From EBNF to PEG*

Roman R. Redziejowski

Abstract

Parsing Expression Grammar (PEG) encodes a recursive-descent parser with limited backtracking. The parser has many useful properties, and with the use of memoization, it works in a linear time. In its appearance, PEG is almost identical to a grammar in Extended Backus-Naur Form (EBNF), but usually defines a different language. However, in some cases only minor typographical changes are sufficient to convert an EBNF grammar into its PEG parser. As recently shown by Medeiros, this is, in particular, true for LL(1) grammars. But this is also true for many non-LL(1) grammars, which is interesting because the backtracking of PEG is often a convenient way to circumvent just the LL(1) restriction. We formulate a number of conditions for EBNF grammar to become its own PEG parser, and arrive at a condition that we call LL(1p), meaning that a top-down parser can choose its next action by looking at the input within the reach of one parsing procedure (rather than by looking at the next letter). An extension to LL(kp) for $k > 1$ seems possible.

1 Introduction

Parsing Expression Grammar (PEG), as introduced by Ford in [4], is a recursive-descent parser with limited backtracking, written in a special language. Using the "memoization" or "packrat" technology described in [2, 3], it works in linear time. The parser does not require a separate "lexer" to preprocess the input, and the limited backtracking lifts the LL(1) restriction usually imposed by top-down parsers. These properties are useful in many applications. However, PEG is not well understood as a language specification tool. In [7], the author tried to find the language defined by a given parsing expression. Unfortunately, it was only possible to obtain some approximations that became more and more difficult as the complexity of the expression increased.

In [5], Medeiros used the opposite approach: given language, construct a PEG parser for it. He constructed PEGs for languages defined by regular expressions, as well as right-linear, LL(1) and LL(k) context-free grammars. The construction for LL(1) grammars is very simple: just replace the unordered choice "`|`" by the ordered choice "`/`". Unfortunately, this is not of much use when we employ PEG just in order to circumvent the LL(1) restriction. But it turns out that this property is not limited to LL(1) grammars. As an example, take the following EBNF grammar:

Literal = *Hex* `|` *Bin*
Hex = `[0-9a-f]`$^+$ `"X"`
Bin = `[01]`$^+$ `"B"`

This grammar is not LL(1), and not even LL(k) for any k: both *Hex* and *Bin* may start with any number of zeros and ones. A classical top-down parser constructed from this grammar cannot choose between the two alternatives of *Literal* by looking at any predefined number of characters ahead. But, treated as PEG with `"|"` denoting the ordered choice, this grammar recognizes exactly the language defined by its EBNF interpretation. Presented with input such as "`1010B`", *Literal* calls *Hex* that consumes "`1010`", fails to recognize `"X"`, and backtracks. *Literal* proceeds then to try *Bin* and succeeds.

In the following, we attempt to answer this question: *given an EBNF grammar, when can we transcribe it directly into a PEG that recognizes the same language, and parses it in the same way?*

To find the answer we introduce, in Section 3, a simple grammar that can be interpreted as either EBNF or PEG. Following [5], we formally define these interpretations using the method of "natural semantics".

*Final version appeared in *Fundamenta Informaticae 128*, 1-2 (2013) 177–191.

The grammar is reduced to bare bones in order to simplify the reasoning. Any full EBNF grammar or PEG without predicates can be reduced to such form by removing syntactic sugar and introducing new nonterminals. By reversing this process, one can apply our results to a full grammar.

In Section 4, we identify a sufficient (but not necessary!) condition for the two interpretations to be equivalent, in the sense of accepting the same input and producing the same parse trees. This condition happens to coincide with the "general semi disjointness" introduced by Schmitz in [8]. Unfortunately, it is difficult (if not impossible) to check this condition in a mechanical way.

After some preliminaries in Section 5, we use, in Section 6, our result to formulate two stronger conditions based on prefixes, and in Section 7, a condition using "first expressions". This leads in Section 8 to the notion of LL(1p) grammars. Unfortunately, all these conditions are not as simple to verify as LL(1).

In the final Section 9 we discuss further development, and indicate the possibility of using our results to insert the "cut" operation introduced by Mizushima et al. in [6].

All lengthy proofs are found in the Appendix.

2 Some Notation

We consider a finite alphabet Σ of *letters*. A finite string of letters is a *word*. The string of 0 letters is called the *empty word* and is denoted by ε. The set of all words is Σ^*. A subset of Σ^* is a *language*.

As usual, we write XY to mean the concatenation of languages X and Y, that is, the set of all words xy where $x \in X$ and $y \in Y$.

For $x, y \in \Sigma^*$, x is a *prefix* of y if $y = xu$ for some $u \in \Sigma^*$. We write $x \leq y$ to mean that x is a prefix of y. For $X \subseteq \Sigma^*$, the set of all prefixes of $x \in X$ is denoted by $\mathrm{Pref}(X)$.

A relation R on $\mathbb{E}$ is a subset of $\mathbb{E} \times \mathbb{E}$. As usual, we write $R(e)$ to mean the set of all e' such that $(e, e') \in R$, and $R(E)$ to mean the union of $R(e)$ for all $e \in E \subseteq \mathbb{E}$. The transitive closure of R is denoted by R^+, and the product of relations R and S by $R \times S$.

3 The Grammar

We consider a grammar $\mathbb{G}$ over the alphabet Σ that will be interpreted as either PEG or EBNF. The grammar consists of a finite nonempty set N of symbols distinct from the letters of Σ, and a set of rules, one for each symbol of N. The rule for $A \in N$ has one of these two forms:

- $A = e_1 e_2$,
- $A = e_1 | e_2$,

where each of e_1, e_2 is an element of N, or a letter from Σ, or the symbol $\boldsymbol{\varepsilon}$. (The boldface $\boldsymbol{\varepsilon}$ here is a special symbol, to be distinguished from the empty word ε.) We define $\mathbb{E} = N \cup \Sigma \cup \{\boldsymbol{\varepsilon}\}$.

We read the rule $A = e$ (where e is either $e_1 e_2$ or $e_1 | e_2$) as saying "A is the name of expression e". The grammar is referred to as $\boldsymbol{\varepsilon}$*-free* if none of the rules contains $\boldsymbol{\varepsilon}$.

When $\mathbb{G}$ is interpreted as EBNF, the rule $A = e$ is a definition sometimes written as $A ::= e$, and $e_1 | e_2$ is the unordered choice. When $\mathbb{G}$ is interpreted as PEG, the rule is a definition of parsing expression, written as $A \leftarrow e$ in [4], and $e_1 | e_2$ is the ordered choice. In each case, the elements of Σ are the "terminals" of the grammar, and the elements of N are the "nonterminals".

3.1 The EBNF Interpretation

The grammar $\mathbb{G}$ interpreted as EBNF is a mechanism for generating words:

- $\boldsymbol{\varepsilon}$ generates ε.
- $a \in \Sigma$ generates itself.
- $A = e_1 e_2$ generates any word generated by e_1 followed by any word generated by e_2.
- $A = e_1 | e_2$ generates any word generated by e_1 or e_2.

The language $\mathcal{L}(e)$ of $e \in \mathbb{E}$ is the set of all words that can be generated by e.

