Functional Programming Skills
Assignment 5

Arthur Nunes-Harwitt

Explicitly write the type of each function.

1. (25 points) An alternative approach to matching regular expression patterns involves computing the "derivative" of a regular expression. Consider the following definitions of regular expressions and functions on regular expressions. Then follow the instructions to implement each.

   We start with the following extended definition of regular expressions.

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

   - $\emptyset$*, or,*
   - ε*, or,*
   - σ *if* $\sigma \in \Sigma$*, or,*
   - $(R_1 \mid R_2)$ *if R_1 and R_2 are regular expressions, or,*
   - $(R_1 R_2)$ *if R_1 and R_2 are regular expressions, or,*
   - (R^*) *if R is a regular expression, or,*

   We define the notion of nullable next.

   Definition 2. *Given a language $\mathcal{L}$ over Σ, we say that $\mathcal{L}$ is* nullable *if $\varepsilon \in \mathcal{L}$.*

   We can understand the property of being nullable on a regular expression R as the answer to the question of being nullable on the regular language that R denotes. We can define this property directly on regular expressions via structural recursion.

   $$\begin{array}{lcl} \mathit{nullable}(\emptyset) & = & \bot \\ \mathit{nullable}(\varepsilon) & = & \top \\ \mathit{nullable}(\sigma) & = & \bot \\ \mathit{nullable}(R_1 \mid R_2) & = & \mathit{nullable}(R_1) \vee \mathit{nullable}(R_2) \\ \mathit{nullable}(R_1 R_2) & = & \mathit{nullable}(R_1) \wedge \mathit{nullable}(R_2) \\ \mathit{nullable}(R^*) & = & \top \end{array}$$

   We can define a variation on the function *nullable* that returns a regular expression instead of a Boolean value.

   Definition 3. *Give a regular expression R over Σ,* $\nu(R) = \begin{cases} \varepsilon & \text{if } \mathit{nullable}(R) \\ \emptyset & \text{otherwise} \end{cases}$.

   Now we define the derivative of a language; it is the language of suffixes.

Definition 4. *Given a language $\mathcal{L}$ over Σ, the* derivative *of $\mathcal{L}$ with respect to a symbol $\sigma \in \Sigma$ is denoted by $\partial_\sigma \mathcal{L}$, where $\partial_\sigma \mathcal{L} = \{x \in \Sigma^* \mid \sigma x \in \mathcal{L}\}$.*

We can understand the derivative of a regular expression R as the derivative of the regular language that R denotes. It happens that the derivative language is also regular, and so we can represent it as a regular expression. We can define the derivative directly on regular expressions via structural recursion.

$$
\begin{array}{lcl}
\partial_\sigma \boldsymbol{\emptyset} & = & \boldsymbol{\emptyset} \\
\partial_\sigma \boldsymbol{\varepsilon} & = & \boldsymbol{\emptyset} \\
\partial_\sigma \hat{\sigma} & = & \begin{cases} \boldsymbol{\varepsilon} & \text{if } \sigma = \hat{\sigma} \\ \boldsymbol{\emptyset} & \text{otherwise} \end{cases} \\
\partial_\sigma (R_1 \mid R_2) & = & (\partial_\sigma R_1) \mid (\partial_\sigma R_2) \\
\partial_\sigma (R_1 R_2) & = & (\partial_\sigma R_1) R_2 \mid \nu(R_1)(\partial_\sigma R_2) \\
\partial_\sigma (R^*) & = & (\partial_\sigma R) R^*
\end{array}
$$

Definition 5. *Given an alphabet Σ, a string $x \in \Sigma^*$, and a regular expression R over Σ, we say that x matches R, or $x \sim R$, if x is in the language that R denotes.*

Using the derivative we can define the match operator.

$$
\begin{array}{lcl}
\varepsilon \sim R & = & \mathit{nullable}(R) \\
\sigma x \sim R & = & x \sim \partial_\sigma R
\end{array}
$$

(a) The set Σ 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) Write a function `nullable` that takes a regular expression and returns a Boolean based on the definition above.

(c) Write a function `nu` that takes a regular expression and returns a regular expression based on the definition of $\nu(R)$ above.

(d) Write a function `deriv` that takes an element of the alphabet and a regular expression, and returns a regular expression based on the definition of the derivative of a regular expression. (Use the `Eq` type class.)

(e) Write a function `match` that takes a list of elements from the alphabet and a regular expression, and returns a Boolean, based on the definition. (Use the `Eq` type class.)

