Springer

Dear Author:

Please find attached the final pdf file of your contribution, which can be viewed using the Acrobat Reader, version 3.0 or higher. We would kindly like to draw your attention to the fact that copyright law is also valid for electronic products. This means especially that:

- You may not alter the pdf file, as changes to the published contribution are prohibited by copyright law.

- You may print the file and distribute it amongst your colleagues in the scientific community for scientific and/or personal use.

- You may make an article published by Springer-Verlag available on your personal home page provided the source of the published article is cited and Springer-Verlag is mentioned as copyright holder. You are requested to create a link to the published article in LINK, Springer's internet service. The link must be accompanied by the following text: The original publication is available on LINK **http://link.springer.de.** Please use the appropriate URL and/or DOI for the article in LINK. Articles disseminated via LINK are indexed, abstracted and referenced by many abstracting and information services, bibliographic networks, subscription agencies, library networks and consortia.

- You are not allowed to make the pdf file accessible to the general public, e.g. your institute/your company is not allowed to place this file on its homepage.

- Please address any queries to the production editor of the journal in question, giving your name, the journal title, volume and first page number.

Yours sincerely,

Springer-Verlag Berlin Heidelberg

Acta Informatica 37, 671–697 (2001)

Yet another generation of LALR parsers for regular right part grammars

Shin-ichi Morimoto[1], **Masataka Sassa**[2]

[1] CS Quality Department, NEC Aerospace Systems, Ltd., 2-4-18, Shin-yokohama, Kouhoku-ku, Yokohama, Japan, (e-mail: morimoto@sf.nas.nec.co.jp)
[2] Department of Mathematical and Computing Sciences, Tokyo Institute of Technology, Ookayama, Meguro-ku, Tokyo, Japan, (e-mail: sassa@is.titech.ac.jp)

Received: 23 September 1998 / 16 March 2001

Abstract. In this paper we introduce two methods for building LALR parsers for regular right part grammars (RRPGs). Both methods build a parser directly from a grammar, require no extra state or data structure, and can deal with all LALR RRPGs.

The first method is quite simple. For almost all LALR RRPGs, including the majority of grammars with *stacking conflicts*, parsing actions are similar to those of LALR parsers for usual context free grammars. No extra action is required to recognize a handle in this case. For other LALR RRPGs, the right hand side of a production is checked to recognize a handle.

The second method does not require checking of the right hand side of a production to recognize a handle. Instead, it records the number of conflicts in LR items and in the stack. Unlike previous methods, our method needs no extra data structure.

1 Introduction

A regular right part grammar (RRPG) is a context free grammar in which regular expressions of grammar symbols are allowed in the right hand sides of productions. RRPGs have many advantages over ordinary context free grammars (CFGs), one of which is the fact that the specification of the syntax of programming languages is shorter and easier to understand. An extended LR(k) (ELR(k)) grammar is an RRPG that can be parsed from left to right with a lookahead of k symbols[5]. Hereafter we assume k=1.

Two approaches have been suggested for parsing ELR(1) grammars.
(A) Transform the ELR(1) grammar to an equivalent LR grammar and apply

standard techniques for constructing the LR parser [3],[6].
(B) Build the ELR parser directly from the ELR(1) grammar [1],[4],[7], [8],[9],[10],[12],[13].

Approach (B) is better than approach (A) because the transformation adds inefficiency and makes it harder to determine the semantic structure due to the additional structure added by the transformation. The most critical problem of approach (B) is to identify the left end of a handle at reduction time because the essential difference about LR parsing between RRPGs and CFGs is that the length of a handle of a production is variable in RRPGs while the length is fixed in CFGs.

A popular way to identify the left end of a handle in approach (B) is to use a special move called *stack shift* move, instead of ordinary shift move, to record the left end of a handle when processing the beginning of the right part of the production. However in this framework, there may be a situation when both stack shift move and ordinary shift move are possible. This situation is called a *stacking conflict*. The problem of approach (B) occurs at reduction when stacking conflicts occur.

Many approaches have been suggested to address this problem.
(B1) Add readback states to the parser [1][4]
(B2) Transform to an equivalent RRPG if stacking conflicts occur [8]
(B3) Use lookback states at reduction if stacking conflicts occur [7],[10]
(B4) Use counters at reduction if stacking conflicts occur [9][12]
(B5) Use path numbers at reduction if stacking conflicts occur [13]
Approach (B1) adds inefficiency to the parser. Approach (B2) has some of the same problems as the approach (A) if stacking conflicts occur. Approach (B3) has a problem that some ELR(1) grammars can not be parsed in this approach[13]. Especially grammars with so called *self conflicts* can not be parsed. A self conflict is a situation where stack conflicts occur for the same production. A comment for a lemma in [10] is stated in [2]. Approach (B4) requires extra data structure such as counters for each kernel element of the parser states. Approach (B5) has a shortcoming in which the operation for path numbers is required not only in stack shift moves but also in ordinary shift moves.

In this paper we present two methods based on approach (B). In the first method, a production symbol is pushed to the stack while reading the first symbol of the right hand side of a production, and the stack is popped to the position for the production at reduction time. If there are no self conflicts, this is straightforward. If there are self conflicts, the correct position in the stack can be found by checking the possible handle against the right hand side of the production. Thus, this method can handle all ELR(1) grammars as will be stated in Sect. 3.1. In this method, no extra work is required at

reduction time for almost all ELR(1) grammars, including the majority of grammars with stacking conflicts.

The second method has the same advantage as the first method for grammars without self conflicts. For grammars with self conflicts, this method does not require checking the right hand side of productions to recognize a handle but records the number of self conflicts. This is similar to approach (B4). This method is significantly more efficient than existing methods of (B4) because it does not require extra data structure for retaining information about the number of self conflicts in LR items and the stack.

Hereafter we deal with ELALR(1) parsers in a similar way as with other methods.

2 Notation and terminology

We borrow notation from [7] with slight modifications.

In the following, we represent regular expressions of the right part of productions by deterministic finite state machines.

Definition 1 *A* **regular right part grammar** *is defined as* $G = (V_N, V_T, S, Q, \partial, F, P)$ *where* V_N *is a finite set of nonterminal symbols,* V_T *is a finite set of terminal symbols,* $S \in V_N$ *is a start symbol,* Q *is a finite set of right part states,* $\partial : Q \times V \rightarrow Q$ *is the transition function of the deterministic finite state machines (where* $V = V_N \cup V_T$*),* $F \subset Q$ *are the final states and* P *is a set of productions.*

A production p may be denoted by (A, q) where $A \in V_N$ is the left part and $q \in Q$ is the initial state of the right part of the production.
For a production p, a set of final states in the right part of p is denoted by F_p.

Example 1. A grammar G1 whose productions are shown in Fig. 1 is an RRPG with G1=$(V_N, V_T, S', Q, \partial, F, P)$, where
V_N={S',A}, V_T={a,c,$}
Q={0,1,2,3,4,5,6}
∂(0,A)=1, ∂(1,$)=2, ∂(3,c)=4,
∂(4,A)=∂(4,c)=5, ∂(5,a)=6
F={2,6}, P={ (S',0), (A,3) }
G1 is the same grammar as G2 of [13]. Hereafter a production may be expressed by a production number such as #0 instead of a pair (A, q) such as (S',0).

Definition 2 *We define the relation* $\downarrow$ *on elements of* Q
by $p \downarrow q$ *iff* $\exists A$ *s.t.* $\partial(p,A) \in Q$ *and* $(A,q) \in P$.

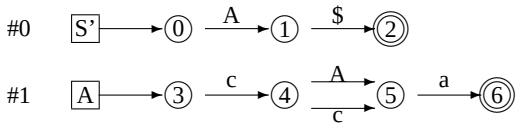

Fig. 1. Regular right part grammar G1

Example 2. In G1, 0↓3, 4↓3 .

We denote the reflexive transitive closure of ↓ by $\downarrow^*$ and the nonreflexive transitive closure of ↓ by $\downarrow^+$.

