Cut points in PEG*

Roman R. Redziejowski

Abstract

Parsing Expression Grammar (PEG) encodes a recursive-descent parser with limited backtracking. It has been recently noticed that in the situation when the parser is to explore several alternatives one after another, no further alternatives need to be explored after the parser reached certain "cut point". This fact can be used to save both processing time and storage. The subject of the paper is identification of cut points, which can also help in producing better diagnostics.

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 is recognition-based; in fact, PEG can be seen as encoding of a recursive-descent parser with limited backtracking. Backtracking lifts the LL(1) restriction usually imposed on top-down parsers. The backtracking being limited means that the parser can work in linear time, at the cost of large memory consumption. This is achieved with the help of "memoization" or "packrat" technology described in [1, 2], The technology consists in saving all partial results for reuse after backtracking.

Each PEG is a collection of rules such as the following:

E = a + b / a c .

This rule encodes a parsing procedure E that is supposed to try the two alternatives separated by /. The first alternative is to consume the consecutive input symbols: a, +, and then b. If E succeeds with this, its task is completed and E returns success. Otherwise, it backtracks and tries the second alternative, which means consuming a and then c.

Suppose that E, applied to some string w, succeeds with a and +, after which it does not find b. As w was verified to start with a+, the second alternative must fail by not finding c after a.

Suppose instead that E fails already with the first a. It means that w does not start with a, so the second alternative must also fail.

We have identified here two points in the first alternative, indicated below by $\downarrow$ and $\uparrow$, such that the second alternative is bound to fail if the first one fails before $\downarrow$ or after $\uparrow$:

E = a$\downarrow$ + $\uparrow$b / a c .

These points are in the following called the *cut points*.

If the parser is constructed never to call the second alternative that is doomed to fail, the string scanned by the first alternative will never be rescanned. It means that packrat parser will not need any saved results after passing $\uparrow$. This observation was used by Mizushima et al. [8] to make the PEG parser work in a nearly linear space. Improving parser performance is thus a good reason to identify cut points in a given grammar.

Another good reason is improving error diagnostics. A failure of parsing procedure need not be caused by a syntax error; it may just be a legitimate start of backtracking. A meaningful error reporting should distinguish between

*Final version appeared in *Fundamenta Informaticae 143* 1-2 (2016), 141–149.

the two kinds of failures. Knowing the cut points may provide a tool for such distinction: a failure before $\downarrow$ or after $\uparrow$ in E indicates syntax error, while a failure between them is just a signal to try another alternative.

The method of identifying cut points used in [8] applies to LL(1) grammars. We discuss here how to identify cut points in a more general case. We use for this purpose a restricted form of PEG, presented in Section 2 below.

In Section 3 we give sufficient conditions for placement of the two kinds of cut points. Unfortunately, these conditions involve inclusion and emptiness of intersection of context-free languages. These are in general undecidable, so there is no general mechanical way to use them. As a solution we suggest, in Section 4, to approximate the involved languages by their "first sets".

In Section 5 we show how knowledge of cut points can be used to provide meaningful diagnostics using the technique of "labeled failures" developed by Maidl et al. in [5].

Finding a cut point requires that the first alternative is a sequence with sufficient granularity. Mizushima et al. [8] circumvent this requirement using the PEG "not-predicate" to imitate the effect of cut point $\uparrow$. We indicate in Section 6 how this method can be extended to non-LL(1) grammars.

The closing Section 7 discusses limitations, problems, and possible further work.

2 Parsing Expression Grammar

We consider a grammar $\mathbb{G}$ over alphabet Σ. It is a simplified PEG without predicates. 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,
- $a \in \Sigma$,
- $A \in N$,
- $e_1 e_2$ ("sequence"),
- e_1 / e_2 ("choice"),

where each of e_1, e_2 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 sequence operator binds stronger than choice.

The expressions represent parsing procedures, and rules represent named such 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 "consumes" the recognized portion and returns "success"; otherwise, it returns "failure" and does not consume anything. The action of different procedures is as follows:

- 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 its result.
- $e_1\, e_2$: 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.

This can be defined formally in the way introduced by [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. We define 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 1.

The grammar has a unique *start symbol* $S \in N$ with the corresponding rule $S = e\,\$$ where $e \in \mathbb{E}$ and $\$ \in \Sigma$ is the *end-of-text marker* that appears only in this rule. A string w is *accepted* by $\mathbb{G}$ if $[S]\ w \overset{\text{PEG}}{\rightsquigarrow} \varepsilon$ can be proved using the rules from Figure 1. The proof tree of $[S]\ w \overset{\text{PEG}}{\rightsquigarrow} \varepsilon$ mimics the process of parsing the string w.

