DOI: 10.2298/CSIS111216023S

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 $\mathrm{LL}(k)$ conflicts using small $\mathrm{LR}(k)$ parsers (called embedded left $\mathrm{LR}(k)$ parsers) is described. An embedded left $\mathrm{LR}(k)$ 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 $\mathrm{LR}(k)$ parser if used within $\mathrm{LL}(k)$ (and similar) parsers are defined and examined in-depth. It is proved that an $\mathrm{LL}(k)$ parser augmented with a set of embedded left $\mathrm{LR}(k)$ parsers can parse any deterministic context-free grammar in the same asymptotic time as $\mathrm{LR}(k)$ parser. As the embedded left $\mathrm{LR}(k)$ parser produces the prefix of the left parse, the $\mathrm{LL}(k)$ parser augmented with embedded left $\mathrm{LR}(k)$ 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

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 qubic time (compared with the linear time achieved by the classical LL and LR methods) [21, 22, 15, 14] while $\mathrm{LL}(*)$ 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 $\mathrm{LL}(*)$ 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 $\mathrm{LR}(k)$ parser which can be used within an $\mathrm{LL}(k)$ 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 $\mathrm{LR}(k)$ grammars.

Another method, namely packrat parsing [4], could perhaps have been used to resolve $\mathrm{LL}(k)$ 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 $\mathrm{LR}(k)$ parser must handle instead of $\mathrm{LL}(k)$ parser.

The problem, i.e., the requirements for embedding an $\mathrm{LR}(k)$ parser into the $\mathrm{LL}(k)$ 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 $\mathrm{LR}(k)$ 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 $\mathrm{LR}(k)$ 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 $\Longrightarrow$ (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 $[A \to \alpha \bullet \beta, x]$ where $S \Longrightarrow^*_{\mathrm{rm}} \gamma' A v \Longrightarrow_{\mathrm{rm}} \gamma' \alpha\beta v = \gamma\beta v$ and $x \in \mathrm{FIRST}_k(z)$, denotes the $\mathrm{LR}(k)$ item valid for γ.

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 $\mathrm{LL}(k)$ conflicts

Consider an $\mathrm{LR}(k)$ but non-$\mathrm{LL}(k)$ grammar $G = \langle N, T, P, S\rangle$, i.e., $G \in \mathrm{LR}(k) \setminus \mathrm{LL}(k)$. If the input string $w \in L(G)$ is derived by a derivation

$$S \Longrightarrow^{\pi_u}_{G,\mathrm{lm}} uA\delta \Longrightarrow^{\pi_{v'}}_{G,\mathrm{lm}} uv'\delta \Longrightarrow^{\pi_{v''}}_{G,\mathrm{lm}} uv'v'' = uv = w \quad , \tag{1}$$

the expected result of parsing it with an $\mathrm{LL}(k)$ parser is the left parse

$$\pi_w = \pi_u \pi_{v'}\ \pi_{v''} \in P^* \quad . \tag{2}$$

Since $G \notin \mathrm{LL}(k)$, an $\mathrm{LL}(k)$ conflict is likely to occur and must therefore be resolved. $\mathrm{LL}(*)$ parsing [11], for instance, tries to determine the next production using a set of DFAs: if A causes an $\mathrm{LL}(1)$ 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 $\mathrm{LL}(*)$ parser continues parsing by reading the entire string v again (not just the unscanned suffix of it). While $\mathrm{LL}(*)$ 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, $\mathrm{LL}(*)$ parsing prohibits left-recursive productions.

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

1. **$\mathrm{LR}(k)$ parsers used within an $\mathrm{LL}(k)$ parser cannot rely on the end-of-input symbol $ to terminate** (unlike the standard $\mathrm{LR}(k)$ parsers can).
   More precisely, if an $\mathrm{LR}(k)$ 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 $x \in \mathrm{FIRST}^G_k(\delta\$)$ in its lookahead buffer (instead of $\$$).
2. **$\mathrm{LR}(k)$ parsers used within an $\mathrm{LL}(k)$ parser must produce the left parse of its input** (instead of the right parse as the standard $\mathrm{LR}(k)$ parsers do).
   More precisely, a standard $\mathrm{LR}(k)$ parser for A produces the right parse of v', but if used within an $\mathrm{LL}(k)$ parser, it should produce the left parse $\pi_{v'}$.

If an $\mathrm{LR}(k)$ parser fulfills both conditions, it is called the *embedded left* $\mathrm{LR}(k)$ *parser*: embedded as it can be used within the *backbone* $\mathrm{LL}(k)$ 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 $\mathrm{LR}(k)$ parser

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

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

$$S \longrightarrow aAb \,|\, bAab \quad \text{and} \quad A \longrightarrow Aa \,|\, a \quad .$$

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

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

$$\$S\,|\,aa\ldots\$ \vdash_{\mathrm{LL}} \$bAa\,|\,aa\ldots\$ \vdash_{\mathrm{LL}} \$bA\,|\,a\ldots\$ \quad ,$$

the backbone $\mathrm{LL}(1)$ parser reaches the configuration $\$bA\,|\,a\ldots\$$ (the strings on the left side and on the right side of $|$ 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 $|$). The configuration $\$bA\,|\,a\ldots\$$ exhibits an $\mathrm{LL}(1)$ conflict on $A\,|\,a$. At this point, an embedded $\mathrm{LR}(1)$ 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 $\mathrm{LL}(1)$ parsing starts as

$$\$S\,|\,baa\ldots\$ \vdash_{\mathrm{LL}} \$baAb\,|\,baa\ldots\$ \vdash_{\mathrm{LL}} \$baA\,|\,aa\ldots\$ \quad .$$

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

$$\$[\varepsilon]\,|\,aa\ldots\$ \vdash_{\mathrm{LR}} \$[\varepsilon][a]\,|\,a\ldots\$ \vdash_{\mathrm{LR}} \$[\varepsilon][A]\,|\,a\ldots\$ \quad ,$$

the embedded $\mathrm{LR}(1)$ 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 $\mathrm{LR}(1)$ parser should terminate and handle the control back to the backbone $\mathrm{LL}(1)$ parser, otherwise it should continue by shifting and reducing using $A \longrightarrow Aa$. Therefore, the embedded $\mathrm{LR}(1)$ 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 $\mathbf{LR}(k)$ parser must sometimes parse substrings derived from a sentential form starting with the $\mathbf{LL}(k)$-conflicting nonterminal instead of from that nonterminal only.** More precisely, if the first part of the derivation (1) is rewritten as

$$S \Longrightarrow^{\pi_{u'}}_{G'\mathrm{lm}} u'B\delta' \Longrightarrow_{G'\mathrm{lm}} u'\beta_1 A\beta_2\delta' \Longrightarrow^{\pi_{u''}}_{G'\mathrm{lm}} u'u''A\beta_2\delta' = uA\delta \quad , (3)$$

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 $\mathbf{LR}(k)$ 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 $\mathrm{LR}(k)$ parser for $A\beta_2'$ depends on the contents of the set $\mathrm{FIRST}^G_k(\beta_2''\delta')$.

