LLLR Parsing: a Combination of LL and LR Parsing

Boštjan Slivnik

University of Ljubljana, Faculty of Computer and Information Science, Ljubljana, Slovenia
bostjan.slivnik@fri.uni-lj.si

Abstract

A new parsing method called LLLR parsing is defined and a method for producing LLLR parsers is described. An LLLR parser uses an LL parser as its backbone and parses as much of its input string using LL parsing as possible. To resolve LL conflicts it triggers small embedded LR parsers. An embedded LR parser starts parsing the remaining input and once the LL conflict is resolved, the LR parser produces the left parse of the substring it has just parsed and passes the control back to the backbone LL parser. The LLLR(k) parser can be constructed for any LR(k) grammar. It produces the left parse of the input string without any backtracking and, if used for a syntax-directed translation, it evaluates semantic actions using the top-down strategy just like the canonical LL(k) parser. An LLLR(k) parser is appropriate for grammars where the LL(k) conflicting nonterminals either appear relatively close to the bottom of the derivation trees or produce short substrings. In such cases an LLLR parser can perform a significantly better error recovery than an LR parser since the most part of the input string is parsed with the backbone LL parser. LLLR parsing is similar to LL(*) parsing except that it (a) uses LR(k) parsers instead of finite automata to resolve the LL(k) conflicts and (b) does not perform any backtracking.

1998 ACM Subject Classification F.4.2 Grammars and Other Rewriting Systems (Parsing), D.3.1 Formal Definitions and Theory (Syntax), D.3.4 Processors (Parsing)

Keywords and phrases LL parsing, LR languages, left parse

Digital Object Identifier 10.4230/OASIcs.SLATE.2016.5

1 Introduction

As the syntax-directed translation represents the core of virtually any modern compiler's front-end, the parser as the implementation of the syntax analysis form an important part of any compiler. Hence, the parser should provide the compiler writer with (a) a solid framework for evaluation of semantic actions during or after parsing and (b) an appropriate support for producing detailed syntax error reports. The two predominant techniques for parsing programming languages are the top-down and the bottom-up parsing. LL parsers represent the core of the first group and LR parsers the core of the second. Both techniques have their advantages and disadvantages. The top-down parsing provides "the readability of recursive descent (RD) implementations of LL parsing along with the ease of semantic action incorporation" while "an LL parser is linear in the size of the grammar"; the bottom-up parsing is regarded highly for "the extended parsing power of LR parsers, in particular the admissibility of left recursive grammar" but "even LR(0) parse tables can be exponential in the size of the grammar" (all quotes are from [12]). The parsers incorporating both top-down and bottom-up parsing (left-corner parsers, etc. [2, 4, 5, 7]) never gained much popularity because of the confusing order in which the semantic actions are triggered.

▶ **Example 1.** Much older XLC(1) and LAXLC(1) parsers [4, 5] are extensions of left-corner parsing and (like LLLR parsing) employ both top-down and bottom-up approaches. However, XLC(1) and LAXLC(1) parsers produce neither left nor right trace of the input string. For instance, if string *abbbbccc* is parsed using an XLC(1) or an LAXLC(1) parser for grammar

$$[A \longrightarrow aBbC], [B \longrightarrow Bb], [B \longrightarrow b], [C \longrightarrow Cc], \text{ and } [C \longrightarrow c],$$

the output

$$[A \longrightarrow aBbC][B \longrightarrow b][B \longrightarrow Bb][B \longrightarrow Bb][C \longrightarrow c][C \longrightarrow Cc][C \longrightarrow Cc]$$

is neither left nor (reverse) right parse of the input string. As the first production printed out expands the start symbol at the top of the derivation tree the evaluation of semantic routines starts at the top. But then the evaluation of semantic routines suddenly changes from the top-down into the bottom up approach since the second production printed out produces the leaf several levels below the root of the tree. Later the evaluation changes direction from bottom-up to top-down and vice-versa a few more times.

In general, during XLC(1) and LAXLC(1) parsing the evaluation of semantic routines might become quite confusing as it changes direction all the time. ◀

Likewise, Packrat parsers [3] used for Parsing Expression Grammars, have not become popular as "... it can be quite difficult to determine what language is defined by a TDPL program." [1]. In recent years the focus has shifted strongly towards the top-down parsing and towards LL parsing in particular [11, 18, 13, 12, 8, 16, 17] – even to the point that yacc was erroneously considered dead [6]. On one hand, a GLL parser capable of parsing a language of any context-free grammar, was formulated [12], but in the worst case it runs in cubic time. Furthermore, an LL(*) parser using (a) finite automata to resolve certain LL(k) conflicts and (b) backtracking to resolve the others, has been implemented in ANTLR [8]. Furthermore, "the LL(*) grammar condition is statically undecidable and grammar analysis sometimes fails to find regular expressions that distinguish between alternative productions" [9].

On the other hand, LR parsers were modified to produce the left parse of its input and thus giving the compiler writer the impression of top-down parsing [11, 18]. However, these parsers either (a) produce the first production of the left parse (and thus trigger the first semantic actions) only after the entire input string has been parsed [11] or (b) they further increase the difference of how much space LR or LL parser needs by introducing two additional parsing tables [18]. One could also use a much stronger GLR parsing but the time complexity is O(np+1) (where p is the length of the longest production). For deterministic grammars, however, both GLL and GLR parsers should run in near-linear time [9].

Finally, ALL(*) parsing performs grammar analysis in parse-time to combine "the simplicity, efficiency, and predictability of conventional top-down LL(k) parsers with the power of a GLR-like mechanism to make parsing decisions" [9]. It achieves almost linear time for majority of grammars used in practice where it is consistently faster than GLL and GLR parsing and can compete with LL or LALR parsing. Nevertheless, its worst case complexity is $O(n^4)$.

