Science of Computer Programming 89 (2014) 235–250

Contents lists available at ScienceDirect

Science of Computer Programming

www.elsevier.com/locate/scico

On the relation between context-free grammars and parsing expression grammars

Fabio Mascarenhas[a,*], Sérgio Medeiros[b,*], Roberto Ierusalimschy[c,*]

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

HIGHLIGHTS

- We present new formalizations of CFGs and PEGs, based on natural semantics.
- We discuss the correspondence between the languages of CFGs and PEGs.
- We show transformations from classes of CFGs to equivalent PEGs.
- Transformations of LL(1), strong-LL(k) and LL-regular grammars are presented.
- The transformations preserve the structure of the original grammars.

ARTICLE INFO

Article history:
Received 29 July 2013
Received in revised form 13 January 2014
Accepted 20 January 2014
Available online 30 January 2014

Keywords:
Context-free grammars
Parsing expression grammars
LL(1)
LL(k)
Natural semantics

ABSTRACT

Context-Free Grammars (CFGs) and Parsing Expression Grammars (PEGs) have several similarities and a few differences in both their syntax and semantics, but they are usually presented through formalisms that hinder a proper comparison. In this paper we present a new formalism for CFGs that highlights the similarities and differences between them. The new formalism borrows from PEGs the use of *parsing expressions* and the recognition-based semantics. We show how one way of removing non-determinism from this formalism yields a formalism with the semantics of PEGs. We also prove, based on these new formalisms, how LL(1) grammars define the same language whether interpreted as CFGs or as PEGs, and also show how strong-LL(k), right-linear, and LL-regular grammars have simple language-preserving translations from CFGs to PEGs. Once these classes of CFGs can be automatically translated to equivalent PEGs, we can reuse classic top-down grammars in PEG-based tools.

1. Introduction

Context-Free Grammars (CFGs) are the formalism of choice for describing the syntax of programming languages. A CFG describes a language as the set of strings generated from the grammar's initial symbol by a sequence of rewriting steps. CFGs do not, however, specify a method for efficiently recognizing whether an arbitrary string belongs to its language. In other words, a CFG does not specify how to *parse* the language, an essential operation for working with the language (in a compiler, for example). Another problem with CFGs is ambiguity, where a string can have more than one parse tree.

Parsing Expression Grammars (PEGs) [1] are an alternative formalism for describing a language's syntax. Unlike CFGs, PEGs are unambiguous by construction, and their standard semantics is based on recognizing strings instead of generating

* Corresponding authors.
E-mail addresses: fabiom@dcc.ufrj.br (F. Mascarenhas), sergiomedeiros@ect.ufrn.br (S. Medeiros), roberto@inf.puc-rio.br (R. Ierusalimschy).

0167-6423/$ – see front matter © 2014 Elsevier B.V. All rights reserved.
http://dx.doi.org/10.1016/j.scico.2014.01.012

them. A PEG can be considered both the specification of a language and the specification of a top-down parser for that language.

The idea of using a formalism for specifying parsers is not new; PEGs are based on two formalisms first proposed in the early seventies, Top-Down Parsing Language (TDPL) [2] and Generalized TDPL (GTDPL) [3]. PEGs have in common with TDPL and GTDPL the notion of *limited backtracking* top-down parsing: the parser, when faced with several alternatives, will try them in a deterministic order (left to right), discarding remaining alternatives after one of them succeeds. Compared with the older formalisms, PEGs introduce a more expressive syntax, based on the syntax of regexes, and add *syntactic predicates* [4], a form of unrestricted lookahead where the parser checks whether the rest of the input matches a parsing expression without consuming the input.

Ford [1] has already proven that PEGs can recognize any deterministic context-free language, but leaves open the question of the relation between context-free grammars and PEGs. In this paper, we argue that the similarities between CFGs and PEGs are deeper than usually thought, and how these similarities have been obscured by the way the two formalisms have been presented. PEGs, instead of a formalism completely unrelated to CFGs, can be seen as a natural outcome of removing the ambiguity of CFGs.