Definition 3 *For $R(\subseteq Q)$, the closure set of R is defined as*
closure $(R) = \{q \mid \exists p \in R$ *s.t.* $p \downarrow^* q\}$

Definition 4 *An* **LR automaton** *is given by defining the initial state* $\mathbf{q_0}$*, the transition function Next and the reduce function Reduce as*
$\mathbf{q_0}$ = *closure* $(\{q \mid \exists(S, q) \in P\})$
Next (**q**,*X*) = *closure* (*succ* (**q**,*X*))
where succ (**q**,*X*) $= \{\partial$ (*q*,*X*) $\mid q \in \mathbf{q}\}$
Reduce(**q**) $= \{p \mid \exists q \in \mathbf{q}$ *s.t.* $q \in F_p\}$

The set of LR automaton states is denoted by $\mathbf{Q}$ (boldface letters are used to distinguish LR automaton states from right part states).

Definition 5 *For $q_1 \in \mathbf{q_1}$ and $q_2 \in \mathbf{q_2}$, a transition $\rightarrow$ is defined as* $(\mathbf{q_1},q_1) \rightarrow^X (\mathbf{q_2},q_2)$ *iff* $\mathbf{q_2}$=*Next*($\mathbf{q_1}$,*X*) *and* $q_2 = \partial(q_1,X)$.

Definition 6 *Each state* **q** *of an LR automaton has two parts, the* **kernel** *and the* **nonkernel**. *These are defined as*
kernel$(\mathbf{q_0}) = \phi$, *nonkernel*$(\mathbf{q_0}) = \mathbf{q_0}$
kernel(*Next* (**q**,*X*)) =*succ*(**q**,*X*)
nonkernel(**q**) $= \{q \mid \exists\, p \in$ *kernel*(**q**) *s.t.* $p \downarrow^+ q\}$

Definition 7 *For* $q_1 \in \mathbf{q_1}$, $q_2 \in \mathbf{q_2}$ *such that* $(\mathbf{q_1},q_1) \rightarrow^X (\mathbf{q_2},q_2)$,
we write $(\mathbf{q_1},q_1) \rightarrow^{X/K} (\mathbf{q_2},q_2)$ *or* $\mathbf{q_1} \rightarrow^{X/K} \mathbf{q_2}$, *if* $q_1 \in$*kernel*$(\mathbf{q_1})$,
and we write $(\mathbf{q_1},q_1) \rightarrow^{X/N} (\mathbf{q_2},q_2)$ *or* $\mathbf{q_1} \rightarrow^{X/N} \mathbf{q_2}$ *if* $q_1 \in$*nonkernel*$(\mathbf{q_1})$.

Example 3. The LR automaton of G1 is shown in Fig. 2. "|" separates the kernel and the nonkernel of each state. In Fig. 2,
kernel(**0**)=ϕ, nonkernel(**0**) = {0,3},
kernel(**3**)={4}, nonkernel(**3**) = {3}
(**3**,4) $\rightarrow^{c/K}$ (**4**,5), (**3**,3) $\rightarrow^{c/N}$ (**4**,4)

In this paper we deal with LALR parser for RRPGs. As in [7], the corresponding grammar, which is called ELALR grammar, is defined as follows:

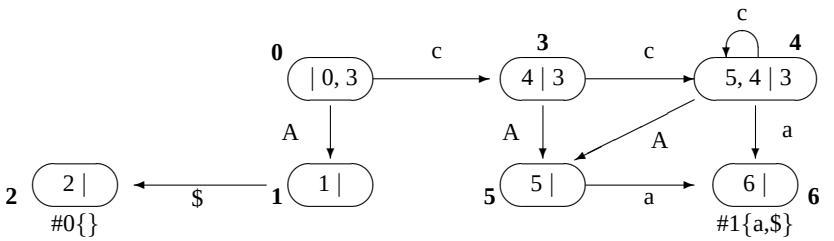

Fig. 2. LR automaton for G1

Definition 8 *An RRPG is said to be an* **ELALR(1) grammar** *if and only if*

1. *parsing conflict in inadequate LR(0) states can be resolved by using lookahead symbols, and*
2. *at any reduction during LALR parsing the handle to be reduced can be uniquely determined.*

For $\mathbf{q}\in\mathbf{Q}$, p∈Reduce($\mathbf{q}$), the set of lookahead symbols of p at $\mathbf{q}$ is denoted as lookahead(p,$\mathbf{q}$).

3 Algorithm

3.1 Moves of the parser

The parsing action of a parser is described in terms of a relation ⊢(move to) defined on configurations. A configuration has a parse stack, a current state and an input string.

At the beginning of the right part of a production, an LR state, a production symbol and a grammar symbol are pushed in the stack, otherwise only a grammar symbol is pushed. When a reduction to a nonterminal *A* occurs, after popping up the handle, *A* is placed on the top of the input string temporarily and then is shifted to the stack.

Definition 9 *A* **configuration** *takes the form*

$(\mathbf{q_0}P_0\beta_0\cdots\mathbf{q_n}P_n\beta_n, \mathbf{q}, z)$

where $\mathbf{q_i}\in\mathbf{Q}$, $P_i\subseteq P$, $\beta_i\in V^*$ $(0\le i\le n)$, $\mathbf{q}\in\mathbf{Q}$, $z\in V\times {V_T}^*$

Definition 10 *Our* **basic ELALR parser** *is composed of the following three kinds of moves.*

1. *shift(X): (if* $\mathbf{q}\rightarrow^{X/K}\mathbf{q'}$*)*
   $(\mathbf{q_0}P_0\beta_0\cdots\mathbf{q_n}P_n\beta_n, \mathbf{q}, Xz)\vdash(\mathbf{q_0}P_0\beta_0\cdots\mathbf{q_n}P_n\beta_nX, \mathbf{q'}, z)$
2. *stack shift(X): (if* $\mathbf{q}\rightarrow^{X/N}\mathbf{q'}$*)*
   $(\mathbf{q_0}P_0\beta_0\cdots\mathbf{q_n}P_n\beta_n, \mathbf{q}, Xz)\vdash(\mathbf{q_0}P_0\beta_0\cdots\mathbf{q_n}P_n\beta_n\mathbf{q}P_{n+1}X, \mathbf{q'}, z)$
   where $P_{n+1}=\{(A,q)|\exists A\in V_N, \exists q'\in\mathbf{q'}$ *s.t.* $(A,q)\in P, (\mathbf{q},q)\rightarrow^{X/N}(\mathbf{q'},q')\}$

3. *reduce(A,p): (if* $\exists p=(A,q'')$
   s.t. $(A,q'') \in Reduce(\mathbf{q})$ *and* $X \in lookahead((A,q''),\mathbf{q}))$
   $(\mathbf{q_0}P_0\beta_0 \cdots \mathbf{q_n}P_n\beta_n, \mathbf{q}, Xz) \vdash (\mathbf{q_0}P_0\beta_0 \cdots \mathbf{q_k}P_k\beta_k, \mathbf{q_{k+1}}, AXz)$
   where P_{k+1} *is the topmost element s.t.* $(A,q'') \in P_{k+1}$ *and* $\beta_{k+1} \ldots \beta_n$ *is a handle of* (A,q'')

Remark 1. The reduce move for production p above means that the stack position up to which the stack is popped is the topmost position that satisfies the following conditions.

1. p was pushed by the stack shift at the position, and
2. the string that was pushed after that position must be accepted by the right hand side of p.

Definition 11 *A* **parser automaton** *is an LR automaton whose actions are moves of a parser as defined above.*

Definition 12 *If there exist* $\mathbf{q}, \mathbf{q'} \in \mathbf{Q}$, $q_1, q_2 \in \mathbf{q}$, $q'_1, q'_2 \in \mathbf{q'}$ *and* $X \in V$ *s.t.* $(\mathbf{q},q_1) \rightarrow^{X/N} (\mathbf{q'},q'_1)$ *and* $(\mathbf{q},q_2) \rightarrow^{X/K} (\mathbf{q'},q'_2)$*, such a case is called a* **stacking conflict** *or a* **stacking conflict with** p*, where* p *is a production that has* q'_1 *in its right part. We call this a* **self conflict***, if further in this case,* p *has both* q'_1 *and* q'_2 *in its right part.*

If a stacking conflict occurs, both shift and stack shift action would be possible, but we select the stack shift as the move of the parser.