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

The easiest way to resolve the right context of the embedded $\mathrm{LR}(k)$ 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 $\mathrm{LL}(k)$ grammar to an $\mathrm{SLL}(k)$ 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

$$\bar{N} = \{\langle A, \mathcal{F}_A\rangle;\ S \Longrightarrow^*_{\mathrm{lm}} uA\delta \wedge \mathcal{F}_A = \mathrm{FIRST}^G_k(\delta)\} \quad .$$

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 $i = 1, 2, \ldots, n$,

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

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

Example 2. If the grammar G_{ex1} is transformed, a grammar $\bar{G}_{\mathrm{ex1}}$

$$\begin{aligned} \langle S, \{\varepsilon\}\rangle &\longrightarrow a\langle A, \{b\}\rangle b \mid b\langle A, \{a\}\rangle ab \\ \langle A, \{a\}\rangle &\longrightarrow \langle A, \{a\}\rangle a \mid a \\ \langle A, \{b\}\rangle &\longrightarrow \langle A, \{a\}\rangle a \mid a \end{aligned}$$

is obtained. Two embedded $\mathrm{LR}(1)$ parsers are needed: $\langle A, \{b\}\rangle$ and $\langle Aa, \{b\}\rangle$:

1. The parser for $\langle A, \{b\}\rangle$ results from production $\langle S, \{\varepsilon\}\rangle \longrightarrow a\langle A, \{b\}\rangle b$: the parser's right context is $\{b\} = \mathrm{FIRST}_k^G(b\{\varepsilon\})$: b follows $\langle A, \{b\}\rangle$ in production $\langle S, \{\varepsilon\}\rangle \longrightarrow a\langle A, \{b\}\rangle b$ and $\{\varepsilon\}$ (from $\langle S, \{\varepsilon\}\rangle$) determines the right context of the entire production $\langle S, \{\varepsilon\}\rangle \longrightarrow a\langle A, \{b\}\rangle b$.
2. The parser for $\langle Aa, \{b\}\rangle$ results from production $\langle S, \{\varepsilon\}\rangle \longrightarrow b\langle A, \{a\}\rangle ab$, again with the right context $\{b\} = \mathrm{FIRST}_k^G(b\{\varepsilon\})$: b follows the sentential form $\langle A, \{a\}\rangle a$ in production $\langle S, \{\varepsilon\}\rangle \longrightarrow b\langle A, \{a\}\rangle ab$ and $\{\varepsilon\}$ (from $\langle S, \{\varepsilon\}\rangle$) determines the right context of the entire production.

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

To resolve conflicts during $\mathrm{LL}(k)$ parsing based on the grammar $\bar{G}$, every production

$$\langle B, \mathcal{F}_B\rangle \longrightarrow \beta_1\langle A, \mathcal{F}_A\rangle\beta_2 \in \bar{P} \quad (4)$$

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

$$\langle B, \mathcal{F}_B\rangle \longrightarrow \beta_1\langle\!\langle A\beta_2', \mathcal{F}_{A\beta_2'}\rangle\!\rangle\beta_2''$$

where $\beta_2 = \beta_2'\beta_2''$ and $\mathcal{F}_{A\beta_2'} = \mathrm{FIRST}_k^{\bar{G}}(\beta_2''\mathcal{F}_B)$. The new symbol $\langle\!\langle A\beta_2', \mathcal{F}_{A\beta_2'}\rangle\!\rangle \notin \bar{N}$ acts as a trigger for the embedded $\mathrm{LR}(k)$ 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, β_2' should be as short as possible, i.e., ε in the best case. If, on the other hand, not even $\beta_2' = \beta_2$ and $\beta_2'' = \varepsilon$ suffices for the safe termination of the embedded $\mathrm{LR}(k)$ parser, $\langle B, \mathcal{F}_B\rangle$ must be declared a conflicting nonterminal.

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

$$\hat{G}_{\beta,\mathcal{F}} = \langle \hat{N}, T, \hat{P}, S_1\rangle$$

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

$$\hat{P} = P \cup \{S_1 \longrightarrow S_2x, S_2 \longrightarrow \beta\ ;\ x \in \mathcal{F}\} \quad .$$

The trick is obvious: the *embedded* $\mathrm{LR}(k)$ *parser for* $\hat{G}_{\beta,\mathcal{F}}$ must accept its input no later than when the reduction on $S_2 \longrightarrow \beta$ is due. In other words, if the reduce on $S_2 \longrightarrow \beta$ 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 $S_2 \longrightarrow \beta$ cannot be determined (because of the $\mathrm{LR}(k)$ conflict), the embedded $\mathrm{LR}(k)$ parser for $\langle\!\langle \beta, \mathcal{F}\rangle\!\rangle$ cannot be used.

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

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

$$S \Longrightarrow^*_{G,\mathrm{lm}} uB\delta \Longrightarrow_{G,\mathrm{lm}} u\beta_1\beta_2'\beta_2''\delta \quad .$$

Grammar $\hat{G} = \langle \hat{N}, T, \hat{P}, S_1\rangle$ where

$$\hat{N} = N \cup \{S_1, S_2\} \text{ for } S_1, S_2 \notin N \text{ and}$$
$$\hat{P} = P \cup \{S_1 \longrightarrow S_2x, S_2 \longrightarrow \beta_2' \ ; \ x \in \mathrm{FIRST}_k^G(\beta_2''\delta)\} \quad ,$$

is not an $\mathrm{LR}(k)$ grammar if and only if

- *either $\beta_2'' = \varepsilon$ and $[S \to \beta_2'\bullet, x'], [B \to \beta_2'\bullet, x'] \in [\$\beta_2']_{\hat{G}}$*
- *or $\beta_2'' \neq \varepsilon$ and $[S_2 \to \beta_2'\bullet, x'], [A \to \alpha\bullet\alpha', y'] \in [\$\beta_2']_{\hat{G}}$ where $\alpha' \neq \varepsilon$ and $x' \in \mathrm{FIRST}_k^G(\alpha'y')$.*

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 $\mathrm{LR}(k)$ machine for G that includes all $[B \to \beta_1\bullet\beta_2'\beta_2'', y]$ where $y \in \mathrm{FIRST}_k^G(\delta)$. This state corresponds to the initial state of the $\mathrm{LR}(k)$ machine for $\hat{G}$. By careful examination of all possibilities only those possibilities permitting $\mathrm{LR}(k)$ conflicts in $\hat{G}$ are singled out. The formal proof follows.