We start with a new semantics for CFGs, using the framework of natural semantics [5,6]. The new semantics borrows the syntax of PEGs and is also based on recognizing strings. We make the source of the ambiguity of CFGs, their non-deterministic alternatives for each non-terminal, explicit in the semantics of our new non-deterministic choice operator. We then remove the non-determinism, and consequently the ambiguity, by adding explicit failure and ordered choice to the semantics, so now we can only use the second alternative in a choice if the first one fails. By that point, we only need to add the *not* syntactic predicate to arrive at an alternative semantics for PEGs (modulo syntactic sugar such as the repetition operator and the *and* syntactic predicate). We prove that our new semantics for both CFGs and PEGs are equivalent to the usual ones.

Our semantics for CFGs and PEGs make it clear that the defining characteristic that sets PEGs apart from CFGs is the ordered choice. For example, the grammar $S \to (aba \mid a)b$ is both a CFG and a PEG in our notation, but consumes the prefix ab out of the subject $abac$ when interpreted as a CFG, and fails as a PEG.

We also show in this paper how our new semantics for CFGs gives us a way to translate some subsets of CFGs to PEGs that parse the same language. The idea is that, as the sole distinction between CFG and PEG semantics is in the choice operator, we will have a PEG that is equivalent to the CFG whenever we can make the PEG choose the correct alternative at each choice, either through reordering or with the help of syntactic predicates. We show transformations from CFGs to PEGs for three unambiguous subsets of CFGs: LL(1), strong-LL(k), and LL-regular.

A straightforward correspondence between LL(1) grammars and PEGs was already noted [7], but never formally proven. The correspondence is that an LL(1) grammar describes the same language whether interpreted as a CFG or as a PEG. The intuition is that, if an LL(1) parser is able to choose an alternative with a single symbol of lookahead, then a PEG parser will fail for every alternative that is not the correct one. We prove that this intuition is correct if none of the alternatives in the CFG can generate the empty string, and also prove that a simple ordering of the alternatives suffices to hold the correspondence even if there are alternatives that can generate the empty string. In other words, any LL(1) grammar is already a PEG that parses the same language, modulo a reordering of the alternatives.

There is no such correspondence between strong-LL(k) grammars and PEGs, not even by imposing a specific order among the alternatives for each non-terminal. Nevertheless, we also prove that we can transform a strong-LL(k) grammar to a PEG, just by adding a predicate to each alternative of a non-terminal. We can either add a predicate to the beginning of each alternative, thus encoding the choice made by a strong-LL(k) parser in the grammar, or, more interestingly, add a predicate to the end of each alternative.

Our transformations lead to efficient parsers for LL(1) and strong-LL(k) grammars, even in PEG implementations that do not use memoization to guarantee $O(n)$ performance, because the resulting PEGs only use backtracking to test the lookahead of each alternative, so their use of backtracking is equivalent to a top-down parser checking the next k symbols of lookahead against the lookahead values of each production.

There is no direct correspondence between LL-regular grammars and PEGs, either, given that strong-LL(k) grammars are a proper subset of LL-regular grammars. But we also show that we can transform any LL-regular grammar into a PEG that recognizes the same language: we first prove that right-linear grammars for languages with the prefix property, a property that is easy to achieve, have the same language whether interpreted as CFGs or as PEGs, then use this result to build lookahead expressions for the alternatives of each non-terminal based on which regular partition this alternative falls.

While LL(1) grammars are a proper subset of strong-LL(k) grammars, which are a proper subset of LL-regular grammars, thus making the LL-regular transformation work on grammars belonging to these simpler classes, the simpler classes have more straightforward transformations which merit a separate treatment.

Given that these classes of top-down CFGs can be automatically translated into equivalent PEGs, we can reuse classic top-down grammars in PEG-based tools. So grammars written for tools such as ANTLR [4] could be reused in a parser tool that has PEGs as its backend language. As PEGs are composable, these grammars can then be used as components in larger grammars. The language designer can then start with a simple, LL(1) or strong-LL(k) subset of the language, and then grow it into the full language.

The rest of this paper is organized as follows: Section 2 presents our new semantics for CFGs and PEGs, showing how to arrive at the latter from the former, and proves their correctness. Section 3 shows how an LL(1) grammar describes the