LL conflict resolution using the embedded left LR parser

Boštjan Slivnik

University of Ljubljana
Faculty of Computer and Information Science
Tržaška cesta 25, 1000 Ljubljana, Slovenia
bostjan.slivnik@fri.uni-lj.si

Abstract. A method for resolving LL($k$) conflicts using small LR($k$) parsers (called embedded left LR($k$) parsers) is described. An embedded left LR($k$) parser is capable of (a) producing the prefix of the left parse of the input string and (b) stopping not on the end-of-file marker but on any string from the set of lookahead strings fixed at the parser generation time. The conditions regarding the termination of the embedded left LR($k$) parser if used within LL($k$) (and similar) parsers are defined and examined in-depth. It is proved that an LL($k$) parser augmented with a set of embedded left LR($k$) parsers can parse any deterministic context-free grammar in the same asymptotic time as LR($k$) parser. As the embedded left LR($k$) parser produces the prefix of the left parse, the LL($k$) parser augmented with embedded left LR($k$) parsers still produces the left parse and the compiler writer does not need to bother with different parsing strategies during the compiler implementation.

Keywords: embedded parsing, left LR parsing, LL conflicts.

1. Introduction

Choosing the right parsing method is an important issue in a design of a modern compiler for at least two reasons. First, the parser represents the backbone of the compiler’s front-end as the syntax-directed translation of the source program to the (intermediate) code is based upon it. And second, syntax errors cannot be scrupulously reported without the appropriate support of the parser.

As the study of available open-source compilers reveal [18], nearly all of the most popular parsing methods nowadays belong to one of the two large classes, namely LL and LR [16, 17]. LR parsing, the most popular bottom-up parsing method, is generally praised for its power while LL parsing, the principal top-down method, is credited for being simpler to implement and debug, and better for error recovery and the incorporation of semantic actions [14].

Many variations of the original LL and LR parsing methods [7, 8] have been devised since their discovery decades ago. Some methods, e.g., SLL, SLR and LALR [16, 17], focus on reducing the space complexity by producing smaller parsers (either less code or smaller parsing tables), and some tend to produce
Boštjan Slivnik

faster parsers [1]. Other methods extend the class of languages that can be parsed by the canonical LL or LR parsers. Methods like GLR and GLL are able to parse all context-free languages in cubic time (compared with the linear time achieved by the classical LL and LR methods) [21, 22, 15, 14] while LL(\*) parsers (produced by the popular ANTLR parser generator) are able to parse even some context-sensitive languages by resorting to backtracking in some cases [11]. Finally, some methods modify the behavior of the LR parsing so that by producing the left parse of the program being compiled instead of the right parse, they behave as if the top-down, e.g., LL, was used [13, 20].

The discourse on whether LL or LR parsing is more suitable either in general or in some particular case still goes on. It has been reignited lately by the online paper entitled “Yacc is dead” [10] and two issues have been made clear (again): first, parser generators are appreciated, and second, both methods, LR and LL, remain attractive [18].

To combine the advantages of both bottom-up and top-down parsing, left corner parsing was introduced [12, 3]. Basically it uses the top-down parsing and switches to bottom-up parsing to parse the left corner of each derivation subtree. However, modern variations switch to bottom-up parsing only when bottom-up parsing is needed indeed [6, 2]. Left corner parsing never gained much popularity, most likely because it produces a mixed order parse which makes incorporating semantic actions tricky.

As described, left corner parsing uses bottom-up parsing to resolve the problems arising during the top-down parsing while LL(\*) parser uses DFAs for LL conflict resolution. The former produces a tricky parse and the latter must always rescan the symbols already scanned by a DFA. In this paper an embedded left LR(k) parser which can be used within an LL(k) parser instead of a DFA, is proposed. As it produces the left parse it does not require rescanning of tokens already scanned or backtracking, and thus guarantees the linear parsing time for all LR(k) grammars.

Another method, namely packrat parsing [4], could perhaps have been used to resolve LL(k) conflicts, but there are two obstacles. First, packrat parsers are made for parsing expression grammars where the productions are ordered — the conversion of a context-free grammar to a parsing expression grammar is tricky even for the human and cannot be made by the parser generator. Second, packrat parsers do not handle left recursion well — something in particular that the embedded left LR(k) parser must handle instead of LL(k) parser.

The problem, i.e., the requirements for embedding an LR(k) parser into the LL(k) parser, is formulated in Section 2. The solution is described in Sections 3 and 4: the former contains the solution of correct termination of the embedded left LR(k) parser while the latter contains how the parser can produce the shortest prefix of the left parse as soon as possible. The evaluation of the embedded left LR(k) parser is given in Section 5 together with a brief evaluation of the new parser.

An intermediate knowledge of LL and LR parsing is presumed. The notation used in [16] and [17] is adopted except in two cases. First, a single parser
step is not described by relation \( \Rightarrow \) (as if a pushdown automaton is defined as one particular kind of a rewriting system \[16\]) but by relation \( \vdash \) among the instantaneous descriptions of a pushdown automaton \[5\]. Second, the notation 
\[ A \rightarrow \alpha \bullet \beta, x \mid S \Rightarrow^* \gamma' A v \Rightarrow^* \gamma' \alpha \beta v = \gamma \beta v \quad \text{and} \quad x \in \text{FIRST}_k(z), \]
denotes the LR\((k)\) item valid for \( \gamma \).

Finally, it is assumed that the result the parser produces is the left (right) parse of the input string, i.e., the (reversed) list of productions needed to derive the input string from the initial grammar symbol using the leftmost (rightmost) derivation.

2. On resolving LL\((k)\) conflicts

Consider an LR\((k)\) but non-LL\((k)\) grammar \( G = \langle N, T, P, S \rangle \), i.e., \( G \in \text{LR}(k) \setminus \text{LL}(k) \). If the input string \( w \in L(G) \) is derived by a derivation 
\[ S \Rightarrow^*_G \, uA \delta \Rightarrow^*_G \, uv' \delta \Rightarrow^*_G \, uv'' = uv = w, \]
the expected result of parsing it with an LL\((k)\) parser is the left parse 
\[ \pi_w = \pi_u \pi_{v'} \pi_{v''} \in P^*. \]

Since \( G \notin \text{LL}(k) \), an LL\((k)\) conflict is likely to occur and must therefore be resolved. LL\((^+)\) parsing \[11\], for instance, tries to determine the next production using a set of DFAs: if \( A \) causes an LL\((1)\) conflict in the derivation (1), a DFA for \( A \) determines the next production by scanning the first few (but sometimes more) tokens of the string \( v = v'v'' \); afterwards the LL\((^+)\) parser continues parsing by reading the entire string \( v \) again (not just the unscanned suffix of it). While LL\((^+)\) parser produces the left parse (2), it reads some tokens more than once and in some cases it must even resort to backtracking (if the DFA cannot determine the next production). Furthermore, LL\((^+)\) parsing prohibits left-recursive productions.

To produce the left parse but to avoid rescanning, backtracking and prohibiting left-recursive productions, small LR\((k)\) parsers can be used instead of DFAs. However, these small LR\((k)\) parsers must differ from the classical LR\((k)\) parsers in two regards:

1. LR\((k)\) parsers used within an LL\((k)\) parser cannot rely on the end-of-input symbol $ to terminate (unlike the standard LR\((k)\) parsers can). More precisely, if an LR\((k)\) parser is to be used for parsing the substring \( v' \) of the string \( w \) derived by the derivation (1), it must be capable of terminating with any string \( x \in \text{FIRST}_k^G(\delta\$) \) in its lookahead buffer (instead of $).

2. LR\((k)\) parsers used within an LL\((k)\) parser must produce the left parse of its input (instead of the right parse as the standard LR\((k)\) parsers do). More precisely, a standard LR\((k)\) parser for \( A \) produces the right parse of \( v' \), but if used within an LL\((k)\) parser, it should produce the left parse \( \pi_{v'} \).
If an LR\(^{(k)}\) parser fulfills both conditions, it is called the **embedded left LR\(^{(k)}\) parser**: embedded as it can be used within the **backbone LL\(^{(k)}\) parser**, and left as it produces the left parse and thus guarantees that the overall result of parsing is also the left parse.