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

$$[S_1 \to \bullet S_2x, \$] \text{ and } [S_2 \to \bullet\beta_2', x'] \quad ,$$

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

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

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

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

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

implies the existence of the rightmost derivation

$$S \Longrightarrow^*_{G,\mathrm{rm}} \gamma Bv'' \Longrightarrow_{G,\mathrm{rm}} \gamma\beta_1\beta_2'\beta_2''v'' \Longrightarrow^*_{G,\mathrm{rm}} w \quad .$$

Moreover, if $\delta \Longrightarrow^*_G v''$, then the viable prefix γ depends only on the left sentential form $uB\delta$, i.e., it is unique for all w. Therefore,

$$\{[B \to \beta_1\bullet\beta_2'\beta_2'', y'];\ y' \in \mathrm{FIRST}_k^G(\delta\$)\} \subseteq [\$\gamma\beta_1]_G$$

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

Consider any two items i_1 and i_2 (except items based on the production $S' \longrightarrow \$S_1\$$ as these items are never involved in an $\mathrm{LR}(k)$ conflict) in any state $[\$\hat{\gamma}]_{\hat{G}}$ of the canonical $\mathrm{LR}(k)$ machine for $\hat{G}$, i.e., $i_1, i_2 \in [\$\hat{\gamma}]_{\hat{G}}$:

1. If i_1 and i_2 are based on productions in P, then $i_1, i_2 \in [\$\gamma\beta_1\hat{\gamma}]_G$ and there is no $\mathrm{LR}(k)$ conflict between i_1 and i_2 since $G \in LR(k)$.
2. If i_1 and i_2 are based on productions in $\hat{P} \setminus P$, the following three cases must be considered:
   (a) $i_1 = [S_1 \to \hat{\gamma}\bullet\alpha, \$]$ and $i_2 = [S_1 \to \hat{\gamma}\bullet\alpha', \$]$:
   If $\hat{\gamma} = \varepsilon$, then i_1 and i_2 imply no actions because α and α' 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 vise versa).
   (b) $i_1 = [S_1 \to \hat{\gamma}\bullet\alpha, \$]$ and $i_2 = [S_2 \to \hat{\gamma}\bullet\alpha', y']$ (or vice-versa):
   i_1 implies no action if $\hat{\gamma} = \varepsilon$ as α starts with S_2. The other case, if $\hat{\gamma} \neq \varepsilon$, is impossible: $\hat{\gamma}$ starts with S_2 in i_1 and does not start with S_2 in i_2.
   (c) $i_1 = [S_2 \to \hat{\gamma}\bullet\alpha, y]$ and $i_2 = [S_2 \to \hat{\gamma}\bullet\alpha', y']$:
   $\alpha = \alpha'$ and both items imply either the same action or imply no action.
3. If i_1 is based on a production in $\hat{P} \setminus P$ and i_2 is based on a production in P (or vice versa), the following two cases must be considered:
   (a) $i_1 = [S_1 \to \hat{\gamma}_1\hat{\gamma}_2\bullet\alpha, \$]$ and $i_2 = [A \to \hat{\gamma}_2\bullet\alpha', y']$:
   If $\hat{\gamma}_1\hat{\gamma}_2 = \varepsilon$, then i_1 implies no action as α starts with S_2. The other case, if $\hat{\gamma}_1\hat{\gamma}_2 \neq \varepsilon$, is impossible: $i_1 \in [\$\hat{\gamma}_1\hat{\gamma}_2]_{\hat{G}}$ while $i_2 \notin [\$\hat{\gamma}_1\hat{\gamma}_2]_{\hat{G}}$.
   (b) $i_1 = [S_2 \to \hat{\gamma}_1\hat{\gamma}_2\bullet\alpha, y]$ and $i_2 = [A \to \hat{\gamma}_2\bullet\alpha', y']$:
   As $i_1, i_2 \in [\$\hat{\gamma}_1\hat{\gamma}_2]_{\hat{G}}$, so does

   $$[B \to \beta_1\gamma_1\hat{\gamma}_2\bullet\alpha\beta_2'', y''], [A \to \hat{\gamma}_2\bullet\alpha', y'] \in [\$\gamma\beta_1\hat{\gamma}_1\hat{\gamma}_2]_G$$

   where $y \in \mathrm{FIRST}_k^G(\beta_2''y_2'')$.
   - 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

     $$[S_2 \to \hat{\gamma}_1\hat{\gamma}_2\bullet\alpha, y], [A \to \hat{\gamma}_2\bullet, y'] \in [\$\hat{\gamma}_1\hat{\gamma}_2]_{\hat{G}}$$

     exhibit a shift-reduce conflict if and only if $y' \in \mathrm{FIRST}_k^G(\alpha y)$. But then items

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

     exhibit a conflict. This is not possible as $G \in \mathrm{LR}(k)$ and therefore items i_1 and i_2 do not exhibit a conflict in $\hat{G}$.
   - If $\alpha = \varepsilon$ and $\alpha' \neq \varepsilon$, then

     $$[S_2 \to \hat{\gamma}_1\hat{\gamma}_2\bullet, y], [A \to \hat{\gamma}_2\bullet\alpha', y'] \in [\$\hat{\gamma}_1\hat{\gamma}_2]_{\hat{G}}$$

     exhibit a shift-reduce conflict if $y \in \mathrm{FIRST}_k^G(\alpha'y')$. But

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

     and the only possibility of a shift-reduce conflict in $[\$\hat{\gamma}_1\hat{\gamma}_2]_{\hat{G}}$ without the conflict in $[\$\gamma\beta_1\hat{\gamma}_1\hat{\gamma}_2]_G$ is that $\beta_2' = \hat{\gamma}_1\hat{\gamma}_2$ and $\beta_2'' \neq \varepsilon$.

– If $\alpha = \varepsilon$ and $\alpha' = \varepsilon$, then

$$[S_2 \to \hat{\gamma}_1\hat{\gamma}_2\bullet, y], [A \to \hat{\gamma}_2\bullet, y'] \in [\$\hat{\gamma}_1\hat{\gamma}_2]_{\hat{G}}$$

exhibit a reduce-reduce conflict if $y = y'$. But

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

and the only possibility of a reduce-reduce conflict in $[\$\hat{\gamma}_1\hat{\gamma}_2]_{\hat{G}}$ without the conflict in $[\$\gamma\beta_1\hat{\gamma}_1\hat{\gamma}_2]_G$ is that $\beta_1\gamma_1 = \varepsilon$, $\beta_2' = \hat{\gamma}_2$ and $\beta_2'' = \varepsilon$.

