Definition 1 *Strings*
For a set X, we define strings over X to be elements of the free monoid over X, called X^*.
We use the notation $\cdot : (String \times String) \to String$ to mean the concatenation of strings.

Definition 2 *Alphabet*
Let Σ be an arbitrary, finite set. This document will call Σ the Alphabet. We will let c be a metavariable ranging over Σ, and s be a metavariable ranging over Σ^*, ie, strings of Σ.

Definition 3 *Parsing Expression Grammars*
We define the set Peg via the following bnf formula, overloading the operator $\cdot$

$$
\begin{array}{lcllc}
\texttt{peg} & \in & Peg & ::= & c \in \Sigma \\
 & & & | & \texttt{peg}_1 \cdot \texttt{peg}_2 \\
 & & & | & \texttt{peg}_1 \,/\, \texttt{peg}_2 \\
 & & & | & \texttt{peg}^*
\end{array}
$$

Figure 1: Parsing Expression Grammar Syntax

These rules are called character, sequence, choice, and possesive star respectively.

Definition 4 *Regular Expressions*
We define the set Reg via the following bnf formula, again overloading the operator $\cdot$

$$
\begin{array}{lcllc}
\texttt{reg} & \in & Reg & ::= & \varepsilon \\
 & & & | & c \in \Sigma \\
 & & & | & . \\
 & & & | & \bot \\
 & & & | & \texttt{reg}_1 \cdot \texttt{reg}_2 \\
 & & & | & \texttt{reg}_1 \cup \texttt{reg}_2 \\
 & & & | & \texttt{reg}_1 \cap \texttt{reg}_2 \\
 & & & | & \neg\, \texttt{reg} \\
 & & & | & \texttt{reg}^*
\end{array}
$$

Figure 2: Regular Expression Grammar Syntax

These rules are empty, character, any, empty, sequence, union, intersection, negation, and kleene star respectively.

Definition 5 *Parsing Expression Grammar Matching*
Letting Matched $= \Sigma^*$ *and Remainder* $= \Sigma^*$,
We inherit the partial function pegMatch : $(Peg \times \Sigma^*) \nrightarrow (Matched \times Remainder) \uplus \{\bot\}$
from the paper "Towards a Typed Semantics for Parsing Expression Grammars".

Definition 6 *Character Of*
For a set X *with elements* $x \in X$, *we say that* x *is a character of the string* $xs \in X^*$ *iff*
$\exists(prf, suf \in X^*), xs = prf \cdot x \cdot suf$
We use the notation $x\, char\text{-}of\ xs$ *to mean* x *is a character of the string* xs.

Definition 7 *Regular Expression Matching*
We define $regMatch : (\mathtt{reg} \times \Sigma^*) \rightarrow \mathbb{B}$ *recursively:*
$regMatch(\varepsilon, \varepsilon) = \mathtt{t}$
$regMatch(\varepsilon, c \cdot s) = \mathtt{f}$
$regMatch(c, c) = \mathtt{t}$
$regMatch(c, \varepsilon) = \mathtt{f}$
$regMatch(c, c')$ *where* $c \neq c' = \mathtt{f}$
$regMatch(c, c' \cdot s) = \mathtt{f}$
$regMatch(\cdot, c) = \mathtt{t}$
$regMatch(\cdot, c' \cdot s) = \mathtt{f}$
$regMatch(\bot, s) = \mathtt{f}$
$regMatch(\mathtt{reg}_1 \cdot \mathtt{reg}_2, s) = \exists(s', s'' \in \Sigma^*), s = s' \cdot s'' \wedge regMatch(\mathtt{reg}_1, s') \wedge regMatch(\mathtt{reg}_2, s'')$
$regMatch(\mathtt{reg}_1 \cup \mathtt{reg}_2, s) = regMatch(\mathtt{reg}_1, s) \vee regMatch(\mathtt{reg}_2, s)$
$regMatch(\mathtt{reg}_1 \cap \mathtt{reg}_2, s) = regMatch(\mathtt{reg}_1, s) \wedge regMatch(\mathtt{reg}_2, s)$
$regMatch(\neg\, \mathtt{reg}_1, s) = \neg\, regMatch(\mathtt{reg}_1, s)$
$regMatch(\mathtt{reg}_1^*, s) = \exists(ss \in \Sigma^{**}), \bigwedge_{s\, char\text{-}of\ ss} regMatch(\mathtt{reg}_1, s)$

While most rules are self explanatory, the rule for concatenation and star may not be.