### 3. Termination of the embedded LR\(^{(k)}\) parser

The main problem regarding the termination of the embedded LR\(^{(k)}\) parser can be explained most conveniently by the following example.

**Example 1.** Consider the grammar \(G_{ex1}\) with the start symbol \(S\) and productions

\[
S \rightarrow aAb \mid bAab \quad \text{and} \quad A \rightarrow Aa \mid a.
\]

As \(A\) is a left-recursive nonterminal, it causes the LL(1) conflict whenever \(a\) is in the lookahead buffer of the LL(1) parser.

If the input string starts with \(aa\), then after the first two steps, namely

\[
\frac{S}{\downarrow \hspace{1cm} \downarrow \hspace{1cm} \downarrow \hspace{1cm} \downarrow} \frac{aAb}{\llcorner} \frac{bAa}{\llcorner} \frac{aa}{\llcorner},
\]

the backbone LL(1) parser reaches the configuration \(\frac{bAa}{\llcorner} \frac{a}{\llcorner} \frac{aa}{\llcorner}\) (the strings on the left side and on the right side of \(\llcorner\) represent the stack contents and the remaining (yet unscanned) part of the input, respectively; the topmost stack symbol and the contents of the lookahead buffer are close to \(\llcorner\)). The configuration \(\frac{bAa}{\llcorner} \frac{a}{\llcorner} \frac{aa}{\llcorner}\) exhibits an LL(1) conflict on \(Aa\). At this point, an embedded LR(1) parser for \(A\) should be used: as \(b\) is never derived from \(A\), it can function as the end-of-input marker.

If the input string starts with \(baa\), then LL(1) parsing starts as

\[
\frac{S}{\downarrow \hspace{1cm} \downarrow \hspace{1cm} \downarrow \hspace{1cm} \downarrow} \frac{baAb}{\llcorner} \frac{baa}{\llcorner} \frac{aa}{\llcorner} \frac{bAbAAa}{\llcorner} \frac{aa}{\llcorner},
\]

The backbone LL(1) parser reaches the configuration \(\frac{bAbAAa}{\llcorner} \frac{aa}{\llcorner} \frac{bAbAAa}{\llcorner} \frac{aa}{\llcorner}\) where the embedded LR(1) parser must be used. This time the embedded LR(1) parser for \(A\) cannot be used as it cannot stop on \(a\) that follows \(A\) in the production \(S \rightarrow bAab\). More precisely, after shifting the first \(a\) on the stack and reducing it to \(A\), i.e.,

\[
\frac{[e]}{\downarrow \hspace{1cm} \downarrow \hspace{1cm} \downarrow \hspace{1cm} \downarrow} \frac{[e]Aa}{\llcorner} \frac{[e]Aa}{\llcorner} \frac{Aa}{\llcorner} \frac{aa}{\llcorner},
\]

the embedded LR(1) parser faces the second \(a\) in its lookahead buffer, but it cannot determine whether it should be shifted or not. If the entire input is \(baab\), the embedded LR(1) parser should terminate and handle the control back to the backbone LL(1) parser, otherwise it should continue by shifting and reducing using \(A \rightarrow Aa\). Therefore, the embedded LR(1) parser for \(Aa\), i.e., one that can terminate on \(b\) for the same reason as above, must be used instead of the one for \(A\).

(Modifying the problem to any \(k\) is left as an exercise.)

Two conclusions follow from Example 1:
1. The embedded LR\((k)\) parser must sometimes parse substrings derived from a sentential form starting with the LL\((k)\)-conflicting nonterminal instead of from that nonterminal only. More precisely, if the first part of the derivation (1) is rewritten as

\[
S \Rightarrow \gamma_{\ell m}^u \ u' B\delta' \Rightarrow \gamma_{\ell m}^u \ u' \beta_1 A \beta_2 \delta' \Rightarrow \gamma_{\ell m}^u u' u'' A \beta_2 \delta' = u A\delta \ ,
\]

the parser for \(A\beta_2'\), where \(\beta_2 = \beta_2' \beta_2''\) in \(B \longrightarrow \beta_1 A \beta_2\), might be needed instead of the parser for \(A\). In Example 1 a parser for \(Aa\) is needed in production \(S \longrightarrow bAab\) instead of a parser for \(A\).

2. The right context of the left sentential form the embedded LR\((k)\) parser is made for, is important. More precisely, the right context is the prefix of the string that comes after the string derived from the sentential form the embedded parser is made for, i.e., in the derivation (3) the termination of the embedded LR\((k)\) parser for \(A\beta_2'\) depends on the contents of the set \(\text{FIRST}_k^G(\beta_2' \delta')\).

Hence, in general an embedded LR\((k)\) parser for \(A\beta_2'\) capable of termination on any string from \(\text{FIRST}_k^G(\beta_2' \delta')\) is needed.

The easiest way to resolve the right context of the embedded LR\((k)\) parser is to transform grammar \(G = \langle N, T, P, S \rangle\) into grammar \(\bar{G} = \langle \bar{N}, T, \bar{P}, \bar{S} \rangle\) by applying the transformation of an LL\((k)\) grammar to an SLL\((k)\) grammar [17]: in the transformed grammar \(\bar{G}\) each nonterminal occurs in exactly one right context. More precisely, the start symbol becomes \(\bar{S} = \langle S, \{\varepsilon\} \rangle\) and the set \(\bar{N}\) of nonterminals is defined as

\[
\bar{N} = \{\langle A, \mathcal{F}_A \rangle; \ S \Rightarrow \bar{\imath}_m^u u A\delta \land \mathcal{F}_A = \text{FIRST}_k^G(\delta')\} \ .
\]

For any nonterminal \(\langle A, \mathcal{F}_A \rangle\) the new set \(\bar{P}\) of productions includes productions

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

where, for any \(i = 1, 2, \ldots, n\),

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

provided that \(A \longrightarrow X_1X_2 \ldots X_n \in P\). (This transformation does not introduce any new LL\((k)\) conflicts; in fact, if \(k > 1\), it even reduces the number of LL\((k)\) conflicts for some non-SLL\((k)\) grammars [17].)

**Example 2.** If the grammar \(G_{\text{ex1}}\) is transformed, a grammar \(\bar{G}_{\text{ex1}}\)

\[
\langle S, \{\varepsilon\} \rangle \longrightarrow a\langle A, \{b\}\rangle b \mid b(A, \{a\})ab \\
\langle A, \{a\} \rangle \longrightarrow \langle A, \{a\}\rangle a \mid a \\
\langle A, \{b\} \rangle \longrightarrow \langle A, \{a\}\rangle a \mid a
\]

is obtained. Two embedded LR\((1)\) parsers are needed: \(\langle A, \{b\}\rangle\) and \(\langle Aa, \{b\}\rangle\):
1. The parser for \( \langle A, \{b\} \rangle \) results from production \( \langle S, \{\varepsilon\} \rangle \rightarrow a \langle A, \{b\} \rangle b \): the parser’s right context is \( \{b\} = \text{FIRST}^G_k(b) \); \( b \) follows \( \langle A, \{b\} \rangle \) in production \( \langle S, \{\varepsilon\} \rangle \rightarrow a \langle A, \{b\} \rangle b \) and \( \{\varepsilon\} \) (from \( \langle S, \{\varepsilon\} \rangle \)) determines the right context of the entire production \( \langle S, \{\varepsilon\} \rangle \rightarrow a \langle A, \{b\} \rangle b \).

2. The parser for \( \langle Aa, \{b\} \rangle \) results from production \( \langle S, \{\varepsilon\} \rangle \rightarrow b \langle A, \{a\} \rangle ab \), again with the right context \( \{b\} = \text{FIRST}^G_k(b) \); \( b \) follows the sentential form \( \langle A, \{a\} \rangle a \) in production \( \langle S, \{\varepsilon\} \rangle \rightarrow b \langle A, \{a\} \rangle ab \) and \( \{\varepsilon\} \) (from \( \langle S, \{\varepsilon\} \rangle \)) determines the right context of the entire production.