In the proposed method, the conflict between parser moves 1,2 and 3 is resolved by the lookahead set, that is by the first condition of the definition of ELALR(1) grammar. Moreover in this method, the stack is popped to the correct position at move 3 because grammar symbols to be popped are checked to be a handle by the second condition of the reduce move, and the handle to be reduced is unique according to the second condition of ELALR(1) grammar. Thus the proposed method can handle all ELALR(1) grammars.

3.2 Example

The parser automaton of G1 is shown in Fig. 3. The notation ss(X) and s(X) denote stack shift(X) and shift(X), respectively. A state where a reduction is possible is annotated by "#" with the production number and lookahead symbols. In Fig. 3, a stacking conflict with #1 occurs in transitions $\mathbf{3} \rightarrow^{\mathbf{c}} \mathbf{4}$ and $\mathbf{4} \rightarrow^{\mathbf{c}} \mathbf{4}$ because
$(\mathbf{3},4) \rightarrow^{c/K} (\mathbf{4},5)$, $(\mathbf{3},3) \rightarrow^{c/N} (\mathbf{4},4)$ and $(\mathbf{4},4) \rightarrow^{c/K} (\mathbf{4},5)$, $(\mathbf{4},3) \rightarrow^{c/N} (\mathbf{4},4)$.

A parsing example for "cccaa" is shown in Fig. 4. In processing a string "cccaa" of G1, the stack position up to which we should pop at "reduce by

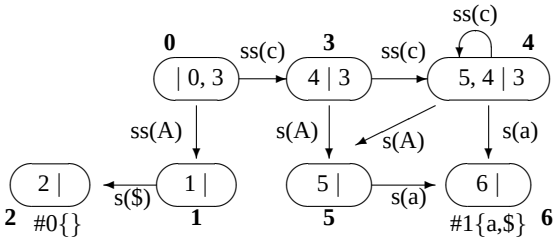

Fig. 3. Parser automaton for G1

parser stack	current state	input	action
0{#1}c**3**{#1}c	**4**	caa$	stack shift(c)
0{#1}c**3**{#1}c**4**{#1}c	**4**	aa$	shift(a)
0{#1}c**3**{#1}c**4**{#1}ca	**6**	a$	reduce by #1(A→cca)
0{#1}c	**3**	Aa$	shift(A)
0{#1}cA	**5**	a$	shift(a)
0{#1}cAa	**6**	$	reduce by #1(A→cAa)
	0	A$	

Fig. 4. Parsing process for “cccaa”

#1” at state **6** (line 3) is determined as follows. The string that was pushed after the last push of #1 is “ca”, which can not be accepted by the right hand side of #1. Therefore this position does not satisfy the above condition for reduction. The string that was pushed after the second to the last push of #1 is “cca”, which can be accepted by the right hand side of #1. Therefore this position satisfies the above condition for reduction.

4 A revised method

4.1 Outline of the revised method

Hereafter we offer a revised method for the method stated in the previous section. In the revised method, we eliminate checking the right hand side of a production to recognize a handle at a reduce move, since it is not efficient in some cases. In such cases, many stack positions for the production symbol to be reduced may have to be checked until the correct position is found, as shown in the above example.

As an alternative, we use revised method where the beginning of the handle can be determined more easily.

To recognize the correct handle at the reduce move by production p without tracing the right hand side of a production, we must know how many production symbol p’s should be popped from the stack. Those p’s

to be popped were pushed in the stack because of self conflicts. Therefore, to recognize the correct handle, we just need to know the number of self conflicts that occurred in parsing the handle. The revised method is based on this principle.

For a handle $v_1...v_n$, a sequence of right part states of a production $q_1...q_n$ corresponding to $v_1...v_n$, is called a *handle path*. The handle path for $v_1...v_n$ *contains a loop* if and only if there exists k, m ($1 \leq k < m \leq$n) s.t. $q_k = q_m$. A handle path for $v_k...v_m$ is called a *self conflict loop* if and only if $q_k = q_m$ and a self conflict occurs in parsing $v_k...v_m$. In this case, the handle path $q_1...q_n$ is said to *contain a self conflict loop*. Hereafter the prefix of a handle path $q_1...q_k$ ($1 \leq k \leq$n) will also be called simply a handle path.

Self conflicts that occur in parsing the handle may arise in two ways.
1) Self conflicts which occur in parsing the nonloop part of the handle path.
2) Self conflicts which occur in parsing the loop part of the handle path.
The number of self conflicts in the first part can be determined during the construction of the parser automaton. The number of self conflicts in the second part cannot be determined during the construction of the parser automaton because the number of loops that the parser goes around is not determined at that time. However, the number of self conflicts occurring in each loop can be determined during parser construction.

Hereafter, we introduce a method in which the number of self conflicts is used in the reduce action for an ELALR grammar that has self conflicts. In this method, the number of self conflicts occurring in the nonloop part of the handle path is recorded in LR items of parser automaton states, and the number of self conflicts occurring in the loop part of the handle path is saved in the parse stack. At the reduce move, information in LR items and in the parse stack is used to count the number of self conflicts that occurred in the handle.

To retain the number of self conflicts in the first case, an LR item of this method is represented by a pair $<q,j>$, where q is a *string of states of right hand side of productions*, and j is the number of self conflicts that occurred in parsing q.

For example, in G1, a self conflict occurs once in parsing an input "cc". Therefore $<$ 345, 1 $>$, a pair consisting of 345 that is a handle path for "cc" and 1 which is the number of self conflicts occurring in parsing "cc", becomes an element of state of the LR automaton.

As for the self conflicts in the second case, a loop may have several routes through it. The number of self conflicts that occurred in a self conflict loop $q_k...q_m$ is recorded in LR automaton, and is saved to the stack each time when parsing q_m.

4.2 Notation and terminology

We denote a set of nonempty string of the right part states of productions by Q^+.

Definition 13 *For $q \in Q^+$, the length of q and the i-th element of q are denoted by* **len(q)** *and* $\mathbf{q^i}$*, respectively. By this notation, q is expressed as $q^1 \cdots q^{len(q)}$. The last element of q is denoted by* **tail(q)**. *A production that has tail(q) in its right part is called a* **production of** q.

Example 4. In G1, for q=3456, len(q)=4, q^1=3, tail(q)=q^4=6, and the production of q is #1.

Definition 14 *For $p \in Q^+$, $q \in Q$, we define the relation $\Downarrow$ by* $p \Downarrow q$ *iff* $\exists A$ *s.t.* ∂*(tail(p),A)*$\in Q$, *(A,q)*$\in P$.

Example 5. In G1, 34$\Downarrow$3.

We denote the reflexive transitive closure of $\Downarrow$ by $\Downarrow^*$ and the nonreflexive transitive closure of $\Downarrow$ by $\Downarrow^+$.

Definition 15 *For $R \subseteq Q^+$, the closure set of R is defined as*
$\overline{closure}(R) = \{q \mid \exists p \in R \text{ s.t. } p \Downarrow^* q\}$

Definition 16 *We define an automaton called a* **base automaton** *with the initial state* $\overline{\mathbf{q_0}}$*, the transition function* $\overline{Next}$ *and the reduce function* $\overline{Reduce}$ *as*

$\overline{\mathbf{q_0}} = \overline{closure}(\{q \mid \exists (S, q) \in P\})$
$\overline{Next}(\overline{\mathbf{q}},X) = \overline{closure}(\overline{succ}(\overline{\mathbf{q}},X))$
$\overline{Reduce}(\overline{\mathbf{q}}) = \{p \mid \exists q \in \overline{\mathbf{q}} \text{ s.t. } tail(q) \in F_p\}$
where $\overline{succ}(\overline{\mathbf{q}},X) = \{\overline{appnd}(q, \partial(tail(q),X)) \mid q \in \overline{\mathbf{q}}\}$
$\overline{appnd}(q,r) = qr$ *(if* $r \neq q^i (1 \leq i \leq len(q)))$
$= q^1 \cdots q^l$ *(if* $r = q^l$*)*

Example 6. For G2 whose productions are shown in Fig. 5,
$\overline{appnd}(345, 7) = 3457$ $\overline{appnd}(3457, 5) = 345$

The set of base automaton states is denoted by $\overline{\mathbf{Q}}$.