Grammar $\mathbb{G}$ may be alternatively interpreted as a grammar in Backus-Naur Form (BNF) where / stands for unordered choice. Denoting by $\mathcal{L}(e)$ the language defined by expression e interpreted as BNF, we have thus:

$$\mathcal{L}(\varepsilon) = \{\varepsilon\},\ \mathcal{L}(a) = \{a\},\ \mathcal{L}(A) = \mathcal{L}(e(A)),\ \mathcal{L}(e_1e_2) = \mathcal{L}(e_1)\mathcal{L}(e_2),\ \mathcal{L}(e_1/\,e_2) = \mathcal{L}(e_1) \cup \mathcal{L}(e_2).$$

The BNF interpretation of $\mathbb{G}$ plays a central role in our results. The following was proved in [6,7,9]:

Proposition 2.1. For each $e \in \mathbb{E}$ and $xy \in \Sigma^*$, if $[e]\ xy \overset{\text{PEG}}{\rightsquigarrow} y$ then $x \in \mathcal{L}(e)$.

It says, in particular, that if $e \in \mathbb{E}$ succeeds on $w \in \Sigma^*$ then $w \in \mathcal{L}(e)\Sigma^*$. However, it does not say what happens if e fails on w. One might expect that in such a case $w \notin \mathcal{L}(e)\Sigma^*$, but this is in general not true. As an example, consider these rules:

```
A = Bb     B = a/aa .
```

Expression `A` fails when applied to $w =$ `aabb` because `B` consumes `a` after which `b` fails on `abb`. But $\mathcal{L}(\texttt{A}) = \{\texttt{ab}, \texttt{aab}\}$, so $w \in \mathcal{L}(\texttt{A})\Sigma^*$. (The other alternative of `B` is never tried once `B` succeeded; this is an example of limited backtracking.)

We shall say that expression e is *strong* if $[e]\ w \overset{\text{PEG}}{\rightsquigarrow} \texttt{fail}$ implies $w \notin \mathcal{L}(e)\Sigma^*$ for each $w \in \Sigma^*$. One can verify by induction on height of the proof of $[e]\ w \overset{\text{PEG}}{\rightsquigarrow} \texttt{fail}$ that e is strong if each sequence expression involved in its definition is strong. In particular, e is strong if its definition does not use sequence expressions.

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

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

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

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

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

Figure 1: PEG semantics

3 Finding cut points

As defined in the Introduction, cut point in a rule of the form $A = e_1e_2 \ldots e_n/e_{n+1}$ is defined by i such that failure of $e_1 \ldots e_i$ implies failure of A (cut point $\downarrow$), or failure of $e_{i+1} \ldots e_n$ after success of $e_1 \ldots e_i$ implies failure of A (cut point $\uparrow$). By representing $e_1e_2 \ldots e_n$ as $(e_1 \ldots e_i)(e_{i+1} \ldots e_n)$ for consecutive i, we reduce the problem of finding cut points to answering the question if we have a cut point between e_0 and e_1 in a rule of the form $A = e_0\, e_1/\, e_2$. We need one more definition to answer that question:

Definition 3.1. For $A \in N$, $\text{Tail}(A)$ is the set of such $y \in \Sigma^*$ that $[A]\ xy \overset{\text{PEG}}{\rightsquigarrow} y$ appears as partial result in the proof of $[S]\ w \overset{\text{PEG}}{\rightsquigarrow} \varepsilon$ for some $w \in \Sigma^*$.

In other words, $\text{Tail}(A)$ is the set of terminated input strings that may follow a substring consumed by A in a string accepted by $\mathbb{G}$. We are now ready to formulate conditions for inserting $\uparrow$ or $\downarrow$ between e_0 and e_1.

Proposition 3.2. A sufficient condition for $\uparrow$ after e_0 in $A = e_0\, e_1/\, e_2$ is:

$$\mathcal{L}(e_0)\Sigma^* \cap \mathcal{L}(e_2)\,\text{Tail}(A) = \varnothing. \tag{1}$$

Proof. Let A satisfy (1). Suppose there exists a successful parse of a string w where A is called at one of the steps, and after a successful call to e_0 it calls e_1 that fails. We show that A also fails.

