Science of Computer Programming 96 (2014) 177–190

Contents lists available at ScienceDirect

Science of Computer Programming

www.elsevier.com/locate/scico

Left recursion in Parsing Expression Grammars

Sérgio Medeiros [a], Fabio Mascarenhas [b,*], Roberto Ierusalimschy [c]

[a] School of Science and Technology, UFRN, Natal, Brazil
[b] Department of Computer Science, UFRJ, Rio de Janeiro, Brazil
[c] Department of Computer Science, PUC-Rio, Rio de Janeiro, Brazil

HIGHLIGHTS

- We present a semantics for left-recursive Parsing Expression Grammars.
- A small extension adds precedence/associativity declarations to operator grammars.
- We give a semantics for compilation of left-recursive PEGs to a parsing machine.
- Our semantics are conservative: non-left-recursive PEGs are unaffected.

ARTICLE INFO

Article history:
Received 31 March 2013
Received in revised form 31 December 2013
Accepted 17 January 2014
Available online 30 January 2014

Keywords:
Parsing Expression Grammars
Parsing
Left recursion
Parsing machine
Packrat parsing

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.

PEGs can also be compiled to programs in a low-level *parsing machine*. We present an extension to the semantics of the operations of this parsing machine that let it interpret left-recursive PEGs, and prove that this extension is correct with regard to our semantics for left-recursive PEGs.

1. Introduction

Parsing Expression Grammars (PEGs) [1] are a formalism for describing a language's syntax, based on the TDPL and GTDPL formalisms for top-down parsing with backtracking [2,3], 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. This *ordered choice* characteristic makes PEGs unambiguous by design, at the cost of making the language of a PEG harder to reason about than the language of

* Corresponding author.
E-mail addresses: sergio@ufs.br (S. Medeiros), mascarenhas@ufrj.br (F. Mascarenhas), roberto@inf.puc-rio.br (R. Ierusalimschy).

0167-6423/$ – see front matter © 2014 Elsevier B.V. All rights reserved.
http://dx.doi.org/10.1016/j.scico.2014.01.013

a CFG. The mindset of the author of a language specification that wants to use PEGs should be closer to the mindset of the programmer of a hand-written parser than the mindset of a grammar writer.

In comparison to CFGs, PEGs add the restriction that the order of alternatives in a production matters (unlike the alternatives in the EBNF notation of CFGs, which can be reordered at will), but introduce a richer syntax, based on the syntax of extended regular expressions. Extended regular expressions are a version of regular expressions popularized by pattern matching tools such as `grep` and languages such as Perl. PEGs also add *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.

As discussed by Ford [4], with the help of syntactic predicates it is easy to describe a scannerless PEG parser for a programming language with reserved words and identifiers, a task that is not easy to accomplish with CFGs. In a parser based on the CFG below, where we omitted the definition of *Letter*, the non-terminal *Id* could also match `for`, a reserved word, because it is a valid identifier:

$$\begin{aligned} Id &\rightarrow Letter\ IdAux \\ IdAux &\rightarrow Letter\ IdAux \mid \varepsilon \\ ResWord &\rightarrow if \mid for \mid while \end{aligned}$$

As CFGs do not have syntactic predicates, and context-free languages are not closed under complement, a separate lexer is necessary to distinguish reserved words from identifiers. In case of PEGs, we can just use the not predicate !, as below:

$$\begin{aligned} Id &\leftarrow !(ResWord)\ Letter\ IdAux \\ IdAux &\leftarrow Letter\ IdAux\ /\ \varepsilon \\ ResWord &\leftarrow if\ /\ for\ /\ while \end{aligned}$$

Moreover, syntactic predicates can also be used to better decide which alternative of a choice should match, and address the limitations of ordered choice. For example, we can use syntactic predicates to rewrite context-free grammars belonging to some classes of grammars that have top-down predictive parsers as PEGs without changing the gross structure of the grammar and the resulting parse trees [5].

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 [6]. While left recursion can be eliminated from the grammar, the necessary transformations can extensively change the shape of the resulting parse trees, making posterior processing of these trees harder than it could be.

There are proposals for adding support for left recursion to PEGs, but they either assume a particular PEG implementation approach, *packrat parsing* [7], or support just direct left recursion [8]. Packrat parsing [9] 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 [10,11]. 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 rewrite the grammar to eliminate it, because this can involve the rewriting of many rules [6].

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 our semantics should be intuitive for grammar writers.

A simple addition to our semantics lets us describe expression grammars with multiple levels of operator precedence and associativity more easily, in the style that users of LR parser generators are used to.

Finally, PEGs can also be compiled to programs in a low-level *parsing machine* [12]. We present an extension to the semantics of the operations of this parsing machine that let it interpret left-recursive PEGs, and prove that this extension is correct with regards to our semantics for left-recursive PEGs.

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 presents a simple addition that makes it easier to specify operator precedence and associativity in expression grammars; Section 5 presents left recursion as an extension to a parsing machine for PEGs; Section 6 reviews some related work on PEGs and left recursion in more detail; finally, Section 7 presents our concluding remarks.

$$\textbf{EmptyString}\quad \frac{}{G[\varepsilon]\ x \stackrel{\text{PEG}}{\rightsquigarrow} (x,\ \varepsilon)}\ (\textbf{empty.1}) \qquad \textbf{Terminal}\quad \frac{}{G[a]\ ax \stackrel{\text{PEG}}{\rightsquigarrow} (x,\ a)}\ (\textbf{char.1})$$

$$\frac{}{G[b]\ ax \stackrel{\text{PEG}}{\rightsquigarrow} \texttt{fail}}\ ,\ b \neq a\ (\textbf{char.2}) \qquad \frac{}{G[a]\ \varepsilon \stackrel{\text{PEG}}{\rightsquigarrow} \texttt{fail}}\ (\textbf{char.3})$$

$$\textbf{Variable}\quad \frac{G[P(A)]\ xy \stackrel{\text{PEG}}{\rightsquigarrow} (y,\ x')}{G[A]\ xy \stackrel{\text{PEG}}{\rightsquigarrow} (y,\ A[x'])}\ (\textbf{var.1}) \qquad \frac{G[P(A)]\ x \stackrel{\text{PEG}}{\rightsquigarrow} \texttt{fail}}{G[A]\ x \stackrel{\text{PEG}}{\rightsquigarrow} \texttt{fail}}\ (\textbf{var.2})$$

$$\textbf{Concatenation}\quad \frac{G[p_1]\ xyz \stackrel{\text{PEG}}{\rightsquigarrow} (yz,\ x') \quad G[p_2]\ yz \stackrel{\text{PEG}}{\rightsquigarrow} (z,\ y')}{G[p_1\, p_2]\ xyz \stackrel{\text{PEG}}{\rightsquigarrow} (z,\ x'y')}\ (\textbf{con.1})$$

