Improved Error Recovery in Generated LR Parsers

Boštjan Slivnik and Boštjan Vilfan
University of Ljubljana
Faculty of Computer and Information Science
Tržaška 25, 1000 Ljubljana, Slovenia
`bostjan.slivnik@fri.uni-lj.si`
`bostjan.vilfan@fri.uni-lj.si`

Keywords: LR parsing, error recovery and reporting

Received: February 17, 2004

A new method for error recovery in LR *parsers is described. An error recovery routine based on this new method can be generated automatically by a parser generator as a part of an* LR *parser. Based on the result that a viable suffix from which the unread part of the input is derived can be computed in certain states of an* LR *parser, the new method uses the viable suffix to discard the erroneous part of the input and to synchronize the parser stack with the rest of the input afterwards. Thus it resembles a simple but efficient error recovery method used by* LL *and other predictive parsers. It is proved that all states suitable for this kind of error recovery can automatically be identified by a parser generator.*

Povzetek: članek opisuje okrevanje po napaki v LR analizatorjih.

1 Introduction

Compilers are programs that mostly process erroneous input. Robust error recovery and meaningful error reporting are therefore essential parts of any industrial-strength compiler.

Nowadays many compilers perform syntax analysis using an LALR parser (more precisely, $\mathrm{LA}(1)\mathrm{LR}(0)$ parser) that is generated automatically by a parser generator like *yacc*, *bison*, *JavaCUP*, etc. [3]. However, none of these parser generators uses any advanced method for error recovery and reporting mainly, because these methods are either (a) time consuming, or (b) require inspection of individual parser states and manual insertion of error recovery routines [4, 5].

LALR parser generators usually provide only a very simple mechanism for error recovery and none for error reporting. If a *yacc* generated parser is to recover after an error is encountered within a sentential form derived from a nonterminal A, a compiler writer should insert a production

$$A \longrightarrow \mathbf{error}\ \alpha$$

manually where $\mathbf{error}$ is a *yacc* reserved word [2, 3]. In case of an error, a parser abandons other productions expanding A, moves forward over the erroneous part of the input and discards it until a string which can be reduced to α is seen. A reduction to A is performed and thus the parser is *resynchronized*. However, "proper placement of $\mathbf{error}$ tokens in a grammar is a black art" [3].

The paper is organized as follows. Section 2 includes definitions of some basic elements of formal language theory and parsing. Following the brief outline of the new method in Section 3, the construction of the finite automaton used by the error recovery routine is described in Section 4. This is followed in Section 5 by (a) the algorithm for computing the viable suffix needed for error recovery and (b) the algorithm for computing a grammatical context of the erroneous part of the input. Examples and figures are given along the way.

2 Basic definitions

Standard terminology of formal language theory and parsing is assumed [4, 5]. Throughout the paper we assume that grammars are reduced (no useless or unreachable symbols) and \$-augmented (production $S' \longrightarrow \$S\$$ is added as the only production expanding the new start symbol S'). A string $\gamma \in V^*$ is a viable prefix (suffix) of G iff there exists a rightmost (leftmost) derivation $S \Longrightarrow^*_{G,\mathrm{rm}} \gamma u$ ($S \Longrightarrow^*_{G,\mathrm{lm}} u\gamma^R$).

The nondeterministic $\mathrm{LR}(k)$ machine N_k for the grammar $G = \langle V, T, P, S\rangle$ (where V contains both nonterminal and terminal symbols) is a finite (semi)automaton with the state set equal to I_k (the set of valid $\mathrm{LR}(k)$-items for G), an initial item $i_0 \in I_k$, and a mapping $\delta_N \colon I_k \times (V \cup \{\varepsilon\}) \longrightarrow 2^{I_k}$ [1].

The deterministic $\mathrm{LR}(k)$ (or $\mathrm{LR}(k)\mathrm{LA}(k')$) machine M_k for the grammar G is a finite (semi)automaton with a set of states $Q \subseteq 2^{I_k}$, an initial state $q_S \in Q$, and a mapping $\delta_M \colon Q \times V \longrightarrow Q$. If $\delta^*_M(q_S, \gamma) = q$ for some $q \in Q$ and $\gamma \in V^*$, then $[\gamma]$ denotes the set of equivalent viable prefixes leading from the initial state q_S to the state q. Furthermore, $[\gamma]$ uniquely determines q (and is thus just another name for q).

The $\mathrm{LR}(k)$ parser is a pushdown transducer $\langle M, \tau\rangle$ (or simply M). M denotes the deterministic pushdown automaton based on the deterministic $\mathrm{LR}(k)$ machine M_k, and τ denotes the output effect (a mapping of parser actions into grammar productions). States of M are the same as the states of M_k. The stack alphabet of M is a set of states of M_k. A configuration (or instantaneous description) of a parser M is represented as $\$\Gamma \shortmid u\$$, where $\Gamma \in Q^*$ and $u \in T^*$ denote the stack contents and the unread part of the input, respectively.

