and possibly several executables and test suites.

Please include a `README` file in the end explaining clearly where within the package the solutions to the individual subtasks are located.

Please do *NOT* try to upload your package to Hackage.

W1.2 Permutations

Reimplement the function `permutations` from the `Data.List` module. Document your function, i.e., explain the algorithm you are using.

W1.3 Merge sort

Implement merge sort in Haskell (on lists). The idea of merge sort is to split a list into two more or less equal parts, sort them recursively, and then merge two sorted lists using a dedicated merge function.

W1.4 Tail recursion

Consider the following code

```
data Tree a = Leaf a | Node (Tree a) (Tree a)

splitleft :: Tree a -> (a, Maybe (Tree a))
splitleft (Leaf a)   = (a, Nothing)
splitleft (Node l r) =
```

```
  case splitleft l of
    (a, Nothing) -> (a, Just r)
    (a, Just l') -> (a, Just (Node l' r))
```

This takes a tree and splits off the left-most leaf of the tree, and returns it together with the rest of the tree (unless the original tree was a single leaf, in which case it returns `Nothing`).

Try to write a *tail-recursive* version of `splitleft`. This may require introducing an auxiliary function.

W1.5 Tries

A trie (sometimes called a prefix tree) is an implementation of a finite map for structured key types where common prefixes of the keys are grouped together. Most frequently, tries are used with strings as key type. As hashes are also strings, such as trie can also be used to map hashes to values.

In Haskell, such a trie can be represented as follows:

```
data Trie a = Fork (Maybe a) (Map Char (Trie a))
```

A node in the trie contains an optional value of type a. If the empty string is in the domain of the trie, the associated value can be stored here. Furthermore, the trie maps single characters to subtries. If a key starts with one of the chracters contained in the map, then the rest of the key is looked up in the corresponding subtrie.

The `Map` type is from `Data.Map` in the containers package.

The following is an example trie that maps "f" to 0, "foo" to 1, "bar" to 2 and "baz" to 3:

```
Fork Nothing
  (fromList
    [('b', Fork Nothing
             (fromList [('a', Fork Nothing
                                (fromList [('r', Fork (Just 2) empty)
                                          ,('z', Fork (Just 3) empty)
                                          ]
                                )
                        )
                       ]
             )
     )
    ,('f', Fork (Just 0)
             (fromList [('o', Fork Nothing
                                (fromList [('o', Fork (Just 1) empty)])
                        )
```

```
                    ]
            )
     )
    ]
 )
```

We call a trie *valid* if all its subtries are non-empty and valid.

Note that even though this definition might sound circular, it is in fact not: For example, the definition shows that a trie *without subtries* is valid, because in this case, the condition is trivially true.

In particular, the trie `Fork Nothing empty` is valid, and it is the only valid trie that does not contain any values.

Note also that the definition implies that *all subtries of a trie are non-empty.*

The implementation should obey the invariant of validity, i.e. *all created tries should be valid.*

Implement the following interface:

```
empty  :: Trie a -- produces an empty trie
null   :: Trie a -> Bool -- checks if a trie is empty
valid  :: Trie a -> Bool -- checks if a trie adheres to the invariant
insert :: String -> a -> Trie a -> Trie a -- inserts/overwrites a key-value pair
lookup :: String -> Trie a -> Maybe a -- looks up the value associated with the key
delete :: String -> Trie a -> Trie a -- deletes the key if it exists
```

Furthermore, make your trie an instance of the `Functor` and `Foldable` classes. Using deriving for these is ok, but please convince yourself that the derived functions work as expected.

In what way is your trie specific to `String`? Can you generalize it?

W1.6 Testing

Define a test suite for the tries. Write a QuickCheck generator for tries. Use QuickCheck to verify that all trie operations maintain the invariant. Try to think of at least three other interesting properties and test them via QuickCheck.

Optional: Try to document the interface of your library with Haddock comments. Look up proper Haddock markdown syntax. Add example GHCi interactions to your Haddock comments, and use `doctest` to test these as well.

W1.7 SHA256 Hashing

Have a look at the `cryptonite` package on Hackage which contains several hashing and encryption functions that will prove useful for implementing

cryptocurrency-related code. For now, the goal is simply to figure out how to use this library to compute a SHA256 hash of a given `ByteString` (from `Data.ByteString` in the `bytestring` package).

Then try to implement a basic form of the unix tool sha256sum. This interprets all command line arguments as file names. Automatically filter out any filenames that do not correspond to regular files (e.g. files that do not exist, or names of directories). For the others, compute a SHA256 hash for every file and then print the hash together with the filename.

Look at the `directory` package for functions to test whether a file exists. Figure out yourself how you can get access to command line arguments.

If you can, verify that your executable produces the correct result on a number of files of which you know the correct SHA256 hash.

Optional: If you have time left, try to implement flags that allow you to select one of several different hashes, so that you can e.g. also compute MD5, SHA1, or SHA512 from the same program.

W1.8 Teletype IO

Consider the following datatype:

```
data GP a =
    End a
  | Get (Int -> GP a)
  | Put Int (GP a)
```

A value of type GP can be used to describe programs that read and write integer values and return a final result of type `a`. Such a program can end immediately (`End`). If it reads an integer, the rest of the program is described as a function depending on this integer (`Get`). If the program writes an integer (`Put`), the value of that integer and the rest of the program are recorded.

The following expression describes a program that continuously reads integers and prints them:

```
echo = Get (\n -> Put n echo)
```

Subtask 1.8.1

Write a function

```
run :: GP a -> IO a
```

that can run a GP-program in `IO`. A `Get` should read an integer from the console, and `Put` should write an integer to the console.

Here is an example run from GHCi:

```
GHCi> run echo
? 42
42
? 28
28
? 1
1
? -5
-5
? Interrupted.
GHCi>
```

[To better distinguish inputs from outputs, this version of run prints a question mark when expecting an input.]

Subtask 1.8.2

Write a `GP`-program add that reads two integers, writes the sum of the two integers, and ultimately returns `()`.

Subtask 1.8.3

Write a `GP`-program accum that reads an integer. If the integer is `0`, it returns the current total. If the integer is not `0`, it adds the integer to the current total, prints the current total, and starts from the beginning.

Subtask 1.8.4

Instead of running a GP-program in the IO monad, we can also simulate the behaviour of such a program by providing a (possibly infinite) list of input values. Write a function

```
simulate :: GP a -> [Int] -> (a, [Int])
```

that takes such a list of input values and returns the final result plus the (possibly infinite) list of all the output values generated.

Optional: Rewrite `simulate` to use the state monad.

Subtask 1.8.5

Define a sensible instance of Monad (and Functor and Applicative) for GP. Rewrite at least one of the `GP`-programs above using do notation.