The idea behind LLLR parsing is to produce the left parse of the input string by using LL parsing as much as possible and to use small LR parsers only to avoid LL conflicts [19]. Hence, LLLR parsing uses LR parsers where LL(*) and ALL(*) parsing use deterministic finite automata "even though lookahead languages (set of all possible remaining input phrases) are often context-free" [9].

Table 1 LLLR(1) parsing of $bbbbaab \in L(G_{ex2})$. Whenever the LR(1) parser for A stops, it handles the remaining part of the right side of production for A, i.e., symbols that has not yet appeared on the stack as the input has not yet been reduced to these symbols, back to the backbone LL(1) parser (bA in the first instance and a in the second).

	STACK	INPUT	ACTION	OUTPUT
1	$\$S$	$bbbbaab\$$	LL produce $[S \longrightarrow bBab]$	$[S \longrightarrow bAab]$
2	$\$baAb$	$bbbbaab\$$	LL shift b	
3	$\$baA$	$bbbaab\$$	— start the LR parser for A	
4	$\$baq_0$	$bbbaab\$$	LR shift b	
5	$\$baq_0bq_1$	$bbaab\$$	LR reduce on $[B \longrightarrow b]$	
6	$\$baq_0Bq_1$	$bbaab\$$	— stop the LR parser for A	$[A \longrightarrow BbA][B \longrightarrow b]$
7	$\$baAb$	$bbaab\$$	LL shift b	
8	$\$baA$	$baab\$$	— start the LR parser for A	
9	$\$baq_0$	$baab\$$	LR shift b	
10	$\$baq_0bq_1$	$aab\$$	— stop the LR parser for A	$[A \longrightarrow ba]$
11	$\$baa$	$aab\$$	LL shift a	
12	$\$ba$	$ab\$$	LL shift a	
13	$\$b$	$b\$$	LL shift b	
14	$\$$	$\$$	LL accpet	

▶ **Example 2.** Consider the grammar $G_{ex2} \in \text{LR}(1) \setminus \text{LL}(1)$ with the start symbol S and productions

$$[S \longrightarrow bAab], [A \longrightarrow ba], [A \longrightarrow BbA], \text{ and } [B \longrightarrow b].$$

The trace of LLLR(1) parsing of string $bbbbaab \in L(G_{ex2})$ is shown in Table 1. LLLR(1) parsing starts with LL(1) parsing, but to avoid the LL(1) conflict on b for A (line 3), the embedded left LR(k) parser for A is started (line 4). After the 2nd b of the input string is shifted and reduced to B using production $[B \longrightarrow b]$, the 3rd b appears in the lookahead buffer and the embedded left LR(k) parser deduces that productions $[A \longrightarrow bbA]$ and $[B \longrightarrow b]$ are part of the left parse if the input string. It prints out both productions and passes the control back to the backbone LL(1) parser together with suffix bA of the right side of $[A \longrightarrow bbA]$ as these two symbols have not yet been handled by LR(1) parser yet (line 6).

The same pattern repeats in lines 8, 9 and 10 except that the production $[A \longrightarrow ba]$ is printed out and that only suffix a is returned to the backbone LL(1) parser.

The mechanism that enables the embedded LR(1) parser to deduce which production expanding A must be used in either case is explained in [16], the procedure for printing out the left parse instead of the right parse during LR parsing has been described in [11]. ◀

Example 2 contains a simple case that can be handled by, for instance, LL(*) parsing. Harder cases, e.g., (chained) left-recursive nonterminals, etc., must be parsed using LLLR parsing which can be used for parsing any language generated by an LR(k) grammar. The paper is organized in the way an LLLR parser is constructed. After Section 2, which provides the reader with some preliminary issues, Section 3 introduces different kinds of conflicts that can appear during LLLR parsing. Section 4 describes the construction of the LLLR parser while Section 5 provides a method used for eliminating redundant entries in the generated parser tables. Before Conslusions, another section is dedicated to a test case – an illustration how LLLR parsing performs on the grammar for the Java programming language.

2 Preliminaries

An intermediate knowledge of LL and LR parsing is presumed and the standard terminology of formal language theory and parsing is assumed [14, 15]. In addition, let

$$\text{FSTFLW}_k^G(\alpha, A) = \cup_{z \in \text{FOLLOW}_k^G(A)} \text{FIRST}_k^G(\alpha z)$$

and let $T^{*k} = T^0 \cup T^1 \cup \ldots \cup T^k$.

The LLLR parser is composed of (a) an LL parser based on the SLL parsing table and (b) the embedded left LR(k) parser. For the lack of space, only a short overview of both methods are given here.

2.1 The SLL(k) parsing

An LL(k) grammar $G = \langle N, T, P, S \rangle$ can be transformed to an equivalent SLL(k) grammar $\bar{G} = \langle \bar{N}, T, \bar{P}, \bar{S} \rangle$ [15] where the set of nonterminals is defined as

$$\bar{N} = \{ \langle A, \mathcal{F}_A \rangle \,;\, S \Longrightarrow_{\text{lm}}^* uA\delta \wedge \mathcal{F}_A = \text{FIRST}_k^G(\delta) \}$$

and the initial symbol is $\bar{S} = \langle S, \{\varepsilon\} \rangle$; for any nonterminal $\langle A, \mathcal{F}_A \rangle \in \bar{N}$ and any production $[A \longrightarrow X_1 X_2 \ldots X_n] \in P$ the grammar $\bar{G}$ contains production

$$[\langle A, \mathcal{F}_A \rangle \longrightarrow \bar{X}_1 \bar{X}_2 \ldots \bar{X}_n]$$

where

$$\bar{X}_i = \begin{cases} X_i & X_i \in T \\ \langle X_i, \text{FIRST}_k^G(X_{i+1} X_{i+2} \ldots X_n \mathcal{F}_A) \rangle & X_i \in N \end{cases} .$$

Regardless of whether $G \in LL(k)$), (a) $L(\bar{G}) = L(G)$, and (b) every nonterminal A of $\bar{G}$ appears in exactly one right-context $\mathcal{F}$(substrings derived from the nonterminal A are always followed by strings from $\mathcal{F}$). However, $G \in \text{LL}(k) \Longleftrightarrow \bar{G} \in \text{SLL}(k)$.

