Definition 1. *Strings*
For a set X, we define strings over X to be elements of the free monoid over X, called X^*, with $\varepsilon \in X^*$ denoting the 0 element. The set X^* will be ranged over by the metavariable xs.

We use the notation $\cdot : (X^* \times X^*) \to X^*$ to mean the concatenation of strings.

The operator $fold : ((X \times A \to A) \times A \times X^*) \to A$ is defined
$fold(f, a, \varepsilon) = a, fold(f, a, (x \in X) \cdot xs) = f(x, fold(f, a, xs))$.

Strings are equivalent to finite sequences and so will be indexed. For an i in $i \in \mathbb{Z}^+ \cup 0$, xs_i is equal to the ith index and is otherwise undefined.

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*
For a set of labels Lab, we define the set Peg(Lab) via the following bnf formula, overloading the operator $\cdot$.

$$\begin{array}{rcll} \mathrm{p} & \in & \mathit{Peg}(\mathit{Lab}) & ::= \ \texttt{lab:}\ c \\ & & & \ \mid \ \texttt{lab:}\ \mathrm{p}_1 \cdot \mathrm{p}_2 \\ & & & \ \mid \ \texttt{lab:}\ \mathrm{p}_1 \,/\, \mathrm{p}_2 \\ & & & \ \mid \ \texttt{lab:}\ \mathrm{p}^* \end{array}$$

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(Lab) via the following bnf formula, again overloading the operator $\cdot$. Lab is a set of labels, and l will be a metavariable ranging over these.

$$\begin{array}{rcll} \mathrm{r} & \in & \mathit{Reg}(\mathit{Lab}) & ::= \ \texttt{lab:}\ \varepsilon \\ & & & \ \mid \ \texttt{lab:}\ c \\ & & & \ \mid \ \texttt{lab:}\ . \\ & & & \ \mid \ \texttt{lab:}\ \bot \\ & & & \ \mid \ \texttt{lab:}\ \mathrm{r}_1 \cdot \mathrm{r}_2 \\ & & & \ \mid \ \texttt{lab:}\ \mathrm{r}_1 \cup \mathrm{r}_2 \\ & & & \ \mid \ \texttt{lab:}\ \mathrm{r}_1 \cap \mathrm{r}_2 \\ & & & \ \mid \ \texttt{lab:}\ \neg\, \mathrm{r} \\ & & & \ \mid \ \texttt{lab:}\ \mathrm{r}^* \end{array}$$

Figure 2: Regular Expression Grammar Syntax

These rules are empty, character, wildcard, 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. *Regular Expression Matches*
When matched against a string, a regular expression $\mathrm{r} \in \mathit{Reg}(\mathit{Lab})$ *will give a structured result,* $\mathit{Match}(\mathit{Lab})$.

$$
\begin{array}{lcll}
\mathrm{m} & \in & \mathit{Match}(\mathit{Lab}) & ::= \; \texttt{Emp}(\texttt{lab}\colon l, \texttt{string}\colon \varepsilon) \\
 & & & \;\;\;\mid \; \texttt{Char}(\texttt{lab}\colon l, \texttt{string}\colon c) \\
 & & & \;\;\;\mid \; \texttt{Any}(\texttt{lab}\colon l, \texttt{string}\colon c) \\
 & & & \;\;\;\mid \; \texttt{Seq}(\texttt{lab}\colon l, \texttt{string}\colon s, \texttt{right}\colon \mathrm{m}_1, \texttt{left}\colon \mathrm{m}_2) \\
 & & & \;\;\;\mid \; \texttt{JoinL}(\texttt{lab}\colon l, \texttt{string}\colon s, \texttt{subexpr}\colon \mathrm{m}) \\
 & & & \;\;\;\mid \; \texttt{JoinR}(\texttt{lab}\colon l, \texttt{string}\colon s, \texttt{subexpr}\colon \mathrm{m}) \\
 & & & \;\;\;\mid \; \texttt{Meet}(\texttt{lab}\colon l, \texttt{string}\colon s, \texttt{left}\colon \mathrm{m}_1, \texttt{right}\colon \mathrm{m}_1) \\
 & & & \;\;\;\mid \; \texttt{Neg}(\texttt{lab}\colon l, \texttt{string}\colon s) \\
 & & & \;\;\;\mid \; \texttt{StarBase}(\texttt{lab}\colon l) \\
 & & & \;\;\;\mid \; \texttt{StarRec}(\texttt{lab}\colon l, \texttt{string}\colon s, \texttt{left}\colon \mathrm{m}, \texttt{right}\colon)
\end{array}
$$

