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, but it is not well understood as a language definition tool. In its appearance, PEG is almost identical to a grammar in the Extended Backus-Naur Form (EBNF), and one may expect it to define the same language. But, due to the limited backtracking, PEG may reject some strings defined by EBNF, which gives an impression of PEG being unpredictable. We note that for some grammars, the limited backtracking is "efficient", in the sense that it exhausts all possibilities. A PEG with efficient backtracking should therefore be easy to understand. There is no general algorithm to check if the grammar has efficient backtracking, but it can be often checked by inspection. The paper outlines an interactive tool to facilitate such inspection.

1 Introduction

One of the most common ways to define the syntax of a formal language is the Extended Backus-Naur Form (EBNF). It originated with the development of Algol in the 1960's. Already at that time it was suggested to parse the language so defined by a collection of procedures that mirror the EBNF rules and call each other recursively [5]. In this approach, known as top-down parsing by recursive descent, the syntax definition becomes identical with its parser.

The problem arises when parsing procedure must choose the next procedure to call. For some grammars, this can be done by looking at the next input symbol. Such grammars are said to have the LL(1) property. An alternative is to proceed by trial and error: examine all alternatives one by one, backtracking in the input until a correct one is found. This no longer depends on any property of the grammar, but an exhaustive search may result in exponential time. A reasonable compromise is *limited backtracking*: once an alternative succeeded, never return to try another one.

The problem with limited backtracking is that by not trying hard it may miss some inputs that it should accept. A notorious example is the rule `A = aAa | aa` that defines the set of strings of `a`'s of even length. Implemented with limited backtracking, this rule accepts only strings of length 2^n.

Parsing Expression Grammars (PEGs) have been introduced by Ford in [4] as a new formalism for describing syntax of programming languages. The formalism encodes a recursive-descent parser with limited backtracking. With "packrat" technology described in [2, 3], this parser can work in a linear time. As reported in [8], it can work in a nearly constant space.

PEG is useful because it avoids the LL(1) restriction and directly provides a parser that may integrate the functions otherwise provided by a separate "scanner". But, it is not well understood as a language definition tool.

In its appearance, PEG is practically identical to EBNF, only extended with an additional parsing operation, the "syntactic predicate". One of the elements of PEG is the "choice expression" e_1 / e_2. To anybody familiar with EBNF, it should define the union of languages defined by e_1 and e_2. But, due to the limited backtracking, the PEG choice may exclude part, or all, of the language of e_2. This exclusion is not always obvious, which gives an impression of PEG being unpredictable.

We note that the exclusion does not occur if the choice expression is "disjoint". We may say that in this case the limited backtracking is "efficient": it finds everything that would be found by full backtracking. A PEG with efficient backtracking defines thus exactly what would be intuitively expected by anybody familiar with EBNF. Which should make it easy to understand.

*To appear in *Fundamenta Informaticae*.

It has been formally proved [6, 7, 10] that PEG without predicates is equivalent to EBNF if it has efficient backtracking. As there is no way to include syntactic predicates in EBNF, the proof cannot be extended to PEG with predicates, but we conjecture that even in this case, PEG with efficient backtracking is sufficiently close to intuition.

Unfortunately, there is no general way to check disjointness of choice expressions. However, it can be often verified by inspection. We outline an interactive tool, called *PEG Explorer*, to assist such inspection.

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. The proofs, and the rules for approximating "tail" have been placed in Appendix.

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:

ε	("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 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.
$e_1 \mid e_2$	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 [6, 7]. 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. By definition, $[e]\ xy \overset{\text{PEG}}{\rightsquigarrow} y$, 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

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

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

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

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

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

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

Figure 2: Formal semantics of PEG

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 [6, 7] by checking that PEG defined by natural semantics is equivalent to that defined by Ford in [4] and using the result from there. An independent proof for grammar without predicates is given in [10]. 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.

We define 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 demonstrated in [6, 7] for PEG without predicates:

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

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

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

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

Figure 3: Formal semantics of BNF

3 Disjoint choice and efficient 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:

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

It states, in effect, that limited backtracking is efficient in the context of C: once e_1 succeeded, trying e_2 is doomed to failure.

Proposition 1. *Assume that $\mathbb{G}$ 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. Together with (1), the Proposition says that these two interpretations are identical if limited backtracking is efficient in the context of S. Which is not surprising since the difference between EBNF and PEG is caused by limited backtracking not exploring some paths. (This is just another way to state the result found in [10].)

