Trying to understand PEG

Roman R. Redziejowski

roman@redz.se

Abstract. Parsing Expression Grammar (PEG) encodes a recursive-descent parser with limited backtracking. Its properties are useful in many applications. In its appearance, PEG is almost identical to a grammar in the Extended Backus-Naur Form (EBNF), but may define a different language. Recent research formulated some conditions under which PEG is equivalent to its interpretation as EBNF. However, PEG has a useful feature, namely syntactic predicate, that is alien to EBNF. The equivalence results apply thus only to PEG without predicates. The paper considers PEG with predicates. Not being able to investigate equivalence, the paper turns to the limited backtracking that is the main source of difficulty in understanding PEG. It is shown that the limitation of backtracking has no effect under conditions similar to those for PEG without predicates. There is, in general, no mechanical way to check these conditions, but they can be often checked by inspection. The paper outlines an experimental tool to facilitate such inspection.

1 Introduction

Parsing Expression Grammars (PEGs) have been introduced by Ford in [3] as a new formalism for describing syntax of programming languages. The formalism encodes a recursive-descent parser with limited backtracking. Backtracking removes the LL(1) restriction usually imposed on top-down parsers. The backtracking being limited makes it possible for the parser to work in a linear time, which is achieved with the help of "memoization" or "packrat" technology described in [1, 2].

In addition to circumventing the LL(1) restriction, PEG can be used to define parsers that do not require a separate "scanner" or "tokenizer". All this makes it useful, but PEG is not well understood as a language definition tool. Literature contains many examples of surprising behavior.

In its appearance, PEG is almost identical to a grammar in the Extended Backus-Naur Form (EBNF). Few minor typographical changes convert EBNF to PEG. As EBNF is familiar to most, one expects that the identically-looking PEG defines the same language. This is often the case, but the confusion comes when it is not.

In [6], the author tried to construct exact formulas for the language defined by a given PEG. This was a failure as the formulas became extremely complex with increasing complexity of the grammar.

In a pioneering work [5][1], Medeiros used "natural semantics" to describe both PEG and EBNF. Using this approach, he demonstrated that any EBNF grammar satisfying the LL(1) condition defines exactly the same language as its PEG counterpart. In [7], the author extended this result to a much wider class of grammars. However, there is no general way to mechanically check the conditions specified there. Some manual methods were suggested, based mainly on inspection.

PEG has one feature that does not have a counterpart in EBNF: the syntactic predicate. All results about the equivalence of PEG and EBNF must be thus restricted to PEG without predicates. And this is the case for all results from [5, 7]. But, predicates are useful in defining some features of the language like the distinction between keywords and identifiers.

This paper is an attempt to better understand PEG even in the presence of predicates. Because one can no longer investigate the equivalence of PEG and EBNF, we concentrate on the main source of confusion: the limited backtracking. We find that limited backtracking has no effect on choice expressions that can be identified as "disjoint". The conditions for disjointness are similar to those from [7], with LL(1) as the strongest one. Again, there is no general mechanical way to check disjointness. We outline an experimental tool, called *PEG Analyzer*, that combines the LL(1) test with heuristics to investigate disjointness of choice expressions.

We start by recalling, in Section 2, the definition of Parsing Expression Grammar and its formal semantics. In Section 3, we discuss limited backtracking and disjoint expressions. Section 4 introduces the PEG Analyzer with the help of three examples. The results obtained in Section 3 require a modification to the relation Follow known from the classical literature. This modification involves a rather tedious treatment not relevant for the main subject, so it is presented separately in Section 5. Finally, Section 6 contains few comments.

2 The grammar

We start with a simplified Parsing Expression Grammar $\mathbb{G}$ over alphabet Σ. The grammar is a set of *rules* of the form $A = e$ where A belongs to a set N of symbols distinct from the letters of Σ and e is an *expression*. Each expression is one of these:

$$\begin{array}{llll} \varepsilon & (\text{"empty"}), & !\,e & (\text{"predicate"}), \\ a \in \Sigma & (\text{"terminal"}), & e_1 e_2 & (\text{"sequence"}), \\ A \in N & (\text{"nonterminal"}), & e_1 |\, e_2 & (\text{"choice"}), \end{array}$$

where each of e_1, e_2, e is an expression. The set of all expressions is in the following denoted by $\mathbb{E}$. There is exactly one rule $A = e$ for each $A \in N$. The expression e appearing in this rule is denoted by $e(A)$. The predicate operator binds stronger than sequence and sequence stronger than choice.

