From Regular Expressions to Parsing Expression Grammars

Sérgio Medeiros[1], Fabio Mascarenhas[2], Roberto Ierusalimschy[3]

[1] Department of Computer Science – UFS – Aracaju – Brazil
sergio@ufs.br
[2] Department of Computer Science – UFRJ – Rio de Janeiro – Brazil
fabiom@dcc.ufrj.br
[3] Department of Computer Science – PUC-Rio – Rio de Janeiro – Brazil
roberto@inf.puc-rio.br

Abstract. Regular expressions are a formalism for expressing regular languages. Regex pattern matching libraries borrow the syntax of regular expressions, but introduce ad-hoc extensions that cannot be expressed within the regular expression formalism, and whose efficiency is difficult to estimate. PEGs are a formalism that can describe all deterministic context-free languages and that has a simple computational model. We present a new formalization of regular expressions via transformation to PEGs, and show that this formalization easily accommodates some of the extensions, and helps estimate the efficiency of expressions.

Keywords: regular expressions, parsing expression grammars, natural semantics, pattern matching

1 Introduction

Regular expressions are a concise way for describing regular languages by using an algebraic notation. Their syntax has seen wide use in pattern matching libraries for programming languages, where they are known as *regexes* and used to specify the pattern that a user is trying to match a string against, or, more commonly, the pattern to search for in a string.

Regexes may look like regular expressions, but they can have syntactical and semantical extensions that are difficult, or even impossible, to express through pure regular expressions. As these extensions do not have a formal model, the meaning of regex patterns that use the extensions is not clear, and may vary among different regex libraries [5], or even among different implementations of the same regex library [4].

Regexes not only do not have a formal model of their behavior but also do not have a cost model of their operation, which makes it difficult for users to determine the efficiency of regex patterns. Simple modifications can make the time complexity of a pattern go from linear to exponential [2, 1].

Another consequence of the lack of a formal model for regexes is making the implementation of a regex library a complex task, further causing problems of different behavior among implementations of regex libraries.

Parsing Expression Grammars (PEGs) [3] are a formalism that can express all deterministic context-free languages, which means that PEGs can also express all regular languages. The syntax of PEGs also borrows from regular expressions, and they can be used as an alternative for regexes in pattern matching libraries that is easier to reason about [7].

In this paper, we want to show that we do not need to abandon regexes to gain the benefits of having a clear formal model that PEGs have. We can describe the meaning of regex patterns by conversion to PEGs, which helps reasoning about the behavior of complex regexes. Moreover, PEGs can be efficiently executed by a parsing machine that has a clear cost model that we can use to reason about the time complexity of matching a given pattern [7, 10]. We believe that the combination of the regex to PEG conversion and the PEG parsing machine can be used to build implementations of regex libraries that are simpler than the current ones.

The main contribution of this paper is the formalization of a translation from plain regular expressions to PEGs. We present formalizations of both regular expressions and PEGs in the framework of natural semantics [8], and use these to show the similarities and differences between regular expressions and PEGs. We then define a transformation that converts a regular expression to a PEG, and prove its correctness. Finally, we show how this transformation can be adapted to accommodate four regex extensions: independent expressions, possessive and lazy repetition, and lookahead.

In the next section, we present our formalizations of regular expressions and PEGs, and discuss when a regular expression has the same meaning when interpreted as a PEG, along with the intuition behind our transformation. In Section 3, we formalize our transformation from regular expressions to PEGs and prove its correctness. In Section 4 we show how our transformation can accommodate some regex extensions. Finally, in Section 5 we discuss some related work and present our conclusions.

2 Regular Expressions and PEGs

Given a finite alphabet T, we can define a regular expression e inductively as follows, where $\mathtt{a} \in T$, and both e_1 and e_2 are also regular expressions:

$$e \;=\; \varepsilon \;\mid\; a \;\mid\; e_1\, e_2 \;\mid\; e_1 \mid e_2 \;\mid\; e_1^*$$

Traditionally, a regular expression can also be $\emptyset$, but we will not consider it, as any expression with $\emptyset$ as a subexpression can either be rewritten without $\emptyset$ or reduces to $\emptyset$.

The language of a regular expression e, $L(e)$, is traditionally defined through operations on sets [6]. Intuitively, the languages of ε and a are singleton sets with the corresponding symbols, the language of $e_1\, e_2$ is given by concatenating all strings of $L(e_1)$ with all strings of $L(e_2)$, the language of $e_1 \mid e_2$ is the union of the languages of e_1 and e_2, and the language of e_1^* is the Kleene star of the language of e_1.

Empty String $\dfrac{}{\varepsilon\ x \overset{\text{RE}}{\rightsquigarrow} x}\,(\textbf{empty.1})$ **Character** $\dfrac{}{a\ ax \overset{\text{RE}}{\rightsquigarrow} x}\,(\textbf{char.1})$

Concatenation $\dfrac{e_1\ xyz \overset{\text{RE}}{\rightsquigarrow} yz \quad e_2\ yz \overset{\text{RE}}{\rightsquigarrow} z}{e_1\, e_2\ xyz \overset{\text{RE}}{\rightsquigarrow} z}\,(\textbf{con.1})$

Choice $\dfrac{e_1\ xy \overset{\text{RE}}{\rightsquigarrow} y}{e_1 \mid e_2\ xy \overset{\text{RE}}{\rightsquigarrow} y}\,(\textbf{choice.1})$ $\dfrac{e_2\ xy \overset{\text{RE}}{\rightsquigarrow} y}{e_1 \mid e_2\ xy \overset{\text{RE}}{\rightsquigarrow} y}\,(\textbf{choice.2})$

Repetition $\dfrac{}{e^*\ x \overset{\text{RE}}{\rightsquigarrow} x}\,(\textbf{rep.1})$ $\dfrac{e\ xyz \overset{\text{RE}}{\rightsquigarrow} yz \quad e^*\ yz \overset{\text{RE}}{\rightsquigarrow} z}{e^*\ xyz \overset{\text{RE}}{\rightsquigarrow} z}\ , x \neq \varepsilon\,(\textbf{rep.2})$

Fig. 1. Natural semantics of relation $\overset{\text{RE}}{\rightsquigarrow}$

Instead of defining the language of an expression e through operations on sets, we will define a *matching* relation for regular expressions, $\overset{\text{RE}}{\rightsquigarrow}$. Informally, we will have $e\ xy \overset{\text{RE}}{\rightsquigarrow} y$ if and only if the expression e matches the prefix x of input string xy. The set of strings that expression e matches will be the language of e, that is, $L(e) = \{x \mid e\ xy \overset{\text{RE}}{\rightsquigarrow} y, xy \in T^*\}$.

