Cut points in PEG
Extended Abstract

Roman R. Redziejowski

roman.redz@swipnet.se

1 Introduction

This is a short note inspired by some ideas and results from [3, 6–8]. It is about Parsing Expression Grammars (PEGs) introduced by Ford in [1]. PEG specifies a language by defining for it a parser with limited backtracking. All of the quoted papers contain a detailed introduction to PEG, so it is not repeated here.

Backtracking means that if e_1 in a choice expression e_1/e_2 fails, the parser returns to the position it had before the attempt at e_1, and tries e_2 on the same input. Limited backtracking means that once e_1 succeeded, e_2 will not be tried on the same input upon a subsequent failure.

In some cases, a failure of e_1 may mean that e_2 is also bound to fail, so there is no need to try it; one can terminate e_1/e_2 straight away and return failure. As an example, consider this grammar:

S = E \$
E = T+E / T
T = a / b

Suppose that at some point during the parse of S, expression E is applied to input w. It starts by calling T. Clearly, if T fails, w does not start with a or b, so the second alternative, being the same T, must also fail. Suppose now that $T+$ succeeds, after which E fails. One can easily see that the only thing that can follow E in the parse of S is \$. Thus, applying the second alternative to w will result in a successful parse only if w = a\$ or w = b\$. But $T+$ succeeding on w means that w is none of these. So, trying the second alternative will not result in a successful parse.

We have identified two points in the expression for E, indicated below by $\downarrow$ and $\uparrow$, such that you do not need to backtrack if you fail before $\downarrow$ or after $\uparrow$.

E = $T\downarrow$+$\uparrow E$ / T

We shall refer to them as "cut points".

Thanks to the backtracking being limited, one can use the so-called "packrat" technology to run the PEG parser in a linear time. The technology consists in saving all results to be reused in the case of backtracking. It means buying speed at the cost of large memory consumption.

Mizushima et al. [6] noted that after passing a $\uparrow$ cut point, one can discard the saved results that would be needed by e_2. In this way, one can greatly reduce the memory requirement.

In a recent paper [3], Maidl et al. discuss the way in which a PEG parser may provide meaningful information on why it failed to parse a given input. The traditional way is to report the failure that occurred farthest down in the input. But, some failures may be quite innocent, such as a failure of e_1 in e_1/e_2 that is followed by a success of e_2. In section 5 of [3], the authors present an extension to PEG to help reporting "real" failures. Its use suggests that a "real" failure is one occurring in the "no return" zone, such as before $\downarrow$ and after $\uparrow$ in the example above.

2 Finding cut points

As follows from the above, i is interesting to find cut points in a given grammar. To see how this can be done, we consider a minimal grammar without iteration and predicates as used in [7, 8]. It has starting symbol S and end-of-input marker \$. As in [7,8], we denote by Tail(e) the set of all terminated strings that can follow an application of e in a parse starting with S. For convenience, we consider choice expressions of the form $e_0\, e_1/e_2$ that can be easily desugared to the primitive form used in [7, 8]. The input alphabet is denoted by Σ.

The grammar may be alternatively interpreted as a grammar in Backus-Naur Form (BNF), with `"/"` denoting the unordered choice. We denote by $\mathcal{L}(e)$ the language defined by e when interpreted as BNF.

The following has been shown in [5] (the proof is found also in [4, 7, 8]):

Proposition 1. *If expression e succeeds on input w, it consumes a string belonging to $\mathcal{L}(e)$, meaning that $w \in \mathcal{L}(e)\Sigma^*$.*

To say something about w in case of failing e requires that each sequence expression e_1e_2 satisfies this condition:

$$\forall_{x,y}\ \ xy \in \mathcal{L}(e_1e_2)\Sigma^*\ \wedge\ x \in \mathcal{L}(e_1)\ \Rightarrow\ y \in \mathcal{L}(e_2)\Sigma^*. \tag{1}$$

Using the formal method from [4, 5, 7, 8] one can verify the following:

Proposition 2. *In a grammar satisfying (1), if expression e fails on input w then $w \notin \mathcal{L}(e)\Sigma^*$.*

Using Proposition 1 one can verify:

Proposition 3. *A sufficient condition for $\uparrow$ after e_0 in $A = e_0\, e_1/e_2$ is:*

$$\mathcal{L}(e_0)\Sigma^* \cap \mathcal{L}(e_2)\,\mathrm{Tail}(A) = \varnothing. \tag{2}$$

Using Propositions 1 and 2 one can verify:

Proposition 4. *A sufficient condition for $\downarrow$ after e_0 in $e_0\, e_1/e_2$ in a grammar satisfying (1) is:*

$$\mathcal{L}(e_2)\,\mathrm{Tail}(A) \subseteq \mathcal{L}(e_0)\Sigma^*. \tag{3}$$

3 Using first expressions

As the inclusion and emptiness of intersection of context-free languages are in general undecidable, there is no mechanical way to to represent a given expression as $e_0\,e_1$ satisfying (2) or (3). Mizushima et al. [6] uses the set of "first terminals" as e_0. This works if the grammar is LL(1). We extend this to a wider class of grammars by using "first expressions". A first expression of e is any expression FIRST such that $\mathcal{L}(\text{FIRST}) \subseteq \Sigma^+$ and $\mathcal{L}(e) \subseteq \mathcal{L}(\text{FIRST})\Sigma^*$.

