Assignment 5

In this assignment we will write an engine for regular expressions. A nice interactive tutorial for regular expressions can be found here[1], if you have never dealt with regular expressions before.

Here are the regular expression behaviors we will implement:

- You can match any of a specific set of characters, the equivalent of the regular expression `[abcd]`.
- You can match the "or" of two regexs, like `(ab|c)` matches either `ab` or `c`, whichever it encounters.
- You can ask for a regex to be matched one or more times, using `+`, or 0 or more times using `*`. For example `(ab)+` will match strings like `ab`, `abab`, `ababab`. These behave in a *greedy* way: They will match the longest possible number of times.
- You can optionally match a regex with a `?`. For example `a?b` will match both `b` and `ab`.
- You can match any letter, or any digit.
- You can *capture* some parts of the match with capture groups. For example `(ab(c))d` against the string `"abcd"` will produce two capture groups, the first containing `abc` and the second containing `c`.

For now we will code the regular expressions directly into suitable Haskell code (i.e. we will not try to parse something like `a?`, we will instead have a Haskell statement like `optional (char 'a')`.

We will represent regular expressions as functions:

> A regular expression for us is a function that takes as input a string and attempts to match the regular expression at the beginning of the string, consuming some amount of the string (not necessarily all). The result consists of:
>
> - The string containing the matched portion.
> - A list of strings corresponding to the captured parts.
> - The string that is remaining.
>
> Moreover the function returns a list of such results, accounting for all the different ways in which it can match. An empty list means that the regular expression does not match.

[1] https://regexone.com/

While a regular expression by its definition must match at the beginning of a string, we will also write a function that allows a regular expression to match anywhere in the string. In that case the result is a list of 4-tuples, each tuple containing the three above items as well as the amount of the string *before* a match starts.

The following type definitions offer us some basic types to work with:

```
type Captures   = [String]
type Match      = String
type After      = String
type Before     = String
type Result     = [(Match, Captures, After)]
type FullResult = [(Before, Match, Captures, After)]

type RegEx      = String -> Result
```

The simplest regular expression is called fails: Given any string, it fails to match (remember that regular expressions are actually functions):

```
fails :: RegEx
fails = \s -> []
-- Alternative definition:    fails s = []
```

Another simple regular expression is anyChar: It simply matches any character (but fails for an empty string). For example anyChar "yup" = [("y", [], "up")]. You will need to provide an implementation for this and other functions to be described shortly.

An important function that you will provide is matchAnywhere. This function takes a regular expression and a string, and it attempts to match the expression starting anywhere in the string. It returns a FullResult. For example matchAnywhere anyChar "yup" will return three results, for the three possible places where the anyChar regular expression can match.

Starting code and some initial tests are at the bottom of this file. You can run the tests via:

```
ghc assignment4.hs
./assignment4 tests
```

You can also see the effect of the regular expressions, once you have implemented all the functions, by feeding a text file in via one of the following:

```
ghc assignment4.hs
./assignment4 words < testFile.txt
./assignment4 numbers < testFile.txt
./assignment4 wordmatches < testFile.txt
./assignment4 numbermatches < testFile.txt
```

Make sure your terminal supports colors. You can also check out a quick string by something like:

```
echo "hello 123 there" | ./assignment4 numbers
echo "hello 123 there" | ./assignment4 words
echo "hello 123 there" | ./assignment4 numbermatches
echo "hello 123 there" | ./assignment4 wordmatches
```

Here is a list of the functions to implement, and their brief specifications (make sure to check the examples):

1. anyChar is a regular expression value. It is a function that matches the first character of a non-empty string, without "capturing" anything, and fails on an empty string.

2. epsilon is a regular expression value. It is a function that matches any string by matching "nothing".

3. Next we will write some helper functions. One is called succeeded. It is a function that takes a Result value as input and returns a boolean indicating whether this is a successful result (i.e. non-empty list) or not. Implement this as a composition of the list method null and the boolean operator not.

4. Now we have a function orElse. It takes as input two Result values and returns a Result value as follows: If the second result is the empty list then returns the first result. If the first result is an empty list then it returns the second result. If both result lists are non-empty, then it compares the heads of both lists and puts the largest value first then recursively continues with the rest of the list (if the two values are equal then only keep one). You can use the compare function between two results, then do a case ... of on the three possible comparison outcomes of LT, EQ, GT. This effectively orders the results starting with the longest match.

5. Now we return to implementing more regular expressions. char is a function that is given a Char value and returns a RegEx value. That value is in turn a function that takes a string and matches it only if the first character in the string is the provided character. You will need a pattern-match for the empty list and two guard cases for the non-empty list.

6. The function anyOne is given a list of characters and returns a RegEx that matches any one of those characters.

7. The regular expressions digit and letter match any digit and any letter respectively. Use anyOne to define them, along with providing suitable list ranges (like ['a'..'z']). Make sure to also handle uppercase letters. Also implement a regular expression space that matches any normal space, tab or newline character.

8. Define the ||| (three lines) operator to be given two regular expressions and to return the regular expression that matches either one (but matching the left one first). You can define it as a function that to a given string returns the orElse of the two results from its two regular expressions.