$$\frac{G[p_1]\ xy \stackrel{\text{PEG}}{\rightsquigarrow} (y,\ x') \quad G[p_2]\ y \stackrel{\text{PEG}}{\rightsquigarrow} \texttt{fail}}{G[p_1\, p_2]\ xy \stackrel{\text{PEG}}{\rightsquigarrow} \texttt{fail}}\ (\textbf{con.2}) \qquad \frac{G[p_1]\ x \stackrel{\text{PEG}}{\rightsquigarrow} \texttt{fail}}{G[p_1\, p_2]\ x \stackrel{\text{PEG}}{\rightsquigarrow} \texttt{fail}}\ (\textbf{con.3})$$

$$\textbf{Choice}\quad \frac{G[p_1]\ xy \stackrel{\text{PEG}}{\rightsquigarrow} (y,\ x')}{G[p_1\,/\,p_2]\ xy \stackrel{\text{PEG}}{\rightsquigarrow} (y,\ x')}\ (\textbf{ord.1}) \qquad \frac{G[p_1]\ x \stackrel{\text{PEG}}{\rightsquigarrow} \texttt{fail} \quad G[p_2]\ x \stackrel{\text{PEG}}{\rightsquigarrow} \texttt{fail}}{G[p_1\,/\,p_2]\ x \stackrel{\text{PEG}}{\rightsquigarrow} \texttt{fail}}\ (\textbf{ord.2})$$

$$\frac{G[p_1]\ xy \stackrel{\text{PEG}}{\rightsquigarrow} \texttt{fail} \quad G[p_2]\ xy \stackrel{\text{PEG}}{\rightsquigarrow} (y,\ x')}{G[p_1\,/\,p_2]\ xy \stackrel{\text{PEG}}{\rightsquigarrow} (y,\ x')}\ (\textbf{ord.3})$$

$$\textbf{Not Predicate}\quad \frac{G[p]\ x \stackrel{\text{PEG}}{\rightsquigarrow} \texttt{fail}}{G[!p]\ x \stackrel{\text{PEG}}{\rightsquigarrow} (x,\ \varepsilon)}\ (\textbf{not.1}) \qquad \frac{G[p]\ xy \stackrel{\text{PEG}}{\rightsquigarrow} (y,\ x')}{G[!p]\ xy \stackrel{\text{PEG}}{\rightsquigarrow} \texttt{fail}}\ (\textbf{not.2})$$

$$\textbf{Repetition}\quad \frac{G[p]\ x \stackrel{\text{PEG}}{\rightsquigarrow} \texttt{fail}}{G[p^*]\ x \stackrel{\text{PEG}}{\rightsquigarrow} (x,\ \varepsilon)}\ (\textbf{rep.1}) \qquad \frac{G[p]\ xyz \stackrel{\text{PEG}}{\rightsquigarrow} (yz,\ x') \quad G[p^*]\ yz \stackrel{\text{PEG}}{\rightsquigarrow} (z,\ y')}{G[p^*]\ xyz \stackrel{\text{PEG}}{\rightsquigarrow} (z,\ x'y')}\ (\textbf{rep.2})$$

Fig. 1. Semantics of the $\stackrel{\text{PEG}}{\rightsquigarrow}$ relation.

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 [1]. The style we use comes from our previous work on PEGs [12,13], and makes the exposition of our extensions, and their 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 ε, 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 $\stackrel{\text{PEG}}{\rightsquigarrow}$ among a PEG, a parsing expression, a subject, and a result. The notation $G[p]\ xy \stackrel{\text{PEG}}{\rightsquigarrow} (y, x')$ 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 $G[p]\ xy \stackrel{\text{PEG}}{\rightsquigarrow} \texttt{fail}$ 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 $\{x \in T^* \mid G[p_S]\ xy \stackrel{\text{PEG}}{\rightsquigarrow} (y, x')\}$.

Fig. 1 presents the definition of $\stackrel{\text{PEG}}{\rightsquigarrow}$ using natural semantics [14,15], as a set of inference rules. Intuitively, ε just succeeds and leaves the subject unaffected; a matches and consumes itself, or fails; A tries to match the expression $P(A)$; $p_1 p_2$ 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; rule **var.1** brackets the parse string yielded by the right side of a non-terminal's rule, and tags it with the left side. The brackets and the tag are part of the parse string, and are assumed to not be in T, the PEG's input alphabet. 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* [1]. 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 [1] (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 [1]. 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 [16]. For example, the following is a simple left-associative CFG for additive expressions, written in EBNF notation:

$$E \to E + T \mid E - T \mid T$$
$$T \to n \mid (E)$$

Rewriting the above grammar as a PEG, by replacing | 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):

$$E \to T\left\{E'\right\}$$
$$T \to n \mid (E)$$
$$E' \to +T \mid -T$$

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:

$$E \leftarrow T\ E'^*$$
$$T \leftarrow n \ / \ (E)$$
$$E' \leftarrow +T \ / \ -T$$

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):

$$L \to P.x \mid x$$
$$P \to P(n) \mid L$$

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.

Table 1
Matching E with different bounds.

Subject	E^0	E^1	E^2	E^3	E^4	E^5	E^6
`n`	`fail`	ε	ε	ε	ε	ε	ε
`n+n`	`fail`	`+n`	ε	`+n`	ε	`+n`	ε
`n+n+n`	`fail`	`+n+n`	`+n`	ε	`+n+n`	`+n`	ε

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

$$L \to P.x \mid x$$

$$P \to x \, \{P'\}$$

$$P' \to (n) \mid .x$$

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):

$$L \leftarrow x \, S^*$$

$$S \leftarrow (\, \mathbf{(}n\mathbf{)} \,)^* .x$$

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 [17] and Lua [18]. 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 section, 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.

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:

$$E^0 \leftarrow \texttt{fail}$$

$$E^1 \leftarrow E^0 + n \,/\, n \;=\; \bot + n \,/\, n \;=\; n$$

$$E^2 \leftarrow E^1 + n \,/\, n \;=\; n + n \,/\, n$$

$$E^3 \leftarrow E^2 + n \,/\, n \;=\; (n + n \,/\, n) + n \,/\, n$$

$$\vdots$$

$$E^n \leftarrow E^{n-1} + n \,/\, n$$

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}

Left-Recursive Variable

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

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

$$\frac{\mathcal{L}(A, xy) = \mathtt{fail}}{G[A]\ xy\ \mathcal{L} \overset{\text{PEG}}{\rightsquigarrow} \mathtt{fail}}\ (\textbf{lvar.3}) \qquad \frac{\mathcal{L}(A, xy) = (y, x')}{G[A]\ xy\ \mathcal{L} \overset{\text{PEG}}{\rightsquigarrow} (y, A[x'])}\ (\textbf{lvar.4})$$

Increase Bound

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

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

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

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 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-recursive 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 Fig. 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.

