Proceedings of the Federated Conference on
Computer Science and Information Systems pp. 863–870

ISBN 978-83-60810-22-4

The embedded left LR parser

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

***Abstract*—A parser called the embedded left LR(k) parser is defined. It 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. It is aimed at automatic construction of LL(k) parsers that use embedded LR(k) parsers to resolve LL(k) conflicts. 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. As the embedded LR(k) parser produces the prefix of the left parse, the LL(k) parser augmented with embedded 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.**

I. Introduction

Parsing is an important phase of virtually any modern compiler because it represents the backbone upon which syntax-directed translation of the source program to the (intermediate) code is based. Furthermore, syntax errors in the source program can be successfully detected and precisely reported only if the appropriate parsing method is chosen.

The two most widely used parsing methods nowadays, i.e., LL and LR parsing [1], [2], are both relatively old [3], [4]. Nevertheless, the discourse on whether LL or LR parsing is more suitable either in general or in some particular case still goes on decades later after both methods have been simplified or strengthen many times since their discovery.

By careful examination of open source compilers for the most popular programming languages one can conclude only that the race between LL (most often implemented as a hand-written recursive-descent parser) and LR parsing remains open. For instance, Sun/Oracle Java compiler (a part of the standard JDK) employs a recursive-descent parser augmented with operator precedence parsers while Eclipse Java compiler uses Jikes-generated LALR parser. Likewise, the distribution of Google's Go includes two parsers, a recursive-descent one and a Bison-generated LALR one. GCC switched from Bison-generated LALR parser to the recursive-descent parser for parsing C++ in 2004 (gcc 3.4.0) and for C/ObjectiveC in 2006 (gcc 4.1.0). Python is parsed using a hand-written LL(1) parser (augmented with DFAs to select the next production at each step), but Ruby and PHP are parsed using Bison-generated LALR parser. Finally, Haskell is parsed using Happy-generated LALR parser (GHC and JHC) or recursive-descent parser (NHC). (No citation is given in this paragraph: the findings can be best verified by downloading and examining the appropriate source code.)

The latest spark in this ongoing debate was triggered by the online publication of the paper entitled "Yacc is dead" [5]. Although the authors's original intent was to popularize a new parsing method, the online discourse quickly focused on whether it is better to use (mostly LALR) parser generators or write recursive-descent parsers by hand. As it might have been expected, no definite conclusion has been reached. However, two issues have been made clear (again). First, parser generators are appreciated, and second, both methods, LR and LL, remain attractive.

On one hand, LR parsing is popular for two reasons. First, unlike LL parsing, it is powerful: all deterministic context-free languages (DCFL) can be parsed using this method, and left-recursive productions (necessary for describing the left associativity of arithmetic operators, for instance) can be used. Second, for nearly every widespread programming language, an LALR parser generator is available (itself a consequence of the first reason).

On the other hand, the popularity of LL parsing stems from its simplicity which makes it suitable even for hand-written recursive-descent parsers, and its error recovery capability that allows generating precise error messages. Many LL parser generators are available, but quite a few include some way of producing parsers beyond the strength of LL(1) parsing: ANTLR employs LL(*) parsing [6] while LISA offers both LL and LR parsing (but the generated parser uses either one method or the other, but never both) [7].

Apart from some major modifications of LL parsing like LL(*) parsing [6], different techniques are used to bolster LL parsing. One way is to augment an LL(1) parser with DFAs (Python). Another way is to use small simple or operator precedence parsers [1] for parsing those phrases of the language (usually declarations or arithmetic expressions) that are too complicated for LL(1) parsing. However, none of these ways make parsing of all DCFL possible.

In this paper, we present yet another way to make LL parsing stronger: to use small LR parsers to resolve LL conflicts. Instead of the standard LR parser a modified LR parser which (a) produces the left parse and (b) stops as soon as the shortest prefix of the left parse can be computed, are to be used within the main LL parser. From the compiler writer's point of view the combined parser acts like a top-down parser capable of good error recovery [1], [10] while it

is as powerful as an LR parser since it can be constructed for any LR grammar.

An intermediate knowledge of LL and LR parsing is presumed. The notation used in [1] and [2] is adopted and all nonstandard symbols are introduced along the way. Furthermore, it is assumed that the result the parser produces is the *left (right) parse* of the input string, i.e., the list of productions needed to derive the input string from the initial grammar symbol using the leftmost (rightmost) derivation.

The paper is organized as follows. In Section II, the basic method for embedding $\mathrm{LR}(k)$ parsers into LL parsing is described and the embedded left $\mathrm{LR}(k)$ parser is formally defined in Sections III. The termination properties are investigated in Section IV. The paper ends with Conclusion containing a list of issues not cover in this paper due to the lack of space.

II. Embedding the $\mathrm{LR}(k)$ parser into the $\mathrm{LL}(k)$ parser

Consider that an $\mathrm{LL}(k)$ parser is being used for parsing a language generated by an $\mathrm{LR}(k)$ grammar, and that small $\mathrm{LR}(k)$ parsers are used to resolve $\mathrm{LL}(k)$ conflicts in the $\mathrm{LL}(k)$ parser. More precisely, let the backbone parser be an $\mathrm{SLL}(k)$, i.e., strong $\mathrm{LL}(k)$, parser. There are several reasons for using $\mathrm{SLL}(k)$ parser instead of the canonical $\mathrm{LL}(k)$ parser [1], [2]. First, the construction and implementation of the $\mathrm{SLL}(k)$ parser are much simpler and memory efficient than that of the canonical $\mathrm{LL}(k)$ parser. Second, every $\mathrm{LL}(k)$ grammar can be transformed into an equivalent $\mathrm{SLL}(k)$ grammar automatically, so no expressive power is lost. And finally, when $k = 1$, the only value of k used in practice, $\mathrm{SLL}(1) = \mathrm{LL}(1)$.