Finally, proving the theorem in the opposite direction is trivial — if the canonical $\mathrm{LR}(k)$ machine for the grammar $\hat{G}$ contains an $\mathrm{LR}(k)$ conflict, then clearly $\hat{G} \notin \mathrm{LR}(k)$. ■

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

$$S \Longrightarrow^*_{G,\mathrm{lm}} uB\delta \Longrightarrow_{G,\mathrm{lm}} u\beta_1\beta_2'\delta \quad .$$

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 $\hat{P} = P \cup \{S_1 \longrightarrow S_2x, S_2 \longrightarrow \beta_2' \; ; \; x \in \mathrm{FIRST}_k^G(\delta)\}$ is not an $\mathrm{LR}(k)$ grammar if and only if

$$[S_2 \to \bullet\beta_2', x'] \;\mathbf{desc}^*\; [B \to \bullet\beta_2', x']$$

where $B \neq S_2$ and $x' \in \mathrm{FIRST}_k^G(\delta\$)$ [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 $\langle A, \mathcal{F}_A\rangle\beta_2$ in production

$$\langle B, \mathcal{F}_B\rangle \longrightarrow \beta_1\langle A, \mathcal{F}_A\rangle\beta_2$$

where the embedded $\mathrm{LR}(k)$ parser must be employed to resolve the $\mathrm{LL}(k)$ conflict caused by $\langle A, \mathcal{F}_A\rangle$. Once Theorem 1 is digested, the algorithm comes out relatively simple: it just checks both conditions exposed by Theorem 1, one for $\beta_2'' = \varepsilon$ and the other for $\beta_2'' \neq \varepsilon$.

4. Terminating while producing the left parse

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

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

$$S \Longrightarrow^*_{G,\mathrm{rm}} \gamma v \Longrightarrow^*_{G,\mathrm{rm}} uv \quad . \tag{5}$$

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

$$\$[\$][\$X_1][\$X_1X_2]\ldots[\$X_1X_2\ldots X_n] \,|\, v\$ \tag{6}$$

Algorithm 1 Computing the shortest prefix β' of the sentential form $\beta = \beta'\beta''$ so that the embedded $\mathrm{LR}(k)$ grammar $\hat{G}_{\beta',\mathcal{F}'} \in \mathrm{LR}(k)$ where $\mathcal{F}' = \mathrm{FIRST}_k^G(\beta''\mathcal{F})$.
INPUT: The sentential form $\beta = X_1X_2\dots X_n$ and the right context $\mathcal{F}$.
OUTPUT: The prefix β' (or $\perp$ if the prefix does not exist).

1: **for** $i \leftarrow 1\dots(n-1)$ **do**
2: $\quad\beta' = X_1X_2\dots X_i$ and $\beta'' = X_{i+1}X_{i+2}\dots X_n$
3: $\quad$**if** $\neg(\exists[A\to\alpha\bullet\alpha', y'] \in [\$\beta']_{\hat{G}} : \mathrm{FIRST}_k^{\hat{G}}(\alpha' y') \cap \mathrm{FIRST}_k^{\hat{G}}(\beta''\mathcal{F}\$) \neq \emptyset)$ **then**
4: $\quad\quad$**return** β'
5: $\quad$**end if**
6: **end for**
7: $\beta' = X_1X_2\dots X_n$ and $\beta'' = \varepsilon$
8: **if** $\neg(\exists[A\to\alpha\bullet, x'] \in [\$\beta']_{\hat{G}} : x' \cap \mathrm{FIRST}_k^{\hat{G}}(\mathcal{F}\$) \neq \emptyset)$ **then**
9: $\quad$**return** β'
10: **end if**
11: **return** $\perp$

where $X_1X_2\dots X_n = \gamma$, $[\$X_1X_2\dots X_n]$ is the current parser state and $x = k{:}\,v\$$ is the contents of the lookahead buffer. ($[\$X_1X_2\dots X_j]$, for $j = 0, 1, \dots, n$, denotes the state of the canonical $\mathrm{LR}(k)$ machine M_G reachable from the state $[\$]$ by string $X_1X_2\dots X_j$ where M_G is based on the $\$$-augmented grammar G' obtained by adding the new start symbol S' with production $S' \longrightarrow \$S\$$ to G).

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

$$\$\langle[\$];\varepsilon\rangle\langle[\$X_1];\pi(X_1)\rangle\langle[\$X_1X_2];\pi(X_2)\rangle\dots\dots\langle[\$X_1X_2\dots X_n];\pi(X_n)\rangle \mathbin{\rule[-0.2ex]{0.15em}{1.6ex}} v\$ \quad (7)$$

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

$$X_1X_2\dots X_n \Longrightarrow_{G,\mathrm{lm}}^{\pi(X_1)\pi(X_2)\dots\pi(X_n)} u \quad .$$

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 ε.
- 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_{ex3} with productions

$$S_1 \longrightarrow S_2c\ , \quad S_2 \longrightarrow A\ , \quad A \longrightarrow aa\,|\,aB\,|\,bBa\,|\,bBaa\ , \quad B \longrightarrow Bb\,|\,\varepsilon \quad .$$

Table 1. Parsing the string $bbbaac \in L(G_{\mathrm{ex3}})$ using the Schmeiser-Barnard $\mathrm{LR}(1)$ parser.