9. Define the andThen operator which takes two regular expressions and returns a regular expression as follows: It matches a string when the first regular expression matches (some part of) the string and the second regular expression matches remaining of the string. It should return a list of all such possible matches. You can do this with a list comprehension. This one is somewhat challenging. You will need to go through each match of the first regular expression, and for each

one of those go through each match of the second regular expression against the "after" string from the first match, then collect the results.

10. Implement a function chained that takes as input a list of regular expressions and returns a single regular expression, by "chaining" those regular expressions using andThen. An empty list should correspond to epsilon. You can do this using a fold. Also implement a function options which takes as input a list of regular expressions and returns a single regular expression by using those regular expressions as alternatives (any one can match) in the order they appear. The empty list case should correspond to fails.

11. Next we write a function optional. In regular expression language this is the question mark (?). It takes as input a regular expression and returns a regular expression. It matches the regular expression if it can, otherwise it matches an empty string (via epsilon). You can write it by simply using the ||| operator on the provided regular expression and epsilon.

12. Now we write a function capture. It takes as input a regular expression and returns a regular expression that does the same match as the provided regular expression, except that it adds the match to the front of the captures list. You can do this with a map or list comprehension.

13. Next we define two functions that depend on each other. plus and star. They correspond to the regular expression operations + and $*$. They both are given a regular expression and return a regular expression. The plus one requires that the provided regular expression is matched one or more times, while the star requires it to be matched zero or more times. Define the plus or a regular expression to equal the regular expression followed by the star of the regular expression (in regular expression speak we define r+ = rr$*$). Then define star of a regular expression as plus of the expression with alternative (|||) the epsilon expression (in regular expression speak we would say r$*$=r+|epsilon). You need to ensure that the match is done "greedily": It should try to match as many occurences of the regular expression as possible, and only fall back to the epsilon match if needed. The order of the expressions around ||| will determine that.

14. Now we will reap the benefits of our work. Define two regular expressions, word and int which match a whole word (one or more letters) and an integer (one or more digits) respectively. You can do this easily using earlier functions.

15. Next, we will write the function matchAnywhere. This takes as input a regular expression and a string, and it tries to match the regular expression anywhere within the string. It returns a FullResult (list of 4-tuples), prioritizing matches that start earlier in the string. This is somewhat complicated. Handle the case of the empty string separately. For a non-empty string, concatenate together two lists: The matches starting at the beginning of the string (adjusted to be 4-tuples with an "empty string" before them) and the matches that skip the first character and match anywhere in the remaining string (adjusted to have the correct "before" part). Each of these can be done via a list comprehension.

16. Now we write a function allMatchesFull which repeatedly matches the regular expression, choosing the earliest and longest match each time, then keeps going with the remaining string. You should use a case ... of statement on the result of matchAnywhere of the regular expression on the string. If the result is an empty list, meaning no matches, then return an empty list. Otherwise, look at the first match (it's the earliest and longest possible by our earlier work), recursively call allMatches on the "after" part of that match, and prepend the matched 4-tuple to that result.

17. Using the previous function, write a function allMatches that instead returns a list of all the matched strings (rather than the whole 4-tuples).

18. Write a function splitOn which takes as input a regular expression and a string and "splits" the string on matches of the regular expression, returning a list of the surrounding text. You can implement this starting with allMatchesFull and then collecting together all the "before" pieces from the 4-tuples, as well as the "after" piece of the very last 4-tuple. You will likely want to use a helper function that picks those pieces, then combine it with allMatchesFull. The empty-list case of the helper is special here because it suggests no match, and would therefore return the entire initial string as is (as a one-element list).

And we are done! You can test your handiwork by running the script on a file as described earlier.

