Non-LR(1) Precedence Cascade Grammars

José-Luis Sierra
Fac. Informática. Universidad Complutense de Madrid
C/ Prof. José García Santesmases 9. 28040 Madrid, Spain
jlsierra@ucm.es
https://orcid.org/0000-0002-0317-0510

Abstract

Precedence cascade is a well-known pattern for writing context-free grammars (CFGs) that model the syntax of expression languages. According to this method, precedence levels are represented by non-terminals, and operators' attributes are used to write syntax rules properly. In most cases, the resulting *precedence cascade grammar* (PCG) has neat properties that facilitate its implementation. In particular, many PCGs are LR(1) grammars, which serve as input for conventional bottom-up parser generators. However, for some cumbersome operator tables the method does not produce such neat grammars. This paper focuses on these cumbersome operator tables by identifying several conditions leading to non-LR(1) PCGs.

2012 ACM Subject Classification Software and its engineering → Syntax

Keywords and phrases grammarware, expression grammars, grammar patterns, grammar ambiguity, LR grammars

Digital Object Identifier 10.4230/OASIcs.SLATE.2018.11

Category Short Paper

Funding This work is supported by the project grants TIN2014-52010-R and TIN2017-88092 R.

1 Introduction

Most computer languages include an *expression sub-language* as their most distinctive feature. This sub-language allows users to begin with a repertoire of primitive expressions and create more complex expressions by combining simpler ones. Such a combination is carried out by operators [13].

In this paper we will focus only in the most common classes of operators: binary infix, and unary prefix and postfix operators. In addition, we will adopt the conventions of the Prolog language to describe the attributes for these operators [5]:

- Each operator will have a *name* (e.g., $+$, $-$, $*$ ...). It will be possible to *overload* this name, allowing different operator definitions to share such a name.
- Each operator will belong to a *precedence level.* Each precedence level will be represented by a positive natural number. Operators in lower precedence levels will take *priority* over (i.e., will bind tighter than) operators in higher ones[1]. In addition, when an operator is used to build an expression, this expression will take the precedence level for that operator. Precedence levels for basic expressions will be 0.

[1] That is, following Prolog conventions, in this paper *precedence* and *priority* of operators will be *contravariant* properties.

Name	Precedence	Type
$\otimes$	3	fy
$\oplus$	3	xfy
$\boxplus$	2	yfx
$\otimes$	2	xfx
$\otimes$	1	yf

(a) Operator table for a sample expression language.

$$\begin{array}{lcl} E_3 & \rightarrow & \otimes E_3 \mid E_2 \oplus E_3 \mid E_2 \\ E_2 & \rightarrow & E_2 \boxplus E_1 \mid E_1 \otimes E_1 \mid E_1 \\ E_1 & \rightarrow & E_1 \otimes \mid E_0 \\ E_0 & \rightarrow & a \mid (E_3) \end{array}$$

(b) PCG for the descriptions in Table 1a; it is an LR(1) grammar.

Figure 1 An operator table and its associated PCG.

- Operators will constrain the precedence levels of their arguments to be: (i) lower than their own precedence level (denoted by x in the description of the operator's argument), or (ii) lower or equal than such a precedence level (which will be denoted by y).

The fixity and the arguments' allowed precedences will together form the operator's *syntactic type*. Following Prolog convention, this type will be one of the following forms: (i) for infix operators, yfx, xfy, xfx; (ii) for prefix operators, fy, fx; and (iii) for postfix operators, yf, xf. This way, yfx operators are left-associative, xfy right-associative, and xfx non-associative. In turn, fy and yf are associative, while xf and fx are non-associative unary (prefix and postfix) operators. All this information can be condensed into an *operator table* for the language. Table 1a gives an example of an operator table[2].

To model the syntax of this kind of expression languages, it is possible to use a *precedence cascade* pattern, which is described to a greater or lesser extent in any typical textbook on compiler construction (e.g., [3, 8]). In order to describe the pattern, we will introduce the following notation:

- By $\downarrow(i)$ we will denote the precedence level immediately smaller than i, or 0 if i is the smallest precedence level.
- By $\top$ we will denote the greatest precedence level.

