Programming in Haskell – Homework Assignment 5

UNIZG FER, 2013/2014

Handed out: December 2, 2013. Due: December 7, 2013 at 23:59

Note: Define each function with the exact name and type specified. You can (and in most cases should) define each function using a number of simpler functions. Provide a type signature above each function definition and comment the function above the type signature. Unless said otherwise, a function may not cause runtime errors and must be defined for all of its input values. Use the `error` function for cases in which a function should terminate with an error message, and return the exact error message specified. Problems marked with a star ($\star$) are optional.

1. Define an explicitly recursive `concatMap'` that behaves exactly like `Prelude.concatMap` but doesn't rely on `Prelude.concat` nor `Prelude.map`.

   ```
   concatMap' :: (a -> [b]) -> [a] -> [b]
   ```

2. Define an explicitly recursive `reduce` that reduces an input list to a single element given a reducing function and a starting element. The reducing function is applied to the first element and the initial element. The so-obtained result is then combined with the second element of the list using the same reduction function. The new result is combined with the third element of the list, and so forth, until the end of the list is reached.

   ```
   reduce :: (a -> b -> a) -> a -> [b] -> a
   reduce (+) 0 [1..4] ⇒ 10
   reduce (-) 0 [1..4] ⇒ -10
   reduce (\a b -> a ++ ", " ++ b) "1" ["2", "3", "4"] ⇒ "1, 2, 3, 4"
   reduce (\s x -> length x `max` s) minBound ["moo", "doodle", "mao"]
   ⇒ 6
   reduce (+) 3 [1] ⇒ 4
   reduce (+) 0 [] ⇒ 0
   ```

3. Define a `reduce1` function that behaves like `reduce` but assumes the input list contains at least two elements and so doesn't take a starting element as input. (Hint: use `reduce`.)

   ```
   reduce1 :: (a -> a -> a) -> [a] -> a
   reduce1 (+) [1..4] ⇒ 10
   reduce1 (\a b -> a ++ ", " ++ b) ["1", "2", "3", "4"] ⇒ "1, 2, 3, 4"
   reduce1 (+) [7] ⇒ 7
   reduce1 (+) [] ⇒ error "reduce1 got an empty list"
   ```

4. Define `shortestPath` to compute the length of the shortest path between two vertices. If the vertices are not connected, return `-1`. To solve this problem, use the

breadth-first search (BFS) algorithm. You will need two data structures: a `Queue` to store the vertices that need to be processes and a `Map` to store the values (the shortest path length) of the already processed vertices. For the sake of simplicity, you should implement them using lists:

```
type Queue = [Vertex]
type Map = [(Vertex, Integer)]
```

(a) Define `popFromQ` that pops the first element from the queue. Return the first element and the queue without it.
`popFromQ :: Queue -> (Vertex, Queue)`

(b) Define `pushToQ` that pushes the given element to the end of the queue.
`pushToQ :: Queue -> Vertex -> Queue`

(c) Define `isEmptyQ` that checks if the queue is empty.
`isEmptyQ :: Queue -> Bool`

(d) Define `isInQ` that checks if an element is in the queue.
`isInQ :: Queue -> Vertex -> Bool`

(e) Define `getQSingleton` that returns a new queue with one given element.
`getQSingleton :: Vertex -> Queue`

(f) Define `getValFor` that returns the value for a given vertex. If the vertex is not stored, return `-1` (this indicates the vertex has not yet been processed).
`getValFor :: Map -> Vertex -> Integer`

(g) Define `putKeyVal` that inserts a given key (vertex in this case) and a value into map. If the key already exists, overwrite it.
`putKeyVal :: Map -> Vertex -> Integer -> Map`

(h) Define `isKeyInMap` that checks if a given key is in the map.
`isKeyInMap :: Map -> Vertex -> Bool`

(i) Define `getMapSingleton` that returns a new map with a single `(key, value)` pair.
`getMapSingleton :: Vertex -> Integer -> Map`

(j) Well done, now we have our implementation of `Queue` and `Map`! From now on you can use `Queue` and `Map` through the interface you have just implemented. If you wish, you can add some more methods. Now implement BFS using these structures (`shortestPath` defined above)!

