

# Left Recursion in Parsing Expression Grammars

Sérgio Medeiros<sup>1</sup>, Fabio Mascarenhas<sup>2</sup>, Roberto Ierusalimsky<sup>3</sup>

<sup>1</sup> Department of Computer Science – UFS – Aracaju – Brazil  
`sergio@ufs.br`

<sup>2</sup> Department of Computer Science – UFRJ – Rio de Janeiro – Brazil  
`fabiom@dcc.ufrj.br`

<sup>3</sup> Department of Computer Science – PUC-Rio – Rio de Janeiro – Brazil  
`roberto@inf.puc-rio.br`

**Abstract.** Parsing Expression Grammars (PEGs) are a formalism that can describe all deterministic context-free languages through a set of rules that specify a top-down parser for some language. PEGs are easy to use, and there are efficient implementations of PEG libraries in several programming languages.

A frequently missed feature of PEGs is left recursion, which is commonly used in Context-Free Grammars (CFGs) to encode left-associative operations. We present a simple conservative extension to the semantics of PEGs that gives useful meaning to direct and indirect left-recursive rules, and show that our extensions make it easy to express left-recursive idioms from CFGs in PEGs, with similar results. We prove the conservativeness of these extensions, and also prove that they work with any left-recursive PEG.

**Keywords:** parsing expression grammars, parsing, left recursion, natural semantics, packrat parsing

## 1 Introduction

Parsing Expression Grammars (PEGs) [3] are a formalism for describing a language’s syntax, and an alternative to the commonly used Context Free Grammars (CFGs). Unlike CFGs, PEGs are unambiguous by construction, and their standard semantics is based on recognizing instead of deriving strings. Furthermore, a PEG can be considered both the specification of a language and the specification of a top-down parser for that language.

PEGs use the notion of *limited backtracking*: the parser, when faced with several alternatives, tries them in a deterministic order (left to right), discarding remaining alternatives after one of them succeeds. They also have an expressive syntax, based on the syntax of extended regular expressions, and *syntactic predicates*, a form of unrestricted lookahead where the parser checks whether the rest of the input matches a parsing expression without consuming the input.

The top-down parsing approach of PEGs means that they cannot handle left recursion in grammar rules, as they would make the parser loop forever. Left recursion can be detected structurally, so PEGs with left-recursive rules can be

simply rejected by PEG implementations instead of leading to parsers that do not terminate, but the lack of support for left recursion is a restriction on the expressiveness of PEGs. The use of left recursion is a common idiom for expressing language constructs in a grammar, and is present in published grammars for programming languages; the use of left recursion can make rewriting an existing grammar as a PEG a difficult task [17].

There are proposals for adding support for left recursion to PEGs, but they either assume a particular PEG implementation approach, *packrat parsing* [23], or support just direct left recursion [21]. Packrat parsing [2] is an optimization of PEGs that uses memoization to guarantee linear time behavior in the presence of backtracking and syntactic predicates, but can be slower in practice [18, 14]. Packrat parsing is a common implementation approach for PEGs, but there are others [12]. Indirect left recursion is present in real grammars, and is difficult to untangle [17].

In this paper, we present a novel operational semantics for PEGs that gives a well-defined and useful meaning for PEGs with left-recursive rules. The semantics is given as a conservative extension of the existing semantics, so PEGs that do not have left-recursive rules continue having the same meaning as they had. It is also implementation agnostic, and should be easily implementable on packrat implementations, plain recursive descent implementations, and implementations based on a parsing machine.

We also introduce *parse strings* as a possible semantic value resulting from a PEG parsing some input, in parallel to the parse trees of context-free grammars. We show that the parse strings that left-recursive PEGs yield for the common left-recursive grammar idioms are similar to the parse trees we get from bottom-up parsers and left-recursive CFGs, so the use of left-recursive rules in PEGs with out semantics should be intuitive for grammar writers.

The rest of this paper is organized as follows: Section 2 presents a brief introduction to PEGs and discusses the problem of left recursion in PEGs; Section 3 presents our semantic extensions for PEGs with left-recursive rules; Section 4 reviews some related work on PEGs and left recursion in more detail; finally, Section 5 presents our concluding remarks.

## 2 Parsing Expression Grammars and Left Recursion

Parsing Expression Grammars borrow the use of non-terminals and rules (or productions) to express context-free recursion, although all non-terminals in a PEG must have only one rule. The syntax of the right side of the rules, the *parsing expressions*, is borrowed from regular expressions and its extensions, in order to make it easier to build parsers that parse directly from characters instead of tokens from a previous lexical analysis step. The semantics of PEGs come from backtracking top-down parsers, but in PEGs the backtracking is local to each choice point.

Our presentation of PEGs is slightly different from Ford's [3], and comes from earlier work [12, 13]. This style makes the exposition of our extensions, and their