Formally, we define $\overset{\text{RE}}{\rightsquigarrow}$ via natural semantics, using the set of inference rules in Figure 1. We have $e\ xy \overset{\text{RE}}{\rightsquigarrow} y$ if and only if we can build a proof tree for this statement using the inference rules. The rules follow naturally from the expected behavior of each expression: rule **empty.1** says that ε matches itself and does not consume the input; rule **char.1** says that a symbol matches and consumes itself if it is the beginning of the input; rule **con.1** says that a concatenation uses the suffix of the first match as the input for the next; rules **choice.1** and **choice.2** say that a choice can match the input using either option; finally, rules **rep.1** and **rep.2** say that a repetition can either match ε and not consume the input or match its subexpression and match the repetition again on the suffix that the subexpression left.

The following lemma proves that the set $L(e)$ given by our relation $\overset{\text{RE}}{\rightsquigarrow}$ is the same as the set $L(e)$ given by the traditional definition of regular expressions for a given expression e.

Lemma 1. *Given a regular expression e and a string x, for any string y we have that $x \in L(e)$ if and only if $e\ xy \overset{\text{RE}}{\rightsquigarrow} y$.*

Proof. ($\Rightarrow$): By induction on the complexity of the pair (e, x). Given the pairs (e_1, x_1) and (e_2, x_2), the first pair is more complex than the second one if and only if either e_2 is a proper subexpression of e_1 or $e_1 = e_2$ and $|x_1| > |x_2|$. The base cases are $(\varepsilon, \varepsilon)$, (a, a), and (e^*, ε), and their proofs follow by application of rules **empty.1**, **char.1**, or **rep.1**, respectively. Cases $(e_1\, e_2, x)$ and $(e_1 \mid e_2, x)$ use

a straightforward application of the induction hypothesis on the subexpressions, followed by application of rule **con.1** or one of the **choice** rules. For case (e^*, x) where $x \neq \varepsilon$, we know by the definition of the Kleene star that $x \in L^i(e)$ with $i > 0$, where $L^i(e)$ is $L(e)$ concatenated with itself i times. This means that we can decompose x into x_1x_2 where $x_1 \in L(e)$ and $x_2 \in L^{i-1}(e)$. Again by the definition of the Kleene star this means that $x_2 \in L(e^*)$. The proof now follows by the induction hypothesis on (e, x_1) and (e^*, x_2) and an application of rule **rep.2**.

($\Leftarrow$): By induction on the height of the proof tree for $e\ xy \overset{\text{RE}}{\rightsquigarrow} y$. Most cases are straightforward; the interesting case is when the proof tree concludes with rule **rep.2**. By the induction hypothesis we have that $x \in L(e)$ and $y \in L(e^*)$. By the definition of the Kleene star we have that $y \in L^i(e)$, so $xy \in L^{i+1}(e)$ and, again by the Kleene star, $xy \in L(e^*)$, which concludes the proof.

Parsing expression grammars borrow the syntax of regular expressions (leaving out the $\emptyset$ expression) and add two new expressions: A for a non-terminal, and $!e$ for a *not-predicate* of expression e. A PEG G is a tuple (V, T, P, p_S) where V is the set of non-terminals, T is the alphabet (set of terminals), P is a function from V to expressions and p_S is the expression that the PEG matches. We will use the notation $G[p]$ for a grammar derived from G where p_S is replaced by p while keeping V, T, and P the same.

While the syntax of the expressions of a PEG are a superset of regular expressions, the behavior of the choice and repetition operators is very different. Choice in PEGs is *ordered*; a PEG will only try to match the right side of a choice if the left side cannot match any prefix of the input. Repetition in PEGs is *greedy*; a repetition will always consume as much of the input as it can match[4]. To formally define ordered choice and greedy repetition we also need a way to express that an expression does not match a prefix of the input, so we need to introduce `fail` as a possible outcome of a match.

Figure 2 gives the definition of $\overset{\text{PEG}}{\rightsquigarrow}$, the matching relation for PEGs. As with regular expressions, we say that $G\ xy \overset{\text{PEG}}{\rightsquigarrow} y$ to express that the grammar G matches the prefix x of input string xy. We mark with a $*$ the rules that have been changed from Figure 1, and mark with a $+$ the rules that were added. Unmarked rules are unchanged from Figure 1.

We have six new rules, and two changed rules. New rules **char.2** and **char.3** generate `fail` in the case that the expression cannot match the symbol in the beginning of the input. New rule **con.2** says that a concatenation fails if its left side fails. New rule **var.1** says that to match a non-terminal we have to match the associated expression. New rules **not.1** and **not.2** say that a not predicate never consumes input, but fails if its subexpression matches a prefix of the input.

The change in rule **con.1** is trivial and only serves to propagate `fail`, so we do not consider it an actual change. The changes to rules **choice.2** and **rep.1** are what actually implements ordered choice and greedy repetition, respectively.

[4] Greedy repetition is a consequence of ordered choice, as e^* is the same as expression A where A is a fresh non-terminal and $P(A) = eA \mid \varepsilon$.