Following [5], we use the method of "natural semantics" for an alternative definition of $\mathcal{L}(e)$: we introduce a relation $\overset{\text{BNF}}{\rightsquigarrow} \subseteq (\mathbb{E} \times \Sigma^*) \times \Sigma^*$, writing $[e]\ w \overset{\text{BNF}}{\rightsquigarrow} y$ to mean that the relation holds for $e \in \mathbb{E}$

and $w, y \in \Sigma^*$. The intended meaning of $[e]\ w \overset{\text{BNF}}{\rightsquigarrow} y$ is that w has a prefix belonging to $\mathcal{L}(e)$. Formally, the relation is defined to hold if and only if this can be proved using the inference rules in Figure 1. An example of such proof is shown in Figure 2.

$$\frac{}{[\boldsymbol{\varepsilon}]\ x \overset{\text{BNF}}{\rightsquigarrow} x}\ \textbf{(empty.b)} \qquad \frac{}{[a]\ ax \overset{\text{BNF}}{\rightsquigarrow} x}\ \textbf{(letter.b)}$$

$$\frac{A = e_1 e_2 \quad [e_1]\ xyz \overset{\text{BNF}}{\rightsquigarrow} yz \quad [e_2]\ yz \overset{\text{BNF}}{\rightsquigarrow} z}{[A]\ xyz \overset{\text{BNF}}{\rightsquigarrow} z}\ \textbf{(seq.b)}$$

$$\frac{A = e_1|e_2 \quad [e_1]\ xy \overset{\text{BNF}}{\rightsquigarrow} y}{[A]\ xy \overset{\text{BNF}}{\rightsquigarrow} y}\ \textbf{(choice.b1)} \qquad \frac{A = e_1|e_2 \quad [e_2]\ xy \overset{\text{BNF}}{\rightsquigarrow} y}{[A]\ xy \overset{\text{BNF}}{\rightsquigarrow} y}\ \textbf{(choice.b2)}$$

Figure 1: EBNF semantics

One can verify by structural induction that, indeed, for any $y \in \Sigma$, we have $[e]\ xy \overset{\text{BNF}}{\rightsquigarrow} y$ if and only if $x \in \mathcal{L}(e)$. This gives, in particular, $\mathcal{L}(e) = \{x \in \Sigma^* : [e]\ x \overset{\text{BNF}}{\rightsquigarrow} \varepsilon\}$.

We note that the proof tree of $[e]\ x \overset{\text{BNF}}{\rightsquigarrow} \varepsilon$, when followed from the bottom up, mimics the behavior of a recursive-descent parser accepting the string x by always choosing the correct alternative of $e_1|e_2$. The proof tree is thus isomorphic to the parse tree of x: a tree showing how x was generated according to the rules of $\mathbb{G}$.

$$\frac{S = aX \qquad \dfrac{}{[a]\ ab \overset{\text{BNF}}{\rightsquigarrow} b} \qquad \dfrac{X = S|b \qquad \dfrac{}{[b]\ b \overset{\text{BNF}}{\rightsquigarrow} \varepsilon}}{[X]\ b \overset{\text{BNF}}{\rightsquigarrow} \varepsilon}}{[S]\ ab \overset{\text{BNF}}{\rightsquigarrow} \varepsilon}$$

Figure 2: An example of proof using EBNF semantics for grammar $S = aX$, $X = S \,|\, b$

As it will be seen later on, some results depend on whether $\varepsilon \in \mathcal{L}(e)$ for given $e \in \mathbb{E}$. This can be decided using a simple iterative algorithm, independent of the above formal definition. Obviously, $\varepsilon \notin \mathcal{L}(e)$ for all e if the grammar is $\boldsymbol{\varepsilon}$-free.

3.2 The PEG Interpretation

When the grammar $\mathbb{G}$ is interpreted as PEG, the elements of $\mathbb{E}$ are parsing procedures that can call each other recursively. In general, parsing procedure is applied to a word from Σ^* and tries to recognize an initial portion of that word. If it succeeds, it "consumes" the recognized portion and returns "success"; otherwise, it returns "failure" and does not consume anything. The action of different procedures is as follows:

- $\boldsymbol{\varepsilon}$: Indicate success without consuming any input.
- $a \in \Sigma$: If the text ahead starts with a, consume it and return success. Otherwise return failure.
- $A = e_1\, e_2$: Call e_1. If it succeeded, call e_2 and return success if e_2 succeeded.
  If e_1 or e_2 failed, backtrack: reset the input as it was before the invocation of e_1 and return failure.
- $A = e_1|e_2$: Call e_1. Return success if it succeeded.
  Otherwise call expression e_2 and return success if e_2 succeeded or failure if it failed.

Note the limited backtracking: $A = e_1|e_2$ has never a chance to try e_2 once e_1 succeeded.

Following [5], we use natural semantic to formally define the action of a parsing procedure: we introduce a relation $\overset{\text{PEG}}{\rightsquigarrow} \subseteq (\mathbb{E} \times \Sigma^*) \times \{\Sigma^* \cup \text{fail}\}$ and write $[e]\ w \overset{\text{PEG}}{\rightsquigarrow} X$ to mean that the relation holds for $e \in \mathbb{E}$, $w \in \Sigma^*$, and $X \in \{\Sigma^* \cup \text{fail}\}$. The intended meaning of $[e]\ w \overset{\text{PEG}}{\rightsquigarrow} X$ is:

- $[e]\ xy \overset{\text{PEG}}{\rightsquigarrow} y$ means: "e applied to input xy consumes x and returns success",
- $[e]\ x \overset{\text{PEG}}{\rightsquigarrow} \text{fail}$ means: "e applied to input x returns failure".

Formally, the relation is defined to hold if and only if this can be proved using the inference rules in Figure 3.

One can easily see that the rules correctly represent the action of parsing procedures in the intended way. For example, **seq.p1** with $Z = \text{fail}$ represents backtracking in $A = e_1e_2$, and **choice.p2** represents $A = e_1|e_2$ calling e_2 after e_1 failed.

The proof tree of $[e]\ x \overset{\text{PEG}}{\rightsquigarrow} X$, when followed from bottom up, mimics procedure calls in the process of parsing the string x. The proof tree of $[e]\ xy \overset{\text{PEG}}{\rightsquigarrow} y$ represents a successful parse of xy, and can be regarded as the parse tree of x.

$$\frac{}{[\varepsilon]\ x \overset{\text{PEG}}{\rightsquigarrow} x}\ \textbf{(empty.p)}$$

$$\frac{}{[a]\ ax \overset{\text{PEG}}{\rightsquigarrow} x}\ \textbf{(letter.p1)} \qquad \frac{b \neq a}{[b]\ ax \overset{\text{PEG}}{\rightsquigarrow} \text{fail}}\ \textbf{(letter.p2)} \qquad \frac{}{[a]\ \varepsilon \overset{\text{PEG}}{\rightsquigarrow} \text{fail}}\ \textbf{(letter.p3)}$$

$$\frac{A = e_1e_2 \quad [e_1]\ xyz \overset{\text{PEG}}{\rightsquigarrow} yz \quad [e_2]\ yz \overset{\text{PEG}}{\rightsquigarrow} Z}{[A]\ xyz \overset{\text{PEG}}{\rightsquigarrow} Z}\ \textbf{(seq.p1)} \qquad \frac{A = e_1e_2 \quad [e_1]\ x \overset{\text{PEG}}{\rightsquigarrow} \text{fail}}{[A]\ x \overset{\text{PEG}}{\rightsquigarrow} \text{fail}}\ \textbf{(seq.p2)}$$

$$\frac{A = e_1|e_2 \quad [e_1]\ xy \overset{\text{PEG}}{\rightsquigarrow} y}{[A]\ xy \overset{\text{PEG}}{\rightsquigarrow} y}\ \textbf{(choice.p1)} \qquad \frac{A = e_1|e_2 \quad [e_1]\ x \overset{\text{PEG}}{\rightsquigarrow} \text{fail} \quad [e_2]\ xy \overset{\text{PEG}}{\rightsquigarrow} Y}{[A]\ xy \overset{\text{PEG}}{\rightsquigarrow} Y}\ \textbf{(choice.p2)}$$

where Y denotes y or fail and Z denotes z or fail.

Figure 3: EBNF semantics

4 When Are the Two Interpretations Equivalent?

The following has been proved as Lemma 4.3.1 in [5]:

Proposition 1. *For any $e \in \mathbb{E}$ and $x, y \in \Sigma^*$, $[e]\ xy \overset{PEG}{\rightsquigarrow} y$ implies $[e]\ xy \overset{BNF}{\rightsquigarrow} y$.*
(Proof is found in the Appendix.)