$$\begin{array}{c}
\text{Empty String} \quad \frac{}{G[\varepsilon] \ x \xrightarrow{\text{PEG}} (x, \varepsilon)} \text{ (empty.1)} \quad \text{Terminal} \quad \frac{}{G[a] \ ax \xrightarrow{\text{PEG}} (x, a)} \text{ (char.1)} \\
\frac{}{G[b] \ ax \xrightarrow{\text{PEG}} \text{fail}} , b \neq a \text{ (char.2)} \quad \frac{}{G[a] \ \varepsilon \xrightarrow{\text{PEG}} \text{fail}} \text{ (char.3)} \\
\text{Variable} \quad \frac{G[P(A)] \ xy \xrightarrow{\text{PEG}} (y, x')}{G[A] \ xy \xrightarrow{\text{PEG}} (y, A[x'])} \text{ (var.1)} \quad \frac{G[P(A)] \ x \xrightarrow{\text{PEG}} \text{fail}}{G[A] \ x \xrightarrow{\text{PEG}} \text{fail}} \text{ (var.2)} \\
\text{Concatenation} \quad \frac{G[p_1] \ xyz \xrightarrow{\text{PEG}} (yz, x') \quad G[p_2] \ yz \xrightarrow{\text{PEG}} (z, y')}{G[p_1 p_2] \ xyz \xrightarrow{\text{PEG}} (z, x'y')} \text{ (con.1)} \\
\frac{G[p_1] \ xy \xrightarrow{\text{PEG}} (y, x') \quad G[p_2] \ y \xrightarrow{\text{PEG}} \text{fail}}{G[p_1 p_2] \ xy \xrightarrow{\text{PEG}} \text{fail}} \text{ (con.2)} \quad \frac{G[p_1] \ x \xrightarrow{\text{PEG}} \text{fail}}{G[p_1 p_2] \ x \xrightarrow{\text{PEG}} \text{fail}} \text{ (con.3)} \\
\text{Choice} \quad \frac{G[p_1] \ xy \xrightarrow{\text{PEG}} (y, x')}{G[p_1 / p_2] \ xy \xrightarrow{\text{PEG}} (y, x')} \text{ (ord.1)} \quad \frac{G[p_1] \ x \xrightarrow{\text{PEG}} \text{fail} \quad G[p_2] \ x \xrightarrow{\text{PEG}} \text{fail}}{G[p_1 / p_2] \ x \xrightarrow{\text{PEG}} \text{fail}} \text{ (ord.2)} \\
\frac{G[p_1] \ xy \xrightarrow{\text{PEG}} \text{fail} \quad G[p_2] \ xy \xrightarrow{\text{PEG}} (y, x')}{G[p_1 / p_2] \ xy \xrightarrow{\text{PEG}} (y, x')} \text{ (ord.3)} \\
\text{Not Predicate} \quad \frac{G[p] \ x \xrightarrow{\text{PEG}} \text{fail}}{G[!p] \ x \xrightarrow{\text{PEG}} (x, \varepsilon)} \text{ (not.1)} \quad \frac{G[p] \ xy \xrightarrow{\text{PEG}} (y, x')}{G[!p] \ xy \xrightarrow{\text{PEG}} \text{fail}} \text{ (not.2)} \\
\text{Repetition} \quad \frac{G[p] \ x \xrightarrow{\text{PEG}} \text{fail}}{G[p^*] \ x \xrightarrow{\text{PEG}} (x, \varepsilon)} \text{ (rep.1)} \quad \frac{G[p] \ xyz \xrightarrow{\text{PEG}} (yz, x') \quad G[p^*] \ yz \xrightarrow{\text{PEG}} (z, y')}{G[p^*] \ xyz \xrightarrow{\text{PEG}} (z, x'y')} \text{ (rep.2)}
\end{array}$$

**Fig. 1.** Semantics of the  $\xrightarrow{\text{PEG}}$  relation

behavior, easier to understand. We define a PEG  $G$  as a tuple  $(V, T, P, p_S)$  where  $V$  is the finite set of non-terminals,  $T$  is the alphabet (finite set of terminals),  $P$  is a function from  $V$  to parsing expressions, and  $p_S$  is the *starting expression*, the one that the PEG matches. Function  $P$  is commonly described through a set of rules of the form  $A \leftarrow p$ , where  $A \in V$  and  $p$  is a parsing expression.

Parsing expressions are the core of our formalism, and they are defined inductively as the empty expression  $\varepsilon$ , a terminal symbol  $a$ , a non-terminal symbol  $A$ , a concatenation  $p_1 p_2$  of two parsing expressions  $p_1$  and  $p_2$ , an ordered choice  $p_1 / p_2$  between two parsing expressions  $p_1$  and  $p_2$ , a repetition  $p^*$  of a parsing expression  $p$ , or a not-predicate  $!p$  of a parsing expression  $p$ . We leave out extensions such as the dot, character classes, strings, and the and-predicate, as their addition is straightforward.

We define the semantics of PEGs via a relation  $\overset{\text{PEG}}{\rightsquigarrow}$  among a PEG, a parsing expression, a subject, and a result. The notation  $G[p] \ xy \overset{\text{PEG}}{\rightsquigarrow} (y, x')$  means that the expression  $p$  matches the input  $xy$ , consuming the prefix  $x$ , while leaving  $y$  and yielding a *parse string*  $x'$  as the output, while resolving any non-terminals using the rules of  $G$ . We use  $G[p] \ xy \overset{\text{PEG}}{\rightsquigarrow} \mathbf{fail}$  to express an unsuccessful match. The language of a PEG  $G$  is defined as all strings that  $G$ 's starting expression consumes, that is, the set  $\{x \in T^* \mid G[p_s] \ xy \overset{\text{PEG}}{\rightsquigarrow} (y, x')\}$ .

Figure 1 presents the definition of  $\overset{\text{PEG}}{\rightsquigarrow}$  using natural semantics [11, 25], as a set of inference rules. Intuitively,  $\varepsilon$  just succeeds and leaves the subject unaffected;  $a$  matches and consumes itself, or fails;  $A$  tries to match the expression  $P(A)$ ;  $p_1p_2$  tries to match  $p_1$ , and if it succeeds tries to match  $p_2$  on the part of the subject that  $p_1$  did not consume;  $p_1/p_2$  tries to match  $p_1$ , and if it fails tries to match  $p_2$ ;  $p^*$  repeatedly tries to match  $p$  until it fails, thus consuming as much of the subject as it can; finally,  $!p$  tries to match  $p$  and fails if  $p$  succeeds and succeeds if  $p$  fails, in any case leaving the subject unaffected. It is easy to see that the result of a match is either failure or a suffix of the subject (not a proper suffix, as the expression may succeed without consuming anything).

Context-Free Grammars have the notion of a *parse tree*, a graphical representation of the structure that a valid subject has, according to the grammar. The proof trees of our semantics can have a similar role, but they have extra information that can obscure the desired structure. This problem will be exacerbated in the proof trees that our rules for left-recursion yield, and is the reason we introduce parse strings to our formalism. A parse string is roughly a linearization of a parse tree, and shows which non-terminals have been used in the process of matching a given subject. Having the result of a parse be an actual tree and having arbitrary semantic actions are straightforward extensions.

When using PEGs for parsing it is important to guarantee that a given grammar will either yield a successful result or **fail** for every subject, so parsing always terminates. Grammars where this is true are *complete* [3]. In order to guarantee completeness, it is sufficient to check for the absence of direct or indirect *left recursion*, a property that can be checked structurally using the *well-formed* predicate from Ford [3] (abbreviated *WF*).

Inductively, empty expressions and symbol expressions are always well-formed; a non-terminal is well-formed if it has a production and it is well-formed; a choice is well-formed if the alternatives are well-formed; a not predicate is well-formed if the expression it uses is well-formed; a repetition is well-formed if the expression it repeats is well-formed and cannot succeed without consuming input; finally, a concatenation is well-formed if either its first expression is well-formed and cannot succeed without consuming input or both of its expressions are well-formed.

