Explicitly write the type of each function.

1. (30 points) Modify the monadic parser (Parser4.hs) so that it implements the grammar below and parses lists of equations. (You may use the same scanner even though it’s for a slightly different language.) The ‘output’ of the parser should be a list of Eqns suitable as input for the function system from assignment three, question five. (More specifically, the output should be either a list wrapped in the Seq constructor or a ParseEqnSeqError.)

**Grammar:**

\[
\begin{align*}
S & ::= Q, S \\
   & ::= Q \\
Q & ::= E = E \\
E & ::= T E' \\
E' & ::= + T E' \\
   & ::= - T E' \\
   & ::= \varepsilon \\
T & ::= F T' \\
T' & ::= \cdot F T' \\
   & ::= / F T' \\
   & ::= \varepsilon \\
F & ::= id \\
   & ::= num \\
   & ::= - F \\
   & ::= ( E )
\end{align*}
\]

**Example:**

```haskell
> parse (tokenStreamFromString "w=1/2*w+20")
Seq [Eqn (Var "w") (Sum (Prod (Quo (RExp 1) (RExp 2)) (Var "w"))) (RExp 20)]
> parse (tokenStreamFromString "2*y+x=10, 2*x+1=9")
Seq [Eqn (Sum (Prod (RExp 2) (Var "y"))) (Var "x")) (RExp 10),
    Eqn (Sum (Prod (RExp 2) (Var "x"))) (RExp 1)) (RExp 9)]
> parse (tokenStreamFromString "w=/2*w+20")
ParseEqnSeqError "Bad input"
```
2. (60 points) Consider the following definitions of regular expressions and NFAs. Then follow the instructions to implement each.

**Definition 1.** Given an alphabet $\Sigma$, a regular expression $R$ (over $\Sigma$) is one of the following.

- $\emptyset$, or,
- $\varepsilon$, or,
- $\sigma$ if $\sigma \in \Sigma$, or,
- $(R_1 \cup R_2)$ if $R_1$ and $R_2$ are regular expressions, or,
- $(R_1R_2)$ if $R_1$ and $R_2$ are regular expressions, or,
- $(R^*)$ if $R$ is a regular expression.

**Definition 2.** Given an alphabet $\Sigma$, the set $\Sigma_\varepsilon = \Sigma \cup \{\varepsilon\}$.

**Definition 3.** A non-deterministic finite automaton (NFA) is a five-tuple $(Q, \Sigma, \delta, q_0, F)$, where

- $Q$ is a finite set of states.
- $\Sigma$ is a finite set of input symbols.
- $\delta : Q \times \Sigma_\varepsilon \rightarrow \mathcal{P}(Q)$ is the transition function.
- $q_0 \in Q$ is the start state.
- $F \subseteq Q$ is the set of accepting states.

**Definition 4.** Given an NFA $N = (Q, \Sigma, \delta, q_0, F)$, the $\varepsilon$-closure of a set of states $S$, $E(S)$, is defined as follows.

- $q \in E(S)$ if $q \in S$, and
- $q' \in E(S)$ if $q \in E(S)$ and $q' \in \delta(q, \varepsilon)$.

**Definition 5.** Given an NFA $N = (Q, \Sigma, \delta, q_0, F)$ and a string $x \in \Sigma^*$, $x$ is accepted by $N$ if $\delta^*(q_0, x) \cap F \neq \emptyset$.

We’ll represent some of the sets above using Haskell lists. Import the module List to get the operations union and intersect. Represent $\Sigma_\varepsilon$ using Haskell’s Maybe type.

(a) The set $\Sigma$ will be represented as a Haskell type. Implement regular expressions as the type RegExp that takes a type parameter sigma. It should have the following constructors: RegEmpty, RegEpsilon, RegSym, RegOr, RegSeq, and RegStar.

(b) The sets $Q$ and $\Sigma$ will be represented as a Haskell types. Implement NFAs as the type NFA that takes two type parameters: q and sigma. The constructor for the NFA should be called NFA. There should be three parts: the first part is the transition function, which should be represented as a Haskell function, where a list of q represents the power set of Q; the second part is the start state; and the third part is the set of final or accepting states, which should also be represented as a list.
(c) Write a function `epsilonClosure` that takes an NFA and a list of states. It should return a list of states. In principle, it should implement $E$ in definition four; however, here the implementation is not obvious. To compute the set, start with the given set and union in all sets of states that we can arrive at using an epsilon transition from the given set. If the set is unchanged, that’s the answer; otherwise, repeat that calculation.