After the LR(1) parsers are embedded, productions for \( \langle A, \{a\} \rangle \) and \( \langle A, \{a\} \rangle \) are eliminated as they are no longer needed — the embedded LR(k) parsers are based on the original grammar \( G_{\text{ex1}} \).

To resolve conflicts during LL(k) parsing based on the grammar \( G \), every production

\[
\langle B, \mathcal{F}_B \rangle \rightarrow \beta_1 \langle A, \mathcal{F}_A \rangle \beta_2 \in \bar{P}
\]

with an LL(k)-conflicting nonterminal \( \langle A, \mathcal{F}_A \rangle \) is supposed to be replaced with a production

\[
\langle B, \mathcal{F}_B \rangle \rightarrow \beta_1 \langle A\beta_2', \mathcal{F}_{A\beta_2'} \rangle \beta_2''
\]

where \( \beta_2 = \beta_2' \beta_2'' \) and \( \mathcal{F}_{A\beta_2'} = \text{FIRST}^G_k(\beta_2', \mathcal{F}_B) \). The new symbol \( \langle A\beta_2', \mathcal{F}_{A\beta_2'} \rangle \notin \bar{N} \) acts as a trigger for the embedded LR(k) parser for \( A\beta_2' \) capable of termination on any string from \( \mathcal{F}_{A\beta_2'} \).

As the amount of LR parsing is to be minimal, \( \beta_2' \) should be as short as possible, i.e., \( \varepsilon \) in the best case. If, on the other hand, not even \( \beta_2' = \varepsilon \) and \( \beta_2'' = \varepsilon \) suffices for the safe termination of the embedded LR(k) parser, \( \langle B, \mathcal{F}_B \rangle \) must be declared a conflicting nonterminal.

Finally, if marker \( \langle \beta, \mathcal{F} \rangle \) is introduced into the grammar \( G = \langle \bar{N}, T, \bar{P}, S \rangle \) (based on \( G = \langle N, T, P, S \rangle \)), an embedded LR(k) parser for \( \beta \) that terminates on any lookahead string \( x \in \mathcal{F} \), is needed. The easiest way to achieve this is to build the LR(k) parser for the embedded grammar

\[
\hat{G}_{\beta, \mathcal{F}} = \langle \bar{N}, T, \bar{P}, S \rangle
\]

where \( \bar{N} = N \cup \{S_1, S_2\} \) for \( S_1, S_2 \notin N \) and

\[
\bar{P} = P \cup \{S_1 \rightarrow S_2x, S_2 \rightarrow \beta : x \in \mathcal{F}\}.
\]

The trick is obvious: the embedded LR(k) parser for \( \hat{G}_{\beta, \mathcal{F}} \) must accept its input no later than when the reduction on \( S_2 \rightarrow \beta \) is due. In other words, if the reduce on \( S_2 \rightarrow \beta \) is replaced with the accept action, the parser never pushes any symbol of any string \( x \in \mathcal{F} \) onto the stack. If the reduce on \( S_2 \rightarrow \beta \) cannot be determined (because of the LR(k) conflict), the embedded LR(k) parser for \( \langle \beta, \mathcal{F} \rangle \) cannot be used.

Determining whether the embedded LR(k) parser does not contain any LR(k) conflicts is time consuming if a brute-force approach of using testing whether \( \hat{G}_{\beta, \mathcal{F}} \in \text{LR}(k) \) is used. However, the method based on the following theorem significantly reduces the time complexity of testing the embedded LR(k) parser for LR(k) conflicts. 

1110 ComSIS Vol. 9, No. 3, Special Issue, September 2012
Theorem 1. Let $G = (N, T, P, S)$ be an LR($k$) grammar with the derivation $\ldots$

Grammar $\hat{G} = (\hat{N}, T, \hat{P}, S_1)$ where

$\hat{N} = N \cup \{S_1, S_2\}$ for $S_1, S_2 \notin N$ and

$\hat{P} = P \cup \{S_1 \rightarrow S_2x, S_2 \rightarrow \beta_2' ; x \in FIRST^G_k(\beta_2' \delta)\} \ldots$

is not an LR($k$) grammar if and only if

- either $\beta_2'' = \varepsilon$ and $[S \rightarrow \beta_2' \bullet, x'], [B \rightarrow \beta_2' \bullet, x'] \in [\beta_2'']_\hat{G}$
- or $\beta_2'' \neq \varepsilon$ and $[S_2 \rightarrow \beta_2' \bullet, x'], [A \rightarrow \alpha \bullet \alpha', y'] \in [\beta_2'']_\hat{G}$

where $\alpha' \neq \varepsilon$ and $x' \in FIRST^G_k(\alpha'y')$.

Proof. The idea the proof is based on is rather simple. Because of the leftmost derivation specified by this theorem, there is a state of the LR($k$) machine for $G$ that includes all $[B \rightarrow \beta_1 \bullet \beta_2' \beta_2'' y]$ where $y \in FIRST^G_k(\delta)$. This state corresponds to the initial state of the LR($k$) machine for $\hat{G}$. By careful examination of all possibilities only those possibilities permitting LR($k$) conflicts in $G$ are singled out. The formal proof follows.

First, the structure of the grammar $\hat{G}$ implies that items

$$[S_1 \rightarrow \bullet S_2 x, \$] \text{ and } [S_2 \rightarrow \bullet \beta_2', x'],$$

where $x \in FIRST^G_k(\beta_2' \delta)$ and $x' \in FIRST^G_k(\beta_2'' \delta)$, appear only in the initial state $[\$]_{\hat{G}}$ of the canonical LR($k$) parser for the ($\$-$augmented version of) grammar $\hat{G}$. Likewise, items

$$[S_1 \rightarrow \psi_1 \bullet \psi_2, \$] \text{ and } [S_2 \rightarrow \psi_1 \bullet \psi_2, x'],$$

where $x' \in FIRST^G_k(\beta_2'' \delta)$, appear only in $[\$\$ \psi_1]_{\hat{G}}$. Furthermore, states $[\$S_2 \psi]_{\hat{G}}$, for various $\psi$, contain only items based on productions $S_1 \rightarrow S_2x$.

Second, as $G \in LR(k)$ and is thus unambiguous, the leftmost derivation

$$S \Rightarrow^{*}_{G,lm} uB\delta \Rightarrow^{*}_{G,lm} w$$

implies the existence of the rightmost derivation

$$S \Rightarrow^{*}_{G,rm} \gamma B\nu'' \Rightarrow^{*}_{G,rm} \gamma \beta_1 \beta_2' \beta_2'' \nu'' \Rightarrow^{*}_{G,rm} w$$

Moreover, if $\delta \Rightarrow^{*}_{G,rm} \nu''$, then the viable prefix $\gamma$ depends only on the left sentential form $uB\delta$, i.e., it is unique for all $w$. Therefore,

$$\{[B \rightarrow \beta_1 \bullet \beta_2' \beta_2'' ; y' \in FIRST^G_k(\delta \$)] \subseteq [\$\gamma\beta_1]_{G}$$

where $[\$\gamma\beta_1]_{G}$ is the state $[\$\gamma\beta_1]_{G}$ of the canonical LR($k$) machine for the ($\$-$augmented version of) grammar $G$.

Consider any two items $i_1$ and $i_2$ (except items based on the production $S' \rightarrow \$S_1 \$ as these items are never involved in an LR($k$) conflict) in any state $[\$i]_{G}$ of the canonical LR($k$) machine for $G$, i.e., $i_1, i_2 \in [\$i]_{G}$:
1. If \( i_1 \) and \( i_2 \) are based on productions in \( P \), then \( i_1, i_2 \in [S\gamma\beta_1\gamma]_G \) and there is no LR(\( k \)) conflict between \( i_1 \) and \( i_2 \) since \( G \in LR(k) \).