Known examples show that the reverse is, in general, not true. One case is when applying procedure e to a string x results in an infinite descent of recursive calls, so that the process never terminates. In this case, there is no proof of $[e]\ x \overset{\text{PEG}}{\rightsquigarrow} X$.

To analyze this situation, we need the relation `first` on $\mathbb{E}$ (that will also be used later on). We define $\texttt{first}(e)$ as the set of parsing procedures that may be called by e to process the start of string x after e was called to process x:

- $\texttt{first}(\varepsilon) = \varnothing$.
- For $a \in \Sigma$, $\texttt{first}(a) = \varnothing$.
- For $A \in N$ where $A = e_1e_2$ and $\varepsilon \notin \mathcal{L}(e_1)$, $\texttt{first}(A) = \{e_1\}$.
- For $A \in N$ where $A = e_1e_2$ and $\varepsilon \in \mathcal{L}(e_1)$, $\texttt{first}(A) = \{e_1, e_2\}$.
- For $A \in N$ where $A = e_1|e_2$, $\texttt{first}(A) = \{e_1, e_2\}$.

This relation can be represented by a directed graph, such as in Figure 4. An arrow from e to e' means $e' \in \texttt{first}(e')$. Define now $\texttt{First} = \texttt{first}^+$, meaning that $\texttt{First}(e)$ is the set of procedures that may be called directly or indirectly by e to process the start of string x. In the graph, $e' \in \texttt{First}(e)$ is represented by a path from e to e'.

The grammar is *left-recursive* if $e \in \texttt{First}(e)$ for some $e \in \mathbb{E}$. If the grammar is not left-recursive, the graph representing the relation is acyclic, which is the case in Figure 4. The important fact is:

Proposition 2. *If the grammar $\mathbb{G}$ is not left-recursive then for every $e \in \mathbb{E}$ and $w \in \Sigma^*$ there exists a proof of either $[e]\ w \overset{PEG}{\rightsquigarrow} \text{fail}$ or $[e]\ w \overset{PEG}{\rightsquigarrow} y$ where $w = xy$.*

In [5], Medeiros shows this by first verifying that his natural semantics is equivalent to the formal definition from [4], and then using a result about "well-formed" grammars from the same paper. A more direct proof is found in the Appendix.

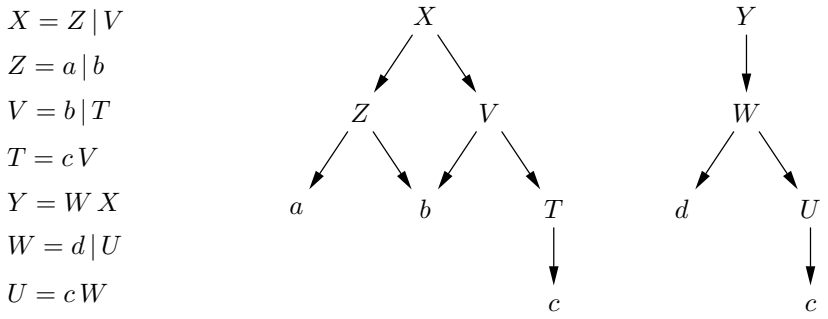

Figure 4: Example of relation `first`

Another problem besides left recursion is the known fact that $[e]\ xy \overset{\text{PEG}}{\rightsquigarrow} y$ does not imply $[e]\ xy' \overset{\text{PEG}}{\rightsquigarrow} y'$ for every $y' \in \Sigma^*$: the action of e depends on the string ahead. We have to consider the action of e in relation to the entire input string.

Let \$, the end-of-text marker, be a letter that does not appear in any rules. Define some $S \in N$ as the starting rule of the grammar. As $\mathcal{L}(e)$ does not depend on the input ahead, we have $[S]\ w\$ \overset{\text{BNF}}{\rightsquigarrow} \$$ if and only if $w \in \mathcal{L}(S)$. We note that every partial result in the proof tree of $[S]\ w\$ \overset{\text{BNF}}{\rightsquigarrow} \$$ has the form $[e]\ xy\$ \overset{\text{BNF}}{\rightsquigarrow} y\$$ for some $e \in \mathbb{E}$ and $x, y \in \Sigma^*$.

For $A \in N$, define $\text{Tail}(A)$ to be the set of terminated input strings that may follow a substring belonging to $\mathcal{L}(A)$. Formally, define $y\$ \in \text{Tail}(A)$ if there exists a proof tree for $[S]\ w\$ \overset{\text{BNF}}{\rightsquigarrow} \$$ that contains $[A]\ xy\$ \overset{\text{BNF}}{\rightsquigarrow} y\$$ as a partial result.

The following extends the results from [5] to non-LL(1) grammars:

Proposition 3. *If the grammar $\mathbb{G}$ is not left-recursive and its every rule $A = e_1|e_2$ satisfies*

$$\mathcal{L}(e_1) \cap \text{Pref}(\mathcal{L}(e_2)\,\text{Tail}(A)) = \varnothing, \tag{1}$$

there exists a proof of $[S]\ w\$ \overset{PEG}{\rightsquigarrow} \$$ for each $w \in \mathcal{L}(S)$. Moreover, for every partial result $[e]\ xy\$ \overset{BNF}{\rightsquigarrow} y\$$ in the proof tree of $[S]\ w\$ \overset{BNF}{\rightsquigarrow} \$$ there exists a proof of $[e]\ xy\$ \overset{PEG}{\rightsquigarrow} y\$$.

(Proof of the Proposition is found in the Appendix.)

The consequence of Propositions 1 and 3 is:

Proposition 4. *The two interpretations of $\mathbb{G}$ are equivalent if $\mathbb{G}$ is not left-recursive and its every rule $A = e_1|e_2$ satisfies (1). The two interpretations are equivalent not only by accepting the same language, but also by parsing the input in the same way.*

Unfortunately, (1) is not easy to check. The languages appearing in it are context-free languages. The problems of inclusion and emptiness of intersection are in general undecidable for these languages, so one has to consider special cases, or use a conservative approximation. In fact, condition (1) is identical to the "general semi disjointness" considered by Schmitz in [8]; he checks it using an own grammar approximation method.

No need to add that the condition stated by Proposition 4 is sufficient, but not necessary.

As it will be seen, the LL(1) property implies (1), which accounts for the result obtained in [5]. We are going to look for a weaker property that still implies (1). One can think of the LL(1) property as approximating words by their one-letter prefixes. We generalize this to approximating words by prefixes of arbitrary lengths.

5 Prefix Cover and Prefix Disjointness

We say that $Y \subseteq \Sigma^*$ is a *prefix cover* of $X \subseteq \Sigma^*$, and write $Y \sqsubseteq X$, if each nonempty word in X has a nonempty prefix in Y, that is, $(X - \varepsilon) \subseteq (Y - \varepsilon)\Sigma^*$.

We say that $X \subseteq \Sigma^*$ and $Y \subseteq \Sigma^*$ are *prefix-disjoint*, and write $X \asymp Y$ to mean that for all nonempty $x \in X$ and $y \in Y$ neither $x \leq y$ nor $y \leq x$. One can easily see that prefix-disjoint languages are disjoint, but the reverse is in general not true.

Lemma 1. *For any X, Y, $Z \in \Sigma^*$ and $\mathcal{P}_X \sqsubseteq X$, $\mathcal{P}_Y \sqsubseteq Y$:*
(a) $X \sqsubseteq X$.
(b) If $X \sqsubseteq Y$ and $Y \sqsubseteq Z$ then $X \sqsubseteq Z$.
(c) $\mathcal{P}_X \cup \mathcal{P}_Y \sqsubseteq X \cup Y$.
(d) $\mathcal{P}_X \sqsubseteq XY$.
(e) If $\varepsilon \in X$ then $\mathcal{P}_X \cup \mathcal{P}_Y \sqsubseteq XY$.