[1] This work is in Portuguese. An extended English version is available in [4].

The expressions represent parsing procedures, and rules represent named parsing procedures. In general, parsing procedure is applied to an input string from Σ^* and tries to recognize an initial portion of that string. If it succeeds, it returns "success" and usually consumes the recognized portion. Otherwise, it returns "failure" and does not consume anything. The actions of different procedures are specified in Figure 1.

ε	Indicate success without consuming any input.
a	If the text ahead starts with a, consume a and return success. Otherwise return failure.
A	Call $e(A)$ and return result.
$!\,e$	Call e. Return failure if succeeded. Otherwise return success without consuming any input.
$e_1\,e_2$	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.
$e_1\|\,e_2$	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.

Fig. 1. Actions of expressions as parsing procedures

We note the limited backtracking: once e_1 in $e_1|\,e_2$ succeeded, e_2 will never be tried. The backtracking done by the sequence expression may only roll back $e_1|\,e_2$ as a whole.

The actions of parsing procedures can be formally defined using "natural semantics" introduced in [4, 5]. For $e \in \mathbb{E}$, we write $[e]\ xy \overset{\text{PEG}}{\leadsto} y$ to mean that e applied to string xy consumes x, and $[e]\ x \overset{\text{PEG}}{\leadsto} \texttt{fail}$ to mean that e fails when applied to x. One can see that $[e]\ xy \overset{\text{PEG}}{\leadsto} y$, respectively $[e]\ x \overset{\text{PEG}}{\leadsto} \texttt{fail}$, holds if and only if it can be formally proved using the inference rules shown in Figure 2.

The PEG parser may end up in an infinite recursion, the well-known nemesis of top-down parsers. Formally, it means that there is no proof according to the rules of Figure 2. It has been demonstrated that if the grammar $\mathbb{G}$ is free from left-recursion, then for every $e \in \mathbb{E}$ and $x \in \Sigma^*$ there exists a proof of $[e]\ x \overset{\text{PEG}}{\leadsto} \texttt{fail}$ or $[e]\ x \overset{\text{PEG}}{\leadsto} y$ for some $y \in \Sigma^*$. This has been shown in [4,5] by checking that PEG defined by natural semantics is equivalent to that defined by Ford in [3] and using the result from there. An independent proof for grammar without predicates is given in [7]. It is easily extended to grammar with predicates. We assume from now on that $\mathbb{G}$ is free from left-recursion.

For $e \in \mathbb{E}$, we denote by $\mathcal{L}(e)$ the set of words $x \in \Sigma^*$ such that $[e]\ xy \overset{\text{PEG}}{\leadsto} y$ for some $y \in \Sigma^*$. This is the language accepted by e. Note that, in general, $x \in \mathcal{L}(e)$ does not mean $[e]\ xy \overset{\text{PEG}}{\leadsto} y$ for each y.

$$\frac{}{[\varepsilon]\ x \overset{\text{PEG}}{\leadsto} x}\ \textit{(empty)} \qquad \frac{[e(A)]\ xy \overset{\text{PEG}}{\leadsto} Y}{[A]\ xy \overset{\text{PEG}}{\leadsto} Y}\ \textit{(rule)}$$

$$\frac{[e]\ xy \overset{\text{PEG}}{\leadsto} y}{[!\,e]\ xy \overset{\text{PEG}}{\leadsto} \texttt{fail}}\ \textit{(not1)} \qquad \frac{[e]\ x \overset{\text{PEG}}{\leadsto} \texttt{fail}}{[!\,e]\ x \overset{\text{PEG}}{\leadsto} x}\ \textit{(not2)}$$

$$\frac{}{[a]\ ax \overset{\text{PEG}}{\leadsto} x}\ \textit{(letter1)} \qquad \frac{b \neq a}{[b]\ ax \overset{\text{PEG}}{\leadsto} \texttt{fail}}\ \textit{(letter2)} \qquad \frac{}{[a]\ \varepsilon \overset{\text{PEG}}{\leadsto} \texttt{fail}}\ \textit{(letter3)}$$

