CS 381, Spring 2021 — Homework 2 (Abstract Syntax)

Please follow carefully ***all*** of the following steps:

1. Prepare a Haskell (or literate Haskell) file (ending in `.hs` or `.lhs`, respectively) that compiles without errors in GHCi. (Put all non-working parts in comments.)
2. Submit ***only one*** solution per group. Each group can have up to 5 members.
3. If you want to submit a solution as a group, do this by creating and using a Canvas group.

Late submissions will ***not*** be accepted. Do ***not*** send solutions by email.

Exercise 1. Mini Logo

Mini Logo is an extremely simplified version of the Logo language for programming 2D graphics. The idea behind Logo and Mini Logo is to describe simple line graphics through commands to move a pen from one position to another. The pen can either be "up" or "down". Positions are given by pairs of integers. Macros can be defined to reuse groups of commands. The syntax of Mini Logo is as follows (nonterminals are typeset in intalics, and terminals are typeset in typewriter font).

cmd	::=	`pen` *mode*
	|	`moveto (`*pos*`,`*pos*`)`
	|	`def` *name* `(` *pars* `)` *cmd*
	|	`call` *name* `(` *vals* `)`
	|	*cmd*`;` *cmd*
mode	::=	`up` | `down`
pos	::=	*num* | *name*
pars	::=	*name*`,` *pars* | *name*
vals	::=	*num*`,` *vals* | *num*

Note: Please remember that unspecified nonterminals, such as *num* and *name*, should be represented by corresponding predefined Haskell types, such as `Int` and `String`.

(a) Define the abstract syntax for Mini Logo as a Haskell data type.

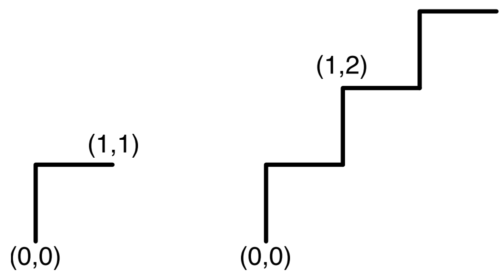
(b) Write a Mini Logo macro `vector` that draws a line from a given position `(x1,y1)` to a given position `(x2,y2)` and represent the macro in abstract syntax, that is, as a Haskell data type value.

> **Note.** What you should actually do is write a Mini Logo program that defines a vector macro. Using *concrete syntax*, the answer would have the following form.
>
> ```
> def vector (...) ...
> ```
>
> It might be a good idea to write the solution in concrete syntax first. But then you should write the same Mini Logo program in *abstract syntax*, that is, you should give a Haskell data type value that starts as follows (assuming `Def` is the constructor name representing the `def` production of the Haskell data type).

```
vector = Def "vector" ... ...
```

You only need to submit this Haskell definition of the value `vector` as part of your Haskell program. (If you like, you can include the concrete syntax as a comment, but it is not required.)

(c) Define a Haskell function `steps :: Int -> Cmd` that constructs a Mini Logo program which draws a stair of n steps. Your solution should ***not*** use the macro `vector`.



Results of the Mini Logo programs produced by `steps 1` and `steps 3`.

Note for parts (b) and (c): The Haskell program you submit doesn't have to draw anything. It only needs to contain the abstract syntax of the macro `vector` (part (b)) and the Haskell function `steps` that produces Mini Logo abstract syntax (part (c)). Only if executed by a Mini Logo interpreter, `vector` called with arguments should result in a line being drawn. And the program that results from the application of `steps` to a number would draw steps only if interpreted by a Mini Logo interpreter.

Exercise 2. Regular Expressions

A regular expression defines a language, that is, a set of words (or strings) over some basic alphabet Σ. For the purpose of this exercise Σ contains all characters of the Haskell built-in type `Char`. The following grammar defines the syntax of regular expressions.

$$\mathit{regex} \quad ::= \quad \epsilon \mid \texttt{.} \mid c \mid \mathit{regex}\texttt{?} \mid \mathit{regex}\texttt{*} \mid \mathit{regex}\texttt{+} \mid \mathit{regex}\ \mathit{regex} \mid \mathit{regex}\texttt{|}\mathit{regex} \mid \texttt{(}\mathit{regex}\texttt{)}$$

Note: The nonterminal c ranges over characters of the alphabet Σ, and the bar symbol `|` in the second-to-last production for alternatives is part of the concrete syntax and must not be confused with the bar symbol | that is part of the grammar meta notation.