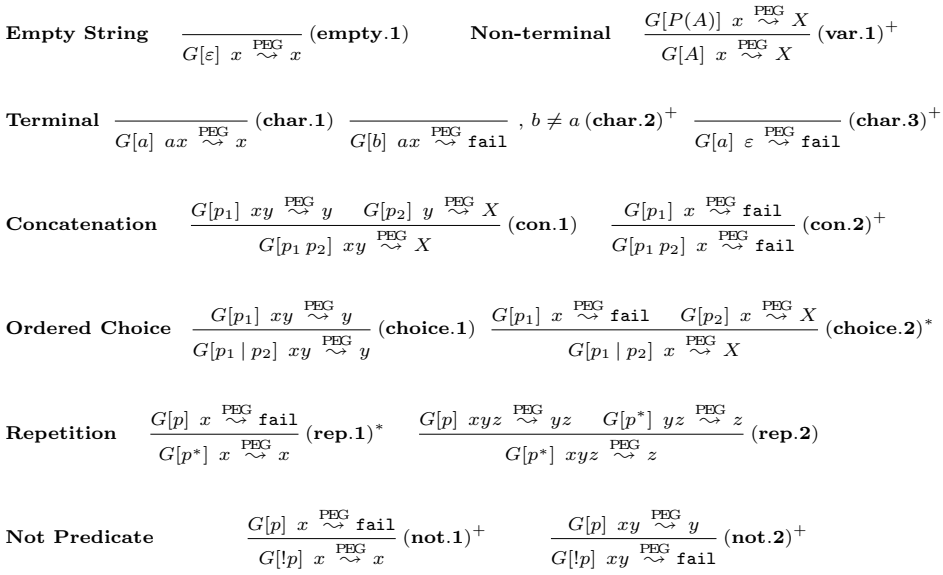

Fig. 2. Definition of Relation $\overset{\text{PEG}}{\rightsquigarrow}$ through Natural Semantics

Rule **choice.2** says that we can only match the right side of the choice if the left side fails, while rule **rep.1** says that a repetition only stops if we try to match its subexpression and fail.

It is easy to see that PEGs are deterministic; that is, a given PEG G can only have a single result (either `fail` or a suffix of x) for some input x, and only a single proof tree for this result. There can still be cases where no result is possible, such as left recursion or a repetition e^* where e matches the empty string. If the PEG G always yields a result for any input in T^* then we say that G is *complete* [3]. From now on we will assume that any PEG we consider is complete unless stated otherwise.

The syntax of the expressions that form a PEG are a superset of the syntax of regular expressions, so syntactically any regular expression e has a corresponding PEG $G_e = (V, T, P, e)$, where V and P can be anything. We can prove that $L(G_e) \subseteq L(e)$ by a simple induction on the height of proof trees for $G_e\ xy \overset{\text{PEG}}{\rightsquigarrow} y$, but it is easy to show examples where $L(G_e)$ is a proper subset of $L(e)$, so the regular expression and its corresponding PEG have different languages.

For example, expression $a \mid ab$ has the language {`a`, `ab`} as a regular expression but {`a`} as a PEG, because on an input with prefix `ab` the left side of the choice always matches and keeps the right side from being tried. The same happens with expression $a\,(b \mid bb)$, which has language {`ab`, `abb`} as a regular expression and {`ab`} as a PEG, and on inputs with prefix `abb` the left side of the choice keeps the right side from matching.

A different situation happens with expression $(a \mid aa)\, b$. As a regular expression, its language is {`ab`, `aab`} while as a PEG it is {`ab`}, but the PEG fails on an input with prefix `aab`.

If we change $a \mid ab$ and $a\,(b \mid bb)$ so they have the *prefix property*[5], by adding an end-of-input marker \$, we have the expressions $(a \mid ab)\,\$$ and $a\,(b \mid bb)\,\$$. Now their languages as regular expressions are {`a$`, `ab$`} and {`ab$`, `abb$`}, respectively, but the first expression fails as a PEG on the input `abb` and the second expression fails on the input `abb`. But now we have a pattern $(e_1 \mid e_2)\, e_3$ on all three expressions. If we distribute e_3 over the choice we have $e_1\, e_3 \mid e_2\, e_3$.

If we do the above operation on all three expressions we have $a\$ \mid ab\$$, $a\,(b\$ \mid bb\$)$, and $ab \mid aab$, respectively, and all three expressions have the same language when interpreted either as a regular expression or as a PEG.

We will say that a PEG G and a regular expression e over the same alphabet T are *equivalent* if the following conditions hold for every input string xy:

$$G\ xy \overset{\text{PEG}}{\rightsquigarrow} y \Rightarrow e\ xy \overset{\text{RE}}{\rightsquigarrow} y \tag{1}$$

$$e\ xy \overset{\text{RE}}{\rightsquigarrow} y \Rightarrow G\ xy \overset{\text{PEG}}{\not\rightsquigarrow} \texttt{fail} \tag{2}$$

That is, a PEG G and a regular expression e are equivalent if $L(G) \subseteq L(e)$ and G does not fail for any string with a prefix in $L(e)$. In the examples above, regular expressions $a \mid ab$, $a\,(b \mid bb)$, $a\$ \mid ab\$$, $a\,(b\$ \mid bb\$)$, and $ab \mid aab$ are all equivalent with their corresponding PEGs, while $(a \mid ab)\,\$$, $a\,(b \mid bb)\,\$$, and $(a \mid aa)\, b$ are not.

Equivalence together with the prefix property yields the following lemma:

Lemma 2. *If a regular expression e with the prefix property and a PEG G are equivalent then $L(G) = L(e)$.*

Proof. We just need to prove that $L(e) \subseteq L(G)$. Suppose there is a string $x \in L(e)$; this means that $e\ xy \overset{\text{RE}}{\rightsquigarrow} y$ for any y. But from equivalence this means that $G\ xy \overset{\text{PEG}}{\not\rightsquigarrow} \texttt{fail}$. As G is complete, we have $G\ xy \overset{\text{PEG}}{\rightsquigarrow} y'$. By equivalence, the prefix of xy that G matches is in $L(e)$. Now y cannot be a proper suffix of y' neither y' a proper suffix of y, or the prefix property would be violated. This means that $y' = y$, and $x \in L(G)$, completing the proof.

We can now present an overview on how we will transform a regular expression e to a PEG that recognizes the same language. We first need to guarantee that e has the prefix property, which is easily done by adding an end-of-input marker to e to get $e\$$. We then need to transform subexpressions of the form $(e_1 \mid e_2)\, e_3$ to $e_1 e_3 \mid e_2 e_3$. The end result will be a regular expression that is equivalent to itself when viewed as a PEG.

Transforming repetition is trickier, but we just have to remember that $e_1^* e_2 \equiv (e_1\, e_1^* \mid \varepsilon)\, e_2 \equiv (e_1\, e_1^*\, e_2) \mid e_2$. Naively transforming the first expression to the third does not work, as we end up with $e_1^* e_2$ in the expression again, but we can add

[5] There are no distinct strings x and y in the language such that x is a prefix of y.

a non-terminal A to the PEG with $P(A) = e_1 A \mid e_2$ and then replace $e_1^* e_2$ with A in the original expression.