The most straightforward method for producing the LL(k) parser for an \$-augmented SLL(k) grammar $G = \langle N, T, P, S \rangle$ is a construction of the LL table

$$\text{LL-TABLE}\colon N \times T^{*k} \longrightarrow 2^P$$

where LL-TABLE(A, x) is defined as

$$\{ [A \longrightarrow \alpha] \,;\, \forall [A \longrightarrow \alpha] \in P \colon x \in \text{FSTFLW}_k^G(\alpha, A) \} \quad .$$

The cardinality of LL-TABLE(A, x) indicates different issues. If $|\text{LL-TABLE}(A, x)| = 1$, the LL($k$) table indicates the regular selection of the next production that is to be used during LL(k) parsing when the nonterminal $A \in N$ is on the top of the parser stack and $x \in T^{*k}$ is in the lookahead buffer. $|\text{LL-TABLE}(A, x)| = 0$ indicates the syntax error while $|\text{LL-TABLE}(A, x)| > 1$ represents the LL(k) conflict.

It is assumed that prior the construction of an LLLR(k) parser, a grammar is transformed by the LL-to-SLL transformation described above.

2.2 The embedded left LR(k) parser

Given a grammar $G = \langle N, T, P, S \rangle$, the embedded left LR($k$) grammar for $\alpha \in (N \cup T)^*$ and $\mathcal{F} \in (T \cup \{\$\})^{*k}$ [16, 17] is an LR(k) parser that

1. expects a string starting with a prefix wz, where $z \in \mathcal{F}$ and $\alpha \Longrightarrow_G^* w$, on its input;
2. parses a prefix u of w, i.e., $w = uv$ for some v, without shifting any symbol of z on the stack;
3. returns the left parse $\pi_u \in P^*$ and the viable suffix $\delta^R \in (N \cup T)^*$ found in the derivation

$$\alpha z \Longrightarrow_{G,\mathrm{lm}}^{\pi_u} u\delta z.$$

The embedded left LR(k) parser is the left LR(k) parser [18] for the (reduced) grammar $G(\alpha, \mathcal{F}) = \langle N \cup \{S_1, S_2\}, T, P_{\alpha,\mathcal{F}}, S_1\rangle$ where $S_1, S_2 \notin (N \cup T)$ and

$$P_{\alpha,\mathcal{F}} = P \cup \{[S_1 \longrightarrow S_2 z], [S_2 \longrightarrow \alpha]\,;\, z \in \mathcal{F}\}.$$

The trick is that, if the parser does not stop earlier, the reduction on $[S_2 \longrightarrow \alpha]$ is regarded as a signal to accept the input string w leaving the string z in the lookahead buffer. Furthermore, if $G \in \mathrm{LR}(k)$, then testing whether $G(\alpha, \mathcal{F})$ can be done very efficiently [17].

The embedded left LR(k) parser is called 'embedded' since it parses only a substring of the entire input string and can thus be embedded into other parsers. It is called 'left' because it produces the left parse (instead of the right parse) of the substring it parsed.

3 Identifying conflicts

Computing a set of LL(k) conflicts for a given \$-augmented grammar $G = \langle N, T, P, S\rangle$ is straightforward, but it gets slightly more complicated if these conflicts are to be resolved using small LR(k) parsers embedded into the backbone LL(k) parser.

Suppose that a string $\$uv\$ \in L(G)$ is derived by the leftmost derivation

$$S \Longrightarrow_{G,\mathrm{lm}}^* uB\delta \Longrightarrow_{G,\mathrm{lm}} u\beta_1 A\beta_2\delta \Longrightarrow_{G,\mathrm{lm}}^* uv$$

and that the grammar exhibits an LL(k) conflict on the lookahead string $x \in \mathrm{FIRST}_k^G(A\beta_2\delta\$)$ for the nonterminal symbol A. However, as demonstrated by Example 3, the correct termination of the LR(k) parser for the substrings derived from A is not always guaranteed.

▶ **Example 3.** Consider the grammar $G_{\mathrm{ex3}} \in \mathrm{LR}(1) \setminus \mathrm{LL}(1)$ with the start symbol S' and productions

$$[S \longrightarrow bBab], [B \longrightarrow A], [A \longrightarrow b] \text{ and } [A \longrightarrow Ba].$$

The trace of parsing the input string starting with bba using the backbone LL(1) parser and the embedded LR(1) parser for the LL(1) conflicting nonterminal A is shown in Table 2. The configuration $\$baA|ba\ldots\$$ results in the LL(1) conflict on b for A where the embedded LR(1) parser for A is started (line 4). However, after pushing b and reducing it to A the LR(1) parser cannot determine whether the first a of $bba\ldots$ is derived from A or not (line 7). Namely, if the input string is $bbab$, the LR(1) parser must accept, terminate and handle the control back to the backbone LL(1) parser; otherwise the LR(1) parser must perform the reduction on $[B \longrightarrow A]$ and continue parsing.

As shown in Table 3, the problem remains even if the embedded LR(1) parser for B is used instead (since A is initially derived from B). However, the embedded LR(1) parser for parsing substrings derived from Ba and followed by $b\$$ can terminate correctly and should be used. ◀

Table 2 Parsing a string starting with bba using an LL(1) parser for G_{ex3} and an LR(1) parser for nonterminal symbol A.