2. If \( i_1 \) and \( i_2 \) are based on productions in \( P \setminus P \), the following three cases must be considered:
   
   (a) \( i_1 = [S_1 \rightarrow \gamma \bullet \alpha, \$] \) and \( i_2 = [S_1 \rightarrow \gamma \bullet \alpha', \$] \):
      
      If \( \gamma = \varepsilon \), then \( i_1 \) and \( i_2 \) imply no actions because \( \alpha \) and \( \alpha' \) start with \( S_2 \).
      Otherwise they imply no reduce action (if \( \alpha \neq \varepsilon \) and \( \alpha' \neq \varepsilon \)), imply the same action (as \( i_1 = i_2 \) if \( \alpha = \varepsilon \) and \( \alpha' = \varepsilon \)), or imply the reduce on \( \$ \) and shift on non-\( \$ \) (if \( \alpha = \varepsilon \) and \( \alpha' \neq \varepsilon \); or vice versa).
   
   (b) \( i_1 = [S_1 \rightarrow \gamma \bullet \alpha, \$] \) and \( i_2 = [S_2 \rightarrow \gamma \bullet \alpha', y'] \) (or vice-versa):
      
      \( i_1 \) implies no action if \( \gamma = \varepsilon \) as \( \alpha \) starts with \( S_2 \). The other case, if \( \gamma \neq \varepsilon \) is impossible: \( \gamma \) starts with \( S_2 \) in \( i_1 \) and does not start with \( S_2 \) in \( i_2 \).
   
   (c) \( i_1 = [S_2 \rightarrow \gamma \bullet \alpha, y] \) and \( i_2 = [S_2 \rightarrow \gamma \bullet \alpha', y'] \):
      
      \( \alpha = \alpha' \) and both items imply either the same action or imply no action.

3. If \( i_1 \) is based on a production in \( P \setminus P \) and \( i_2 \) is based on a production in \( P \) (or vice versa), the following two cases must be considered:
   
   (a) \( i_1 = [S_1 \rightarrow \gamma_1 \gamma_2 \bullet \alpha, \$] \) and \( i_2 = [A \rightarrow \gamma_2 \bullet \alpha', y'] \):
      
      If \( \gamma_1 \gamma_2 = \varepsilon \), then \( i_1 \) implies no action as \( \alpha \) starts with \( S_2 \). The other case, if \( \gamma_1 \gamma_2 \neq \varepsilon \), is impossible: \( i_1 \in \{ \gamma_1 \gamma_2 \}_G \) while \( i_2 \notin \{ \gamma_1 \gamma_2 \}_G \).
   
   (b) \( i_1 = [S_2 \rightarrow \gamma_1 \gamma_2 \bullet \alpha, y] \) and \( i_2 = [A \rightarrow \gamma_2 \bullet \alpha', y'] \):
      
      As \( i_1, i_2 \in \{ \gamma_1 \gamma_2 \}_G \), so does
      
      \[ [B \rightarrow \beta_1 \gamma_1 \gamma_2 \bullet \alpha \beta_2', y'], [A \rightarrow \gamma_2 \bullet \alpha', y'] \in \{ \gamma \beta_1 \gamma_1 \gamma_2 \}_G \]
      
      where \( y \in \text{FIRST}^G_{\kappa} (\beta_2' y') \).
      
      – If \( \alpha \neq \varepsilon \) and \( \alpha' \neq \varepsilon \), then neither \( i_1 \) nor \( i_2 \) implies a reduce action.
      
      – If \( \alpha \neq \varepsilon \) and \( \alpha' = \varepsilon \), then

      \[ [S_2 \rightarrow \gamma_1 \gamma_2 \bullet \alpha, y], [A \rightarrow \gamma_2 \bullet \alpha', y'] \in \{ \gamma \gamma_1 \gamma_2 \}_G \]
      
      exhibit a shift-reduce conflict if and only if \( y' \in \text{FIRST}^G_{\kappa} (\alpha y) \). But then items

      \[ [B \rightarrow \beta_1 \gamma_1 \gamma_2 \bullet \alpha \beta_2', y'], [A \rightarrow \gamma_2 \bullet \alpha', y'] \in \{ \gamma \beta_1 \gamma_1 \gamma_2 \}_G \]
      
      exhibit a conflict. This is not possible as \( G \in LR(k) \) and therefore items \( i_1 \) and \( i_2 \) do not exhibit a conflict in \( G \).
      
      – If \( \alpha = \varepsilon \) and \( \alpha' \neq \varepsilon \), then

      \[ [S_2 \rightarrow \gamma_1 \gamma_2 \bullet \alpha', y], [A \rightarrow \gamma_2 \bullet \alpha', y'] \in \{ \gamma \gamma_1 \gamma_2 \}_G \]
      
      exhibit a shift-reduce conflict if \( y \in \text{FIRST}^G_{\kappa} (\alpha' y') \). But

      \[ [B \rightarrow \beta_1 \gamma_1 \gamma_2 \bullet \beta_2', y'], [A \rightarrow \gamma_2 \bullet \alpha', y'] \in \{ \gamma \beta_1 \gamma_1 \gamma_2 \}_G \]
      
      and the only possibility of a shift-reduce conflict in \( \{ \gamma \beta_1 \gamma_1 \gamma_2 \}_G \) without the conflict in \( \{ \gamma \beta_1 \gamma_1 \gamma_2 \}_G \) is that \( \beta_2' = \gamma_1 \gamma_2 \) and \( \beta_2' \neq \varepsilon \).
If \( \alpha = \epsilon \) and \( \alpha' = \epsilon \), then

\[
[S_2 \rightarrow \hat{\gamma}_1 \hat{\gamma}_2 \bullet \gamma, A \rightarrow \hat{\gamma}_2 \bullet \gamma] \in [\$\hat{\gamma}_1 \hat{\gamma}_2]_G
\]

exhibit a reduce-reduce conflict if \( y = y' \). But

\[
[B \rightarrow \beta_1 \hat{\gamma}_1 \hat{\gamma}_2 \bullet \beta_2'' \gamma', A \rightarrow \hat{\gamma}_2 \bullet \gamma] \in [\$\beta_1 \hat{\gamma}_1 \hat{\gamma}_2]_G
\]

and the only possibility of a reduce-reduce conflict in \([\$\hat{\gamma}_1 \hat{\gamma}_2]_G\) without the conflict in \([\$\beta_1 \hat{\gamma}_1 \hat{\gamma}_2]_G\) is that \( \beta_1 \hat{\gamma}_1 = \epsilon, \beta_2'' = \hat{\gamma}_2 \) and \( \beta_2'' = \epsilon \).

Finally, proving the theorem in the opposite direction is trivial — if the canonical LR\((k)\) machine for the grammar \( \hat{G} \) contains an LR\((k)\) conflict, then clearly \( \hat{G} \not\in \text{LR}(k) \).

**Corollary 1.** Let \( G = \langle N, T, P, S \rangle \) be an LR\((k)\) grammar with the derivation

\[
S \Rightarrow^{*}_{G,\text{lm}} uB\delta \Rightarrow^{*}_{G,\text{lm}} u\beta_1\beta_2'\delta .
\]

Grammar \( \hat{G} = \langle \hat{N}, T, \hat{P}, S_1 \rangle \) where \( \hat{N} = N \cup \{ S_1, S_2 \} \) for \( S_1, S_2 \notin N \) and \( \hat{P} = P \cup \{ S_1 \rightarrow S_2 x, S_2 \rightarrow \beta_2'' x; x \in \text{FIRST}_k^G(\delta) \} \) is not an LR\((k)\) grammar if and only if

\[
[S_2 \rightarrow \bullet \beta_2', x'] \text{ desc}^* [B \rightarrow \bullet \beta_2'', x']
\]

where \( B \neq S_2 \) and \( x' \in \text{FIRST}_k^G(\delta) \) [18].

