Information Processing Letters 47 (1993) 123–129
Elsevier

14 September 1993

An improved LALR(k) parser generation for regular right part grammars

Heung-Chul Shin and Kwang-Moo Choe

Department of Computer Science, Korea Advanced Institute of Science and Technology, 373-1 Kusong-dong, Yusong-gu, Taejon 305-701, South Korea

Communicated by L. Boasson
Received 16 December 1992
Revised 21 June 1993

Keywords: Formal languages; regular right part grammars; finite automata; extended LR(0) automaton; extended LALR(k) parser

1. Introduction

A regular right part grammar (RRPG) is a context-free grammar, in which right parts of productions are finite automata to extend the descriptive power of context-free grammar by including notations for describing repetitions and alternations [6,8]. On LR parsing of RRPGs, extra work is required to identify the left end of a handle at reduction time because a nonterminal can derive potentially infinite number of strings via a single production.

For parsing RRPGs, some methods such as grammar transformation from RRPGs to LR(k) context-free grammars [5,8], augmenting LR(0) automaton with readback machines to recognize the reverse of state sequences leading to a reduction [2,6,7], and stacking an LR state only when the symbol being processed indicates the beginning of a new right part [11], are suggested. In stacking method, the parser is efficient because exactly one state entry is popped from the stack when the right end of a production is found. However, if stacking conflicts occur, the transformations to another RRPG whose parser has no stacking conflict are necessary [11].

Another method is the combination of stacking method and readback method [9]. When stacking conflicts occur while reading the right part of a production, several state entries are popped until one of the lookback states (in which parser may be restarted after reduction [2]) for a reduction appears. Readback states which are used to construct readback machine need not be added. Moreover, stacking conflicts are resolved by using lookback states without grammar transformation. Both the parser and the generation of the parsers are efficient [9]. In the method, the right parts of productions are deterministic finite automata.

Our approach is based on the method of Nakata and Sassa [9]. Even though their method is very simple and can produce small and fast parsers if there is no stacking conflict, it requires a very large amount of space for realistic grammars in keeping the relations on (state, item) pairs [9]. To reduce space requirements, we use only kernel items of parser states on building the parser. An improved parser generation algorithm for RRPGs which allows the right parts of productions to be nondeterministic finite automata, is presented. It is well known that for succinct-

Correspondence to: H.-C. Shin, Department of Computer Science, Korea Advanced Institute of Science and Technology, 373-1 Kusong-dong, Yusong-gu, Taejon 305-701, South Korea.

0020-0190/93/$06.00 © 1993 – Elsevier Science Publishers B.V. All rights reserved

ness of classes of language descriptions, nondeterministic finite automata can be exponentially more succinct than deterministic finite automata [12].

2. Terminology and basic definitions

The reader is referred to [1,12] for notations and definitions relating to relations, strings, languages, and automata.

A *nondeterministic finite automaton* $M_0 = (Q_0, V, \delta_0, q_0, F_0)$ where Q_0 is a finite set of *states*, V is a finite set of *transition symbols*, $\delta_0 : Q_1 \times V \to 2^{Q_0}$ is the *state transition function*, $q_0 \in Q_0$ is the *initial* state, and $F_0 \subseteq Q_0$ is the set of *final* states. δ_0 is extended to a mapping $\hat{\delta}_0 : Q_0 \times V^* \to 2^{Q_0}$ as follows:

$$\hat{\delta}_0(q, \varepsilon) = \{q\},$$

$$\hat{\delta}_0(q, \alpha X) = \left\{p \mid \exists r \in \hat{\delta}_0(q, \alpha): p \in \delta_0(r, X)\right\}$$

where $Z \in V$, $\alpha \in V^*$.