The pattern itself is based on the following conventions (Figure 1b shows the CFG that results from applying these conventions to the Table 1a):

- Each precedence level i has a non-terminal E_i associated with it that represents expressions built with operators at that level.
- Each operator $\odot$ in level i has a rule associated with it that characterizes the syntax of the expressions formed with that operator. This rule depends on the operator's type: (i) $E_i \rightarrow E_i \odot E_{\downarrow(i)}$ if the type is yfx; (ii) $E_i \rightarrow E_{\downarrow(i)} \odot E_i$ if it is xfy; (iii) $E_i \rightarrow E_{\downarrow(i)} \odot E_{\downarrow(i)}$ if xfx; (iv) $E_i \rightarrow \odot E_i$ if fy; (v) $E_i \rightarrow \odot E_{\downarrow(i)}$ if fx; (vi) $E_i \rightarrow E_i \odot$ if yf; and (vii) $E_i \rightarrow E_{\downarrow(i)} \odot$ if the type is xf.
- There is an additional rule $E_i \rightarrow E_{\downarrow(i)}$ for each level i.
- Finally, there is a non-terminal symbol E_0 that models the basic (i.e., literals, variables, function calls, etc.) and parenthesized expressions. In the sequel we will abstract all the basic expressions with a single a symbol. Thus, there will be an additional pair of rules $E_0 \rightarrow a \mid (E_\top)$

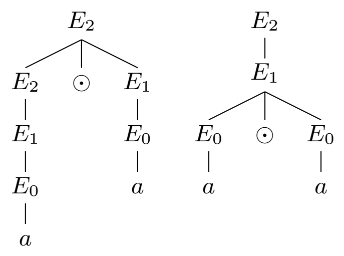
[2] Notice that, according to this operator table, an expression like "$\otimes a \oplus a \oplus a \otimes a \otimes$" will mean "$\otimes(a \oplus (a \oplus (a \otimes (a \otimes))))$", while another one like "$a \oplus \otimes a$" will be ill-formed (it should be written "$a \oplus (\otimes a)$").

Name	Prec.	Type
$\odot$	2	yfx
$\odot$	1	xfx

(a) Operator table with multiple definitions of the infix operator $\odot$.

$$\begin{array}{lcl} E_2 & \rightarrow & E_2 \odot E_1 \mid E_1 \\ E_1 & \rightarrow & E_0 \odot E_0 \mid E_0 \\ E_0 & \rightarrow & a \mid (E_2) \end{array}$$

(b) PCG resulting of the operator table presented in Table 2a.



(c) Two different parse trees for "$a \odot a$".

Figure 2 Example regarding multiple operator definitions with the same name and fixity.

We will refer to the CFGs produced by this pattern as *precedence cascade grammars* (PCGs). A well-known example of using this pattern for a real programming language is Jeff Lee's YACC grammar for ANSI C[3].

For most operator tables, the PCGs are LR(1) grammars [6] suitable for typical bottom-up, YACC-like, parser generators (this is the case, for instance, of the PCG in Figure 1b)[4]. However, there are also operator tables that lead to non-LR(1) grammars. Most of the time, this is due to contradictory operator definitions, which in turn produce ambiguous PCGs. Other times, such contradictions do not exist, but even so the resulting PCGs require more than one look-ahead symbol. In this paper we address these concerns by identifying common situations leading to non-LR(1) PCGs.

The rest of the paper is structured as follows. Section 2 describes the problems caused by multiple operator definitions with the same name and fixity. Section 3 addresses the problems caused by operators with opposite associativities at the same precedence level. Section 4 analyzes the concerns caused by the overloading of an operator in infix and postfix forms. Section 5 analyzes potential ambiguities caused by operators overloaded simultaneously in infix, prefix and postfix forms. Section 6 summarizes some work related to ours. Finally, Section 7 presents some conclusions and lines of future work.

2 Multiple operator definitions with the same name and fixity

Operator tables containing multiple operator definitions with the same name and fixity, but with different types and/or different precedence levels are intrinsically ambiguous, since any occurrence of the multiple-defined operator names can be explained indistinctly for either one or another definition. Therefore, the resulting PCG will be ambiguous.