As an example, let us consider the regular expression $b^* b\$$. Its language is $\{\texttt{b\$}, \texttt{bb\$}, \ldots\}$, but when interpreted as a PEG the language is $\emptyset$. This expression gets rewritten as A with $P(A) = bA \mid b\$$. This PEG has the same language as the original regular expression; for example, given the input `bb$`, this grammar matches the first `b` through subexpression b of $b\,A$, and then A tries to match the rest of the input, `b$`. So, once more subexpression b of $b\,A$ matches `b` and then A tries to match the rest of the input, `$`. Since both $b\,A$ and $b\$$ fail to match `$`, A fails, and thus $b\,A$ fails for input `b$`. Now we try $b\$$, which successfully matches `b$`, and the complete match succeeds.

The next section formalizes our transformation, and proves that for any regular expression e it will give a PEG that is equivalent to e, which by extension yields a PEG that recognizes the same language as e if it has the prefix property.

3 Transforming Regular Expressions to PEGs

This section presents function Π, a formalization of the transformation we outlined in the previous section. The function Π transforms a regular expression e using a PEG G_k that is equivalent to a regular expression e_k to yield a PEG that is equivalent to the regular expression $e\,e_k$.

The intuition behind Π is that G_k is a *continuation* for the regular expression e, being what should be matched after matching e. We use this continuation when transforming choices and repetitions to do the transformations of the previous section; for a choice, the continuation is distributed to both sides of the choice. For a repetition, it is used as the right side for the new non-terminal, and the left side of this non-terminal is the transformation of the repetition's subexpression with the non-terminal as continuation.

For a concatenation, the transformation is the result of transforming the right side with G_k as continuation, then using this as continuation for transforming the left side. This lets the transformation of $(e_1 \mid e_2)\,e_3$ works as expected: we transform e_3 and then use the PEG as the continuation that we distribute over the choice.

We can transform a standalone regular expression e by passing a PEG with ε as starting expression as the continuation; this gives us a PEG that is equivalent to the regular expression $e\,\varepsilon$, or e.

Figure 3 has the definition of function Π. Notice how repetition introduces a new non-terminal, and the transformation of choice has to take this into account by using the set of non-terminals and the productions of the result of transforming one side to transform the other side, so there will be no overlap.

We will show how Π works using some examples. In the following discussion, we use the alphabet $T = \{\texttt{a}, \texttt{b}, \texttt{c}\}$, and the continuation grammar $G_k = (\emptyset, T, \emptyset, \varepsilon)$ that is equivalent to the regular expression ε. In our first example, we use the regular expression $(a \mid b \mid c)^*\, a\, (a \mid b \mid c)^*$, which matches an input that has at least one `a`.

$$\Pi(\varepsilon, G_k) = G_k \quad \Pi(a, G_k) = G_k[a\, p_k] \quad \Pi(e_1\, e_2, G_k) = \Pi(e_1, \Pi(e_2, G_k))$$
$$\Pi(e_1 \mid e_2, G_k) = G_2[p_1 \mid p_2], \text{ where } G_2 = (V_2, T, P_2, p_2) = \Pi(e_2, (V_1, T, P_1, p_k)) \text{ and}$$
$$G_1 = (V_1, T, P_1, p_1) = \Pi(e_1, G_k)$$
$$\Pi(e_1^*, G_k) = G\,, \text{ where } G = (V_1, T, P_1 \cup \{A \to p_1 \mid p_k\}, A) \text{ and}$$
$$(V_1, T, P_1, p_1) = \Pi(e_1, (V_k \cup \{A\}, T, P_k, A)) \text{ with } A \notin V_1$$

Fig. 3. Definition of Function Π, where $G_k = (V_k, T, P_k, p_k)$

We first transform the second repetition by evaluating $\Pi((a \mid b \mid c)^*, G_k)$; we first transform $a \mid b \mid c$ with a new non-terminal A as continuation, yielding the PEG $aA \mid bA \mid cA$, then combine it with ε to yield the PEG A where A has the production below:

$$A \to a\,A \mid b\,A \mid c\,A \mid \varepsilon$$

Next is the concatenation with a, yielding the PEG aA. We then use this PEG as continuation for transforming the first repetition. This transformation uses a new non-terminal B as a continuation for transforming $a \mid b \mid c$, yielding $aB \mid bB \mid cB$, then combines it with aA to yield the PEG B with the productions below:

$$B \to a\,B \mid b\,B \mid c\,B \mid a\,A \qquad A \to a\,A \mid b\,A \mid c\,A \mid \varepsilon$$

When the original regular expression matches a given input, we do not know how many `a`'s the first repetition matches, because the semantics of regular expressions is non-deterministic. Implementations usually resolve ambiguities by the longest match rule, where the first repetition will match all but the last `a` of the input. PEGs are deterministic by construction, and the PEG generated by Π obeys the longest match rule. The alternative $a\,A$ of non-terminal B will only be tried if all the alternatives fail, which happens in the end of the input. The PEG then backtracks until the last `a` is found, where it matches the last `a` and proceeds with non-terminal A.

The regular expression $(b \mid c)^*\, a\, (a \mid b \mid c)^*$ defines the same language as the regular expression of the first example, but without the ambiguity.

Now Π with continuation G_k yields the following PEG B:

$$B \to b\,B \mid c\,B \mid a\,A \qquad A \to a\,A \mid b\,A \mid c\,A \mid \varepsilon$$

Although the productions of this PEG and the previous one recognize the same language, the second PEG is more efficient, as it will not have to reach the end of the input and then backtrack until finding the last `a`. This is an example on how we can use our semantics and the transformation Π to reason about the behavior of a regular expression.

The expressions in the two previous examples are *well-formed*. A regular expression e is well-formed if it does not have a subexpression e_i^* where $\varepsilon \in L(e_i)$.

If e is a well-formed regular expression and G_k is a complete PEG then $\Pi(e, G_k)$ is also complete. In Section 3.1 we will show how to obtain a well-formed regular expression that recognizes the same language as a non-well-formed regular expression.