Given $A = e_1/e_2$, suppose there exist first expressions $\text{FIRST}_1, \text{FIRST}_2$ such that:

$$\mathcal{L}(e_1) \subseteq \mathcal{L}(\text{FIRST}_1)\Sigma^*, \tag{4}$$

$$\mathcal{L}(e_2)\,\text{Tail}(A) \subseteq \mathcal{L}(\text{FIRST}_2)\Sigma^*, \tag{5}$$

$$\mathcal{L}(\text{FIRST}_1)\Sigma^* \cap \mathcal{L}(\text{FIRST}_2)\Sigma^* = \varnothing. \tag{6}$$

One can easily see that $\mathcal{L}(\text{FIRST}_1)\Sigma^* \cap \mathcal{L}(e_2)\,\text{Tail}(A) = \varnothing$. If $e_1 = \text{FIRST}_1\,e_1{}'$ for some $e_1{}'$, we have, according to (2), a $\uparrow$ cut point in e_1 after FIRST_1.
For E in our example, (4–6) are satisfied by $\text{FIRST}_1 = T+$ and $\text{FIRST}_2 = T\$$, identifying the $\uparrow$ cut point after $T+$.

Suppose that instead of (6), FIRST_1 and FIRST_2 satisfy

$$\mathcal{L}(\text{FIRST}_2) \subseteq \mathcal{L}(\text{FIRST}_1)\Sigma^*. \tag{7}$$

We have then $\mathcal{L}(e_2)\,\text{Tail}(A) \subseteq \mathcal{L}(\text{FIRST}_1)\Sigma^*$. If $e_1 = \text{FIRST}_1\,e_1{}'$ for some $e_1{}'$, we have, according to (3), a $\downarrow$ cut point in e_1 after FIRST_1.
For E in our example, the conditions are satisfied by $\text{FIRST}_1 = \text{FIRST}_2 = T$, identifying the $\downarrow$ cut point after T.

A special case of FIRST_1 and FIRST_2 are the sets of "first terminals" used in [6]. In that special case, (4–6) are conditions for the grammar being LL(1). Allowing FIRST_1 and FIRST_2 to be arbitrary expressions extends the results from [6] to grammars that have been in [7,8] referred to as 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 is LL(2P).)

4 External cut point

It is often difficult to represent e_1 as $\text{FIRST}_1\,e_1{}'$. This is solved in [6] by replacing $A = e_1/e_2$ with:

$$A = \ (!\,\text{FIRST}_2)\,e_1\,/\,e_2. \tag{8}$$

Obviously e_2 must fail once $!\,\text{FIRST}_2$ succeeded, so we have $\uparrow$ after $(!\,\text{FIRST}_2)$. To verify that this also works for first expressions, we have to introduce the not-predicate `"!"` into our grammar. It can be done in a restricted way, by defining $!\,e_0\,e_1/e_2$ as a new expression and formally specifying its semantics.

Assuming that the grammar satisfies (1) and $\textsc{first}_1, \textsc{first}_2$ satisfy (4–6), one can use Propositions 1 and 2 to verify that:

- $(!\,\textsc{first}_2)\, e_1 \,/\, e_2$ is equivalent to $e_1 \,/\, e_2$ in the sense that both either consume the same text, or both fail, on the same input.
- If $!\,\textsc{first}_2$ succeeds then e_2 fails.

5 Labeled failures

In the standard version of PEG, a failing expression returns just an indication that it failed. In the modification suggested in Section 5 of [3], failing expression returns a label which may conveniently be a complete error message. One distinguished such label is just "fail". A failing terminal returns "fail" by default. Other labels are created by the new expression $\Uparrow^l$ which forces an immediate failure with label l. 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. Of course, this can only be safe before $\downarrow$ and after $\uparrow$.

The technique can be used in our example like this:

$$E = (T \,/\, \Uparrow^t) + (E \,/\, \Uparrow^e) \;/\; T$$

where t may be the message "Term expected" and e the message "Expression expected". The result is the message "Term expected" for a failure before $\downarrow$ and "Expression expected" for a failure after $\uparrow$.

6 Problems

The sets of "first terminals" of e can be mechanically computed by restricting to terminals the set $\texttt{First}^*(e)$ where `First` is the relation describing which expressions appear as first in the definition of a given expression. Condition (6) boils down to checking that two sets of letters are disjoint. This means that the cut points (8) can always be automatically inserted. But it works only for LL(1) grammars, while the backtracking of PEG is often used just to avoid the LL(1) restriction.

It is suggested in [7,8] how to find $\textsc{first}_1, \textsc{first}_2$ satisfying (4,5) among the subsets of $\texttt{First}^*(e_1)$ and $\texttt{First}^*(e_2)$. But, checking (6) for these subsets cannot, in general, 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.

An anonymous reviewer pointed out that manually inserted cut points and labeled failures make 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 [2] 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.

References

1. 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)
2. Hobbelt, G.: Breaking the Mizushima(2010) cut operator, PEG Archives, July 2014, `https://lists.csail.mit.edu/pipermail/peg/2014-July/000629.html`
3. Maidl, A.M., Medeiros, S., Mascarenhas, F., 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`
4. Mascarenhas, F., Medeiros, S., Ierusalimschy, R.: On the relation between context-free grammars and Parsing Expression Grammars. Tech. rep., UFRJ Rio de Janeiro, UFS Aracaju, PUC-Rio, Brazil (2013), `http://arxiv.org/pdf/1304.3177v1`
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. Mizushima, K., Maeda, A., Yamaguchi, Y.: Packrat parsers can handle practical grammars in mostly constant space. In: Lerner, S., Rountev, A. (eds.) 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. pp. 29–36. ACM (2010)
7. Redziejowski, R.R.: From EBNF to PEG. Fundamenta Informaticae 128(1-2), 177–191 (2013)
8. Redziejowski, R.R.: More about converting BNF to PEG. Fundamenta Informaticae (2014), to appear, `http://www.romanredz.se/papers/FI2014.pdf`