Figure 2 illustrates the problems caused by this kind of tables. Notice that, since there are two definitions of the infix operator $\odot$, it is not possible to discern which version of $\odot$ is used. In consequence, the resulting PCG (Figure 2b) is ambiguous (and, therefore, non-LR(1)), as illustrated by the two different parse trees for the witness expression "$a \odot a$" in Figure 2c .

Finally, notice that the conditions reported in this section only affect multiple operator definitions with the same name and fixity. On the other hand, it is perfectly feasible to have multiple definitions with the same name, but with different fixities, and still obtain LR(1) PCGs (e.g., infix, prefix and postfix $\otimes$ in Figure 1).

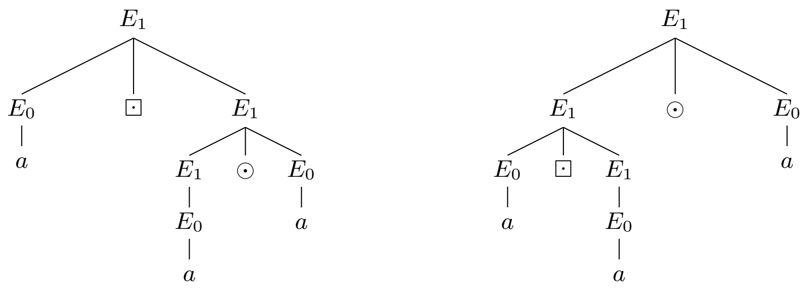
[3] `https://www.lysator.liu.se/c/ANSI-C-grammar-y.html`
[4] These and other similar assertions on the LR(1) condition of particular CFGs can be verified, for instance, with the tools available online at `http://smlweb.cpsc.ucalgary.ca/`.

Name	Prec.	Type
$\odot$	1	yfx
$\boxdot$	1	xfy

(a) Operators precedence table.

$$\begin{aligned} E_1 &\rightarrow E_1 \odot E_0 \mid E_0 \boxdot E_1 \mid E_0 \\ E_0 &\rightarrow a \mid (E_1) \end{aligned}$$

(b) PCG for the operators described in Table 3a.



(c) Two different parse trees for "$a \boxdot a \odot a$".

Figure 3 Example regarding two operators with opposite associativies at the same precedence level.

3 Opposite associativities at the same precedence level

Another cause of non-LR(1) PCGs is the confluence, in the same precedence level, of two operators with *opposite* associativites, i.e., (i) one operator of type xfy with another one of type yfx or yf; (ii) a yfx operator with one of type fy; or (iii) a fy operator with a yf one. This confluence leads to ambiguity.

This situation is illustrated, for instance, by the operator Table 3a, which includes at the same precedence level a $\odot$ operator of type yfx and another one $\boxdot$ of type xfy. The resulting PCG is shown in Figure 3b. Thus, an expression like "$a \boxdot a \odot a$" will have two possible interpretations, depending on which of the two operators is applied first: "$(a \boxdot a) \odot a$" if $\boxdot$ is applied first, or "$a \boxdot (a \odot a)$" if it is $\odot$ that is applied first. As a result, the PCG in Figure 3b is ambiguous, as is proven in Figure 3c, which gives two different parse trees for "$a \boxdot a \odot a$". The other aforementioned unsuitable combinations due to opposite associatives can be illustrated in similar terms.

Finally notice that the existence of operators with different associativies at the same level only proves cumbersome for the aforementioned combinations. In this way, it is possible to find associative and non-associative operators at the same precedence level (e.g., $\boxplus$ and infix $\otimes$ in Figure 1), as well as several operators with the same associativity direction (e.g., $\oplus$ and prefix $\otimes$ in Figure 1), and still obtain LR(1) PCGs.

4 Overloading an operator with infix and postfix fixities

Definitions of an operator $\odot$ as an infix and a postfix one leads, in most of the situations, to non-LR(1) PCGs. Table 1 summarizes the different combinations and whether the resulting PCGs are LR(1) or not. These facts can be readily verified by providing the corresponding definitions, and generating and checking the resulting grammars[5].

