Let $\Sigma = \{a, b, c\}$ for questions 1–4.

1. Explicitly compute the languages denoted by the following regular expressions:
   \begin{enumerate}
   \item $ab + c^*\emptyset + \epsilon$
   \item $a(b + c)b + (\emptyset + c)\epsilon$
   \end{enumerate}

2. Give regular expressions denoting the following languages:
   \begin{enumerate}
   \item $\{\epsilon, a, b, ac, bc\}$
   \item $\{a b^n c \mid n \in \mathbb{N}, n > 2\}$
   \end{enumerate}

3. Give regular expressions defining the following languages:
   \begin{enumerate}
   \item All words.
   \item All words that do not contain any $a$s.
   \item All words that contain the sequence $bbc$.
   \item All words that contain at least two $a$s.
   \item All words such that all $a$s appear before all $c$s.
   \item All words such that the total number of $b$s is even.
   \item All words that do not contain the sequence $cc$.
   \item All words that do not contain the sequence $ccc$.
   \end{enumerate}

4. For each of the following regular expressions, construct an equivalent NFA following the graphical construction given in the lectures (and lecture notes). You may eliminate unreachable and “dead-end” (those from which no accepting state can be reached) states, but you should not perform any other reductions.
   \begin{enumerate}
   \item $a + (bc)^*$
   \item $\emptyset a + (b + c)^*a + \epsilon$
   \end{enumerate}

5. **Bonus Exercise**

   Consider the following data type encoding regular expressions:

   ```haskell
   data RE \sigma = Empty
             | Epsilon
             | Symbol \sigma
             | Plus (RE \sigma) (RE \sigma)
             | Sequence (RE \sigma) (RE \sigma)
             | Star (RE \sigma)
             | Paren (RE \sigma)
   deriving (Eq, Show)
   
   ```

   The type parameter \(\sigma\) is the underlying alphabet.
For example, some regular expressions over the alphabets of characters and integers are as follows:

- \( \epsilon + abc \)
- \( \text{re1} ::= \text{RE Char} \)
- \( \text{re1} = \text{Epsilon} \cdot \text{Plus} \cdot (\text{Symbol } 'a' \cdot \text{Sequence} \cdot \text{Symbol } 'b' \cdot \text{Sequence} \cdot \text{Symbol } 'c') \)
- \( \text{re2} ::= \text{RE Char} \)
- \( \text{re2} = \text{Star} \cdot (\text{Paren} \cdot \text{Symbol } '0' \cdot \text{Plus} \cdot \text{Symbol } '1') \)
- \( \text{re3} ::= \text{RE Int} \)
- \( \text{re3} = \text{Star} \cdot (\text{Symbol } 1) \)

Consider also the following encoding of words and languages:

**type** \( \text{Word } \sigma = [\sigma] \)

**type** \( \text{Language } \sigma = [\text{Word } \sigma] \)

(a) Define the empty word for any alphabet:

\( \epsilon :: \text{Word } \sigma \)

(b) Define a function that concatenates two languages.

\( \text{langConcat} :: \text{Language } \sigma \rightarrow \text{Language } \sigma \rightarrow \text{Language } \sigma \)

Note that this is substantially more challenging for infinite languages than for finite languages. I suggest that you first define \( \text{langConcat} \) for finite languages, and then only attempt to extend it to infinite languages if you are feeling particularly adventurous.

(c) Define a function that raises a language to an integer power (you can ignore negative integers).

\( \text{langExp} :: \text{Language } \sigma \rightarrow \text{Int} \rightarrow \text{Language } \sigma \)

(d) Define a function that applies the Kleene Star operation to a language.

\( \text{kleeneStar} :: \text{Eq } \sigma \Rightarrow \text{Language } \sigma \rightarrow \text{Language } \sigma \)

Note that while this function will not be terminating, it should be **productive**. That is, it should enumerate all words in the (infinite) resultant language, rather than hanging. Thus, for example, \( \text{take } n \cdot (\text{kleeneStar } l) \) should terminate for any language \( l \) and positive integer \( n \).

(e) Define a function that enumerates the language of a regular expression.

\( \text{re2lang} :: \text{Eq } \sigma \Rightarrow \text{RE } \sigma \rightarrow \text{Language } \sigma \)

**Hint:** You may find the following functions helpful:

\( \text{import Data.List (union)} \)

\( \text{unions} :: \text{Eq } a \Rightarrow [[a]] \rightarrow [a] \)

\( \text{unions} = \text{foldr union []} \)

Note that \( \text{unions} \) has been defined using \text{foldr} rather than \text{foldl}. If you have a working solution, try using \text{foldl} instead and see if it makes a difference.