A *regular right part grammar* $G = (N, \Sigma, P, S)$ where N and Σ are finite set of *nonterminal symbols* and *terminal symbols* such that $N \cap \Sigma = \emptyset$, respectively, $P \subseteq N \times M_N$ is a finite set of *productions* of the form $A \to M_A$ where A, the *left part*, is in N and M_A, the *right part*, is a *nondeterministic finite automaton* recognizing a subset of V^* where $M_A = (Q_A, V, \delta_A, q_A, F_A)$ and $V = N \cup \Sigma$, and $S \in N$ is the *start symbol*; in which

$$M_N = \bigcup_{A \in N} \{M_A\}, \quad Q = \bigcup_{A \in N} Q_A,$$

$$A_I = \bigcup_{A \in N} \{q_A\}, \quad Q_F = \bigcup_{A \in N} F_A,$$

$$|N| = |M_N| = |Q_I|,$$

and δ is the collection of δ_A for all A in N. For each production $A \to M_A$, we write it more conveniently as $A \to q_A$ or simply $A \to q$ if no ambiguity can arise, and the sets Q_A are assumed to be disjoint as original definition of standard form defined in [6]. The definitions of useless symbol and reduced grammar can be adapted from context-free grammars in a straightforward way. It will be assumed that RRPGs are reduced and in \$-augmented form [2]. Figure 1 shows an RRPG $G1$ whose productions are represented by transition graphs. Using regular expressions, these productions might be written as

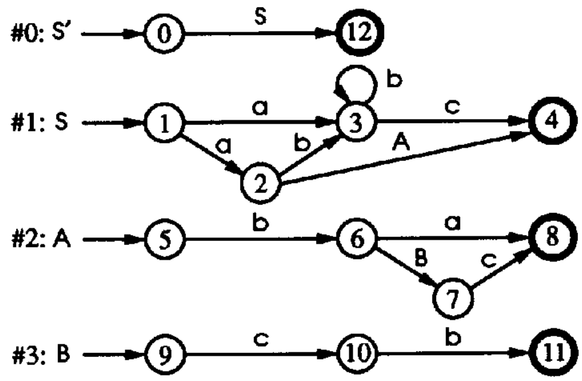

Fig. 1. Regular right part grammar $G1$.

$$S' \to S, \quad S \to (a|ab)b^*c|aA,$$

$$A \to b(a|Bc), \quad B \to cb.$$

The formalism used in [4] to describe LR parsers for context-free grammars can be adapted for RRPGs with slight modifications.

An *extended* LR(0) (ELR(0)) *automaton* for an RRPG G, $\mathrm{ELR}_0(G) = (I, V, P, I_0, \mathrm{Next}, \mathrm{Reduce})$ where I is a finite set of *states*, V and P are as in G, $I_0 \in I$ is the *start state*, $\mathrm{Next}: I \times V \to I$ is the *transition function* which may be a partial function, and $\mathrm{Reduce}: I \times \Sigma \to 2^P$ is the *reduce function*. To avoid confusion an ELR(0) automaton state is referred to as a state and a state of production right part is referred to as a right part state. A state I_q is *inconsistent* iff there exists an $a \in \Sigma$ such that $\mathrm{Next}(I_q, a) \in I$ and $\mathrm{Reduce}(I_q, a) \neq \emptyset$, $|\mathrm{Reduce}(I_q, a)| > 1$, or both.

The classical construction for building LR(0) automata [3] can be applied to ELR(0) automata by defining an *item* to be a dotted right part state which is of the form $[A \to q{:}p]$ or simply $[p]$ where $A \to q$ is a production and p is its right part state, and identifying ELR(0) states with sets of items. Each state I_q consists of kernel items and nonkernel items which are denoted as Kernel(I_q) and Nonkernel(I_q). In particular, initial state I_0 has only nonkernel items [9].