Examples:

```
*Assign5> match "" RegEmpty
False
*Assign5> match "a" RegEmpty
False
*Assign5> match "" RegEpsilon
True
*Assign5> match "a" RegEpsilon
```

```
False
*Assign5> match "" (RegSym 'a')
False
*Assign5> match "a" (RegSym 'a')
True
*Assign5> match "b" (RegSym 'a')
False
*Assign5> match "aa" (RegSym 'a')
False
*Assign5> match "" (RegOr (RegSym 'a') (RegSym 'b'))
False
*Assign5> match "a" (RegOr (RegSym 'a') (RegSym 'b'))
True
*Assign5> match "b" (RegOr (RegSym 'a') (RegSym 'b'))
True
*Assign5> match "ab" (RegOr (RegSym 'a') (RegSym 'b'))
False
*Assign5> match "" (RegSeq (RegSym 'a') (RegSym 'b'))
False
*Assign5> match "a" (RegSeq (RegSym 'a') (RegSym 'b'))
False
*Assign5> match "b" (RegSeq (RegSym 'a') (RegSym 'b'))
False
*Assign5> match "ab" (RegSeq (RegSym 'a') (RegSym 'b'))
True
*Assign5> match "ba" (RegSeq (RegSym 'a') (RegSym 'b'))
False
*Assign5> match "aba" (RegSeq (RegSym 'a') (RegSym 'b'))
False
*Assign5> match "" (RegStar (RegSym 'a'))
True
*Assign5> match "a" (RegStar (RegSym 'a'))
True
*Assign5> match "aa" (RegStar (RegSym 'a'))
True
*Assign5> match "b" (RegStar (RegSym 'a'))
False
*Assign5> match "ab" (RegStar (RegSym 'a'))
False
*Assign5> match "abb" (RegSeq (RegSym 'a') (RegStar (RegSym 'b')))
True
*Assign5> match "aba" (RegSeq (RegSym 'a') (RegStar (RegSym 'b')))
False
```

2. (50 points) In this problem we will use the derivative of regular expressions to create a small lexical analyzer generator. To do this, we will need the notion of a regular vector.

Definition 6. *A* regular vector $\vec{R}$ *is a vector of the form* $\vec{R} = \langle (R_1, a_1), \cdots, (R_n, a_n) \rangle$*, where* R_i *is a regular expression and* a_i *is a function from strings to tokens.*

We then define the derivative on a regular vector: $\partial_\sigma \langle (R_1, a_1), \cdots, (R_n, a_n) \rangle = \langle (\partial_\sigma R_1, a_1), \cdots, (\partial_\sigma R_n, a_n) \rangle$.

Here we fix the alphabet so that it corresponds to the `Char` type.

To determine if the beginning of a string matches a pattern, we go character by character taking the derivative of the vector repeatedly and recording the state in a stack. We stop advancing when all of the regular expressions correspond to the empty language. Then we go through the stack looking (in order) through the regular vector for a nullable regular expression.

To make the lexical analyzer generator, implement the following.

(a) Write a function `empty` that takes a regular expression and returns a Boolean. This function determines whether or not the regular expression denotes the empty language.

(b) Make sure that the `Token` type is defined. You may use the definition from the scanner lecture.

(c) Define a type synonym `RegVec` that is a list of pairs. The first element of the pair is a regular expression on `Char` and the second element of the pair is a function from strings to tokens.

(d) Write a function `derivVec` that takes a character and a `RegVec` and returns a `RegVec`.

(e) Write a function `emptyVec` that takes a `RegVec` and returns a Boolean. True is returned when all the regular expressions in the vector denote the empty language.

(f) Write a function `nullableInVec` that takes a `RegVec` and returns a `Maybe` function from strings to tokens. This function goes through the regular vector (in order) looking for a nullable regular expression. If one is found, it returns just the associated action. If not, it returns `Nothing`.

(g) Write a function `findNullable` that takes a list of triples and returns a pair. The triple contains a string, a `RegVec`, and another string. The first string is the list of characters seen (in reverse order); the other string is the list of characters remaining. The pair contains a token and a string. This function goes through the list of triples, applying the `nullableInVec` function to each regular vector. If there is an action, it is applied to the seen string which is paired with the remaining characters. If the list of triples is empty, the first element of the pair returned is a lexical error.

(h) Write the functions `splitInputHelper` and `splitInput`.

The function `splitInputHelper` takes four arguments: the input string, a regular vector, the list of seen characters, and a list/stack of triples; it returns a token-string pair. If the input string is empty, `findNullable` is invoked on the stack. Otherwise, we compute the vector derivative with respect to the first character in the input string, augment the list of seen characters, and then check whether or not the derivative computed is all empty. If so, `findNullable` is invoked on the stack; if not, processing continues and a new triple is pushed on the stack.

The function `splitInput` takes the input string and a regular vector. It calls `splitInputHelper` initializing the list of seen characters and the stack.