Fig. 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 Fig. 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 n+n+n as the subject. When first matching E against n+n+n we have $(E, \mathtt{n+n+n}) \notin \mathcal{L}$, as $\mathcal{L}$ is initially empty, so we have to match $E + n\ /\ n$ against n+n+n with $\mathcal{L} = \{(E, \mathtt{n+n+n}) \mapsto \mathtt{fail}\}$. We now have to match $E + n$ against n+n+n, which means matching E again, but now we use rule **lvar.3**. The first alternative, $E + n$, fails, and we have $G[E + n\ /\ n]\ \mathtt{n+n+n}\ \{(E, \mathtt{n+n+n}) \mapsto \mathtt{fail}\} \overset{\text{PEG}}{\rightsquigarrow} (\mathtt{+n+n}, \mathtt{n})$ 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 $\overset{\text{INC}}{\leadsto}$ with $\mathcal{L} = \{(E, \mathtt{n+n+n}) \mapsto (\mathtt{+n+n}, \mathtt{n})\}$. This means we must try to match $E + n \ / \ n$ against n+n+n again, using the new $\mathcal{L}$. When we try the first alternative and match E with n+n+n the result will be $(\mathtt{+n+n}, E[\mathtt{n}])$ via **lvar.4**, and we can then use **con.1** to match $E + n$ yielding $(\mathtt{+n}, E[\mathtt{n}]\mathtt{+n})$. We have successfully increased the bound, and are in rule **inc.1**, with $x = \mathtt{n}$, $y = \mathtt{+n}$, and $zw = \mathtt{+n}$.

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

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

We can see that the parse string $E[E[E[\mathtt{n}]\mathtt{+n}]\mathtt{+n}]$ 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 $-$:

$$E \leftarrow M + E \ / \ M$$

$$M \leftarrow M - n \ / \ n$$

Matching E with n+n+n yields $E[M[\mathtt{n}]\mathtt{+}E[M[\mathtt{n}]\mathtt{+}E[M[\mathtt{n}]]]]$, as matching M against n+n+n, n+n, and n all consume just the first n while generating $M[\mathtt{n}]$, because we have $G[M - n \ / \ n]\ \mathtt{n+n+n}\ \{(M, \mathtt{n+n+n}) \mapsto \mathtt{fail}\} \overset{\text{PEG}}{\leadsto} (\mathtt{+n+n}, \mathtt{n})$ via **lvar.3**, **con.3**, and **ord.3**, and we have $G[M - n \ / \ n]\ \mathtt{n+n+n}\ \{(M, \mathtt{n+n+n}) \mapsto (\mathtt{+n+n}, \mathtt{n})\} \overset{\text{INC}}{\leadsto} (\mathtt{+n+n}, \mathtt{n})$ 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, with 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 yields $E[M[M[M[\mathtt{n}]\mathtt{-n}]\mathtt{-n}]]$, 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 Fig. 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:

$$L \leftarrow P.x \ / \ x$$

$$P \leftarrow P(n) \ / \ L$$

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 x(n)(n).x again, so P^2 is the correct bound, and L^3 matches x(n)(n).x(n).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 (n) and the outer repetition consuming the result of the inner repetition plus .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 (n) and .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 $G[(P(A)]\ xy\ \mathcal{L}[(A,xy)\mapsto \mathtt{fail}] \overset{\mathrm{PEG}}{\leadsto} (y,x')$ then $G[(P(A)]\ xy\ \mathcal{L}[(A,xy)\mapsto (y,x')] \overset{\mathrm{PEG}}{\leadsto} (y,x')$ in the absence of left recursion, so we will always have $G[A]\ xy\ \mathcal{L}[(A,xy)\mapsto (y,x')] \overset{\mathrm{INC}}{\leadsto} (y,x')$ via **inc.3**, and **lvar.1** is equivalent to **var.1**. We can formalize this argument with the following lemma:

Lemma 3.1 *(Conservativeness). Given a PEG G, a parsing expression p and a subject xy, we have one of the following: if $G[p]\ xy \overset{PEG}{\leadsto} X$, where X is $\mathtt{fail}$ or (y,x'), then $G[p]\ xy\ \mathcal{L} \overset{PEG}{\leadsto} X$, 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 $G[p]\ xy \overset{PEG}{\leadsto} X$.*

Proof. By induction on the height of the proof tree for $G[p]\ xy \overset{\mathrm{PEG}}{\leadsto} X$. 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 $(A,xy)\mapsto\mathtt{fail}$ in $\mathcal{L}$, but $G[A]\ xy$ cannot appear in any part of the proof tree of $G[P(A)]\ xy \overset{\mathrm{PEG}}{\leadsto} \mathtt{fail}$, so we can just use the induction hypothesis.

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

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 an $x\neq\varepsilon$ predicate to rule **rep.2** and then add a new rule:

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

We also need a well-founded ordering $<$ among the elements of the left side of relation $\overset{\mathrm{PEG}}{\leadsto}$. 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 3.2 *(Completeness). Given a closed PEG G, a parsing expression p, a subject xy, and a memoization table $\mathcal{L}$, we have either $G[p]\ xy\ \mathcal{L} \overset{PEG}{\leadsto} (y,x')$ or $G[p]\ xy\ \mathcal{L} \overset{PEG}{\leadsto} \mathtt{fail}$.*

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. □

We can write expression grammars with several operators with different precedences and associativities with the use of multiple non-terminals for the different precedence levels, and the use of left and right recursion for left and right associativity. A simple addition to the semantics, though, lets us write these grammars in a style that is more familiar to users of LR parsing tools. The next section presents this addition.

4. Controlling operator precedence

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 `n+n+n` yields the parse string $E[E[n]+E[E[n]+E[n]]]$, where the $+$ operator is right-associative. The reason is that E^2 already matches the whole string:

$$E^1\leftarrow E^0+E\ /\ n=n$$
$$E^2\leftarrow E^1+E\ /\ n=n+E\ /\ n$$

We have the first alternative of E^2 matching `n+` and then trying to match E with `n+n`. Again we will have E^2 matching the whole string, with the first alternative matching `n+` and then matching E with `n` via E^1.

The algorithm proposed by Warth et al. to handle left-recursion in PEGs [7] also has a right-recursive bias in rules that have both left and right-recursion, as noted by Tratt [8]. In this section, we give an extension to our semantics that lets the user control the associativity instead of defaulting to right-associativity in grammars like the above one.