An operator ∂ is a mapping in the power set of items as used in [10], which is defined as

$$\partial\{[p]\} = \{[q] \mid \exists A \in N: \delta(p, A) \neq \emptyset \wedge A \to q \in P\}.$$

The closure set of an item set is given by reflexive transitive closure ∂^*. A correct ELR(0) automaton is given by putting

$$I_0 = \partial^*\{[S' \to q_{S'}:q_{S'}]\},$$
$$\text{Next}(I_q, Z) = \partial^*\tau(I_q, X),$$

and

$$\text{Reduce}(I_q, a) = \{A \to q \in P \mid \exists[p] \in I_1,\ \alpha \in V^*: p \in (\hat{\delta}(q, \alpha) \cap Q_F)\}$$

where

$$\tau(I_q, X) = \{[p] \mid \exists[q] \in I_q,\ X \in V: p \in \delta(q, X)\}.$$

For example, $\partial\{[0]\} = \{[1]\}$, $\partial\{[2]\} = \{[5]\}$, and $\partial\{[6]\} = \{[9]\}$ for $G1$. Moreover, $I_0 = \partial^*\{[0]\} = \{[0], [1]\}$ and $\text{Next}(I_0, a) = \partial^*\tau(I_0, a) = \{[2], [3], [5]\}$ because $\partial\{[0]\} = \{[1]\}$, $\tau(I_0, a) = \{[2], [3]\}$, and $\partial\{[2]\} = \{[5]\}$.

The set I is identified with the set of time sets recursively, which is the smallest satisfying

$$I = I_0 \cup \{\partial^*(I_p) \mid \exists I_q \in I,\ X \in V: I_p = \tau(I_q, X)\}.$$

The behavior and properties of an ELR(0) automaton can be understood in terms of *transitions* and *paths*. A transition (I_q, X) is represented by $I_q \xrightarrow{X} I_p$, where $I_p = \text{Next}(I_q, X)$. A path is a sequence of states $I_0, I_1, \ldots, I_n$ such that for some $X_1, X_2, \ldots, X_n$,

$$I_0 \xrightarrow{X_1} I_1 \xrightarrow{X_2} I_2 \to \cdots \to I_{n-1} \xrightarrow{X_n} I_n$$

which is abbreviated $I_0 \xrightarrow{\alpha}{}^* I_n$ where $\alpha = X_1X_2 \ldots X_n$, that is, α *accesses* I_n.

It is important to note that, whereas in the LR(0) automaton for a context-free grammar accessing strings can easily be deduced from state sequences because each state has a unique entry symbol [12], in the ELR(0) automaton for an RRPG accessing strings may not be deduced from state sequences because each state may have several distinct entry symbols. It comes from repetitions and alternations in the description of an RRPG. The number of states of ELR(0) automaton for an RRPG may be less than that of LR(0) automaton for corresponding context-free grammar. Therefore, for an RRPG, it is preferred to use the ELR(0) automaton rather than the LR(0) automaton.

A relation T on (state, item) pairs is defined by

$$(I_q, [q])T(I_p, [p]) \quad \text{iff} \quad \exists X \in V: I_q \xrightarrow{X} I_p \wedge p \in \delta(q, X).$$

As shown in [2],

$$(I_q, [q])T^*(I_p, [p]) \quad \text{iff} \quad \exists \alpha \in V^*: I_q \xrightarrow{\alpha}{}^* I_p \wedge p \in \hat{\delta}(q, \alpha).$$

Moreover, T is the union of two different sorts of relation:

$$(I_q, [q])_K T(I_p, [p]) \quad \text{iff} \quad (I_q, [q])T(I_p, [p]) \wedge [q] \in \text{Kernel}(I_q),$$

$$(I_q, [q])_N T(I_p, [p]) \quad \text{iff} \quad (I_q, [q])T(I_p, [p]) \wedge [q] \in \text{Nonkernel}(I_q).$$

3. Extended LALR(k) parsing of RRPGs

An *extended* LALR(k) (ELALR(k)) *parser* for an RRPG G, $\text{ELALR}_k(G) = (I, V, P, I_0, PT)$ where I, V, P, and $I_0 \in I$ are as in $\text{ELR}_0(G)$ except that each item in $I_q \in I$ is augmented with the set of lookahead strings of length k, respectively, and PT is a *parsing table* which consists of two parts, a parsing action function *Action* and a goto function *Goto*. Action is a mapping from $I \times k{:}\Sigma^*$ to subset of {shift I, stack-shift I, reduce $A \to q$, accept, error} where $k{:}\Sigma^*$ denotes the prefix of length k of terminal string [1,12] and

Goto is a mapping from $I \times N$ to {goto I, stack-goto I}.