Theoretically, the $\mathrm{SLL}(k)$ parser is a produce-shift parser and produces the left parse of the input string. For instance, after reading the prefix u of the input string $w = uv$ that is derived by the derivation

$$S \Longrightarrow^{\pi_u}_{G,\mathrm{lm}} u\delta \Longrightarrow^{\pi_v}_{G,\mathrm{lm}} uv = w \quad ,$$

the $\mathrm{SLL}(k)$ parser for $G = \langle N, T, P, S\rangle$ reaches configuration $\$\delta^R \mid v\$$ with viable suffix $\$\delta^R$ on the stack and lookahead string $x = k{:}\, v\$$ in the lookahead buffer; the parser's output contains the left parse $\pi_u \in P^*$. Furthermore, the parser is said to be in *position* $X \mid x$ if symbol X is the topmost stack symbol ($\$\delta^R = \$\delta'^R X$).

By theory [1], [2], the $\mathrm{SLL}(k)$ parser for G (based on the \$-augmented grammar G') exhibits produce-produce conflict in *conflicting position* $A \mid x$ if and only if there exist productions $A \longrightarrow \alpha_1, A \longrightarrow \alpha_2 \in P$ where $\alpha_1 \neq \alpha_2$ so that

$$x \in (\ \mathrm{FIRST}^{G'}_k(\alpha_1 \mathrm{FOLLOW}^{G'}_k(A)) \cap \mathrm{FIRST}^{G'}_k(\alpha_2 \mathrm{FOLLOW}^{G'}_k(A)))\quad .$$

As the conflicting position $A \mid x$ is the result of every production $B \longrightarrow \beta_1 A \beta_2$ where

$$x \in (\ \mathrm{FIRST}^{G'}_k(\alpha_1\beta_2 \mathrm{FOLLOW}^{G'}_k(B)) \cap \mathrm{FIRST}^{G'}_k(\alpha_2\beta_2 \mathrm{FOLLOW}^{G'}_k(B)))\quad ,$$

the basic idea (borrowed from the combination of LL and simple/operator precedence parsing) is to replace the production $B \longrightarrow \beta_1 A \beta_2$ with production $B \longrightarrow \beta_1 \langle\!\langle A \rangle\!\rangle \beta_2$ where $\langle\!\langle A \rangle\!\rangle \notin T$ is a marker that triggers an $\mathrm{LR}(k)$ parser for A.

However, the embedded parser for A cannot assume that the end-of-input marker (denoted \$) is at the end of the substring being parsed, i.e., the substring derived from A. It must stop when a string $x \in \mathrm{FIRST}^{G'}_k(\beta_2 \mathrm{FOLLOW}^{G'}_k(B))$ is in the lookahead buffer, and then handle the control back to the backbone $\mathrm{SLL}(k)$ parser. This is not always possible as the following two examples demonstrate.

Example 1: Consider the grammar G_{ex1} with productions

$$S \longrightarrow bAab,\ A \longrightarrow Aa \,|\, a \quad .$$

The position $A \mid a$ exhibits the conflict, but the $\mathrm{LR}(1)$ parser for a grammar with the new symbol A cannot stop in the right moment. Suppose that an input string starts with baa and that the $\mathrm{LR}(1)$ parser for A has been triggered in configuration $\$baA \mid aa \ldots \$$. After the first a is shifted and reduced to A, the lookahead buffer contains the second a. As $k = 1$, the $\mathrm{LR}(1)$ parser cannot decide whether the particular a in the buffer is derived from S (if $baab$ is being parsed) or from A (if $baaab$ is being parsed, for instance). The solution is to parse not just A but Aa using $\mathrm{LR}(1)$ parser as b is never a part of the input for this embedded LR parser and can thus stop on b. ■

Example 2: Consider the grammar G_{ex2} with productions

$$S \longrightarrow bBab \,|\, abBb,\ B \longrightarrow A,\ A \longrightarrow Ba \,|\, a \quad .$$

The conflicting position is again $A \mid a$. If the input string starts with baa, the $\mathrm{LR}(1)$ parser for a new start symbol A cannot stop correctly for the same reason as in Example 1. But now A is the rightmost symbol in production $A \longrightarrow B$ and thus the solution from Example 1 cannot be used. Hence, position $B \mid a$ must be declared conflicting instead.

However, if the input string starts with $abaa$, then the $\mathrm{LR}(1)$ parser for A used for resolving the conflict in position $A \mid a$ can stop on b — stopping of the embedded parsers clearly depends on the wider context within which the conflicting position occurs. ■

To avoid the problem of the context that is exposed in Example 2, the grammar $G = \langle N, T, P, S\rangle$ for which the $\mathrm{SLL}(k)$ parser using embedded $\mathrm{LR}(k)$ parsers is to be constructed, is transformed into grammar $\bar{G} = \langle \bar{N}, T, \bar{P}, \bar{S}\rangle$ where each nonterminal occurs in exactly one FOLLOW-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 $k > 1$ and $\mathrm{SLL}(k)$ parser is used instead of $\mathrm{LL}(k)$ parser, it even reduces the number of $\mathrm{LL}(k)$ conflicts for some non-$\mathrm{SLL}(k)$ grammars [2].)