A grammar is well-formed if its non-terminals and starting expression are all well-formed. The test of whether an expression cannot succeed while not consuming input is also computable from the structure of the expression and its grammar from an inductive definition [3]. The rule for well-formedness of repetitions just derives from writing a repetition  $p^*$  as a recursion  $A \leftarrow pA / \varepsilon$ , so a non-well-formed repetition is just a special case of a left-recursive rule.

Left recursion is not a problem in the popular bottom-up parsing approaches, and is a natural way to express several common parsing idioms. Expressing repetition using left recursion in a CFG yields a left-associative parse tree, which is often desirable when parsing programming languages, either because operations have to be left-associative or because left-associativity is more efficient in bottom-up parsers [6]. For example, the following is a simple left-associative CFG for additive expressions, written in EBNF notation:

$$\begin{aligned} E &\rightarrow E + T \mid E - T \mid T \\ T &\rightarrow n \mid (E) \end{aligned}$$

Rewriting the above grammar as a PEG, by replacing  $\mid$  with the ordered choice operator, yields a non-well-formed PEG that does not have a proof tree for any subject. We can rewrite the grammar to eliminate the left recursion, giving the following CFG, again in EBNF (the curly brackets are metasymbols of EBNF notation, and express zero-or-more repetition, while the parentheses are terminals):

$$\begin{aligned} E &\rightarrow T\{E'\} \\ T &\rightarrow n \mid (E) \\ E' &\rightarrow +T \mid -T \end{aligned}$$

This is a simple transformation, but it yields a different parse tree, and obscures the intentions of the grammar writer, even though it is possible to transform the parse tree of the non-left-recursive grammar into the left-associative parse tree of the left-recursive grammar. But at least we can straightforwardly express the non-left-recursive grammar with the following PEG:

$$\begin{aligned} E &\leftarrow T E'^* \\ T &\leftarrow n / (E) \\ E' &\leftarrow +T / -T \end{aligned}$$

Indirect left recursion is harder to eliminate, and its elimination changes the structure of the grammar and the resulting trees even more. For example, the following indirectly left-recursive CFG denotes a very simplified grammar for l-values in a language with variables, first-class functions, and records (where  $x$  stands for identifiers and  $n$  for expressions):

$$\begin{aligned} L &\rightarrow P.x \mid x \\ P &\rightarrow P(n) \mid L \end{aligned}$$

This grammar generates  $x$  and  $x$  followed by any number of  $(n)$  or  $.x$ , as long as it ends with  $.x$ . An l-value is a prefix expression followed by a field access, or a single variable, and a prefix expression is a prefix expression followed by an operand, denoting a function call, or a valid l-value. In the parse trees for this grammar each  $(n)$  or  $.x$  associates to the left.

Writing a PEG that parses the same language is difficult. We can eliminate the indirect left recursion on  $L$  by substitution inside  $P$ , getting  $P \rightarrow P(n) \mid P.x \mid x$ , and then eliminate the direct left recursion on  $P$  to get the following CFG:

$$\begin{aligned} L &\rightarrow P.x \mid x \\ P &\rightarrow x\{P'\} \\ P' &\rightarrow (n) \mid .x \end{aligned}$$

But a direct translation of this CFG to a PEG will not work because PEG repetition is greedy; the repetition on  $P'$  will consume the last  $.x$  of the l-value, and the first alternative of  $L$  will always fail. One possible solution is to not use the  $P$  non-terminal in  $L$ , and encode l-values directly with the following PEG (the bolded parentheses are terminals, the non-bolded parentheses are metasymbols of PEGs that mean grouping):

$$\begin{aligned} L &\leftarrow x S^* \\ S &\leftarrow ( (n) )^* .x \end{aligned}$$

The above uses of left recursion are common in published grammars, with more complex versions (involving more rules and a deeper level of indirection) appearing in the grammars in the specifications of Java [5] and Lua [10]. Having a straightforward way of expressing these in a PEG would make the process of translating a grammar specification from an EBNF CFG to a PEG easier and less error-prone.

In the next session we will propose a semantic extension to the PEG formalism that will give meaningful proof trees to left-recursive grammars. In particular, we want to have the straightforward translation of common left-recursive idioms such as left-associative expressions to yield parse strings that are similar in structure to parse trees of the original CFGs.

### 3 Bounded Left Recursion

Intuitively, *bounded left recursion* is a use of a non-terminal where we limit the number of left-recursive uses it may have. This is the basis of our extension for supporting left recursion in PEGs. We use the notation  $A^n$  to mean a non-terminal where we can have less than  $n$  left-recursive uses, with  $A^0$  being an expression that always fails. Any left-recursive use of  $A^n$  will use  $A^{n-1}$ , any left-recursive use of  $A^{n-1}$  will use  $A^{n-2}$ , and so on, with  $A^1$  using  $A^0$  for any left-recursive use, so left recursion will fail for  $A^1$ .

Subject	$E^0$	$E^1$	$E^2$	$E^3$	$E^4$	$E^5$	$E^6$
<b>n</b>	fail	$\varepsilon$	$\varepsilon$	$\varepsilon$	$\varepsilon$	$\varepsilon$	$\varepsilon$
<b>n+n</b>	fail	<b>+n</b>	$\varepsilon$	<b>+n</b>	$\varepsilon$	<b>+n</b>	$\varepsilon$
<b>n+n+n</b>	fail	<b>+n+n</b>	<b>+n</b>	$\varepsilon$	<b>+n+n</b>	<b>+n</b>	$\varepsilon$

**Table 1.** Matching  $E$  with different bounds

For the left-recursive definition  $E \leftarrow E + n / n$  we have the following progression, where we write expressions equivalent to  $E^n$  on the right side:

$$\begin{aligned}
E^0 &\leftarrow \text{fail} \\
E^1 &\leftarrow E^0 + n / n = \perp + n / n = n \\
E^2 &\leftarrow E^1 + n / n = n + n / n \\
E^3 &\leftarrow E^2 + n / n = (n + n / n) + n / n \\
&\vdots \\
E^n &\leftarrow E^{n-1} + n / n
\end{aligned}$$

It would be natural to expect that increasing the bound will eventually reach a fixed point with respect to a given subject, but the behavior of the ordered choice operator breaks this expectation. For example, with a subject **n+n** and the previous PEG,  $E^2$  will match the whole subject, while  $E^3$  will match just the first **n**. Table 1 summarizes the results of trying to match some subjects against  $E$  with different left-recursive bounds (they show the suffix that remains, not the matched prefix).