Proof. (a) Follows from $X \subseteq \mathrm{Pref}(X)$.
(b) Follows from transitivity of $\leq$.
(c) If a nonempty $x \in X \cup Y$ is in X, it has a nonempty prefix in $\mathcal{P}_X$. If it is in Y, it has a nonempty prefix in $\mathcal{P}_Y$.
(d) If $\varepsilon \notin X$, each $x \in XY$ has a nonempty prefix in X. This, in turn, has a nonempty prefix in $\mathcal{P}_X$.
(e) If $\varepsilon \in X$, we have $XY = (X - \varepsilon)Y \cup Y$. From (d) we have $\mathcal{P}_X \sqsubseteq (X - \varepsilon)Y$. The stated property follows from (c).

□

6 Approximation by Prefixes

We can now apply the above to approximate (1). In order not to repeat this over and over again, we assume from now on that our grammar $\mathbb{G}$ is not left-recursive.

Proposition 5. *The two interpretations of $\mathbb{G}$ are equivalent if for every rule $A = e_1|e_2$:*
- *$\varepsilon \notin \mathcal{L}(e_1)$, and*
- *there exist $\mathcal{P}_1 \sqsubseteq \mathcal{L}(e_1)$ and $\mathcal{P}_2 \sqsubseteq \mathcal{L}(e_2)\,\mathrm{Tail}(A)$ such that $\mathcal{P}_1 \asymp \mathcal{P}_2$.*

Proof. Consider any $A = e_1|e_2$. Assume $\varepsilon \notin \mathcal{L}(e_1)$ and $\mathcal{P}_1$, $\mathcal{P}_2$ as required.
Suppose that (1) does not hold, so there exists x such that $x \in \mathcal{L}(e_1)$ and $x \in \mathrm{Pref}(\mathcal{L}(e_2)\,\mathrm{Tail}(A))$. Since $x \neq \varepsilon$, we have $x = uv$ for some nonempty $u \in \mathcal{P}_1$ and $v \in \Sigma^*$. As $x \in \mathrm{Pref}(\mathcal{L}(e_2)\,\mathrm{Tail}(A))$, we have $xt \in \mathcal{L}(e_2)\,\mathrm{Tail}(A)$ for some $t \in \Sigma^*$. As $xt \neq \varepsilon$, we have $xt = pq$ for some nonempty $p \in \mathcal{P}_2$ and $q \in \Sigma^*$. This gives $uvt = pq$; this means either $u \leq p$ or $p \leq u$, which contradicts $\mathcal{P}_1 \asymp \mathcal{P}_2$. Thus, (1) does hold, and the two interpretations are equivalent by Proposition 4. □

The following is a stronger, but easier to check, condition for equivalence:

Proposition 6. *The two interpretations of $\mathbb{G}$ are equivalent if for every rule $A = e_1|e_2$:*
- *$\varepsilon \notin \mathcal{L}(e_1)$, and*
- *if $\varepsilon \notin \mathcal{L}(e_2)$, there exist $\mathcal{P}_1 \sqsubseteq \mathcal{L}(e_1)$ and $\mathcal{P}_2 \sqsubseteq \mathcal{L}(e_2)$ such that $\mathcal{P}_1 \asymp \mathcal{P}_2$;*
- *if $\varepsilon \in \mathcal{L}(e_2)$, there exist $\mathcal{P}_1 \sqsubseteq \mathcal{L}(e_1)$, $\mathcal{P}_2 \sqsubseteq \mathcal{L}(e_2)$, and $\mathcal{P}_T \sqsubseteq \mathrm{Tail}(A)$ such that $\mathcal{P}_1 \asymp (\mathcal{P}_2 \cup \mathcal{P}_T)$.*

Proof. Consider any $A = e_1|e_2$. Assume $\varepsilon \notin \mathcal{L}(e_1)$, $\mathcal{P}_1 \sqsubseteq \mathcal{L}(e_1)$ and $\mathcal{P}_2 \sqsubseteq \mathcal{L}(e_2)$. Suppose $\varepsilon \notin \mathcal{L}(e_2)$. By Lemma 1(d), $\mathcal{P}_2 \sqsubseteq \mathcal{L}(e_2)\,\mathrm{Tail}(A)$, so $\mathcal{P}_1 \asymp \mathcal{P}_2$ satisfies Proposition 5. Suppose now that $\varepsilon \in \mathcal{L}(e_2)$ and $\mathcal{P}_T \sqsubseteq \mathrm{Tail}(A)$. By Lemma 1(e), $\mathcal{P}_2 \cup \mathcal{P}_T \sqsubseteq \mathcal{L}(e_2)\,\mathrm{Tail}(A)$, so $\mathcal{P}_1 \asymp \mathcal{P}_2 \cup \mathcal{P}_T$ satisfies Proposition 5. □

We note once again that the equivalence conditions just obtained are sufficient, but not necessary.

7 Approximation by First Expressions

For $E \subseteq \mathbb{E}$, let $\mathcal{FIRST}(E)$ be the family of subsets of $\mathbb{E}$ defined as follows:

- $E \in \mathcal{FIRST}(E)$.
- If F belongs to $\mathcal{FIRST}(E)$, the result of replacing its member $A \in N$ by all elements of $\texttt{first}(A)$ also belongs to $\mathcal{FIRST}(E)$.
- nothing else belongs to $\mathcal{FIRST}(E)$ unless its being so follows from the above.

Two examples for the grammar of Figure 4 are:

$$\mathcal{FIRST}(X) = \{\{X\}, \{Z, V\}, \{a, b, V\}, \{Z, b, T\}, \{a, b, T\}, \{a, b, c\}\};$$
$$\mathcal{FIRST}(Y) = \{\{Y\}, \{W\}, \{d, U\}, \{d, c\}\}.$$

In the Figure, one can see that, in general, the elements of $\mathcal{FIRST}(A)$ are leaves of a truncated graph of $\texttt{First}(A)$ obtained by treating some nodes as terminals. Figure 5 shows the graphs from Figure 4 truncated at T and U, giving the elements $\{a, b, T\} \in \mathcal{FIRST}(X)$ and $\{d, U\} \in \mathcal{FIRST}(Y)$.

$X = Z \,|\, V$
$Z = a \,|\, b$
$V = b \,|\, T$
$T = c\, V$
$Y = W\, X$
$W = d \,|\, U$
$U = c\, W$

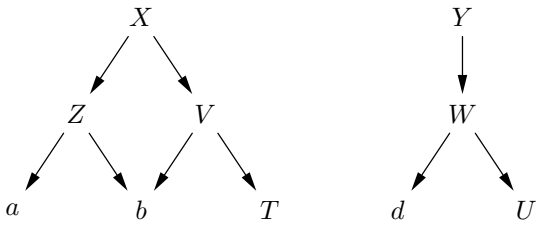

Figure 5: Example of truncated relation $\texttt{First}$

Note that $\mathcal{FIRST}(A)$ always contains a set consisting of all leaves of the full graph. This set is identical to the classical $\textsc{First}(A)$ as defined in, for example, [1, 9].

The idea with a set E of expressions belonging to $\mathcal{FIRST}(A)$ is that any nonempty word in $\mathcal{L}(A)$ must begin with one of these expressions or, more precisely, with a nonempty string belonging to the language of such expression. We are now going to formalize this idea. For $E \subseteq \mathbb{E}$, we denote the union $\bigcup_{e \in E} \mathcal{L}(e)$ by $\mathcal{L}(E)$.