We will now prove that our transformation Π is correct, that is, if e is a well-formed regular expression and G_k is a PEG equivalent to a regular expression e_k then $\Pi(e, G_k)$ is equivalent to $e\,e_k$. The proofs use a small technical lemma: each production of PEG G_k is also in PEG $\Pi(e, G_k)$, for any regular expression e. This lemma is straightforward to prove by structural induction on e.

We will prove each property necessary for equivalence separately; equivalence will then be a direct corollary of those two proofs. To prove the first property we need an auxiliary lemma that states that the continuation grammar is indeed a continuation, that is if the PEG $\Pi(e, G_k)$ matches a prefix x of a given input xy then we can split x into v and w with $x = vw$ and G_k matching w.

Lemma 3. *Given a regular expression e, a PEG G_k, and an input string xy, if $\Pi(e, G_k)\ xy \overset{PEG}{\rightsquigarrow} y$ then there is a suffix w of x such that $G_k\ wy \overset{PEG}{\rightsquigarrow} y$.*

Proof. By induction on the complexity of the pair (e, xy). The interesting case is e^*. In this case $\Pi(e^*, G_k)$ gives us a grammar $G = (V_1, T, P, A)$, where $A \to p_1 \mid p_k$. By **var.1** we know that $G[p_1 \mid p_k]\ xy \overset{\text{PEG}}{\rightsquigarrow} y$. There are now two subcases to consider, **choice.1** and **choice.2**.

For subcase **choice.2**, we have $G[p_k]\ xy \overset{\text{PEG}}{\rightsquigarrow} y$. But then we have that $G_k[p_k]\ xy \overset{\text{PEG}}{\rightsquigarrow} y$ because any non-terminal that p_k uses to match xy is in both G and G_k and has the same production in both. The string xy is a suffix of itself, and p_k is the starting expression of G_k, closing this part of the proof.

For subcase **choice.1** we have $\Pi(e, \Pi(e^*, G_k))\ xy \overset{\text{PEG}}{\rightsquigarrow} y$, and by the induction hypothesis $\Pi(e^*, G_k)\ wy \overset{\text{PEG}}{\rightsquigarrow} y$. We can now use the induction hypothesis again, on the length of the input, as w must be a proper suffix of x. We conclude that $G_k\ w'y \overset{\text{PEG}}{\rightsquigarrow} y$ for a suffix w' of w, and so a suffix of x, ending the proof.

The following lemma proves that if the first property of equivalence holds between a regular expression e_k and a PEG G_k then it will hold for $e\,e_k$ and $\Pi(e, G_k)$ given a regular expression e.

Lemma 4. *Given regular expressions e and e_k and a PEG G_k, where $G_k\ wy \overset{PEG}{\rightsquigarrow} y \Rightarrow e_k\ wy \overset{RE}{\rightsquigarrow} y$, if $\Pi(e, G_k)\ vwy \overset{PEG}{\rightsquigarrow} y$ then $e\,e_k\ vwy \overset{RE}{\rightsquigarrow} y$.*

Proof. By induction on the complexity of the pair (e, vwy). The interesting case is e^*. In this case, $\Pi(e^*, G_k)$ gives us a PEG $G = (V_1, T, P, A)$, where $A \to p_1 \mid p_k$. By **var.1** we know that $G[p_1 \mid p_k]\ vwy \overset{\text{PEG}}{\rightsquigarrow} y$. There are now two subcases, **choice.1** and **choice.2** of $\overset{\text{PEG}}{\rightsquigarrow}$.

For subcase **choice.2**, we can conclude that $G_k\,vwy \overset{\text{PEG}}{\rightsquigarrow} y$ because p_k is the starting expression of G_k and any non-terminals it uses have the same production both in G and G_k. We now have $e_k\ \ vwy \overset{\text{RE}}{\rightsquigarrow} y$. By **choice.2** of $\overset{\text{RE}}{\rightsquigarrow}$ we have

$e\,e^*\,e_k \mid e_k\ vwy \overset{\text{RE}}{\rightsquigarrow} y$, but $e\,e^*\,e_k \mid e_k \equiv e^*\,e_k$, so $e^*\,e_k\ vwy \overset{\text{RE}}{\rightsquigarrow} y$, ending this part of the proof.

For subcase **choice.1**, we have $\Pi(e,\ \Pi(e^*,\ G_k))\ vwy \overset{\text{PEG}}{\rightsquigarrow} y$, and by Lemma 3 we have $\Pi(e^*,\ G_k)\ wy \overset{\text{PEG}}{\rightsquigarrow} y$. The string v is not empty, so we can use the induction hypothesis and Lemma 3 again to conclude $e^*\,e_k\ wy \overset{\text{RE}}{\rightsquigarrow} y$. Then we use the induction hypothesis on $\Pi(e,\ \Pi(e^*,\ G_k))\ vwy \overset{\text{PEG}}{\rightsquigarrow} y$ to conclude $e\,e^*\,e_k\ vwy \overset{\text{RE}}{\rightsquigarrow} y$. We can now use rule **choice.1** of $\overset{\text{RE}}{\rightsquigarrow}$ to get $e\,e^*\,e_k \mid e_k\ vwy \overset{\text{RE}}{\rightsquigarrow} y$, but $e\,e^*\,e_k \mid e_k \equiv e^*\,e_k$, so $e^*\,e_k\ vwy \overset{\text{RE}}{\rightsquigarrow} y$, ending the proof.

The following lemma proves that if the second property of equivalence holds between a regular expression e_k and a PEG G_k then it will hold for $e\,e_k$ and $\Pi(e,\ G_k)$ given a regular expression e.

Lemma 5. *Given regular expressions e and e_k and a PEG G_k, where Lemma 4 holds and we have $e_k\ wy \overset{RE}{\rightsquigarrow} y \Rightarrow G_k\ wy \overset{PEG}{\not\rightsquigarrow} \texttt{fail}$, if $e\,e_k\ vwy \overset{RE}{\rightsquigarrow} y$ then $\Pi(e,\ G_k)\ vwy \overset{PEG}{\not\rightsquigarrow} \texttt{fail}$.*

Proof. By induction on the complexity of the pair $(e,\ vwy)$. The interesting case is e^*. We will use again the equivalence $e^*\,e_k \equiv e\,e^*\,e_k \mid e_k$. There are two subcases, **choice.1** and **choice.2** of $\overset{\text{RE}}{\rightsquigarrow}$.