	STACK	INPUT	ACTION
1	$\$S$	$bba\ldots\$$	LL produce $[S \longrightarrow bBab]$
2	$\$baBb$	$bba\ldots\$$	LL shift b
3	$\$baB$	$ba\ldots\$$	LL produce $[B \longrightarrow A]$
4	$\$baA$	$ba\ldots\$$	— start a parser for A —
5	$\$baq_0$	$ba\ldots\$$	LR shift b
6	$\$baq_0bq_1$	$a\ldots\$$	LR reduce on $[A \longrightarrow a]$
7	$\$baq_0Aq_2$	$a\ldots\$$	LR reduce on $[B \longrightarrow a]$ or accept?

Table 3 Parsing a string starting with bba using an LL(1) parser for G_{ex3} and an LR(1) parser for nonterminal symbol B.

	STACK	INPUT	ACTION
1	$\$S$	$baa\ldots\$$	LL produce $[S \longrightarrow bBab]$
2	$\$baBb$	$baa\ldots\$$	LL shift b
3	$\$baB$	$aa\ldots\$$	— start a parser for B —
4	$\$baq_0$	$aa\ldots\$$	LR shift a
5	$\$baq_0aq_1$	$a\ldots\$$	LR reduce on $[A \longrightarrow a]$
6	$\$baq_0Aq_2$	$a\ldots\$$	LR reduce on $[B \longrightarrow a]$
7	$\$baq_0Bq_2$	$a\ldots\$$	LR shift a or accept?

Example 3 shows that there are two kinds of conflicting nonterminals in LLLR(k) parsing:

1. *Genuine conflicting nonterminals* are all SLL(k) conflicting nonterminals (symbol A in Example 3).
2. *Induced conflicting nonterminals* are all nonterminals which are not SLL(k) conflicting but must be treated as conflicting so that the embedded LR(k) parser can terminate correctly (symbol B in Example 3).

Algorithm 1 computes the set of all LLLR conflicting nonterminals. It starts with computing genuine conflicting nonterminals (set $C^{(0)}$ in lines 2–3). Afterwards, it adds all induced conflicting nonterminals: if A is a conflicting nonterminal in production $[B \longrightarrow \beta_1 A \beta_2]$, then there should exist a prefix β_2' of β_2 so that the embedded LR(k) parser for $A\beta_2'$ stops correctly – if β_2' does not exist, then A induces B and B is added to the set of conflicting nonterminals (lines 6–9). Finally, the set C contains all conflicting nonterminals (while $C \setminus C^{(0)}$ contains all genuine conflicting nonterminals).

The most time consuming part of Algorithm 1 is testing whether the embedded LR(k) parser can always stop – the condition in line 9. Fortunately, if the grammar G for which the LLLR(k) parser is being made is an LR(k) grammar, the test can be performed quite efficiently [17].

4 Constructing the LLLR parser

After the conflicting nonterminals have been computed, the parsing table of the backbone LL(k) parser can be computed using Algorithm 2. The basic idea is simple: replace all (genuine or induced) conflicting nonterminals appearing on the right side of any production with markers which trigger the appropriate embedded left LR(k) parsers.

Algorithm 1 Evaluation of LLLR conflicting nonterminals.
INPUT: A grammar G after the LL-to-SLL transformation.
OUTPUT: A set $C^{(0)}$ of genuine conflicting nonterminals and a set C of all conflicting nonterminals.

1: $i \leftarrow 0$
2: $C^{(0)} \leftarrow \{A\,;\, \exists[A \longrightarrow \alpha_1], [A \longrightarrow \alpha_2] \in P\colon \mathrm{FSTFLW}_k^G(\alpha_1,A) \cap \mathrm{FSTFLW}_k^G(\alpha_2,A) \neq \emptyset\}$
3: **repeat**
4: $\quad i \leftarrow i+1$
5: $\quad C^{(i)} \leftarrow C^{(i-1)} \cup$
6: $\qquad \{B\,;\, \exists[B \longrightarrow \beta_1 A \beta_2] \in P, A \in C^{(i-1)}\colon$
7: $\qquad\quad \forall \beta_2', \beta_2''\colon \big(\beta_2'\beta_2'' = \beta_2 \Longrightarrow G(A\beta_2', \mathrm{FSTFLW}_k^G(\beta_2'',B)) \notin \mathrm{LR}(k)\big)\}$
8: **until** $C^{(i)} = C^{(i-1)}$
9: $C = C^{(i)}$

More precisely, if A is the only conflicting nonterminal in $[B \longrightarrow \beta_1 A \beta_2]$, then

$$\text{LL-TABLE}(B,x) = [B \longrightarrow \beta_1 [\![A\beta_2', \mathrm{FSTFLW}_k^G(\beta_2'', B)]\!] \beta_2'']$$

where β_2' is the shortest prefix of $\beta_2 = \beta_2'\beta_2''$ such that the embedded left LR(k) parser for $A\beta_2'$ can stop on any string $x \in \mathrm{FSTFLW}_k^G(\beta_2'', B)$. The newly created nonterminal symbol $[\![A\beta_2', \mathrm{FSTFLW}_k^G(\beta_2'', B)]\!]$ acts as a trigger for starting the embedded left LR(k) parser. If there are more conflicting nonterminals on the right hand side of a production, then the described substitution is performed from left to right. Furthermore, the embedded left LR(k) grammar for $G(A\beta_2', \mathrm{FSTFLW}_k^G(\beta_2'', B))$ must be constructed regardless of whether B is a conflicting nonterminal or not.

The left-to-right scan of the sentential form on the right side of a production being transformed is performed in lines 3–17 of Algorithm 2; the computation of the shortest sentential form a particular embedded left LR(k) parser is to be made for, is computed in lines 8–15.