The parse is described formally by a proof of $[S]\ w \overset{\text{PEG}}{\rightsquigarrow} \varepsilon$ that contains as a partial result $[A]\ xy \overset{\text{PEG}}{\rightsquigarrow} Y$ obtained from $[e_0]\ xy \overset{\text{PEG}}{\rightsquigarrow} z$, $[e_1]\ z \overset{\text{PEG}}{\rightsquigarrow} \texttt{fail}$ and $[e_2]\ xy \overset{\text{PEG}}{\rightsquigarrow} Y$ using **seq1**, **choice2**, and **rule**.

According to **choice2**, either $Y = y$ or $Y = \texttt{fail}$. Suppose $Y = y$, so $[e_2]\ xy \overset{\text{PEG}}{\rightsquigarrow} y$ and $[A]\ xy \overset{\text{PEG}}{\rightsquigarrow} y$. By Proposition 2.1, $x \in \mathcal{L}(e_2)$. By Definition 3.1, $y \in \text{Tail}(A)$ so $xy \in \mathcal{L}(e_2)\,\text{Tail}(A)$. By Proposition 2.1, $[e_0]\ xy \overset{\text{PEG}}{\rightsquigarrow} z$ means that $xy \in \mathcal{L}(e_0)\Sigma^*$. This contradicts (1), so must be $Y = \texttt{fail}$. □

Proposition 3.3. A sufficient condition for $\downarrow$ after e_0 in $e_0\, e_1/e_2$ is:

$$e_0 \text{ is strong and } \mathcal{L}(e_0)\Sigma^* \supseteq \mathcal{L}(e_2)\,\text{Tail}(A). \tag{2}$$

Proof. Let A satisfy (2). Suppose there exists a successful parse of a string w where A is called at one of the steps and calls e_0 that fails. We show that A also fails.

The parse is described formally by a proof of $[S]\ w \overset{\text{PEG}}{\rightsquigarrow} \varepsilon$ that contains as a partial result $[A]\ xy \overset{\text{PEG}}{\rightsquigarrow} Y$ obtained from $[e_0]\ xy \overset{\text{PEG}}{\rightsquigarrow} \texttt{fail}$ and $[e_2]\ xy \overset{\text{PEG}}{\rightsquigarrow} Y$ using **seq2**, **choice2**, and **rule**.

According to **choice2**, either $Y = y$ or $Y = \texttt{fail}$. Suppose $Y = y$. Then $[e_2]\ xy \overset{\text{PEG}}{\rightsquigarrow} y$ and $[A]\ xy \overset{\text{PEG}}{\rightsquigarrow} y$. By Proposition 2.1, $x \in \mathcal{L}(e_2)$. By Definition 3.1, $y \in \text{Tail}(A)$ so $xy \in \mathcal{L}(e_2)\,\text{Tail}(A)$. From e_0 being strong and $[e_0]\ xy \overset{\text{PEG}}{\rightsquigarrow} \texttt{fail}$ follows $xy \notin \mathcal{L}(e_0)\Sigma^*$. This contradicts (2), so must be $Y = \texttt{fail}$. □

As an example, take the rule E = a + b / a c from the Introduction. For this rule, the conditions (1) and (2) are, respectively, $\{\texttt{a+}\}\Sigma^* \cap \{\texttt{ac}\}\,\text{Tail(E)} = \varnothing$ and $\{\texttt{a}\}\Sigma^* \supseteq \{\texttt{ac}\}\,\text{Tail(E)}$, which agrees with E = a$\downarrow$ + $\uparrow$ b / a c that we found before.

Note that we could check the above conditions without knowing Tail(E). This is not always the case. Suppose that E is altered to E = a + b / a. Condition (1) is now $\{\texttt{a+}\}\Sigma^* \cap \{\texttt{a}\}\,\text{Tail(E)} = \varnothing$, which cannot be decided without knowing Tail(E). The rule has to be considered in its context; the context can be, for example S = E $, giving Tail(E) = {$}. Condition (1) becomes now $\{\texttt{a+}\}\Sigma^* \cap \{\texttt{a}\}\{\$\} = \varnothing$, identifying the cut point E = a + $\uparrow$ b / a, applicable *in this specific context*.