5. Implement a couple of functions that operate on a file system. The filesystem structure is implemented as a list of `(dir_absolute_path, list_of_contents)` pairs, representing the paths and their corresponding contents:

   ```
   type FileSystem = [(FilePath, [FilePath])]
   ```

   For example:

   ```
   testFS = [("/", ["dir1/", "homework.hs"]),
             ("/dir1/", ["dir2/"]), ("/dir1/dir2/", [])]
   ```

   The state of the filesystem is represented as a pair `(curr_dir_absolute_path, file_system)`, representing the current directory and the filesystem structure:

```
type FileSystemState = (FilePath, FileSystem)
```

To make things simpler, assume that the directory and the file names don't contain the '/' character as part of their name. In `FileSystem` and `FileSystemState`, all directories will have an '/' added at the end. For every directory, there will be element in `FileSystem` describing it. You can expect that `FileSystemState` will always be valid (you do not have to handle any errors that could be caused by this not being the case).

(a)
```
pwd :: FileSystemState -> String
pwd ("/home/user/", _) ⇒ "/home/user"
```

(b)
```
ls :: FileSystemState -> String
ls ("/", testFS) ⇒ "dir1/ homework.hs"
ls ("/dir1/", testFS) ⇒ "dir2/"
ls ("/dir1/dir2/", testFS) ⇒ ""
```

(c) Define `buildAbsPath` that takes an absolute path of a directory (ending in '/') and some absolute or relative path and returns a new absolute path (not ending in '/', except for root "/"). This function will be helpful for later problems. (Hint: solve it recursively.)

```
buildAbsPath :: String -> String -> String
buildAbsPath "/" "dir1/dir2" ⇒ "/dir1/dir2"
buildAbsPath "/dir1/" "." ⇒ "/dir1"
buildAbsPath "/dir1/" "./dir2/.." ⇒ "/dir1"
buildAbsPath "/dir1/dir2/" "../.." ⇒ "/"
buildAbsPath "/dir1/dir2/" "dir3/file.txt"
⇒ "/dir1/dir2/dir3/file.txt"
buildAbsPath _ "/a/b/c" ⇒ "/a/b/c"
```

(d) Define a directory changing function:

```
cd :: FileSystemState -> String -> FileSystemState
cd ("/", testFS) "dir1" ⇒ ("/dir1/", testFS)
cd ("/", testFS) "dir1/dir2" ⇒ ("/dir1/dir2/", testFS)
cd ("/dir1/", testFS) "dir1"
⇒ error "There is no such directory"
cd ("/dir1/", testFS) "homework.hs"
⇒ error "There is no such directory"
cd ("/dir1/", testFS) "." ⇒ ("/dir1/", testFS)
cd ("/dir1/", testFS) "./dir2" ⇒ ("/dir1/dir2/", testFS)
cd ("/dir1/dir2/", testFS) ".." ⇒ ("/dir1/", testFS)
cd ("/dir1/dir2/", testFS) "../.." ⇒ ("/", testFS)
cd ("/dir1/dir2/", testFS) "../../.." ⇒ ("/", testFS)
cd (_, testFS) "/dir1/dir2" ⇒ ("/dir1/dir2/", testFS)
cd (_, testFS) "/" ⇒ ("/", testFS)
```

(e) Define `rm` that removes file or directory with all its contents.

```
rm :: FileSystemState -> String -> FileSystemState
rm ("/", testFs) "dir1" ⇒ ("/", [("/", ["homework.hs"])])
rm ("/", testFs) "dir5" ⇒ error "There is no such file/directory"
rm ("/", _) "/" ⇒ error "Can not remove current directory"
rm (_, _) "." ⇒ error "Can not remove current directory"
```

```
rm (_, _) ".." ⇒ error "Can not remove parent directory"
rm (_, testFs) "/dir1/dir2" ⇒ (_, [("/", ["dir1/", "homework.hs"]),
("/dir1/", [])])
rm ("/", testFs) "homework.hs"  ⇒   [("/", ["dir1/"]), ("/dir1/",
["dir2/"]), ("/dir1/dir2/", [])]
```

Corrections

```
v2:     Updated entire homework.
v3:  5: Removed type FilePath
    5e: Fyle => File
v4: 5c: Output must not contain '/' at the end (before it had to for directories).
        Added example for absolute path.
```