There are two key insights into the correctness of Algorithm 2: (a) the loop in lines 8–15 terminates successfully, i.e., by finding an appropriate prefix $X_i X_{i+1} \ldots X_j$ and exiting after reaching line 11, and (b) no element of LL-TABLE contains more than one production. For if either of these two statements were false, the nonterminal A should have been considered a conflicting nonterminal by Algorithm 1.

▶ **Example 4.** Consider a grammar G_{ex4} with the start symbol S and productions

$[S \longrightarrow aAabCb]$,
$[A \longrightarrow aEB]$, $[A \longrightarrow Aaa]$, $[C \longrightarrow aaD]$, $[C \longrightarrow abE]$,
$[B \longrightarrow bb]$, $[D \longrightarrow aa]$, $[E \longrightarrow a]$ and $[E \longrightarrow Aa]$.

As symbols A, C and E are conflicting nonterminals, the LL(1) parsing table contains the following entries:

$$\begin{aligned}
\text{LL-TABLE}(S,a) &= [S \longrightarrow a[\![Aa, \{b\}]\!] b [\![C, \{b\}]\!] b] \\
\text{LL-TABLE}(B,b) &= [B \longrightarrow bb] \\
\text{LL-TABLE}(D,a) &= [D \longrightarrow aa]
\end{aligned}$$

The backbone LL(1) parser is augmented with the embedded left LR(1) parsers for the following grammars:

$G(Aa, \{b\})$, $G(C, \{b\})$ and $G(E, \{b\})$.

Algorithm 2 Construction of the backbone LL(k) parser.
INPUT: A grammar G after the LL-to-SLL transformation and a set C of all conflicting nonterminals for G.
OUTPUT: The LL parsing table for the backbone LL parser.

```
1: for [A ⟶ X1X2 ... Xn] ∈ P ∧ A ∉ C do
2:   α ← ε ; i ← 1
3:   while (i ≤ n) do
4:     if Xi ∉ C then
5:       α ← αXi ; i ← i + 1
6:     else
7:       j ← i
8:       while (i ≤ j) do
9:         F = FSTFLW_k^G(Xj+1 ... Xn, A)
10:        if G(Xi ... Xj, F) ∈ LR(k) then
11:          α ← α⟦Xi ... Xj, F⟧ ; i ← j + 1
12:        else
13:          j ← j + 1
14:        end if
15:      end while
16:    end if
17:  end while
18:  for x ∈ FSTFLW_k^G(X1 ... Xn, A) do
19:    LL-TABLE(A, x) ← {[A ⟶ α]}
20:  end for
21: end for
```

LLLR(1) parsing of string *aaaabbaaabaaaab* is shown in Table 4:

1. The parsing starts with printing out the the first production of the left parse, i.e., $[S \longrightarrow aAa\ bCb]$ (line 1) and shifting the first symbol.
2. When $[\![Aa, \{b\}]\!]$ appears on the top of the stack, the embedded LR(1) parser for $G(Aa, \{b\})$ is started. While parsing the substring *aaabbaaa* derived from Aa, the left parse $\pi_{Aa} = [A \longrightarrow Aaa][A \longrightarrow aEB][E \longrightarrow Ea][E \longrightarrow a][B \longrightarrow bb]$ is accumulated on the stack (lines 3–17). When b is in the lookahead buffer, the left parse π_{Aa} is printed out and the viable suffix ε returned to the backbone LL(1) parser (line 17).
3. After shifting b, the embedded parser for $G(C, \{b\})$ is started. After shifting a, the lookahead buffer contains another a and the parser recognizes the production $[C \longrightarrow aaD]$ (line 22). It prints this production out and returns the viable suffix aD – the part of the production not yet matched against the input.
4. Afterwards, the backbone parser finishes the job.

Hence, the LLLR(1) parser prints the left parse

$$[S \longrightarrow aAabCb][A \longrightarrow Aaa][A \longrightarrow aEB]$$
$$[E \longrightarrow Ea][E \longrightarrow a][B \longrightarrow bb][C \longrightarrow aaD].$$

As the symbols A and E are left-recursive, the parsers for $G(Aa, \{b\})$ and $G(E, \{b\})$ must always parse the entire string generated by either grammar. However, the parser for $G(C, \{b\})$ always shifts only the first a and returns the viable suffix aD (if a is in the lookahead buffer) or the viable suffix $b[\![E, \{b\}]\!]$ (if b is in the lookahead buffer).

Table 4 Parsing a string starting with bba using an LL(1) parser for G_{ex3} and an LR(1) parser for B. (All LR(1) states of both embedded left LR(1) parsers are presented generically with q; the subscripts denote left parses accumulated on stack during left LR parsing [11].)