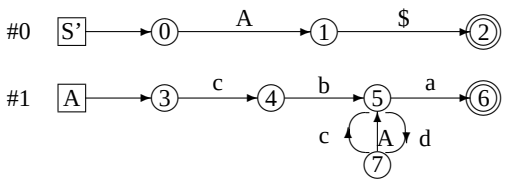

Fig. 5. Regular right part grammar G2

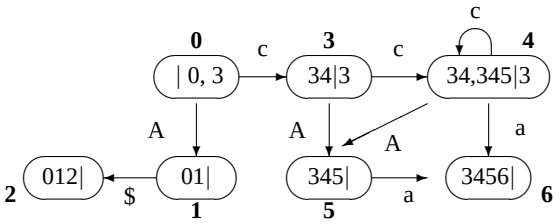

Fig. 6. Base automaton for G1

Definition 17 *For* $\overline{\mathbf{q}} \in \overline{\mathbf{Q}}$*, we define* $\overline{kernel}(\overline{\mathbf{q}})$, $\overline{nonkernel}(\overline{\mathbf{q}})$ *as follows.*

$\overline{kernel}(\overline{\mathbf{q_0}}) = \phi$ $\quad$ $\overline{nonkernel}(\overline{\mathbf{q_0}}) = \overline{\mathbf{q_0}}$

$\overline{kernel}(\overline{Next}(\overline{\mathbf{q}},X)) = \overline{succ}(\overline{\mathbf{q}},X)$

$\overline{nonkernel}(\overline{\mathbf{q}}) = \{q | \exists\, p \in \overline{kernel}(\overline{\mathbf{q}})$ *s.t.* $p \Downarrow^{+} q\}$

We define relations $\rightarrow$, $\rightarrow^{X}$, $\rightarrow^{X/K}$, $\rightarrow^{X/N}$ and the stacking conflict and self conflict similarly to those defined for the parser automaton in Sect. 2.

Definition 18 *We define a predicate SC:* $\overline{\mathbf{Q}} \times V \times Q^{+} \rightarrow \{T,F\}$ *as SC(*$\overline{\mathbf{q}}$*,*X*,*q*)=T iff a stacking conflict with the production of* q *occurs at state* $\overline{\mathbf{q}}$ *for an input* X.

Example 7. The base automaton for G1 is shown in Fig. 6 and for this automaton

$\overline{kernel}(\mathbf{0})=\phi$, $\overline{nonkernel}(\mathbf{0}) = \{0,3\}$

$\overline{kernel}(\mathbf{3})=\{34\}$, $\overline{nonkernel}(\mathbf{3}) = \{3\}$

$(\mathbf{3},34) \rightarrow^{c/K} (\mathbf{4},345)$, $(\mathbf{3},3) \rightarrow^{c/N} (\mathbf{4},34)$

SC($\mathbf{3}$,c,34)=T, SC($\mathbf{0}$,c,3)=F

Definition 19 *For* $\overline{\mathbf{q_i}} \in \overline{\mathbf{Q}}$, $q_i \in \overline{\mathbf{q_i}}$, $X_i \in V (1 \leq i \leq n)$*, we say that* $(\overline{\mathbf{q_i}}, q_i, X_i)(k \leq i \leq n)$ **constitute a loop** *if and only if the following conditions hold.*

- $(\overline{\mathbf{q_1}}, q_1) \rightarrow^{X_1/N} (\overline{\mathbf{q_2}}, q_2)$
- $(\overline{\mathbf{q_i}}, q_i) \rightarrow^{X_i/K} (\overline{\mathbf{q_{i+1}}}, q_{i+1}) (1 < i < n)$
- $\exists k (1 \leq k \leq n)$ *s.t.* $(\overline{\mathbf{q_n}}, q_n) \rightarrow^{X_n/K} (\overline{\mathbf{q_k}}, q_k)$

$(\overline{\mathbf{q_n}}, q_n, X_n)$ *and* $(\overline{\mathbf{q_n}}, X_n)$ *are called the* **loop tail**.

Example 8. The base automaton for G2 is shown in Fig. 7. In this automaton, ($\mathbf{5}$,3457,c) and ($\mathbf{6}$,345,d) constitute a loop and ($\mathbf{6}$,345,d) is the loop tail because

$(\mathbf{0},3) \rightarrow^{c/N} (\mathbf{3},34) \rightarrow^{b/K} (\mathbf{4},345) \rightarrow^{d/K} (\mathbf{5},3457) \rightarrow^{c/K} (\mathbf{6},345) \rightarrow^{d/K} (\mathbf{5},3457)$.

We define the LR automaton for the new method from the base automaton.

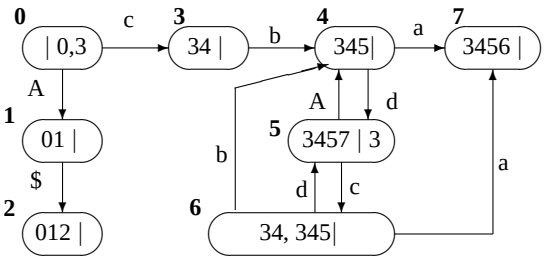

Fig. 7. Base automaton for G2

Definition 20 *The* **LR automaton** *is given by defining the initial state* $\mathbf{q_0}$*, the transition function Next and the reduce function Reduce as*

$\mathbf{q_0} = \{<q,0>|q \in \overline{\mathbf{q_0}}\}$

Next($\mathbf{q}$*,X)* $= \{<q\prime,j\prime>|q\prime \in \overline{Next}$*(Item(*$\mathbf{q}$*),X)*$\}$

Reduce($\mathbf{q}$*)* $= \overline{Reduce}$*(Item(*$\mathbf{q}$*))*

where Item($\mathbf{q}$*)* $= \{q|<q,j> \in \mathbf{q}\}$

For an element $<q\prime,j\prime>$ *of Next(*$\mathbf{q}$*,X),* $j\prime$ *is the number of self conflicts occurring in parsing* $q\prime$*, and* $j\prime$ *is defined as follows:*

1. $j\prime$*=0 iff* $q\prime \notin \overline{succ}$*(Item(*$\mathbf{q}$*),X)*
2. $j\prime$*=*j *iff* $\exists\langle q, j\rangle \in \mathbf{q}$ *s.t.*
   - $q\prime = \overline{appnd}$*(q,* ∂*(tail(q),X))*
   - *(Item(*$\mathbf{q}$*),q,X) is not a loop tail*
   - *SC(Item(*$\mathbf{q}$*),*X*,q)=F*
3. $j\prime = j+1$ *iff* $\exists\langle q, j\rangle \in \mathbf{q}$ *s.t.*
   - $q\prime = \overline{appnd}$*(q,* ∂*(tail(q),X))*
   - *(Item(*$\mathbf{q}$*),q,X) is not a loop tail*
   - *SC(Item(*$\mathbf{q}$*),*X*,q)=T*
4. $j\prime = j\prime\prime$ *iff* $\exists\langle q, j\rangle \in \mathbf{q}$*,* $\exists \mathbf{q}\prime\prime \in \mathbf{Q}$*,* $\exists\langle q\prime\prime, j\prime\prime\rangle \in \mathbf{q}\prime\prime$ *s.t.*
   - *(Item(*$\mathbf{q}$*),q)*$\rightarrow^X$*(Item(*$\mathbf{q}\prime\prime$*),*$q\prime\prime$*)*
   - $q\prime\prime = \overline{appnd}$*(q,* ∂*(tail(q),X))*
   - *(Item(*$\mathbf{q}$*),q,X) is the loop tail*

Boldface letters are used to distinguish LR automaton states from right part states. The set of LR automaton states is denoted by $\mathbf{Q}$.

We define the parser automaton from the LR automaton similarly to Sect. 3.1.