$$\frac{[e_1]\ xyz \overset{\text{PEG}}{\leadsto} yz \quad [e_2]\ yz \overset{\text{PEG}}{\leadsto} Z}{[e_1e_2]\ xyz \overset{\text{PEG}}{\leadsto} Z}\ \textit{(seq1)} \qquad \frac{[e_1]\ x \overset{\text{PEG}}{\leadsto} \texttt{fail}}{[e_1e_2]\ x \overset{\text{PEG}}{\leadsto} \texttt{fail}}\ \textit{(seq2)}$$

$$\frac{[e_1]\ xy \overset{\text{PEG}}{\leadsto} y}{[e_1|\,e_2]\ xy \overset{\text{PEG}}{\leadsto} y}\ \textit{(choice1)} \qquad \frac{[e_1]\ xy \overset{\text{PEG}}{\leadsto} \texttt{fail} \quad [e_2]\ xy \overset{\text{PEG}}{\leadsto} Y}{[e_1|\,e_2]\ xy \overset{\text{PEG}}{\leadsto} Y}\ \textit{(choice2)}$$

where Y denotes y or $\texttt{fail}$ and Z denotes z or $\texttt{fail}$.

Fig. 2. Formal semantics of PEG

We define the EBNF interpretation of $\mathbb{G}$ as the language $\mathcal{L}^E(e)$ accepted by expression $e \in \mathbb{E}$. It is defined recursively as

$$\begin{aligned} &\mathcal{L}^E(\varepsilon) = \{\varepsilon\}, && \mathcal{L}^E(!\,e) = \{\varepsilon\}, \\ &\mathcal{L}^E(a) = \{a\}, && \mathcal{L}^E(e_1e_2) = \mathcal{L}^E(e_1)\mathcal{L}^E(e_2), \\ &\mathcal{L}^E(A) = \mathcal{L}^E(e(A)), && \mathcal{L}^E(e_1|\,e_2) = \mathcal{L}^E(e_1) \cup \mathcal{L}^E(e_2). \end{aligned}$$

By defining $\mathcal{L}^E(!\,e) = \{\varepsilon\}$, we extended the interpretation to PEG with predicates. This is not what one can expect looking at the grammar, just an approximation. The following result from [4,5] is easily extended to our interpretation of predicates:

$$\mathcal{L}(e) \subseteq \mathcal{L}^E(e) \text{ for any } e \in \mathbb{E}\,. \tag{1}$$

The opposite of (1) does not, in general hold. This is, to some extent, due to the different interpretation of predicates, but the main cause is limited backtracking.

3 Disjoint choice and limited backtracking

We say that choice $e = e_1|\,e_2$ is *strictly disjoint* if

$$\mathcal{L}(e_1)\Sigma^* \cap \mathcal{L}(e_2)\Sigma^* = \varnothing. \tag{2}$$

Such choice is not affected by the limitation of backtracking. Indeed, suppose that e is applied to some string s and e_1 succeeds. This means $s \in \mathcal{L}(e_1)\Sigma^*$.

After this, any attempt to backtrack must result in a failure. It would mean applying e_2 to s, and a success would mean $s \in \mathcal{L}(e_2)\Sigma^*$, which is excluded by (2). So, not attempting it does not make any difference.

The choice `aa|a` is not disjoint in the sense of (2). However, it is not affected by partial backtracking when it appears in some contexts, for example, in `(aa|a)b`. A success of `aa` means that input has the form $\mathtt{aa}\Sigma^*$, so backtracking to `a` is useless as it will later result in a failure by not finding `b`.

We are going to look at limited backtracking in the context of our grammar $\mathbb{G}$ successfully parsing a complete input string. Thus, we assume that $\mathbb{G}$ has a unique *start symbol* $S \in N$ with the corresponding rule $S = e\,\mathtt{\#}$ where e is an expression and `#` is a unique end-of-text marker that appears only in this rule. We say that a string $w \in \Sigma^*$ is *accepted* by S to mean that $[S]\ w \overset{\text{PEG}}{\leadsto} \varepsilon$.