To resolve the $\mathrm{SLL}(k)$ conflicts during $\mathrm{SLL}(k)$ parsing, every production

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

must be replaced with

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

where $\beta_2 = \beta_2'\beta_2''$ and $\mathcal{F}_{A\beta_1'} = \mathrm{FIRST}_k^G(\beta_2''\mathcal{F}_B)$. The new symbol $\langle\!\langle A\beta_2', \mathcal{F}_{A\beta_2'} \rangle\!\rangle \notin \bar{N}$ acts as a trigger for starting the embedded $\mathrm{LR}(k)$ parser for substrings derived from $A\beta_2'$ that can stop on strings in $\mathcal{F}_{A\beta_2'}$.

Furthermore, as the amount of LR parsing is to be minimal, β_2' should be as short as possible or even ε in the best case. If, on the other hand, not even $\beta_2'' = \varepsilon$ suffices for the safe termination of the embedded $\mathrm{LR}(k)$ parser, $\langle B, \mathcal{F}_B \rangle$ must be declared conflicting nonterminal.

Example 3: Using the transformation described just above, grammar G_{ex1} is transformed to grammar $\bar{G}_{\mathrm{ex1}}$ with a single production

$$\langle S, \{\varepsilon\} \rangle \longrightarrow b\langle\!\langle Ab, \{b\} \rangle\!\rangle b \quad .$$

Likewise, grammar G_{ex2} is transformed to grammar $\bar{G}_{\mathrm{ex2}}$ with productions

$$\langle S, \{\varepsilon\} \rangle \longrightarrow b\langle\!\langle Ba, \{b\} \rangle\!\rangle b \,|\, ba\langle\!\langle B, \{b\} \rangle\!\rangle b$$

despite the fact that symbol B is not part of any conflicting position. ■

Finally, if marker $\langle\!\langle \beta, \mathcal{F} \rangle\!\rangle$ is given for grammar $G = \langle N, T, P, S \rangle$, an embedded $\mathrm{LR}(k)$ parser that stops (no later than) 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*

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

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* $\hat{G}$ must accept its input no later than when the reduction on $S_2 \longrightarrow \beta$ is due. In that way, it never pushes any symbol of any string $x \in \mathcal{F}$ onto the stack. If this cannot be done, the embedded $\mathrm{LR}(k)$ parser for $\langle\!\langle \beta, \mathcal{F} \rangle\!\rangle$ cannot be used.

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

As mentioned in the introduction, the left $\mathrm{LR}(k)$ parser that is embedded into the backbone $\mathrm{LL}(k)$ parser must fulfill two requirements. First, it must produce the prefix of the left parse instead of the right parse so that the compiler writer can concentrate on the implementation of attribute grammar as if the entire input is being parsed using $\mathrm{LL}(k)$ parser. Second, it must stop and handle the control to the backbone parser as soon as possible, preferably after the first production of the left parse is produced. In that way, the amount of $\mathrm{LR}(k)$ parsing is minimized. Furthermore, the embedded left $\mathrm{LR}(k)$ parser must be able to accept its input and terminate without the end-of-input marker in the lookahead buffer since it parses only a substring of the entire input — but this issue has been resolved in the previous section.

To meet these two goals, the embedded left $\mathrm{LR}(k)$ parser is a modification of the *left* $\mathrm{LR}(k)$ *parser*. The left $\mathrm{LR}(k)$ parser produces the left parse of the input string during bottom-up parsing using two methods [8].

The first method, first introduced in the Schmeiser-Barnard $\mathrm{LR}(k)$ parser [9], augments each nonterminal pushed on the LR stack with the left parse of the substring derived from that nonterminal:

- 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 that are 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 using this method, the first production of the left parse is produced only at the very end of parsing.

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

$$S \Longrightarrow_{G,\mathrm{rm}}^{*} \gamma v \Longrightarrow_{G,\mathrm{rm}}^{*} uv \quad . \tag{1}$$

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

$$\$[\$][\$X_1][\$X_1X_2] \ldots [\$X_1X_2 \ldots X_n] \,|\, v\$ \tag{2}$$

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

On the other hand, the Schmeiser-Barnard $\mathrm{LR}(k)$ parser (which is based on the canonical $\mathrm{LR}(k)$ machine as well) reaches the configuration

$$\begin{aligned} \$\langle[\$]; \varepsilon\rangle \langle[\$X_1]; \pi(X_1)\rangle \langle[\$X_1X_2]; \pi(X_2)\rangle \ldots \\ \ldots \langle[\$X_1X_2 \ldots X_n]; \pi(X_n)\rangle \,|\, v\$ \end{aligned} \tag{3}$$

where $\pi(X_j)$ denote the left parse of the substring derived from X_j, i.e., $X_1X_2 \ldots X_n \Longrightarrow_{G,\mathrm{lm}}^{\pi(X_1)\pi(X_2)\ldots\pi(X_n)} u$.

Example 4: Consider the embedded grammar G_{ex4} with productions

$$\begin{gathered} S_1 \longrightarrow S_2c,\; S_2 \longrightarrow A, \\ A \longrightarrow aa \,|\, aB \,|\, bBa \,|\, bBaa,\; B \longrightarrow Bb \,|\, \varepsilon \quad . \end{gathered}$$

Parsing of the input string *bbbaac* using the Schmeiser-Barnard $\mathrm{LR}(1)$ parser is shown in Table I. ■

TABLE I
PARSING THE STRING $bbbaac \in L(G_{\mathrm{ex4}})$ USING THE SCHMEISER-BARNARD LR(1) PARSER.

