Assignment 5 (Due: Fri Oct 15 at 5pm) Computational Linguistics (LING 455)

(1) Assume Σ is the set of alphabetic characters (both upper- and lowercase). `Regexp.hs` defines `alpha` as a `Regexp` that matches any of these characters (type `alpha` into `ghci` to see for yourself). It also provides `lower` for the lowercase letters and `upper` for the uppercase ones. Using these definitions, together with the Haskell primitives for building regular expressions (string syntax for `Regexp's`, `<.>`, `<|>`, `star`, `zero`, `one`), define Haskell `Regexp`'s for the following string patterns:

a. Any string that begins with uppercase `S`.

```haskell
startsWithS :: Regexp
startsWithS = undefined
```

b. Any string with an even number of letters (0, 2, 4, . . .).

```haskell
evenLetters :: Regexp
evenLetters = undefined
```

c. Any string that is capitalized — that is, which begins with an uppercase letter followed by any number of lowercase letters.

```haskell
capitalized :: Regexp
capitalized = undefined
```

d. Any string that begins or ends with a lowercase `z`.

```haskell
startsOrEndsz :: Regexp
startsOrEndsz = undefined
```

e. Any string with at least one `i` or `I`.

```haskell
atLeastOnei :: Regexp
atLeastOnei = undefined
```

(2) `Regexp.hs` defines a `match` function with type `Regexp -> String -> Bool` taking a `Regexp` and a `String` and returning `True` or `False` depending on whether the `String` matches the `Regexp`. For example, `match (("a" <|> "b") <.> "c") "ac"` evaluates to `True`. Use `match` to answer the following:

a. Below I define 5 test suites for (1a–e). Define a `helper` function which takes a `Regexp` and a list of `String`'s and returns `True` if all of these strings `match` the `Regexp` and `False` otherwise. Hint: use `map` and `and`; `and` takes a list of `Bool`'s and requires them all to be `True`. For example, `and [True, True]` is `True`, but `and [True, False]` is `False`.

```haskell
suiteA = ["S","Super","Sensational","Supercalifragilisticexpialidocious","Si"]
suiteB = ["", "ab", "xyyz", "HAMMER", "fRaNcAiS"]
suiteC = ["The", "Brooklyn", "Nebraska", "Volkswagen", "A"]
suiteD = ["zoo", "dayz", "zzz", "frazz", "zazzle"]
suiteE = ["stink", "I", "Irish", "Simon", "xxxxxxIxxx"]

helper :: Regexp -> [String] -> Bool
helper r ss = undefined
```

Use `helper` to check your answers to part (1)!

b. Regexps can be used as *tests* that help to process a data set. Write a function `filterMatch` that takes a `Regexp` and a list of potential `String` matches, and keeps only the `String`'s that `match` the `Regexp`.

```
filterMatch :: Regexp -> [String] -> [String]
filterMatch = undefined
```

Test your answer by calling `filterMatch capitalized ["The", "the", "A", "a"]`.

(3) Let's make a simple chatbot using `Regexp`'s! We'll define a `reply` function that takes some input from a user and returns some output that depends on the input. Let's have it behave in the following way:

- If the input is a "I feel ..." statement, the chatbot responds with "Why do you feel that way"?
- If the input is a "Can you ...?", the chatbot says "Sure, I can do that all day!"
- If the input is "Did you ...?", the chatbot responds "No, I didn't! Did you?"
- If the input is some variant of `quit`, the chatbot says "Ok bye!"
- If the input is anything else, the chatbot replies "That's so interesting. What else is on your mind?"

The easiest way to do this is using `match` with **guards**. Here's a refresher on how guards work:

```
guess5 :: Int -> String
guess5 n
  | n < 5 = "too low"          -- the vertical pipes are required
  | n > 5 = "too high"         -- for each case
  | otherwise = "you got it!" -- including the last
```

This function takes a number `n` and tests whether `n < 5` (in which case it reports `"too low"`), `n > 5` (in which case it reports `"too high"`), and otherwise reports `"you got it!"`. Pretty straightforward!

Your job: use `match` and `Regexp`'s to add conditions to `reply` below that make it behave in the way described above given some input string `s`. Remember that string syntax is super handy for specifying `Regexp`'s: one can simply write `"abc" <.> r` instead of `"a" <.> "b" <.> ...`

```
reply :: String -> String
reply s
  -- add your conditions here
  -- and here
  -- and here
  -- and here
  | otherwise = "That's so interesting. What else is on your mind?"
```

Your work is done! (If you're feeling ambitious, you can try to anticipate unexpected variation in the user inputs — for example, variation in capitalization, spelling, punctuation, etc. But that's really not required.) The rest of the file just adds some `IO` code for interacting with the chatbot. Call `main` to talk to it.

```
main = do
  putStrLn "> Hi! I'm the chatbot! Talk to me..."
  sub
sub = do
  s <- getLine
  putStrLn $ "> " ++ reply s
  if (reply s == "Ok bye!") then return () else do sub
```