The fact that increasing the bound can lead to matching a smaller prefix means we have to pick the bound carefully if we wish to match as much of the subject as possible. Fortunately, it is sufficient to increase the bound until the size of the matched prefix stops increasing. In the above example, we would pick 1 as the bound for **n**, 2 as the bound for **n+n**, and 3 as the bound for **n+n+n**.

When the bound of a non-terminal  $A$  is 1 we are effectively prohibiting a match via any left-recursive path, as all left-recursive uses of  $A$  will fail.  $A^{n+1}$  uses  $A^n$  on all its left-recursive paths, so if  $A^n$  matches a prefix of length  $k$ ,  $A^{n+1}$  matching a prefix of length  $k$  or less means that either there is nothing to do after matching  $A^n$  (the grammar is cyclic), in which case it is pointless to increase the bound after  $A^n$ , or all paths starting with  $A^n$  failed, and the match actually used a non-left-recursive path, so  $A^{n+1}$  is equivalent with  $A^1$ . Either option means that  $n$  is the bound that makes  $A$  match the longest prefix of the subject.

We can easily see this dynamic in the  $E \leftarrow E + n / n$  example. To match  $E^{n+1}$  we have to match  $E^n + n / n$ . Assume  $E^n$  matches a prefix  $x$  of the input. We then try to match the rest of the input with  $+n$ , if this succeeds we will have

### Left-Recursive Variable

$$\frac{(A, xyz) \notin \mathcal{L} \quad G[P(A)] \quad xyz \quad \mathcal{L}[(A, xyz) \mapsto \text{fail}] \xrightarrow{\text{PEG}} (yz, x') \quad G[P(A)] \quad xyz \quad \mathcal{L}[(A, xyz) \mapsto (yz, x')] \xrightarrow{\text{INC}} (z, (xy)')}{G[A] \quad xyz \quad \mathcal{L} \xrightarrow{\text{PEG}} (z, A[(xy)'])} \quad (\text{lvar.1})$$

$$\frac{(A, x) \notin \mathcal{L} \quad G[P(A)] \quad x \quad \mathcal{L}[(A, x) \mapsto \text{fail}] \xrightarrow{\text{PEG}} \text{fail}}{G[A] \quad x \quad \mathcal{L} \xrightarrow{\text{PEG}} \text{fail}} \quad (\text{lvar.2})$$

$$\frac{\mathcal{L}(A, xy) = \text{fail}}{G[A] \quad xy \quad \mathcal{L} \xrightarrow{\text{PEG}} \text{fail}} \quad (\text{lvar.3}) \quad \frac{\mathcal{L}(A, xy) = (y, x')}{G[A] \quad xy \quad \mathcal{L} \xrightarrow{\text{PEG}} (y, A[x'])} \quad (\text{lvar.4})$$

### Increase Bound

$$\frac{G[P(A)] \quad xyzw \quad \mathcal{L}[(A, xyzw) \mapsto (yzw, x')] \xrightarrow{\text{PEG}} (zw, (xy)') \quad G[P(A)] \quad xyzw \quad \mathcal{L}[(A, xyzw) \mapsto (zw, (xy)')] \xrightarrow{\text{INC}} (w, (xyz)')}{G[P(A)] \quad xyzw \quad \mathcal{L}[(A, xyzw) \mapsto (yzw, x')] \xrightarrow{\text{INC}} (w, (xyz)')}, \text{ where } y \neq \varepsilon \quad (\text{inc.1})$$

$$\frac{G[P(A)] \quad x \quad \mathcal{L} \xrightarrow{\text{PEG}} \text{fail}}{G[P(A)] \quad x \quad \mathcal{L} \xrightarrow{\text{INC}} \mathcal{L}(A, x)} \quad (\text{inc.2}) \quad \frac{G[P(A)] \quad xyz \quad \mathcal{L}[(A, xyz) \mapsto (z, (xy)')] \xrightarrow{\text{PEG}} (yz, x')}{G[P(A)] \quad xyz \quad \mathcal{L}[(A, xyz) \mapsto (z, (xy)')] \xrightarrow{\text{INC}} (z, (xy)')} \quad (\text{inc.3})$$

**Fig. 2.** Semantics for PEGs with left-recursive non-terminals

matched  $x+n$ , a prefix bigger than  $x$ . If this fails we will have matched just  $n$ , which is the same prefix matched by  $E^1$ .

Indirect, and even mutual, left recursion is not a problem, as the bounds are on left-recursive *uses* of a non-terminal, which are a property of the proof tree, and not of the structure of the PEG. The bounds on two mutually recursive non-terminals  $A$  and  $B$  will depend on which non-terminal is being matched first, if it is  $A$  then the bound of  $A$  is fixed while varying the bound of  $B$ , and vice-versa. A particular case of mutual left recursion is when a non-terminal is both left and right-recursive, such as  $E \leftarrow E + E/n$ . In our semantics,  $E^n$  will match  $E^{n-1} + E/n$ , where the right-recursive use of  $E$  will have its own bound. Later in this section we will elaborate on the behavior of both kinds of mutual recursion.

In order to extend the semantics of PEGs with bounded left recursion, we will show a conservative extension of the rules in Figure 1, with new rules for left-recursive non-terminals. For non-left-recursive non-terminals we will still use rules **var.1** and **var.2**, although we will later prove that this is unnecessary, and the new rules for non-terminals can replace the current ones. The basic idea of the extension is to use  $A^1$  when matching a left-recursive non-terminal  $A$  for the first time, and then try to increase the bound, while using a memoization table  $\mathcal{L}$  to keep the result of the current bound. We use a different relation, with its own inference rules, for this iterative process of increasing the bound.



Figure 2 presents the new rules. We give the behavior of the memoization table  $\mathcal{L}$  in the usual substitution style, where  $\mathcal{L}[(A, x) \mapsto X](B, y) = \mathcal{L}(B, y)$  if  $B \neq A$  or  $y \neq x$  and  $\mathcal{L}[(A, x) \mapsto X](A, x) = X$  otherwise. All of the rules in Figure 1 just ignore this extra parameter of relation  $\overset{\text{PEG}}{\rightsquigarrow}$ . We also have rules for the new relation  $\overset{\text{INC}}{\rightsquigarrow}$ , responsible for the iterative process of finding the correct bound for a given left-recursive use of a non-terminal.