Figure 3: Regular Expression Matches

Fields, e.g. `.lab`*, are maps from* $\mathit{Match}(\mathit{Lab})$ *with obvious semantics. The field* `.submatch` *is identity where otherwise undefined.*

The function $\cdot_* : \mathit{Match}(\mathit{Lab}) \rightarrow \mathit{Match}(\mathit{Lab}) \rightarrow \mathit{Match}(\mathit{Lab})$ *concatenates a star recursion to a match and is defined*

$\cdot_*(\mathrm{m}_1, \mathrm{m}_2) = \texttt{StarRec}(\texttt{lab}\colon \mathrm{m}_2\, .\mathit{lab}, \texttt{string}\colon \mathrm{m}_1\, .\mathit{string} \cdot \mathrm{m}_2\, .\mathit{string}, \texttt{left}\colon \mathrm{m}_1, \texttt{right}\colon \mathrm{m}_2)$

Definition 7. *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*

$\exists(\mathit{prf}, \mathit{suf} \in X^*), xs = \mathit{prf} \cdot x \cdot \mathit{suf}$

We use the notation $x\, \mathit{char\text{-}of}\; xs$ *to mean* x *is a character of the string* xs.

Definition 8. *Regular Expression Matching*
We define $\mathit{regMatch} : (\mathit{Reg}(\mathit{Lab}) \times \Sigma^*) \rightarrow \mathit{Pow}(\mathit{Match}(\mathit{Lab}))$ *recursively:*
$\mathit{regMatch}(l\colon \varepsilon, \varepsilon) = \{\texttt{Emp}(\texttt{lab}\colon l, \texttt{string}\colon \varepsilon)\}$
$\mathit{regMatch}(l\colon \varepsilon, c \cdot s) = \emptyset$
$\mathit{regMatch}(l\colon c, c) = \{\texttt{Char}(\texttt{lab}\colon l, \texttt{string}\colon c)\}$
$\mathit{regMatch}(l\colon c, s)$ *where* $s \neq c = \emptyset$
$\mathit{regMatch}(l\colon ., c) = \texttt{Any}(\texttt{lab}\colon l, \texttt{string}\colon c)$
$\mathit{regMatch}(l\colon ., c \cdot c' \cdot s) = \emptyset$
$\mathit{regMatch}(l\colon \bot, s) = \emptyset$
$\mathit{regMatch}(l\colon \mathrm{r}_1 \cdot \mathrm{r}_2, s) = \{\texttt{Seq}(\texttt{lab}\colon l, \texttt{string}\colon s, \texttt{right}\colon \mathrm{m}_1, \texttt{left}\colon \mathrm{m}_2)$
$\qquad \mid \exists(s', s'' \in \Sigma^*), s' \cdot s'' = s \wedge \mathrm{m}_1 \in \mathit{regMatch}(\mathrm{r}_1, s') \wedge \mathrm{m}_2 \in \mathit{regMatch}(\mathrm{r}_2, s'')\}$

$\mathit{regMatch}(l\colon \mathrm{r}_1 \cup \mathrm{r}_2, s) =$
$\qquad \{\texttt{JoinL}(\texttt{lab}\colon l, \texttt{string}\colon s, \texttt{subexpr}\colon \mathrm{m}) \mid \mathrm{m}\, \mathit{regMatch}(\mathrm{r}_1, s)\} \cup$
$\qquad \{\texttt{JoinR}(\texttt{lab}\colon l, \texttt{string}\colon s, \texttt{subexpr}\colon \mathrm{m}) \mid \mathrm{m}\, \mathit{regMatch}(\mathrm{r}_2, s)\}$