	STACK	INPUT
1	$\$\ \langle[\$];\varepsilon\rangle$	$bbbaac\$$
2	$\$\ \langle[\$];\varepsilon\rangle\ \langle[\$b];\varepsilon\rangle$	$bbaac\$$
3	$\$\ \langle[\$];\varepsilon\rangle\ \langle[\$b];\varepsilon\rangle\ \langle[\$bB];\pi_1 = B\rightarrow\varepsilon\rangle$	$bbaac\$$
4	$\$\ \langle[\$];\varepsilon\rangle\ \langle[\$b];\varepsilon\rangle\ \langle[\$bB];\pi_1 = B\rightarrow\varepsilon\rangle\ \langle[\$bBb];\varepsilon\rangle$	$baac\$$
5	$\$\ \langle[\$];\varepsilon\rangle\ \langle[\$b];\varepsilon\rangle\ \langle[\$bB];\pi_2 = B\rightarrow Bb\cdot\pi_1\rangle\ \langle[\$bBb];\varepsilon\rangle$	$aac\$$
6	$\$\ \langle[\$];\varepsilon\rangle\ \langle[\$b];\varepsilon\rangle\ \langle[\$bB];\pi_3 = B\rightarrow Bb\cdot\pi_2\rangle$	$aac\$$
7	$\$\ \langle[\$];\varepsilon\rangle\ \langle[\$b];\varepsilon\rangle\ \langle[\$bB];\pi_3 = B\rightarrow Bb\cdot\pi_2\rangle\ \langle[\$bBa];\varepsilon\rangle$	$ac\$$
8	$\$\ \ldots\ \langle[\$bB];\pi_3 = B\rightarrow Bb\cdot\pi_2\rangle\ \langle[\$bBa];\varepsilon\rangle\ \langle[\$bBaa];\varepsilon\rangle$	$c\$$
9	$\$\ \langle[\$];\varepsilon\rangle\ \langle[\$A];\pi_4 = A\rightarrow bBaa\cdot\pi_3\rangle$	$c\$$
10	$\$\ \langle[\$];\varepsilon\rangle\ \langle[\$S_2];\pi_5 = S_2\rightarrow A\cdot\pi_4\rangle$	$c\$$
11	$\$\ \langle[\$];\varepsilon\rangle\ \langle[\$S_2];\pi_6 = S_2\rightarrow A\cdot\pi_5\rangle\ \langle[\$S_2c],\varepsilon\rangle$	$\$$
12	$\$\ \langle[\$];\varepsilon\rangle\ \langle[\$S_1];\pi_7 = S_1\rightarrow S_2c\cdot\pi_6\rangle$	$\$$

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

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

The left $\mathrm{LR}(k)$ 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

$$S \Longrightarrow^{\pi(u)}_{G,\mathrm{lm}} u\delta \Longrightarrow^{*}_{G,\mathrm{lm}} uv \quad , \tag{8}$$

then the left $\mathrm{LR}(k)$ 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 $\mathrm{LR}(k)$ 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

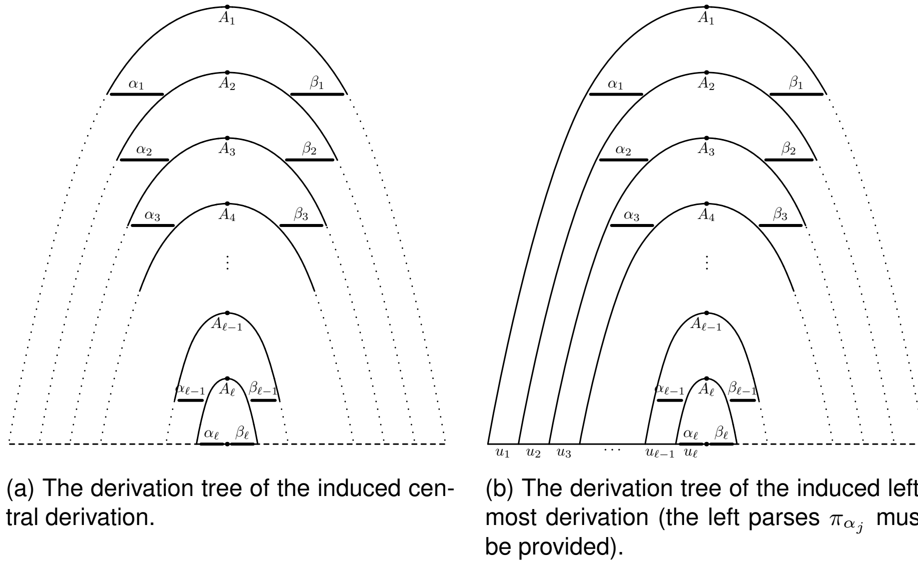
$$\begin{array}{c}[A_0 \rightarrow \bullet\alpha_0 A_1\beta_0, x_0]\cdot\ldots\cdot[A_0 \rightarrow \alpha_0\bullet A_1\beta_0, x_0]\cdot\\ \cdot[A_1 \rightarrow \bullet\alpha_1 A_2\beta_1, x_1]\cdot\ldots\cdot[A_1 \rightarrow \alpha_1\bullet A_2\beta_1, x_1]\cdot\\ \vdots\\ \cdot[A_\ell \rightarrow \bullet\alpha_\ell A_{\ell+1}\beta_\ell, x_\ell]\cdot\ldots\cdot[A_\ell \rightarrow \alpha_\ell\bullet A_{\ell+1}\beta_\ell, x_\ell]\end{array} \tag{9}$$

where $[A_0 \rightarrow \bullet\alpha_0 A_1\beta_0, x_0] = [S' \rightarrow \bullet\$S\$, \varepsilon]$, $\gamma = \alpha_0\alpha_1\ldots\alpha_\ell$ and $k{:}\,v\$ \in \mathrm{FIRST}^{G'}_k(A_{\ell+1}\beta_\ell x_\ell)$ (and $A_{\ell+1} = \varepsilon$); the horizontal dots denote repetitive application of operation $\mathbf{passes}$ (or GOTO) while the vertical dots denote the application of $\mathbf{desc}$ (or $\mathrm{CLOSURE}$).

Sequence (9) induces the *(induced) central derivation*

$$S' = A_0 \Longrightarrow_G \alpha_0 A_1 \beta_0 \Longrightarrow_G \alpha_0 \alpha_1 A_2 \beta_1 \beta_0 \Longrightarrow_G \ldots \Longrightarrow_G \\ \Longrightarrow_G \alpha_0 \alpha_1 \ldots \alpha_\ell A_{\ell+1} \beta_\ell \beta_{\ell-1} \ldots \beta_0 \quad ;$$