Lemma 2. *For each $A \in N$ holds $\mathcal{L}(\texttt{first}(A)) \sqsubseteq \mathcal{L}(A)$.*

Proof. In case $A = e_1 | e_2$ we have $\mathcal{L}(\texttt{first}(A)) = \mathcal{L}\{e_1, e_2\} = \mathcal{L}(e_1) \cup \mathcal{L}(e_2) = \mathcal{L}(A)$. By Lemma 1(a), $\mathcal{L}(\texttt{first}(A)) \sqsubseteq \mathcal{L}(A)$.
In case $A = e_1\, e_2$ where $\varepsilon \notin e_1$, we have $\mathcal{L}(\texttt{first}(A)) = \mathcal{L}(e_1)$. By Lemma 1(a), $\mathcal{L}(e_1) \sqsubseteq \mathcal{L}(e_1)$. By Lemma 1(d), $\mathcal{L}(e_1) \sqsubseteq \mathcal{L}(e_1)\mathcal{L}(e_2) = \mathcal{L}(A)$.
In case $A = e_1\, e_2$ where $\varepsilon \in e_1$, $\mathcal{L}(\texttt{first}(A)) = \mathcal{L}\{e_1, e_2\} = \mathcal{L}(e_1) \cup \mathcal{L}(e_2)$. By Lemma 1(a), $\mathcal{L}(e_1) \sqsubseteq \mathcal{L}(e_1)$ and $\mathcal{L}(e_2) \sqsubseteq \mathcal{L}(e_2)$. By Lemma 1(e), $\mathcal{L}(e_1) \cup \mathcal{L}(e_2) \sqsubseteq \mathcal{L}(e_1)\mathcal{L}(e_2) = \mathcal{L}(A)$. □

Lemma 3. *For each $F \in \mathcal{FIRST}(E)$ holds $\mathcal{L}(F) \sqsubseteq \mathcal{L}(E)$.*

Proof. The proof is by induction on the number of replacements used to construct $\mathcal{FIRST}(E)$.
(Induction base) By Lemma 1(a), $\mathcal{L}(E) \sqsubseteq \mathcal{L}(E)$.
(Induction step) Assume $\mathcal{L}\{e_1, e_2, \ldots, e_n\} \sqsubseteq \mathcal{L}(E)$. As sets are not ordered, we can always consider replacing e_1. Let $\texttt{first}(e_1) = \{f_1, f_2\}$. (The proof will be similar if $\texttt{first}(e_1)$ has only one element.) According to Lemma 2, $\mathcal{L}\{f_1, f_2\} \sqsubseteq \mathcal{L}(e_1)$. We also have $\mathcal{L}\{e_2, \ldots, e_n\} \sqsubseteq \mathcal{L}\{e_2, \ldots, e_n\}$ by Lemma 1(a). Using Lemma 1(c), we obtain:

$$\mathcal{L}\{f_1, f_2, e_2, \ldots, e_n\} = \mathcal{L}\{f_1, f_2\} \cup \mathcal{L}\{e_2, \ldots, e_n\} \sqsubseteq \mathcal{L}(e_1) \cup \mathcal{L}\{e_2, \ldots, e_n\} = \mathcal{L}\{e_1, e_2, \ldots, e_n\}.$$

From Lemma 1(b) follows $\mathcal{L}\{f_1, f_2, e_2, \ldots, e_n\} \sqsubseteq \mathcal{L}(E)$. □

We can apply this result directly to ε-free grammars:

Proposition 7. *The two interpretations of* $\mathbb{G}$ *are equivalent if* $\mathbb{G}$ *is* ε*-free and for every rule* $A = e_1|e_2$ *there exist* $\textsc{First}_1 \in \mathcal{FIRST}(e_1)$ *and* $\textsc{First}_2 \in \mathcal{FIRST}(e_2)$ *such that* $\mathcal{L}(\textsc{First}_1) \asymp \mathcal{L}(\textsc{First}_2)$.

Proof. Suppose the required $\textsc{First}_1$ and $\textsc{First}_2$ exist. If $\mathbb{G}$ is ε-free, we have $\varepsilon \notin \mathcal{L}(e_1)$ and $\varepsilon \notin \mathcal{L}(e_2)$. From Lemma 3 follows $\mathcal{L}(\textsc{First}_1) \sqsubseteq \mathcal{L}(e_1)$ and $\mathcal{L}(\textsc{First}_2) \sqsubseteq \mathcal{L}(e_2)$. The stated result follows from Proposition 6. □

To handle the case where the grammar $\mathbb{G}$ is not ε-free, we need to find suitable first expressions of Tail(A). For this purpose, we borrow from [9] the definition of $\texttt{Follow} \subseteq \mathbb{E} \times \mathbb{E}$.

For $e \in \mathbb{E}$, define $\texttt{Last}(e)$ to be the set of all $A \in N$ such that:

- $A = e|e_1$ for some $e_1 \in \mathbb{E}$, or
- $A = e_1|e$ for some $e_1 \in \mathbb{E}$, or
- $A = e_1\, e$ for some $e_1 \in \mathbb{E}$, or
- $A = e\, e_1$ for some $e_1 \in \mathbb{E}$ where $\varepsilon \in \mathcal{L}(e_1)$.

For $e \in \mathbb{E}$, define $\texttt{Next}(e) = \{e_1 \in \mathbb{E} :$ exists $A \in N$such that $A = e\, e_1\}$.
Finally, define $\texttt{Follow} = \texttt{Last}^* \times \texttt{Next}$.

The idea with `Follow` is that any nonempty word in Tail(A) must begin with one of the expressions in $\texttt{Follow}(A)$ or, more precisely, with a nonempty string belonging to the language of such expression. The following is a formal statement of this fact.

Lemma 4. *For each* $A \in N$, $\mathcal{L}(\texttt{Follow}(A)) \sqsubseteq \text{Tail}(A)$.
(Proof is found in the Appendix.)

We can now apply this result to the general case.

Proposition 8. *The two interpretations of* $\mathbb{G}$ *are equivalent if for every rule* $A = e_1|e_2$*:*

- $\varepsilon \notin \mathcal{L}(e_1)$*, and*
- *if* $\varepsilon \notin \mathcal{L}(e_2)$*, there exist* $\textsc{First}_1 \in \mathcal{FIRST}(e_1)$ *and* $\textsc{First}_2 \in \mathcal{FIRST}(e_2)$ *such that* $\mathcal{L}(\textsc{First}_1) \asymp \mathcal{L}(\textsc{First}_2)$*;*
- *if* $\varepsilon \in \mathcal{L}(e_2)$*, there exist* $\textsc{First}_1 \in \mathcal{FIRST}(e_1)$, $\textsc{First}_2 \in \mathcal{FIRST}(e_2)$, *and* $\textsc{Follow} \in \mathcal{FIRST}(\texttt{Follow}(A))$ *such that* $\mathcal{L}(\textsc{First}_1) \asymp \mathcal{L}(\textsc{First}_2 \cup \textsc{Follow})$.

Proof. The case of $\varepsilon \notin \mathcal{L}(e_2)$ is identical to Proposition 7.
Suppose $\varepsilon \in \mathcal{L}(e_2)$ and the required $\textsc{First}_1$, $\textsc{First}_2$, and $\textsc{Follow}$ exist. From Lemma 3 we have $\mathcal{L}(\textsc{First}_1) \sqsubseteq \mathcal{L}(e_1)$, $\mathcal{L}(\textsc{First}_2) \sqsubseteq \mathcal{L}(e_2)$, and $\mathcal{L}(\textsc{Follow}) \sqsubseteq \mathcal{L}(\texttt{Follow}(A))$. From Lemma 4 and 1(b) we have $\mathcal{L}(\textsc{Follow}) \sqsubseteq \text{Tail}(A)$. The stated result follows from Proposition 6. □

8 LL(1p) Grammars

As remarked before, $\mathcal{FIRST}(e)$ contains the classical $\textsc{First}(e)$: the set of possible starting letters of words in $\mathcal{L}(e)$. Similarly, $\mathcal{FIRST}(\texttt{Follow}(A))$ contains the classical $\textsc{Follow}(A)$: the set of possible starting letters of words in Tail(A). For sets $X, Y \subseteq \Sigma$, $X \asymp Y$ is the same as $X \cap Y = \varnothing$. The LL(1) conditions $\textsc{First}(e_1) \cap \textsc{First}(e_2) = \varnothing$ and $\textsc{First}(e_1) \cap (\textsc{First}(e_2) \cup \textsc{Follow}(A)) = \varnothing$ are thus special cases of those stated by Proposition 8, meaning that the two interpretations are equivalent for LL(1) grammars. This is the result obtained by Medeiros in [5].