3 The outline of the new method

To relieve a compiler writer of the "black art" of proper placement of **error** productions, a better error recovery method is needed. Let us suppose that the input string uv' is being parsed with an $\mathrm{LR}(k)$ parser M. Starting with the initial stack contents Γ_0, the parser M performs the parser steps $\pi(u)$ corresponding to the derivation

$$\$\Gamma_0 \shortmid uv'\$ \Longrightarrow_M^{\pi(u)} \$\Gamma \shortmid v'\$ \tag{1}$$

and enters a configuration $\$\gamma \shortmid v'\$$ (note that γ and v' denote the stack contents and the unread part of the input, respectively). $\mathrm{LR}(k)$ parsers have *the correct prefix property* [4, 5], i.e., any string of terminals pushed on the parser stack is a prefix of some valid input. It follows from Derivation (1) that there exist derivations

$$S \Longrightarrow_{\mathrm{rm}}^* \gamma v_i \Longrightarrow_{\mathrm{rm}}^* uv_i \tag{2}$$

for various v_i and a single viable prefix γ where $\Gamma = \Gamma'[\gamma]$ (the stack contents Γ of parser M corresponds to the viable prefix γ of Derivations 2). Therefore, there exist leftmost derivations

$$S \Longrightarrow_{\mathrm{lm}}^* u\delta_i \Longrightarrow_{\mathrm{lm}}^* uv_i \tag{3}$$

for various viable suffixes δ_i.

Now suppose that $\$\Gamma \shortmid v'\$$ is an error configuration. In other words, v' cannot be derived from any viable suffix δ_i in any of Derivations (3). The idea, on which the new method is based, can be outlined as follows:

1. If there is only one viable suffix $\delta = X_1\hat{\delta}$ such that $\delta_i = \delta$ for any i in Derivations (3), and
2. if this particular δ is known in the error configuration $\$\Gamma \shortmid v'\$$,

then the parser should discard the next few tokens of input, resynchronize and continue parsing the string derived from $\hat{\delta}$.

If there exists a unique viable suffix δ, there are two approaches to discard the erroneous part of the input. The first is to skip everything until a string from $\mathrm{FIRST}_k(\hat{\delta}\$)$ is seen, and then to resynchronize by pushing the symbol X_1 on the stack. Using this approach, Derivation (1) can be extended with the derivation

$$\begin{aligned} \$\Gamma \shortmid v'\$ &= \$\Gamma \shortmid v_1'\hat{v}\$ \\ &\Longrightarrow_M^{\pi(v_1')} \$\Gamma \shortmid \hat{v}\$ \\ &\Longrightarrow_M \$\Gamma[\gamma X_1] \shortmid \hat{v}\$ \end{aligned}$$

where the steps denoted by $\pi(v_1')$ are used to skip the part of the input derived from symbol X_1. Thus, the string $\hat{v}\$$ is the longest suffix of $v'\$$ having a property that $(k\colon \hat{v}\$) \in \mathrm{FIRST}_k(\hat{\delta}\$)$.

The second approach is to skip everything until a string which can be reduced to X_2, the first symbol of $\hat{\delta}$, is read, and then to resynchronize by pushing X_1 and then X_2 on the stack. Formally, Derivation (1) is extended with the derivation

$$\begin{aligned} \$\Gamma \shortmid v'\$ &= \$\Gamma \shortmid v_1'v_2\hat{v}\$ \\ &\Longrightarrow_M^{\pi(v_1')} \$\Gamma[\gamma X_1] \shortmid v_2\hat{v}\$ \\ &\Longrightarrow_M^{\pi(v_2)} \$\Gamma[\gamma X_1][\gamma X_1 X_2] \shortmid \hat{v}\$ \end{aligned}$$

where the steps denoted by $\pi(v_1')$ are used to skip the part of the input derived from symbol X_1 (as in the case above) and where $\pi(v_2)$ is the parse of v_2, the correct part of the input, in M.

The parser using an error recovery routine based on either of the two strategies tries to skip as few symbols of the input string as possible. More precisely, there may be many different, but viable splittings of the input string v' either to strings v_1' and $\hat{v}$ (as in the first approach) or to strings v_1', v_2 and $\hat{v}$ (as in the second approach). However, the problem of splitting the string v' is beyond the scope of this paper.

Finally, if the viable suffix δ cannot be determined uniquely in the parser state $[\gamma]$, the parser removes one state at the time from the parser stack until a state with a unique viable suffix is at the top of the stack. Then, one of the approaches described above is applied.