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

Condition (2) may involve checking the disjointness of two context-free languages, for which no algorithm exists. Thus, there is no general mechanical way to check it for a given grammar. But, in many practical cases one can solve the problem by approximating the languages "from above".

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):

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

Let $F_1, F_2 \subseteq \Sigma^*$ be such that $F_1\Sigma^* \supseteq \mathcal{L}(e_1)\Sigma^*$ and $F_2\Sigma^* \supseteq \mathcal{L}(e_2)T(e)$. Condition (3) is obviously satisfied if $F_1\Sigma^* \cap F_2\Sigma^* = \varnothing$.

The challenge is to choose F_1 and F_2. One possible choice is to use the sets of possible first letters of $\mathcal{L}(e_1)$ respectively $\mathcal{L}(e_2)T_C(e)$. The property of these sets being disjoint is the same as the LL(1) property. In the absence of predicates, the sets of first letters can be computed by known procedures [1, 12]. Assuming that all predicates succeed (that is, replacing them by ε) introduces an additional approximation "from above" and makes the classical procedures available also in the presence of predicates.

An expression can satisfy (3) even if it does not have the LL(1) property. This can be often verified by inspection. In Section 4 we present an experimental tool to facilitate such inspection.

The full grammar

The grammar $\mathbb{G}$ considered up to now is a simplified version of full PEG. This latter allows expressions such as $e_1 \,|\, e_2 \,|\, \ldots \,|\, e_n$, $e_1 e_2 \ldots e_n$, 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 Σ^*, 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 interactive tool, the *PEG Explorer*, to facilitate such inspection.

The Explorer takes a grammar, tests all choice expressions for disjointness of first terminals (referred to in the following as "LL(1) test") and presents for inspection those that did not pass the test.

A detailed description of Explorer is found in [11]. The examples below should give some feeling of how it works.

Example 1: Primitive calculator

The following grammar defines the syntax of a primitive calculator ("eot" stands for end of text).

```
Sum     = Sign Number (AddOp Number)* eot
Number  = Digits? "." Digits | Digits
Sign    = "-"?
AddOp   = [-+]
Digits  = [0-9]+
```

When Explorer is applied to this grammar, it opens, after a couple of steps, a window showing this text:

```
Number: Digits? "." Digits <==> Digits Tail(Number)

  [0-9]  <==>  [0-9]

Digits? "." Digits
====================
Digits Tail(Number)
```

It tells that choice between the two alternatives of `Number` did not pass the LL(1) test because both alternatives may start with `[0-9]`.
The lower part is an invitation to verify that the language defined by `(Digits? "." Digits)`Σ^* has an empty intersection with that defined by `Digits Tail(Number)`. In other words, that `Digits? "." Digits` cannot define a prefix of any string in `Digits Tail(Number)`.

By clicking on an expression, you replace it by its definition. A click on `Digits?` replaces it by `Digits | `ε so that the expression above the line becomes

```
Digits "." Digits | "." Digits
```

which is presented like this:

```
Digits "." Digits
"." Digits
====================
Digits Tail(Number)
```

One can easily see that `"." Digits` cannot be a prefix of expression under the line, so it can be ignored. To facilitate this, Explorer has a button "Filter" that removes such alternatives. A click on that button has this result:

```
Digits "." Digits
====================
Digits Tail(Number)
```

With both expressions starting with `Digits`, we need now to check whether the remaining part of the first line can be a prefix of the remaining part of the other. To facilitate this, Explorer has a button "Strip" that removes identical initial parts. Clicking on that button produces this:

```
Digits -- "." Digits
====================
Digits -- Tail(Number)
```

We need to see now what is at the beginning of `Tail(Number)`. A click on `Tail(Number)` replaces it by the approximation obtained in the way described in the Appendix:

```
Digits -- "." Digits
====================
Digits -- (AddOp Number)* eot
```

We can see that `(AddOp Number)* eot` cannot start with `"."`, so it cannot have `"." Digits` as prefix. The examined choice expression turns out to be disjoint. As it was the only one to be checked, the grammar has efficient backtracking.

Note that our grammar has the property that was called LL(2p) in [10]: the parser can choose its way by looking at the input within the reach of two parsing procedures. To choose between the two alternatives of `Number`, it has to first call `Digits`, and if this succeeds, either proceed or backtrack depending on the outcome of `"."`.