$\mathit{regMatch}(l\colon \mathrm{r}_1 \cap \mathrm{r}_2, s) =$

$\{\texttt{Meet}(\texttt{lab:}\ l, \texttt{string:}\ s, \texttt{left:}\ \mathrm{m}_1, \texttt{right:}\ \mathrm{m}_1) \mid \mathrm{m}_1 \in \mathit{regMatch}(\mathrm{r}_1, s) \wedge \mathrm{m}_2 \in \mathit{regMatch}(\mathrm{r}_2, s)\}$

$$\mathit{regMatch}(l\colon \neg\,\mathrm{r}, s) = \begin{cases} \mathit{match}(\mathit{regMatch}(\mathrm{r}, s)) \equiv \emptyset & \{\texttt{Neg}(\texttt{lab:}\ l, \texttt{string:}\ s)\} \\ \mathit{otherwise} & \emptyset \end{cases}$$

$$\mathit{regMatch}(l\colon\ \mathrm{r}^*, s) = \{\mathit{fold}(\cdot_*, \texttt{StarBase}(\texttt{lab:}\ l), \mathit{ms} \in \mathit{Match}(\mathit{Lab})^*) \mid \exists \mathit{ss} \in \Sigma^{**}, \bullet \mathit{ss} = s \wedge \forall i, \mathit{ms}_i \in \mathit{regMatch}(\mathrm{r}, \mathit{ss}_i)\}$$

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.

It follows from the definitions alone that for arbitrary (r, s)*, and for an arbitrary match* m *in* $\mathit{regMatch}(\mathrm{r}, s)$, $\mathrm{m}.\mathit{string} = s$.

Definition 9. *Well Founded Pairs*
We define $\mathit{labelToExpr} : \mathit{Peg}(\mathit{Lab}_1) \cup \mathit{Reg}(\mathit{Lab}_2) \to (\mathit{Lab}_1 \cup \mathit{Lab}_2 \to \mathit{Pow}(\mathit{Peg}(\mathit{Lab}_1) \cup \mathit{Reg}(\mathit{Lab}_2)))$ *recursively on the syntax of* $\mathit{Peg}(\mathit{Lab}_1)$ *and* $\mathit{Reg}(\mathit{Lab}_2)$*, collecting the parsing expression grammar corresponding to the label.*

A parsing expression grammar p *or regular expression* r *is in the set of well formed expressions,* $\mathit{WF}(\mathit{Lab})$*, iff for every label* $l \in \mathit{Lab}$, $\mathit{labelToExpr}(\mathrm{p})(l)$ *(resp* $\mathit{labelToExpr}(\mathrm{r})(l)$*) has no more than one element.*

A pair $(\mathrm{p}, \mathrm{r}) \in \mathit{Peg}(\mathit{Lab}_1) \times \mathit{Reg}(\mathit{Lab}_2)$ *is in the set of well formed pairs* $\mathit{WF2}(\mathit{Lab}_1 \cup \mathit{Lab}_2)$ *if for every label, the pointwise union* $\mathit{labelToExpr}(\mathrm{p})$ *and* $\mathit{labelToExpr}(\mathrm{r})$ *applied to the label has no more than one element.*

WF and WF2 informally mean expressions and pairs where each label is different.