4 The construction of the error recovery routine

Two conditions were set in the previous section that must be fulfilled if a parser is to recover from the error: (1) the viable suffix δ must be unique and (2) it must be known. In general, a viable prefix γ and thus a state $[\gamma]$ (a set of $\mathrm{LR}(k)$-equivalent viable prefixes) can have many corresponding viable suffixes δ_i. To identify states of an $\mathrm{LR}(k)$ parser suitable for performing error recovery we start with the following two definitions [6]:

Definition 1 *Let $N_k = \langle I_k^G, V, P_{N_k}, i_0\rangle$ be a nondeterministic $\mathrm{LR}(k)$ machine for a grammar $G = \langle V, T, P, S\rangle$. A string of $\mathrm{LR}(k)$-valid items $i_0 i_1 \ldots i_n \in (I_k^G)^*$ is called a $\langle\gamma, k\rangle$-path if there exists a sequence $X_1, X_2, \ldots, X_n \in (V \cup \{\varepsilon\})$ so that $X_n \neq \varepsilon$ and $[i_{j-1}X_j \to i_j] \in P_{N_k}$ where $i = 1, 2, \ldots, n$.*

Definition 2 *$\langle\gamma, k\rangle$-paths ρ_1 and $\rho 2$, where $\rho_1 = i_0 i_1 \ldots i_n$ and $\rho_2 = i_0 i_1' \ldots i_m'$ are 0-equivalent iff $n = m$ and items i_j and i_j' differ only in lookahead strings for all $j = 1, 2, \ldots, n$ (i.e., if $i_j = [A \to \alpha \bullet \beta, x]$ and $i_j' = [A' \to \alpha' \bullet \beta', x']$, then $A = A'$, $\alpha = \alpha'$, and $\beta = \beta'$).*

The reason for defining 0-equivalence of $\langle\gamma,k\rangle$-paths becomes obvious with the following lemma, which establishes a relationship between viable prefixes and suffixes on the one hand and $\mathrm{LR}(k)$ machines on the other.

Lemma 1 *Any two* 0*-equivalent* $\langle\gamma,k\rangle$*-paths define the same viable suffix.*

PROOF: Any $\langle\gamma,k\rangle$-path $\rho = i_0 i_1 \dots i_n$ specifies a set of leftmost derivations all having the form

$$\begin{aligned}
A_1 &\Longrightarrow^{p_1}_{\mathrm{lm}} \alpha_1 A_2 \beta_1\\
&\Longrightarrow^{\pi(\alpha_1)}_{\mathrm{lm}} u_1 A_2 \beta_1\\
&\Longrightarrow^{p_2}_{\mathrm{lm}} u_1 \alpha_2 A_3 \beta_2 \beta_1\\
&\Longrightarrow^{\pi(\alpha_1)}_{\mathrm{lm}} u_1 u_2 A_3 \beta_2 \beta_1\\
&\vdots\\
&\Longrightarrow^{p_m}_{\mathrm{lm}} u_1 u_2 \dots u_{m-1} \alpha_m A_{m+1} \beta_m \beta_{m-1} \dots \beta_1\\
&\Longrightarrow^{\pi(\alpha_1)}_{\mathrm{lm}} u_1 u_2 \dots u_m A_{m+1} \beta_m \beta_{m-1} \dots \beta_1,
\end{aligned}$$

where $p_j = A_j \longrightarrow \alpha_j A_{j+1} \beta_j$ is the production of the j-th $\mathrm{LR}(k)$-item of ρ having the dot in the initial position at the far left (and A_{m+1} may be ε). As any change of lookahead strings in $\mathrm{LR}(k)$-items of ρ does not affect the leftmost derivations above, all $\langle\gamma,k\rangle$-paths which are 0-equivalent to ρ define the same viable suffix δ, where $\delta^R = A_{m+1}\beta_m\beta_{m-1}\dots\beta_1$.
□

When the traditional $\mathrm{LR}(k)$ parser enters the error configuration $\$\Gamma | v'\$$, the error is recognized because no action is specified for q and $x = k\colon v'$, where $\Gamma = \Gamma' q$, i.e., $\mathtt{ACTION}(q,x) = \mathit{error}$. But as $\mathrm{LR}(k)$ parsers have the correct prefix property, the first $(k-1)$ symbols of the lookahead buffer are correct — otherwise the error would have been detected earlier.