Grammars that mix left and right recursion are also problematic when parsing CFGs, as mixing left and right recursion leads to ambiguous grammars. The CFG $E\to E+E\mid n$ is ambiguous, and will cause a shift-reduce conflict in an LR parser. Resolving the conflict by shifting (the default in LR parser generators) also leads to right-associative operators.

Mixing operators, such as in the grammar $E\to E+E\mid E*E\mid n$, will also cause LR conflicts. Writing these grammars is convenient when the number of operators is large, though, and an LR parser can control the associativity and precedence

$$\frac{(A, xyz) \notin \mathcal{L} \quad G[P(A)]\ xyz\ \mathcal{L}[(A, xyz) \mapsto \texttt{fail}] \overset{\text{PEG}}{\rightsquigarrow} (yz, x') \qquad G[P(A)]\ xyz\ \mathcal{L}[(A, xyz) \mapsto (yz, x', k)] \overset{\text{INC}}{\rightsquigarrow} (z, (xy)')}{G[A_k]\ xyz\ \mathcal{L} \overset{\text{PEG}}{\rightsquigarrow} (z, A[(xy)'])}\ (\textbf{lvar.1})$$

$$\frac{\mathcal{L}(A, xy) = (y, x', k') \quad k \geqslant k'}{G[A_k]\ xy\ \mathcal{L} \overset{\text{PEG}}{\rightsquigarrow} (y, A[x'])}\ (\textbf{lvar.4}) \qquad \frac{\mathcal{L}(A, xy) = (y, x', k') \quad k < k'}{G[A_k]\ xy\ \mathcal{L} \overset{\text{PEG}}{\rightsquigarrow} \texttt{fail}}\ (\textbf{lvar.5})$$

Fig. 3. Semantics for left-recursive non-terminals with precedence levels.

among different operators by picking which action to perform in the case of those shift-reduce conflicts. LR parser generators turn a specification of the precedence and associativity of each operator into a recipe for resolving conflicts in a way that will yield the correct parses for operator grammars that mix left and right recursion.

It is possible to adapt our left-recursive PEG semantics to also let the user control precedence and associativity in PEGs such as $E \leftarrow E + E\ /\ n$ and $E \leftarrow E + E\ /\ E * E\ /\ n$. The idea is to attach a *precedence level* to each occurrence of a left-recursive non-terminal, even if it is not in a left-recursive position. The level of a non-terminal is stored in the memoization table $\mathcal{L}$, and a left-recursive use of the non-terminal fails if its level is less than the level stored in the table. These precedence levels match the way left and right precedence numbers are assigned to operators in *precedence climbing* parsers [19].

Fig. 3 gives the semantics for left-recursive non-terminals with precedence levels. The default level is 1. The changes are minimal: rule **lvar.1** now stores the attached level in the memoization table before trying to increase the bound, while **lvar.4** guards against the attached level being lesser than the stored level. A new rule **lvar.5** fails the match of A_k if the attached level k is less than the level stored in the memoization table. Rules **lvar.2**, **lvar.3**, and **inc.2** are unchanged, and rules **inc.1** and **inc.3** just propagate the store level without changes. These five rules have been elided.

Going back to our previous example, the grammar $E \leftarrow E + E\ /\ n$, we can get a left-associative $+$ by giving the left-recursive E a precedence level that is greater than the precedence level of the right-recursive E. So we should rewrite the previous grammar as $E \leftarrow E_1 + E_2\ /\ n$. As the precedence level of the right-recursive E is 2, after a right-recursive call the left-recursive call E_1, with precedence level 1, always fails when trying to match. This means that E_2 can match just a prefix n of the subject, via the second alternative of the ordered choice.

In a similar way, we could rewrite $E \leftarrow E + E\ /\ n$ as $E \leftarrow E_2 + E_1\ /\ n$, or just use the default level and interpret it as $E \leftarrow E_1 + E_1\ /\ n$, to get a right-associative $+$ in our new semantics that has precedence levels. Now after a right-recursive call the left-recursive call can match an unlimited prefix of the subject.

We could impose left-associativity as the default in case of a grammar that mixes left and right recursion just by changing $k \geqslant k'$ in rule **lvar.4** to $k > k'$ and by changing $k < k'$ in rule **lvar.5** to $k \leqslant k'$. The disadvantage is that this makes specifying precedence levels with right-associative operators a little harder, but this is a user interface issue: a tool could use a YACC-style interface, with `%left` and `%right` precedence directives and the order these directives appear giving the relative precedence, with the tool assigning precedence levels automatically. We left right-associativity as the default for the semantics presented in this section because it makes the multi-operator and multi-associativity grammar example we give next easier to understand.

In a grammar with several operators grouped in different precedence classes, the precedence classes would map to different levels in the non-terminals. A grammar with left-associative $+$ and $-$ operators in the lowest precedence class, left-associative $*$ and $\div$ operators in the middle precedence class, a right-associative $**$ operator in second-to-highest precedence class, an unary $-$ in the highest precedence class, and parentheses, could be described by the following PEG:

$$E \leftarrow E_1 + E_2\ /\ E_1 - E_2\ /\ E_2 * E_3\ /\ E_2 \div E_3\ /\ E_3 ** E_3\ /\ - E_4\ /\ (E_1)\ /\ n$$

As an example of how precedence levels work in practice, let us consider the PEG $E \leftarrow E_1 + E_2\ /\ E_2 * E_2\ /\ n$, where we set precedence levels so a right-associative $*$ has precedence over a left-associative $+$, and check what happens on the inputs n+n+n, n*n*n, n*n+n, and n+n*n. We will assume that we are trying to match E_1 against each subject.

For n+n+n we first try to match $E_1 + E_2\ /\ E_2 * E_2\ /\ n$ with $\mathcal{L}(E, \mathrm{n+n+n}) = \texttt{fail}$. The first two alternatives will fail via **lvar.3**, and the third matches the first n. We then try to increase the bound with $\mathcal{L}(E, \mathrm{n+n+n}) = (\mathrm{+n+n}, n, 1)$. In the first alternative, E_1 matches via **lvar.4**, and we start again with E_2 by trying to match $E_1 + E_2\ /\ E_2 * E_2\ /\ n$ against n+n with $\mathcal{L}(E, \mathrm{n+n}) = \texttt{fail}$.

Again, the third alternative matches the first n, and we will try to increase the bound with $\mathcal{L}(E, \mathrm{n+n}) = (\mathrm{+n}, n, 2)$. Notice the precedence level is 2. This level makes the first alternative of $E_1 + E_2\ /\ E_2 * E_2\ /\ n$ fail against n+n, and the second alternative fails on $*$. The third alternative succeeds, but does not increase the bound, so matching E_2 against n+n will match just the first n, and $E_1 + E_2$ will match the first n+n of the input n+n+n.