Concatenation splits the string into two halves, the first of which matches to the left regex, and the second to the right.

Star considers all possible splits of the string, and requires that the constituent strings of the split are each matched by the subexpression.

Definition 8 *Parsing Expression Grammar Translation*

On input $(\texttt{peg}, \texttt{reg})$*, the function pegreg produces a regular expression with the characters consumed by* `peg` *removed. It is defined in mutual recursion with the negate function, which on input* `peg`*, generates a regular expression corresponding to the set of characters that cause* `peg` *to match* $\perp$*.*

$$\begin{aligned}
&\mathit{pegreg} : (\mathit{Peg} \times \mathit{Reg}) \to \mathit{Reg} \\
&\mathit{pegreg}(c, \texttt{reg}) = c \cdot \texttt{reg} && (1) \\
&\mathit{pegreg}(\texttt{peg}_1 \cdot \texttt{peg}_2, \texttt{reg}) = \mathit{pegreg}(\texttt{peg}_1, \mathit{pegreg}(\texttt{peg}_2, \texttt{reg})) && (2) \\
&\mathit{pegreg}(\texttt{peg}_1 \,/\, \texttt{peg}_2, \texttt{reg}) = \mathit{pegreg}(\texttt{peg}_1, \texttt{reg}) \cup (\mathit{pegreg}(\texttt{peg}_2, \texttt{reg}) \cap \mathit{negate}(\texttt{peg}_1)) && (3) \\
&\mathit{pegreg}(\texttt{peg}^*, \texttt{reg}) = \mathit{pegreg}(\texttt{peg}, \varepsilon)^* \cdot (\texttt{reg} \cap \mathit{negate}(\texttt{peg})) && (4)
\end{aligned}$$

$$\begin{aligned}
&\mathit{negate} : \mathit{Peg} \to \mathit{Reg} \\
&\mathit{negate}(c) = (\Sigma \setminus \{c\}) \cdot \Sigma^* && (5) \\
&\mathit{negate}(\texttt{peg}_1 \cdot \texttt{peg}_2) = \mathit{negate}(\texttt{peg}_1) \cup \mathit{pegreg}(\texttt{peg}_1, \mathit{negate}(\texttt{peg}_2)) && (6) \\
&\mathit{negate}(\texttt{peg}_1 \,/\, \texttt{peg}_2) = \mathit{negate}(\texttt{peg}_1) \cap \mathit{negate}(\texttt{peg}_2) && (7) \\
&\mathit{negate}(\texttt{peg}^*) = \perp && (8)
\end{aligned}$$

Proposition 1 *Let* $(\texttt{peg}, \texttt{reg})$ *be an arbitrary pair of parsing and regular expressions. Then the following will hold:*

1. *Let* s *be an arbitrary string. Then* $\mathit{regMatch}(\mathit{pegreg}(\texttt{peg}, \texttt{reg}), s)$
   if and only if
   $\exists(\mathit{prf}, \mathit{suf} \in \Sigma^*)$,
   $\mathit{pegMatch}(\texttt{peg}, s) = (\mathit{prf}, \mathit{suf}) \wedge$
   $\mathit{regMatch}(\texttt{reg}, \mathit{suf})$.

2. *Let* s *be an arbitrary string. Then* $\mathit{pegMatch}(\texttt{peg}, s) = \perp \iff \mathit{regMatch}(\mathit{negate}(\texttt{peg}), s)$.

I think this ought to be provable by induction. I'll have to think about it.