The first step is to identify all states $[\gamma]$ with the property that for any $\gamma' \in [\gamma]$, all $\langle\gamma',k\rangle$-paths ending with an item $(k-1)$-active for $(k-1)\colon v'$ are 0-equivalent (an item $[A \to \alpha X \bullet \beta, y]$ is k-active for x if and only if $x \in \mathrm{FIRST}^G_k(\beta y)$). In other words, the stack contents of the $\mathrm{LR}(k)$ parser and the first $(k-1)$ symbols in the lookahead buffer must define the viable suffix uniquely (remember that in error configuration $\$\Gamma | v'\$$ the k-th symbol of v' is erroneous and thus not useful for error recovery). To do so, we change the focus to the nondeterministic $\mathrm{LR}(0)$ machine N_0.

Definition 3 *An* $\mathrm{LR}(0)$*-item* $[A \to \alpha \bullet \beta]$ *of* N_0 *is relevant for state* $[\gamma]$ *of the deterministic* $\mathrm{LR}(k)$ *machine for* G *if and only if, for all* $\gamma' \in \gamma$ *and* $i' = [A \to \alpha \bullet \beta, y] \in [\gamma]$*, all* $\langle\gamma',k\rangle$*-paths* $\rho = \rho' i'$ *are* 0*-equivalent.*

During the parser construction, we compute a directed graph G_N with a set of vertices V_N and a set of edges E_N defined as

$$\begin{aligned}
V_N = \{ &\langle [A \to \alpha \bullet \beta], q\rangle\\
&| \exists q \in Q_M, y \in T^*\colon [A \to \alpha \bullet \beta, y] \in q\}
\end{aligned}$$

and

$$\begin{aligned}
E_N = \{ &\langle\langle i_1,q_1\rangle,\langle i_2,q_2\rangle\rangle | \exists X \in V \cup \{\varepsilon\}:\\
&[q_1 X \to q_2] \in P_M \wedge [i_1 X \to i_2] \in P_{N_0}\},
\end{aligned}$$

respectively. It is derived from the graph of the nondeterministic $LR(0)$ machine by (1) replicating each $LR(0)$-item as many times as there are states in M in which an LR(k) item with the corresponding core appears, and (2) erasing edge labels. Thus every path in G_N starting with $\langle [S' \to \bullet\$S\$], q_S\rangle$ has its corresponding path in N_0 (and vice versa).

As an example, consider the following \$-augmented LALR grammar G_{ex} with productions

$$\begin{aligned}
&S' \longrightarrow \$S\$,\ S \longrightarrow AB,\\
&A \longrightarrow aA,\ A \longrightarrow \varepsilon,\\
&B \longrightarrow Bb,\ B \longrightarrow b.
\end{aligned}$$

The LALR machine for the grammar G_{ex} as constructed by a modified version of *bison* is shown in Figure 1. The graph G_N for the grammar G_{ex} is shown in Figure 2.

The irrelevant vertices of G_N (i.e., vertices $\langle i,q\rangle$ where i is irrelevant for q) can be identified by the following simple algorithm:

1. Compute the set of all conflicting vertices, i.e., those with at least two different predecessors in the same state:

$$\begin{aligned}
\bar{V}_N^{(1)} = \{v| \ &v = \langle [A \to \bullet\beta], q\rangle \wedge\\
&\langle\langle i_1,q\rangle, v\rangle, \langle\langle i_2,q\rangle, v\rangle \in E_N \wedge\\
&i_1 \neq i_2\}
\end{aligned}$$

   (Different predecessors from different states represent no problem as the state itself helps determining the right path.)

2. Compute the set of successors of conflicting vertices:

$$\bar{V}_N^{(2)} = \{v | v \text{ is-reachable-from } v' \in \bar{V}_N^{(1)}\}$$

In Figure 2 the irrelevant vertices are shaded while in Figure 1 they are closed between double lines.

We define *the skeleton automaton U* with the set of states

$$Q_U = \{i | \langle i,q\rangle \in V_N \setminus (\bar{V}_N^{(1)} \cup \bar{V}_N^{(2)})\},$$

alphabet Q_M, and the mapping $\delta_U\colon Q_U \times Q_M \longrightarrow Q_U$ defined as

$$\begin{aligned}
&\delta_U(i,q') = i' \Longleftrightarrow\\
&\quad \langle\langle i',q'\rangle,\langle i,q\rangle\rangle \in E_N \wedge\\
&\quad \langle i,q\rangle, \langle i',q'\rangle \in V_N \setminus (\bar{V}_N^{(1)} \cup \bar{V}_N^{(2)}).
\end{aligned}$$

The skeleton automaton is just a compact representation of the graph G_N with irrelevant vertices removed. The skeleton automaton for the grammar G_{ex} is shown in Figure 3. The aforementioned modification of *bison* makes