Example 9. The LR automaton for G1 is shown in Fig. 8. Hereafter, an element of the form $<q,0>$ of LR automaton and parser automaton is represented simply by q. An element of a base automaton state is a handle path, while an element of an LR automaton state is a pair consisting of a handle path and the number of self conflicts occurring in parsing the

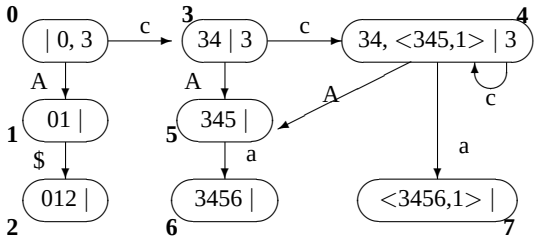

Fig. 8. LR automaton for G1

handle path. Therefore, one state in a base automaton may correspond to several states in an LR automaton. For example, state **6** in Fig. 6 corresponds to state **6** and state **7** in Fig. 8. In LR state **4**, <345,1> is defined from 34=<34,0> of LR state **3** by case 3 of Definition 20. Namely, 345=$\overline{appnd}$(34, ∂(4,c)) and SC(**3**,c,34)=T (cf. Example 7).

Now, in order to count the number of conflicts in a loop of a handle path, we introduce the notion of *inloop conflict information*.

Definition 21 *For* $\mathbf{q_i} \in \mathbf{Q}$, $<q_i,j_i> \in \mathbf{q_i}$, $X_i \in V (1 \leq i \leq n)$, $\mathbf{q_1}$ *is called a* **head state** *of* $\mathbf{q_i}$ *for* q_i *if the following conditions hold.*

- *(Item(*$\mathbf{q_1}$*),*q_1*)* $\rightarrow^{X_1/N}$ *(Item(*$\mathbf{q_2}$*),*q_2*)*
- *(Item(*$\mathbf{q_i}$*),*q_i*)* $\rightarrow^{X_i/K}$ *(Item(*$\mathbf{q_{i+1}}$*),*q_{i+1}*) (*$1<i<n$*)*

Definition 22 *If* $<q_i,j_i> \in \mathbf{q_i}$, $X_i \in V (1 \leq i \leq n)$, $p \in P$ *satisfy the following conditions,*

- *(Item(*$\mathbf{q_i}$*),*q_i*,*X_i*)(*$1 \leq i \leq n$*) constitute a loop*
- *(Item(*$\mathbf{q_n}$*),*q_n*,*X_n*) is the loop tail*
- *A self conflict with* p *occurs in parsing* $X_1 \cdots X_n$ *from* $\mathbf{q_1}$

we call $<p,l>$ **inloop conflict information** *at* $<\mathbf{q_n},X_n>$*, where*
$l=j_n-j_1$ *(if SC(Item(*$\mathbf{q_n}$*),*X_n*,p)=F)*
$l=j_n-j_1+1$ *(if SC(Item(*$\mathbf{q_n}$*),*X_n*,p)=T).*
We call (Item($\mathbf{q_n}$*),*q_n*,*X_n*) and (Item(*$\mathbf{q_n}$*),*X_n*) the* **tail of a self conflict loop**.

Example 10. The LR automaton for G2 is shown in Fig. 9. Similar to Example 8, (Item(**5**),3457,c) and (Item(**6**),345,d) constitute a loop and (Item(**6**), 345,d) is the loop tail. A self conflict for #1 occurs at state **5** for input 'c' and a self conflict for #1 does not occur at state **6** for input 'd', which means SC(Item(**6**),d,#1)=F. Therefore, in this case, $l=j_n-j_1=1-0=1$, and <#1,1> is the inloop conflict information for the loop of (Item(**5**),3457,c) and (Item(**6**),345,d).

Remark 2. In inloop conflict information $<p,l>$ at $<\mathbf{q_n},X_n>$, l represents the number of self conflicts occurred in parsing $X_1 \cdots X_n$ in the loop from

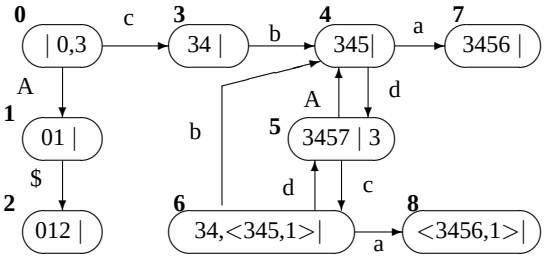

Fig. 9. LR automaton for G2

$\mathbf{q_1}$ for production p. Therefore $l = l'$ if $<p,l>$ and $<p,l'>$ are inloop conflict information for the same loop.

Inloop conflict information is information about the number of self conflicts occurring in a self conflict loop stated in Sect. 4.1. This information is saved to the stack at every transition from the loop tail state of an LR automaton, and is used at reduction time to determine the stack position up to which the stack is popped. However, if the head state of a handle is in a loop, the inloop conflict information of this loop should not be used (see Sect. 5.3 for details). To handle this case, we introduce the notion of an *inloop generated handle prefix*.

Definition 23 *For* $\mathbf{q_i} \in \mathbf{Q}$, $<q_i,j_i> \in \mathbf{q_i}$, $X_i \in V (1 \leq i \leq n)$ *that satisfy the conditions for inloop conflict information, we call* q'_n *an* **inloop generated handle prefix with p** *where* p *is the production of* q'_n, *if* $\exists <q'_1,j'_1> \in \mathbf{q_1}$, $\exists <q'_n,j'_n> \in \mathbf{q_n}$, $\exists k (1 \leq k \leq n)$ *satisfy the following conditions.*

- *(Item(*$\mathbf{q_n}$*),*q'_n*)*$\rightarrow^{X_n}$*(Item(*$\mathbf{q_1}$*),*q'_1*)*
- $\mathbf{q_k}$ *is the head state of* $\mathbf{q_1}$ *for* q'_1
- *A self conflict with* p *occurs at* $\mathbf{q_k}$ *for an input* X_k

An inloop generated handle prefix with p *is also called an* **inloop prefix with p** *for short.*

Example 11. G3 whose productions are shown in Fig. 10 is an example of RRPGs that have inloop prefixes.

The LR automaton for G3 is shown in Fig. 11. In this automaton, (Item($\mathbf{5}$), $v_1v_2v_3$,d) and (Item($\mathbf{9}$),$v_1v_2v_3v_6$,b) constitute a loop and (Item($\mathbf{9}$),b) is the loop tail. Furthermore, v_1 in state $\mathbf{9}$ is an inloop prefix because the following conditions hold.

- (Item($\mathbf{9}$),v_1)$\rightarrow^b$(Item($\mathbf{5}$),v_1v_7)
- State $\mathbf{9}$ is the head state of state $\mathbf{5}$ for v_1v_7
- A self conflict for #1 occurs at state $\mathbf{9}$ for an input 'b'

A handle which has an inloop prefix is called an *inloop generated handle*. If a handle is an inloop generated handle, the inloop conflict information of the loop should not be used at reduction time.

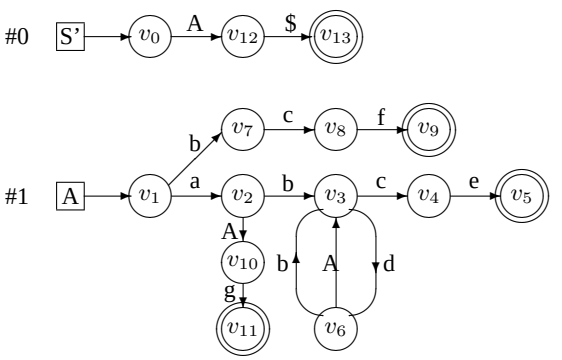

Fig. 10. Regular right part grammar G3

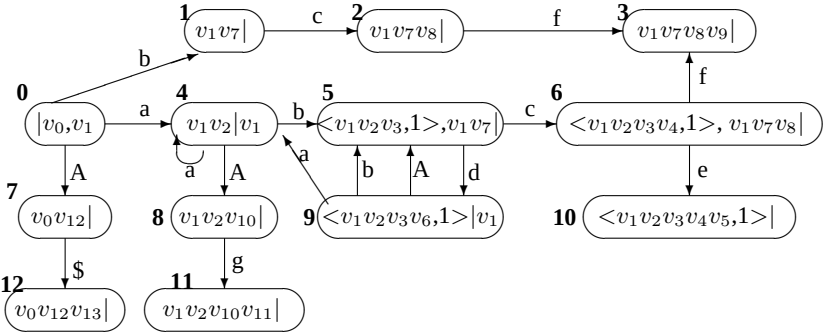

Fig. 11. LR automaton for G3

Definition 24 *A handle $X_1 \cdots X_n$ is called an* **inloop generated handle**, *or an* **inloop handle** *for short, if and only if $\exists k(1 \leq k \leq n)$ s.t. $q_1 \cdots q_k$ is an inloop prefix and $q_1 \cdots q_k$ is the sequence of right part states corresponding to $X_1 \cdots X_k$.*

4.3 Moves of the parser

In this method, a set of inloop conflict information in addition to LR states, grammar symbols and production symbols are pushed in the parse stack at the tail of a self conflict loop.

