# Trying to understand PEG*

Roman Redziejowski

**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. Moreover, PEG has a feature that is useful but alien to EBNF, namely syntactic predicate. Recent research found the condition under which PEG without predicates defines the same language as its EBNF look-alike. The condition is that the limited backtracking of PEG is efficient as replacement for full backtracking. No EBNF equivalence can be defined for PEG with predicates, but we conjecture that satisfying the same condition makes the grammar well-behaved. There is, in general, no mechanical way to check the condition, but it 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 "lexer". 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 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 [6, 7], the author extended this result to a much wider class of grammars. The equivalence condition found there states that the limited backtracking of PEG should be "efficient" in the sense that full backtracking can not discover anything new. Thus, PEG with efficient backtracking is as easy to understand as the familiar EBNF.

PEG has a feature that does not have any counterpart in EBNF: the syntactic predicate. As the concept of predicate is recognition-oriented, there is no obvious way to introduce it in the construction-oriented EBNF. The equivalence results from [4–7] are thus restricted to PEG without predicates. But, predicates are useful in defining some language features like distinction between keywords and identifiers.

With no equivalence proof possible, we conjecture that satisfying the condition for efficient backtracking makes PEG well-behaved also in the presence of predicates. Unfortunately, there is no general mechanical way to check the condition. We outline an experimental tool, called *PEG Explorer*, that combines the LL(1) test with heuristics to investigate disjointness of choice expressions.

We start by recalling, in Section 2, the definition of PEG, EBNF, and their formal semantics. In Section 3, we discuss conditions for efficient backtracking. Section 4 introduces PEG Explorer with the help of three examples. Finally, Section 5 contains few comments. Proofs have been placed in the Appendix.

*Submitted for publication in *Fundamenta Informaticae*.

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

## 2 The grammar

We start with a simplified Parsing Expression Grammar $\mathbb{G}$ over alphabet $\Sigma$. 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 $\Sigma$ and $e$ is an *expression*. Each expression is one of these:

| | | | |
|---|---|---|---|
| $\varepsilon$ | ("empty"), | $!\,e$ | ("predicate"), |
| $a \in \Sigma$ | ("terminal"), | $e_1 e_2$ | ("sequence"), |
| $A \in N$ | ("nonterminal"), | $e_1 \mid e_2$ | ("choice"), |

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.

The expressions represent parsing procedures, and rules represent named parsing procedures. In general, parsing procedure is applied to an input string from $\Sigma^*$ 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.

| | |
|---|---|
| $\varepsilon$ | 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. |
| $e_1 \mid e_2$ | 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. |

Figure 1: Actions of expressions as parsing procedures

We note the limited backtracking: once $e_1$ in $e_1 \mid e_2$ succeeded, $e_2$ will never be tried. The backtracking done by the sequence expression may only roll back $e_1 \mid e_2$ 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 $[e]\ xy \overset{\text{PEG}}{\rightsquigarrow} y$ to mean that $e$ applied to string $xy$ consumes $x$, and $[e]\ x \overset{\text{PEG}}{\rightsquigarrow} \texttt{fail}$ to mean that $e$ fails when applied to $x$. One can see that $[e]\ xy \overset{\text{PEG}}{\rightsquigarrow} y$, respectively $[e]\ x \overset{\text{PEG}}{\rightsquigarrow} \texttt{fail}$, 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 $[e]\ x \overset{\text{PEG}}{\rightsquigarrow} \texttt{fail}$ or $[e]\ x \overset{\text{PEG}}{\rightsquigarrow} y$ 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 [6, 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 $[e]\ xy \overset{\text{PEG}}{\rightsquigarrow} y$ for some $y \in \Sigma^*$. This is the language accepted by $e$. Note that, in general, $x \in \mathcal{L}(e)$ does not mean $[e]\ xy \overset{\text{PEG}}{\rightsquigarrow} y$ for each $y$.