For an example of a non-LL(1) grammar, take the grammar from Figures 4 and 5. Consider an additional rule $S = X|Y$. This rule is not LL(1), as the intersection of $\textsc{First}(X) = \{a, b, c\}$ and $\textsc{First}(Y) = \{c, d\}$ is not empty.

Instead of $\{a, b, c\}$ and $\{c, d\}$, consider now $\{a, b, T\} \in \mathcal{FIRST}(X)$ and $\{d, U\} \in \mathcal{FIRST}(Y)$. One can verify that $\mathcal{L}(T) = cc^*b$ and $\mathcal{L}(U) = cc^*d$, so $\mathcal{L}\{a, b, T\} = a \cup c^*b$ and $\mathcal{L}\{d, U\} = c^*d$. One can also see that $(a \cup c^*b) \asymp (c^*d)$; thus, according to Proposition 7, $S = X|Y$ satisfies the equivalence condition (1).

We have here a rule that is not LL(1), but still satisfies (1). Imagine now for a while that T and U are terminals – elements of an extended alphabet Σ. In a practical parser, they could be tokens produced by some preprocessor – usually known as a "lexer" or "scanner". Because $\{a, b, T\} \cap \{d, U\} = \varnothing$, the rule

$S = X|Y$ is now perfectly LL(1). A top-down parser can choose between X and Y by looking at the next token.

In a backtracking parser, procedures such as those for T or U act in the same way as those for terminals. In each case, the procedure looks at the input ahead, and either succeeds and consumes a portion, or fails and leaves the state unchanged. The only difference is the amount of the input being examined. The condition such as $\{a, b, T\} \asymp \{d, U\}$ tells that the parser can choose between X and Y by looking at the input within the range of one procedure call.

We can call this condition "LL(1p)", with "1p" standing for "one procedure". Proposition 8 can now be understood as stating that the two interpretations of $\mathbb{G}$ are equivalent if the grammar is LL(1p). Note that the grammar shown in the Introduction is LL(1p).

9 Conclusions

We have shown that a direct transcription of an EBNF grammar to equivalent PEG is possible for grammars outside the LL(1) class. Among them are the grammars where the choice of the way to proceed is made by examining the input within the reach of one parsing procedure, instead of examining a single letter. We called them "LL(1p)" grammars, the "1p" meaning "one procedure".

Unfortunately, checking the conditions stated by Proposition 8 is not as simple as verifying the LL(1) property. The families $\mathcal{FIRST}$ can be constructed in a mechanical way, and are presumably not very large. But, checking the disjointness of their members may be difficult. It becomes more difficult as one moves up the graph in Figure 4; at the top of that graph we come close to the condition (1).

If the sets $\textsc{First}_1$, $\textsc{First}_2$, and $\textsc{Follow}$ satisfying Proposition 8 exist, the information can be used for the improvement suggested by Mizushima et al. in [6]. The set $\textsc{First}_1$ lists the alternative parsing procedures that will be eventually invoked to start the processing of e_1. If a procedure $P \in \textsc{First}_1$ succeeds, but the further processing of e_1 fails, the conditions of Proposition 8 mean that e_2 is doomed to fail, unless it succeeds on empty string. But in this case, the subsequent processing of Tail(A) will fail. One can thus insert a "cut" operator after P to save memoization and an unnecessary attempt at e_2.

The class of EBNF grammars that can be directly transcribed into PEG includes cases where the choice of the way to proceed is made by looking at the input within the reach of more than one parsing procedure. The following is an example of such grammar:

InputLine = *Assignment* `|` *Expression*
Assignment = *Name* `"="` *Expression*
Expression = *Primary* `([+-]` *Primary*`)`*
Primary = *Name* `|` *Number*
Name = `[a-z]`$^+$
Number = `[0-9]`$^+$

Here both alternatives of *InputLine* may begin with *Name*. It is the next procedure after *Name*, namely one for `"="`, that decides whether to proceed or backtrack. Still, the grammar works perfectly as PEG parser. This class of grammars, which may be called "LL(2p)", or in general "LL(kp)" requires further analysis. We may need to define $\mathcal{FIRST}$ families as consisting of sequences of k expressions and use Proposition 5.

Last but not least: it appears that natural semantics, as used by Medeiros to define the meaning of a grammar, is a very convenient tool to investigate the grammar's properties.

Appendix

Proof of Proposition 1

We spell out the proof sketched in [5]. It is by induction on the height of the proof tree.

(Induction base) Suppose the proof of $[e]\ xy \overset{\text{PEG}}{\rightsquigarrow} y$ consists of one step. Then it has to be the proof of $[\boldsymbol{\varepsilon}]\ x \overset{\text{PEG}}{\rightsquigarrow} x$ or $[a]\ ax \overset{\text{PEG}}{\rightsquigarrow} x$ using **empty.p** or **letter.p1**, respectively. But then, $[\boldsymbol{\varepsilon}]\ x \overset{\text{BNF}}{\rightsquigarrow} x$ respectively $[a]\ ax \overset{\text{BNF}}{\rightsquigarrow} x$ by **empty.b** or **letter.b**.

(Induction step) Assume that for every proof tree for $[e]\ xy \overset{\text{PEG}}{\rightsquigarrow} y$ of height $n \geq 1$ exists a proof of $[e]\ xy \overset{\text{BNF}}{\rightsquigarrow} y$. Consider a proof tree for $[e]\ xy \overset{\text{PEG}}{\rightsquigarrow} y$ of height $n+1$. Its last step must be one of these:

- $[A]\ xyz \overset{\text{PEG}}{\rightsquigarrow} x$ derived from $A = e_1 e_2$, $[e_1]\ xyz \overset{\text{PEG}}{\rightsquigarrow} yz$, and $[e_2]\ yz \overset{\text{PEG}}{\rightsquigarrow} z$ using **seq.p1**.
  By induction hypothesis, $[e_1]\ xyz \overset{\text{BNF}}{\rightsquigarrow} yz$ and $[e_2]\ yz \overset{\text{BNF}}{\rightsquigarrow} z$, so $[A]\ xy \overset{\text{BNF}}{\rightsquigarrow} x$ follows from **seq.b**.
- $[A]\ xy \overset{\text{PEG}}{\rightsquigarrow} x$ derived from $A = e_1 e_2$ and $[e_1]\ xy \overset{\text{PEG}}{\rightsquigarrow} y$ using **choice.p1**.
  By induction hypothesis, $[e_1]\ xy \overset{\text{BNF}}{\rightsquigarrow} y$, so $[A]\ xy \overset{\text{BNF}}{\rightsquigarrow} x$ follows from **choice.b1**.
- $[A]\ xy \overset{\text{PEG}}{\rightsquigarrow} x$ derived from $A = e_1 e_2$ and $[e_1]\ xy \overset{\text{PEG}}{\rightsquigarrow}$ fail, $[e_2]\ xy \overset{\text{PEG}}{\rightsquigarrow} y$, using **choice.p2**.
  By induction hypothesis, $[e_2]\ xy \overset{\text{BNF}}{\rightsquigarrow} y$, so $[A]\ xy \overset{\text{BNF}}{\rightsquigarrow} x$ follows from **choice.b2**.