[5] In particular, to check the LR(2) condition we used SLK (`http://www.slkpg.com/`), a parser generator that supports arbitrary look-ahead to resolve LR conflicts, and JikesPG (`http://jikes.sourceforge.net/`), another parser generator supporting arbitrary LALR(k) grammars.

Table 1 Classes of PCGs for tables overloading a $\odot$ operator in infix (precedence level l_i) and postfix (precedence level l_p) forms (**LR(1)**: the resulting grammar is LR(1); **LR(2)**: the resulting grammar is LR(2), but not LR(1); **AMB**: the resulting grammar is ambiguous).

	$\tau_i = yfx$ $\tau_p = yf$	$\tau_i = yfx$ $\tau_p = xf$	$\tau_i = xfy$ $\tau_p = yf$	$\tau_i = xfy$ $\tau_p = xf$	$\tau_i = xfx$ $\tau_p = yf$	$\tau_i = xfx$ $\tau_p = xf$
$l_p > l_i$	**LR(2)**	**LR(1)**	**LR(2)**	**LR(2)**	**LR(2)**	**LR(2)**
$l_p = l_i$	**LR(1)**	**LR(2)**	**AMB**	**LR(1)**	**LR(2)**	**LR(1)**
$l_p < l_i$	**LR(2)**	**LR(2)**	**LR(1)**	**LR(2)**	**LR(1)**	**LR(2)**

Therefore, most of the combinations produce non-LR(1) PCGs. However, unlike previous scenarios, and with the exception of the case corresponding to the same precedence and yf and xfy types, which as indicated in the previous section leads to ambiguity, the resulting PCGs that are non-LR(1) are not ambiguous. On the contrary, they are LR(2) grammars.

Finally, remember that, as indicated in Table 1, there is also room for LR(1) PCGs for operator tables involving the infix and postfix forms of an operator. Indeed, an example is given in Figure 1, which overloads the $\otimes$ operator in infix and postfix forms.

5 Overloading an operator with infix, prefix and postfix fixities

The combinations of two operator definitions do not exhaust the conditions hindering LR(1) PCGs. Indeed, the overloading of an operator $\odot$ in infix, prefix and postfix forms can lead to ambiguity. The reason is that, in an expression like "$a \odot \odot a$", it is possible to consider: (i) the first occurrence of $\odot$ as a postfix operator and the second as an infix one; or (ii) the first as the infix operator and the second as a prefix one. In consecuence, let l_i be the precedence level of the infix definition, let τ_i be its type, let l_{pre} be the precedence level of the prefix definition, and l_{post} that of the postfix one. Then, any of the following conditions lead to an ambiguous PCG [6]:

- $\tau_i = xfy$, $l_{post} < l_i$, $l_{pre} \leqslant l_i$.
- $\tau_i = yfx$, $l_{post} \leqslant l_i$, $l_{pre} < l_i$.
- $\tau_i = xfx$, $l_{post} < l_i$, $l_{pre} < l_i$.

Figure 4 illustrates one of these cumbersome combinations. The resulting PCG (Figure 4b) is ambiguous, as Figure 4c makes apparent. The ambiguity of the PCGs produced by the other cumbersome combinations can be illustrated in an analogous way.

Finally, notice that, by avoiding the cumbersome combinations described in this and the previous sections, it is possible to find tables with an operator overloaded in infix, prefix and postfix forms that lead to LR(1) PCGs. Again an example is given by the $\otimes$ operator in Figure 1.

6 Related work

As illustrated in this paper, ambiguity caused by cumbersome combinations of operator attributes is one of the main causes of non-LR(1) PCGs. In [7], starting from a characterization of the expression languages defined through precedence relations between operators, it is proved that, in the absence of operator overloading, ambiguity can be prevented by avoiding

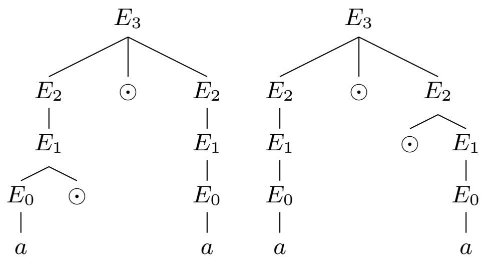
[6] Any of these conditions make the ambiguous sentence "$a \odot \odot a$" a valid expression of the language.