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

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

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

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

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

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

Figure 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

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

It can be described by natural semantics shown in Figure 3. One can verify that $x \in \mathcal{L}^E(e)$ if and only if $[e]\ xy \overset{\text{BNF}}{\rightsquigarrow} y$, for any $y$. Defining $\mathcal{L}^E(!\,e) = \{\varepsilon\}$ is not an attempt to introduce predicates in EBNF, but a useful approximation. One can verify that it preserves the property

$$[e]\ xy \overset{\text{PEG}}{\rightsquigarrow} y \ \Rightarrow\ [e]\ xy \overset{\text{BNF}}{\rightsquigarrow} y. \tag{1}$$

established in [4, 5] for PEG without predicates.

$$\frac{[e(A)]\ xy \overset{\text{BNF}}{\rightsquigarrow} y}{[A]\ xy \overset{\text{BNF}}{\rightsquigarrow} y}\ \textit{(rule.b)} \qquad \frac{}{[\varepsilon]\ x \overset{\text{BNF}}{\rightsquigarrow} x}\ \textit{(empty.b)} \qquad \frac{}{[a]\ ax \overset{\text{BNF}}{\rightsquigarrow} x}\ \textit{(letter.b)}$$

$$\frac{}{[!\,e]\ x \overset{\text{BNF}}{\rightsquigarrow} x}\ \textit{(not.b)} \qquad \frac{[e_1]\ xyz \overset{\text{BNF}}{\rightsquigarrow} yz \quad [e_2]\ yz \overset{\text{BNF}}{\rightsquigarrow} z}{[e_1e_2]\ xyz \overset{\text{BNF}}{\rightsquigarrow} z}\ \textit{(seq.b)}$$

$$\frac{[e_1]\ xy \overset{\text{BNF}}{\rightsquigarrow} y}{[e_1|\,e_2]\ xy \overset{\text{BNF}}{\rightsquigarrow} y}\ \textit{(choice1.b)} \qquad \frac{[e_2]\ xy \overset{\text{BNF}}{\rightsquigarrow} y}{[e_1|\,e_2]\ xy \overset{\text{BNF}}{\rightsquigarrow} y}\ \textit{(choice2.b)}$$

Figure 3: Formal semantics of BNF

# 3 Disjoint choice and limited backtracking

It is known that the meaning of a parsing expression often depends on the context. To begin with, we assume that $\mathbb{G}$ does not use predicates, and take as context the proof of $[C]\ w \overset{\text{BNF}}{\rightsquigarrow} z$ for some $C \in \mathbb{E}$ and $w, z \in \Sigma^*$. For $e \in \mathbb{E}$, we define $\mathit{Tail}_C(e)$ to be the set of all strings $y$ in partial results $[e]\ xy \overset{\text{BNF}}{\rightsquigarrow} y$ appearing in the proof of $[C]\ w \overset{\text{BNF}}{\rightsquigarrow} z$ for any $w, z \in \Sigma^*$. We say the choice expression $e = e_1 \,|\, e_2$ is *disjoint in the context of* $C$ if it satisfies:

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

**Proposition 1.** *Assume that $\mathbb{G}$ is not left-recursive and does not contain predicates. If each choice expression accessible from $C$ is disjoint in the context of $C$ then $\mathcal{L}(C) = \mathcal{L}^E(C)$. (Proof is found in the Appendix.)*

Let some $S \in N$ be defined as the *starting symbol* of $\mathbb{G}$. The language $\mathcal{L}(S)$, respectively $\mathcal{L}^E(S)$, is then *the language accepted by* $\mathbb{G}$ under the PEG or EBNF interpretation. The Proposition says that these two interpretations are identical if all choice expressions are disjoint in the context of $S$.

As the EBNF interpretation of predicates is not defined, verifying $\mathcal{L}(S) = \mathcal{L}^E(S)$ does not make sense in the presence of predicates. A possible way out is to take a closer look at condition (2). It postulates, in effect, that limited backtracking be efficient in the sense of finding everything that would be found by full backtracking. Indeed, it says that once $e_1$ succeeded, trying $e_2$ is unnecessary as it cannot lead to a success. Proposition 1 states that PEG with efficient backtracking is equivalent to EBNF. Which is not surprising since the difference between EBNF and PEG is caused by limited backtracking not exploring some paths.

We conjecture that PEG with predicates is well-behaved (whatever that means) if (2) holds in the context of the starting symbol $S$, **and** in the context of each $e$ appearing in $!e \in \mathbb{E}$.