	STACK	INPUT	ACTION
1	$\$S$	$aaaabbaaabaaaab\$$	LL produce $[S \longrightarrow a[\![Aa, \{b\}]\!]b[\![C, \{b\}]\!]b]$
2	$\$b[\![C]\!]b[\![Aa]\!]a$	$aaaabbaaabaaaab\$$	LL shift a
3	$\$b[\![C, \{b\}]\!]b[\![Aa, \{b\}]\!]$	$aaabbaaabaaaab\$$	LL starts the LR parser for grammar $G(Aa, \{b\})$
4	$\$b[\![C, \{b\}]\!]b\langle q_\varepsilon\rangle$	$aaabbaaabaaaab\$$	LR shift a
5	$\$b[\![C, \{b\}]\!]b\langle q_\varepsilon\rangle a\langle q_\varepsilon\rangle$	$aabbaaabaaaab\$$	LR shift a
6	$\$b[\![C, \{b\}]\!]b\langle q_\varepsilon\rangle a\langle q_\varepsilon\rangle a\langle q_\varepsilon\rangle$	$abbaaabaaaab\$$	LR reduce on $p_1 = [E \longrightarrow a]$
7	$\$b[\![C, \{b\}]\!]b\langle q_\varepsilon\rangle a\langle q_\varepsilon\rangle E\langle q_{p_1}\rangle$	$abbaaabaaaab\$$	LR shift a
8	$\$b[\![C, \{b\}]\!]b\langle q_\varepsilon\rangle a\langle q_\varepsilon\rangle E\langle q_{p_1}\rangle a\langle q_\varepsilon\rangle$	$bbaaabaaaab\$$	LR reduce on $p_2 = [E \longrightarrow Ea]$
9	$\$b[\![C, \{b\}]\!]b\langle q_\varepsilon\rangle a\langle q_\varepsilon\rangle E\langle q_{p_2p_1}\rangle$	$bbaaabaaaab\$$	LR shift b
10	$\$b[\![C, \{b\}]\!]b\langle q_\varepsilon\rangle a\langle q_\varepsilon\rangle E\langle q_{p_2p_1}\rangle b\langle q_\varepsilon\rangle$	$baaabaaaab\$$	LR shift b
11	$\$b[\![C, \{b\}]\!]b\langle q_\varepsilon\rangle a\langle q_\varepsilon\rangle E\langle q_{p_2p_1}\rangle b\langle q_\varepsilon\rangle b\langle q_\varepsilon\rangle$	$aaabaaaab\$$	LR reduce on $p_3 = [B \longrightarrow bb]$
12	$\$b[\![C, \{b\}]\!]b\langle q_\varepsilon\rangle a\langle q_\varepsilon\rangle E\langle q_{p_2p_1}\rangle B\langle q_{p_3}\rangle$	$aaabaaaab\$$	LR reduce on $p_4 = [A \longrightarrow aEB]$
13	$\$b[\![C, \{b\}]\!]b\langle q_\varepsilon\rangle A\langle q_{p_4p_2p_1p_3}\rangle$	$aaabaaaab\$$	LR shift a
14	$\$b[\![C, \{b\}]\!]b\langle q_\varepsilon\rangle A\langle q_{p_4p_2p_1p_3}\rangle a\langle q_\varepsilon\rangle$	$aabaaaab\$$	LR shift a
15	$\$b[\![C, \{b\}]\!]b\langle q_\varepsilon\rangle A\langle q_{p_4p_2p_1p_3}\rangle a\langle q_\varepsilon\rangle a\langle q_\varepsilon\rangle$	$abaaaab\$$	LR reduce on $p_5 = [A \longrightarrow Aaa]$
16	$\$b[\![C, \{b\}]\!]b\langle q_\varepsilon\rangle A\langle q_{p_5p_4p_2p_1p_3}\rangle$	$abaaaab\$$	LR shift a
17	$\$b[\![C, \{b\}]\!]b\langle q_\varepsilon\rangle A\langle q_{p_5p_4p_2p_1p_3}\rangle a\langle q_\varepsilon\rangle$	$baaaab\$$	LR stops: lm parse $= p_5p_4p_2p_1p_3$ remaining suffix $= \varepsilon$
18	$\$b[\![C, \{b\}]\!]b$	$baaaab\$$	LL shift b
20	$\$b[\![C, \{b\}]\!]$	$aaaab\$$	LL starts the parser for grammar $G(C, \{b\})$
21	$\$b\langle q_\varepsilon\rangle$	$aaaab\$$	LR shift a
22	$\$b\langle q_\varepsilon\rangle a\langle q_\varepsilon\rangle$	$aaab\$$	LR stops: lm parse $= p_4$ remaining suffix Da
23	$\$bDa$	$aaab\$$	LL shift a
24	$\$bD$	$aab\$$	LL produce $[D \longrightarrow aa]$
25	$\$baa$	$aab\$$	LL shift a
26	$\$ba$	$ab\$$	LL shift a
27	$\$b$	$b\$$	LL shift b
28	$\$$	$\$$	LL accept `:-)`

Finally, note that not all LL-TABLE entries generated by Algorithm 2 are needed – LL-TABLE(B, b) is not needed as substrings derived from B will always be parsed by a parser for $G(Aa, \{b\})$. Likewise, if C is eliminated from the grammar, the parser for $G(E, \{b\})$ is not needed anymore for the same reason. ◀

An LLLR(k) parser can be constructed for any LR(k) grammar G: if not otherwise, the start symbol gets included into the set of induced conflicting symbols and hence the entire input string is parse by a single embedded left LR(k) parser. However, this is not the advised use of LLLR parsing.

5 Reducing the LLLR Parser

As illustrated by Example 4, the construction of the LLLR parser described in Section 4 might produce some redundant LL-TABLE entries or even some redundant embedded parsers.

Since it it obvious that LL-TABLE entries for the start symbol of the grammar are not redundant, it is trivial to compute what entries and parsers are needed by the backbone LL(k) parser and thus the rest of this section is dedicated to identifying all symbols that can form a viable suffix returned by an embedded left LR(k) parser. Once this symbols are known, it is again straightforward to see which entries and parsers are needed on the basis of embedded parsers.

The key insight comes from the understanding of how the embedded left LR(k) parser computes the viable suffix. Namely, that parser uses an additional parsing table LEFT that maps certain parser states and lookahead strings to LR(0) items. More precisely, if

$$\text{LEFT}(q, x) = [A \longrightarrow \alpha \bullet \beta],$$

then for each viable prefix γ the state q is associated with, i.e., for every LR(k) stack contents where q is at the top, there exists a single γ-path of items in the nondeterministic LR(0) machine starting with the initial LR(0) item and terminating with item $[A \longrightarrow \alpha \bullet \beta]$ [18, 17]. The viable suffixes consist from the sentential forms found on the right side of $\bullet$ in this LR(0) items and thus these sentential forms must be checked to find symbols that can form a viable suffix returned by an embedded left LR(k) parser. This in fact is done by Algorithm 3 which must be run for every embedded parser.