The ELALR(k) parser can be constructed by computing the collection of sets of ELR(0) items and augmenting each item with set of lookahead strings of length k. The parsing actions of an ELALR(k) parser and described in terms of relation $\vdash$ (read as *moves to*) defined on configurations which consist of state stack and input string. States and vocabularies are stacked as usual. However, a state which is not a lookback state for a reduction by particular production is not stacked [9,11], where lookback state is the state from which the ELR(0) automaton began its search for the handle which is about to reduce. Therefore, a configuration of an ELALR(k) parser is of the form $(I_0\alpha_0 I_1\alpha_1 \ldots I_{n-1}\alpha_{n-1} I_n\alpha_n I_q,\ z)$ where $I_i \in I$, $\alpha_i \in V^*$ for all i such that $0 \leqslant i \leqslant n$, $z \in \Sigma^*$, and $I_q \in I$ is the current state.

Basic ELALR(k) parser for RRPG is composed of five kinds of moves as follows:

(1) shift I_p:

$$(I_0\alpha_0 \ldots I_n\alpha_n I_q,\ az) \vdash (I_0\alpha_0 I_1\alpha_1 \ldots I_n\alpha_n a I_p,\ z)$$
$$\text{if } I_p = \text{Next}(I_q, a) \text{ and } \tau_K(I_q, a) \neq \emptyset,$$

(2) stack-shift I_p:

$$(I_0\alpha_0 \ldots I_n\alpha_n I_q,\ az)$$
$$\vdash (I_0\alpha_0 I_1\alpha_1 \ldots I_n\alpha_n I_q a I_p,\ z)$$
$$\text{if } I_p = \text{Next}(I_q, a) \text{ and } \tau_K(I_q, a) = \emptyset,$$

(3) reduce $A \to r$:

$$(I_0\alpha_0 \ldots I_n\alpha_n I_q,\ az)$$
$$\vdash (I_0\alpha_0 I_1\alpha_1 \ldots I_{n-1}\alpha_{n-1} A I_p,\ az)$$
$$\text{if } q \in Q_F \text{ for } [A \to r{:}q] \in I_q,$$
$$k{:}az \in LA_k(I_1, [A \to r{:}q]),$$
$$\text{and Goto}(I_n, A) = \text{goto } I_p\text{; or}$$

$$(I_0\alpha_0 \ldots I_n\alpha_n I_q,\ az)$$
$$\vdash (I_0\alpha_0 I_1\alpha_1 \ldots I_{n-1}\alpha_{n-1} I_n A I_p,\ az)$$
$$\text{if } q \in Q_F \text{ for } [A \to r{:}q] \in I_q,$$
$$k{:}az \in LA_k(I_1, [A \to r{:}q]),$$
$$\text{and Goto}(I_n, A) = \text{stack-goto } I_p,$$

(4) accept: if configuration is of the form $(I_0 S I_q,\ \$^k)$,

(5) error: otherwise,

where $\tau_K(I_q, X)$ is the subset of $\tau(I_q, X)$ such that

$$\tau_K(I_q, Z) = \{[p] \mid \exists [q] \in \text{Kernel}(I_q),\ X \in V:\ p \in \delta(q, X)\},$$