(i) Write a function `makeTokenStreamFromString` that takes a regular vector and the input string. It returns a list of tokens. When the input string is empty, the singleton list containing the eof-token is returned. Otherwise, if the first character is white-space, skip over it, but if not then use `splitInput` to break the input string into the first token and the remaining string — then generate the rest of the output list recursively. (Feel free to import `Data.Char` to get the white-space predicate.)

(j) Write a function `regOrFromList` that takes a list of regular expressions and returns a regular expression that is the disjunction of the given regular expressions.

Examples:

```
*Assign5> regOrFromList [RegSym d | d <- ['0'..'9']]
RegOr (RegSym '0')
      (RegOr (RegSym '1')
             (RegOr (RegSym '2')
                    (RegOr (RegSym '3')
                           (RegOr (RegSym '4')
                                  (RegOr (RegSym '5')
                                         (RegOr (RegSym '6')
                                                (RegOr (RegSym '7')
                                                       (RegOr (RegSym '8')
                                                              (RegSym '9')))))))))
```

For the remaining examples for this problem assume the following definition.

```
tokenStreamFromString =
  makeTokenStreamFromString [((RegSym ','), \s->Simple COMMA),
                             ((RegSym '+'), \s->Simple PLUS),
                             ((RegSym '-'), \s->Simple MINUS),
                             ((RegSym '*'), \s->Simple STAR),
                             ((RegSym '/'), \s->Simple SLASH),
                             ((RegSym '='), \s->Simple EQ1),
                             ((RegSym '('), \s->Simple OP),
                             ((RegSym ')'), \s->Simple CP),
                             ((RegSeq (RegSym 'l')
                                      (RegSeq (RegSym 'e') (RegSym 't'))),
                               \s->Simple LET),
                             ((RegSeq (RegSym 'i') (RegSym 'n')),
                               \s->Simple IN),
                             ((RegSeq letterClass (RegStar letterClass)),
                               \s->Compound (Id s)),
                             ((RegSeq digitClass (RegStar digitClass)),
                               \s->Compound (Num (read s)))]


*Assign5> tokenStreamFromString "2+3"
[Compound (Num 2),Simple PLUS,Compound (Num 3),Simple EOF]
*Assign5> tokenStreamFromString "letter let"
[Compound (Id "letter"),Simple LET,Simple EOF]
```

```
*Assign5> tokenStreamFromString "let xx = 21, yy = 99 in xx * yy"
[Simple LET,Compound (Id "xx"),Simple EQ1,Compound (Num 21),
Simple COMMA,Compound (Id "yy"),Simple EQ1,Compound (Num 99),
Simple IN,Compound (Id "xx"),Simple STAR,Compound (Id "yy"),
Simple EOF]
*Assign5> tokenStreamFromString "2~3"
[Compound (Num 2),LexError "Unidentified text.",Simple EOF]
```

3. (10 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 from the lecture even though it's for a slightly different language.)

   - First define a type `EqnSeq`. There should be two constructors: `Seq` and `ParseEqnSeqError`. The `Seq` constructor should take a list of `Eqn`s as defined in assignment three, question five; the `ParseEqnSeqError` constructor should take a string. It should derive `Eq` and `Show`.
   - The modification of the parser entails that the function `parse` returns an `EqnSeq` object. Note that the list in a `Seq` should be almost suitable as input for the function `system` from assignment three: The `Exp` data structure here is similar but not identical to the one in assignment three. As in assignment three, we want to generalize constants to rational numbers. However, the expressions in assignment three were more limited in that there were only sums and products; no differences, quotients, or negations. Those extensions must be preserved; of course, the data structure should *not* include cases for powers or for a let-expression.

   Grammar:

S	::=	Q , S
	::=	Q
Q	::=	$E = E$
E	::=	$T\ E'$
E'	::=	$+\ T\ E'$
	::=	$-\ T\ E'$
	::=	ε
T	::=	$F\ T'$
T'	::=	$*\ F\ T'$
	::=	$/\ F\ T'$
	::=	ε
F	::=	id
	::=	num
	::=	$-\ F$
	::=	$(\ E\ )$

Example:

```
*Assign5> parse (tokenStreamFromString "w=1/2*w+20")
Seq [Eqn (Var "w") (Sum (Prod (Quo (RExp 1) (RExp 2)) (Var "w")) (RExp 20))]
*Assign5> 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)]
*Assign5> parse (tokenStreamFromString "w=/2*w+20")
ParseEqnSeqError "Bad input"
```

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 (which will need to be modified slightly due to the change in `Exp`), and question three above (which should be used as is).

   Example:

   ```
   *Assign5> interface
   Enter a system of equations.
   When finished, type 'done'.
   System:  w = 1/2*w + 20
   w = 40
   System:  x + y = 5, y - x = 1
   x = 2, y = 3
   System:  x + y = z + 1, z = 2*y - 2, z + 3*x = 10
   x = 2, y = 3, z = 4
   System:  done
   ```