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



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

(b) The derivation tree of the induced leftmost derivation (the left parses π_{α_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 $\alpha_j \Longrightarrow^{\pi(\alpha_j)}_{G',\mathrm{lm}} u_j$ for $j = 0, 1, \ldots, \ell$, are provided, sequence (9) induces the *(induced) leftmost derivation*

$$\begin{aligned} S' = A_0 &\Longrightarrow_{G,\mathrm{lm}} \alpha_0 A_1 \beta_0 \Longrightarrow^{\pi(\alpha_0)}_{G,\mathrm{lm}} u_0 A_1 \beta_0 \\ &\Longrightarrow_{G,\mathrm{lm}} u_0 \alpha_1 A_2 \beta_1 \beta_0 \Longrightarrow^{\pi(\alpha_1)}_{G,\mathrm{lm}} u_0 u_1 A_2 \beta_1 \beta_0 \\ &\vdots \\ &\Longrightarrow_{G,\mathrm{lm}} u_0 u_1 \ldots u_{\ell-1} \alpha_\ell A_{\ell+1} \beta_\ell \beta_{\ell-1} \ldots \beta_0 \\ &\qquad \Longrightarrow^{\pi(\alpha_\ell)}_{G,\mathrm{lm}} u_0 u_1 \ldots u_\ell A_{\ell+1} \beta_\ell \beta_{\ell-1} \ldots \beta_0 \end{aligned}$$

where $u = u_0 u_1 \ldots u_\ell$ and $k{:}\, v\$ \in \mathrm{FIRST}^{G'}_k(\beta_\ell \beta_{\ell-1} \ldots \beta_0 \$)$. The corresponding derivation tree is shown in Figure 1(b) and the left parse of the induced leftmost

derivation is therefore

$$\pi(u) = A_0 \longrightarrow \alpha_0 A_1 \beta_0 \cdot \pi(\alpha_0) \cdot A_1 \longrightarrow \alpha_1 A_2 \beta_1 \cdot \pi(\alpha_1) \cdot \quad \ldots \quad \cdot \\ \cdot A_\ell \longrightarrow \alpha_\ell A_{\ell+1} \beta_\ell] \cdot \pi(\alpha_\ell) \quad . \tag{10}$$

(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 $\pi(\alpha_j)$ of the left parse (10) are available on the parser stack because $\alpha_0\alpha_1 \ldots \alpha_\ell = \gamma = X_1 X_2 \ldots X_n$, but productions $A_j \longrightarrow \alpha_j A_{j+1} \beta_j$ 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 $i = [A_\ell \to \alpha_\ell \bullet A_{\ell+1}\beta_\ell, x_\ell]$, the stack is traversed downwards:

- If $i = [A \to \bullet\beta, x]$, then (a) i expands the nonterminal A by production $A \longrightarrow \beta$ and (b) i', the item that precedes i in sequence (9), is in the same state. Hence, let $\pi := A \longrightarrow \beta \cdot \pi$ and $i := i'$.
- If $i = [A \to \alpha X \bullet \beta, x] \in [\$\gamma X]$ for some γ, then (a) the left parse $\pi(X)$ is available on the stack and (b) i' is in the state $[\$\gamma]$ (which is found beneath $[\$\gamma X]$). 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 $[S_2 \to \bullet\beta, x] \in [\$]$, for some $\beta \in (N \cup T)^*$ and $x \in (T \cup \{\$\})^{*k}$, is reached (the production $S_2 \longrightarrow \beta$ 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 δ^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 $i = [A \to \bullet\beta, x]$, let $\delta := \delta \cdot \beta'$ where $i' = [A' \to \alpha' \bullet A\beta', x']$ is the item preceding i in sequence (9).

Example 4. Consider again the grammar G_{ex3} and the input string $bbbaac \in L(G_{\text{ex3}})$ 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 $[\$bBa]$, namely $[A \to bBa\bullet a, \$]$. Furthermore, there exist exactly one sequence of $\mathrm{LR}(1)$ items starting with $[S' \to \bullet\$S_1\$, \varepsilon] \in [\varepsilon]$ and ending with $[A \to bBa\bullet a, \$] \in [\$bS_2a]$:

$$[S' \to \bullet\$S_1\$, \varepsilon] \cdot [S' \to \$\bullet S_1\$, \varepsilon] \cdot [S_1 \to \bullet S_2 c, \$] \cdot [S_2 \to \bullet A, c] \cdot \\ \cdot [A \to \bullet bBaa, \$] \cdot [A \to b\bullet Baa, \$] \cdot \ \ldots \cdot [A \to bB\bullet aa, \$] \cdot [A \to bBa\bullet a, \$]$$

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.

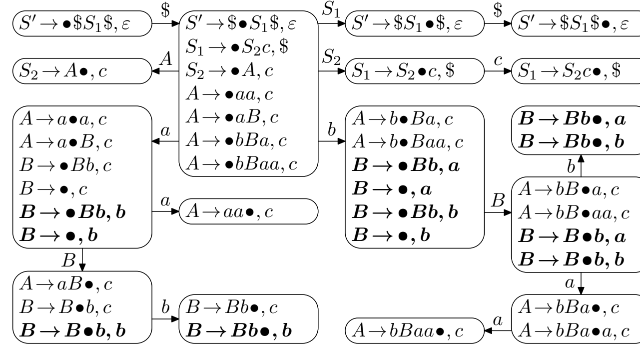


Fig. 2. The canonical $\mathrm{LR}(1)$ machine for G_{ex3} — items that end multiple sequences starting with $[S' \to \bullet\$S\$, \varepsilon] \in [\varepsilon]$ are shown in bold face.

The left $\mathrm{LR}(k)$ parser uses an additional parsing table called LEFT to establish whether the prefix of the left parse can be computed in some state $[\$\gamma]$ 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

$$\text{LEFT: } Q_k^G \times (T \cup \{\$\})^{*k} \longrightarrow (I_0^G \cup \{\bot\})$$

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

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

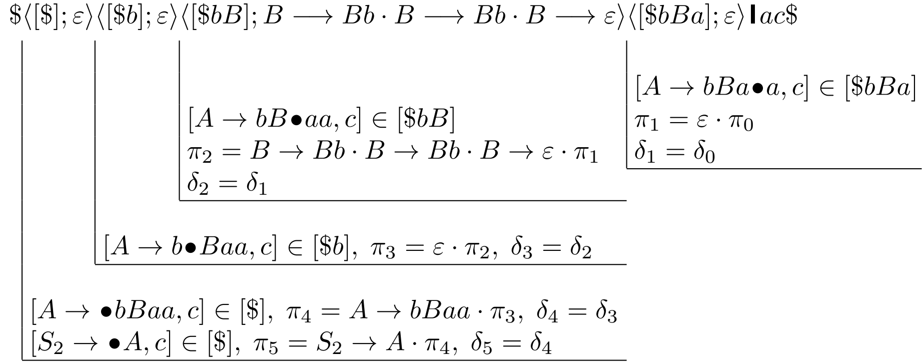
Hence, the parser can produce the prefix of the left parse and compute the viable suffix if and only if $\text{LEFT}([\$\gamma], x) \neq \bot$.