For subcase **choice.1**, we have that e matches a prefix of vwy by rule **con.1**. As e^* is well-formed this prefix is not empty, so $e^*\,e_k\ v'wy \overset{\text{RE}}{\rightsquigarrow} y$ for a proper suffix v' of v. By the induction hypothesis we have $\Pi(e^*,\ G_k)\ v'wy \overset{\text{PEG}}{\not\rightsquigarrow} \texttt{fail}$, and we use the induction hypothesis again to conclude $\Pi(e,\ \Pi(e^*,\ G_k))\ vwy \overset{\text{PEG}}{\not\rightsquigarrow} \texttt{fail}$. This PEG is complete, so we can conclude $\Pi(e^*,\ G_k)[p_1 \mid p_k]\ vwy \overset{\text{PEG}}{\not\rightsquigarrow} \texttt{fail}$ using rule **choice.1** of $\overset{\text{PEG}}{\rightsquigarrow}$, and then $\Pi(e^*,\ G_k)\ vwy \overset{\text{PEG}}{\not\rightsquigarrow} \texttt{fail}$ by rule **var.1**, ending this part of the proof.

For subcase **choice.2**, we can assume that there is no proof tree for the statement $e\,e^*\,e_k\ vwy \overset{\text{RE}}{\rightsquigarrow} y$, or we could reduce this subcase to the first one by using **choice.1** instead of **choice.2**. Because $\Pi(e,\ \Pi(e^*,\ G_k))$ is complete we can use modus tollens of Lemma 4 to conclude that $\Pi(e,\ \Pi(e^*,\ G_k))\ vwy \overset{\text{PEG}}{\rightsquigarrow} \texttt{fail}$. We also have $e_k\ vwy \overset{\text{RE}}{\rightsquigarrow} y$, so $G_k\ vwy \overset{\text{PEG}}{\not\rightsquigarrow} \texttt{fail}$. Now we can use rule **choice.2** of $\overset{\text{PEG}}{\rightsquigarrow}$ to conclude $G[p_1 \mid p_k]\ vwy \overset{\text{PEG}}{\not\rightsquigarrow} \texttt{fail}$, and then $\Pi(e^*,\ G_k)\ vwy \overset{\text{PEG}}{\not\rightsquigarrow} \texttt{fail}$ by rule **var.1**, ending the proof.

The correctness lemma for Π is a corollary of the two previous lemmas:

Lemma 6. *Given regular expressions e and e_k and a PEG G_k, where e_k and G_k are equivalent, then $\Pi(e,\ G_k)$ and $e\,e_k$ are equivalent.*

Proof. The proof that first property of equivalence holds for $\Pi(e,\ G_k)$ and $e\,e_k$ follows from the first property of equivalence for e_k and G_k plus Lemma 4. The proof that the second property of equivalence holds follows from the first property of equivalence for $\Pi(e,\ G_k)$ and $e\,e_k$, the second property of equivalence for e_k and G_k, plus Lemma 5.

A corollary of the previous lemma combined with Lemma 2 is that $L(e\,\$) = L(\Pi(e, \$))$, proving that our transformation can yield a PEG that recognizes the same language as any well-formed regular expression e just by using an end-of-input marker, even if the language of e does not have the prefix property.

We still need to show that any regular expression can be made well-formed without changing its language. This is the topic of the next section, where we give a transformation that rewrites non-well-formed repetitions so they are well-formed with minimal changes to the structure of the original regular expression.

3.1 Transformation of Repetitions e^* where $\varepsilon \in L(e)$

A regular expression e that has a subexpression e_i^* where e_i can match the empty string is not well-formed. As e_i can succeed without consuming any input one outcome of e_i^* is to stay in the same place of the input indefinitely. Pattern matching libraries that rely on backtracking may enter an infinite loop with non-well-formed expressions unless they take measures to avoid it [12].

When e is not well-formed, the PEG we obtain through transformation Π is not complete. A PEG that is not complete can make a PEG library enter an infinite loop. To show an example on how a non-well-formed regular expression leads to a PEG that is not complete, let us transform $(a \,|\, \varepsilon)^*\, b$ using Π. Using ε as continuation this yields the following PEG A:

$$A \;\rightarrow\; a\,A \;\mid\; A \;\mid\; b$$

The PEG above is *left recursive*, so it is not complete. In fact, this PEG does not have a proof tree for any input, so it is not equivalent to the regular expression $(a \,|\, \varepsilon)^*\, b$.

Transformation Π is not correct for non-well-formed regular expressions, but we can make any non-well-formed regular expression well-formed by rewriting repetitions e_i^* where $\varepsilon \in L(e_i)$ as $e_i'^*$ where $\varepsilon \notin L(e_i')$ and $L(e_i'^*) = L(e_i^*)$. The regular expression above would become $a^* \,|\, b$, which Π transforms into an equivalent complete PEG.

This section presents a transformation that does this rewriting. We use a pair of functions to rewrite an expression, f_{out} and f_{in}. Function f_{out} recursively searches for a repetition that has ε in the language of its subexpression, while f_{in} rewrites the repetition's subexpression so it is well-formed, does not have ε in its language, and does not change the language of the repetition. Both f_{in} and f_{out} use two auxiliary predicates, *empty* and *null*, that respectively test if an expression can be reduced to ε and if an expression has ε in its language. Figure 4 has inductive definitions for the *empty* and *null* predicates.