We then try to increase the bound of E_1 against n+n+n again, with $\mathcal{L}(E, \mathrm{n+n+n}) = (\mathrm{+n}, E[n] + E[n], 1)$. The first alternative of E will now match the whole subject, yielding the parse string $E[E[n] + E[n]] + E[n]$, and a third try to increase the bound will fail, and the final result is $E[E[E[n] + E[n]] + E[n]]$, with a left-associative $+$.

For the subject n*n*n, after matching the first n, we try to increase the bound with $\mathcal{L}(E, \mathrm{n*n*n}) = (\mathrm{*n*n}, n, 1)$. The precedence level is still 1 because we are matching E_1 against the whole subject. The first alternative fails on $+$, the first E_2

of the second alternative succeeds via **lvar.4**, and we will try the second E_2 of the second alternative against n∗n. The first n matches with $\mathcal{L}(E, \texttt{n} * \texttt{n}) = \texttt{fail}$, and then we try to increase the bound with $\mathcal{L}(E, \texttt{n} * \texttt{n}) = (*\texttt{n}, n, 2)$. Now the second alternative succeeds again with **lvar.4**, as the precedence level is 2, and eventually the final result is $E[E[n] * E[E[n] * E[n]]]$, with a right-associative $*$.

The behavior on the other two subjects, n∗n+n and n+n∗n, are respectively analogous to the first two, with the right-recursive E_2 on $E_2 * E_2$ failing to match n+n on the first subject because the level is 2, and the right-recursive E_2 on $E_1 + E_2$ matching n∗n on the second subject due to the same reason, yielding the results $E[E[E[n] * E[n]] + E[n]]$ and $E[E[n] + E[E[n] * E[n]]]$. Both results have the correct precedence.

While it is natural to view a PEG as a blueprint for a recursive descent parser with local backtracking, with optional use of memoization for better time complexity, an alternative execution model compiles PEGs to instructions on a lower-level *parsing machine* [12]. The next section briefly describes the formal model of one such machine, and extends it to have the same behavior on left-recursive PEGs as the semantics of Figs. 2 and 3.

5. A parsing machine for left-recursive PEGs

The core of LPEG, an implementation of PEGs for the Lua language [20], is a virtual parsing machine. LPEG compiles each parsing expression to a program of the parsing machine, and the program for a compound expression is a combination of the programs for its subexpressions. The parsing machine has a formal model of its operation, and proofs that compiling a PEG yields a program that is equivalent to the original PEG [12].

The parsing machine has a register to hold the program counter used to address the next instruction to execute, a register to hold the subject, and a stack that the machine uses for pushing call frames and backtrack frames. A call frame is just a return address for the program counter, and a backtrack frame is an address for the program counter and a subject. The machine's instructions manipulate the program counter, subject, and stack.

The compilation of the following PEG (for a sequence of zero or more *a*s followed by any character other than *a* followed by a *b*) uses all of the basic instructions of the parsing machine:

$$A \rightarrow !a\,.\,B \;/\; aA$$

$$B \rightarrow b$$

This PEG compiles to the following program:

```
    Call A
A:  Choice A1
    Choice A2
    Char a
    Commit A3
A3: Fail
A2: Any
    Call B
    Commit A4
A1: Char a
    Jump A
A4: Return
B:  Char b
    Return
```

The behavior of each instruction is straightforward: Call pushes a call frame with the address of the next instruction and jumps to a label, Return pops a call frame and jumps to its address, Jump is an unconditional jump, Char tries to match a character with the start of the subject, consuming the first character of the subject if successful, Any consumes the first character of the subject (failing if the subject is ε), Choice pushes a backtrack frame with the subject and the address of the label, Commit discards the backtrack frame in the top of the stack and jumps, and Fail forces a failure. When the machine enters a failure state it pops call frames from the stack until reaching a backtrack frame, then pops this frame and resumes execution with the subject and address stored in it.

Formally, the program counter register, the subject register, and the stack form a machine state. We represent it as a tuple $\mathbb{N} \times \mathrm{T}^* \times \mathrm{Stack}$, in the order above. A machine state can also be a failure state, represented by $\mathbf{Fail}\langle e\rangle$, where e is the stack. Stacks are lists of $(\mathbb{N} \times \mathrm{T}^*) \cup \mathbb{N}$, where $\mathbb{N} \times \mathrm{T}^*$ represents a backtrack frame and $\mathbb{N}$ represents a call frame.

Fig. 4 presents the operational semantics of the parsing machine as a relation between machine states. The program $\mathcal{P}$ that the machine executes is implicit. The relation $\xrightarrow{\text{Instruction}}$ relates two states when *pc* in the first state addresses an instruction matching the label, and the guard (if present) is valid.

We can extend the parsing machine to handle left recursion with precedence levels by changing the semantics of the Call and Return instructions, as well as the semantics of failure states. The new Call instruction also needs a second parameter, the precedence level. We also need to add a new kind of frame to the machine stack: a *left-recursive call* frame.

$$\langle pc,\ a{:}s,\ e\rangle \xrightarrow{\texttt{Char}\ a} \langle pc+1,\ s,\ e\rangle$$
$$\langle pc,\ b{:}s,\ e\rangle \xrightarrow{\texttt{Char}\ a} \mathbf{Fail}\langle e\rangle, a \neq b$$
$$\langle pc,\ \varepsilon,\ e\rangle \xrightarrow{\texttt{Char}\ a} \mathbf{Fail}\langle e\rangle$$
$$\langle pc,\ a{:}s,\ e\rangle \xrightarrow{\texttt{Any}} \langle pc+1,\ s,\ e\rangle$$
$$\langle pc,\ \varepsilon,\ e\rangle \xrightarrow{\texttt{Any}} \mathbf{Fail}\langle e\rangle$$
$$\langle pc,\ s,\ e\rangle \xrightarrow{\texttt{Choice}\ i} \langle pc+1,\ s,\ (pc+i, s){:}e\rangle$$
$$\langle pc,\ s,\ e\rangle \xrightarrow{\texttt{Jump}\ l} \langle pc+l,\ s,\ e\rangle$$
$$\langle pc,\ s,\ e\rangle \xrightarrow{\texttt{Call}\ l} \langle pc+l,\ s,\ (pc+1){:}e\rangle$$
$$\langle pc_1,\ s,\ pc_2{:}e\rangle \xrightarrow{\texttt{Return}} \langle pc_2,\ s,\ e\rangle$$
$$\langle pc,\ s,\ h{:}e\rangle \xrightarrow{\texttt{Commit}\ l} \langle pc+l,\ s,\ e\rangle$$
$$\langle pc,\ s,\ e\rangle \xrightarrow{\texttt{Fail}} \mathbf{Fail}\langle e\rangle$$
$$\mathbf{Fail}\langle pc{:}e\rangle \xrightarrow{\quad} \mathbf{Fail}\langle e\rangle$$
$$\mathbf{Fail}\langle (pc, s){:}e\rangle \xrightarrow{\quad} \langle pc,\ s,\ e\rangle$$