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

$$[\$] = \mathbf{desc}^*(\{[S' \to \$\bullet S_1\$, \varepsilon]\})$$

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

Thus, an exception must be made in state $[\$]$. Provided that the grammar includes the productions $S_1 \longrightarrow S_2 y$ and $S_2 \longrightarrow A\beta$, the value of $\text{LEFT}([\$], x)$ must be set to either



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

Fig. 3. Computing the prefix of the left parse of the string $bbbaac \in L(G_{\text{ex3}})$ and the corresponding viable suffix after $bbba$ 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).

- $[A_\ell \to \bullet A_{\ell+1}\beta_\ell]$ if all sequences (9) that are active for x, i.e., they end with some some $\mathrm{LR}(k)$ item $[A_\ell \to \bullet A_{\ell+1}\beta_\ell, x_\ell]$ (for different x_ℓ) where $x \in \mathrm{FIRST}_k^{G'}(A_{\ell+1}\beta_\ell x_\ell)$, differ in lookahead strings only and

$$[S_2 \to \bullet A_\ell\beta, y] \ \mathbf{desc}\ [A_\ell \to \bullet A_{\ell+1}\beta_\ell, x_\ell] \quad ,$$

  or
- $\perp$ otherwise.

The left-parse-prefix automaton represents mapping

$$\text{LPP: } I_0^G \times Q_k^G \longrightarrow I_0^G$$

which is a compact representation of all possible sequences (9) with lookahead strings stripped off. Hence, $\text{LPP}(i_0, [\$\gamma]) = i'_0$ if and only if there exists some sequence (9) with two consecutive $\mathrm{LR}(k)$ items i'_k, i_k, where $i_k \in [\$\gamma]$, 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].)

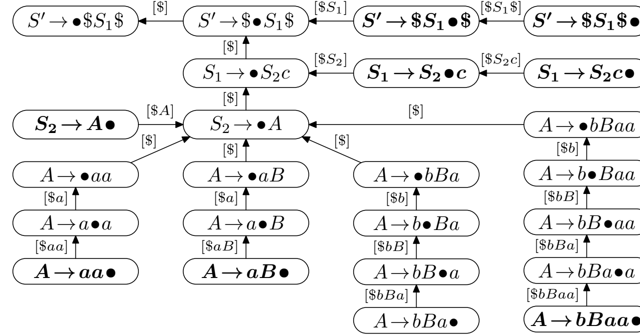


Fig. 4. The left-parse-prefix automaton for G_{ex3} — items that are not needed during embedded left $\mathrm{LR}(1)$ parsing are shown in bold face.

Mapping LEFT for G_{ex3} is defined as

$$\begin{aligned}
\text{LEFT}([\$S_2], c) &= [S_2 \to A\bullet c]\\
\text{LEFT}([\$a], a) &= [A \to a\bullet a]\\
\text{LEFT}([\$a], b) &= [A \to a\bullet B]\\
\text{LEFT}([\$bBa], \$) &= [A \to bBa\bullet]\\
\text{LEFT}([\$bBa], b) &= [A \to bBa\bullet a]
\end{aligned}$$

(in all other cases, the value of LEFT equals $\perp$). Note that LEFT$([\$], a) = \perp$ and LEFT$([\$], b) = \perp$ because of $A \longrightarrow aa|aB$ and $A \longrightarrow bBa|bBaa$, respectively. ■

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.

Algorithm 2 Computing the prefix of the left parse and the viable suffix.
INPUT: Stack contents of the left $\mathrm{LR}(k)$ parser and a state of LPP automaton.
OUTPUT:The prefix of the left parse and the corresponding viable suffix.

long-reduction $(\Gamma, [A \to \alpha\bullet\beta]) = \langle \pi, \beta \cdot \delta \rangle$ where
 $\langle \pi, \delta \rangle =$ *long-reduction′* $(\Gamma, [A \to \alpha\bullet\beta])$

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

 long-reduction′ $(\Gamma \cdot \langle [\$\gamma X], \pi(X) \rangle, [A \to \bullet\beta]) = \langle A \longrightarrow \beta \cdot \pi, \delta \cdot \beta' \rangle$
  where $[A' \to \alpha' \bullet A\beta'] =$ LPP$([A \to \bullet\beta], [\$\gamma X])$
   $\langle \pi, \delta \rangle =$ *long-reduction′* $(\Gamma \cdot \langle [\$\gamma X], \pi(X) \rangle, [A' \to \alpha' \bullet A\beta'])$

 long-reduction′ $(\Gamma \cdot \langle [\$\gamma X'], \pi(X') \rangle \cdot \langle [\$\gamma X' X], \pi(X) \rangle, [A \to \alpha\bullet\beta]) = \langle \pi(X) \cdot \pi, \delta \rangle$
  where $\langle \pi, \delta \rangle =$ *long-reduction′* $(\Gamma \cdot \langle [\$\gamma X'], \pi(X') \rangle,$ LPP$([A \to \alpha\bullet\beta], [\$\gamma X]))$

Algorithm 3 Embedded left $\mathrm{LR}(k)$ parsing.

1: let $q \in Q_k^G$ denote the topmost state
2: let $x \in (T \cup \{\$\})^{*k}$ denote the LA buffer contents
3: **while** $(i \leftarrow \text{LEFT}(q, x)) = \bot$ **do**
4: perform a step of the Schmeiser-Barnard $\mathrm{LR}(k)$ parser
5: **end while**
6: $\langle \pi, \delta \rangle \leftarrow$ *long-reduction* (stack, i)
7: PRINT π
8: **return** δ

If compared with the similar method used by the left $\mathrm{LR}(k)$ 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 $\mathrm{LL}(k)$ parser.

Finally, for the sake of completeness, the sketch of the embedded left $\mathrm{LR}(k)$ parser is given as Algorithm 3: in essence, it is a Schmeiser-Barnard $\mathrm{LR}(k)$ 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 $[\$S_2] = \{[S_1 \to S_2 \bullet x, \$]\}$ where $\text{LEFT}([\$S_2], \$) = [S_1 \to S_2 \bullet x]$ causing it to exit the loop in lines 3–5.

5. The embedded left $\mathrm{LR}(k)$ parser

The *embedded left* $\mathrm{LR}(k)$ *parser* is the left $\mathrm{LR}(k)$ parser for the embedded grammar (with a modified mapping 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 $\mathrm{LL}(k)$ parsing and $\mathrm{LR}(k)$ parsing is asymptotically as fast as $\mathrm{LR}(k)$ parsing, and the second states that it is just as powerful as $\mathrm{LR}(k)$ parsing.