Rules **lvar.1** and **lvar.2** apply the first time a left-recursive non-terminal is used with a given subject, and they try to match  $A^1$  by trying to match the production of  $A$  using **fail** for any left-recursive use of  $A$  (those uses will fail through rule **lvar.3**). If  $A^1$  fails we do not try bigger bounds (rule **lvar.2**), but if  $A^1$  succeeds we store the result in  $\mathcal{L}$  and try to find a bigger bound (rule **lvar.1**). Rule **lvar.4** is used for left-recursive invocations of  $A^n$  in the process of matching  $A^{n+1}$ .

Relation  $\overset{\text{INC}}{\rightsquigarrow}$  tries to find the bound where  $A$  matches the longest prefix. Which rule applies depends on whether matching the production of  $A$  using the memoized value for the current bound leads to a longer match or not; rule **inc.1** covers the first case, where we use relation  $\overset{\text{INC}}{\rightsquigarrow}$  again to continue increasing the bound after updating  $\mathcal{L}$ . Rules **inc.2** and **inc.3** cover the second case, where the current bound is the correct one and we just return its result.

Let us walk through an example, again using  $E \leftarrow E + n / n$  as our PEG, with  $\mathbf{n+n+n}$  as the subject. When first matching  $E$  against  $\mathbf{n+n+n}$  we have  $(E, \mathbf{n+n+n}) \notin \mathcal{L}$ , as  $\mathcal{L}$  is initially empty, so we have to match  $E + n / n$  against  $\mathbf{n+n+n}$  with  $\mathcal{L} = \{(E, \mathbf{n+n+n}) \mapsto \text{fail}\}$ . We now have to match  $E + n$  against  $\mathbf{n+n+n}$ , which means matching  $E$  again, but now we use rule **lvar.3**. The first alternative,  $E + n$ , fails, and we have  $G[E + n / n] \mathbf{n+n+n} \{(E, \mathbf{n+n+n}) \mapsto \text{fail}\} \overset{\text{PEG}}{\rightsquigarrow} (+\mathbf{n+n}, \mathbf{n})$  using the second alternative,  $n$ , and rule **ord.3**.

In order to finish rule **lvar.1** and the initial match we have to try to increase the bound through relation  $\overset{\text{INC}}{\rightsquigarrow}$  with  $\mathcal{L} = \{(E, \mathbf{n+n+n}) \mapsto (+\mathbf{n+n}, \mathbf{n})\}$ . This means we must try to match  $E + n / n$  against  $\mathbf{n+n+n}$  again, using the new  $\mathcal{L}$ . When we try the first alternative and match  $E$  with  $\mathbf{n+n+n}$  the result will be  $(+\mathbf{n+n}, E[\mathbf{n}])$  via **lvar.4**, and we can then use **con.1** to match  $E + n$  yielding  $(+\mathbf{n}, E[\mathbf{n}]+\mathbf{n})$ . We have successfully increased the bound, and are in rule **inc.1**, with  $x = \mathbf{n}$ ,  $y = +\mathbf{n}$ , and  $zw = +\mathbf{n}$ .

In order to finish rule **inc.1** we have to try to increase the bound again using relation  $\overset{\text{INC}}{\rightsquigarrow}$ , now with  $\mathcal{L} = \{(E, \mathbf{n+n+n}) \mapsto (+\mathbf{n}, E[\mathbf{n}]+\mathbf{n})\}$ . We try to match  $P(E)$  again with this new  $\mathcal{L}$ , and this yields  $(\varepsilon, E[E[\mathbf{n}]+\mathbf{n}]+\mathbf{n})$  via **lvar.4**, **con.1**, and **ord.1**. We have successfully increased the bound and are using rule **inc.1** again, with  $x = \mathbf{n+n}$ ,  $y = +\mathbf{n}$ , and  $zw = \varepsilon$ .

We are in rule **inc.1**, and have to try to increase the bound a third time with  $\overset{\text{INC}}{\rightsquigarrow}$ , with  $\mathcal{L} = \{(E, \mathbf{n+n+n}) \mapsto (\varepsilon, E[E[\mathbf{n}]+\mathbf{n}]+\mathbf{n})\}$ . We have to match  $E + n / n$  against  $\mathbf{n+n+n}$  again, using this  $\mathcal{L}$ . In the first alternative  $E$  matches and yields  $(\varepsilon, E[E[E[\mathbf{n}]+\mathbf{n}]+\mathbf{n}])$  via **lvar.4**, but the first alternative itself fails via **con.2**. We then have to match  $E + n / n$  against  $\mathbf{n+n+n}$  using **ord.2**, yielding  $(+\mathbf{n+n}, \mathbf{n})$ . The attempt to increase the bound for the third time failed (we are back to the same result we had when  $\mathcal{L} = \{(A, \mathbf{n+n+n}) \mapsto \text{fail}\}$ ), and we use rule **inc.3**

once and rule **inc.1** twice to propagate  $(\varepsilon, E[E[n]+n]+n)$  back to rule **lvar.1**, and use this rule to get the final result,  $G[E] \ n + n + n \ \{\} \xrightarrow{\text{PEG}} (\varepsilon, E[E[E[n]+n]+n])$ .

We can see that the parse string  $E[E[E[n]+n]+n]$  implies left-associativity in the  $+$  operations, as intended by the use of a left-recursive rule.

More complex grammars, that encode different precedences and associativities, behave as expected. For example, the following grammar has a right-associative  $+$  with a left-associative  $-$ :

$$\begin{aligned} E &\leftarrow M + E / M \\ M &\leftarrow M - n / n \end{aligned}$$

Matching  $E$  with  $n+n+n$  yields  $E[M[n]+E[M[n]+E[M[n]]]]$ , as matching  $M$  against  $n+n+n$ ,  $n+n$ , and  $n$  all consume just the first  $n$  while generating  $M[n]$ , because  $G[M - n / n] \ n + n + n \ \{(M, n + n + n) \mapsto \text{fail}\} \xrightarrow{\text{PEG}} (+n + n, n)$  via **lvar.3**, **con.3**, and **ord.3**, and  $G[M - n / n] \ n + n + n \ \{(M, n + n + n) \mapsto (+n + n, n)\} \xrightarrow{\text{INC}} (+n + n, n)$  via **inc.3**. The same holds for subjects  $n+n$  and  $n$  with different suffixes. Now, when  $E$  matches  $n+n+n$  we will have  $M$  in  $M + E$  matching the first  $n$ , while  $E$  recursively matching the second  $n+n$ , with  $M$  again matching the first  $n$  and  $E$  recursively matching the last  $n$  via the second alternative.