4 Approximation by first sets

The languages appearing in (1) and (2) may be any context-free languages. As the inclusion and emptiness of intersection of such languages are in general undecidable, there is no general mechanical way to check these conditions. We can try to approximate the involved languages by their "first sets", in the way it was suggested in [9]. A first set of e is any subset $\mathcal{F}$ of Σ^+ such that $\mathcal{L}(e) \subseteq \mathcal{F}\Sigma^*$. If $\mathcal{F}_1$ and $\mathcal{F}_2$ are first sets of e_1 and e_2, respectively, then $\mathcal{F}_1\Sigma^* \cap \mathcal{F}_2\Sigma^* = \varnothing$ guarantees $\mathcal{L}(e_1) \cap \mathcal{L}(e_2) = \varnothing$. The idea is that this condition may be easily checked.

Given $A = e_0\, e_1/e_2$, suppose there exist $\mathcal{F}_0, \mathcal{F}_2 \subseteq \Sigma^+$ such that:

$$\mathcal{L}(e_0) \subseteq \mathcal{F}_0\Sigma^*, \tag{3}$$
$$\mathcal{L}(e_2)\,\text{Tail}(A) \subseteq \mathcal{F}_2\Sigma^*, \tag{4}$$
$$\mathcal{F}_0\Sigma^* \cap \mathcal{F}_2\Sigma^* = \varnothing. \tag{5}$$

We have then $\mathcal{L}(e_0)\Sigma^* \cap \mathcal{L}(e_2)\,\text{Tail}(A) = \varnothing$ which, according to (1), gives $\uparrow$ after e_0.

As a special case, $\mathcal{F}_0$ and $\mathcal{F}_2$ may be subsets of Σ. In that case, (5) is reduced to checking disjointness of two such subsets. These are the sets used in [8] to identify the $\uparrow$ cut points. The reader may recognize in them the sets of "first letters" used to check the LL(1) property, and in (3 – 5) the exact conditions for the grammar to be LL(1).

Allowing $\mathcal{F}_0$ and $\mathcal{F}_2$ to be subsets of Σ^+ extends the approach from [8] to a wider class of grammars. In particular, to grammars that have been in [9] called LL(kP). These are the grammars where a top-down parser can choose its

way by examining the input within the reach of k parsing procedures. Note that our example in the Introduction happens to be LL(2P).

The first sets can also be used to identify the $\downarrow$ cut point. Suppose that $\mathcal{F}$ satisfies

$$\mathcal{L}(e_2)\,\mathrm{Tail}(A) \subseteq \mathcal{F}\Sigma^*, \tag{6}$$
$$\mathcal{F}\Sigma^* \subseteq \mathcal{L}(e_0)\Sigma^*. \tag{7}$$

We have then $\mathcal{L}(e_2)\,\mathrm{Tail}(A) \subseteq \mathcal{L}(e_0)\Sigma^*$. If e_0 is strong we have, according to (2), $\downarrow$ after e_0.

5 Labeled failures

Reporting syntax errors in a backtracking parser is not easy. A failure of parsing procedure need not be caused by a syntax error; it may just be a legitimate start of backtracking. A meaningful error reporting should distinguish between these two kinds of failures.

As noted in the Introduction, a failure before $\downarrow$ or after $\uparrow$ indicates a genuine syntax error. A failure between these cut points is likely to be just a switch to another alternative.

In a recent paper [5], Maidl et al. suggest a scheme that combines the semantics of cut points with generation of messages. The idea is that failing expression returns a label which may conveniently be a complete error message. Labels are created by the new expression $\Uparrow^L$ which forces an immediate failure with label L. In addition, failure to consume a letter automatically returns label "fail". The meaning of choice e_1/e_2 is redefined so that if e_1 fails with label "fail", e_2 is tried in the normal way and the expression terminates with the result of e_2. If e_1 fails with label other than "fail", the whole expression fails immediately with that label without trying e_2 [1].

The technique can be used in our example like this:

$$\mathrm{E} = (\mathtt{a} \,/\Uparrow^A) + (\mathtt{b} \,/\Uparrow^B)\ /\ \mathtt{a}\ \mathtt{c}$$

where A may be the message "a expected" and B the message "b expected".

The expression $(\mathtt{a} \,/\Uparrow^A)$ fails with label A upon the failure of a; similarly, $(\mathtt{b} \,/\Uparrow^B)$ fails with label B upon the failure of b. Thus the first alternative fails with one of these labels after a failure of a or b, causing an immediate failure of E with the appropriate message. A failure of + is ignored and makes E try its second alternative.

6 Artificial cut point

Identifying cut points requires that e_1 in e_1/e_2 is a sequence with sufficient granularity. Mizushima et al. [8] circumvent this with the help of PEG's "not-predicate". The predicate, written $!e$, means "fail if e succeeds on the text ahead, but do not consume anything".