Name	Prec.	Type
$\odot$	3	xfx
$\odot$	2	fx
$\odot$	1	xf

(a) Operator table.

$$
\begin{aligned}
E_3 &\rightarrow E_2 \odot E_2 \mid E_2 \\
E_2 &\rightarrow \odot E_1 \mid E_1 \\
E_1 &\rightarrow E_0 \odot \mid E_0 \\
E_0 &\rightarrow a \mid (E_3)
\end{aligned}
$$

(b) Resulting PCG.



(c) Two parse trees for "$a \odot \odot a$"

Figure 4 Example regarding the overloading of an operator with infix, prefix and postfix fixities..

cycles in the operators' dependence graph. For the operators considered in our work these cycles only can arise between operators with the same precedence level and with opposite associativities. Therefore, this result is reflected in section 3. The work described in [1] proves that the syntax of expression languages without operator overloading in which each operator belongs to a different precedence level can be readily described with unambiguous CFGs. It is consistent with our analysis, since it leaves out all the cumbersome situations analyzed in the previous sections.

To a greater or lesser extent, languages with user-defined operators must cope with the aspects analyzed in this paper. Some representative examples of languages of this kind are Haskell, Scala, Sparrow and Prolog. For instance, Haskell [9] only provides support for user-defined infix operators with 10 precedence levels. No syntactic operator overloading is allowed. In addition, it is possible to find opposite associativies at the same precedence level, but expressions chaining left and right associative operators with the same precedence are rejected during parsing time. Scala [10] supports user-defined infix and postfix operators. It also supports a predefined set of prefix operators. Precedences are structured in two pre-established precedence classes (one class for prefix operators, another for infix ones), and each precedence class at a pre-established set of precedence levels. Actual precedence level and associativities are not literally declared but are derived from the operator's name. Associativity conflicts are managed as in Haskell. A similar approach is followed in Sparrow [14], although this language also allows user-defined prefix operators as well as the explicit declaration of precedences and associativies within each precedence class. Finally, as mentioned earlier, Prolog [5] supports definitions of operators analogous to those considered in this paper. In addition, the language includes some constraints on user-defined operators that, on one hand, avoid ambiguities and, on the other hand, facilitate parsing by limiting look-ahead. Specifically, it is not possible to define two operators with the same name and the same fixity (any attempt to do so redefines the operator instead of overloading it). It avoids the shortcomings described in section 2. Also, it is not possible to overload an operator as both an infix and a prefix one, which, on one hand avoids situations requiring more than one look-ahead symbol (like that described in section 4), and, on the other hand, avoids the potential ambiguities described in section 5. Finally, it solves the ambiguities derived from opposite associativities by making left-associative operators take priority over right-associative ones at the same precedence level. Therefore, as analyzed in this paper, all these languages exclude some perfectly valid combinations of operators from the point of view of unambiguity and limited (one symbol) look-ahead.

Finally, the implementation techniques for languages with user-defined operators are also relevant in the context of the current paper, since these techniques must also deal with valid and disallowed combinations of operator attributes. The work described in [11] describes how to use YACC to implement a parser for an expression language with arbitrary user-defined (not only prefix, postfix or infix) disfix operators. The approach does not support operator overloading. In addition, prefix operators always have a greater precedence level than infix ones, which in turn always belong to a greater level than the other dixfix operators, the ones with the highest priority. The work in [12] extends the approach based on ambiguous grammars and disambiguating rules in [2] by allowing the addition of new disambiguation rules during parsing time, which enables user-defined operators. It also provides some constraints to avoid cumbersome operator tables (that can lead to ambiguity or demand more than one look-ahead symbol), which are more restrictive than those posed by Prolog and those analyzed in the previous sections (for instance, one of the constraints requires all the overloaded operators to belong to the same precedence level). Finally, the work reported in [4] describes how to support dixfix operators by instantiating a PCG-like grammar scheme from precedence graphs that provide partial orderings on the precedence of the operators (instead of the total ordering provided by precedence levels). It also shows how, under the assumptions described in [7], the resulting CFG is unambiguous. Once again, none of these approaches accept all the legal combinations of operators identified in this paper.