To conclude this section, Algorithm 1 is given. It is based on Theorem 1 and is (to be) used for computing the shortest prefix of \( \langle A, f_A \rangle \beta_2 \) in production \( \langle B, f_B \rangle \rightarrow \beta_1 \langle A, f_A \rangle \beta_2 \) where the embedded LR\((k)\) parser must be employed to resolve the LL\((k)\) conflict caused by \( \langle A, f_A \rangle \). Once Theorem 1 is digested, the algorithm comes out relatively simple: it just checks both conditions exposed by Theorem 1, one for \( \beta_2'' = \epsilon \) and the other for \( \beta_2'' \neq \epsilon \).

### 4. Terminating while producing the left parse

As mentioned in Section 2, the embedded LR\((k)\) parser must produce the left parse instead of the right parse. To achieve this, the left LR\((k)\) parser [20] (based on the Schmeiser-Barnard LR\((k)\) parser [13]) is taken as the starting point.

Consider an LR\((k)\) grammar \( G = \langle N, T, P, S \rangle \) and the input string \( w = uv \) derived by the rightmost derivation

\[
S \Rightarrow^{*}_{G,\text{rm}} \gamma v \Rightarrow^{*}_{G,\text{rm}} \gamma u v . \quad (5)
\]

After reading the prefix \( u \), the canonical LR\((k)\) parser for grammar \( G \) reaches the configuration

\[
\$[\$][\$X_1][\$X_1 X_2] \ldots [\$X_1 X_2 \ldots X_n] \text{ | v} \$
\]

\( (6) \)
Consider the embedded grammar only at the very end of parsing. Note that if this method is used, the first production of the left parse is produced 1114 ComSIS Vol. 9, No. 3, Special Issue, September 2012 To accumulate left parses on the stack, the actions are modified as follows: LR(\text{denotes the state of the canonical \text{[$\$]} by string obtained by adding the new start symbol} S \text{and thus reaches the configuration} on the stack with the left parse of the substring derived from that nonterminal} \pi \text{instead.}$.  

\[ \text{Algorithm 1: Computing the shortest prefix} \beta'' \text{of the sentential form} \beta = \beta' \beta'' \text{so that the embedded LR}(k) \text{grammar} G_{\beta', \beta''} \in \text{LR}(k) \text{where} J' = \text{FIRST}_k^G(\beta' J). \]

**INPUT:** The sentential form \( \beta = X_1 X_2 \ldots X_n \) and the right context \( J \).

**OUTPUT:** The prefix \( \beta' \) (or \( \bot \) if the prefix does not exist).

\begin{center}
\begin{tabular}{l}
1: \textbf{for} \( i \leftarrow 1 \ldots (n - 1) \) \textbf{do} \\
2: \( \beta' = X_1 X_2 \ldots X_i \) and \( \beta'' = X_{i+1} X_{i+2} \ldots X_n \) \\
3: \textbf{if} \( \neg(\exists[A \rightarrow \alpha \bullet \alpha', y'] \in [\mathcal{S}\beta']_G : \text{FIRST}_k^G(\alpha' y') \cap \text{FIRST}_k^G(\beta'' J\$) \neq \emptyset) \) \textbf{then} \\
4: \textbf{return} \( \beta'' \) \\
5: \textbf{end if} \\
6: \textbf{end for} \\
7: \( \beta' = X_1 X_2 \ldots X_n \) and \( \beta'' = \varepsilon \) \\
8: \textbf{if} \( \neg(\exists[A \rightarrow \alpha \bullet x'] \in [\mathcal{S}\beta']_G : x' \cap \text{FIRST}_k^G(\mathcal{J}\$) \neq \emptyset) \) \textbf{then} \\
9: \textbf{return} \( \beta'' \) \\
10: \textbf{end if} \\
11: \textbf{return} \( \bot \) \\
\end{tabular}
\end{center}

where \( X_1 X_2 \ldots X_n = \gamma, [\mathcal{S}X_1 X_2 \ldots X_n] \) is the current parser state and \( x = k: \text{v}\$ \) is the contents of the lookahead buffer. ([\mathcal{S}X_1 X_2 \ldots X_j], for \( j = 0, 1, \ldots, n \), denotes the state of the canonical LR(\( k \)) \text{machine} \mathcal{M}_G \text{reachable from the state} \mathcal{S} by string \( X_1 X_2 \ldots X_j \), where \( \mathcal{M}_G \) is based on the \$-augmented grammar \( G' \) obtained by adding the new start symbol \( S' \) with production \( S' \rightarrow \$\mathcal{S}\$ \text{to} \mathcal{G} \).

The Schmeiser-Barnard LR(\( k \)) parser augments each nonterminal pushed on the stack with the left parse of the substring derived from that nonterminal and thus reaches the configuration

\[ [\mathcal{S}] \varepsilon ([\mathcal{S}X_1]; \pi(X_1)) ([\mathcal{S}X_1 X_2]; \pi(X_2)) \ldots ([\mathcal{S}X_1 X_2 \ldots X_n]; \pi(X_n))] \varepsilon \$ (7)\]

instead. \( \pi(X_j) \) denotes the left parse of the substring derived from \( X_j \) and thus

\[ X_1 X_2 \ldots X_n \Rightarrow ^{\pi(X_1) \pi(X_2) \ldots \pi(X_n)}_{G, \text{lm}} u . \]

To accumulate left parses on the stack, the actions are modified as follows:

- If the parser performs the shift action, no production is pushed on the stack, i.e., the terminal pushed is augmented with the empty left parse \( \varepsilon \).
- If the parser performs the reduce action, the left parses accumulated in states removed from the stack are concatenated, and prefixed by the production the reduction is made on. The resulting left parse is pushed on the stack together with the new nonterminal.

Note that if this method is used, the first production of the left parse is produced only at the very end of parsing.

**Example 3.** Consider the embedded grammar \( G_{\text{ex3}} \) with productions

\[ S_1 \rightarrow S_2 c , \quad S_2 \rightarrow A , \quad A \rightarrow a a | a B | b B a a , \quad B \rightarrow B b | \varepsilon . \]
Table 1. Parsing the string $bbaac \in L(G_{ex3})$ using the Schmeiser-Barnard LR(1) parser.

<table>
<thead>
<tr>
<th>STACK</th>
<th>INPUT</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$\langle [S]; \varepsilon \rangle \ bbaac$</td>
</tr>
<tr>
<td>2</td>
<td>$\langle [S]; \varepsilon \rangle \langle [bB]; \varepsilon \rangle \ bbaac$</td>
</tr>
<tr>
<td>3</td>
<td>$\langle [S]; \varepsilon \rangle \langle [bB]; \varepsilon \rangle \langle [bB]; \pi_1 = B \rightarrow \varepsilon \rangle \ bbaac$</td>
</tr>
<tr>
<td>4</td>
<td>$\langle [S]; \varepsilon \rangle \langle [bB]; \pi_2 = B \rightarrow Bb; \pi_1 \rangle \langle [bB]; \varepsilon \rangle \ bbaac$</td>
</tr>
<tr>
<td>5</td>
<td>$\langle [S]; \varepsilon \rangle \langle [bB]; \pi_3 = B \rightarrow Bb; \pi_2 \rangle \langle [bB]; \varepsilon \rangle \ baa$</td>
</tr>
<tr>
<td>6</td>
<td>$\langle [S]; \varepsilon \rangle \langle [S_2]; \pi_4 = A \rightarrow bBaa; \pi_3 \rangle \langle [S_2]; \varepsilon \rangle \ baa$</td>
</tr>
<tr>
<td>7</td>
<td>$\langle [S]; \varepsilon \rangle \langle [S_2]; \pi_5 = S_2 \rightarrow A; \pi_4 \rangle \langle [S_2]; \varepsilon \rangle \ baa$</td>
</tr>
<tr>
<td>8</td>
<td>$\langle [S]; \varepsilon \rangle \langle [S_2]; \pi_6 = S_2 \rightarrow A; \pi_5 \rangle \langle [S_2]; \varepsilon \rangle \ baa$</td>
</tr>
<tr>
<td>9</td>
<td>$\langle [S]; \varepsilon \rangle \langle [S_1]; \pi_7 = S_1 \rightarrow S_2; \pi_6 \rangle \langle [S_1]; \varepsilon \rangle \ baa$</td>
</tr>
</tbody>
</table>

where $\pi_7 = S_1 \rightarrow S_2c; S_2 \rightarrow A; A \rightarrow bBaa; B \rightarrow Bb; B \rightarrow Bb \rightarrow \varepsilon$

Parsing of the input string $bbaac$ using the Schmeiser-Barnard LR(1) parser is shown in Table 1. Note that the first production of the resulting left parse, namely $S_1 \rightarrow S_2c$, is not known until the end of parsing.