Proof of Proposition 2

To be short, we shall say that "e handles w" if for every $w \in \Sigma^*$ there exists a proof of either $[e]\ w \overset{\text{PEG}}{\rightsquigarrow}$ fail or $[e]\ w \overset{\text{PEG}}{\rightsquigarrow} y$ where $w = xy$. To show that each $e \in \mathbb{E}$ handles every $w \in \Sigma^*$, we are use induction on the length of w. The induction base is proved below as Lemma 2.B and induction step as Lemma 2.S. Both are proved by induction on the size of $\texttt{First}(e)$. We use an observation that if $\mathbb{G}$ is not left-recursive, the set $\texttt{First}(e')$ where $e' \in \texttt{first}(e)$ has fewer elements than $\texttt{First}(e)$.

Lemma 2.S. Suppose $\mathbb{G}$ is not left-recursive, and each $e \in \mathbb{E}$ handles all words of length less than given $n > 0$. Then each $e \in \mathbb{E}$ also handles all words of length n.

Proof:
Is by induction on the size of $\texttt{First}(e)$. Consider a word w of length n.

(Induction base) Each e with $\texttt{First}(e)$ of size 0 handles w.
Such e is either $\boldsymbol{\varepsilon}$ or $a \in \Sigma$. From **empty.p**, **letter.p1**, and **letter.p2** follows, respectively, $[\boldsymbol{\varepsilon}]\ w \overset{\text{PEG}}{\rightsquigarrow} w$, $[a]\ w \overset{\text{PEG}}{\rightsquigarrow} x$ if $w = ax$, and $[a]\ w \overset{\text{PEG}}{\rightsquigarrow}$ fail otherwise.

(Induction step) Suppose each e with $\texttt{First}(e)$ of size less than $m > 0$ handles w. Consider e' with $\texttt{First}(e')$ of size m. By induction hypothesis, each $e \in \texttt{first}(e')$ handles w. Since $m > 0$, must be $e' = A \in N$. Two cases are possible:
(Case 1) $A = e_1 e_2$. We have $e_1 \in \texttt{first}(A)$, so e_1 handles w.

- If $[e_1]\ w \overset{\text{PEG}}{\rightsquigarrow}$ fail, we have $[A]\ w \overset{\text{PEG}}{\rightsquigarrow}$ fail by **seq.p2**.
- If $[e_1]\ w \overset{\text{PEG}}{\rightsquigarrow} w$, we have $\varepsilon \in \mathcal{L}(e_1)$ by Proposition 1, so $e_2 \in \texttt{first}(A)$, and e_2 handles w. We have $[e_2]\ w \overset{\text{PEG}}{\rightsquigarrow} Z$ where Z is either fail or z where $w = yz$. This gives $[A]\ \varepsilon \overset{\text{PEG}}{\rightsquigarrow} Z$ by **seq.p1**.
- If $[e_1]\ w \overset{\text{PEG}}{\rightsquigarrow} y$ where $w = xy$ and $x \neq \varepsilon$, y has length less than n and e_2 handles y according to the Lemma's assumption. We have $[e_2]\ y \overset{\text{PEG}}{\rightsquigarrow} Z$ where Z is either fail or z where $w = yz$. This gives $[A]\ \varepsilon \overset{\text{PEG}}{\rightsquigarrow} Z$ by **seq.p1**.

(Case 2) $A = e_1 | e_2$. We have $e_1 \in \texttt{first}(A)$ and $e_2 \in \texttt{first}(A)$, so each of e_1, e_2 handles w.

- If $[e_1]\ w \overset{\text{PEG}}{\rightsquigarrow} y$ where $w = xy$, we have $[A]\ w \overset{\text{PEG}}{\rightsquigarrow} y$ by **choice.p1**.
- If $[e_1]\ \varepsilon \overset{\text{PEG}}{\rightsquigarrow}$ fail, we have $[e_2]\ w \overset{\text{PEG}}{\rightsquigarrow} Y$ where Y is either fail or y where $w = yz$. This gives $[A]\ w \overset{\text{PEG}}{\rightsquigarrow} Y$ by **choice.p2**.

In each case, $e' = A$ handles w. □

Lemma 2.B. If $\mathbb{G}$ is not left-recursive, each $e \in \mathbb{E}$ handles the word of length 0.

Proof:
The proof is almost identical to that of Lemma 2.S if we take $w = \varepsilon$ and use **letter.p3** instead of **letter.p1**, and **letter.p2** in the induction base. The third item in Case 1 is not applicable. □

Proof of Proposition 3

Assume $\mathbb{G}$ satisfies the stated conditions. Take any $w \in \mathcal{L}(S)$. It means there exists proof tree for $[S]\ w\$ \overset{\text{BNF}}{\rightsquigarrow} \$$. We are going to show that for each partial result $[e]\ xy\$ \overset{\text{BNF}}{\rightsquigarrow} y\$$ in that tree there exists a proof of $[e]\ xy\$ \overset{\text{PEG}}{\rightsquigarrow} \$$. We show it using induction on the height of the proof tree.

(Induction base) Suppose the proof of the partial result $[e]\ xy\$ \overset{\text{BNF}}{\rightsquigarrow} y\$$ consists of one step. Then it has to be the proof of $[\varepsilon]\ x\$ \overset{\text{BNF}}{\rightsquigarrow} x\$$ or $[a]\ ax\$ \overset{\text{BNF}}{\rightsquigarrow} x\$$ using **empty.b** or **letter.b**, respectively. But then, $[\varepsilon]\ x\$ \overset{\text{PEG}}{\rightsquigarrow} x\$$ respectively $[a]\ ax\$ \overset{\text{PEG}}{\rightsquigarrow} x\$$ by **empty.p** or **letter.p1**.

(Induction step) Assume that for every partial result $[e]\ xy\$ \overset{\text{BNF}}{\rightsquigarrow} y\$$ that has proof tree of height $n \geq 1$ there exists a proof of $[e]\ xy\$ \overset{\text{PEG}}{\rightsquigarrow} y\$$. Consider a partial result $[e]\ xy\$ \overset{\text{BNF}}{\rightsquigarrow} y\$$ with proof tree of height $n+1$. Its last step must be one of these:

- $[A]\ xyz\$ \overset{\text{BNF}}{\rightsquigarrow} z\$$ derived from $A = e_1e_2$, $[e_1]\ xyz\$ \overset{\text{BNF}}{\rightsquigarrow} yz\$$, and $[e_2]\ yz\$ \overset{\text{BNF}}{\rightsquigarrow} z\$$ using **seq.b**. By induction hypothesis, $[e_1]\ xyz\$ \overset{\text{PEG}}{\rightsquigarrow} yz\$$ and $[e_2]\ yz\$ \overset{\text{PEG}}{\rightsquigarrow} z\$$, so $[A]\ xyz\$ \overset{\text{PEG}}{\rightsquigarrow} z\$$ follows from **seq.p1**.
- $[A]\ xy\$ \overset{\text{BNF}}{\rightsquigarrow} y\$$ derived from $A = e_1|e_2$ and $[e_1]\ xy\$ \overset{\text{BNF}}{\rightsquigarrow} y\$$ using **choice.b1**. By induction hypothesis, $[e_1]\ xy\$ \overset{\text{PEG}}{\rightsquigarrow} y\$$, so $[A]\ xy\$ \overset{\text{PEG}}{\rightsquigarrow} y\$$ follows from **choice.p1**.
- $[A]\ xy\$ \overset{\text{BNF}}{\rightsquigarrow} y\$$, derived from $A = e_1|e_2$ and $[e_2]\ xy\$ \overset{\text{BNF}}{\rightsquigarrow} y\$$ using **choice.b2**. By induction hypothesis, $[e_2]\ xy\$ \overset{\text{PEG}}{\rightsquigarrow} y\$$. But, to use **choice.p2** we also need to verify that $[e_1]\ xy\$ \overset{\text{PEG}}{\rightsquigarrow} \text{fail}$.
  Suppose that there is no proof of $[e_1]\ xy\$ \overset{\text{PEG}}{\rightsquigarrow} \text{fail}$. Then, according to Proposition 2, there exists a proof of $[e_1]\ uv\$ \overset{\text{PEG}}{\rightsquigarrow} v\$$ where $uv\$ = xy\$$. According to Proposition 1, there exists a proof of $[e_1]\ uv\$ \overset{\text{BNF}}{\rightsquigarrow} v\$$, so $u \in \mathcal{L}(e_1)$. From $[e_2]\ xy\$ \overset{\text{BNF}}{\rightsquigarrow} y\$$ follows $x \in \mathcal{L}(e_2)$. From $[A]\ xy\$ \overset{\text{BNF}}{\rightsquigarrow} y\$$ follows $y\$ \in \text{Tail}(A)$. From $uv\$ = xy\$$ follows $u \leq xy$, so $u \in \text{Pref}(\mathcal{L}(e_2)\,\text{Tail}(A))$, which contradicts (1). We must thus conclude that there exists a proof of $[e_1]\ xy\$ \overset{\text{PEG}}{\rightsquigarrow} \text{fail}$, so there exists a proof of $[A]\ xy\$ \overset{\text{PEG}}{\rightsquigarrow} y\$$ using **choice.p2**.