STACK	INPUT
$\$\ \langle[\$];\varepsilon\rangle$	$bbbaac\$$
$\$\ \langle[\$];\varepsilon\rangle\ \langle[\$b];\varepsilon\rangle$	$bbaac\$$
$\$\ \langle[\$];\varepsilon\rangle\ \langle[\$b];\varepsilon\rangle\ \langle[\$bB];\pi_1 = B\to\varepsilon\rangle$	$bbaac\$$
$\$\ \langle[\$];\varepsilon\rangle\ \langle[\$b];\varepsilon\rangle\ \langle[\$bB];\pi_1 = B\to\varepsilon\rangle\ \langle[\$bBb];\varepsilon\rangle$	$baac\$$
$\$\ \langle[\$];\varepsilon\rangle\ \langle[\$b];\varepsilon\rangle\ \langle[\$bB];\pi_2 = B\to Bb\cdot\pi_1\rangle\ \langle[\$bBb];\varepsilon\rangle$	$aac\$$
$\$\ \langle[\$];\varepsilon\rangle\ \langle[\$b];\varepsilon\rangle\ \langle[\$bB];\pi_3 = B\to Bb\cdot\pi_2\rangle$	$aac\$$
$\$\ \langle[\$];\varepsilon\rangle\ \langle[\$b];\varepsilon\rangle\ \langle[\$bB];\pi_3 = B\to Bb\cdot\pi_2\rangle\ \langle[\$bBa];\varepsilon\rangle$	$ac\$$
$\$\ \ldots\ \langle[\$bB];\pi_3 = B\to Bb\cdot\pi_2\rangle\ \langle[\$bBa];\varepsilon\rangle\ \langle[\$bBaa];\varepsilon\rangle$	$c\$$
$\$\ \langle[\$];\varepsilon\rangle\ \langle[\$A];\pi_4 = A\to bBaa\cdot\pi_3\rangle$	$c\$$
$\$\ \langle[\$];\varepsilon\rangle\ \langle[\$S_2];\pi_5 = S_2\to A\cdot\pi_4\rangle$	$c\$$
$\$\ \langle[\$];\varepsilon\rangle\ \langle[\$S_2];\pi_6 = S_2\to A\cdot\pi_5\rangle\ \langle[\$S_2c],\varepsilon\rangle$	$\$$
$\$\ \langle[\$];\varepsilon\rangle\ \langle[\$S_1];\pi_7 = S_1\to S_2c\cdot\pi_6\rangle$	$\$$

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

The second method, first introduced in the left $\mathrm{LR}(k)$ parser [8], enables the parser to compute the prefix of the left parse of the substring corresponding to the prefix of the input string read so far (although this is not possible in every parser configuration). In other words, if apart from derivation (1) the input string $w = uv$ is derived by the leftmost derivation

$$S \Longrightarrow_{G,\mathrm{lm}}^{\pi(u)} u\delta \Longrightarrow_{G,\mathrm{lm}}^{*} uv \quad , \tag{4}$$

then the left $\mathrm{LR}(k)$ parser computes the left parse $\pi(u)$ in configuration (3). As this part of the left $\mathrm{LR}(k)$ parser is modified, it deserves more attention.

By theory [2], configurations (2) and (3) imply that machine M_G contains at least one sequence of valid k-items

$$\begin{aligned}&[A_0 \to \bullet\alpha_0 A_1\beta_0, x_0]\cdot\ldots\cdot[A_0\to\alpha_0\bullet A_1\beta_0, x_0]\cdot\\ &\cdot[A_1\to\bullet\alpha_1 A_2\beta_1, x_1]\cdot\ldots\cdot[A_1\to\alpha_1\bullet A_2\beta_1, x_1]\cdot\\ &\qquad\qquad\vdots\\ &\cdot[A_\ell\to\bullet\alpha_\ell A_{\ell+1}\beta_\ell, x_\ell]\ldots[A_\ell\to\alpha_\ell\bullet A_{\ell+1}\beta_\ell, x_\ell]\end{aligned} \tag{5}$$

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

Sequence (5) induces the *(induced) central derivation*

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

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

However, if the left parses $\pi(\alpha_0), \pi(\alpha_1), \ldots, \pi(\alpha_\ell)$, where $\alpha_j \Longrightarrow_{G',\mathrm{lm}}^{\pi(\alpha_j)} u_j$ for $j = 0, 1, \ldots, \ell$, are provided, then sequence (5) induces the *(induced) leftmost derivation*

$$\begin{aligned}S' = A_0 &\Longrightarrow_{G,\mathrm{lm}} \alpha_0 A_1\beta_0 \Longrightarrow_{G,\mathrm{lm}}^{\pi(\alpha_0)} u_0 A_1\beta_0\\ &\Longrightarrow_{G,\mathrm{lm}} u_0\alpha_1 A_2\beta_1\beta_0 \Longrightarrow_{G,\mathrm{lm}}^{\pi(\alpha_1)} u_0u_1 A_2\beta_1\beta_0\\ &\ \ \vdots\\ &\Longrightarrow_{G,\mathrm{lm}} u_0u_1\ldots u_{\ell-1}\alpha_\ell A_{\ell+1}\beta_\ell\beta_{\ell-1}\ldots\beta_0\\ &\qquad\Longrightarrow_{G,\mathrm{lm}}^{\pi(\alpha_\ell)} u_0u_1\ldots u_\ell A_{\ell+1}\beta_\ell\beta_{\ell-1}\ldots\beta_0\end{aligned}$$

where $u = u_0u_1\ldots u_\ell$ and $k{:}\, v\$ \in \mathrm{FIRST}_k^{G'}(\beta_\ell\beta_{\ell-1}\ldots\beta_0\$)$. The corresponding derivation tree is shown in Figure 1(b) and the left parse of the induced leftmost derivation is therefore

$$\begin{aligned}\pi(u) = A_0 \longrightarrow \alpha_0 A_1\beta_0\cdot\pi(\alpha_0)\cdot\\ \cdot A_1 \longrightarrow \alpha_1 A_2\beta_1\cdot\pi(\alpha_1)\cdot\\ \vdots\\ \cdot A_\ell \longrightarrow \alpha_\ell A_{\ell+1}\beta_\ell]\cdot\pi(\alpha_\ell) \quad .\end{aligned} \tag{6}$$

(Likewise, if the right parses $\pi(\beta_1), \pi(\beta_2), \ldots, \pi(\beta_\ell)$ are known, then sequence (5) induces the *(induced) rightmost derivation* producing the derivation tree in Figure 1(c).)