A mechanical checking of (2) is in general impossible because of complexity of $\mathcal{L}(e)$ and $\mathit{Tail}_C(e)$. One can to some degree simplify the task using approximation. It is shown in the Appendix how to obtain $\mathcal{T}_C(e)$ such that $\mathit{Tail}_C(e) \subseteq \mathcal{T}_C(e)$. This gives a condition stronger than (2):

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

This can be often checked by inspection, as illustrated in the next section.

The grammar $\mathbb{G}$ considered up to now is a simplified version of full PEG. This latter allows expressions such as $e_1|\,e_2|\,\ldots|\,e_n$, $e_1e_2\ldots e_n$, $e^*$, $e^+$, and $e?$. The expression $E = e_1|\,e_2|\,\ldots|\,e_n$ is a syntactic sugar for $E = e_1|\,E_1$, $E_1 = e_2|\,E_2, \ldots, E_n = e_n$ so (2) must hold for all of $E, E_1, \ldots, E_{n-1}$. One can verify that this is true if

$$\mathcal{L}(e_i)\Sigma^* \cap \mathcal{L}(e_j)\mathit{Tail}_C(E) = \varnothing \quad \text{for } 1 \le i < j < n. \tag{4}$$

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 (2) must hold for each of them. One can verify that this is true if

$$\mathcal{L}(e)\Sigma^* \cap \mathit{Tail}_C(E) = \varnothing. \tag{5}$$

The rules for computing $\mathcal{T}_C(e)$ can be similarly extended to the full PEG.

The terminals in full PEG are not necessary single letters from $\Sigma^*$, and may be multi-letter quoted strings, or sets of letters defined as `[abc]` or `[a-z]`. 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 Explorer

The experience shows that (3) can often be verified by inspection. Giving up all hope for an automatic verification, the author created an experimental tool, the *PEG Explorer*, to facilitate such inspection.

Using known methods one can compute sets of letters that appear as first in strings from $\mathcal{L}(e_1)\Sigma^*$ respectively $\mathcal{L}(e_2)\,\mathcal{T}_C(e)$. Condition (3) is then obviously satisfied if these sets are disjoint. This is the familiar LL(1) condition. The Explorer takes a grammar, tests all choice expressions for LL(1) and presents for inspection those that did not pass the test.

## 4.1 Example 1: Simple calculator

To give some idea of the Explorer, we apply it to the following grammar:

```
Sum      = Product ("+" Product)*
Product  = Factor ("*" Factor)*
Factor   = Digits? Fraction | Digits | "(" Sum ")"
Fraction = "." Digits
Digits   = [0-9]+
```

With $S$ = `Sum`, it defines the syntax of a primitive calculator. When Explorer 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 showing this text:

```
Digits? Fraction
====================
Digits Tail(Factor)
```

It is an invitation to verify `Digits? Fraction` $\Sigma^* \cap$ `Digits` $\mathcal{T}_S$(`Factor`) = $\varnothing$. The second line is a pseudo-expression with `Tail(Factor)` representing $\mathcal{T}_S$(`Factor`). A click on `Digits?` replaces it by the two alternatives represented by `Digits?`, namely `Digits` and $\varepsilon$:

```
Digits Fraction
Fraction
====================
Digits Tail(Factor)
```

A click on Explorer button "Filter" leaves only the alternatives not satisfying LL(1):

```
Digits Fraction
====================
Digits Tail(Factor)
```

Clicking on `Fraction` replaces it by `"." Digits`. Clicking on `Tail(Factor)` replaces it by all its alternatives according to Proposition 2:

```
Digits "." Digits
====================
Digits ")" Tail(Factor.3.3)
Digits ("+" Product)* Tail(Sum.2)
Digits "+" Product Tail(Sum.2.1)
Digits ("*" Factor)* Tail(Product.2)
Digits "*" Factor Tail(Product.2.1)
```

(The numbers after expression names identify embedded expressions, for example, `Sum.2.1` means the first subexpression of the second subexpression of `Sum`.)
Clicking on Explorer button "Strip" removes `Digits` from further investigation:

```
Digits -- "." Digits
====================
Digits -- ")" Tail(Factor.3.3)
Digits -- ("+" Product)* Tail(Sum.2)
Digits -- "+" Product Tail(Sum.2.1)
Digits -- ("*" Factor)* Tail(Product.2)
Digits -- "*" Factor Tail(Product.2.1)
```

Finally, clicking on "Filter" produces this:

```
====================
```

meaning that expressions to the right of "--" satisfy LL(1) and the investigated choice is disjoint. The grammar has efficient backtracking.