The left LR($k$) parser [20] is able to compute the prefix of the left parse of the substring corresponding to the prefix of the input string read so far during parsing (although this is not possible in every parser configuration). In other words, if corresponding to the derivation (5) the input string $w = uv$ is derived by the leftmost derivation

$$S \xrightarrow{\pi(u)} \pi_{im} \ u \delta \xrightarrow{\pi_{im}} uv ,$$

then the left LR($k$) parser can compute the left parse $\pi(u)$ in configuration (7) provided that certain conditions specified later on are met. As this part of the left LR($k$) parser is modified, it deserves more attention.

By theory [17], configurations (6) and (7) imply that machine $M_G$ contains at least one sequence of valid $k$-items

$$[A_0 \rightarrow \alpha_0 A_1 \beta_0, x_0] \ldots [A_0 \rightarrow \alpha_0 A_1 \beta_0, x_0] .$$

$$[A_1 \rightarrow \alpha_1 A_2 \beta_1, x_1] \ldots [A_1 \rightarrow \alpha_1 A_2 \beta_1, x_1] .$$

$$\vdots$$

$$[A_k \rightarrow \alpha_k A_{k+1} \beta_k, x_k] \ldots [A_k \rightarrow \alpha_k A_{k+1} \beta_k, x_k]$$

where $[A_0 \rightarrow \alpha_0 A_1 \beta_0, x_0] = [S', \rightarrow \bullet S S, \varepsilon], \gamma = \alpha_0 \alpha_1 \ldots \alpha_k$ and $k; vS \in$ FIRST$_k$($A_{k+1}; \beta_{k+1}; x_k$) (and $A_{k+1} = \varepsilon$); the horizontal dots denote repetitive application of operation passes (or GOTO) while the vertical dots denote the application of desc (or CLOSURE).
Boštjan Slivnik

Sequence (9) induces the *induced* central derivation

$$S' = A_0 \Rightarrow G \alpha_0 A_1 \beta_0 \Rightarrow G \alpha_0 \alpha_1 A_2 \beta_1 \beta_0 \Rightarrow G \cdots \Rightarrow G \Rightarrow G \alpha_0 \alpha_1 \cdots \alpha_{\ell} A_{\ell+1} \beta_{\ell} \beta_{\ell-1} \cdots \beta_0$$

the name “central” becomes obvious if the corresponding derivation tree presented in Figure 1(a) is observed.

![Derivation tree of the induced central derivation](image)

(a) The derivation tree of the induced central derivation.

![Derivation tree of the induced leftmost derivation](image)

(b) The derivation tree of the induced leftmost derivation (the left parses $\pi_{\alpha_j}$ must be provided).

Fig. 1. The derivation trees corresponding to various kinds of induced derivations; remember that $A_{\ell+1} = \varepsilon$ in all three cases.

However, if the left parses $\pi(\alpha_0), \pi(\alpha_1), \ldots, \pi(\alpha_{\ell})$, where $\alpha_j \Rightarrow \pi(\alpha_j)$ for $j = 0, 1, \ldots, \ell$, are provided, sequence (9) induces the *induced* leftmost derivation

$$S' = A_0 \Rightarrow G_{\text{lm}} u_0 A_1 \beta_0 \Rightarrow \pi(\alpha_0) u_0 A_1 \beta_0$$

$$\Rightarrow G_{\text{lm}} u_0 \alpha_1 A_2 \beta_1 \beta_0 \Rightarrow \pi(\alpha_1) u_0 u_1 A_2 \beta_1 \beta_0$$

$$\cdots$$

$$\Rightarrow G_{\text{lm}} u_0 u_1 \cdots u_{\ell-1} \alpha_{\ell} A_{\ell+1} \beta_{\ell} \beta_{\ell-1} \cdots \beta_0 \Rightarrow \pi(\alpha_{\ell}) u_0 u_1 \cdots u_{\ell} A_{\ell+1} \beta_{\ell} \beta_{\ell-1} \cdots \beta_0$$

where $u = u_0 u_1 \cdots u_{\ell}$ and $k; v \in \text{FIRST}_k^G(\beta_{\ell} \beta_{\ell-1} \cdots \beta_0 \$). The corresponding derivation tree is shown in Figure 1(b) and the left parse of the induced leftmost
LL conflict resolution using the embedded left LR parser

derivation is therefore

\[
\pi(u) = A_0 \rightarrow \alpha_0 A_1 \beta_0 \cdot \pi(\alpha_0) \cdot A_1 \rightarrow \alpha_1 A_2 \beta_1 \cdot \pi(\alpha_1) \cdot \ldots \cdot A_\ell \rightarrow \alpha_\ell A_{\ell+1} \beta_\ell \cdot \pi(\alpha_\ell) \cdot
\]

(10)

(Likewise, if the right parses \(\pi(\beta_1), \pi(\beta_2), \ldots, \pi(\beta_\ell)\) are known, then sequence (9) induces the (induced) rightmost derivation.)

Subparses \(\pi(\alpha_i)\) of the left parse (10) are available on the parser stack because \(\alpha_0 \alpha_1 \ldots \alpha_\ell = \gamma = X_1 X_2 \ldots X_\ell\), but productions \(A_j \rightarrow \alpha_j A_{j+1} \beta_j\) are not. However, if sequence (9) is known, the missing productions and in fact the entire prefix of the left parse can be computed [20]. Starting with \(\pi = \varepsilon\) and \(i = [A_\ell \rightarrow \alpha_\ell \cdot A_{\ell+1} \beta_\ell, x_\ell]\), the stack is traversed downwards:

- If \(i = [A \rightarrow \bullet \beta, x]\), then (a) \(i\) expands the nonterminal \(A\) by production \(A \rightarrow \beta\) and (b) \(i'\), the item that precedes \(i\) in sequence (9), is in the same state. Hence, let \(\pi := A \rightarrow \beta \cdot \pi\) and \(i := i'\).
- If \(i = [A \rightarrow \alpha X \bullet \beta, x] \in [\$;X]\) for some \(\gamma\), then (a) the left parse \(\pi(X)\) is available on the stack and (b) \(i'\) is in the state \([\$;\gamma]\) (which is found beneath \([\$;X]\)). Hence, let \(\pi := \pi(X) \cdot \pi\) and \(i := i'\); furthermore, proceed one step downwards along the stack, i.e., to the state \([\$;\gamma]\).

The downward traversal stops when the item \([S_2 \rightarrow \bullet \beta, x] \in [\$]\), for some \(\beta \in (N \cup T)^*\) and \(x \in (T \cup \{\$\})^k\), is reached (the production \(S_2 \rightarrow \beta\) is not added to the resulting left parse).

This method can be upgraded to compute the prefix of the left parse and the viable suffix \(\delta^R\) in derivation (8) as well since \(\delta = A_\ell+1 \beta_\ell \beta_{\ell-1} \ldots \beta_0\) — see Figure 1(b). Hence, start with \(\delta = A_\ell+1 \beta_\ell\) and whenever \(i = [A \rightarrow \bullet \beta, x]\), let \(\delta := \delta \cdot \beta'\) where \(i' = [A' \rightarrow \alpha' \cdot A' \beta', x']\) is the item preceding \(i\) in sequence (9).