Fig. 4. Operational semantics of the parsing machine.

$$\langle pc,\ s,\ e\rangle \xrightarrow{\texttt{Call}\ l\ k} \langle pc+l,\ s,\ (pc+1, pc+l, s, \texttt{fail}, k){:}e\rangle,\ \text{where } (pc+l, s) \notin \mathcal{L}_e$$
$$\langle pc,\ s,\ e\rangle \xrightarrow{\texttt{Call}\ l\ k} \mathbf{Fail}\langle e\rangle,\ \text{where } \mathcal{L}_e(pc+l, s) = \texttt{fail}$$
$$\langle pc,\ s,\ e\rangle \xrightarrow{\texttt{Call}\ l\ k} \langle pc+1,\ s',\ e\rangle,\ \text{where } \mathcal{L}_e(pc+l, s) = (s', k') \text{ and } k \geqslant k'$$
$$\langle pc,\ s,\ e\rangle \xrightarrow{\texttt{Call}\ l\ k} \mathbf{Fail}\langle e\rangle,\ \text{where } \mathcal{L}_e(pc+l, s) = (s', k') \text{ and } k < k'$$
$$\langle pc,\ s'',\ (pc_r, pc_A, s, s', k){:}e\rangle \xrightarrow{\texttt{Return}} \langle pc_A,\ s,\ (pc_r, pc_A, s, s'', k) : e\rangle,\ \text{where } |s''| < |s'| \text{ or } s' = \texttt{fail}$$
$$\langle pc,\ s'',\ (pc_r, pc_A, s, s', k){:}e\rangle \xrightarrow{\texttt{Return}} \langle pc_r,\ s',\ e\rangle,\ \text{where } |s''| \geqslant |s'|$$
$$\mathbf{Fail}\langle (pc_r, pc_A, s, s', k){:}e\rangle \xrightarrow{\quad} \langle pc_r,\ s',\ e\rangle$$
$$\mathbf{Fail}\langle pc{:}e\rangle \xrightarrow{\quad} \mathbf{Fail}\langle e\rangle$$
$$\mathbf{Fail}\langle (pc, s){:}e\rangle \xrightarrow{\quad} \langle pc,\ s,\ e\rangle$$

Fig. 5. Semantics of left recursion in the parsing machine.

A left-recursive call frame to a non-terminal A is a tuple (pc_r, pc_A, s, X, k) where pc_r is the address of the instruction that follows the call instruction, pc_A is the address of the first instruction of the non-terminal A, s is the subject at the call site, X is either `fail` or a suffix of s, and is the memoized result, and k is the precedence level. These frames encode both the control information for the machine, and the memoization table for left recursion.

The idea of the extension is for the `Call` instruction to handle the **lvar.3**, **lvar.4**, and **lvar.5** cases, as well as parts of **lvar.1** and **lvar.2**. `Return` then handles the rest of **lvar.1**, as well as **inc.1** and **inc.3**, while the semantics of the failure states handle **inc.2** and the rest of **lvar.2**.

Fig. 5 gives the operational semantics of the extension, and replace the rules of Fig. 4 that deal with `Call`, `Return`, and failure states. As a convenience, we use the notation $(pc, s) \notin \mathcal{L}_e$ to mean that there is no left-recursive call frame in e with pc in the second position and s in the third position, the notation $\mathcal{L}_e(pc, s) = \texttt{fail}$ to mean that in the left-recursive call frame of e with pc in the second position and s in the third position, the value in the fourth position is `fail`, and the notation $\mathcal{L}_e(pc, s) = (s', k)$ to mean that in the left-recursive call frame of e with pc in the second position and s in the third position, the values of the fourth and fifth positions are s' and k, respectively.

The formal model of the parsing machine represents the compilation process using a transformation function Π. The term $\Pi(G, i, p)$ is the translation of pattern p in the context of the grammar G, where i is the position where the program starts relative to the start of the compiled grammar. We use the notation $|\Pi(G, i, p)|$ to mean the number of instructions in the program $\Pi(G, i, p)$.

The following lemma gives the correctness condition for the transformation Π for non-left-recursive PEGs in the original semantics of PEGs and the parsing machine, based on an extension of the relation $\xrightarrow{\text{Instruction}}$ for a concatenation of multiple instructions.

Lemma 5.1 *(Correctness of Π without left recursion). Given a PEG G, a parsing expression p, and a subject xy, if $G[p]\ xy \overset{\text{PEG}}{\rightsquigarrow} (y, x')$ then $\langle pc, xy, e\rangle \xrightarrow{\Pi(G,i,p)} \langle pc + |\Pi(G, i, p)|, y, e\rangle$, and if $G[p]\ xy \overset{\text{PEG}}{\rightsquigarrow} \mathtt{fail}$ then $\langle pc, xy, e\rangle \xrightarrow{\Pi(G,i,p)} \mathbf{Fail}\langle e\rangle$, where pc is the address of the first instruction of $\Pi(G, i, p)$.*

Proof. Given in the paper that describes the parsing machine [12]. □

Before proving a revised correctness lemma for the compilation of left-recursive PEGs, we need to relate memoization tables $\mathcal{L}$ with stacks that have left-recursive call frames. We say that a stack e is *consistent* with a memoization table $\mathcal{L}$ if and only if, for every entry in $\mathcal{L}$ with a non-terminal A, a subject s, a result X (either a suffix of s or $\mathtt{fail}$), and a precedence level k, there is a left-recursive call frame (pc_r, pc_A, s, X, k) in e where pc_A is the position where the program for $P(A)$ starts, and pc_R is any position.

Intuitively, a stack e is consistent with a memoization table $\mathcal{L}$ if each entry in $\mathcal{L}$ has a corresponding left-recursive call frame in e, and vice-versa.