Matching  $E$  with  $n-n-n$  will yield  $E[M[M[M[n]-n]-n]]$ , as  $M$  now matches  $n-n-n$  with a proof tree similar to our first example ( $E \leftarrow E + n / n$  against  $n+n+n$ ). The first alternative of  $E$  fails because  $M$  consumed the whole subject, and the second alternative yields the final result via **ord.3** and **var.1**.

The semantics of Figure 2 also handles indirect and mutual left recursion well. The following mutually left-recursive PEG is a direct translation of the CFG used as the last example of Section 2:

$$\begin{aligned} L &\leftarrow P.x / x \\ P &\leftarrow P(n) / L \end{aligned}$$

It is instructive to work out what happens when matching  $L$  with a subject such as  $x(n)(n).x(n).x$ . We will use our superscript notation for bounded recursion, but it is easy to check that the explanation corresponds exactly with what is happening with the semantics using  $\mathcal{L}$ .

The first alternative of  $L^1$  will fail because both alternatives of  $P^1$  fail, as they use  $P^0$ , due to the direct left recursion on  $P$ , and  $L^0$ , due to the indirect left recursion on  $L$ . The second alternative of  $L^1$  matches the first  $x$  of the subject. Now  $L^2$  will try to match  $P^1$  again, and the first alternative of  $P^1$  fails because it uses  $P^0$ , while the second alternative uses  $L^1$  and matches the first  $x$ , and so  $P^1$  now matches  $x$ , and we have to try  $P^2$ , which will match  $x(n)$  through the first alternative, now using  $P^1$ .  $P^3$  uses  $P^2$  and matches  $x(n)(n)$  with the first alternative, but  $P^4$  matches just  $x$  again, so  $P^3$  is the answer, and  $L^2$  matches  $x(n)(n).x$  via its first alternative.

$L^3$  will try to match  $P^1$  again, but  $P^1$  now matches  $x(n)(n).x$  via its second alternative, as it uses  $L^2$ . This means  $P^2$  will match  $x(n)(n).x(n)$ , while

$P^3$  will match  $\mathbf{x}(\mathbf{n})(\mathbf{n}).\mathbf{x}$  again, so  $P^2$  is the correct bound, and  $L^3$  matches  $\mathbf{x}(\mathbf{n})(\mathbf{n}).\mathbf{x}(\mathbf{n}).\mathbf{x}$ , the entire subject. It is easy to see that  $L^4$  will match just  $x$  again, as  $P^1$  will now match the whole subject using  $L^3$ , and the first alternative of  $L^4$  will fail.

Intuitively, the mutual recursion is playing as nested repetitions, with the inner repetition consuming  $(\mathbf{n})$  and the outer repetition consuming the result of the inner repetition plus  $.\mathbf{x}$ . The result is a PEG equivalent to the PEG for l-values in the end of Section 2 in the subjects it matches, but that yields parse strings that are correctly left-associative on each  $(\mathbf{n})$  and  $.\mathbf{x}$ .

We presented the new rules as extensions intended only for non-terminals with left-recursive rules, but this is not necessary: the **lvar** rules can replace **var** without changing the result of any proof tree. If a non-terminal does not appear in a left-recursive position then rules **lvar.3** and **lvar.4** can never apply by definition. These rules are the only place in the semantics where the contents of  $\mathcal{L}$  affects the result, so **lvar.2** is equivalent to **var.2** in the absence of left recursion. Analogously, if  $G[(P(A)]\ xy\ \mathcal{L}[(A, xy) \mapsto \mathbf{fail}] \xrightarrow{\text{PEG}} (y, x')$  then  $G[(P(A)]\ xy\ \mathcal{L}[(A, xy) \mapsto (y, x')] \xrightarrow{\text{PEG}} (y, x')$  in the absence of left recursion, so we will always have  $G[A]\ xy\ \mathcal{L}[(A, xy) \mapsto (y, x')] \xrightarrow{\text{INC}} (y, x')$  via **inc.3**, and **lvar.1** is equivalent to **var.1**. We can formalize this argument with the following lemma:

**Lemma 1 (Conservativeness).** *Given a PEG  $G$ , a parsing expression  $p$  and a subject  $xy$ , we have one of the following: if  $G[p]\ xy \xrightarrow{\text{PEG}} X$ , where  $X$  is **fail** or  $(y, x')$ , then  $G[p]\ xy\ \mathcal{L} \xrightarrow{\text{PEG}} X$ , as long as  $(A, w) \notin \mathcal{L}$  for any non-terminal  $A$  and subject  $w$  appearing as  $G[A]\ w$  in the proof tree of if  $G[p]\ xy \xrightarrow{\text{PEG}} X$ .*

*Proof.* By induction on the height of the proof tree for  $G[p]\ xy \xrightarrow{\text{PEG}} X$ . Most cases are trivial, as the extension of their rules with  $\mathcal{L}$  does not change the table. The interesting cases are **var.1** and **var.2**.

For case **var.2** we need to use rule **lvar.2**. We introduce  $(A, xy) \mapsto \mathbf{fail}$  in  $\mathcal{L}$ , but  $G[A]\ xy$  cannot appear in any part of the proof tree of  $G[P(A)]\ xy \xrightarrow{\text{PEG}} \mathbf{fail}$ , so we can just use the induction hypothesis.

For case **var.1** we need to use rule **lvar.1**. Again we have  $(A, xy) \mapsto \mathbf{fail}$  in  $\mathcal{L}$ , but we can use the induction hypothesis on  $G[P(A)]\ xy\ \mathcal{L}[(A, xy) \mapsto \mathbf{fail}]$  to get  $(y, x')$ . We also use **inc.3** to get  $G[P(A)]\ xy\ \mathcal{L}[(A, xy) \mapsto (y, x')] \xrightarrow{\text{INC}} (y, x')$  from  $G[P(A)]\ xy\ \mathcal{L}[(A, xy) \mapsto (y, x')]$ , using the induction hypothesis, finishing **lvar.1**.

