Introduction to Programming: Assignment 2

Due: September 23, 2022. 11.59 pm

Instructions:
Submit your solution in a single file named `cmiemailid.hs` on Moodle. For example, if I were to submit a solution, the file would be called `spsuresh.hs`. You may define auxiliary functions in the same file, but the solutions should have the function names specified by the problems.

---

1. A *segment* of a list *xs* of length n is any sublist of the form $xs[i \cdots (j-1)]$, with $0 \leqslant i \leqslant j \leqslant n$. (In case $i = j$, the segment is the empty list.)

   Write a program

   ```
   segments :: [a] → [[a]]
   ```

   that produces all the segments in a given list. (The order in which the segments are to be listed is indicated by the sample cases below. Notice that the empty list is included only once, but other lists can repeat, as the third example below shows:)

   Sample cases:

   ```
   segments []          = [[]]
   segments [0]         = [[0],[]]
   segments [0..3]      = [[0],[0,1],[0,1,2],[0,1,2,3],
                            [1],[1,2],[1,2,3],
                            [2],[2,3],
                            [3],
                            []]
   segments [0,1,0,1,0,1,2]
                        = [[0],[0,1],[0,1,0],[0,1,0,1],[0,1,0,1,0],
                                 [0,1,0,1,0,1],[0,1,0,1,0,1,2],
                            [1],[1,0],[1,0,1],[1,0,1,0],[1,0,1,0,1],
                                 [1,0,1,0,1,2],
                            [0],[0,1],[0,1,0],[0,1,0,1],[0,1,0,1,2],
                            [1],[1,0],[1,0,1],[1,0,1,2],
                            [0],[0,1],[0,1,2],
                            [1],[1,2],
                            [2],
                            []]
   ```

2. An *uprun* of a list *xs* is a maximal nonempty segment that is sorted in ascending order (i.e. it is a segment $xs[i \cdots (j-1)]$ such that $xs[k] \leqslant xs[k+1]$ for $i \leqslant k < j-1$, and $xs[i-1] > xs[i]$ and $xs[j-1] > xs[j]$.

   Write a program

   ```
   upRuns :: Ord a ⇒ [a] → [[a]]
   ```

   that produces a list of all upruns in the input list.

   Sample cases:

   ```
   upRuns []    = []
   upRuns [0]   = [[0]]
   upRuns [0 .. 4]
                = [[0,1,2,3,4]]
   upRuns [0,1,2,3,5,4]
                = [[0,1,2,3,5],[4]]
   upRuns [0,1,0,1,0,1]
                = [[0,1],[0,1],[0,1]]
   upRuns [0,1,2,3,0,1,0,1,2,3,4]
                = [[0,1,2,3],[0,1],[0,1,2,3,4]]
   upRuns [0,1,2,3,0,0,0,1,1,0,1,2,3,4]
                = [[0,1,2,3],[0,0,0,1,1],[0,1,2,3,4]]
   upRuns [5,4 .. 0]
                = [[5],[4],[3],[2],[1],[0]]
   ```

3. This problem is about the word game **Stackle**, available at `https://www.stackle.fun`. In this game, you are given two 5-letter words at the start. The aim is to build as long a *stack of words* as possible. The stack is grown by adhering to these rules:

   (a) Each word has 5 letters.

   (b) No letter repeats in any word.

   (c) To go from one word to the next, one eliminates a letter and introduces a new letter (and perhaps jumbles the order).

   (d) Eliminated letters cannot be used again.

(e) Each word is a valid word in the Stackle dictionary. (We do not have access to the Stackle dictionary, but we have an approximation in the file `Dict.hs`.)

Note that since we start with a 5-letter word, and in each move we eliminate a letter, the stack can have a maximum length of 22.

There is a further *availability constraint*. There are three lists of letters given, call them `l1`, `l2` and `l3`. Each list can possibly be empty, and `l1` and `l2` usually contain at most 4 letters, and `l3` has at most 1 letter, and the lists are mutually disjoint. Letters in `l1` can appear only at the tenth word of the stack or later, letters in `l2` can appear only at the eighteenth word of the stack or later, and letters in `l3` can appear only at the twenty second word.

We define the following two *type synonyms* (a simpler name for an existing type, to improve readability):

```
type Game = (String, String, ([Char], [Char], [Char]))
type Solution = [String]
```

Write a program

```
checkStack :: Game → Solution → Bool
```

such that `checkStack gm sol` returns `True` if `sol` is a valid stack of words according to the above rules, and returns `False` otherwise.

Sample cases:

```
gm1, gm2, gm3 :: Game
gm1 = ("round", "mound", (['a'], ['i','t'], ['k']))
gm2 = ("bloke", "block", ([], [], []))
gm3 = ("flunk", "funks", (['a'], ['g','r','x','z'], []))

sol1, sol2, sol3, nosol1a, nosol1b, nosol1c, nosol1d :: Solution
sol1 = [
    "round", "mound", "found", "wound", "hound", "dough"
  , "cough", "chugs", "cushy", "saucy", "quays", "squab"
  , "abuse", "pause", "japes", "paxes", "pales", "lazes"
  , "tales", "tiles", "lives", "likes"]
sol2 = [
    "bloke", "block", "black", "slack", "racks", "czars"
  , "chars", "scarf", "cards", "drams", "yards", "daisy"
```

```
  , "qadis", "wadis", "divas", "staid", "taxis", "pitas"
  , "satin", "aunts", "gaunt", "jaunt"]
sol3 = [
    "flunk", "funks", "bunks", "hunks", "junks", "stunk"
  , "tunes", "quest", "suety", "cutes", "mutes", "mites"
  , "tomes", "stove", "stave", "waste", "paste", "gates"
  , "zetas", "taxes", "dates", "rates"]
nosol1a = ["round", "found", "gound"]
        -- "gound" is not a valid word in the dictionary
nosol1b = ["round", "crown"]
        -- "We are eliminating both 'u' and 'd'
nosol1c = ["round", "hound", "dough", "tough"]
        -- 't' is used before the eighteenth word
nosol1d = ["round", "hound", "dough", "cough", "couch"]
        -- 'c' is repeated twice in the last word
nosol1e = ["mound", "round", "found", "wound"]
        -- First two words must exactly match the
        -- starting words given in the game
        -- in the correct order.
partsol1 = ["round", "mound", "hound", "dough", cough"]
        -- It is okay to stop short.
        -- This is a valid partial solution.

checkStack gm1 sol1         = True
checkStack gm1 partsol1     = True
checkStack gm1 nosol1a      = False
checkStack gm1 nosol1b      = False
checkStack gm1 nosol1c      = False
checkStack gm1 nosol1d      = False
checkStack gm1 nosol1e      = False
checkStack gm2 sol2         = True
checkStack gm3 sol3         = True
```