6 A test case: the Java Language

To test the new parsing method against the real programming language, Java 1.0 has been chosen. There are two reasons for this: (a) there is an official LALR grammar for Java 1.0 available (Gosling et. al, The Java™ Language Specification, 1996), and (b) choosing a language primarily made for parsing in a top-down manner would be unfair.

As the official Java 1.0 grammar is LALR, it contains a lot of left-recursive nonterminals. However, one must distinguish between two kinds of left-recursive nonterminals:

1. *Essential left-recursive nonterminals* are those where left recursion is used to achieve the proper form of derivation trees.

   The best examples of essential left-recursive nonterminals are those describing arithmetic expressions as the left recursion is needed to emphasize the left associativity of various arithmetic operators. For instance, productions

   AdditiveExpression ⟶ MultiplicativeExpression
   AdditiveExpression ⟶ AdditiveExpression + MultiplicativeExpression
   AdditiveExpression ⟶ AdditiveExpression – MultiplicativeExpression

Algorithm 3 Enumeration of all symbols that can form a viable suffix returned by an embedded left LR(k) parser.
INPUT: An embedded left LR(k) parser.
OUTPUT: A set of symbols $\mathcal{L}$.

enumerate all items needed to compute the viable suffix:
1: $i \leftarrow 0$
2: $\mathcal{I}^{(0)} \leftarrow \{\ \langle [A \to \alpha \bullet \beta, x], q \rangle\ ;$
3: $\quad \mathrm{LEFT}(q, z) = [A \to \alpha \bullet \beta]$
4: $\quad \wedge\, [A \to \alpha \bullet \beta, x] \in q$
5: $\quad \wedge\, z \in \mathrm{FIRST}_k^G(\beta x)\ \}$
6: **repeat**
7: $\quad i \leftarrow i + 1$
8: $\quad \mathcal{I}^{(i)} \leftarrow \mathcal{I}^{(i-1)} \cup \{\ \langle [A' \to \alpha' \bullet A\beta', x'], q' \rangle\ ;$
9: $\quad \langle [A \to \alpha \bullet \beta, x], q \rangle \in \mathcal{I}^{(i-1)}$
10: $\quad \wedge\, \hat{\delta}(q', \alpha) = q$
11: $\quad \wedge\, x \in \mathrm{FIRST}_k^G(\beta' x')\ \}$
12: **until** $\mathcal{I}^{(i)} = \mathcal{I}^{(i-1)}$
13: $\mathcal{I} = \mathcal{I}^{(i)}$
enumerate all symbols
14: $\mathcal{L} = \emptyset$
15: **for** $\langle [A \to \alpha \bullet X_1 X_2 \ldots X_n, x], q \rangle \in \mathcal{I}^{(0)}$ **do**
16: $\quad \mathcal{L} \leftarrow \mathcal{L} \cup \{X_i, X_{i+1}, \ldots, X_n\}$
17: **end for**
18: **for** $\langle [A \to \alpha \bullet A' X_1 X_2 \ldots X_n, x], q \rangle \in \mathcal{I} \setminus \mathcal{I}^{(0)}$ **do**
19: $\quad \mathcal{L} \leftarrow \mathcal{L} \cup \{X_i, X_{i+1}, \ldots, X_n\}$
20: **end for**

describing the structure of additive expressions should not be changed as otherwise the programmer needs to modify the derivation tree manually.

2. *Nonessential left-recursive nonterminals* are those where the left recursion is used to make a grammar more suitable for a particular parsing algorithm.
   For instance, the Java 1.0 grammar includes a number of non-essential left recursion like

   $$\begin{aligned} &\text{ClassBodyDeclarations} \longrightarrow \text{ClassBodyDeclaration} \\ &\text{ClassBodyDeclarations} \longrightarrow \text{ClassBodyDeclarations ClassBodyDeclaration} \end{aligned}$$

   that can easily be rewritten to

   $$\begin{aligned} &\text{ClassBodyDeclarations} \longrightarrow \text{ClassBodyDeclaration ClassBodyDeclarations}_{\text{opt}} \\ &\text{ClassBodyDeclarations}_{\text{opt}} \longrightarrow \varepsilon \\ &\text{ClassBodyDeclarations}_{\text{opt}} \longrightarrow \text{ClassBodyDeclaration ClassBodyDeclarations}_{\text{opt}} \end{aligned}$$

   using the established method for immediate left recursion elimination.

Nonterminal symbol ClassBodyDeclarations describes the entire contents of a class – all declarations within a class together. If it is left-recursive (the original productions), virtually entire Java source is parsed by the embedded left LR(1) parser – everything except the package header, import declarations and the class header. If the non-left-recursive productions are used, then every declaration within a class can be parsed separately using

either LL or LR parsing, depending on (a) the structure of a particular declaration and (b) what other left-recursive productions are replaced by non-left-recursive ones.

By eliminating the left recursion (using the well-known recipe illustrated above) in productions for only

ImportDeclarations, TypeDeclarations,
InterfaceMemberDeclarations, ClassBodyDeclarations, and
BlockStatements

the percentage of the input that is parsed using the embedded LR(1) parsers drops significantly: (typically) from 98 % to 54 % (but one should be warned that by using the right set of Java sources, especially the second number can be tailored to whatever value one chooses). Instead of parsing the entire contents of an interface or a class using the embedded left LR(1) parser, individual interface members or statements within a method can be reached using the backbone LL(1) parser – and at the same time, expressions can be parsed using the left-recursive productions producing the most appropriate derivation trees. Furthermore, approx. 54 % of the input is not parsed using one embedded LR(1) parser as a single substring but rather as a number of short substrings parsed by several instances of embedded LR(1) parsers.