The lemma below revises the correctness condition for the transformation Π to account for both the left-recursive PEG semantics and the left-recursive parsing machine semantics:

Lemma 5.2 *(Correctness of Π with left recursion). Given a PEG G, a parsing expression p, a non-terminal A of G, a subject xyz, a memoization table $\mathcal{L}$, and a stack e consistent with $\mathcal{L}$, if $G[p]\ xyz\ \mathcal{L} \overset{\text{PEG}}{\rightsquigarrow} (z, w)$ then $\langle pc, xyz, e\rangle \xrightarrow{\Pi(G,i,p)} \langle pc + |\Pi(G, i, p)|, z, e\rangle$, if $G[p]\ xyz\ \mathcal{L} \overset{\text{PEG}}{\rightsquigarrow} \mathtt{fail}$ then $\langle pc, xyz, e\rangle \xrightarrow{\Pi(G,i,p)} \mathbf{Fail}\langle e\rangle$, and if $G[P(A)]\ xyz\ \mathcal{L} \cup \{(A, xyz) \mapsto (yz, v)\} \overset{\text{INC}}{\rightsquigarrow} (z, w)$ then $\langle pc, xyz, (pc_r, pc_A, xyz, yz, k) : e\rangle \xrightarrow{\Pi(G,i,P(A))\ \text{Return}} \langle pc_r, y, e\rangle$.*

Proof. By induction on the heights of the proof trees for the antecedents. Most cases are similar to the ones in the proof of the previous lemma [12], as they involve parts of the semantics of PEGs and the parsing machine that are unchanged. The interesting cases occur when p is a non-terminal A (in the case of the relation $\overset{\text{PEG}}{\rightsquigarrow}$) or the right side of the production of a non-terminal A (in the case of relation $\overset{\text{INC}}{\rightsquigarrow}$).

The subcases where the proof tree of A ends with **lvar.3**, **lvar.4**, or **lvar.5** follow from the consistency constraint on the stack e and the corresponding rule for the `Call` instruction. Subcase **lvar.2** follows from an application of the induction hypothesis, and the rule for failure states with a left-recursive call frame on the top of the stack.

Subcase **lvar.1** follows from the induction hypothesis on the subtree ending with relation $\overset{\text{PEG}}{\rightsquigarrow}$, the first transition rule for `Return`, and the induction hypothesis on the subtree ending with relation $\overset{\text{INC}}{\rightsquigarrow}$.

Subcase **inc.1** is similar to **lvar.1**, while subcase **inc.2** is similar to **lvar.3**. Subcase **inc.3** follows from the induction hypothesis and the second transition rule for `Return`. □

The use of left recursion is common in context free grammar definitions, and several attempts have been made to give meaning to left-recursive PEGs that would make it easier to reuse left-recursive CFGs. The next section reviews previous work on left recursion for PEGs, and related work on left recursion for top-down parsing approaches.

6. Related work

Warth et al. [7] describe 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 [21]. The released versions of the authors' PEG parsing library, OMeta [22], only implement support for direct left recursion to avoid the extra complexity [21]. Our approach does not rely on memoization; although the left recursion table $\mathcal{L}$ looks like a memoization table, the recursive structure of the natural semantics makes it work more like a stack. The translation of our semantics to a parsing machine in Section 5 makes this point clear.

Warth et al.'s algorithm also produces surprising results with some grammars, both directly and indirectly left-recursive, that are related to its reuse of the memoization table, as reported by some users of Warth et al.'s algorithm [23–25]. As an example, consider the following grammar, based on the report of an user [23]:

$$S \leftarrow X$$

$$X \leftarrow X\ Y\ /\ \varepsilon$$

$$Y \leftarrow x$$

The algorithm fails to parse strings of the form x^+, because the nullable left recursion of X will make the algorithm mistakenly fail the matching of the non-terminal Y that follows X (if the first alternative was $X\,x$ instead of $X\,Y$ the grammar would work).

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 fixing and implementing Warth et al.'s close to fifty lines of pseudo-code (see the discussion about IronMeta below).

Tratt [8] 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. Tratt's algorithm fixes the issues that Warth et al.'s algorithm has with nullable left recursion, but only supports direct left recursion, while our solution works with any kind of left recursion.

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. Although we do not believe this bias is a problem, our semantics can give a more general solution to operator precedence and associativity in PEGs, as we have shown in Section 4.

IronMeta [26] is a PEG library for the Microsoft Common Language Runtime, based on OMeta [22], 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 and the presence of precedence levels, so IronMeta can be considered a working implementation of the semantics of Fig. 2. Initial versions of IronMeta used Warth et al.'s algorithm for left recursion [7], but in version 2.0 the author switched to an implementation of our semantics, which he considered "much simpler and more general" [26].

Parser combinators [27] 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. [28] describe 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 [29] presents another way of implementing the same approach for handling left recursion, and has the same issues regarding its application to PEGs.

ANTLR [30] 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 [31], handling precedence and associativity by rewriting the grammar to encode a *precedence climbing* parser [19]. This support is heavily dependent on ANTLR extensions such as semantic predicates and backtracking, and not translatable to PEGs.

Generalized LL parsing [32,33] solves the problem of left recursion in LL(1) recursive descent parsers, along with the problem of LL(1) conflicts, by making the parse proceed among the different conflicting alternatives "in parallel", replacing the recursive descent stack with a *graph structured stack* (GSS), a data structure adapted from Generalized LR parsers [34]. The resulting parsing technique can parse the language of any context-free language. As this includes ambiguous grammars, the tree construction part of the parser is non-trivial to implement efficiently, using *shared packed parse forest* data structure also adapted from generalized bottom-up techniques [35]. The behavior of the algorithm is tied to the non-deterministic semantics of CFG alternatives, and it relies on consuming the whole input to detect a successful parse, making the technique not applicable as a solution for left recursion in PEGs, which have deterministic, ordered choice, and can successfully consume just a prefix of the input.

7. Conclusion

We presented a conservative extension to the semantics of PEGs that gives a 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.

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.

We have also described a simple addition to our semantics that makes it easier to describe expression grammars with multiple levels of operator precedence and associativity, a pattern that users of LR parser generators are used to.

Finally, we presented a semantics for describing left-recursive PEGs as programs in a low-level *parsing machine*, as an extension to the semantics of a parsing machine for non-left-recursive PEGs [12]. We prove that this extension is correct with regards to our semantics for left-recursive PEGs.

Our semantics has already been implemented in a PEG library that uses packrat parsing [26]. We also have a prototype of the left-recursive parsing machine semantics built on top of LPEG [20], an implementation of the parsing machine for regular PEGs that we extend. Preliminary benchmarks show that left-recursive grammars perform similarly to the same grammars with left recursion removed.