(d) Write a function `deltaStar` that takes an NFA and returns a function that takes a pair, where the first element of the pair is a state and the second element of the pair is a string representation; the function returns a list of states. Since the definition assumes a symbol comes off the end of a string rather than the beginning, we will represent strings using the type `Str` modified to take a type parameter (see Haskell Essentials Program Three). The implementation of this function should follow from definition five.

(e) Write a function `doesAccept` that takes an NFA $N$ and a Haskell list $s$, and returns a Boolean indicating whether or not $N$ accepts $s$. The list $s$ should be converted to type `Str`. The implementation should follow from definition six.

(f) Write a function `nfaFromRegExp` that takes a regular expression and returns an equivalent NFA. The algorithm for this conversion will be available as a separate sheet. **NOTE:** Combining $\delta$ functions is no problem since they return lists. **HINT:** It is typical to use integers as states. To generate new states you will need a counter; you will want a helper function that keeps track of the counter. Further, you may want to hide the counter using a monad.

Examples:

```haskell
*Assign5> doesAccept (nfaFromRegExp RegEmpty) ""
False
*Assign5> doesAccept (nfaFromRegExp RegEmpty) "a"
False
*Assign5> doesAccept (nfaFromRegExp RegEpsilon) ""
True
*Assign5> doesAccept (nfaFromRegExp RegEpsilon) "a"
False
*Assign5> doesAccept (nfaFromRegExp RegEpsilon) "b"
False
*Assign5> doesAccept (nfaFromRegExp (RegSym 'a')) ""
False
*Assign5> doesAccept (nfaFromRegExp (RegSym 'a')) "a"
True
*Assign5> doesAccept (nfaFromRegExp (RegSym 'a')) "b"
False
*Assign5> doesAccept (nfaFromRegExp (RegSym 'a')) "aa"
False
*Assign5> doesAccept (nfaFromRegExp (RegOr (RegSym 'a') (RegSym 'b'))) ""
False
*Assign5> doesAccept (nfaFromRegExp (RegOr (RegSym 'a') (RegSym 'b'))) "a"
True
*Assign5> doesAccept (nfaFromRegExp (RegOr (RegSym 'a') (RegSym 'b'))) "b"
True
*Assign5> doesAccept (nfaFromRegExp (RegOr (RegSym 'a') (RegSym 'b'))) "ab"
False
*Assign5> doesAccept (nfaFromRegExp (RegSeq (RegSym 'a') (RegSym 'b'))) ""
```
False
*Assign5> doesAccept (nfaFromRegExp (RegSeq (RegSym 'a') (RegSym 'b'))) "a" False
*Assign5> doesAccept (nfaFromRegExp (RegSeq (RegSym 'a') (RegSym 'b'))) "b" False
*Assign5> doesAccept (nfaFromRegExp (RegSeq (RegSym 'a') (RegSym 'b'))) "ba" False
*Assign5> doesAccept (nfaFromRegExp (RegSeq (RegSym 'a') (RegSym 'b'))) "ab" True
*Assign5> doesAccept (nfaFromRegExp (RegSeq (RegSym 'a') (RegSym 'b'))) "aba" False
*Assign5> doesAccept (nfaFromRegExp (RegStar (RegSym 'a'))) "" True
*Assign5> doesAccept (nfaFromRegExp (RegStar (RegSym 'a'))) "a" True
*Assign5> doesAccept (nfaFromRegExp (RegStar (RegSym 'a'))) "aa" True
*Assign5> doesAccept (nfaFromRegExp (RegStar (RegSym 'a'))) "b" False
*Assign5> doesAccept (nfaFromRegExp (RegStar (RegSym 'a'))) "ab" False

**Graduate Problems/Undergraduate Extra Credit**

1. (10 points) Build an interactive program (i.e., a program that performs input/output) that reads in a string representing a linear system of equations, and displays the solution to the system. You should make use of the code you've written for assignment three, and question one above.