Function f_{out} is simple: for the base expressions it is the identity, for the composite expressions f_{out} applies itself recursively to subexpressions unless the expression is a repetition where the repetition's subexpression matches ε. In this case f_{out} reduces the repetition to ε if the subexpression reduces to ε (as $\varepsilon^* \equiv \varepsilon$), or uses f_{in} to rewrite the subexpression. Figure 5 has the inductive definition of f_{out}. It obeys the following lemma:

$$empty(\varepsilon) = \mathbf{true} \quad empty(a) = \mathbf{false} \quad empty(e^*) = empty(e)$$
$$empty(e_1\, e_2) = empty(e_1) \ \wedge \ empty(e_2) \quad empty(e_1 \mid e_2) = empty(e_1) \ \wedge \ empty(e_2)$$

$$null(\varepsilon) = \mathbf{true} \quad null(a) = \mathbf{false} \quad null(e^*) = \mathbf{true}$$
$$null(e_1\, e_2) = null(e_1) \ \wedge \ null(e_2) \quad null(e_1 \mid e_2) = null(e_1) \ \vee \ null(e_2)$$

Fig. 4. Definition of predicates *empty* and *null*

$$\begin{aligned}
f_{out}(e) &= e, \text{ if } e = \varepsilon \text{ or } e = a \\
f_{out}(e_1\, e_2) &= f_{out}(e_1)\, f_{out}(e_2) \\
f_{out}(e_1 \mid e_2) &= f_{out}(e_1) \ \mid \ f_{out}(e_2) \\
f_{out}(e^*) &= \begin{cases} f_{out}(e)^* & \text{if } \neg null(e) \\ \varepsilon & \text{if } empty(e) \\ f_{in}(e)^* & \text{otherwise} \end{cases}
\end{aligned}$$

Fig. 5. Definition of Function f_{out}

Lemma 7. *If $f_{in}(e_k)$ is well-formed, $\varepsilon \notin L(f_{in}(e_k))$, and $L(f_{in}(e_k)^*) = L(e_k^*)$ for any e_k with $\varepsilon \in L(e_k)$ and $L(e_k) \neq \varepsilon$ then, for any e, $f_{out}(e)$ is well-formed and $L(e) = L(f_{out}(e))$.*

Proof. By structural induction on e. Inductive cases follow directly from the induction hypothesis, except for e^* where $\varepsilon \in L(e)$, where it follows from the properties of f_{in}.

Function f_{in} does the heavy lifting of the rewriting. If its argument is a repetition it throws away the repetition because it is superfluous. Then f_{in} applies f_{out} or itself to the subexpression depending on whether it matches ε or not. If the argument is a choice f_{in} throws away one of the sides if it reduces to ε, as it is superfluous because of the repetition, and rewrites the remaining side using f_{out} or f_{in} depending on whether it matches ε or not. In case both sides do not reduce to ε f_{in} rewrites both. If the argument is a concatenation f_{in} rewrites it as a choice and applies itself to the choice. This works because both subexpressions need to match ε for f_{in}'s argument to match ε, and $(AB)^* = (A \cup B)^*$ if $\varepsilon \in A$ and $\varepsilon \in B$. Figure 6 has the inductive definition of f_{in}. It obeys the following lemma:

Lemma 8. *If $f_{out}(e_k)$ is well-formed and $L(f_{out}(e_k)) = L(e_k)$ for any e_k then, for any e with $\varepsilon \in L(e)$ and $L(e) \neq \{\varepsilon\}$, $\varepsilon \notin L(f_{in}(e))$, $L(e^*) = L(f_{in}(e)^*)$, and $f_{in}(e)$ is well-formed.*

Proof. By structural induction on e. Most cases follow directly from the induction hypothesis and the properties of f_{out}. The subcases of choice where the

$$
\begin{aligned}
f_{in}(e_1\, e_2) &= f_{in}(e_1 \mid e_2) \\
f_{in}(e_1 \mid e_2) &= \begin{cases}
f_{in}(e_2) & \text{if } \mathit{empty}(e_1) \text{ and } \mathit{null}(e_2) \\
f_{out}(e_2) & \text{if } \mathit{empty}(e_1) \text{ and } \neg\mathit{null}(e_2) \\
f_{in}(e_1) & \text{if } \mathit{null}(e_1) \text{ and } \mathit{empty}(e_2) \\
f_{out}(e_1) & \text{if } \neg\mathit{null}(e_1) \text{ and } \mathit{empty}(e_2) \\
f_{out}(e_1) \mid f_{in}(e_2) & \text{if } \neg\mathit{null}(e_1) \text{ and } \neg\mathit{empty}(e_2) \\
f_{in}(e_1) \mid f_{out}(e_2) & \text{if } \neg\mathit{empty}(e_1) \text{ and } \neg\mathit{null}(e_2) \\
f_{in}(e_1) \mid f_{in}(e_2) & \text{otherwise}
\end{cases} \\
f_{in}(e^*) &= \begin{cases}
f_{in}(e) & \text{if } \mathit{null}(e) \\
f_{out}(e) & \text{otherwise}
\end{cases}
\end{aligned}
$$

Fig. 6. Definition of Function $f_{in}(e)$, where $\neg\mathit{empty}(e)$ and $\mathit{null}(e)$

$$
\begin{aligned}
\Pi(?\rangle e_1, G_k) &= (V_1, T, P_1, p_1\, p_k), \text{ where } (V_1, T, P_1, p_1) = \Pi(e_1, G_k[\varepsilon]) \\
\Pi(e_1^{*+}, G_k) &= \Pi(?\rangle e^*, G_k) \\
\Pi(e_1^{*?}, G_k) &= G \text{ , where } G = (V_1, T, P_1 \cup \{A \to p_k \mid p_1\}, A) \text{ ,} \\
&\quad (V_1, T, P_1, p_1) = \Pi(e_1, (V_k \cup \{A\}, T, P_k, A)), \text{ and } A \notin V_k \\
\Pi(?!e_1, G_k) &= (V_1, T, P_1, !p_1\, p_k), \text{ where } (V_1, T, P_1, p_1) = \Pi(e_1, G_k[\varepsilon]) \\
\Pi(?{=}e_1, G_k) &= (V_1, T, P_1, !!p_1\, p_k), \text{ where } (V_1, T, P_1, p_1) = \Pi(e_1, G_k[\varepsilon])
\end{aligned}
$$

Fig. 7. Adapting Function Π to Deal with Regex Extensions

result is also a choice use the Kleene star property $(A \cup B)^* = (A^* \cup B^*)^*$ together with the induction hypothesis and the properties of f_{out}. Concatenation reduces to a choice using the property mentioned above this lemma.