To imitate the effect of $\uparrow$ without decomposing e_1, one can replace the rule $A = e_1/e_2$ by

$$A' = (!e_0)\, e_1 \,/\, e_2, \tag{8}$$

where e_0 is a strong expression such that:

$$\mathcal{L}(e_2)\,\mathrm{Tail}(A) \subseteq \mathcal{L}(e_0)\Sigma^*, \tag{9}$$
$$\mathcal{L}(e_1)\Sigma^* \cap \mathcal{L}(e_0)\Sigma^* = \varnothing. \tag{10}$$

We are going to check that A' has the same result as A, and that it fails if e_1 fails after successful $!e_0$ so we have in effect a $\uparrow$ cut point after $(!e_0)$:

$$A' = (!e_0) \uparrow e_1 \,/\, e_2. \tag{11}$$

[1] This is a simplified description, sufficient to explain the principle.

We need first to extend $\mathbb{G}$ with the not-predicate, which we define formally by means of two inference rules shown in Figure 2.

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

Figure 2: Semantics of not-predicate

We also need to make sure that parser can handle e_0. For this purpose, we assume that the grammar together with e_0 does not contain left-recursion[2], and use this fact:

Proposition 6.1. If $\mathbb{G}$ does not contain left-recursion then for each expression $e \in \mathbb{E}$ and string $w \in \Sigma^*$, either $[e]\ w \overset{\text{PEG}}{\rightsquigarrow} \texttt{fail}$ or $[e]\ w \overset{\text{PEG}}{\rightsquigarrow} z$ for some $z \in \Sigma^*$.

This fact has been proved in [6, 7] by showing that the formal definition of PEG used there is equivalent to that given by Ford in [3]. A direct proof from the rules of Figure 1 is found in [9] and can be easily extended to include the not-predicate. We proceed now to verify (11):

Proposition 6.2. Let $A = e_1/e_2$ be a rule of $\mathbb{G}$. Consider a proof of $[S]\ w \overset{\text{PEG}}{\rightsquigarrow} \varepsilon$ for some $w \in \Sigma^*$ that contains as partial result a proof of $[A]\ xy \overset{\text{PEG}}{\rightsquigarrow} Y$ for some $xy \in \Sigma^*$ where $Y = y$ or $Y = \texttt{fail}$. Let A' be as defined by (8 – 10). Then:

(a) $[A']\ xy \overset{\text{PEG}}{\rightsquigarrow} Y$.
(b) If $[!e_0]\ xy \overset{\text{PEG}}{\rightsquigarrow} xy$ and $[e_1]\ xy \overset{\text{PEG}}{\rightsquigarrow} \texttt{fail}$ then $Y = \texttt{fail}$.

Proof. The partial result $[A]\ xy \overset{\text{PEG}}{\rightsquigarrow} Y$ could be obtained in one of two ways:

(Case 1) From $[e_1]\ xy \overset{\text{PEG}}{\rightsquigarrow} y$ using **choice1** and **rule**. We have there $Y = y$. By Proposition 6.1 there exists a proof of either $[e_0]\ xy \overset{\text{PEG}}{\rightsquigarrow} z$ or $[e_0]\ xy \overset{\text{PEG}}{\rightsquigarrow} \texttt{fail}$. Suppose $[e_0]\ xy \overset{\text{PEG}}{\rightsquigarrow} z$. We would have then, by Proposition 2.1, $xy \in \mathcal{L}(e_0)\Sigma^*$ and $xy \in \mathcal{L}(e_1)\Sigma^*$, which contradicts (10). We must thus have $[e_0]\ xy \overset{\text{PEG}}{\rightsquigarrow} \texttt{fail}$. This gives $[A']\ xy \overset{\text{PEG}}{\rightsquigarrow} Y$ using **not2**, **seq1**, **choice1**, **rule**, and $Y = y$.

(Case 2) From $[e_1]\ xy \overset{\text{PEG}}{\rightsquigarrow} \texttt{fail}$ and $[e_2]\ xy \overset{\text{PEG}}{\rightsquigarrow} Y$ using **choice2** and **rule**. Again, by Proposition 6.1 exists a proof of either $[e_0]\ xy \overset{\text{PEG}}{\rightsquigarrow} z$ or $[e_0]\ xy \overset{\text{PEG}}{\rightsquigarrow} \texttt{fail}$. We consider these two cases separately.