For an expression $e \in \mathbb{E}$, we define Tail(e) to be the set of strings y such that $[e]\ xy \overset{\text{PEG}}{\leadsto} y$ appears as partial result in the proof of $[S]\ w \overset{\text{PEG}}{\leadsto} \varepsilon$ for some w. We say that the choice $e = e_1 |\, e_2$ is *disjoint (in the context of* $\mathbb{G}$*)* if

$$\mathcal{L}(e_1)\Sigma^* \,\cap\, \mathcal{L}(e_2)\,\mathrm{Tail}(e) = \varnothing. \tag{3}$$

The same argument as before shows that such choice is not affected by the limitation of backtracking. If e is applied to some string s in a proof $[S]\ w \overset{\text{PEG}}{\leadsto} \varepsilon$ and e_1 succeeds, we have $s \in \mathcal{L}(e_1)\Sigma^*$. An attempt to backtrack would mean applying e_2 to s, and a success would mean $s \in \mathcal{L}(e_2)\,\mathrm{Tail}(e)$, which is excluded by (3).

A mechanical checking of (3) is in general impossible because of complexity of $\mathcal{L}(e)$ and Tail(e), and because it may involve checking emptiness of intersection of context-free languages - known to be undecidable. However, it can be often checked using approximation.

With (1), we can approximate $\mathcal{L}(e_1)$ and $\mathcal{L}(e_2)$ by $\mathcal{L}^E(e_1)$ respectively $\mathcal{L}^E(e_2)$. This gives a stronger condition:

$$\mathcal{L}^E(e_1)\Sigma^* \,\cap\, \mathcal{L}^E(e_2)\,\mathrm{Tail}(e) = \varnothing. \tag{4}$$

It was shown in [7] that if this condition holds for all choice expressions in a grammar without predicates, we have $\mathcal{L}(e) = \mathcal{L}^E(e)$ for all $e \in \mathbb{E}$.

To approximate Tail(e), we need to modify the relation Follow known from the classical literature. The modification is described in Section 5, where it is shown (Proposition 1) that with the modified relation, we have $\mathrm{Tail}(e) \subseteq \mathcal{L}^E(\mathrm{Follow}(e))\Sigma^*$ where $\mathcal{L}^E(\mathrm{Follow}(e)) = \cup_{x\in\mathrm{Follow}(e)}\mathcal{L}^E(x)$. This gives an even stronger condition:

$$\mathcal{L}^E(e_1)\Sigma^* \,\cap\, \mathcal{L}^E(e_2)\mathcal{L}^E(\mathrm{Follow}(e))\Sigma^* = \varnothing. \tag{5}$$

Using known methods one can compute the sets of symbols that appear as first in strings from $\mathcal{L}^E(e_1)\Sigma^*$ respectively $\mathcal{L}^E(e_2)\mathcal{L}^E(\mathrm{Follow}(e))\Sigma^*$. Condition (5) is then obviously satisfied if these sets are disjoint. This is the familiar LL(1) condition.

As suggested in [7, 8], one can obtain a weaker condition by approximating $\mathcal{L}^E(e_1)\Sigma^*$ and $\mathcal{L}^E(e_2)\mathcal{L}^E(\text{Follow}(e))\Sigma^*$ with sets of the form $F\Sigma^*$ where F is some suitably chosen subset using the classical relation First. Some ways of choosing F have been suggested, but they can not, in general, be mechanized. Another approximation was suggested by Schmitz in [9].

The grammar $\mathbb{G}$ considered up to now is a simplified version of full PEG. This latter allows expressions such as $e_1|\,e_2|\,\dots|\,e_n$, $e_1e_2\dots e_n$, e^*, e^+, and $e?$. The expression $E = e_1|\,e_2|\,\dots|\,e_n$ is a syntactic sugar for $E = e_1|\,E_1$, $E_1 = e_2|\,E_2$, $\dots$, $E_n = e_n$ so (3) must hold for all of $E, E_1, \dots, E_{n-1}$. One can verify that this is true if

$$\mathcal{L}(e_i)\Sigma^* \cap \mathcal{L}(e_j)\,\text{Tail}(E) = \varnothing \quad \text{for } 1 \le i < j < n. \tag{6}$$

The expressions $E = e^*$, $E = e^+$, and $E = e?$ constitute syntactic sugar for, respectively $E = eE/\varepsilon$, $E = eE/e$, and $E = e/\varepsilon$ so (3) must hold for each of them. One can verify that this is true if

$$\mathcal{L}(e)\Sigma^* \cap \text{Tail}(E) = \varnothing. \tag{7}$$

The rules for computing Follow(e) given in the Section 5 can be similarly extended to the full PEG.

The terminals in full PEG are not necessary single letters, and may be multi-letter quoted strings. Instead of sets of "first letters" used in the test for LL(1), one has to compute sets of "first terminals" and check their disjointness.