Definition 10. *Parsing Expression Grammar Translation*

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

In order to define the output set, we need to have a labelling scheme for combining these; so the labelling scheme will first be detailed, followed by the definition of the function.

$$
\begin{array}{lclll}
l_{pr} & \in & \mathit{PegregLab}(\mathit{Lab}) & ::= & (l_1 \in \mathit{Lab}) \\
 & & & | & \mathit{PRLabChar}(l \in \mathit{Lab}) \\
 & & & | & \mathit{PRLabSeq}(l \in \mathit{Lab}) \\
 & & & | & \mathit{PRLabUnionWildcard}(l \in \mathit{Lab}) \\
 & & & | & \ldots
\end{array}
$$

Figure 4: Labelling Scheme

Below, the labels in the output will be given implicitly, with examples shown in more detail to make it clear how the labels are formed.

$$
\begin{aligned}
&\mathit{pegreg} : (\mathit{Peg}(\mathit{Lab}_1) \times \mathit{Reg}(\mathit{Lab}_2)) \to \mathit{Reg}(\mathit{PegregLab}(\mathit{Lab}_1 \cup \mathit{Lab}_2)) \\
&\mathit{pegreg}(c, \mathrm{r}) = c \cdot \mathrm{r} && (1) \\
&\mathit{pegreg}(\mathrm{p}_1 \cdot \mathrm{p}_2, \mathrm{r}) = \mathit{pegreg}(\mathrm{p}_1, \mathit{pegreg}(\mathrm{p}_2, \mathrm{r})) && (2) \\
&\mathit{pegreg}(\mathrm{p}_1 \,/\, \mathrm{p}_2, \mathrm{r}) = \mathit{pegreg}(\mathrm{p}_1, \mathrm{r}) \cup (\mathit{pegreg}(\mathrm{p}_2, \mathrm{r}) \cap \mathit{negate}(\mathrm{p}_1) \cdot .^*) && (3) \\
&\mathit{pegreg}(\mathrm{p}^*, \mathrm{r}) = \mathit{pegreg}(\mathrm{p}, \varepsilon)^* \cdot (\mathrm{r} \cap \mathit{negate}(\mathrm{p}) \cdot .^*) && (4)
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{negate} : \mathit{Peg}(\mathit{Lab}_1) \to \mathit{Reg}(\mathit{PegregLab}(\mathit{Lab}_1 \cup \mathit{Lab}_2)) \\
&\mathit{negate}(c) = . \cap \neg c && (5) \\
&\mathit{negate}(\mathrm{p}_1 \cdot \mathrm{p}_2) = \mathit{negate}(\mathrm{p}_1) \cup \mathit{pegreg}(\mathrm{p}_1, \mathit{negate}(\mathrm{p}_2)) && (6) \\
&\mathit{negate}(\mathrm{p}_1 \,/\, \mathrm{p}_2) = \mathit{negate}(\mathrm{p}_1) \cap \mathit{negate}(\mathrm{p}_2) && (7) \\
&\mathit{negate}(\mathrm{p}^*) = \bot && (8)
\end{aligned}
$$

To make it clear how the labels are formed, let .lab be, by abuse of notation, a projection from regular exprssions to their labels, and see the case for char in more detail:

$$
\begin{aligned}
&\mathit{pegreg} : (\mathit{Peg}(\mathit{Lab}_1) \times \mathit{Reg}(\mathit{Lab}_2)) \to \mathit{Reg}(\mathit{PegregLab}(\mathit{Lab})) \\
&\mathit{pegreg}(c, \mathrm{r}) = \mathit{PRLabSeq}(\mathrm{r}.\,\mathit{lab})\colon (\mathit{PRLabChar}(\mathrm{r}\,.\mathit{lab})\colon c) \cdot \mathrm{r}
\end{aligned}
$$

For another example, consider the choice case. If viewed in the order from the innermost subexpression to the outermost, the recursion constructs a wildcard, then a kleene star with this subexpression, then negate(p_1)*, then the concatenation of the two, etc.*

So negate(p_1) *is first computed, then PRLabUnionWildcard(negate*(p_1)*.lab) is used to give a label to the expression that matches any character. This pattern of computing new, unique labels from the subexpressions continues until the entire expression is labelled.*

This labelling scheme will give pegreg the property that it maps WF2 pairs to WF regular expressions.