As an example, let us use f_{out} and f_{in} to rewrite the regular expression $((a \mid \varepsilon)\, b^*)^*$ into a well-formed regular expression. We show the sequence of steps below:

$$
\begin{aligned}
f_{out}(((a \mid \varepsilon)\, b^*)^*) &= f_{in}((a \mid \varepsilon)\, b^*)^* = f_{in}((a \mid \varepsilon) \mid b^*)^* \\
&= (f_{in}(a \mid \varepsilon) \mid f_{in}(b^*))^* = (f_{out}(a) \mid f_{out}(b))^* = (a \mid b)^*
\end{aligned}
$$

4 Transforming Regex Extensions

Regexes add several ad-hoc extensions to regular expressions. We can easily adapt transformation Π to deal with some of these extensions, and will show here how to use Π with independent expressions, possessive repetitions, lazy repetitions, and lookahead. An informal but broader discussion of regex extensions in the context of translation to PEGs was published by Oikawa et al. [11].

The regex $?\rangle e_1$ is an independent expression (also known as atomic grouping). It matches independently of the expression that follows it, so a failure when

matching the expression that follows $?\rangle e_1$ does not force a backtracking regex matcher to backtrack to $?\rangle e_1$'s alternative matches. This is the same behavior as a PEG, so to transform $?\rangle e_1$ we first transform it using an empty continuation, then concatenate the result with the original continuation.

The regex e_1^{*+} is a possessive (or greedy) repetition. It always matches as most as possible of the input, even if this leads to a subsequent failure. It is the same as $?\rangle e^*$ if the longest-match rule is used. The semantics of Π guarantees longest match, so it uses this identity to transform e_1^{*+}.

The regex $e_1^{*?}$ is a lazy repetition. It always matches as little of the input as necessary for the rest of the expression to match (shortest match). The transformation of this regex is very similar to the transformation of e_1^*, we just flip p_1 and p_k in the production of non-terminal A. Now the PEG tries to match the rest of the expression first, and will only try another step of the repetition if the rest fails.

The regex $?!e_1$ is a negative lookahead. The regex matcher tries to match the subexpression; it it fails then the negative lookahead succeeds without consuming any input, and if the subexpression succeeds the negative lookahead fails. Negative lookahead is also an independent expression. Transforming this regex is just a matter of using PEGs negative lookahead, which works in the same way, on the result of transforming the subexpression as an independent expression.

Finally, the regex $?= e_1$ is a positive lookahead, where the regex matcher tries to match the subexpression and fails if the subexpression fails and succeeds if the subexpression succeeds, but does not consume any input. It is also an independent expression. We transform a positive lookahead by transforming the subexpression as an independent expression and then using PEGs negative lookahead twice.

We cannot provide formal proofs that the extended transformation Π correctly captures the ad-hoc semantics of these regex extensions, as their behavior cannot be expressed using the semantics of regular expressions. But we believe that the transformation and the semantics of PEGs is a correct formalization of these regex extensions.

5 Conclusion

We presented a new formalization of regular expressions that uses natural semantics and a transformation Π that converts a given regular expression into an equivalent PEG. If the regular expression's language has the prefix property, easily guaranteed by using an end-of-input marker, the transformation yields a PEG that recognizes the same language as the regular expression.

Moreover, we have shown how this transformation can be easily adapted to accommodate several extensions used by pattern matching libraries: independent expressions, possessive and lazy repetition, and lookahead. Our transformation gives a precise semantics to what were ad-hoc extensions with behavior specified in terms of how regex matchers are implemented.

Another approach to establish the correspondence between regular expressions and PEGs was suggested by Ierusalimschy [7]. In this approach we convert Deterministic Finite Automata (DFA) into right-linear LL(1) grammars. Medeiros [9] proves that an LL(1) grammar has the same language when interpreted as a CFG and as a PEG. But this approach cannot be used with regex extensions, as they cannot be expressed by a DFA.

The transformation Π is a formalization of the continuation-based conversion presented by Oikawa et al. [11]. That work only presents an informal discussion of the correctness of the conversion, while we proved our transformation correct with regards to the semantics of regular expressions and PEGs.

We can also benefit from the LPEG parsing machine [10, 7], a virtual machine for executing PEGs. We can use the cost model of the parsing machine instructions to estimate how efficient a given regular expression or regex is. The parsing machine has a simple architecture with just nine basic instructions and four registers, and implementations of our transformation coupled with implementations of the parsing machine can be the basis for simpler implementations of regex libraries.

References

1. Abigail: Reduction of 3-CNF-SAT to Perl regular expression matching. `http://perl.plover.com/NPC/NPC-3SAT.html`
2. Aho, A.V.: Algorithms for finding patterns in strings, pp. 255–300. MIT Press, Cambridge, MA, USA (1990), `http://portal.acm.org/citation.cfm?id=114872.114877`
3. Ford, B.: Parsing expression grammars: A recognition-based syntactic foundation. In: POPL '04: Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages. pp. 111–122. ACM, New York, USA (2004)
4. Fowler, G.: An interpretation of the POSIX regex standard. `http://www2.research.att.com/~gsf/testregex/re-interpretation.html` (2003), [Visited on May 2011]
5. Friedl, J.: Mastering Regular Expressions. O'Reilly Media, Inc. (2006)
6. Hopcroft, J., Ullman, J.: Introduction to Automata Theory, Languages, and Computation. Addison-Wesley Longman Publishing Co., Inc., Boston, USA (1979)
7. Ierusalimschy, R.: A text pattern-matching tool based on parsing expression grammars. Software - Practice & Experience 39(3), 221–258 (2009)
8. Kahn, G.: Natural semantics. In: STACS '87: Symposium on Theoretical Aspects of Computer Science. pp. 22–39. Springer-Verlag, Londres, Reino Unido (1987)
9. Medeiros, S.: Correspondência entre PEGs e Classes de Gramáticas Livres de Contexto. Ph.D. thesis, PUC-Rio (2010)
10. Medeiros, S., Ierusalimschy, R.: A parsing machine for PEGs. In: DLS '08: Dynamic languages Symposium. pp. 1–12. ACM, New York, USA (2008)
11. Oikawa, M., Ierusalimschy, R., Moura, A.: Converting regexes to Parsing Expression Grammars. In: Brazilian Symposium on Programming Languages (2010)
12. perldoc.perl.org — official documentation for the Perl programming language. `http://perldoc.perl.org/perlre.html`, [Visited on May 2011]