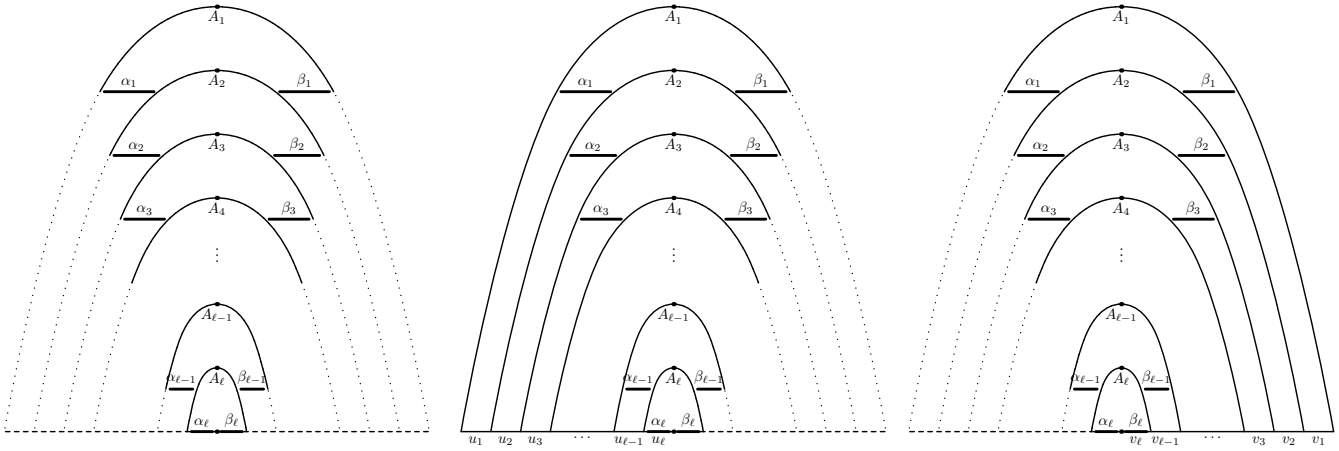
Subparses $\pi(\alpha_j)$ of the left parse (6) are available on the parser stack because $\alpha_0\alpha_1\ldots\alpha_\ell = \gamma = X_1X_2\ldots X_n$, but productions $A_j \longrightarrow \alpha_j A_{j+1}\beta_j$ are not. However, if sequence (5) is known, the missing productions and in fact the entire prefix of the left parse can be computed [8]. 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 (5), 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 (4) 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 (5).

Example 5: Consider again grammar G_{ex4} and the input string $bbbaac \in L(G_{\mathrm{ex4}})$ from Example 4. After the prefix $bbba$ of the input string has been read, the parser reaches the configuration shown in the 7th line of Table I. But as illustrated in Figure 2, there is only one item active for the current lookahead string a in state $[\$bBa]$, namely $[S_2 \to bBa\bullet a, \$]$. Furthermore, there exist exactly one sequence of $\mathrm{LR}(1)$ items starting with $[S' \to \bullet\$S\$, \varepsilon] \in [\varepsilon]$ and ending



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

(b) The derivation tree of the induced leftmost derivation (the left parses π_{α_j} ought to be provided).

(c) The derivation tree of the induced rightmost derivation (the right parses π_{β_j} ought to be provided).

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

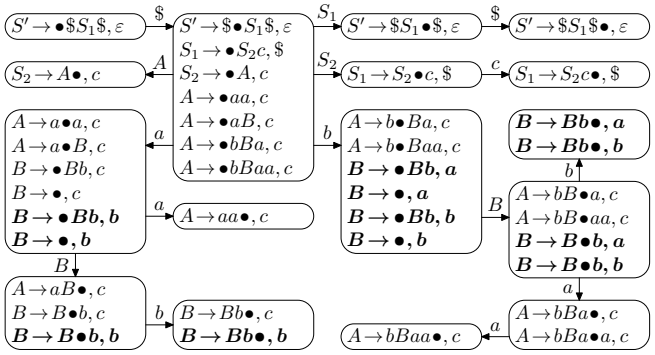


Fig. 2. The canonical $\mathrm{LR}(1)$ machine for G_{ex4} — items that end multiple sequences starting with $[S' \to \bullet\$S\$, \varepsilon] \in [\varepsilon]$ are shown in bold face.

with $[S_2 \to bBa\bullet a, \$] \in [\$bS_2a]$:

$$[S' \to \bullet\$S_1\$, \varepsilon] \cdot [S' \to \$\bullet S_1\$, \varepsilon] \cdot [S_1 \to \bullet S_2c, \$] \cdot \\ \cdot [S_2 \to \bullet bBaa, \$] \cdot [S_2 \to b\bullet Baa, \$] \cdot \\ \cdot [S_2 \to bB\bullet aa, \$] \cdot [S_2 \to bBa\bullet a, \$]$$

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

In general, cases where exactly one sequence (5) exists (as in Example 5) are extremely rare, but all sequences (5) 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 $\mathrm{LR}(k)$ 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 (5) 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 (5) 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 [8]. But as (a)

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

(note that the embedded grammar is being used) and (b) 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 5 is used, the embedded $\mathrm{LR}(k)$ 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