LA_k is set of lookahead strings of length k for an item in a state, and

$$\text{Goto}(I_n, A) = \begin{cases} \text{goto } I_p & \text{if } I_p = \text{Next}(I_n, A) \text{ and } \tau_K(I_n, A) \neq \emptyset, \\ \text{stack-goto } I_p & \text{if } I_p = \text{Next}(I_n\ A) \text{ and } \tau_K(I_n, A) = \emptyset. \end{cases}$$

For a transition $I_q \xrightarrow{X} I_p$, if there exist items $[q]$ and $[q']$ in I_q such that $(I_q, [q])_K T(I_p, [p])$ and $(I_q, [q'])_N T(I_p, [p'])$ then *stacking conflict* of parsing action occurs [9,11]. On the other hand, if $\tau_K(I_q, X) \neq \emptyset$ and $\tau_K(I_q, X) \neq \text{Kernel}(I_p)$ for a transition $I_q \xrightarrow{X} I_p$ then stacking conflict occurs. Therefore, it can be tested by using kernel items.

An RRPG G is said to be LALR(k) RRPG iff ELALR(k) parser for G is deterministic. To make ELALR(k) parser deterministic each entry in parsing table should be unique, in other words must exist no stacking conflict and parsing conflict in the ELALR(k) parser.

Stacking conflicts are resolved indirectly at reduction time by using lookback states [9]. *Lookback states* for a reduction by particular production $A \to r$ in state I_q including an final item $[q]$ of that production are defined as

$$LB(I_q, [q]) = \{I_p \mid I_p \in \text{Source}(I_q, [q]) \wedge q \in Q_F\}$$

where

$$\text{Source}(I_q, [q]) = \{I_r \mid \exists I_r \in I\text{: } (I_r, [r])_N T^*(I_q, [q])\}.$$

Lookback states can be computed efficiently at no additional time by adding the beginning states to each kernel item of a state during the construction of the ELR(0) automaton.

To resolve the stacking conflicts: (i) if stacking conflict of shift action occurs at read time then select stack-shift action, (ii) if there exists stacking conflict transition related to current reduction at reduction time then remove overstacked states from the stack until a lookback state for current reduction appears and perform a reduce action [9]. Also, if stacking conflict of goto occurs then select stack-goto. Additional reduce action is defined as follows:

"reduce $A \to r$ to $LB(I_q, [q])$" is the move

$$
\begin{aligned}
&(I_0\alpha_0 \dots I_n\alpha_n I_q,\ az)\\
&\quad\vdash (I_0\alpha_0 I_1\alpha_1 \dots I_{m-1}\alpha_{m-1} A I_p,\ az)\\
&\quad\text{if } q \in Q_F \text{ for } [A \to :q] \in I_q,\\
&\quad k{:}az \in LA_k(I_q, [A \to r{:}q]),\\
&\quad\text{and Goto}(I_m.\ A) = \text{goto } I_p;\ \text{or}\\
&(I_0\alpha_0 \dots I_n\alpha_n I_q,\ az)\\
&\quad\vdash (I_0\alpha_0 I_1\alpha_1 \dots I_{m-1}\alpha_{m-1} I_m A I_p,\ az)\\
&\quad\text{if } q \in Q_F \text{ for } [A \to r{:}q] \in I_q,\\
&\quad k{:}az \in LA_k(I_q, [A \to r{:}q]),\\
&\quad\text{and Goto}(I_m, A) = \text{stack-goto } I_p
\end{aligned}
$$

where I_m is the topmost I_i such that $I_i \in LB(I_q, [q])$, and $\alpha_m\alpha_{m+1} \dots \alpha_n$ is reduced to A.

The above resolution method can be applied to any ELALR(k) grammar which can be tested during the construction of ELR(0) automaton by the following lemma (the proof of which is analogous to that of the theorem in [9]).