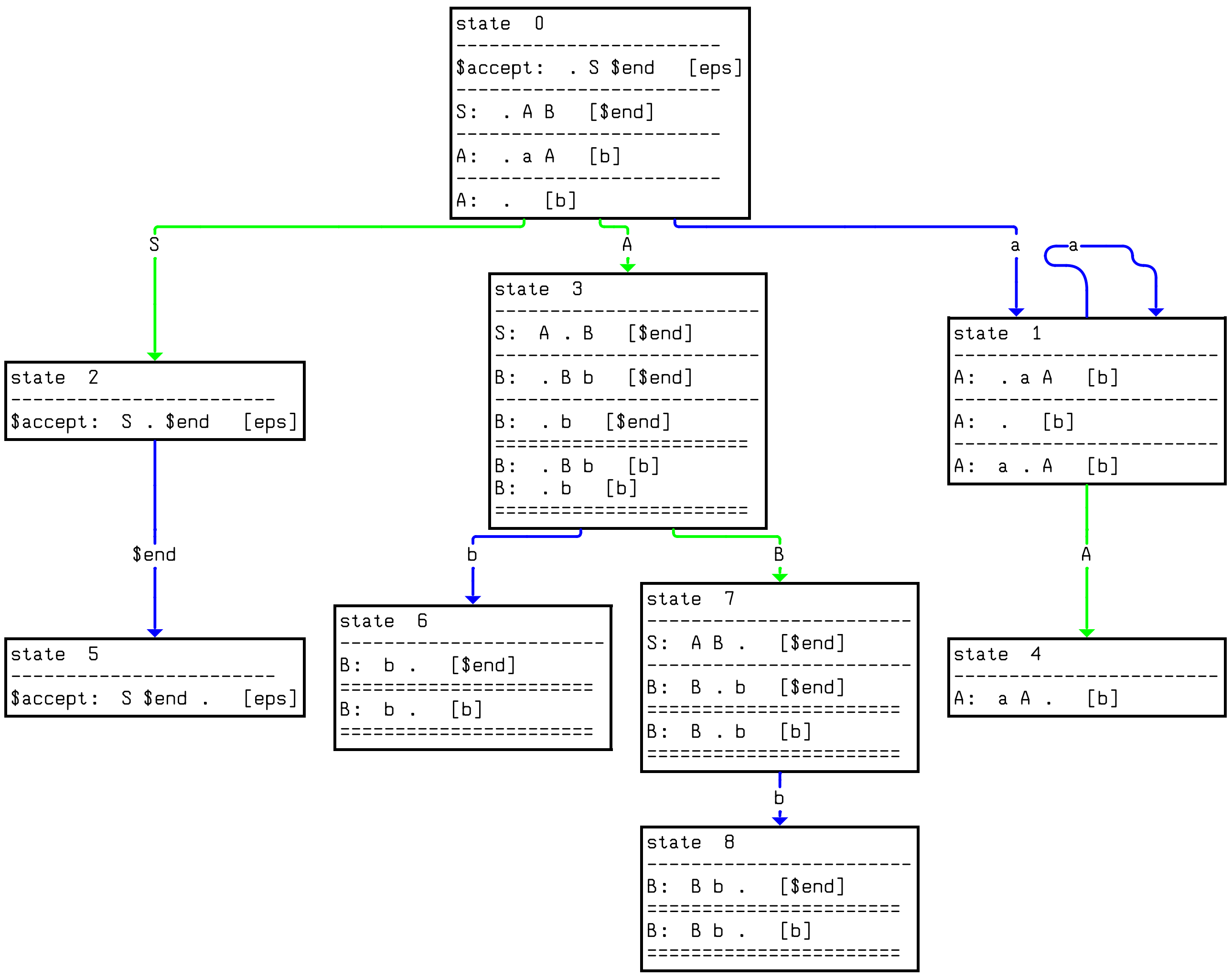

Figure 1: *Bison*-generated LALR machine for the grammar G. Note the different grammar augmentation: instead of using production $S' \longrightarrow \$S\$$, *bison* uses a production $\$_{\text{accept}} \longrightarrow S\$$.

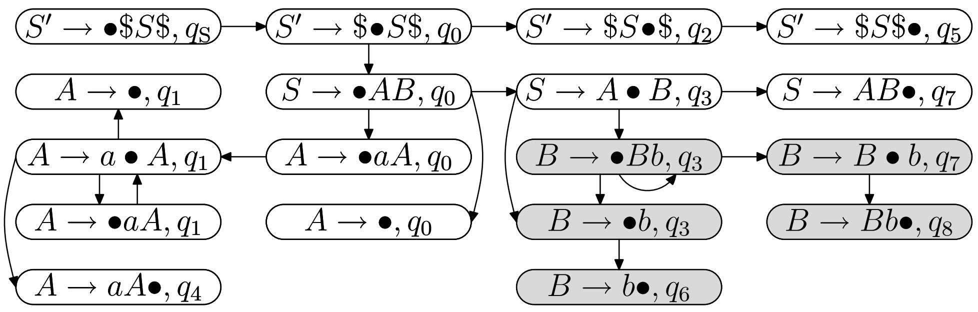

Figure 2: Graph G_N for the grammar G_{ex}.