A non-obvious consequence of our bounded left recursion semantics is that a rule that mixes left and right recursion is right-associative. For example, matching  $E \leftarrow E + E / n$  against  $\mathbf{n+n+n}$  yields the parse string  $E[E[n] + E[E[n] + E[n]]]$ . The reason is that  $E^2$  already matches the whole string:

$$\begin{aligned} E^1 &\leftarrow E^0 + E / n = n \\ E^2 &\leftarrow E^1 + E / n = n + E / n \end{aligned}$$

We have the first alternative of  $E^2$  matching  $\mathbf{n+}$  and then trying to match  $E$  with  $\mathbf{n+n}$ . Again we will have  $E^2$  matching the whole string, with the first alternative matching  $\mathbf{n+}$  and then matching  $E$  with  $\mathbf{n}$  via  $E^1$ . In practice this behavior is not a problem, as similar constructions are also problematic in parsing CFGs, and grammar writers are aware of them.

An implementation of our semantics can use *ad-hoc* extensions to control associativity in this kind of PEG, by having a right-recursive use of non-terminal  $A$  with a pending left-recursive use match through  $A^1$  directly instead of going through the regular process. Similar extensions can be used to have different associativities and precedences in operator grammars such as  $E \leftarrow E + E / E - E / E * E / (E) / n$ .

In order to prove that our semantics for PEGs with left-recursion gives meaning to any closed PEG (that is, any PEG  $G$  where  $P(A)$  is defined for all non-terminals in  $G$ ) we have to fix the case where a repetition may not terminate ( $p$  in  $p^*$  has not failed but not consumed any input). We can add a  $x \neq \varepsilon$  predicate to rule **rep.2** and then add a new rule:

$$\frac{G[p] \ x \ \mathcal{L} \ \overset{\text{PEG}}{\rightsquigarrow} (x, \varepsilon)}{G[p^*] \ x \ \mathcal{L} \ \overset{\text{PEG}}{\rightsquigarrow} (x, \varepsilon)} \quad (\mathbf{rep.3})$$

We also need a well-founded ordering  $<$  among the elements of the left side of relation  $\overset{\text{PEG}}{\rightsquigarrow}$ . For the subject we can use  $x < y$  if and only if  $x$  is a proper suffix of  $y$  as the order, for the parsing expression we can use  $p_1 < p_2$  if and only if  $p_1$  is a proper part of the structure of  $p_2$ , and for  $\mathcal{L}$  we can use  $\mathcal{L}[A \mapsto (x, y)] < \mathcal{L}$  if and only if either  $\mathcal{L}(A)$  is not defined or  $x < z$ , where  $\mathcal{L}(A) = (z, w)$ . Now we can prove the following lemma:

**Lemma 2 (Completeness).** *Given a closed PEG  $G$ , a parsing expression  $p$ , a subject  $xy$ , and a memoization table  $\mathcal{L}$ , we have either  $G[p] \ xy \ \mathcal{L} \ \overset{\text{PEG}}{\rightsquigarrow} (y, x')$  or  $G[p] \ xy \ \mathcal{L} \ \overset{\text{PEG}}{\rightsquigarrow} \text{fail}$ .*

*Proof.* By induction on the triple  $(\mathcal{L}, xy, p)$ . It is straightforward to check that we can always use the induction hypothesis on the antecedent of the rules of our semantics.

## 4 Related Work

Warth et al. [23] describes a modification of the packrat parsing algorithm to support both direct and indirect left recursion. The algorithm uses the packrat memoization table to detect left recursion, and then begins an iterative process that is similar to the process of finding the correct bound in our semantics.

Warth et al.'s algorithm is tightly coupled to the packrat parsing approach, and its full version, with support for indirect left recursion, is complex, as noted by the authors [22]. The release versions of the authors' PEG parsing library, OMeta [24], only implement support for direct left recursion to avoid the extra complexity [22].

The algorithm also produces surprising results with some grammars, both directly and indirectly left-recursive, due to the way it tries to reuse the packrat memoization table [1]. Our semantics does not share these issues, although it shows that a left-recursive packrat parser cannot index the packrat memoization table just by a parsing expression and a subject, as the  $\mathcal{L}$  table is also involved. One solution to this issue is to have a scoped packrat memoization table, with a new entry to  $\mathcal{L}$  introducing a new scope. We believe this solution is simpler to implement in a packrat parser than Warth et al.’s.

Tratt [21] presents an algorithm for supporting direct left recursion in PEGs, based on Warth et al.’s, that does not use a packrat memoization table and does not assume a packrat parser. The algorithm is simple, although Tratt also presents a more complex algorithm that tries to “fix” the right-recursive bias in productions that have both left and right recursion, like the  $E \leftarrow E + E / n$  example we discussed at the end of Section 3. We do not believe this bias is a problem, although it can be fixed in our semantics with ad-hoc methods.

IronMeta [20] is a PEG library for the Microsoft Common Language Runtime, based on OMeta [24], that supports direct and indirect left recursion using an implementation of an unpublished preliminary version of our semantics. This preliminary version is essentially the same, apart from notational details, so IronMeta can be considered a working implementation of our semantics. Initial versions of IronMeta used Warth et al.’s algorithm for left recursion [23], but in version 2.0 the author switched to an implementation of our semantics, which he considered “much simpler and more general” [20].

Parser combinators [8] are a top-down parsing method that is similar to PEGs, being another way to declaratively specify a recursive descent parser for a language, and share with PEGs the same issues of non-termination in the presence of left recursion. Frost et al. [4] describes an approach for supporting left recursion in parser combinators where a count of the number of left-recursive uses of a non-terminal is kept, and the non-terminal fails if the count exceeds the number of tokens of the input. We have shown in Section 3 that such an approach would not work with PEGs, because of the semantics of ordered choice (parser combinators use the same non-deterministic choice operator as CFGs). Ridge [19] presents another way of implementing the same approach for handling left recursion, and has the same issues regarding its application to PEGs.

ANTLR [16] is a popular parser generator that produces top-down parsers for Context-Free Grammars based on  $LL(*)$ , an extension of  $LL(k)$  parsing. Version 4 of ANTLR will have support for direct left recursion that is specialized for expression parsers [15], handling precedence and associativity by rewriting the grammar to encode a *precedence climbing* parser [7]. This support is heavily dependent on ANTLR extensions such as semantic predicates and backtracking.