- $[A_\ell \to \bullet A_{\ell+1}\beta_\ell]$ if all sequences (5) 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

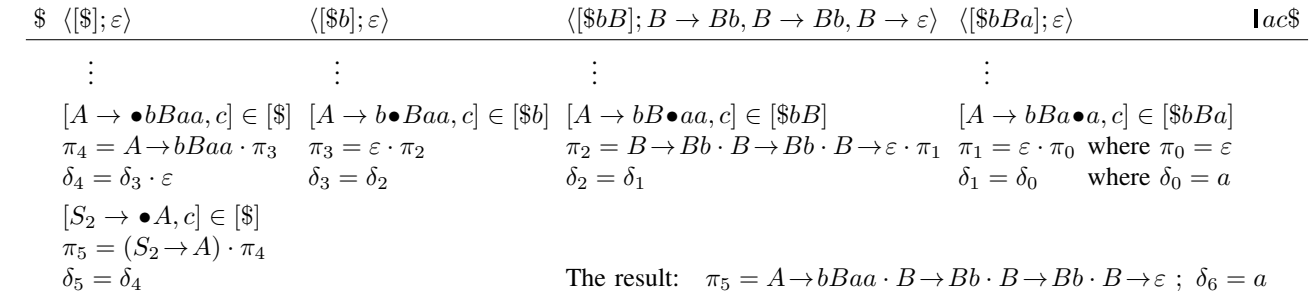


Fig. 3. Computing the prefix of the left parse of the string $bbbaac \in L(G_{\mathrm{ex4}})$ after *bbba* has been read: the computation starts at the top of the stack (right side of the figure) with $\pi_0 = \varepsilon$ and $\delta_0 = a$, and traverses the stack downwards (towards the left side of the figure, and then downwards).

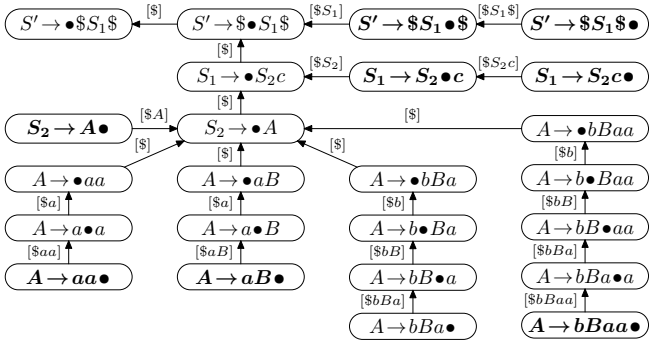


Fig. 4. The left-parse-prefix automaton for G_{ex4} — items that are not needed during embedded left LR(1) parsing are shown in bold face.

- $\perp$ 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 (5) with lookahead strings stripped off. Hence, $\text{LPP}(i_0, [\$\gamma]) = i_0'$ if and only if there exists some sequence (5) with two consecutive 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 6: The left-parse-prefix automaton for grammar G_{ex4} 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 some more complex language, the corresponding LPP gets more complicated — see [8].)

Mapping LEFT for G_{ex4} 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 $\text{LEFT}([\$], a) = \perp$ and $\text{LEFT}([\$], b) = \perp$ because of $A \longrightarrow aa|aB$ and $A \longrightarrow bBa|bBaa$, respectively. ■

The algorithms for computing LEFT and LPP can be found in [8]. 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 5 can be formalized as Algorithm 1. 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.

If compared with the similar method used by the left LR(k) parser [8], 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 2: in essence, it is a Schemiser-Barnard LR(k) parser [9] with the option of (a) premature termination and (b) computing the viable suffix.

Algorithm 2 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.

To conclude, the embedded left LR(k) parser is the left 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.

IV. The termination of the embedded left LR parser

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 $\hat{G} \in \text{LR}(k)$ 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.

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

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

Algorithm 1 *long-reduction*: computing the prefix of the left parse and the 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 $[A' \to \alpha'\bullet A\beta'] = \text{LPP}([A \to \bullet\beta], [\$\gamma X])$
$\langle \pi, \delta \rangle =$ *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*$'$ $(\Gamma \cdot \langle [\$\gamma X'], \pi(X') \rangle, \text{LPP}([A \to \alpha\bullet\beta], [\$\gamma X]))$

Algorithm 2 Embedded LR(k) parsing.

```
let q ∈ Q_k^G denote the topmost state
let x ∈ (T ∪ {$})^{*k} denote the LA buffer contents
while (i ← LEFT(q, x)) = ⊥ do
    perform a step of the Schmeiser-Barnard LR(k) parser
end while
⟨π, δ⟩ ← long-reduction (stack, i)
PRINT π
return δ
```

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

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

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

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

where $x' = k{:}\, x\$$ for some $x \in \text{FIRST}_k^G(\delta)$.

Proof: First, the structure of grammar $\hat{G}$ implies that items $[S_1 \to \bullet S_2 x, \$]$ and $[S_2 \to \bullet\beta_2, x']$, where $x' = k{:}\, x\$$, appear in the state $[\$]_{\hat{G}} = \mathbf{desc}^*(\{[S' \to \$\bullet S_1\$, \varepsilon]\})$ only. Hence, any item $[S_1 \to \psi_1\bullet\psi_2, \$]$ or $[S_2 \to \psi_1\bullet\psi_2, x']$ can appear in state $[\$\psi_1]_{\hat{G}}$ only.