Start code:

```
module Main where

import Test.HUnit
import Control.Monad (when)
import Data.Char (isAlpha, toUpper, isPunctuation)
import System.Environment (getArgs)

type Captures   = [String]
type Match      = String
type After      = String
type Before     = String
type Result     = [(Match, Captures, After)]
type FullResult = [(Before, Match, Captures, After)]

type RegEx      = String -> Result

fails :: RegEx
fails = \s -> []

-- Your code here



tests = TestList [
   fails ""                    ~?= [],
   fails "yup"                 ~?= [],
   anyChar "yup"               ~?= [("y", [], "up")],
```

```
anyChar ""                    ~?= [],
epsilon "yup"                 ~?= [("", [], "yup")],
epsilon ""                    ~?= [("", [], "")],
succeeded []                  ~?= False,
succeeded [("", [], "")]      ~?= True,
[("a", [], "b")] `orElse`
   [("", [], "ab")]           ~?= [("a", [], "b"), ("", [], "ab")],
[("", [], "ab")] `orElse`
   [("a", [], "b")]           ~?= [("a", [], "b"), ("", [], "ab")],
[("", [], "ab")] `orElse`
   [("ab", [], ""),
    ("a", [], "b")]           ~?= [("ab", [], ""), ("a", [], "b"),
                                   ("", [], "ab")],
[("a", [], "b")] `orElse`
   [("ab", [], ""),
    ("", [], "ab")]           ~?= [("ab", [], ""), ("a", [], "b"),
                                   ("", [], "ab")],
(char 'a') "abc"              ~?= [("a", [], "bc")],
(char 'A') "abc"              ~?= [],
(char 'a') ""                 ~?= [],
(anyOne "dae") "abc"          ~?= [("a", [], "bc")],
(anyOne "bde") "abc"          ~?= [],
digit  "12a"                  ~?= [("1", [], "2a")],
digit  "a23"                  ~?= [],
letter "a23"                  ~?= [("a", [], "23")],
letter "ABC"                  ~?= [("A", [], "BC")],
letter "2a3"                  ~?= [],
space  "␣23"                  ~?= [("␣", [], "23")],
space  "A␣C"                  ~?= [],
space  "\t␣g"                 ~?= [("\t", [], "␣g")],
space  "fgr"                  ~?= [],
space  ""                     ~?= [],
(digit ||| letter) "abc"      ~?= [("a", [], "bc")],
(digit ||| letter) "1bc"      ~?= [("1", [], "bc")],
(digit `andThen` letter)
   "abc"                      ~?= [],
(digit `andThen` letter)
   "1bc"                      ~?= [("1b", [], "c")],
(digit `andThen` letter)
   "123"                      ~?= [],
chained [
   digit, letter, digit
   ] "2a3b"                   ~?= [("2a3", [], "b")],
options [
   digit, letter,
   chained [
      letter, letter,
      letter]
   ] "abc"                    ~?= [("abc", [], ""), ("a", [], "bc")],
(optional digit) "123"        ~?= [("1", [], "23"), ("", [], "123")],
(optional digit) "a23"        ~?= [("", [], "a23")],
chained [
   char 'a',
   capture (char 'b')
   ] "ab3"                    ~?= [("ab", ["b"], "3")],
(capture $
```

```
      chained [
         char 'a',
         capture (char 'b')
         ]) "ab3"                ~?= [("ab", ["ab", "b"], "3")],
   (plus digit) "12b"            ~?= [("12", [], "b"), ("1", [], "2b")],
   (plus digit) "a12b"           ~?= [],
   (star digit) "12b"            ~?= [("12", [], "b"), ("1", [], "2b"),
                                      ("", [], "12b")],
   (star digit) "a12b"           ~?= [("", [], "a12b")],
   word "acme␣star"              ~?= [("acme", [], "␣star"),
                                      ("acm", [], "e␣star"),
                                      ("ac", [], "me␣star"),
                                      ("a", [], "cme␣star")],
   int "123a"                    ~?= [("123", [], "a"),
                                      ("12", [], "3a"),
                                      ("1", [], "23a")],
   matchAnywhere letter "12a3"   ~?= [("12", "a", [], "3")],
   matchAnywhere letter ""       ~?= [],
   matchAnywhere
      (optional letter) ""       ~?= [("", "", [], "")],
   matchAnywhere word "2abc3"    ~?= [("2", "abc", [], "3"),
                                      ("2", "ab", [], "c3"),
                                      ("2", "a", [], "bc3"),
                                      ("2a", "bc", [], "3"),
                                      ("2a", "b", [], "c3"),
                                      ("2ab", "c", [], "3")],
   allMatchesFull
      letter "1a23b"             ~?= [("1", "a", [], "23b"),
                                      ("23", "b", [], "")],
   allMatches word "1ab2cd5"     ~?= ["ab", "cd"],
   splitOn int "1ab2cd5"         ~?= ["", "ab", "cd", ""],
   splitOn int "abc"             ~?= ["abc"],
   splitOn space "big␣bad␣wolf" ~?= ["big", "bad", "wolf"]
   ]

printMatches :: RegEx -> String -> IO ()
printMatches re s = let ms = allMatchesFull re s
    in do printResults ms
          putStrLn $ "Total␣matches:␣" ++ show (length ms)

printOnlyMatches :: RegEx -> String -> IO ()
printOnlyMatches re s = sequence_ $ map putStrLn $ allMatches re s

printResults :: FullResult -> IO ()
printResults []                 = return ()
printResults ((b,m,_,a):rest)  = do
      putStr b
      putMatch m
      if null rest
         then putStrLn a
         else printResults rest

putMatch :: String -> IO ()
putMatch m = do
   -- Set background and foreground colors
   -- See https://en.wikipedia.org/wiki/ANSI_escape_code#CSI_codes
```

```
    putStr "\x1b[32m"
    putStr "\x1b[44m"
    putStr m
    -- Set background and foreground colors
    putStr "\x1b[39m"
    putStr "\x1b[49m"

main :: IO ()
main = do
    args <- getArgs
    txt <- getContents
    case args of
       ("tests" : _) -> do runTestTT tests
                           return ()
       ("numbers" : _) -> printMatches int txt
       ("words" : _)  -> printMatches word txt
       ("wordmatches": _) -> printOnlyMatches word txt
       ("numbermatches": _) -> printOnlyMatches int txt
```