7 Conclusions and Future work

In this paper we have explored the conditions under which the PCGs that model the syntax of expression languages become non-LR(1) CFGs. For this purpose, we have carried out a systematic analysis of combinations of two and three operator definitions, and we have characterized several problematic scenarios in grammatical terms. Most of them concern ambiguity in operator descriptions, which reflects ambiguous PCGs. Others expose the need for more than one look-ahead symbol. Although scenarios are avoided in most of the languages that support user-defined operators, from our analysis it is apparent that these design decisions could have been further refined (e.g., while Prolog prohibits overloading an operator in infix and prefix forms to limit the need for look-ahead, in this paper we have found that only certain combinations of this type of definitions lead to grammars which are not LR (1), but LR (2)). These findings also enable the direct analysis of operator tables, in order to diagnose potential problems and to explain such problems at the level of operator definitions (instead of, for instance, at the level of parsing conflicts in the generated PCGs).

While the set of combinations identified seems broad enough, the analysis performed has been fundamentally empirical. It is not possible, therefore, to affirm that an assertion like "*if an operator table does not contain any of the problematic combinations analyzed, then the resulting grammar will be* LR(1)" has been proved, but only that evidence in favor of it has been provided. It is necessary to carry out a more formal work oriented to proving this result or another similar to it, completing the catalog of problematic combinations if necessary. Another line of work is to consider where the resulting PCGs can be successfully transformed into appropriate CFGs for top-down parsing, as well as what the classes of these CGFs are (specially when these transformed CFGs are LL(1), or when they require more than one look-ahead symbol). Finally, we plan to run a similar analysis on the specifications based on ambiguous grammars plus disambiguating rules like those described by Aho et al [2].

References

1 Annika Aasa. Precedences in specifications and implementations of programming languages. *Theoretical Computer Science*, 142(1):3–26, 1995. doi:10.1016/0304-3975(95)90680-J.

2 Alfred. V. Aho, Sethi Johnson, and Jeffrey D. Ullman. Deterministic parsing of ambiguous grammars. In *ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 1–21, 1973. doi:10.1145/512927.512928.

3 Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 2nd edition, 2006.

4 Nils Anders Danielsson and Ulf Norell. Parsing mixfix operators. In *20th International Conference on Implementation and Application of Functional Languages*, pages 80–99, 2011.

5 Pierre Deransart, AbdelAli Ed-Dbali, and Laurent Cervoni. Prolog syntax. In *Prolog: The Standard: Reference Manual*, pages 221–238. Springer, 1996. doi:10.1007/978-3-642-61411-8_9.

6 Donald E. Knuth. On the translation of languages from left to right. *Information and Control*, 8(6):607–639, 1965.

7 Wafik Boulos Lotfallah. Characterizing unambiguous precedence systems in expressions without superfluous parentheses. *International Journal of Computer Mathematics*, 86(1):1–20, 2009. doi:10.1080/00207160802166499.

8 Kenneth C. Louden. *Compiler Construction: Principles and Practice*. PWS Publishing, 1997.

9 Simon Marlow, editor. *Haskell 2010 Language Report*. Haskell Community, 2010.

10 Martin Odersky. The Scala language specification, version 2.9. Technical report, Programming Methods Laboratory, EPFL, 2014.

11 Simon L. Peyton Jones. Parsing distfix operators. *Communications of the ACM*, 29(2):118–122, 1986. doi:10.1145/5657.5659.

12 Kjell Post, Allen Van Gelder, and James Kerr. Deterministic parsing of languages with dynamic operators. In *International Symposium on Logic Programming*, pages 456–472, 1993.

13 Ravi Sethi. *Programming Languages: Concepts and Constructs*. Addison-Wesley, 1989.

14 Lucian Radu Teodorescu, Vlad Dumitrel, and Rodica Potolea. Flexible operators in Sparrow. *International Journal of Research in Engineering and Technology*, 3(17):40–45, 2014.