Second, because of the leftmost derivation above, there exists the rightmost derivation

$$S \Longrightarrow^*_{G,\text{rm}} \gamma B v \Longrightarrow_{G,\text{rm}} \gamma\beta_1\beta_2 v$$

and thus

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

where $[\$\gamma\beta_1]_G$ is a state of the canonical LR(k) machine for grammar G. Therefore,

- $[S_1 \to \bullet S_2 x, \$] \in [\$]_{\hat{G}}$ implies $[B \to \beta_1\bullet\beta_2, x'] \in [\$\gamma\beta_1]_G$ where $x' = k{:}\, x\$$, and
- $[S_2 \to \hat{\gamma}\bullet\psi, x'] \in [\$\hat{\gamma}]_{\hat{G}}$ implies $[B \to \beta_1\hat{\gamma}\bullet\psi, x'] \in [\$\gamma\beta_1\hat{\gamma}]_G$.

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 LR(k) conflict) in any state $[\$\hat{\gamma}]_{\hat{G}}$ of the canonical LR(k) machine for $\hat{G}$, i.e., $i_1, i_2 \in [\$\hat{\gamma}]_{\hat{G}}$:

- 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 LR(k) conflict between i_1 and i_2 since $G \in LR(k)$.
- If i_1 and i_2 are based on productions in $\hat{P} \setminus P$, three cases must be considered:
  - If $i_1 = [S_1 \to \hat{\gamma}\bullet\alpha, \$]$ and $i_2 = [S_1 \to \hat{\gamma}\bullet\alpha', \$]$, then either $\hat{\gamma} = \varepsilon$ and both items imply the shift action since $\alpha, \alpha' \neq \varepsilon$ or $k{:}\, \alpha\$ \neq k{:}\, \alpha'\$$ so no conflict is possible.
  - If $i_1 = [S_1 \to \hat{\gamma}\bullet\alpha, \$]$ and $i_2 = [S_2 \to \hat{\gamma}\bullet\alpha', y']$ (or vice-versa), then $\hat{\gamma} = \varepsilon$ (otherwise $\hat{\gamma} = S_2\hat{\gamma}'$ because of i_1 but $\hat{\gamma} \neq S_2\hat{\gamma}'$ because of i_2) and both items imply the shift action since $\alpha, \alpha' \neq \varepsilon$.
  - If $i_1 = [S_2 \to \hat{\gamma}\bullet\alpha, y]$ and $i_2 = [S_2 \to \hat{\gamma}\bullet\alpha', y']$, then $\alpha = \alpha'$ and both items imply either the reduce action on $S_2 \longrightarrow \beta_2$ or the shift action.
- 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), two cases must be considered:
  - If $i_1 = [S_1 \to \hat{\gamma}_1\hat{\gamma}_2\bullet\alpha, \$]$ and $i_2 = [A \to \hat{\gamma}_2\bullet\alpha', y']$, then obviously $\hat{\gamma}_1\hat{\gamma}_2 = \varepsilon$ and $\alpha = S_2 x$ for some $x \in \text{FIRST}_k^G(\delta)$ (otherwise $\hat{\gamma}_1\hat{\gamma}_2 = S_2\hat{\gamma}'$ because of i_1 but $\hat{\gamma}_1\hat{\gamma}_2 \neq S_2\hat{\gamma}'$ because of i_2). Thus

    $$[S_1 \to \bullet S_1 x, \$], [A \to \bullet\alpha', y'] \in [\$]_{\hat{G}}$$

    implies

    $$[B \to \beta_1\bullet\beta_2, x'], [A \to \bullet\alpha', y'] \in [\$\gamma\beta_1]_G$$

    where $x' = k{:}\, x\$$. But as

    $$\text{FIRST}_k^{\hat{G}'}(S_2 x\$) = \text{FIRST}_k^{G'}(\beta_2 x')$$

    and no items in $[\$\gamma\beta_1]_G$ exhibit any LR(k) conflict, the items $[S_1 \to \bullet S_1 x, \$]$ and $[A \to \bullet\alpha', y']$ do not exhibit LR(k) conflict either.
  - If $i_1 = [S_2 \to \hat{\gamma}_1\hat{\gamma}_2\bullet\alpha, x']$ and $i_2 = [A \to \hat{\gamma}_2\bullet\alpha', y']$ where $x' = k{:}\, x\$$ for some $x \in \text{FIRST}_k^{\hat{G}'}(\delta)$, then $\hat{\gamma}_1\hat{\gamma}_2 = \beta_2$ because of i_1 and $[B \to \beta_1\hat{\gamma}_1\hat{\gamma}_2\bullet\alpha, x'], [A \to \hat{\gamma}_2\bullet\alpha', y'] \in [\$\gamma\beta_1\hat{\gamma}_1\hat{\gamma}_2]_G$.
    If $\alpha \neq \varepsilon$ and $\alpha' \neq \varepsilon$, then items i_1 and i_2 both imply the shift action.
    If $\alpha \neq \varepsilon$ but $\alpha' = \varepsilon$, then $y' \notin \text{FIRST}_k^{\hat{G}'}(\alpha x')$ as otherwise $[B \to \beta_1\hat{\gamma}_1\hat{\gamma}_2\bullet\alpha, x']$ and $[A \to \hat{\gamma}_2\bullet, y']$

would exhibit a shift-reduce conflict; hence, items i_1 and i_2 do not exhibit any $\mathrm{LR}(k)$ conflict.
If $\alpha' \neq \varepsilon$ but $\alpha = \varepsilon$, then $x' \notin \mathrm{FIRST}_k^{\hat{G}'}(\alpha' y')$ as otherwise $[B \to \beta_1 \hat{\gamma}_1 \hat{\gamma}_2 \bullet, x']$ and $[A \to \hat{\gamma}_2 \bullet \alpha', y']$ would exhibit a reduce-shift conflict; hence, items i_1 and i_2 do not exhibit any $\mathrm{LR}(k)$ conflict.
If $\alpha = \varepsilon$ and $\alpha' = \varepsilon$, then

$$[S_2 \to \beta_2 \bullet, x'], [A \to \hat{\gamma}_2 \bullet, y'] \in [\$\beta_2]_{\hat{G}}$$

implies

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

Therefore, if and only if

$$[B \to \beta_1 \beta_2 \bullet, x'] = [A \to \hat{\gamma}_2 \bullet, y']$$

where $\hat{\gamma}_2 = \beta_1\beta_2$ (and thus $\beta_1 = \varepsilon$) can there be a (reduce-reduce) conflict in $[\$\beta_2]_{\hat{G}}$ without a conflict in $[\$\gamma\beta_1\beta_2]_G$.