**Example 4.** Consider again the grammar \(G_{ex3}\) and the input string \(bbbaac\in L(G_{ex3})\) from Example 3. After the prefix \(bbba\) of the input string has been read, the parser reaches the configuration shown in the 7th line of Table 1. But as illustrated in Figure 2, there is only one item active for the current lookahead string \(a\) in state \([bbBa]\), namely \([A \rightarrow bBa \bullet a, \$]\). Furthermore, there exist exactly one sequence of LR(1) items starting with \([S' \rightarrow \bullet S_1 S_1, \varepsilon] \in [\varepsilon]\) and ending with \([A \rightarrow bBa \bullet a, \$] \in [bbBa]:\)

\[
[S' \rightarrow \bullet S_1 S_1, \varepsilon] \cdot [S' \rightarrow S_1 S_1, \varepsilon] \cdot [S_1 \rightarrow \bullet S_2 c, \$] \cdot [S_2 \rightarrow \bullet A, \varepsilon] \cdot [A \rightarrow bBa, \$] \cdot [A \rightarrow bBa, \$] \cdot \ldots \cdot [A \rightarrow bBa, \$] \cdot [A \rightarrow bBa, \$]
\]

Hence, the prefix of the left parse and the corresponding viable suffix can be computed as shown in Figure 3 using the method outlined above.

In general, cases where exactly one sequence (9) exists (as in Example 4) are extremely rare, but all sequences (9) that differ only in lookahead strings \(x_j\), where \(j = 1, 2, \ldots, \ell\), induce the same (leftmost) derivation. In other words, the lookahead strings \(x_j\) are not needed for computing the prefix of the left parse and the viable suffix.
The left LR($k$) parser uses an additional parsing table called LEFT to establish whether the prefix of the left parse can be computed in some state [$S\gamma$] for some lookahead string $x$, and the left-parse-prefix automaton (LPP) to actually compute sequence (9) with the lookahead strings omitted.

The LEFT table implements mapping

\[
\text{LEFT}: Q_k^G \times (T \cup \{\}$) \times (I_0^G \cup \{\bot\})
\]

where $Q_k^G$ and $I_0^G$ denote the set of LR($k$) states and the set of LR(0) items for grammar $G'$, respectively. It maps LR($k$) state [$S\gamma$] and the contents $x$ of the lookahead buffer to either

- $[A_\ell \rightarrow \alpha_\ell \bullet A_{\ell+1} \beta_\ell]$, where $\alpha_\ell \neq \varepsilon$, if all sequences (9) that are active for $x$, i.e., they end with some some LR($k$) item $[A_\ell \rightarrow \alpha_\ell \bullet A_{\ell+1} \beta_\ell, x_\ell]$ (for different $x_\ell$) where $x \in \text{FIRST}_k^G(A_{\ell+1} \beta_\ell x_\ell)$, differ in lookahead strings only, or
- $\bot$ otherwise.

Hence, the parser can produce the prefix of the left parse and compute the viable suffix if and only if LEFT([S\gamma], x) $\neq \bot$.

The above definition of LEFT works well for the left LR($k$) parser [20]. But as

\[
[S] = \text{desc}^*([S' \rightarrow S_1 S, \varepsilon])
\]

(note that the embedded grammar is being used) and there is only one path to $[[S' \rightarrow S_1 S, \varepsilon]] \in [S]$, the value of LEFT([S\gamma], x) is set to $[S' \rightarrow S_1 S, \varepsilon]$ for all $x \in \text{FIRST}_k^G(S_1)$ if the definition suitable for the let LR($k$) parser is used. It is valid but useless because if the method outlined in Example 4 is used, the embedded left LR($k$) parser would print $\varepsilon$ and stop before ever producing any production of the left parse.

Thus, an exception must be made in state [$S\gamma$]. Provided that the grammar includes the productions $S_1 \rightarrow S_2 \varepsilon$ and $S_2 \rightarrow A \beta$, the value of LEFT([S\gamma], x) must be set to either

![Diagram of LR(1) machine for G_{ax3}](image)
LL conflict resolution using the embedded left LR parser

$\{[S]; \varepsilon\} \rightarrow \{[S]; \varepsilon\} \rightarrow \{[bB]; B \rightarrow Bb \cdot B \rightarrow Bb \cdot B \rightarrow \varepsilon\} \rightarrow \{[bbaa]; \varepsilon\} \rightarrow \{[ac]; \varepsilon\}$

\[
\begin{array}{c}
\begin{array}{c}
\{A \rightarrow bB \cdot a, c\} \in [bB] \\
\pi_2 = B \rightarrow Bb \cdot B \rightarrow Bb \cdot B \rightarrow \varepsilon \cdot \pi_1 \\
\delta_2 = \delta_1
\end{array}
\end{array}
\]

\[
\begin{array}{c}
\begin{array}{c}
\{A \rightarrow bBa \cdot a, c\} \in [bBa] \\
\pi_1 = \varepsilon \cdot \pi_0 \\
\delta_1 = \delta_0
\end{array}
\end{array}
\]

The result: $\pi = S_2 \rightarrow A \cdot A \rightarrow bBaa \cdot B \rightarrow Bb \cdot B \rightarrow Bb \cdot B \rightarrow \varepsilon$ and $\delta = a$

**Fig. 3.** Computing the prefix of the left parse of the string $bbaac \in L(G_{ex3})$ and the corresponding viable suffix after $bbaa$ has been read: the computation starts at the top of the stack (right side of the figure) with $\pi_0 = \varepsilon$ and $\delta_0 = a$, and traverses the stack downwards (towards the left side of the figure, and then downwards).

- $[A_\ell \rightarrow \bullet A_{\ell+1} \beta_\ell]$ if all sequences (9) that are active for $x$, i.e., they end with some some LR($k$) item $[A_\ell \rightarrow \bullet A_{\ell+1} \beta_\ell, x_\ell]$ (for different $x_\ell$) where $x \in \text{FIRST}^G_{k}(A_{\ell+1} \beta_{k} x_\ell)$, differ in lookahead strings only and $\pi = S_2 \rightarrow \bullet A_\ell \beta, y$ desc $[A_\ell \rightarrow \bullet A_{\ell+1} \beta_\ell, x_\ell]$, or

  - $\bot$ otherwise.

The left-parse-prefix automaton represents mapping

$$LPP: \mathbb{I}_0^G \times Q_k^G \rightarrow \mathbb{I}_0^G$$

which is a compact representation of all possible sequences (9) with lookahead strings stripped off. Hence, $LPP(i_0, [S\gamma]) = i'_0$ if and only if there exists some sequence (9) with two consecutive LR($k$) items $i'_k, i_k$, where $i_k \in [S\gamma]$, so that $i_0 (i'_0)$ is equal to $i_k (i'_k)$ without the lookahead string.

**Example 5.** The left-parse-prefix automaton for the grammar $G_{ex3}$ is shown in Figure 4. (In this example, the left-parse-prefix automaton is trivial, i.e., without any loop, but if the grammar is bigger and describes a more complex language, the corresponding LPP gets more complicated — see [20].)
Algorithm 2 Computing the prefix of the left parse and the viable suffix.

**INPUT:** Stack contents of the left LR($k$) parser and a state of LPP automaton.

**OUTPUT:** The prefix of the left parse and the corresponding viable suffix.

```
long-reduction($\Gamma, [A \rightarrow \alpha \bullet \beta]$) = $\langle \pi, \beta \cdot \delta \rangle$ where
$\langle \pi, \delta \rangle = \text{long-reduction}'(\Gamma, [A \rightarrow \alpha \bullet \beta]$)

long-reduction'( $\Gamma, [S' \rightarrow S \bullet S]$) = $\langle \varepsilon, \varepsilon \rangle$

long-reduction'( $\Gamma, ([S_2 \rightarrow \alpha \bullet \beta]) = (A \rightarrow \alpha \bullet \beta)$ where $A \rightarrow \alpha \bullet A\beta$)

long-reduction'( $\Gamma, ([S_1 \rightarrow \alpha \bullet \beta]) = (A \rightarrow \beta \cdot \dot{\pi} \cdot \delta \cdot \beta')$
where $[A' \rightarrow \alpha' \bullet A\beta'] = \text{LPP}(A \rightarrow \beta, [S_1 \rightarrow \alpha \bullet \beta])$

$\langle \pi, \delta \rangle = \text{long-reduction}'(\Gamma, ([S_1 \rightarrow \alpha \bullet \beta]), [A' \rightarrow \alpha' \bullet A\beta'])$

long-reduction'( $\Gamma, ([S \rightarrow \alpha \bullet \beta], [S_2 \rightarrow \alpha \bullet \beta]) = (A \rightarrow \alpha \bullet \beta)$
where $A \rightarrow \alpha \bullet A\beta$

long-reduction'( $\Gamma, ([S \rightarrow \alpha \bullet \beta], [S_2 \rightarrow \alpha \bullet \beta]) = (A \rightarrow \beta \cdot \dot{\pi} \cdot \delta \cdot \beta')$
```

Boštjan Slivnik

Fig. 4. The left-parse-prefix automaton for $G_{ex3}$ — items that are not needed during embedded left LR(1) parsing are shown in bold face.