4 PEG Analyzer

Giving up all hope for an automatic verification of (3), the author created an experimental tool, the *PEG Analyzer*, that combines the LL(1) check with heuristics. It takes a grammar, tests all choice expressions for LL(1) using (5), and presents for inspection those that did not pass the test. To facilitate inspection, it gradually expands the involved expressions by replacing them with their definitions, somewhat in the spirit of what was suggested in [7, 8].

4.1 Example 1: Simple calculator

To give some idea of the Analyzer, we apply it to the grammar shown in Figure 3. The grammar defines the syntax of a simple calculator.

When Analyzer is applied to this grammar, it indicates that the choice between the first two alternatives of `Factor` does not satisfy LL(1), and opens a window shown in Figure 4. It is an invitation to verify (3) for $e_1 =$ `Digits? Fraction`, $e_2 =$ `Digits`, and Tail(`Factor`).

The first two lines show these two expressions. The second is, in fact, a pseudo-expression, with pseudo-expression `Tail(Factor)` representing the tail. The third line tells that both expressions have `[0-9]` as "first terminal".

```
Start    = Sum #
Sum      = Product (AddOp Product)*
Product  = Factor (MultOp Factor)*
Factor   = Digits? Fraction | Digits | Lparen Sum Rparen
Fraction = "." Digits
AddOp    = [-+]
MultOp   = [*/]
Lparen   = "("
Rparen   = ")"
Digits   = [0-9]+
```

Fig. 3. A simple calculator

```
Factor.1 = Digits? Fraction
Factor.2 = Digits Tail(Factor)
  [0-9]  <==>  [0-9]

Digits? Fraction
  Digits Fraction
  [0-9] [0-9]* "." Digits
    <==>
Digits Tail(Factor)
Digits (AddOp Product | MultOp Factor | Rparen) ...
[0-9] [0-9]* (AddOp Product | MultOp Factor | Rparen) ...
```

Fig. 4. Presentation of a non-LL(1) case

The subsequent lines show the two expressions in more detail. Thus, the first expression stands for two alternative expressions, `Digits Fraction` and `Fraction`. Since this latter does not start with `[0-9]`, only `Digits Fraction` appears, with an indentation showing that it is one of alternatives. In the next line, `Digits` is expanded following its definition to `[0-9] [0-9]*` and `Fraction` to `"." Digits`. The result is supposed to give an idea of strings starting with `[0-9]` that are accepted by `Digits? Fraction`.

In the second expression, `Tail(Factor)` is replaced by the approximation $\mathcal{L}^E(\text{Follow}(\texttt{Factor}))\Sigma^*$ in the form of pseudo-expression. Again, `Digits` is expanded to `[0-9] [0-9]*`.

Verifying (3) means checking if any string represented by e_1 can be a prefix of any string in $\mathcal{L}(e_2)\,\text{Tail}(\texttt{Factor})$. One can easily see that `Digits` in the first expression is always followed by a dot, while in the second it can be only followed by `AddOp`, `MultOp`, or `Rparen`, none of which is a dot. The condition (3) is thus satisfied.

4.2 Example 2: Grammar with predicates

The second example illustrates treatment of predicates. The grammar in Figure 5 is a fragment of larger grammar that uses identifiers, with some of them being reserved as ”keywords”. Only one keyword, `"print"` is shown. Its definition is followed by `!Letter` to make sure it is not recognized as a prefix of an identifier. The definition of `Identifier` is preceded by `!Keyword` to ensure that keyword is not recognized as an identifier.

```
Statement  = (Keyword Number | Identifier Number) ";" #
Keyword    = "print" !Letter
Identifier = !Keyword Letter+
Letter = [a-z]
Number = [0-9]+
```

Fig. 5. Grammar with predicates

The Analyzer applied to this grammar indicates that the choice in `Statement` does not satisfy LL(1), and opens the window shown in Figure 6.
To check LL(1), the Analyzer approximated $\mathcal{L}(\texttt{"print"!\,Letter})$ with $\mathcal{L}^E(\texttt{"print"})$ and $\mathcal{L}(\texttt{!\,Keyword Letter+})$ with $\mathcal{L}^E(\texttt{Letter+})$:

$$\mathcal{L}(\texttt{"print"!\,Letter}) \subseteq \mathcal{L}^E(\texttt{"print"!Letter}) = \mathcal{L}^E(\texttt{"print"})\,,$$
$$\mathcal{L}(\texttt{!\,Keyword Letter+}) \subseteq \mathcal{L}^E(\texttt{!\,Keyword Letter+}) = \mathcal{L}^E(\texttt{Letter+})\,.$$