References

[1] B. Ford, Parsing expression grammars: a recognition-based syntactic foundation, in: POPL '04: Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, ACM, New York, NY, USA, ISBN 1-58113-729-X, 2004, pp. 111–122.
[2] A. Birman, J.D. Ullman, Parsing algorithms with backtrack, Inf. Control 23 (1) (1973) 1–34, http://dx.doi.org/10.1016/S0019-9958(73)90851-6.
[3] A.V. Aho, J.D. Ullman, The Theory of Parsing, Translation, and Compiling, Prentice-Hall, Inc., Upper Saddle River, NJ, USA, ISBN 0-13-914556-7, 1972.
[4] B. Ford, Packrat parsing: a practical linear-time algorithm with backtracking, Master's thesis, Massachusetts Institute of Technology, 2002.
[5] F. Mascarenhas, S. Medeiros, R. Ierusalimschy, On the relation between context-free grammars and parsing expression grammars, arXiv:1304.3177 [abs], 2013.
[6] R. Redziejowski, Parsing expression grammar as a primitive recursive-descent parser with backtracking, Fundam. Inform. 79 (3–4) (2008) 513–524.
[7] A. Warth, J. Douglass, T. Millstein, Packrat parsers can support left recursion, in: PEPM '08: Proceedings of the 2008 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation, ACM, New York, NY, USA, ISBN 978-1-59593-977-7, 2008, pp. 103–110.
[8] L. Tratt, Direct left-recursive parsing expression grammars, Tech. Rep. EIS-10-01, School of Engineering and Information Sciences, Middlesex University, 2010.
[9] B. Ford, Packrat parsing: Simple, powerful, lazy, linear time, in: Proceedings of the 7th ACM SIGPLAN International Conference on Functional Programming, ICFP '02, ACM, New York, NY, USA, ISBN 1-58113-487-8, 2002, pp. 36–47.
[10] R.R. Redziejowski, Some aspects of parsing expression grammar, Fundam. Inform. 85 (2008) 441–451.
[11] K. Mizushima, A. Maeda, Y. Yamaguchi, 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, PASTE '10, ACM, New York, NY, USA, ISBN 978-1-4503-0082-7, 2010, pp. 29–36.
[12] S. Medeiros, R. Ierusalimschy, A parsing machine for PEGs, in: DLS '08: Dynamic Languages Symposium, ACM, New York, USA, ISBN 978-1-60558-270-2, 2008, pp. 1–12.
[13] S. Medeiros, F. Mascarenhas, R. Ierusalimschy, From regular expressions to Parsing Expression Grammars, in: SBLP '11: Brazilian Programming Languages Symposium, 2011.
[14] G. Kahn, Natural semantics, in: Proceedings of the 4th Annual Symposium on Theoretical Aspects of Computer Science, STACS '87, Springer-Verlag, London, UK, ISBN 3-540-17219-X, 1987, pp. 22–39.
[15] G. Winskel, The Formal Semantics of Programming Languages: An Introduction, Foundations of Computing, MIT Press, ISBN 9780262731034, 1993.
[16] D. Grune, C.J. Jacobs, Parsing Techniques – A Practical Guide, Ellis Horwood, 1991.
[17] J. Gosling, B. Joy, G. Steele, G. Bracha, The Java Language Specification, Addison-Wesley Professional, ISBN 0321246780, 2005.
[18] R. Ierusalimschy, L.H.d. Figueiredo, W. Celes, Lua 5.1 Reference Manual, Lua.Org, ISBN 8590379833, 2006.
[19] D.R. Hanson, Compact recursive-descent parsing of expressions, Softw. Pract. Exp. 15 (12) (1985) 1205–1212, http://dx.doi.org/10.1002/spe.4380151206.
[20] R. Ierusalimschy, A text pattern-matching tool based on Parsing Expression Grammars, Softw. Pract. Exp. 39 (3) (2009) 221–258.
[21] A. Warth, OMeta squeak left recursion, OMeta Mailing List, http://vpri.org/pipermail/ometa/2008-June/000006.html, 2008.
[22] A. Warth, I. Piumarta, OMeta: an object-oriented language for pattern matching, in: DLS '07: Proceedings of the 2007 Symposium on Dynamic Languages, ACM, New York, NY, USA, ISBN 978-1-59593-868-8, 2007, pp. 11–19.
[23] D. Cooney, Problem with nullable left recursion and trailing rules in "Packrat Parsers can support left recursion", PEG Mailing List, https://lists.csail.mit.edu/pipermail/peg/2009-November/000244.html, 2009.
[24] G. Tisher, IronMeta 2.0 released, PEG Mailing List, https://lists.csail.mit.edu/pipermail/peg/2010-December/000339.html, 2010.
[25] O. Bílka, Another issue with left recursion, OMeta Mailing List, http://vpri.org/pipermail/ometa/2011-January/000393.html, 2011.
[26] G. Tisher, IronMeta parser generator, http://ironmeta.sourceforge.net, 2012.
[27] G. Hutton, Higher-order functions for parsing, J. Funct. Program. 2 (3) (1992) 323–343.
[28] R.A. Frost, R. Hafiz, P. Callaghan, Parser combinators for ambiguous left-recursive grammars, in: Proceedings of the 10th International Conference on Practical Aspects of Declarative Languages, PADL'08, Springer-Verlag, Berlin, Heidelberg, ISBN 3-540-77441-6, 2008, pp. 167–181, http://dl.acm.org/citation.cfm?id=1785754.1785766, 978-3-540-77441-9.
[29] T. Ridge, Simple, functional, sound and complete parsing for all context-free grammars, in: Proceedings of the First International Conference on Certified Programs and Proofs, CPP'11, Springer-Verlag, Berlin, Heidelberg, ISBN 978-3-642-25378-2, 2011, pp. 103–118.
[30] T. Parr, K. Fisher, LL(*): the foundation of the ANTLR parser generator, in: Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '11, ACM, New York, NY, USA, ISBN 978-1-4503-0663-8, 2011, pp. 425–436.
[31] T. Parr, ANTLR's left-recursion prototype, PEG mailing list, https://lists.csail.mit.edu/pipermail/peg/2011-April/000414.html, 2011.
[32] E. Scott, A. Johnstone, GLL parse-tree generation, Sci. Comput. Program. 78 (10) (2013) 1828–1844, http://dx.doi.org/10.1016/j.scico.2012.03.005.
[33] A. Johnstone, E. Scott, Modelling GLL parser implementations, in: B. Malloy, S. Staab, M. Brand (Eds.), Software Language Engineering, in: Lect. Notes Comput. Sci., vol. 6563, Springer, Berlin, Heidelberg, ISBN 978-3-642-19439-9, 2011, pp. 42–61.
[34] M. Tomita, Graph-structured stack and natural language parsing, in: Proceedings of the 26th Annual Meeting of the Association for Computational Linguistics, Association for Computational Linguistics, Buffalo, New York, USA, 1988, pp. 249–257, http://www.aclweb.org/anthology/P88-1031.
[35] E. Scott, A. Johnstone, R. Economopoulos, BRNGLR: a cubic Tomita-style GLR parsing algorithm, Acta Inform. 44 (6) (2007) 427–461, http://dx.doi.org/10.1007/s00236-007-0054-z.