The proof that $[S]\ w\$ \overset{\text{PEG}}{\rightsquigarrow} \$$ exists is a special case of the above.

Proof of Lemma 4

Consider some $A \in N$ and $y\$ \in \text{Tail}(A)$. By definition, there is a proof of $[S]\ w\$ \overset{\text{BNF}}{\rightsquigarrow} \$$ that contains $[A]\ xy\$ \overset{\text{BNF}}{\rightsquigarrow} y\$$ as one of the partial results. This partial result must be used in a subsequent derivation. This derivation can only result in one of the following:

(a) $[A_1]\ xy\$ \overset{\text{BNF}}{\rightsquigarrow} y\$$ where $A_1 = A \,|\, e$ from $[e]\ xy\$ \overset{\text{BNF}}{\rightsquigarrow} y\$$ using **choice.b1**.
(b) $[A_1]\ xy\$ \overset{\text{BNF}}{\rightsquigarrow} y\$$ where $A_1 = e \,|\, A$ from $[e]\ xy\$ \overset{\text{BNF}}{\rightsquigarrow} y\$$ using **choice.b2**.
(c) $[A_1]\ zxy\$ \overset{\text{BNF}}{\rightsquigarrow} y\$$ where $A_1 = e\,A$ from $[e]\ zxy\$ \overset{\text{BNF}}{\rightsquigarrow} xy\$$ using **seq.b**.
(d) $[A_1]\ xy\$ \overset{\text{BNF}}{\rightsquigarrow} y\$$ where $A_1 = A\,e$ from $[e]\ \varepsilon y\$ \overset{\text{BNF}}{\rightsquigarrow} y\$$ using **seq.b**.
(e) $[A_1]\ xy'y''\$ \overset{\text{BNF}}{\rightsquigarrow} y''\$$ where $A_1 = A\,B$, $y'y'' = y$ and $y' \neq \varepsilon$ from $[B]\ y'y''\$ \overset{\text{BNF}}{\rightsquigarrow} y''\$$ using **seq.b**.

In each of the cases (a)-(d), the result is similar to the original one, and the alternative derivations (a)-(e) apply again. We may have a chain of derivations (a)-(d), but it must end with (e) as y must eventually be reduced. We have thus in general a sequence of steps of this form:

$$\dfrac{\dfrac{\dfrac{}{[A_0]\ xy\$ \overset{\text{BNF}}{\rightsquigarrow} y\$}\ \ldots\ \text{as in (a)-(d)}}{[A_1]\ x_1y\$ \overset{\text{BNF}}{\rightsquigarrow} y\$}\ \ldots\ \text{as in (a)-(d)}}{[A_2]\ x_2y\$ \overset{\text{BNF}}{\rightsquigarrow} y\$}\ \ldots\ \text{as in (a)-(d)}$$

$$\ldots$$

$$\dfrac{\dfrac{}{[A_{n-1}]\ x_{n-1}y\$ \overset{\text{BNF}}{\rightsquigarrow} y\$}\ \ldots\ \text{as in (a)-(d)} \quad A_{n+1} = A_nB \quad [A_n]\ x_ny\$ \overset{\text{BNF}}{\rightsquigarrow} y\$ \quad [B]\ y'y''\$ \overset{\text{BNF}}{\rightsquigarrow} y''\$}{[A_{n+1}]\ x_ny'y''\$ \overset{\text{BNF}}{\rightsquigarrow} y''\$}$$

where $A_0 = A$ and $n \geq 0$. We have $A_1 \in \texttt{Last}(A_0)$, $A_2 \in \texttt{Last}(A_1)$, etc., $A_n \in \texttt{Last}(A_{n-1})$, and $B \in \texttt{Next}(A_n)$, which means $B \in \texttt{Follow}(A)$. From $[B]\ y'y''\$ \overset{\text{BNF}}{\rightsquigarrow} y''\$$ we have $y' \in \mathcal{L}(B)$; since $y = y'y''$ and $y' \neq \varepsilon$, $y\$$ has a nonempty prefix in $\mathcal{L}(B) \subseteq \mathcal{L}(\texttt{Follow}(A))$.

References

[1] Aho, A. V., Sethi, R., and Ullman, J. D. *Compilers. Principles, Techniques, and Tools*. Addison-Wesley, 1987.

[2] Ford, B. Packrat parsing: a practical linear-time algorithm with backtracking. Master's thesis, Massachusetts Institute of Technology, Sept. 2002.
`http://pdos.csail.mit.edu/papers/packrat-parsing:ford-ms.pdf`.

[3] Ford, B. Packrat parsing: simple, powerful, lazy, linear time, functional pearl. In *Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming (ICFP '02), Pittsburgh, Pennsylvania, USA, October 4-6, 2002* (2002), M. Wand and S. L. P. Jones, Eds., ACM, pp. 36–47.

[4] Ford, B. Parsing expression grammars: A recognition-based syntactic foundation. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2004* (Venice, Italy, 14–16 January 2004), N. D. Jones and X. Leroy, Eds., ACM, pp. 111–122.

[5] Medeiros, S. *Correspondência entre PEGs e Classes de Gramáticas Livres de Contexto*. PhD thesis, Pontifícia Universidade Católica do Rio de Janeiro, Aug. 2010.
`http://www2.dbd.puc-rio.br/pergamum/tesesabertas/0611957_10_pretextual.pdf`
`http://www2.dbd.puc-rio.br/pergamum/tesesabertas/0611957_10_cap_01.pdf`
`http://www2.dbd.puc-rio.br/pergamum/tesesabertas/0611957_10_cap_02.pdf`
etc.
`http://www2.dbd.puc-rio.br/pergamum/tesesabertas/0611957_10_cap_05.pdf`
`http://www2.dbd.puc-rio.br/pergamum/tesesabertas/0611957_10_postextual.pdf`.

[6] Mizushima, K., Maeda, A., and 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, PASTE'10, Toronto, Ontario, Canada, June 5-6, 2010* (2010), S. Lerner and A. Rountev, Eds., ACM, pp. 29–36.

[7] Redziejowski, R. R. Some aspects of Parsing Expression Grammar. *Fundamenta Informaticae 85*, 1–4 (2008), 441–454.

[8] Schmitz, S. Modular syntax demands verification. Tech. Rep. ISRN I3S/RR 2006-32-FR, Laboratoire I3S, Sophia Antipolis, Oct. 2006.

[9] Tremblay, J.-P., and Sorenson, P. G. *The Theory and Practice of Compiler Writing*. McGraw-Hill, 1985.