It found their first terminals to be, respectively, `"print"` and `[a-z]`. As they are not disjoint, the choice does not satisfy LL(1) and is signaled as such. But, this is a false alarm. One can easily see that

$$\mathcal{L}(\texttt{"print" !\,Letter ...}) \cap \mathcal{L}(\texttt{!\,Keyword Letter+ ...}) = \varnothing$$

showing that the expression satisfies (3).

```
Statement.1.1 = Keyword Number
Statement.1.2 = Identifier Number Tail(Statement.1)
  "print"  <==>  [a-z]

Keyword Number
"print" !Letter Number
  <==>
Identifier Number Tail(Statement.1)
Identifier Number ";" ...
!Keyword Letter+ Number ";" ...
```

Fig. 6. Presentation of a non-LL(1) case

4.3 Example 3: Non-disjoint expressions

Suppose now that in the calculator from Example 1, we want sometimes to skip the multiplication sign and write, for example `2(.3+4)` instead of `2*(.3+4)`. To achieve this, we replace the definition of `MultOp` by `MultOp = "*"? | "/"`.

The Analyzer applied to the modified grammar shows now two cases not satisfying LL(1). The first produces the window shown in Figure 7. It says that `[0-9]` in `[0-9]+` is followed by something that may start with `[0-9]`, namely any of two different alternatives of `Factor` after omitted first alternative of `MultOp`. Clearly, `[0-9]` is a prefix of each alternative. The condition (3) is not satisfied.

```
Digits.1 = [0-9]
Tail(Digits)
  [0-9]  <==>  [0-9]

 [0-9]
   <==>
 Tail(Digits)
 MultOp Factor ...
   Factor ...
     Digits? Fraction ...
       Digits Fraction ...
       [0-9] [0-9]* Fraction ...
     Digits  ...
     [0-9] [0-9]* ...
```

Fig. 7. Presentation of a non-LL(1) case

How does this happen and what does it mean? A look at the grammar shows that the offending `[0-9]+` is one appearing at the end of the first `Factor` in `Factor (MultOp Factor)*`. With omitted `MultOp` it will gobble up any digits at the beginning of the second `Factor`, without any attempt to backtrack. Thus, for example, `234` will be treated by PEG as a single `Factor`, and not as shorthand for `2*3*4`.

In this example, the grammar interpreted as EBNF has an ambiguity, and PEG just selects one of the possible parses.

The second case not satisfying LL(1) is the same as for the original grammar: the choice between the first two alternatives of `Factor`, and is reported exactly as shown in Figure 4. However, `MultOp` in `MultOp Factor` that appears in the tail of `Factor` may be omitted. And `Factor` has an alternative that begins with a dot. Thus, `Digits` in `Digits Tail(Factor]` can be followed by a dot and (3) is not satisfied. It means that, for example, `2.34` will be treated by PEG as a single `Factor`, and not as shorthand for `2*.34`.

Here the grammar interpreted as EBNF has another ambiguity and PEG chooses one possible parse. In both cases, we may accept the PEG's choice because it corresponds to the perception of a human reader.

5 Computation of Follow

We define a number of relations $R \subseteq \mathbb{E} \times \mathbb{E}$, writing $e' \in R(e)$ to mean $(e, e') \in R$.

