Science of Computer Programming 132 (2016) 129–140

Contents lists available at ScienceDirect

Science of Computer Programming

www.elsevier.com/locate/scico

Error reporting in Parsing Expression Grammars

André Murbach Maidl [a], Fabio Mascarenhas [b,*], Sérgio Medeiros [c], Roberto Ierusalimschy [d]

[a] Polytechnic School, PUCPR, Curitiba, Brazil
[b] Department of Computer Science, UFRJ, Rio de Janeiro, Brazil
[c] School of Science and Technology, UFRN, Natal, Brazil
[d] Department of Computer Science, PUC-Rio, Rio de Janeiro, Brazil

ARTICLE INFO

Article history:
Received 11 April 2014
Received in revised form 3 August 2016
Accepted 12 August 2016
Available online 20 August 2016

Keywords:
Parsing
Error reporting
Parsing expression grammars
Packrat parsing
Parser combinators

ABSTRACT

Parsing Expression Grammars (PEGs) describe top-down parsers. Unfortunately, the error-reporting techniques used in conventional top-down parsers do not directly apply to parsers based on Parsing Expression Grammars (PEGs), so they have to be somehow simulated. While the PEG formalism has no account of semantic actions, actual PEG implementations add them, and we show how to simulate an error-reporting heuristic through these semantic actions.

We also propose a complementary error reporting strategy that may lead to better error messages: labeled failures. This approach is inspired by exception handling of programming languages, and lets a PEG define different kinds of failure, with each ordered choice operator specifying which kinds it catches. Labeled failures give a way to annotate grammars for better error reporting, to express some of the error reporting strategies used by deterministic parser combinators, and to encode predictive top-down parsing in a PEG.

1. Introduction

When a parser receives an erroneous input, it should indicate the existence of syntax errors. Errors can be handled in various ways. The easiest is just to report that an error was found, where it was found, and what was expected at that point and then abort. At the other end of the spectrum we find mechanisms that attempt to parse the complete input, and report as many errors as best as possible.

The LL(k) and LR(k) methods detect syntax errors very efficiently because they have the *viable prefix* property, that is, these methods detect a syntax error as soon as k tokens are read and cannot be used to extend the thus far accepted part of the input into a viable prefix of the language [1]. LL(k) and LR(k) parsers can use this property to produce suitable, though generic, error messages.

Parsing Expression Grammars (PEGs) [2] are a formalism for describing the syntax of programming languages. We can view a PEG as a formal description of a top-down parser for the language it describes. PEGs have a concrete syntax based on the syntax of *regexes*, or extended regular expressions. Unlike Context-Free Grammars (CFGs), PEGs avoid ambiguities in the definition of the grammar's language due to the use of an *ordered choice* operator.

* Corresponding author.
E-mail addresses: andre.murbach@pucpr.br (A.M. Maidl), mascarenhas@ufrj.br (F. Mascarenhas), sergiomedeiros@ect.ufrn.br (S. Medeiros), roberto@inf.puc-rio.br (R. Ierusalimschy).

http://dx.doi.org/10.1016/j.scico.2016.08.004
0167-6423/

More specifically, a PEG can be interpreted as a the specification of a recursive descent parser with restricted (or local) backtracking. This means that the alternatives of a choice are tried in order; as soon as an alternative recognizes an input prefix, no other alternative of this choice will be tried, but when an alternative fails to recognize an input prefix, the parser backtracks to try the next alternative.

On the one hand, PEGs can be interpreted as a formalization of a specific class of top-down parsers [2]; on the other hand, PEGs cannot use error handling techniques that are often applied to predictive top-down parsers, because these techniques assume the parser reads the input without backtracking [3]. In top-down parsers without backtracking, it is possible to signal a syntax error as soon as the next input symbol cannot be accepted. In PEGs, it is more complicated to identify the cause of an error and the position where it occurs, because failures during parsing are not necessarily errors, but just an indication that the parser cannot proceed and a different choice should be made *elsewhere*.