(Case 2.1) $[e_0]\ xy \overset{\text{PEG}}{\rightsquigarrow} z$. We have then $[A']\ xy \overset{\text{PEG}}{\rightsquigarrow} Y$ from **not1**, **seq2**, **choice2**, and **rule**.

(Case 2.2) $[e_0]\ xy \overset{\text{PEG}}{\rightsquigarrow} \texttt{fail}$. Suppose $Y = y$. Then $[e_2]\ xy \overset{\text{PEG}}{\rightsquigarrow} y$ and $[A]\ xy \overset{\text{PEG}}{\rightsquigarrow} y$. By Proposition 2.1, $x \in \mathcal{L}(e_2)$. By Definition 3.1, $y \in \text{Tail}(A)$, so $xy \in \mathcal{L}(e_2)\,\text{Tail}(A)$. According to (9), $xy \in \mathcal{L}(e_0)\Sigma^*$. As e_0 is strong, $[e_0]\ xy \overset{\text{PEG}}{\rightsquigarrow} \texttt{fail}$ implies $xy \notin \mathcal{L}(e_0)\Sigma^*$. This is a contradiction, so $Y = \texttt{fail}$. We have $[A']\ xy \overset{\text{PEG}}{\rightsquigarrow} Y$ from **not2**, **seq2**, **choice2**, and **rule**.

In each case we have $[A']\ xy \overset{\text{PEG}}{\rightsquigarrow} Y$ which proves (a). In Case 2.2 we have $Y = \texttt{fail}$ which proves (b). □

The expression e_0 used in [8] represents the set of "first letters" of e_2, so it satisfies (9). This set is checked to be disjoint with the set of "first letters" of e_1, which satisfies (10), but also requires the grammar to be LL(1). The general conditions formulated above extend the applicability of this technique to non-LL(1) grammars.

7 Limitations and further work

The sets of "first letters" of an expression can be mechanically computed and the conditions (5) and (10) for them can be easily checked. The cut points according to (11) can always be automatically created. But this works only for LL(1) grammars, while the backtracking of PEG is often used just to avoid the LL(1) restriction.

[2] This assumption was not needed for Propositions 3.2 and 3.3 because we had there an existing parse.

It is suggested in [9] how to generate possible "first sets" using relations `First` and `Follow` to find all "first expressions". But, checking (5) and (10) for these expressions cannot always be done in a mechanical way. It appears that finding cut points in non-LL(1) grammars must to a large extent be done manually. The same applies to insertion of labeled failures, even if one solves the problem of a mechanical generation of meaningful message texts.

A practical problem is that inserting cut points and labeled failures into the grammar text makes the grammar completely unreadable. One has to find a way of conveying the information in another way. This can, for example, be done in semantic procedures, which in some parser generators (such as the author's "Mouse") are separated from the grammar.

A recent note [4] to the PEG discussion forum pointed out an important fact: cut points are local to a specific expression. Their use for discarding saved results and for generating diagnostics must be considered in the context in which the expression is invoked. Suppose expression e invoked e'. Passing a cut point in e' does not mean that e passed its cut point; e may still need its saved results. Similarly, a "serious" termination of e' may turn out to be an "innocent" one in e. This must be taken into account when designing any scheme using cut points.

Acknowledgement

The autor thanks two anonymous referees for significant improvements to the presentation.

References

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

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

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

[4] HOBBELT, G. Breaking the Mizushima(2010) cut operator. PEG Archives, July 2014, `https://lists.csail.mit.edu/pipermail/peg/2014-July/000629.html`.

[5] MAIDL, A. M., MEDEIROS, S., MASCARENHAS, F., AND IERUSALIMSCHY, R. Error reporting in Parsing Expression Grammars. Tech. rep., PUC-Rio, UFRJ Rio de Janeiro, UFRN Natal, Brazil, 2014. `http://arxiv.org/pdf/1405.6646v1.pdf`.

[6] MASCARENHAS, F., MEDEIROS, S., AND 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*. PhD thesis, Pontifícia Universidade Católica do Rio de Janeiro, Aug. 2010.

[8] MIZUSHIMA, K., MAEDA, A., AND YAMAGUCHI, Y. Packrat parsers can handle practical grammars in mostly constant space. In *Proceedings of the 9th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering, PASTE'10, Toronto, Ontario, Canada, June 5-6, 2010* (2010), S. Lerner and A. Rountev, Eds., ACM, pp. 29–36.

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