Example 2: Non-disjoint choice

The following grammar defines another calculator, with addition, multiplication, and parentheses. The multiplication operator may be omitted so one can write, for example `2(.3+4)` instead of `2*(.3+4)`.

```
Start    = Sum !_
Sum      = Product ("+" Product)*
Product  = Factor ("*"? Factor)*
Factor   = Digits? "." Digits | Digits | "(" Sum ")"
Digits   = [0-9]+ ;
```

The Explorer applied to this grammar finds two cases that did not pass the LL(1) test. It shows this for one of them:

```
Digits? "." Digits
====================
Digits Tail(Factor)
```

It is about the first choice in `Factor`, where both alternatives may start with `[0-9]`. After a couple of clicks we arrive at this:

```
Digits -- "." Digits
====================
Digits -- Tail(Factor)
```

After some more clicks we have:

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

One could now click on `Factor`, but a look at the grammar shows that `Factor` may start with `"."Digits`. 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. We might accept this ambiguity if we find that the PEGs choice coincides with human perception.

Example 3: Grammar with predicates

This 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 that prefix of an identifier is not recognized as a keyword. 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 shows this:

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

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

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

It is now clear why this choice did not pass the LL(1) test: `!Keyword` may define empty string, so both alternatives may start with `Letter`.
But one can easily see that no string beginning with `Keyword` can be a prefix of any string defined by `!Keyword Letter+`. Which shows that this is a false alarm and the choice in `Statement` is disjoint.

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

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

The terminals `"interface"` and `"int"` are not disjoint, so Explorer signals this as LL(1) violation. (`Keyword.1` is the first subexpression of `Keyword`.) A click on `Tail(Keyword.1)` has this result:

```
"interface"
====================
"int" !Letter Tail(Keyword)
```

One can see that `"interface"` cannot be a prefix of anything starting with `"int" !Letter`, so this is again a false alarm and the choice in `Keyword` is disjoint.

5 Final remarks

We noted that the condition for equivalence of EBNF and PEG without predicates found in [7] and generalized in [10] can be viewed as a requirement that the limited backtracking of PEG is ”efficient”. Using a new definition of ”tail”, we extended this condition to PEG with predicates, and believe that PEG with efficient backtracking is easier to understand.

A general procedure for checking the condition is not possible, but it can often be checked by inspection. We presented an interactive tool that facilitates such inspection.

The inspection may result in additional insights into the working of PEG parser. In Example 1 we find that PEG parser can choose its way by examining the input within the reach of two parsing procedures. In Example 2 we find that backtracking is not efficient because it omits one of the alternatives permitted by an ambiguous EBNF grammar. The inspection helps to understand which alternative is chosen by the PEG parser.

Example 3 shows that presence of predicates produces false alarms that have to be resolved by inspection. In languages using reserved words, like Java, such false alarms constitute a large percentage of cases presented for inspection.

Predicates are mainly used to define low-level structures that otherwise need a separate "scanner". The question is if they can be replaced in this role by new operators that would simplify mechanical analysis. For example, the use of predicates to define keywords could be replaced by an operator inspired by Boolean grammars of Okhotin [9]. New expressions such as $e_1 :\& e_2$ and $e_1 :! e_2$ could mean: apply e_1 and then check if the consumed text conforms, respectively does not conform to e_2.

Another frequent use of predicates is the construction $(!e_1 e_2)^* e_1$ that could be replaced by an "until" operator.

It is a subject of further research and experiments to identify such operators, embed them in PEG, possibly in EBNF, define formally, and handle in testing of (3).

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* $y \subseteq T(e, E)\, y'$.

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. *$Tail_C(e) \subseteq \mathcal{T}_C(e)$.*

Proof. Consider any $y \in Tail_C(e)$. 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 $i \geq 1 < n$. 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] Aho, A. V., Sethi, R., Ullman, J. D.: *Compilers, Principles,Techniques, and Tools*, Addison-Wesley, 1987.

[2] 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`.

[3] 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.

[4] 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.

[5] Lucas, P.: The Structure of Formula-Translators, *ALGOL Bulletin Supplement*, **16**, September 1961, 1–27.

[6] 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.

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

[8] Mizushima, K., Maeda, A., Yamaguchi, Y.: Packrat parsers can handle practical grammars in mostly constant space, *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* (S. Lerner, A. Rountev, Eds.), ACM, 2010.

[9] Okhotin, A.: Boolean grammars, *Inf. Comput.*, **194**(1), 2004, 19–48.

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

[11] Redziejowski, R. R.: PEG Explorer, 2017, `http://mousepeg.sourceforge.net/explorer.htm`.

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