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

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

Each production in π is either produced by the backbone $\mathrm{LL}(k)$ parser or reduced upon by one of the embedded left $\mathrm{LR}(k)$ 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 $A \longrightarrow \alpha$ (but $|\alpha|$ is bounded by a constant depending on the grammar only). Hence the $\mathcal{O}(|\pi|)$ part. ■

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

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

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

By repeatedly applying this trick all $\mathrm{LL}(k)$ conflicts get resolved — if not otherwise, when the initial symbol of $\bar{G}$ is declared to be an $\mathrm{LL}(k)$-conflicting symbol (note that the embedded left $\mathrm{LR}(k)$ 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 $\mathrm{LL}(k)$ conflicts with embedded left $\mathrm{LR}(k)$ parsers. Namely, if embedded left $\mathrm{LR}(k)$ parsers are triggered for $\mathrm{LL}(k)$ conflicting nonterminals deriving relatively short substrings, then employing embedded left $\mathrm{LR}(k)$ 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 $\mathrm{LR}(k)$ 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 $\mathrm{LR}(k)$ parser and the method loses much of its appeal (to the point that perhaps the left $\mathrm{LR}(k)$ parser is more suitable [20]).

6. Conclusion

The embedded left $\mathrm{LR}(k)$ parser has been obtained by modifying the left $\mathrm{LR}(k)$ parser in two ways. First, the left $\mathrm{LR}(k)$ 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 $\mathrm{LR}(k)$ 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 $\mathrm{LA}(k)\mathrm{LL}(\ell)$ parser (where $k \geq \ell$) as defined in [17], then the combined parsing could most probably be formulated as the combination of $\mathrm{LL}(\ell)$ and $\mathrm{LR}(k)$ parsing (note that $\mathrm{LL}(\ell) \subseteq \mathrm{LA}(\ell')\mathrm{LL}(\ell)$ for any $\ell' \geq \ell$). This would make the combined parser even more memory efficient.

The left $\mathrm{LR}(k)$ parser could be based on the $\mathrm{LA}(k)\mathrm{LR}(\ell)$ parser (most likely for $\ell = 0$) instead of on the canonical $\mathrm{LR}(k)$ parser. This would further reduce the parsing tables while the strength of the resulting combined parser would be reduced from $\mathrm{LR}(k)$ to $\mathrm{LA}(k)LR(\ell)$: not a significant issue as today $\mathrm{LA}(1)\mathrm{LR}(0)$ is used instead of $\mathrm{LR}(1)$ whenever LR parsing is applied.

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

Finally, apart from using the embedded left $\mathrm{LR}(k)$ parser for $\mathrm{LL}(k)$ conflict resolution, the embedded left $\mathrm{LR}(k)$ 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

1. Aycock, J., Horspool, N., Janoušek, J., Melichar, B.: Even faster generalized LR parsing. Acta Informatica 37(9), 633–651 (2001)
2. Boyland, J., Spiewak, D.: TOOL PAPER: ScalaBison recursive ascent-descent parser generator. Electronic Notes in Theoretical Computer Science 253(7), 65–74 (2010)
3. Demers, A.J.: Generalized left corner parsing. In: Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages POPL'77. pp. 170–182. ACM, ACM, Los Angeles, CA, USA (1977)
4. Ford, B.: Parsing expression grammars: a recognition-based syntactic foundation. In: Proceedings of the 31st ACM SIGACT-SIGPLAN symposium on Principles of programming languages POPL'04. pp. 111–122. ACM, ACM, Venice, Italy (2004)
5. Hopcroft, J.E., Ullman, J.D.: Introduction to Automata Theory, Languages, and Computation. Addison-Wesley, Reading, MA, USA (1979)
6. Horspool, R.N.: Recursive ascent-descent parsers. In: Hammer, D. (ed.) Compiler Compilers, Third International Workshop CC '90, Schwerin, FRG, Lecture Notes in Computer Science, vol. 477, pp. 1–10. Springer-Verlag (1990)
7. Knuth, D.E.: On the translation of languages from left to right. Information and Control 8(6), 607–639 (1965)
8. Lewis II, P.M., Stearns, R.E.: Syntax directed transduction. In: Proceedings of the 7th Annual Symposion on Switching and Automata Theory (SWAT'66). pp. 21–35. IEEE Computer Society Press, Berkeley, CA, USA (1966)
9. Mernik, M., Heering, J., Sloane, A.M.: When and how to develop domain-specific languages. ACM Computing Surveys 37(4), 316–344 (2005)
10. Might, M., Darais, D.: Yacc is dead. Available online at Cornell University Library (arXiv.org:1010.5023) (2010)
11. Parr, T., Fischer, K.: LL(*): The foundation of the ANTLR parser generator. ACM SIGPLAN Notices - PLDI'10 46(6), 425–436 (2011)

12. Rosenkrantz, D.J., Lewis, P.M.: Deterministic left corner parsing. In: Proceedings of the 11th Annual Symposium on Switching and Automata Theory (SWAT 1970). pp. 139–152. IEEE Computer Society, Washington, DC, USA (1970)
13. Schmeiser, J.P., Barnard, D.T.: Producing a top-down parse order with bottom-up parsing. Information Processing Letters 54(6), 323–326 (1995)
14. Scott, E., Johnstone, A.: GLL parsing. Electronic Notes in Theoretical Computer Science 253(7), 177–189 (2010)
15. Scott, E., Johnstone, A., Economopoulos, R.: BRNGLR: a cubic Tomita-style GLR parsing algorithm. Acta Informatica 44(6), 427–461 (2007)
16. Sippu, S., Soisalon-Soininen, E.: Parsing Theory, Volume I: Languages and Parsing, EATCS Monographs on Theoretical Computer Science, vol. 15. Springer-Verlag, Berlin, Germany (1988)
17. Sippu, S., Soisalon-Soininen, E.: Parsing Theory, Volume II: LR(k) and LL(k) Parsing, EATCS Monographs on Theoretical Computer Science, vol. 20. Springer-Verlag, Berlin, Germany (1990)
18. Slivnik, B.: The embedded left LR parser. In: Proceedings of the Federated Conference on Computer Science and Information Systems. pp. 871–878. IEEE Computer Society Press, Szczecin, Poland (2011)
19. Slivnik, B., Vilfan, B.: Improved error recovery in generated LR parsers. Informatica 28(3), 257–263 (2004)
20. Slivnik, B., Vilfan, B.: Producing the left parse during bottom-up parsing. Information Processing Letters 96(6), 220–224 (2005)
21. Tomita, M.: Efficient Parsing for Natural Language. Kluwer Academic Publisher, Boston, MA, USA (1985)
22. Tomita, M. (ed.): Generalized LR Parsing. Springer-Verlag, Berlin, Germany (1991)

Boštjan Slivnik 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.