Ford [3] has already identified this limitation of error reporting in PEGs, and, in his parser generators for PEGs, included a heuristic for better error reporting. This heuristic simulates the error reporting technique that is implemented in top-down parsers without backtracking. The idea is to track the position in the input where the farthest failure occurred, as well as what the parser was expecting at that point, and report this to the user in case of errors.

Tracking the farthest failure position and context gives us PEGs that produce error messages similar to the automatically produced error messages of other top-down parsers; they tell the user the position where the error was encountered, what was found in the input at that position, and what the parser was expecting to find.

In this paper, we show how grammar writers can use this error reporting technique even in PEG implementations that do not implement it, by making use of semantic actions that expose the current position in the input and the possibility to access some form of mutable state associated with the parsing process.

We also propose a complementary approach for error reporting in PEGs, based on the concept of *labeled failures*, inspired by the standard exception handling mechanisms as found in programming languages. Instead of just failing, a labeled PEG can produce different kinds of failure labels using a *throw* operator. Each label can be tied to a more specific error message. PEGs can also *catch* such labeled failures, via a change to the ordered choice operator. We formalize labeled failures as an extension of the semantics of regular PEGs.

With labeled PEGs we can express some alternative error reporting techniques for top-down parsers with local backtracking. We can also encode predictive parsing in a PEG, and we show how to do that for $LL(*)$ parsing, a powerful predictive parsing strategy.

The rest of this paper is organized as follows: Section 2 contextualizes the problem of error handling in PEGs, explains in detail the failure tracking heuristic, and shows how it can be realized in PEG implementations that do not support it directly; Section 3 discusses related work on error reporting for top-down parsers with backtracking; Section 4 introduces and formalizes the concept of labeled failures, and shows how to use it for error reporting; Section 5 compares the error messages generated by a parser based on the failure tracking heuristic with the ones generated by a parser based on labeled failures; Section 6 shows how labeled failures can encode some of the techniques of Section 3, as well as predictive parsing; finally, Section 7 gives some concluding remarks.

2. Handling syntax errors with PEGs

In this section, we use examples to present in more detail how a PEG behaves badly in the presence of syntax errors. After that, we present a heuristic proposed by Ford [3] to implement error reporting in PEGs. Rather than using the original notation and semantics of PEGs given by Ford [2], our examples use the equivalent and more concise notation and semantics proposed by Medeiros et al. [4–6]. We will extend both the notation and the semantics in Section 4 to present PEGs with labeled failures.

A PEG G is a tuple (V, T, P, p_S) where V is a finite set of non-terminals, T is a finite set of terminals, P is a total function from non-terminals to *parsing expressions* and p_S is the initial parsing expression. We describe the function P as a set of rules of the form $A \leftarrow p$, where $A \in V$ and p is a parsing expression. A parsing expression, when applied to an input string, either fails or consumes a prefix of the input and returns the remaining suffix. The abstract syntax of parsing expressions is given as follows, where a is a terminal, A is a non-terminal, and p, p_1 and p_2 are parsing expressions:

$$p = \varepsilon \mid a \mid A \mid p_1 p_2 \mid p_1 / p_2 \mid p* \mid !p$$

Intuitively, ε successfully matches the empty string, not changing the input; a matches and consumes itself or fails otherwise; A tries to match the expression $P(A)$; $p_1 p_2$ tries to match p_1 followed by p_2; p_1 / p_2 tries to match p_1; if p_1 fails, then it tries to match p_2; $p*$ repeatedly matches p until p fails, that is, it consumes as much as it can from the input; the matching of $!p$ succeeds if the input does not match p and fails when the input matches p, not consuming any input in either case; we call it the negative predicate or the lookahead predicate.

Fig. 1 presents a PEG for the Tiny language [7]. Tiny is a simple programming language with a syntax that resembles Pascal's. We will use this PEG, which can be seen as the equivalent of an LL(1) CFG, to show how error reporting differs between top-down parsers without backtracking and PEGs.