Definition 25 *A* **configuration** *of this method takes the form:*

($\mathbf{q_0}\overline{P}_0 P_0 \beta_0 \cdots \mathbf{q_n}\overline{P}_n P_n \beta_n$, $\mathbf{q}$, z),

where $\mathbf{q_i} \in \mathbf{Q}$, $\overline{P}_i$ is a set of inloop conflict information, $P_i \subseteq P$, $\beta_i \in V^*$ $(0 \leq i \leq n)$, $\mathbf{q} \in \mathbf{Q}$, $z \in V \times V_T{}^*$.

To identify the left end of a handle at the reduce move, we define $N_p(i)$ for a configuration ($\mathbf{q_0}\overline{P}_0 P_0 \beta_0 \cdots \mathbf{q_n}\overline{P}_n P_n \beta_n$, $\mathbf{q}$, z) at the reduction time. $N_p(i)$ denotes the number of self conflicts with production p occurring in

the nonloop part of a path from the head state for $\mathbf{q}$ to $\mathbf{q_i}$. For example, let $q_1 \cdots q_l$ be a handle path in Item($\mathbf{q}$) and $q_j \cdots q_k$ ($1<j<k<l$) be in Item($\mathbf{q_i}$), $N_p(i)$ denotes the number of self conflicts occurring in the nonloop part of $q_1 \cdots q_k$. The proof that $N_p(i)$ denotes the number is given in the Appendix.

Definition 26 *For a configuration* ($\mathbf{q_0}\overline{P}_0P_0\beta_0 \cdots \mathbf{q_n}\overline{P}_nP_n\beta_n$, $\mathbf{q}$, *z*) *that satisfies* $\exists < q, m > \in \mathbf{q}$, $\exists p \in P$ *s.t. tail(q)*$\in F_p$ *(that is, at the time of reduction by* p*), we define* $N_p(i)$, $L_p(i)$, $C_p(i)$ ($0 \le i \le n$) *as follows:*

$N_p(n+1) = m$

$N_p(i) = N_p(i+1) + L_p(i) - C_p(i)$

$L_p(i) = l$ *(if* $\exists < p, l > \in \overline{P_i}$ *and either a) or b) holds)*

a)Item($\mathbf{q_i}$*) does not contain an inloop prefix with* p

b)$N_p(i+1) \ge l$

$L_p(i) = 0$ *(otherwise)*

$C_p(i) = 1$ *(if* $p \in P_i$*)*

$C_p(i) = 0$ *(if* $p \notin P_i$*)*

Remark 3. $L_p(i)$ counts the number of inloop conflicts. If there is no inloop conflict information $\overline{P_i}$, $L_p(i)$ is 0. If there is an inloop conflict information $< p, l > \in \overline{P_i}$, $L_p(i)$ is that l. The value l is uniquely determined as stated in *Remark 2* about inloop conflict information. However if the handle for this reduction is an inloop handle, $L_p(i)$ is again 0. Condition a) and b) in the definition of $L_p(i)$ check whether the handle for this reduction is an inloop handle or not.

Using Definition 26 and introducing two new moves "save" and "stack save" we get our ELALR parser.

Definition 27 *Our* **revised ELALR parser** *is composed of the following five kinds of moves:*

1. *shift(X): if Item(*$\mathbf{q}$*)*$\rightarrow^{X/K}$*Item(*$\mathbf{q'}$*) and (Item(*$\mathbf{q}$*), X) is not a tail of a self conflict loop.*
   ($\mathbf{q_0}\overline{P}_0P_0\beta_0 \cdots \mathbf{q_n}\overline{P}_nP_n\beta_n$, $\mathbf{q}$, *Xz)*
   $\vdash$ ($\mathbf{q_0}\overline{P}_0P_0\beta_0 \cdots \mathbf{q_n}\overline{P}_nP_n\beta_nX$, $\mathbf{q'}$, *z)*
2. *save(X): if Item(*$\mathbf{q}$*)*$\rightarrow^{X/K}$*Item(*$\mathbf{q'}$*) and (Item(*$\mathbf{q}$*), X) is the tail of a self conflict loop.*
   ($\mathbf{q_0}\overline{P}_0P_0\beta_0 \cdots \mathbf{q_n}\overline{P}_nP_n\beta_n$, $\mathbf{q}$, *Xz)*
   $\vdash$ ($\mathbf{q_0}\overline{P}_0P_0\beta_0 \cdots \mathbf{q_n}\overline{P}_nP_n\beta_n\mathbf{q}\overline{P}_{n+1}\phi X$, $\mathbf{q'}$, *z),*
   where $\overline{P}_{n+1}$ *is the set of inloop conflict information at* $<\mathbf{q}, X>$
3. *stack shift(X): if Item(*$\mathbf{q}$*)*$\rightarrow^{X/N}$*Item(*$\mathbf{q'}$*) and (Item(*$\mathbf{q}$*), X) is not a tail of a self conflict loop.*
   ($\mathbf{q_0}\overline{P}_0P_0\beta_0 \cdots \mathbf{q_n}\overline{P}_nP_n\beta_n$, $\mathbf{q}$, *Xz)*
   $\vdash$ ($\mathbf{q_0}\overline{P}_0P_0\beta_0 \cdots \mathbf{q_n}\overline{P}_nP_n\ \beta_n\mathbf{q}\phi P_{n+1}X$, $\mathbf{q'}$, *z),*
   where $P_{n+1} = \{(A,q) | (A,q) \in P, \exists q' \in$ *Item(*$\mathbf{q'}$*)*
   s.t. (Item($\mathbf{q}$*),q)*$\rightarrow^{X/N}$*(Item(*$\mathbf{q'}$*),q')*$\}$

4. *stack save(X): if Item(**q**)$\rightarrow^{X/N}$Item(**q′**) and (Item(**q**), X) is the tail of a self conflict loop.*
   *(**q**$_0\overline{P}_0P_0\beta_0\cdots$**q**$_n\overline{P}_nP_n\beta_n$, **q**, Xz)*
   $\vdash$ *(**q**$_0\overline{P}_0P_0\beta_0\cdots$**q**$_n\overline{P}_nP_n\,\beta_n$**q**$\overline{P}_{n+1}P_{n+1}X$,**q′**,z),*
   where $P_{n+1}=\{(A,q)|(A,q)\in P, \exists q' \in$ *Item(**q′**)*
   *s.t. (Item(**q**),q)*$\rightarrow^{X/N}$*(Item(**q′**),q′)}*
   $\overline{P}_{n+1}$ *is the set of inloop conflict information at* $<$**q**,X$>$
5. *reduce: if* $\exists$*(A,q″)*$\in P$
   s.t.(A,q″)$\in\overline{Reduce}$*(**q**) and X*$\in$*lookahead((A,q″), **q**)*
   *(**q**$_0\overline{P}_0P_0\beta_0\cdots$**q**$_n\overline{P}_nP_n\beta_n$, **q**, Xz)*
   $\vdash$ *(**q**$_0\overline{P}_0P_0\beta_0\cdots$**q**$_k\ \overline{P}_kP_k\beta_k$, **q**$_{k+1}$, AXz),*
   where $k=max\{i|i<n, N_{(A,q'')}(i+1)<0\}$

Remark 4. save(X) is a move to save a set of inloop conflict information for a self conflict loop at the tail of the loop. stack save(X) is a move to save a set of inloop conflict information as save(X) and to push the production symbol as stack shift(X).