By inductively defining what words are matched by a regular expression *regex* we can specify the language that is defined by *regex*.

- ϵ matches the empty word, which is also often written as ϵ.
- `.` matches any character in Σ.
- Each character $c \in \Sigma$ matches itself.
- *regex*`?` matches the empty word or whatever is matched by *regex*.
- *regex*`*` matches zero or more occurrences of the words matched by *regex*.
- *regex*`+` matches one or more occurrences of the words matched by *regex*.
- $\mathit{regex}\ \mathit{regex}'$ matches any word ww' when *regex* matches w and regex' matches w'.
- $\mathit{regex}\texttt{|}\mathit{regex}'$ matches any word that is matched by either *regex* or regex'.

Note that the parentheses are needed for grouping subexpressions. For example, while the regular expression `(a|b)c` defines the language {`ac`, `bc`}, the regular expression `a|(bc)` defines the language {`a`, `bc`}. Here are a few more examples.

- `(b|c)ar(s?)` defines the language {`bar`, `car`, `bars`, `cars`}.

- `.|(ab)` defines the language {`a`, `ab`, `b`, `c`, `d`, ...}.
- `(ab)∗` defines the language {ϵ, `ab`, `abab`, `ababab`, ...}.
- `ab+c?` defines the language {`ab`, `abc`, `abb`, `abbc`, `abbb`, `abbbc`, ...}.
- `a(b+c)?` defines the language {`a`, `abc`, `abbc`, `abbbc`, `abbbbc`, ...}.
- `a(b?c)+` defines the language {`ac`, `abc`, `acc`, `abcc`, `acbc`, `abcbc`, `accc`, `abccc`, `acbcc`, `accbc`, `abcbcc`, ...}.

(a) Define the abstract syntax for regular expressions as a Haskell data type `RegEx`.

```
data RegEx = ...
```

(b) Define a function `accept` that takes a regular expression *regex* and a string *w* and that returns `True` if *regex* matches *w* and `False` otherwise.

```
accept :: RegEx -> String -> Bool
```

The definition for the sequential composition case (*regex regex*) can be a bit tricky. Assuming you represent this case by the constructor `Seq` in your abstract syntax definition, you can employ the following definition, which uses the shown auxiliary function `splits`.

```
accept ...
accept (Seq e1 e2) s = or [accept e1 v && accept e2 w | (v,w) <- splits s]
accept ...

splits :: [a] -> [([a],[a])]
splits []     = []
splits [x]    = [([],[x]),([x],[])]
splits (x:xs) = [([],x:xs)] ++ [(x:s,t) | (s,t) <- splits xs]
```

Hint: Your definition can exploit the following equivalences ($regex \equiv regex'$ means that *regex* defines the same language as $regex'$); your implementation does ***not*** need to handle regular expressions of the form (*regex*∗)∗.

$$regex? \equiv \epsilon \,|\, regex$$
$$regex* \equiv regex+ \,|\, \epsilon$$
$$regex+ \equiv regex\; regex*$$

You can use the following function to test your implementation.

```
classify :: RegEx -> [String] -> IO ()
classify e ws = putStrLn ("ACCEPT:\n"++show acc++"\nREJECT:\n"++show rej)
   where acc = filter (accept e) ws
         rej = filter (not.(accept e)) ws
```

Consider, for example, given the following list of test cases.

```
commaSepTest = ["cat","cat,bat","cat,cat","bat","",",","dog",
                ",cat","cat,","catcat","cat,,bat","cat,bat,"]
```

Applying `classify` to your solution for part (c) and the list of test cases `commaSepTest`, you want the following result.

```
ghci> classify commaSep commaSepTest
ACCEPT:
["cat","cat,bat","cat,cat","bat"]
REJECT:
["",",","dog",",cat","cat,","catcat","cat,,bat","cat,bat,"]
```

(c) Define a value `commaSep :: RegEx` that represent a regular expression for comma-separated lists of the words `cat` and `bat` (examples shown in paet (b)).