- $\text{Derive}_{ss}(e)$ is the set of all $e' \in \mathbb{E}$ such that $[e']\ x'y \overset{\text{PEG}}{\rightsquigarrow} y$ can be derived from $[e]\ xy \overset{\text{PEG}}{\rightsquigarrow} y$ using one inference rule.
  Thus, $e' \in \text{Derive}_{ss}(e)$ if and only if:
  - $e' = A \in N$ where $e(A) = e$,
  - $e' = e\, e_2$ for some e_2 such that $\varepsilon \in \mathcal{L}(e_2)$,
  - $e' = e_1\, e$ for some e_1,
  - $e' = e |\, e_2$ for some e_2,
  - $e' = e_1 |\, e$ for some e_1.
- $\text{Derive}_{sf}(e)$ is the set of all $e' \in \mathbb{E}$ such that $[e']\ xy \overset{\text{PEG}}{\rightsquigarrow} \texttt{fail}$ can be derived from $[e]\ xy \overset{\text{PEG}}{\rightsquigarrow} y$ using one inference rule.
  Thus, $e' \in \text{Derive}_{sf}(e)$ if and only if:
  - $e' = !\, e$,
  - $e' = e\, e_2$ for some e_2.
- $\text{Derive}_{ff}(e)$ is the set of all $e' \in \mathbb{E}$ such that $[e']\ x \overset{\text{PEG}}{\rightsquigarrow} \texttt{fail}$ can be derived from $[e]\ x \overset{\text{PEG}}{\rightsquigarrow} \texttt{fail}$ using one inference rule.
  Thus, $e' \in \text{Derive}_{ff}(e)$ if and only if:
  - $e' = A \in N$ where $e(A) = e$,
  - $e' = e_1\, e$ for some e_1,
  - $e' = e\, e_2$ for some e_2,
  - $e' = e |\, e_2$ for some e_2,
  - $e' = e_1 |\, e$ for some e_1 that can fail.
- $\text{Next}_s(e)$ is the set of all $e' \in \mathbb{E}$ such that $\varepsilon \notin \mathcal{L}(e')$ and there exists $e\, e' \in \mathbb{E}$.
- $\text{Next}_f(e) = \{\varepsilon\}$ for all e such that there exists $!\, e \in \mathbb{E}$ or $e |\, e_2 \in \mathbb{E}$ for some e_2.

Define $\text{Follow} = \text{Derive}^*_{ss} \times \text{Next}_s \cup \text{Derive}^*_{ss} \times \text{Derive}_{sf} \times \text{Derive}^*_{ff} \times \text{Next}_f$.

Proposition 1. *For each partial result $[e]\ xy \overset{\text{PEG}}{\rightsquigarrow} y$ in the proof of $[S]\ w \overset{\text{PEG}}{\rightsquigarrow} \varepsilon$ holds $y \subseteq \mathcal{L}^E(\text{Follow}(e))\Sigma^*$ where $\mathcal{L}^E(\text{Follow}(e)) = \bigcup_{e' \in \text{Follow}(e)} \mathcal{L}^E(e')$.*

Proof. Consider any partial result $[E_1]\ xy \overset{\text{PEG}}{\rightsquigarrow} y$ in the proof of $[S]\ w \overset{\text{PEG}}{\rightsquigarrow} \varepsilon$. It is the first in a chain of $n \geq 1$ partial results derived successively using the rules of Figure 2 and other partial results. The chain ends with final result $[S]\ w \overset{\text{PEG}}{\rightsquigarrow} \varepsilon$ derived from $[E_n\, \texttt{\#}]\ x_n\, \texttt{\#} \overset{\text{PEG}}{\rightsquigarrow} \varepsilon$. In this chain, the first $j \geq 1$ partial results are of the form $[E_i]\ x_i y \overset{\text{PEG}}{\rightsquigarrow} y$. By definition of Derive_{ss}, we have $E_i \in \text{Derive}^*_{ss}(E_1)$ for $1 \leq i \leq j$. The first partial result in a different form must be one of these:

(a) $[E_j\, e_2]\ x_j uv \overset{\text{PEG}}{\rightsquigarrow} v$ where $y = uv$, $u \neq \varepsilon$, and $[e_2]\ uv \overset{\text{PEG}}{\rightsquigarrow} v$.
(b) $[!\, E_j]\ x_j y \overset{\text{PEG}}{\rightsquigarrow} \texttt{fail}$.
(c) $[E_j\, e_2]\ x_j y \overset{\text{PEG}}{\rightsquigarrow} \texttt{fail}$ where $[e_2]\ x_j y \overset{\text{PEG}}{\rightsquigarrow} \texttt{fail}$.

In case (a) we have $e_2 \in \text{Next}_s(E_j)$, so $e_2 \in (\text{Derive}^*_{ss} \times \text{Next}_f)(E_1) \subseteq \text{Follow}(E_1)$. We have also $y = uv$ where $u \in \mathcal{L}(e_2) \subseteq \mathcal{L}^E(e_2) \subseteq \mathcal{L}^E(\text{Follow}(E_1))$, so $y \in \mathcal{L}^E(\text{Follow}(E_1))\Sigma^*$.