5 Example

5.1 An Example for a grammar without self conflict loops

Consider the RRPG G1 defined in Fig. 1 as an example of RRPGs without self conflict loops. The parser automaton of G1 is shown in Fig. 12. In this automaton, a self conflict with #1 occurs at state **3** for input 'c'. Items $<$345,1$>$ in state **4** and $<$3456,1$>$ in state **7** show that a self conflict has occurred during a path to these items.

A parsing example for "cccaa" is shown in Fig. 13. Values of $L_{\#1}$,$C_{\#1}$ and $N_{\#1}$ in the reduction at state **7** and state **6** are shown in Fig. 14 and Fig. 15, respectively.

As no inloop conflict information is saved in parsing "cccaa", all values of $L_{\#1}$ in Fig. 14 and Fig. 15 are 0. All values of $C_{\#1}$ in Fig. 14 and Fig. 15 are 1 because #1 is pushed at every stack shift move in this parsing.

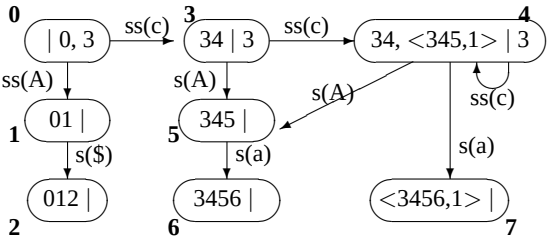

Fig. 12. Parser automaton for G1

parser stack	current state	input	action
$\mathbf{0}\phi${#1}c$\mathbf{3}\phi${#1}c	**4**	caa$	stack shift(c)
$\mathbf{0}\phi${#1}c$\mathbf{3}\phi${#1}c$\mathbf{4}\phi${#1}c	**4**	aa$	shift(a)
$\mathbf{0}\phi${#1}c$\mathbf{3}\phi${#1}c$\mathbf{4}\phi${#1}ca	**7**	a$	reduce by #1(A→cca)
$\mathbf{0}\phi${#1}c	**3**	Aa$	shift(A)
$\mathbf{0}\phi${#1}cA	**5**	a$	shift(a)
$\mathbf{0}\phi${#1}cAa	**6**	$	reduce by #1(A→cAa)
	0	A$	

Fig. 13. Parsing process for "cccaa"

i	0	1	2	3
stack	$\mathbf{0}\phi${#1}c	$\mathbf{3}\phi${#1}c	$\mathbf{4}\phi${#1}ca	
$L_{\#1}$	0	0	0	
$C_{\#1}$	1	1	1	
$N_{\#1}$		-1	0	1

Fig. 14. Values of $L_{\#1}$,$C_{\#1}$,$N_{\#1}$ at state 7 in G1

i	0	1
stack	$\mathbf{0}\phi${#1}cAa	
$L_{\#1}$	0	
$C_{\#1}$	1	
$N_{\#1}$	-1	0

Fig. 15. Values of $L_{\#1}$,$C_{\#1}$,$N_{\#1}$ at state 6 in G1

In Fig. 14, $N_{\#1}(3)$=1 because the element of state **7** for reduction is <3456,1>. $N_{\#1}(2)$ and $N_{\#1}(1)$ at state **7** are evaluated as

$N_{\#1}(2)$=$N_{\#1}(3)$+$L_{\#1}(2)$−$C_{\#1}(2)$=1+0−1=0

$N_{\#1}(1)$=$N_{\#1}(2)$+$L_{\#1}(1)$−$C_{\#1}(1)$=0+0−1=−1

Therefore at this reduction, the stack is popped up to the position for i=1 because the maximum value of i that satisfies $N_{\#1}(i)< 0$ is 1.

In Fig. 15, $N_{\#1}(1)$=0 because the element of state **6** for reduction is <3456,0>. $N_{\#1}(0)$ at state **6** is evaluated as

$N_{\#1}(0)$=$N_{\#1}(1)$+$L_{\#1}(0)$−$C_{\#1}(0)$=0+0−1=−1

Therefore, at this reduction, the stack is popped up to the position for i=0 because the maximum value of i that satisfies $N_{\#1}(i)<0$ is 0.

5.2 An Example for a grammar with self conflict loops

Consider the RRPG G2 defined in Fig. 5 as an example of RRPGs with self conflict loops. The parser automaton of G2 is shown in Fig. 16. The notation sv(x) denotes save(x). In this automaton, a self conflict with #1 occurs at state **5** for input 'c'. Inloop conflict information <#1,1> will be

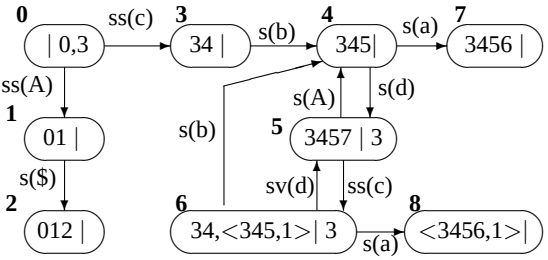

Fig. 16. Parser automaton for G2

parser stack	current state	input	action
$\mathbf{0}\phi\{1\}$cbd$\mathbf{5}\phi\{1\}$cbd	**5**	cdcaa$	stack shift(c)
$\mathbf{0}\phi\{1\}$cbd$\mathbf{5}\phi\{1\}$cbd$\mathbf{5}\phi\{1\}$c	**6**	dcaa$	save(d)
$\mathbf{0}\phi\{1\}$cbd$\mathbf{5}\phi\{1\}$cbd$\mathbf{5}\phi\{1\}$c$\mathbf{6}\{(1,1)\}\phi$d	**5**	caa$	stack shift(c)
$\mathbf{0}\phi\{1\}$cbd$\mathbf{5}\phi\{1\}$cbd$\mathbf{5}\phi\{1\}$c$\mathbf{6}\{(1,1)\}\phi$d$\mathbf{5}\phi\{1\}$c	**6**	aa$	shift(a)
$\mathbf{0}\phi\{1\}$cbd$\mathbf{5}\phi\{1\}$cbd$\mathbf{5}\phi\{1\}$c$\mathbf{6}\{(1,1)\}\phi$d$\mathbf{5}\phi\{1\}$ca	**8**	a$	reduce by #1 (A→cbdcdca)
$\mathbf{0}\phi\{1\}$cbd	**5**	Aa$	shift(A)
$\mathbf{0}\phi\{1\}$cbdA	**4**	a$	shift(a)
$\mathbf{0}\phi\{1\}$cbdAa	**7**	$	reduce by #1 (A→cbdAa)
	0	A$	

Fig. 17. Parsing process for “cbdcbdcdcaa”

pushed at state **6** for input ‘d’ by sv(d) because <#1,1> is the inloop conflict information of the loop constituted by (Item(**5**),3457,c) and (Item(**6**),345,d), and (Item(**6**),345,d) is the loop tail as stated in Example 10.

A parsing example for “cbdcbdcdcaa” is shown in Fig. 17. Values of $L_{\#1}$,$C_{\#1}$ and $N_{\#1}$ in the reduction at state **8** are shown in Fig. 18. In both figures, {#1} and {<#1,1>} are abbreviated as {1} and {(1,1)} respectively. As inloop conflict information is saved only at state **6** in reading ‘d’ and inloop prefixes do not exist in G2, $L_{\#1}(3)=1$ and $L_{\#1}(i)=0$ (i=0,1,2,4). $C_{\#1}(i)=1$ (i=0,1,2,4) because #1 is pushed at every stack shift move, and $C_{\#1}(3)=0$. $N_{\#1}(5)=1$ because the element of state **8** for reduction is <3456,1>. Therefore at this reduction, the stack is popped up to the position for i=1 because the maximum value of i that satisfies $N_{\#1}(i)<0$ is 1.