As determined, the only possibility for an $\mathrm{LR}(k)$ conflict in the canonical $\mathrm{LR}(k)$ machine for $\hat{G}$ is the reduce-reduce conflict exhibited by items

$$[S_2 \to \beta \bullet, x'], [B \to \beta_2 \bullet, x'] \in [\$\beta_2]_{\hat{G}}$$

which are derived from items

$$[S_2 \to \bullet\beta, x'], [B \to \bullet\beta_2, x'] \in [\$]_{\hat{G}} \quad .$$

But as

$$\begin{aligned}[\$]_{\hat{G}} &= \mathbf{desc}^*\{[S' \to \$ \bullet S_1 \$, \varepsilon]\} \\ &= \{[S' \to \$ \bullet S_1 \$, \varepsilon]\} \\ &\quad \cup \{[S_1 \to \bullet S_2 x, \$] \ ; \ x \in \mathrm{FIRST}_k^G(\delta)\} \\ &\quad \cup \{[S_2 \to \bullet\beta_2, x'] \ ; \ x' \in \mathrm{FIRST}_k^{G'}(\delta\$)\} \\ &\quad \cup \mathbf{desc}^*\{[S_2 \to \bullet\beta_2, x'] \ ; \ x' \in \mathrm{FIRST}_k^{G'}(\delta\$)\} \quad ,\end{aligned}$$

item $[B \to \bullet\beta_2, x']$ can belong to the state $[\$]_{\hat{G}}$ only if $[S_2 \to \bullet\beta, x']$ $\mathbf{desc}^*$ $[B \to \bullet\beta_2, x']$.

Finally, proving the theorem in the opposite direction is trivial: items $[S_2 \to \bullet\beta_2, x']$ and $[A \to \bullet\beta_2, x']$ in $[\$]_{\hat{G}}$ imply a reduce-reduce conflict between items $[S_2 \to \beta_2 \bullet, x']$ and $[A \to \beta_2 \bullet, x']$ in $[\$\beta_2]_{\hat{G}}$, so $\hat{G} \notin \mathrm{LR}(k)$. ■

Theorem 1 provides two important insights into the embedded left $\mathrm{LR}(k)$ parsing. First, it guarantees that by simply checking the state $[\$\beta_2]$ for reduce-reduce conflicts one can tell whether the embedded left $\mathrm{LR}(k)$ parser for parsing substrings derived from the sentential form β_2 can be made. As the sentential form β_2 is a part of the right side of a production, it is usually short and therefore the method based on Theorem 1 is significantly faster than the brute-force approach.

Second, Theorem 1 illustrates that once again it is the left recursion that causes problems. But this is not to be worried about since it is clear that any substring derived from the left recursive nonterminal must be parsed entirely by an LR parser. In other words, Theorem 1 indicates that if the grammar is made so that the left recursive nonterminals are kept as low as possible in the resulting derivative trees, the substrings actually parsed using the embedded left $\mathrm{LR}(k)$ parsers tend to be short.

V. 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 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 [2], 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.

Furthermore, 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.

Finally, 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 [10].

References

[1] S. Sippu and E. Soisalon-Soininen, *Parsing Theory, Vol. I: Languages and Parsing*, Berlin Heidelberg, Germany: Springer-Verlag, 1988.

[2] S. Sippu and E. Soisalon-Soininen, *Parsing Theory, Vol. II: LL(k) and LR(k) Parsing*, Berlin Heidelberg, Germany: Springer-Verlag, 1990.

[3] D. E. Knuth, *On the translation of languages from left to right*, Information and Control (1965), vol. 8, no. 6, pp. 607–639.

[4] P. M. Lewis II and R. E. Stearns, *Syntax-directed transduction*, Proceedings of the 7th Annual IEEE Symposium on Switching and Automata Theory, New York, USA (1966), pp. 21-35.

[5] M. Might and D. Darais, *Yacc is dead*, available online at Cornell University Library (arXiv.org:1010.5023), 2009.

[6] T. Parr and K. S. Fischer, *LL(*): The Foundation of the ANTLR Parser Generator*, accepted at the 32nd ACM SIGPLAN conference PLDI 2011.

[7] P. R. Henriques, M. J. Varando Pereira, M. Mernik, M. Lenič, J. G. Gray, H. Wui, *Automatic generation of language-based tools using the LISA system*, IEE Proceedings - Software (2005), vol. 152, no. 2, pp. 54–69.

[8] B. Slivnik and B. Vilfan, *Producing the left parse during bottom-up parsing*, Information Processing Letters (2005), vol. 96, no. 6, pp. 220–224.

[9] J. P. Schmeiser and D. T. Barnard, *Producing a top-down parse order with bottom-up parsing*, Information Processing Letters (1995), vol. 54, no. 6, pp. 323–326.

[10] B. Slivnik and B. Vilfan, *Improved error recovery in generated LR parsers*, Informatica (2004), vol. 28, no. 3, pp. 257–263.