In cases (b) and (c), we have partial result $[E_{j+1}]\ x \overset{\text{PEG}}{\leadsto} \texttt{fail}$. According to the definition of Derive_{sf}, we have $E_{j+1} \in \text{Derive}_{sf}(E_j)$. It is first in the chain of $k \geq 1$ partial results in the form $[E_i]\ x \overset{\text{PEG}}{\leadsto} \texttt{fail}$ derived successively using the rules of Figure 2 and other partial results. By definition of Derive_{ff}, we have $E_i \in \text{Derive}^*_{ff}(E_j)$ for $j+1 \leq i \leq j+k$. The first partial result in a different form must be one of these:

(d) $[!\,E_{j+k}]\ uv \overset{\text{PEG}}{\leadsto} uv$.
(e) $[E_{j+k}|\,e_2]\ uv \overset{\text{PEG}}{\leadsto} v$ where $[e_2]\ uv \overset{\text{PEG}}{\leadsto} v$,

where $uv = x$. We only know that the original y is a suffix of uv, but is very unlikely to be the same as v. We can only approximate it as $y \in \Sigma^*$. In each of (d)-(e), we have $\text{Next}_f(E_{j+k}) = \{\varepsilon\}$, so

$$\{\varepsilon\} = (\text{Derive}^*_{ss} \times \text{Derive}_{sf} \times \text{Derive}^*_{ff} \times \text{Next}_f)(E_1) \subseteq \text{Follow}(E_1).$$

We have $\varepsilon \in \mathcal{L}^E(\text{Follow}(E_1))$, so $y \in \mathcal{L}^E(\text{Follow}(E_1))\Sigma^*$. $\square$

Note that the very rough approximation of $\text{Tail}(e)$ by Σ^*, changing (5) into a strict disjointness, applies to e that appears in a partial proof resulting in a failure (which is eventually needed to prove $[S]\ w \overset{\text{PEG}}{\leadsto} \varepsilon$).

6 Final remarks

The fact that EBNF does not have predicates spoils the useful correspondence with PEG. The basic conditions for absence of effects of limited backtracking remain in principle unchanged, but there is no way of talking about equivalence.

There is no way to just include the recognition-oriented predicates as part of the construction-oriented EBNF. But one may consider some more natural extensions to EBNF that could serve the same purpose as predicates in defining useful grammars.

The example of PEG Analyzer shows that disjointness can often be checked by inspection. The tool as described here is quite primitive. One can improve it by letting the user interactively choose specific parts of expressions for detailed inspection.

The subject for further research is a careful analysis of what happens in the case of non-disjoint choice.

References

1. Ford, B.: Packrat Parsing: a Practical Linear-Time Algorithm with Backtracking. Master's thesis, Massachusetts Institute of Technology (Sep 2002), `http://pdos.csail.mit.edu/papers/packrat-parsing:ford-ms.pdf`
2. Ford, B.: Packrat parsing: simple, powerful, lazy, linear time, functional pearl. In: Wand, M., Jones, S.L.P. (eds.) Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming (ICFP '02), Pittsburgh, Pennsylvania, USA, October 4-6, 2002. pp. 36–47. ACM (2002)
3. Ford, B.: Parsing expression grammars: A recognition-based syntactic foundation. In: Jones, N.D., Leroy, X. (eds.) Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2004. pp. 111–122. ACM, Venice, Italy (14–16 January 2004)
4. Mascarenhas, F., Medeiros, S., Ierusalimschy, R.: On the relation between context-free grammars and Parsing Expression Grammars. Science of Computer Programming 89, 235–250 (2014)
5. Medeiros, S.: Correspondência entre PEGs e Classes de Gramáticas Livres de Contexto. Ph.D. thesis, Pontifícia Universidade Católica do Rio de Janeiro (Aug 2010)
6. Redziejowski, R.R.: Some aspects of Parsing Expression Grammar. Fundamenta Informaticae 85(1–4), 441–454 (2008)
7. Redziejowski, R.R.: From EBNF to PEG. Fundamenta Informaticae 128, 177–191 (2013)
8. Redziejowski, R.R.: More about converting BNF to PEG. Fundamenta Informaticae 133(2-3), 177–191 (2014)
9. Schmitz, S.: Modular syntax demands verification. Tech. Rep. I3S/RR-2006-32-FR, Laboratoire I3S, Université de Nice - Sophia Antipolis (Oct 2006)