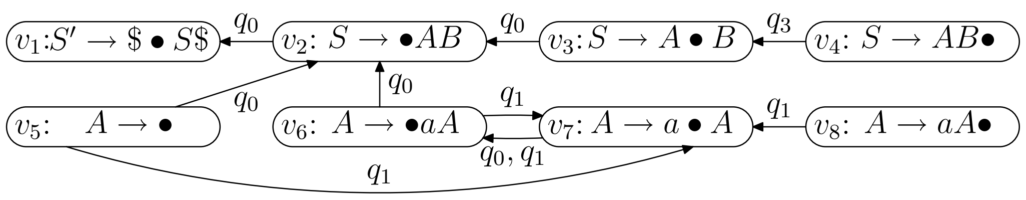
v_1:$S' \to \$ \bullet S\$$
v_2: $S \to \bullet AB$
v_3:$S \to A \bullet B$
v_4: $S \to AB\bullet$
v_5: $A \to \bullet$
v_6: $A \to \bullet aA$
v_7: $A \to a \bullet A$
v_8: $A \to aA\bullet$
q_0
q_0
q_3
q_0
q_0
q_1
q_1
q_0, q_1
q_1

Figure 3: Skeleton automaton U for the grammar G_{ex}.

bison capable of computing the skeleton automaton — *bison*-generated skeleton automaton for the same grammar (although differently augmented) is shown in Figure 4.

(the modification of *bison* makes *bison* capable of computing the skeleton automaton as described below).

If the $\mathrm{LR}(k)$ parser is to start the error recovery process in state q and with the string x in the lookahead buffer, it should be able to select the right vertex of the skeleton automaton U. Hence, apart from the skeleton automaton, the parser must contain the table ERROR, which maps the topmost state and the first $(k-1)$ symbols of the lookahead buffer to a vertex of U:

$$\mathtt{ERROR}\colon Q \times T^{*(k-1)} \longrightarrow V_N.$$

Construction of the table ERROR is straightforward. To compute the value of ERROR(q,x), apply the following procedure:

1. If there exists a state of the skeleton automaton U corresponding to an item i where $\langle i, q\rangle \in V_N \setminus (\bar{V}_N^{(1)} \cup \bar{V}_N^{(2)})$ and all items of the core of the state q which are $(k-1)$-active for x map to i, i.e.,

   $$\forall [A \to \alpha \bullet \beta, y] \in \mathrm{Core}(q)\colon$$
   $$x \in \mathrm{FIRST}_{k-1}(\beta y) \Longrightarrow [A \to \alpha \bullet \beta] = i,$$

   then ERROR$(q,x) = i$. Otherwise, the value of ERROR(q,x) is undefined.

   (The set $\mathrm{Core}(q)$ contains either all items $[A \to \alpha \bullet \beta, y]$ where $\alpha \neq \varepsilon$ if $q \neq q_S$ or the item $[S \to \bullet \$S\$, \varepsilon]$ if $q = q_S$.)

2. If there exists exactly one node $\langle i', q\rangle \in V_N \setminus (\bar{V}_N^{(1)} \cup \bar{V}_N^{(2)})$ where $\delta_U(i', q) = i$ and there exists an item $[A \to \alpha \bullet \beta, y] \in q$ so that $x \in \mathrm{FIRST}_{k-1}(\beta y)$, then set ERROR$(q,x) = i'$ and repeat Step 2; otherwise, terminate.

   In other words, make the path leading from q to q_S as long as possible, but keep it unique in respect with the first $(k-1)$ symbols of the lookahead buffer.

Table 1 shows the ERROR table for the grammar G_{ex}.

5 Computing the context of the syntax error

As mentioned at the end of Section 2, not every state of $\mathrm{LR}(k)$ parser is suitable for error recovery. If an error is detected in the state where error recovery cannot start, i.e., in the error configuration $\$\Gamma \,|\, v'\$$ where $\Gamma = \Gamma' q$, $x = k\colon v'$ and ERROR$(q,x) = \bot$, then the $\mathrm{LR}(k)$ parser must remove the topmost state from the stack and repeat the spawning of the error recovery. But as the lookahead string x in that state is no longer available, the parser must push the appropriate vertex of the skeleton automaton together with the at the time when the state itself.

