Cut points in PEG
Extended Abstract

Roman R. Redziejowski

roman.redz@swipnet.se

1 Introduction

This is a short note that combines some ideas and results from papers [2–5]. 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 the quoted papers contain a detailed introduction to PEG, so it is not repeated here.

Thanks to the backtracking being limited, one can use the so-called "packrat" technology to run the PEG parser in a linear time, at the cost of large memory consumption. Mizushima et al. [3] noted that by introducing a "cut point" one can greatly reduce the memory requirement. The point, indicated by $\uparrow$, is inserted in a choice expression $e_1 e_2 / e$ like this: $e_1 \uparrow e_2 \,/\, e$. It means that once e_1 succeeded on some input, e cannot possibly succeed on the same input, so there is no need to backtrack and try e if e_2 fails. Thus, once $\uparrow$ is passed, there is no need to store the information needed for backtracking.

The method of inserting $\uparrow$ described in [3] applies to grammars with LL(1) property. These are the languages where a top-down parser can choose the correct action by looking one character ahead. In [4,5], this author considered grammars where a backtracking parser can make its choice by examining the input within the reach of k parsing procedures. This property was called LL(kP). We are going to discuss the insertion of cut markers in such grammars.

In Section 5 of a recent paper [2], Maidl et al. introduced an elegant extension to PEG for reporting syntax failures. One of its main features is to optionally prevent backtracking after a failure - exactly the function of a cut point. We are going to give an example of its usage.

2 Cut points in PEG

Consider this grammar:

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

To choose the correct alternative for E, the parser must call two procedures: that for T and that for +. So the grammar is LL(2P).

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 here two cut points, $\downarrow$ and $\uparrow$, meaning "do not backtrack if you fail before $\downarrow$ or after $\uparrow$":

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

In order to find out where to insert the cut points, we consider a grammar as used in [4, 5], with the starting symbol S and end-of-input marker \$. As in these papers, we denote by $\mathcal{L}(e)$ the language defined by expression e when interpreted as BNF, and by $\mathrm{Tail}(e)$ the set of all terminated strings that can follow an application of e in a parse starting with S. We assume further that each choice expression $A = e_1/e_2$ satisfies the condition

$$\mathcal{L}(e_1)\Sigma^* \cap \mathcal{L}(e_2)\,\mathrm{Tail}(A) = \varnothing. \tag{1}$$

As shown in [4], every parsing expression e accepts then the language $\mathcal{L}(e)$. One can show that if e fails on input w, we have $w \notin \mathcal{L}(e)\Sigma^*$. For convenience, we consider choice expressions of the form e_1e_2/e that can be easily desugared to the primitive form used in [4, 5].

Proposition 1. *The sufficient condition for $\downarrow$ after e_1 in $A = e_1e_2/e$ is:*

$$\mathcal{L}(e_1)\Sigma^* \supseteq \mathcal{L}(e)\,\mathrm{Tail}(A). \tag{2}$$

Proof. Suppose the parsing expression e_1 fails on input w. We have then $w \notin \mathcal{L}(e_1)\Sigma^*$. Suppose that e succeeds when applied to w. We would have then $w \in \mathcal{L}(e)\,\mathrm{Tail}(A)$, which contradicts (2). □

Proposition 2. *The sufficient condition for $\uparrow$ after e_1 in $A = e_1e_2/e$ is:*

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

Proof. Suppose e_1 succeeds on input w and then e_2 fails on what was left by e_1. From e_1 succeeding follows $w \in \mathcal{L}(e_1)\Sigma^*$. Suppose that e succeeds when applied to w. We would have then $w \in \mathcal{L}(e)\,\mathrm{Tail}(A)$, which contradicts (3). □

(The above proofs can be fully formalized with the "natural semantics" definitions of BNF and PEG given in [4, 5].)

3 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 [2], 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 other than "fail", the whole expression fails immediately with that label without trying e_2. 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. (This is a simplified description; in the full version, the choice may "catch" other labels than "fail".)

The cut points from our example may be encoded like this:

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

where t may be the message "Term expected" and e the message "Expression expected". If the input does not start with a or b, the first T ends with "fail" and $(T \,/\Uparrow^t)$ proceeds to its second alternative, which forces its termination with label "Term expected". The sequence $(T \,/\Uparrow^t) + (E \,/\Uparrow^e)$ terminates with this label, so the alternative T is not attempted and the whole expression ends with "Term expected". If the first T succeeds and + fails, the sequence $(T \,/\Uparrow^t) + (E \,/\Uparrow^e)$ terminates with label "fail" and the alternative T is tried. If the first T and + succeed, and the following E fails, the sequence $(T \,/\Uparrow^t) + (E \,/\Uparrow^e)$ terminates with "Expression expected" so the alternative T is not tried.

4 Problems

The above example shows that, in addition to saving memory and unnecessary processing, the proper placement of cut points can result in meaningful diagnostics. However, the languages appearing in (2) and (3) are, in general, context-free languages. The inclusion and emptiness of intersection of such languages are, in general, undecidable, meaning there is no general algorithm to check (2) and (3). One may reduce the ambition to special cases or use approximations.

The solution suggested in [3] approximates the languages appearing in (3) by their first letters (more precisely, first terminals): the condition is satisfied if words in $\mathcal{L}(e_1)$ do not start with the same letters as words in $\mathcal{L}(e)\,\mathrm{Tail}(A)$. This can be checked in a fully mechanical way, but is restricted to LL(1) grammars. One can use instead approximations by initial nonterminals as suggested in [4,5], but here one can again run into undecidable situations.

We note that as the result of checking (1) by approximations in [4, 5], one finds e_1 satisfying (3), which thus produces a cut point $\uparrow$ as a by-product. It seems possible to use similar approximation method for identifying the $\downarrow$ cut point.

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. 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
3. 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)
4. Redziejowski, R.R.: From EBNF to PEG. Fundamenta Informaticae 128(1-2), 177–191 (2013)
5. Redziejowski, R.R.: More about converting BNF to PEG. Fundamenta Informaticae (2014), to appear, http://www.romanredz.se/papers/FI2014.pdf