Note that the grammar has the property called LL(2P) in [7]: The parser can choose its way by looking at the input within the reach of two parsing procedures. To choose between the first two alternatives of `Factor`, it has to first call `Digits`, and if this succeeds, either proceed or backtrack, depending on the outcome of `"."`.

## 4.2 Example 2: Grammar with predicates

The second example illustrates treatment of predicates. The following is a fragment of larger grammar that uses identifiers, with some of them being reserved as ”keywords”. Only two keywords, `"interface"` and `"int"` are shown. A keyword 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    = ("interface" | "int") !Letter
Identifier = !Keyword Letter+
Letter = [a-z]
Number = [0-9]+
```

The Explorer applied to this grammar indicates that the choices in `Statement` and `Keyword` do not satisfy LL(1). For the first it opens a window showing this text:

```
Keyword Number
====================
Identifier Number Tail(Statement.1)
```

Clicking on `Identifier` results in replacing it by its definition:

```
Keyword Number
====================
!Keyword Letter+ Number Tail(Statement.1)
```

It is now clear why Explorer signaled an LL(1) violation: as `!Keyword` may consume empty string, `!Keyword Letter+` may start with a letter. And so does `Keyword`. But one can easily see that no string beginning with `Keyword` can be a prefix of any string defined by `!Keyword Letter+`. Which proves that the choice in `Statement` is disjoint.

For the choice in `Keyword`, Explorer shows this:

```
"interface"
====================
"int" Tail(Keyword.1)
```

Again, the terminals `"interface"` and `"int"` are not disjoint, so Explorer signaled this as LL(1) violation. A click on `Tail(Keyword1)` has this result:

```
"interface"
====================
"int" !Letter Any
```

where ”`Any`” means ”any string”. One can see that `"interface"` cannot be a prefix of any string accepted by `"int" !Letter Any`, so also this choice is disjoint.

## 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 `"*"` by `"*"?`:

```
Sum      = Product ("+" Product)*
Product  = Factor ("*"? Factor)*
Factor   = Digits? Fraction | Digits | "(" Sum ")"
Fraction = "." Digits
Digits   = [0-9]+
```

The Explorer applied to the modified grammar shows now two cases not satisfying LL(1). One of them is the first choice in `Factor`, exactly as in Example 1:

```
Digits? Fraction
====================
Digits Tail(Factor)
```

But, after a sequence of steps, we obtain this:

```
Digits -- Fraction
====================
Digits -- Fraction ("*"? Factor)* Tail(Product.2)
Digits -- Fraction Tail(Product.2.1)
```

This shows that the choice is not disjoint. With omitted `*`, two `Factors` may follow directly after each other, like this: `1.23.4`, which may mean either `1.2*3.4` or `1.23*.4`. Our grammar is ambiguous: there are two EBNF interpretations, and PEG will choose one of them, namely the second.

Another LL(1) violation is `Digits` against `Tail(Digits)` which indicates a similar ambiguity. We might accept these ambiguities if we find that the PEGs choice coincides with human perception.

# 5 Final remarks

Not being able to investigate equivalence of EBNF and PEG with predicates, we restricted ourselves to a guess that "eficient backtracking" makes PEG easier to understand also in the presence of predicates. A general procedure for checking the eficiency condition is not possible but the example of PEG Explorer shows that it can often be checked by inspection.

The author believes that it is possible to replace predicates by a less powerful feature that would be: compatible with EBNF, recognizable by limited backtracking, and sitll fill the role played by predicates in the definition of programming languages. This should be the subject of further research.

# A Appendix

## A.1 Proof of Proposition 1

Assume $\mathbb{G}$ satisfies the stated conditions. Take any $u, v \in \Sigma^*$ such that $[C]\ u \overset{\text{PEG}}{\rightsquigarrow} v$. By (1), we have $[C]\ u \overset{\text{BNF}}{\rightsquigarrow} v$. We are going to show that for each result $[e]\ xy \overset{\text{BNF}}{\rightsquigarrow} y$ in the proof tree of $[C]\ u \overset{\text{BNF}}{\rightsquigarrow} v$, including the final result, there exists a proof of $[e]\ xy \overset{\text{PEG}}{\rightsquigarrow} y$. We use induction on the height of the proof tree.

(Induction base) Suppose the proof of $[e]\ xy \overset{\text{BNF}}{\rightsquigarrow} y$ has height 1. 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* or *letter1*.

(Induction step) Assume that for every 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 result $[e]\ xy \overset{\text{BNF}}{\rightsquigarrow} y$ with proof tree of height $n+1$. It must be one of these:

- $[A]\ xy \overset{\text{BNF}}{\rightsquigarrow} y$ derived from $[e(A)]\ xy \overset{\text{BNF}}{\rightsquigarrow} y$, using *rule.b*. By induction hypothesis, $[e(A)]\ xy \overset{\text{PEG}}{\rightsquigarrow} y$, so $[A]\ xy \overset{\text{PEG}}{\rightsquigarrow} y$ follows from *rule*.
- $[e_1 e_2]\ xyz \overset{\text{BNF}}{\rightsquigarrow} z$ derived from $[e_1]\ xyz \overset{\text{BNF}}{\rightsquigarrow} yz$ 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 $[e_1 e_2]\ xyz \overset{\text{PEG}}{\rightsquigarrow} z$ follows from *seq1*.
- $[e_1 / e_2]\ xy \overset{\text{BNF}}{\rightsquigarrow} y$ derived from$[e_1]\ xy \overset{\text{BNF}}{\rightsquigarrow} y$ using *choice1.b*. By induction hypothesis, $[e_1]\ xy \overset{\text{PEG}}{\rightsquigarrow} y$, so $[e_1 / e_2]\ xy \overset{\text{PEG}}{\rightsquigarrow} y$ follows from *choice1*.
- $[e_1 / e_2]\ xy \overset{\text{BNF}}{\rightsquigarrow} y$ derived from $[e_2]\ xy \overset{\text{BNF}}{\rightsquigarrow} y$ using *choice2.b*. By induction hypothesis, $[e_2]\ xy \overset{\text{PEG}}{\rightsquigarrow} y$. But, to use *choice2*, we need to verify that $[e_1]\ xy \overset{\text{PEG}}{\rightsquigarrow} \texttt{fail}$.