## 5 Conclusion

We presented a conservative extension to the semantics of PEGs that gives an useful meaning for PEGs with left-recursive rules. It is the first extension that

is not based on packrat parsing as the parsing approach, while supporting both direct and indirect left recursion. The extension is based on bounded left recursion, where we limit the number of left-recursive uses a non-terminal may have, guaranteeing termination, and we use an iterative process to find the smallest bound that gives the longest match for a particular use of the non-terminal.

We also presented some examples that show how grammar writers can use our extension to express in PEGs common left-recursive idioms from Context-Free Grammars, such as using left recursion for left-associative repetition in expression grammars, and the use of mutual left recursion for nested left-associative repetition. We augmented the semantics with *parse strings* to show how we get a similar structure with left-recursive PEGs that we get with the parse trees of left-recursive CFGs.

Finally, we have proved the conservativeness of our extension, and also proved that all PEGs are complete with the extension, so termination is guaranteed for the parsing of any subject with any PEG, removing the need for any static checks of well-formedness beyond the simple check that every non-terminal in the grammar has a rule.

Our semantics has already been implemented in a PEG library that uses packrat parsing [20]. We are now working on adapting the semantics to a PEG parsing machine [12], as the first step towards an alternative implementation based on LPEG [9]. This implementation will incorporate ad-hoc extensions for controlling precedence and associativity in grammars mixing left and right recursion in the same rule, leading to more concise grammars.

## References

1. Cooney, D.: Problem with nullable left recursion and trailing rules in Packrat Parsers Can Support Left Recursion. PEG Mailing List (2009), available at <https://lists.csail.mit.edu/pipermail/peg/2009-November/000244.html>
2. Ford, B.: Packrat parsing: Simple, powerful, lazy, linear time. In: Proceedings of the 7th ACM SIGPLAN International Conference on Functional Programming. pp. 36–47. ICFP '02, ACM, New York, NY, USA (2002)
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, NY, USA (2004)
4. Frost, R.A., Hafiz, R., Callaghan, P.: Parser combinators for ambiguous left-recursive grammars. In: Proceedings of the 10th international conference on Practical aspects of declarative languages. pp. 167–181. PADL'08, Springer-Verlag, Berlin, Heidelberg (2008), <http://dl.acm.org/citation.cfm?id=1785754.1785766>
5. Gosling, J., Joy, B., Steele, G., Bracha, G.: The Java Language Specification. Addison-Wesley Professional (2005)
6. Grune, D., Jacobs, C.J.: Parsing Techniques – A Practical Guide. Ellis Horwood (1991)
7. Hanson, D.R.: Compact recursive-descent parsing of expressions. *Software: Practice and Experience* 15(12), 1205–1212 (1985), <http://dx.doi.org/10.1002/spe.4380151206>

8. Hutton, G.: Higher-order Functions for Parsing. *Journal of Functional Programming* 2(3), 323–343 (Jul 1992)
9. Ierusalimschy, R.: A text pattern-matching tool based on Parsing Expression Grammars. *Software - Practice and Experience* 39(3), 221–258 (2009)
10. Ierusalimschy, R., Figueiredo, L.H.d., Celes, W.: *Lua 5.1 Reference Manual*. Lua.Org (2006)
11. Kahn, G.: Natural semantics. In: *Proceedings of the 4th Annual Symposium on Theoretical Aspects of Computer Science*. pp. 22–39. STACS '87, Springer-Verlag, London, UK (1987)
12. Medeiros, S., Ierusalimschy, R.: A parsing machine for PEGs. In: *DLS '08: Dynamic languages Symposium*. pp. 1–12. ACM, New York, USA (2008)
13. Medeiros, S., Mascarenhas, F., Ierusalimschy, R.: From Regular Expressions to Parsing Expression Grammars. In: *SBLP '11: Brazilian Programming Languages Symposium* (2011)
14. Mizushima, K., Maeda, A., Yamaguchi, Y.: Packrat parsers can handle practical grammars in mostly constant space. In: *Proceedings of the 9th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*. pp. 29–36. PASTE '10, ACM, New York, NY, USA (2010)
15. Parr, T.: ANTLR's left-recursion prototype. PEG mailing list (2011), available at <https://lists.csail.mit.edu/pipermail/peg/2011-April/000414.html>
16. Parr, T., Fisher, K.: LL(\*): the foundation of the ANTLR parser generator. In: *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*. pp. 425–436. PLDI '11, ACM, New York, NY, USA (2011), <http://doi.acm.org/10.1145/1993498.1993548>
17. Redziejowski, R.: Parsing expression grammar as a primitive recursive-descent parser with backtracking. *Fundamenta Informaticae* 79(3-4), 513–524 (2008)
18. Redziejowski, R.R.: Some aspects of parsing expression grammar. *Fundamenta Informaticae* 85, 441–451 (January 2008)
19. Ridge, T.: Simple, functional, sound and complete parsing for all context-free grammars. In: *Proceedings of the First international conference on Certified Programs and Proofs*. pp. 103–118. CPP'11, Springer-Verlag, Berlin, Heidelberg (2011), [http://dx.doi.org/10.1007/978-3-642-25379-9\\_10](http://dx.doi.org/10.1007/978-3-642-25379-9_10)
20. Tisher, G.: IronMeta parser generator (2012), available at <http://ironmeta.sourceforge.net>
21. Tratt, L.: Direct left-recursive parsing expression grammars. Tech. Rep. EIS-10-01, School of Engineering and Information Sciences, Middlesex University (Oct 2010)
22. Warth, A.: OMeta squeak left recursion? OMeta Mailing List (June 2008), available at <http://vpri.org/pipermail/ometa/2008-June/000006.html>
23. Warth, A., Douglass, J., Millstein, T.: Packrat parsers can support left recursion. In: *PEPM '08: Proceedings of the 2008 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*. pp. 103–110. ACM, New York, NY, USA (2008)
24. Warth, A., Piumarta, I.: OMeta: an object-oriented language for pattern matching. In: *DLS '07: Proceedings of the 2007 symposium on Dynamic languages*. pp. 11–19. ACM, New York, NY, USA (2007)
25. Winskel, G.: *The Formal Semantics of Programming Languages: An Introduction*. Foundations of Computing, MIT Press (1993)