The algorithms for computing LEFT and LPP can be found in [20]. Once mappings LEFT and LPP are available, the method for computing the prefix of the left parse and the viable suffix as outlined above and illustrated by Example 4 can be formalized as Algorithm 2. It is basically an algorithm which performs a long reduction: a sequence of reductions on productions whose right sides have been only partially pushed on the stack.
LL conflict resolution using the embedded left LR parser

Algorithm 3 Embedded left LR(k) parsing.
1: let $q \in Q^U_k$ denote the topmost state
2: let $x \in \left( T \cup \{\$\} \right)^+k$ denote the LA buffer contents
3: while $(i \leftarrow \text{LEFT}(q, x)) = \perp$ do
4: perform a step of the Schmeiser-Barnard LR(k) parser
5: end while
6: $\langle \pi, \delta \rangle \leftarrow \text{long-reduction}(\text{stack}, i)$
7: PRINT $\pi$
8: return $\delta$

If compared with the similar method used by the left LR(k) parser [20], this one is not only augmented to compute the viable suffix but also simplified in that it does not leave any markers on the stack about which subparses accumulated on the stack have already been printed out. It does not need to do this as after the first long reduction the LR parsing stops, the LR stack is cleared, and the control is given back to the backbone LL(k) parser.

Finally, for the sake of completeness, the sketch of the embedded left LR(k) parser is given as Algorithm 3: in essence, it is a Schmeiser-Barnard LR(k) parser [13] with the option of (a) premature termination and (b) computing the viable suffix.

Algorithm 3 always terminates: if not sooner (including cases where it detects a syntax error), the parser eventually reaches the (final) state $[\$S_2] = \{[S_1 \rightarrow S_2 \cdot x, \$]\}$ where $\text{LEFT}([\$S_2], \$) = [S_1 \rightarrow S_2 \cdot x]$ causing it to exit the loop in lines 3–5.

5. The embedded left LR(k) parser

The embedded left LR(k) parser is the left LR(k) parser for the embedded grammar (with a modified mapping $\text{LEFT}$) which (a) produces the left parse of the substring parsed and the remaining viable suffix, and (b) terminates after the first (simplified) long reduction.

Below, the first theorem establishes that the combination of LL(k) parsing and LR(k) parsing is asymptotically as fast as LR(k) parsing, and the second states that it is just as powerful as LR(k) parsing.

Theorem 2. A backbone LL(k) parser augmented with embedded left LR(k) parsers can parse the input string $w$ derived by the derivation $S \Rightarrow^\pi w$ in time $O(|w|) + O(|\pi|)$.

Proof. Each symbol of $w$ is shifted only once, either by the backbone LL(k) parser or one of the embedded left LR(k) parsers, hence the $O(|w|)$ part.

Each production in $\pi$ is either produced by the backbone LL(k) parser or reduced upon by one of the embedded left LR(k) parsers. There are two different kinds of reductions: reductions performed during the long reduction require
time $k_1|\alpha|$ and ordinary “left” reductions require time $k_2|\alpha|$ for a reduction on $A \rightarrow \alpha$ (but $|\alpha|$ is bounded by a constant depending on the grammar only). Hence the $O(|\pi|)$ part.

**Theorem 3.** A backbone LL$(k)$ parser augmented with embedded left LR$(k)$ parsers can parse any deterministic context-free language.

**Proof.** If $L$ is DCFL, then there exists an LR$(k)$ grammar $G$ so that $L(G) = L$. For each LL$(k)$-conflicting nonterminal $A$ of $\bar{G}$ (the “SLL$(k)$” variant of $G$)

- either an embedded left LR$(k)$ parser can be constructed
- or a nonterminal on the left side of the production where $A$ appear on the right side can be declared LL$(k)$-conflicting nonterminal.

By repeatedly applying this trick all LL$(k)$ conflicts get resolved — if not otherwise, when the initial symbol of $\bar{G}$ is declared to be an LL$(k)$-conflicting symbol (note that the embedded left LR$(k)$ parser for $G$ with the terminating set \{$\$$\}$ can always be constructed).

It must be admitted that Theorem 3 should be taken with a grain of salt. While its proof is technically correct, it exposes the true nature of resolving LL$(k)$ conflicts with embedded left LR$(k)$ parsers. Namely, if embedded left LR$(k)$ parsers are triggered for LL$(k)$ conflicting nonterminals deriving relatively short substrings, then employing embedded left LR$(k)$ parsers makes sense as the amount of a hidden bottom-up parsing is kept within some reasonable limits. Otherwise, if the grammar requires that an embedded left LR$(k)$ parser is triggered relatively close to the root of the derivation tree, then a large part of the input string is going to be parsed by the embedded LR$(k)$ parser and the method loses much of its appeal (to the point that perhaps the left LR$(k)$ parser is more suitable [20]).

6. Conclusion

The embedded left LR$(k)$ parser has been obtained by modifying the left LR$(k)$ parser in two ways. First, the left LR$(k)$ parser was made capable of computing the viable suffix which the unread part of the input string is derived from. Second, if was simplified not to leave any markers on the stack about which subparses accumulated on the stack have been printed out already — as the parser stops after the first “long” reduction anyway. However, the algorithm for minimizing the embedded left LR$(k)$ parser, i.e., for removing states that are not reachable before the first long reduction is performed, is still to be formalized.

At present, both, the backbone LL parser and the embedded left LR parsers, need to use the lookahead buffer of the same length. However, if the LL parser was built around LA$(k)$LL$(\ell)$ parser (where $k \geq \ell$) as defined in [17], then the combined parsing could most probably be formulated as the combination of LL$(\ell)$ and LR$(k)$ parsing (note that LL$(\ell) \subseteq LA(\ell')LL(\ell)$ for any $\ell' \geq \ell$). This would make the combined parser even more memory efficient.
The left LR($k$) parser could be based on the LA($k$)LR($\ell$) parser (most likely for $\ell = 0$) instead of on the canonical LR($k$) parser. This would further reduce the parsing tables while the strength of the resulting combined parser would be reduced from LR($k$) to LA($k$)LR($\ell$): not a significant issue as today LA(1)LR(0) is used instead of LR(1) whenever LR parsing is applied.

By using an LL($k$) parser augmented by the embedded left LR($k$) parsers instead of the left LR($k$) parser the error recovery can be made much better — especially if the error recovery of the embedded left LR($k$) parsers is made using the method described in [19].

Finally, apart from using the embedded left LR($k$) parser for LL($k$) conflict resolution, the embedded left LR($k$) parser can be a convenient method for parsing the embedded domain-specific languages [9]. Furthermore, the termination condition formulated in Section 3 can be considered as a guideline for designing an embedded domain-specific language which fits gently into the enclosing (usually general-purpose) programming language, i.e., without explicit markers denoting the border between the embedded and the enclosing language; the termination condition also provides an efficient automatic method for detecting any syntactic problems arising from the embedding itself.

References


Boštjan Silnik received the M.Sc. and Ph.D. degrees in computer science from the University of Ljubljana in 1996 and 2003 respectively. He is currently at the University of Ljubljana, Faculty of Computer and Information Science. His research interests include parsing algorithms, compilers, formal languages, and distributed algorithms. He has been a member of the ACM since 1996.

Received: December 16, 2011; Accepted: April 2, 2012.