Definition 11. *The function*

$\mathit{extract}_{\mathit{peg}} : (\mathit{Peg}(\mathit{Lab}_1) \times \mathit{Match}(\mathit{PegregLab}(\mathit{Lab}_1 \cup \mathit{Lab}_2))) \nrightarrow \mathit{Match}(\mathit{PegregLab}(\mathit{Lab}_1 \cup \mathit{Lab}_2))$

applied to a pair (p, m) *extracts the part corresponding to the peg match.*

For any parsing expression grammar p*, any regular expression* r*, any string* s*, and any match* m *in* $\mathit{regMatch}(\mathit{pegreg}(\mathrm{p}, \mathrm{r}), s)$*,* $\mathit{extract}_{\mathit{peg}}$ *is total.*

Similarly, $\mathit{extract}_{\mathit{reg}} : (\mathit{Reg}(\mathit{Lab}_1) \times \mathit{Match}(\mathit{PegregLab}(\mathit{Lab}_1 \cup \mathit{Lab}_2))) \nrightarrow \mathit{Match}(\mathit{PegregLab}(\mathit{Lab}_1 \cup \mathit{Lab}_2))$ *is defined.*

Proposition 1. *Translation Correspondance*
Let (p, r) *be an arbitrary pair of parsing and regular expressions, in* $\mathit{WF2}(\mathit{Lab}_1 \cup \mathit{Lab}_2)$*. Then the following will hold:*

1. *Let* s *be an arbitrary string, and* m *be an arbitrary match in* $\mathit{regMatch}(\mathit{pegreg}(\mathrm{p}, \mathrm{r}), s)$*. Then* $\mathrm{m}.\mathit{string} = \mathit{extract}_{\mathit{peg}}(\mathrm{p}, \mathrm{m}).\mathit{string} \cdot \mathit{extract}_{\mathit{reg}}(\mathrm{r}, \mathrm{m}).\mathit{string}$*.*

2. *Let* s *be an arbitrary string. Suppose* $\mathit{pegMatch}(\mathrm{p}, s) = \bot$*. Then the set* $\mathit{regMatch}(\mathit{negate}(\mathrm{p}), s)$ *contains at least one element, and for all matches* m *in* $\mathit{regMatch}(\mathit{negate}(\mathrm{p}), s)$*, the following holds:* $\mathrm{m}.\mathit{string} = \mathit{extract}_{\mathit{peg}}(\mathrm{p}, \mathrm{m}).\mathit{string} = s$*.*

3. *Let* s *be an arbitrary string. Supose* $\mathit{pegMatch}(\mathrm{p}, s) = (s', k')$*, and* $\mathit{regMatch}(\mathrm{r}, k)$ *is nonempty. Then* $\mathit{regMatch}(\mathit{pegreg}(\mathrm{p}, \mathrm{r}), s)$ *is nonempty.*

4. *Let* s *be an arbitrary string. Suppose* $\mathit{regMatch}(\mathit{negate}(\mathrm{p}), s)$ *is not empty. Then it contains one element, and* $\mathit{pegMatch}(\mathrm{p}, s) = \bot$*.*

5. *Let* s *be an arbitrary string. For all matches* $\mathrm{m} \in \mathit{regMatch}(\mathit{pegreg}(\mathrm{p}, \mathrm{r}), s)$*,* $\mathit{pegMatch}(\mathrm{p}, s) = (\mathit{extract}_{\mathit{peg}}(\mathrm{m}), \mathit{extract}_{\mathit{reg}}(\mathrm{r}, \mathrm{m}))$ *and* $\mathit{extract}_{\mathit{reg}}(\mathrm{r}, \mathrm{m}) \in \mathit{regMatch}(\mathrm{r}, \mathit{extract}_{\mathit{reg}}(\mathrm{r}, \mathrm{m}).\mathit{string})$*.*

In the above proposition, the first property requires that matches from *pegreg* split the string into a peg portion and a reg portion. The rest of the properties give forward and backwards correctness for *negate* and *pegreg*, in that order.

Proof. I think we can do a proof by induction. I'll have to take the leap and try them out. □