Reducing the length of the substrings that need to be parsed using the individual embedded LR(k) parser is crucial for efficient LLLR parsing. In case of Java 1.0, these substrings encompass mostly expressions (which are seldom longer than a few dozen symbols) and prefixes of the field or method declarations.

Finally, as the left parse of the Java program is produced during LLLR(1) parsing, semantic routines can be inserted at every position within a grammar, i.e., on both sides of every symbol found on the right side of some production. This is also true for several other parsing methods like SLL, LL(*) and ALL(*), but not for left-corner parsing and its derivatives like XLC(1) and LAXLC(1) parsing. For instance, methods like the one described in [10] can handle semantic routines in only 41.95 % of all positions within a grammar.

7 Conclusions

The main advantage of LLLR parsing (which can be used to parse any LR(k) language) over left-corner parsing is that it produces the left parse of the input string while left-corner parsing produces the mixed-strategy parse. If compared with LL(*) and ALL(*) parsing, LLLR parser runs in linear time and does not involve backtracking.

However, it must be admitted once again that there is no silver bullet in parsing. The LLLR(k) parsing works well for certain LR(k) grammars, i.e., typically for grammars where LL(k) conflicting nonterminals appear close to the bottom of the derivation trees. Otherwise, if LL(k) conflicting nonterminals are found near the top of large subtrees of the derivation tree, LLLR(k) parsing is not to be advised – one should either modify the grammar or use some other parsing method (the situation is similar to LR(k) parsing where, if an LR(k) grammar is right-recursive, the parser works but consumes a lot of stack unnecessarily).

There are basically two issues left undone in regard to LLLR parsing. First, a combination of LL and LR parsing using lookahead buffers of different sizes is worth investigating. Second, a clear theoretical formulation of LLLR parsing using the combination of the canonical LL(k_1) and the canonical LR(k_2) machines would be a challenging task.

References

1. Alfred V. Aho and Jeffrey D. Ullman. *The Theory of Parsing, Translation and Compiling*, volume Volume I: Parsing. Prentice-Hall, Englewood Cliffs, N.J., USA, 1972.
2. Alan J. Demers. Generalized left corner parsing. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages POPL'77*, pages 170–182, Los Angeles, CA, USA, 1977. ACM, ACM.
3. Bryan Ford. Parsing expression grammars: a recognition-based syntactic foundation. In *Proceedings of the 31st ACM SIGACT-SIGPLAN symposium on Principles of programming languages POPL'04*, pages 111–122, Venice, Italy, 2004. ACM, ACM.
4. R. Nigel Horspool. Recursive ascent-descent parsers. In Dieter Hammer, editor, *Compiler Compilers, Third International Workshop CC '90, Schwerin, FRG*, volume 477 of *Lecture Notes in Computer Science*, pages 1–10. Springer-Verlag, 1990.
5. R. Nigel Horspool. Recursive ascent-descent parsing. *Journal of Computer Languages*, 18(1):1–16, 1993.
6. Matthew Might and David Darais. Yacc is dead. Available online at Cornell University Library (arXiv.org:1010.5023), 2010.
7. Mark-Jan Nederhof. Generalized left corner parsing. In *Proceedings of the sixth conference on European chapter of the Association for Computational Linguistics EACL'93*, pages 305–314, Stroudsburg, PA, USA, 1993. Association for Computational Linguistics.
8. Terence Parr and Kathleen Fischer. LL(*): The foundation of the ANTLR parser generator. *ACM SIGPLAN Notices - PLDI'10*, 46(6):425–436, 2011.
9. Terence Parr, Sam Harwell, and Kathleen Fischer. Adaptive LL(*) parsing: The power of dynamic analysis. In *Proceedings of the 2014 ACM SIGPLAN International Conference on Object-Oriented Programming Systems Languages and Applications (OOPSLA'14)*, volume 579–598, Portland, OR, USA, 2014. ACM, ACM.
10. Paul Purdom and Cynthia A. Brown. Semantic routines and LR(k) parsers. *Acta Informatica*, 14(4):299–315, 1980.
11. James P. Schmeiser and David T. Barnard. Producing a top-down parse order with bottom-up parsing. *Information Processing Letters*, 54(6):323–326, 1995.
12. Elizabeth Scott and Adrian Johnstone. GLL parsing. *Electronic Notes in Theoretical Computer Science*, 253(7):177–189, 2010.
13. Elizabeth Scott, Adrian Johnstone, and Rob Economopoulos. BRNGLR: a cubic Tomita-style GLR parsing algorithm. *Acta Informatica*, 44(6):427–461, 2007.
14. Seppo Sippu and Eljas Soisalon-Soininen. *Parsing Theory, Volume I: Languages and Parsing*, volume 15 of *EATCS Monographs on Theoretical Computer Science*. Springer-Verlag, Berlin, Germany, 1988.
15. Seppo Sippu and Eljas Soisalon-Soininen. *Parsing Theory, Volume II: LR(k) and LL(k) Parsing*, volume 20 of *EATCS Monographs on Theoretical Computer Science*. Springer-Verlag, Berlin, Germany, 1990.
16. Boštjan Slivnik. The embedded left LR parser. In *Proceedings of the Federated Conference on Computer Science and Information Systems*, pages 871–878, Szczecin, Poland, 2011. IEEE Computer Society Press.
17. Boštjan Slivnik. LL conflict resolution using the embedded left LR parser. *Computer Science and Information Systems*, 9(3):1105–1124, 2012.
18. Boštjan Slivnik and Boštjan Vilfan. Producing the left parse during bottom-up parsing. *Information Processing Letters*, 96(6):220–224, 2005.
19. Boštjan Slivnik. LLLR parsing. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing SAC'13*, pages 1698–1699, Coimbra, Portugal, 2013. ACM.