LR STATE		SKELETON STATE
S		$[S' \to \bullet \$S\$]$
0	v_2:	$[S \to \bullet AB]$
1	v_7:	$[A \to a \bullet A]$
2	v_9:	$[S' \to \$S \bullet \$]$
3	v_3:	$[S \to A \bullet B]$
4	v_8:	$[A \to aA\bullet]$
5	v_{10}:	$[S' \to \$S\$\bullet]$
6		undefined
7	v_4:	$[S \to AB\bullet]$
8		undefined

Table 1: The ERROR table for grammar G_{ex}.

More precisely, the parser stack should not contain just parser states, but pairs consisting of a parser state and a vertex of the skeleton automaton which is to be used if an error occurs. Hence, whenever the state q is pushed onto the stack (as a result of either shift or reduce action), it should be pushed as a pair $\langle q, \bot\rangle$. In the next step, before checking the action table and deciding on the next action, the parser must check the table ERROR and correct the value of the second component of the topmost pair on the stack.

The algorithm for computing the context in which a syntax error occurs is presented in Figure 5 (δ_U is a transition function corresponding to the set of rewriting rules P_U of skeleton automaton U). It starts at the top of the stack and proceeds downward. It produces a list of $\mathrm{LR}(0)$-items $i_1 i_2 \ldots i_m$ where $i_j = [A_j \to \alpha_j \bullet A_{j+1}\beta_j, y_j]$, which determine the derivation

$$\begin{aligned} A_1 \quad &\Longrightarrow^{i_1}_{\mathrm{lm}} \alpha_1 A_2 \beta_1 \\ &\Longrightarrow^{i_2}_{\mathrm{lm}} \alpha_1 \alpha_2 A_3 \beta_2 \beta_1 \\ &\vdots \\ &\Longrightarrow^{i_m}_{\mathrm{lm}} \alpha_1 \alpha_2 \ldots u_{m-1} \alpha_m A_{m+1} \beta_m \beta_{m-1} \ldots \beta_1 \end{aligned}$$

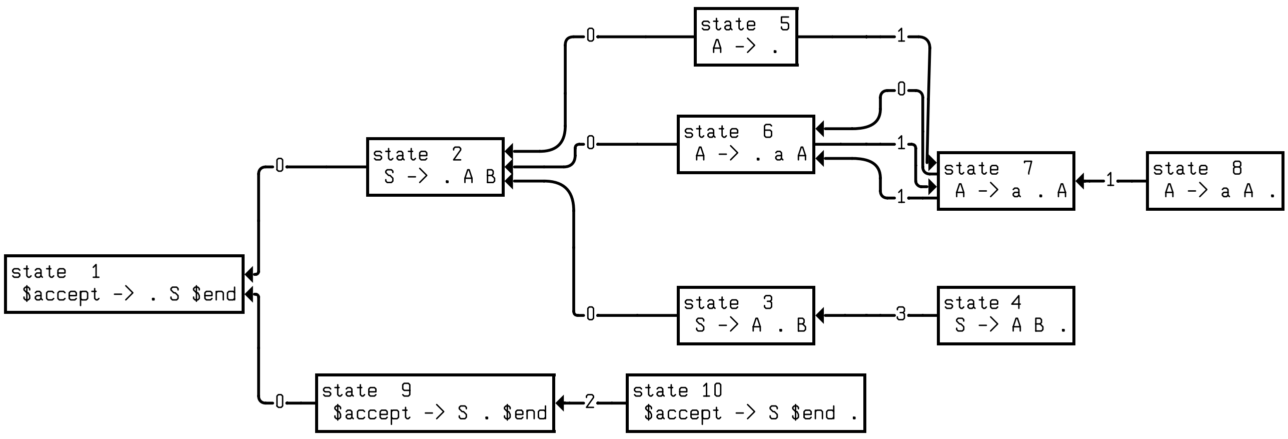

Figure 4: *Bison*-generated skeleton automaton U for the grammar G_{ex}.

$$\begin{aligned}
&\mathit{context\ stack}\ i \mid i == [S' \to \$ \bullet S\$] \ = \ \varepsilon \\
&\mathit{context\ stack}@((q,\_):st)\ [A \to \bullet\beta] \ = \ \mathit{ctx} \circ [A \to \bullet\beta] \\
&\qquad \text{where } i \ = \ \delta_U([A \to \bullet\beta], q) \\
&\qquad\qquad \mathit{ctx} \ = \ \mathit{context\ stack}\ i \\
&\mathit{context\ stack}@((q,\_):st)\ [A \to \alpha X \bullet \beta] \ = \ \mathit{ctx} \circ [A \to \alpha X \bullet \beta] \\
&\qquad \text{where } \mathit{ctx} \ = \ \mathit{context\ st}\ [A \to \alpha \bullet X\beta]
\end{aligned}$$

Figure 5: Algorithm for computing the viable suffix.

and the viable suffix $\beta_1^R\beta_2^R\dots\beta_m^R$. Hence, we can write down the following theorem:

Theorem 1 *If any two $\langle\gamma', k\rangle$-paths ending with items which are $(k-1)$-active for x, are 0-equivalent for each $\gamma' \in [\gamma]$, then a viable suffix δ_i in derivation (3) can be computed from the stack contents $\Gamma = \Gamma'[\gamma]$ in the parser configuration $\$\Gamma \mid v\$$ for any $v = xv'$.*

A parse of erroneous string *aacbb* is shown in Tables 2 and 3. Both possible solutions mentioned in Section 2 are shown. In Table 2, the erroneous part of the input is discarded until the string $b \in \text{FIRST}_1(AB\$)$ is found in the lookahead buffer. The resulting stack contents after error recovery is performed is therefore $\$[\$][\$a][\$aa][\$aaA]$. This is a simple but efficient solution.

In Table 3, however, the string *bb* is reduced to the second symbol of the viable suffix $AB\$$, namely B. Hence, the resulting stack contents is $\$[\$][\$A][\$B]$. Finding and reducing the substring *bb* can be performed in the same way as by parsers generated by existing LALR parser generators if **error** productions are used.

Finally, the list of items

$$\begin{aligned}
&[S' \to \$ \bullet S\$], [S \to \bullet AB], \\
&[A \to \bullet aA], [A \to a \bullet A], \\
&[A \to \bullet aA], [A \to a \bullet A]
\end{aligned}$$

	STACK	INPUT
1.	$\$(q_0, v_2)$	*aacbb*\$
2.	$\$(q_0, v_2)(q_1, v_7)$	*acbb*\$
3.	$\$(q_0, v_2)(q_1, v_7)(q_1, v_7)$	*cbb*\$
	$\Rightarrow$ *context* returns	
	$[S' \to \$ \bullet S\$]$	
	$[S \to \bullet AB]$	
	$[A \to \bullet aA]$	
	$[A \to a \bullet A]$	
	$[A \to \bullet aA]$	
	$[A \to a \bullet A]$	
	yielding viable suffix $(AB\$)^R$:	
4.	$\$(q_0, v_2)(q_1, v_7)(q_1, v_7)(q_4, v_8)$	*bb*\$

Table 2: A trace of parsing with error recovery: erroneous part of the input is discarded until $b \in \text{FIRST}_1(AB\$)$ is seen.

	STACK	INPUT
1.	$\$(q_0, v_2)$	$aacbb\$$
2.	$\$(q_0, v_2)(q_1, v_7)$	$acbb\$$
3.	$\$(q_0, v_2)(q_1, v_7)(q_1, v_7)$	$cbb\$$
	$\Rightarrow$ *context* returns	
	$[S' \to \$ \bullet S\$]$	
	$[S \to \bullet AB]$	
	$[A \to \bullet aA]$	
	$[A \to a \bullet A]$	
	$[A \to \bullet aA]$	
	$[A \to a \bullet A]$	
	yielding viable suffix $(AB\$)^R$:	
4.	$\$(q_0, v_2)(q_3, v_3)(q_7, v_4)$	$\$$

Table 3: A trace of parsing with error recovery: erroneous part of the input is discarded until bb derived from B in $AB\$$ is reduced.

represents a particularly good starting point for printing out helpful error messages because it provides the compiler writer with the exact grammatical context within which the error occurred.

6 Conclusion

The presented method works with both, canonical $\mathrm{LR}(k)$ parsers as well as $\mathrm{LA}(k)\mathrm{LR}(k')$ parsers. The error recovery routine does not slow down or influence the parser until it encounters the first error, and it can be generated automatically. Besides, it has two main benefits: (a) a compiler writer needs not add any additional productions to the grammar and (b) it is a good starting point for meaningful error reporting. However, the generation of parser is slower and the generated parser is larger.

References

[1] A. V. Aho and J. D. Hopcroft. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, 1979.

[2] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley series in Computer Science. Addison-Wesley, 1986.

[3] J. Levine, T. Mason, and D. Brown. *Lex & Yacc*. O'Reilly & Associates, 1992.

[4] S. Sippu and E. Soisalon-Soininen. *Parsing Theory, Volume I: Languages and Parsing*, volume 15 of *EATCS Monographs on Theoretical Computer Science*. Springer-Verlag, 1988.

[5] S. Sippu and E. Soisalon-Soininen. *Parsing Theory, Volume II: LR(k) and LL(k) Parsing*, volume 20 of *EATCS Monographs on Theoretical Computer Science*. Springer-Verlag, 1990.

[6] B. Slivnik. *Kombinacija Knuthovega in Lewis-Stearnsovega sintaksnega analizatorja z minimalno uporabo Knuthove analize*. PhD thesis, University of Ljubljana, Ljubljana, Slovenia, 2003.