Lemma. *The* ELALR(k) *parser for an RRPG is deterministic iff* (i) *parsing conflicts in inconsistent* ELR(0) *states can be resolved by using lookahead strings of length k and* (ii) *there does not exist a transition* $I_q \xrightarrow{X} I_p$ *such that there exist distinct two items* $[q],[q'] \in I_q$, $[p] \in I_p$, $[r],[r'] \in I_r$, *and*

$$
\begin{aligned}
&(I_q, [q])T(I_p, [p])\\
&\quad\textit{and}\quad (I_q, [q'])T(I_p, [p])
\end{aligned}
$$

where

$$
\begin{aligned}
&(I_r, [r]_K T^*(I_q, [q])\\
&\quad\textit{and}\quad (I_r, [r'])_N T^*(I_q, [q']). \qquad \square
\end{aligned}
$$

Ir $[r] \in \text{Nonkernel}(I_r)$ in the lemma, lookback state for a reduction is unique. And then the handle to be reduced is uniquely determined. Therefore, the ELALR(k) parser for an RRPG is deterministic iff parsing conflicts in inconsistent ELR(0) states can be resolved by using lookahead strings of length k and lookback states.

Algorithm E. Generation of ELALR parser from ELR(0) automaton for RRPG G with underlying item sets of Q.

for $I_q \in I$
  for $X \in V$ where $\exists I_p \in I$: $I_p = \text{Next}(I_q, X)$
    Action$[I_q, X] :=$ shift I_p
      if $\tau_K(I_q, X) = \text{Kernel}(I_p) \wedge X \in \Sigma$;
    Action$[I_q, X] :=$ stack-shift I_p
      if $\tau_K(I_q, X) \neq \text{Kernel}(I_p) \wedge X \in \Sigma$;
    Action$[I_q, X] :=$ goto I_p
      if $\tau_K(I_q, X) = \text{Kernel}(I_p) \wedge X \in N$;
    Action$[I_q, X] :=$ stack-goto I_p
      if $\tau_K(I_q, X) \neq \text{Kernel}(I_p) \wedge X \in N$;
    if $\tau_K(I_q, X) \neq \emptyset$ and
        $\tau_K(I_q, X) \neq \text{Kernel}(I_p)$ **then**
      for $[p] \in \tau_K(I_q, X)$
        mark $[p]$ and all $[r]$ such that
          $(I_p, [p])T^*(I_r, [r])$;
for $I_p \in I$ where $\exists p \in Q_F$: $[A \to q{:}p] \in I_p$
  for $z \in LA_k(I_p, A \to q)$
    Action$[I_p, z] :=$ reduce $A \to q$
      if $[p]$ is not *marked*;
    Action$[I_p, z]$
        $:=$ reduce $A \to q$ to $LB(I_p, [p])$
      if $[p]$ is *marked*;
    Action$[I_p, \$^k] :=$ accept
      if $[S' \to q_{S'}{:}p] \in I_p$;

Note. Algorithm E uses only kernel items. However, when the final right part state of some reduce item is also initial right part state, that is, a production generates ε, the reduce item is nonkernel. Such item should be used to generate ELALR parser in Algorithm E even though it is nonkernel item.

Example. The ELALR(1) parser for $G1$ is shown in Fig. 2 using pictorial representation for parsing table. "|" separates kernel items and nonkernel items of each state. For example, Kernel(I_2) = {[2], [3]} and Nonkernel(I_2) = {[5]}. The edges are labeled by s, g, ss, and sg denoting shift, goto, stack-shift, and stack-goto on X, respectively. A state where a reduction is possible, is annotated by "#" with production number and lookahead set. Also, #1{\$}: $LB = \{I_0\}$ indicates that the action is reduce #1 to lookback state set $\{I_0\}$ with lookahead set {\$}. Moreover, τ_K and relation T only on (state, kernel item) pairs are as follows:

$$\tau_K(I_2, b) = \{[3]\},$$
$$\tau_K(I_2, c) = \tau_K(I_2, A) = \tau_K(I_5, c) = \{[4]\},$$
$$\tau_K(I_3, b) = \tau_K(I_5, b) = \{[3]\},$$
$$\tau_K(I_3, B) = \{[7]\},$$
$$\tau_K(I_3, a) = \tau_K(I_6, c) = \{[8]\},$$
$$\tau_K(I_3, c) = \{[4]\},$$
$$\tau_K(I_8, b) = \{[11]\},$$
$$(I_2, [2])T(I_3, [3]), \quad (I_2, [3])T(I_3, [3]),$$
$$(I_2, [2])T(I_4, [4]), \quad (I_2, [3])T(I_4, [4]),$$
$$(I_3, [3])T(I_5, [3]), \quad (I_3, [3])T(I_8, [4]),$$
$$(I_3, [6])T(I_6, [7]), \quad (I_3, [6])T(I_7, [8]),$$
$$(I_5, [3])T(I_4, [4]), \quad (I_5, [3])T(I_5, [3]),$$
$$(I_6, [7])T(I_7, [8]), \quad (I_8, [10])T(I_9, [11]).$$

Action[I_0, S] = stack-goto I_1 because $\tau_K(I_0, S) = \emptyset$, Kernel($I_1$) = {[12]}, and $S \in N$. Action[I_0, a] = stack-shift I_2 because $\tau_K(I_0, a) = \emptyset$, Kernel($I_2$) = {[2], [3]}, and $a \in \Sigma$. Action[I_2, c] = shift I_4 because $\tau_K(I_2, c) = \{[4]\}$, Kernel(I_4) = {[4]}, and $c \in \Sigma$. Action[I_2, A] = goto I_4 because $\tau_K(I_2, A) = \{[4]\}$, Kernel(I_4) = {[4]}, and $A \in N$.

There are two distinct accessing strings on the sequence of state I_0-I_2-I_4 such as ac and aA. Stacking conflict occurs at the transition $I_2 \xrightarrow{b} I_3$ because $\tau_K(I_2, b) = \{[3]\} \neq \emptyset$ and Kernel(I_3) = {[3], [6]}. Also, stacking conflict occurs at the transition $I_3 \xrightarrow{c} I_8$. Therefore, $[3] \in I_3$ is marked as [3]* and this marking is transferred to $[3] \in I_5$, $[4] \in I_4$, and $[4] \in I_8$ because $(I_3, [3])T(I_5, [3])$, $(I_3, [3])T^*(I_4, [4])$, and $(I_3, [3])T(I_8, [4])$. To resolve stacking conflicts, Action[I_2, b] = stack-shift I_3, Action[I_3, c] = stack-shift I_8. Although $(I_2, [2])T(I_3, [3])$ and $(I_2, [3])T(I_3, [3])$, the handle to be reduced by production $S \to 1$ in state I_4 can be uniquely determined by using lookback state I_0 because $(I_0, [1])_N T(I_2, [2])$ and $(I_0, [1])_N T(I_2, [3])$. Therefore, the action for lookahead {\$} at I_4 and I_8 is "remove the states from stack until I_0 appears, then reduce by the rule #1" because $LB(I_4, [4]) = LB(I_8, [4]) = \{I_0\}$.

If the input string is "$abcbc\$$" then the parsing will proceed as

$$\cdots (I_0 a I_2 b I_3 c I_8,\ bc\$) \vdash_{shift} (I_0 a I_2 b I_3 c b I_9,\ c\$)$$
$$\vdash_{reduce\#3(B \to cb)} (I_0 a I_2 b B I_6,\ c\$)$$