4. This problem is related to rational numbers and their continued fraction representation.

   Rational numbers are represented using the data type `Rational` in Haskell. The ratio p/q is represented using the `%` operator defined in `Data.Ratio`. Acquaint yourself with other functions defined in `Data.Ratio`, like `numerator` and `denominator`, and the function `fromIntegral`. (Note: `numerator rat` can be positive or negative, but `denominator rat` is always positive.)

A finite continued fraction is any expression of the form

$$a_0 + \cfrac{1}{a_1 + \cfrac{1}{a_2 + \cdots \cfrac{1}{a_n}}}$$

where a_0 is an integer and each a_i is a positive integer, for $i \geqslant 1$. This is succinctly represented as the list $[a_0; a_1, a_2, \ldots, a_n]$. (Note the semicolon after the first entry.) A finite continued fraction can be calculated to a rational of the form $\frac{p}{q}$. On the other hand, every rational number can be expressed as a finite continued fraction. For example, the rational number $\frac{42}{31}$ can be rendered as a continued fraction using the following steps:

$$\begin{aligned}
\frac{42}{31} &= 1 + \frac{11}{31} \\
&= 1 + \cfrac{1}{\cfrac{31}{11}} \\
&= 1 + \cfrac{1}{2 + \cfrac{9}{11}} \\
&= 1 + \cfrac{1}{2 + \cfrac{1}{\cfrac{11}{9}}} \\
&= 1 + \cfrac{1}{2 + \cfrac{1}{1 + \cfrac{2}{9}}} \\
&= 1 + \cfrac{1}{2 + \cfrac{1}{1 + \cfrac{1}{\cfrac{9}{2}}}} \\
&= 1 + \cfrac{1}{2 + \cfrac{1}{1 + \cfrac{1}{4 + \cfrac{1}{2}}}}
\end{aligned}$$

Thus one continued fraction corresponding to $\frac{42}{31}$ is $[1; 2, 1, 4, 2]$. A continued fraction corresponding to $-\frac{26}{21}$ is $[-2; 1, 3, 5]$. The continued fraction representation is not unique. For instance, both $[0; 1, 1, 1, 1]$ and $[0; 1, 1, 2]$ represent $\frac{3}{5}$.

Define a function `computeRat :: [Integer] → Rational` that takes a *nonempty* list of integers, such that all but the first element is positive, and returns the rational number corresponding to it.

Define a function `cf :: Rational → [Integer]` that takes a rational number as input and returns a continued fraction corresponding to it.

Sample cases: (Since there are multiple answers possible for `cf`, the cases below are only indicative. We will check the correctness of your solution by actually computing the inverse and checking.)

```
cf (26%21)         = [1,4,5]
cf (-26%21)        = [-2,1,3,5]
cf (42%31)         = [1,2,1,4,2]
cf (-42%31)        = [-2,1,1,1,4,2]
computeRat [1,4,5]         = 26%21
computeRat [-1,1,1,1,1]    = (-2)%5
```

5. Just like finite continued fractions represent rationals, infinite continued fractions of the form

$$a_0 + \cfrac{1}{a_1 + \cfrac{1}{a_2 + \cdots}}$$

correspond to irrational numbers. For example, the **golden ratio** $\phi = \frac{1+\sqrt{5}}{2}$ can be written as[1]

$$1 + \cfrac{1}{1 + \cfrac{1}{1 + \cfrac{1}{1 + \ldots}}}$$

This is represented more succinctly as $[1; 1, 1, 1, 1, \ldots]$. If we truncate this list at some finite point, we get a finite rational approximation for ϕ.

Assume the following Haskell definitions:

```
phi :: Double
phi = (1 + sqrt 5) / 2

computeFrac :: Rational → Double
computeFrac x = fromIntegral (numerator x)
```

[1] This can be verified by denoting the continued fraction as x and observing that $x = 1 + \frac{1}{x}$, i.e. $x^2 = x + 1$, and solving for x (and taking the positive solution).

```
                    / fromIntegral (denominator x)
```

Write a function `approxGR :: Double → Rational` which returns a close rational approximation for the golden ratio, i.e. on input `epsilon` it returns some `r :: Rational` such that `abs (phi - computeFrac r) < epsilon`.

Note: Do not use the `approxRatio` function from `Data.Ratio`.

Sample cases: (Since there are multiple answers possible, the cases below are only indicative. We will check the correctness of your solution by actually calculating if the error is less than `epsilon`.)

```
approxGR 0.0001          = 144 % 89
approxGR 0.0000000000001 = 3524578 % 2178309
```