  Suppose there is no proof of $[e_1]\ xy \overset{\text{PEG}}{\rightsquigarrow} \texttt{fail}$. As $\mathbb{G}$ is not left-recursive, there exists a proof of $[e_1]\ wz \overset{\text{PEG}}{\rightsquigarrow} z$ where $wz = xy$. By definition, $w \in \mathcal{L}(e_1)$ and $x \in \mathcal{L}(e_2)$. As $[e]\ xy \overset{\text{BNF}}{\rightsquigarrow} y$ appears in the proof of $[C]\ u \overset{\text{BNF}}{\rightsquigarrow} v$, we have $y \in \mathit{Tail}_C(e)$. From this follows $\mathcal{L}(e_1)\Sigma^* \cap \mathcal{L}(e_2)\mathit{Tail}_C(e) \neq \varnothing$, which contradicts (2). We must thus conclude that there exists a proof of $[e_1]\ xy \overset{\text{PEG}}{\rightsquigarrow} \texttt{fail}$, so there exists a proof of $[e_1 / e_2]\ xy \overset{\text{PEG}}{\rightsquigarrow} y$ using *choice2*.

## A.2 Approximating `Tail`

For a pair of expressions $(e, E)$ such that $[E]\ x' \overset{\text{BNF}}{\rightsquigarrow} y'$ can be derived from $[e]\ x \overset{\text{BNF}}{\rightsquigarrow} y$,
define $T(e, E) = \mathcal{L}^E(e_2)$ if $E = e_1\, e_2$, or $T(e, E) = \{\varepsilon\}$ otherwise.

**Lemma 1.** *If $[E]\ x' \overset{BNF}{\rightsquigarrow} y'$ can be derived from $[e]\ x \overset{BNF}{\rightsquigarrow} y$ then $y \in T(e, E)\, y'$.*

*Proof.* In all cases except one using *seq.b*, $[E]\ xy \overset{\text{BNF}}{\rightsquigarrow} y$ is derived from $[e]\ xy \overset{\text{BNF}}{\rightsquigarrow} y$, so $y = \{\varepsilon\}\, y'$ and $\varepsilon = \mathcal{T}(e, E)$.
In case of *seq.b*, $[E]\ stu \overset{\text{BNF}}{\rightsquigarrow} u$ is derived from $[e_1]\ stu \overset{\text{BNF}}{\rightsquigarrow} tu$ and $[e_2]\ tu \overset{\text{BNF}}{\rightsquigarrow} u$. Case $e = e_2$ is the same as before. In case $e = e_1$ we have $y' = u$, $y = tu$, and $t \in T(e_2) = \mathcal{L}^E(e, E)$. That means $y \in T(e, E)y'$. □

Define:

- $\mathcal{T}_C(C) = \Sigma^*$.
- For $e \neq C$: $\mathcal{T}_C(e) = T(e, E_1)\mathcal{T}_C(E_1) \cup \cdots \cup T(e, E_n)\mathcal{T}_C(E_n)$ where $E_1, \ldots, E_n$ are all expressions accessible from $C$ that contain $e$.

**Proposition 2.** $\mathit{Tail}_C(e) \subseteq \mathcal{T}_C(e)$.

*Proof.* Consider any $y \in \mathit{Tail}_C(e)$. By definition, exists $[e]\ x \overset{\text{BNF}}{\rightsquigarrow} y$ that appears in a proof of $[C]\ w \overset{\text{BNF}}{\rightsquigarrow} z$. It is first in the chain of $n$ results $[E_i]\ x_i \overset{\text{BNF}}{\rightsquigarrow} y_i$ where $E_1 = C$ and $E_n = e$. We show that $y_i \in \mathcal{T}_C(E_i)$.
$[E_1]\ x_1 \overset{\text{BNF}}{\rightsquigarrow} y_1$ is the same as $[C]\ w \overset{\text{BNF}}{\rightsquigarrow} z$. We have $y_1 \in \mathcal{T}_C(C) = \Sigma^*$.
Suppose we found that $y_i \in \mathcal{T}_C(E_i)$ for some $1 \leq i < n$. The result $[E_i]\ x_i \overset{\text{BNF}}{\rightsquigarrow} y_i$ is derived from $[E_{i+1}]\ x_{i+1} \overset{\text{BNF}}{\rightsquigarrow} y_{i+1}$. By Lemma 1, $y_{i+1} \in T(e_{i+1}, E_i)\, y_i$ so $y_{i+1} \in T(e_{i+1}, E_i)\mathcal{T}_C(E_i)$. As $E_i$ is clearly accessible from $C$, $T(e_{i+1}, E_i)\mathcal{T}_C(E_i) \subseteq \mathcal{T}_C(E_{i+1})$, so $y_{i+1} \in \mathcal{T}_C(E_{i+1})$.
By induction, $y_n \in \mathcal{T}_C(e_n)$, that is, $y \in \mathcal{T}_C(e)$. □

# References


[1] Ford, B.: *Packrat Parsing: a Practical Linear-Time Algorithm with Backtracking*, Master Thesis, Massachusetts Institute of Technology, September 2002, `http://pdos.csail.mit.edu/papers/packrat-parsing:ford-ms.pdf`.

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

[3] Ford, B.: Parsing Expression Grammars: A Recognition-Based Syntactic Foundation, *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2004* (N. D. Jones, X. Leroy, Eds.), 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**, 2014, 235–250.

[5] Medeiros, S.: *Correspondência entre PEGs e Classes de Gramáticas Livres de Contexto*, Ph.D. Thesis, Pontifícia Universidade Católica do Rio de Janeiro, August 2010.

[6] Redziejowski, R. R.: From EBNF to PEG, *Fundamenta Informaticae*, **128**, 2013, 177–191.

[7] Redziejowski, R. R.: More about converting BNF to PEG, *Fundamenta Informaticae*, **133**(2-3), 2014, 177–191.