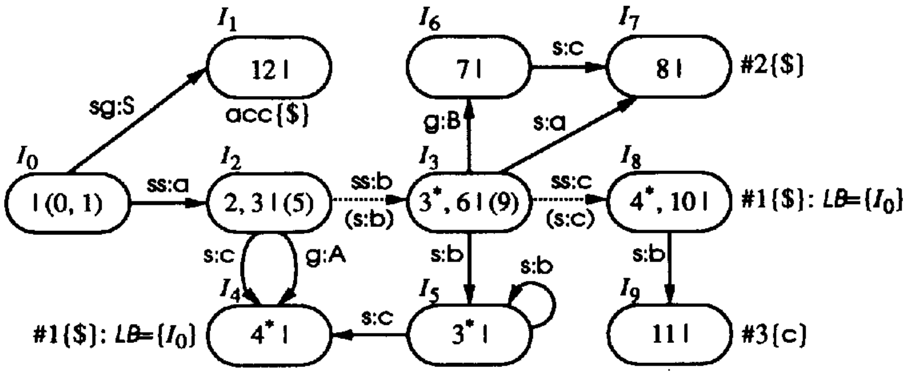

Fig. 2. ELALR(1) parser for $G1$.

$\vdash_{shift}(I_0 a I_2 b B c I_7, \$)$

$\vdash_{reduce\#2(A \to bBc)}(I_0 a A I_4, \$) \cdots$

and if "$abbbc\$$" is given then

$\cdots (I_0 a I_2 b I_3, bbc\$) \vdash_{shift}(I_0 a I_2 bb I_5, bc\$)$

$\vdash_{shift}(I_0 a I_2 bbb I_5, c\$) \vdash_{shift}(I_0 a I_2 bbbc I_4, \$)$

$\vdash_{reduce\#1\ to\ \{I_0\}\ (S \to abbbc)}(I_0 S I_1, \$) \cdots .$

4. Conclusion

We presented an improved method in generation of efficient ELALR(k) parsers for RRPGs, in which only kernel items of the ELR(0) states are used. It is likely to incur large space overheads in explicitly keeping relations between (state, item) pairs [2,9]. It can be reduced by using only kernel items on building the parsers with the new operator τ_K which is used to test and resolve stacking conflicts.

Moreover, in our method, right parts of productions of RRPGs are nondeterministic finite automata. During the construction of ELR(0) automaton, fewer number of items may be required because nondeterministic finite automata can be exponentially more succinct than deterministic finite automata as shown in [12].

References

[1] A.V. Aho and J.D. Ullman, *The Theory of Parsing, Translation, and Compiling, Vols.* 1 *and* 2 (Prentice-Hall, Englewood Cliffs, NJ, 1972 and 1973).

[2] N.P. Chapman, LALR(1, 1) parser generation for regular right part grammars, *Acta Inform.* **21** (1984) 29–45.

[3] F.L. DeRemer, Simple LR(k) grammars, *Comm. ACM* **14** (1971) 453–460.

[4] F.L. DeRemer and T.J. Pennello, Efficient computation of LALR(1) lookahead sets, *ACM Trans. Programming Language Systems* **4** (1982) 615–649.

[5] S. Heilbrunner, On the definition of ELR(k) and ELL(k) grammars, *Acta Inform.* **11** (1979) 169–176.

[6] W.R. LaLonde, Regular right part grammars and their parsers, *Comm. ACM* **20** (1977) 731–741.

[7] W.R. LaLonde, Constructing LR parsers for regular right part grammars, *Acta Inform.* **11** (1979) 177–193.

[8] O.L. Madsen and B.B. Kristensen, LR-parsing of extended context free grammars, *Acta Inform.* **7** (1976) 61–73.

[9] I. Nakata and M. Sassa, Generation of efficient LALR parsers for regular right part grammars, *Acta Inform.* **23** (1986) 149–162.

[10] J.C.H. Park, K.M. Choe and C.H. Chang, A new analysis of LALR formalisms, *ACM Trans. Programming Languages Systems* **7** (1985) 159–175.

[11] P.W. Purdom and C.A. Brown, Parsing extended LR(k) grammars, *Acta Inform.* **15** (1981) 115–127.

[12] S. Sippu and E. Soisalon-Soininen, *Parsing Theory, Vols.* 1 *and* 2 (Springer, Berlin, 1990).