5.3 An Example for a grammar with inloop prefixes

Consider the RRPG G3 defined in Fig. 10 as an example of RRPGs with inloop prefixes. The parser automaton of G3 is shown in Fig. 19. The notation

i	0	1	2	3	4	5
stack	$\mathbf{0}\phi${1}cbd	$\mathbf{5}\phi${1}cbd	$\mathbf{5}\phi${1}c	$\mathbf{6}${(1,1)}ϕd	$\mathbf{5}\phi${1}ca	
L_1	0	0	0	1	0	
C_1	1	1	1	0	1	
N_1		-1	0	1	0	1

Fig. 18. Values of $L_{\#1}$,$C_{\#1}$,$N_{\#1}$ at state 8 in G2

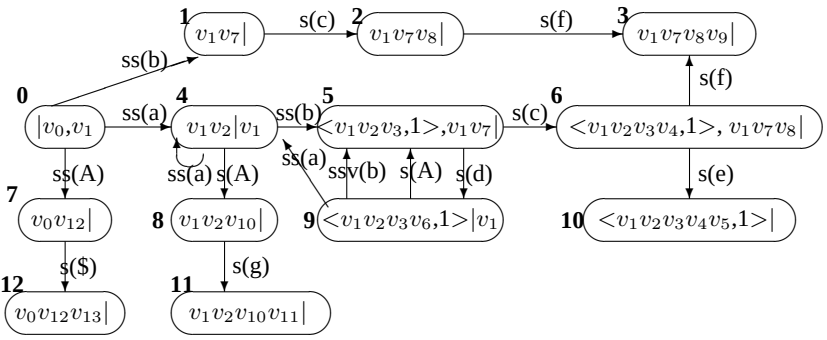

Fig. 19. Parser automaton for G3

parser stack	current state	input	action
$\mathbf{0}\phi${#1}a$\mathbf{4}\phi${#1}bd	**9**	bcfce$	stack save(b)
$\mathbf{0}\phi${#1}a$\mathbf{4}\phi${#1}bd$\mathbf{9}${<#1,1>}{#1}b	**5**	cfce$	shift(c)
$\mathbf{0}\phi${#1}a$\mathbf{4}\phi${#1}bd$\mathbf{9}${<#1,1>}{#1}bc	**6**	fce$	shift(f)
$\mathbf{0}\phi${#1}a$\mathbf{4}\phi${#1}bd$\mathbf{9}${<#1,1>}{#1}bcf	**3**	ce$	reduce by #1 (A→bcf)
$\mathbf{0}\phi${#1}a$\mathbf{4}\phi${#1}bd	**9**	Ace$	shift(A)
$\mathbf{0}\phi${#1}a$\mathbf{4}\phi${#1}bdA	**5**	ce$	shift(c)
$\mathbf{0}\phi${#1}a$\mathbf{4}\phi${#1}bdAc	**6**	e$	shift(e)
$\mathbf{0}\phi${#1}a$\mathbf{4}\phi${#1}bdAce	**10**	$	reduce by #1 (A→abdAce)
	0	A$	

Fig. 20. Parsing process for “abdbcfce”

ss(v) denotes stack save(x). In this automaton, a self conflict with #1 occurs at state **9** for input ‘b’. (Item(**5**),$v_1v_2v_3$,d) and (Item(**9**),$v_1v_2v_3v_6$,b) constitute a loop, and (Item(**9**),$v_1v_2v_3v_6$,b) is the loop tail. Inloop conflict information of this loop is <#1,1>. Therefore, inloop conflict information <#1,1> is pushed to the stack at state **9** for input ‘b’. Moreover, v_1 in state **9** is an inloop prefix as stated in Example 11.

A parsing example for “abdbcfce” is shown in Fig. 20. Consider the reduction ‘A→bcf’ at state **3** (Fig. 21) as an example of reduction for inloop prefixes. In this parsing, inloop conflict information <#1,1> is pushed at state **9** for input ‘b’. However, this inloop conflict information is not con-

i	0	1	2	3
stack	$\mathbf{0}\phi${#1}a	$\mathbf{4}\phi${#1}bd	$\mathbf{9}${<#1,1>}{#1}bcf	
$L_{\#1}$	0	0	0	
$C_{\#1}$	1	1	1	
$N_{\#1}$			-1	0

Fig. 21. Values of $L_{\#1}$,$C_{\#1}$,$N_{\#1}$ at state 3 in G3

parser stack	current state	input	action
$\mathbf{0}\phi${#1}a$\mathbf{4}\phi${#1}bd	**9**	bce$	stack save(b)
$\mathbf{0}\phi${#1}a$\mathbf{4}\phi${#1}bd$\mathbf{9}${<#1,1>}{#1}b	**5**	ce$	shift(c)
$\mathbf{0}\phi${#1}a$\mathbf{4}\phi${#1}bd$\mathbf{9}${<#1,1>}{#1}bc	**6**	e$	shift(e)
$\mathbf{0}\phi${#1}a$\mathbf{4}\phi${#1}bd$\mathbf{9}${<#1,1>}{#1}bce	**10**	$	reduce by #1 (A→abdbce)
	0	A$	

Fig. 22. Parsing process for "abdbce"

sidered in deciding the stack position at the reduce move because the handle of this reduction is an inloop handle as we will show in the following.

Values of $L_{\#1}$,$C_{\#1}$ and $N_{\#1}$ in this reduction are shown in Fig. 21. $N_{\#1}$(3)=0 because the element of state **3** for reduction is $<v_1v_7v_8v_9$,0>. As inloop conflict information is not saved at i=0 and i=1, $L_{\#1}$(0)=0, $L_{\#1}$(1)=0. Inloop conflict information is saved at i=2.
However, $L_{\#1}$(2)=0 because

- Item(**9**) contains an inloop prefix v_1 with #1 as stated before.
- $N_{\#1}$(3)=0<1=l

hold, which means conditions a) and b) in the definition of $L_{\#1}$ of Definition 26 do not hold. All values of $C_{\#1}$ are 1 because #1 is pushed at every stack shift move or stack save move in this parsing. $N_{\#1}$(2) is evaluated as

$N_{\#1}(2)=N_{\#1}(3)+L_{\#1}(2)-C_{\#1}(2)=0+0-1=-1$

Therefore, in this reduce move, the stack is popped up to the position for i=2 because the maximum value of i that satisfies $N_{\#1}$(i)<0 is 2.

As another example, parsing for "abdbce" is shown in Fig. 22. Consider the reduction 'A→abdbce' at state **10** (Fig. 23). In this parsing, an inloop conflict information <#1,1> is also pushed at state **9** for input 'b'. In this case, this inloop conflict information must be considered in deciding the stack position at the reduction because the handle of this reduction is not an inloop handle as we will show in the following.

Values of $L_{\#1}$,$C_{\#1}$ and $N_{\#1}$ in this reduction is shown in Fig. 23. $N_{\#1}$(3)=1 because the element of state **10** for reduction is $<v_1v_2v_3v_4v_5$,1>. As inloop conflict information is not saved at i=0 and i=1, $L_{\#1}$(0)=0, $L_{\#1}$(1)=0. Inloop conflict information is saved at i=2. $L_{\#1}$(2)=1 because

i	0	1	2	3
stack	$\mathbf{0}\phi${#1}a	$\mathbf{4}\phi${#1}bd	$\mathbf{9}${<#1,1>}{#1}bce	
$L_{\#1}$	0	0	1	
$C_{\#1}$	1	1	1	
$N_{\#1}$	-1	0	1	1

Fig. 23. Values of $L_{\#1}$,$C_{\#1}$,$N_{\#1}$ at state 10 in G3

– Item($\mathbf{9}$) contains an inloop prefix v_1 with #1, and
– $N_{\#1}(3)=1\geq 1=l$

hold, which means condition b) in the definition of $L_{\#1}$ holds. All values of $C_{\#1}$ are 1 because #1 is pushed at every stack shift move or stack save move in this parsing. Therefore in this reduce move, the stack is popped up to the position for i=0 because the maximum value of i that satisfies $N_{\#1}(\mathrm{i})<0$ is 0.

6 Conclusion

Two methods for building LALR parsers for regular right part grammars are presented in this paper. Neither method require grammar transformation, and no extra state or data structure is needed. As the production symbols are pushed at the stack shift actions in these methods, extra work at the reduce action is required only when self conflicts occur. Since self conflicts are relatively rare among stacking conflicts, this is a great improvement over existing methods in which extra work is always required when stacking conflicts occur.

The first method of this paper is very simple. Grammars without self conflicts can be parsed as simply as usual LALR grammars. The parsing actions given here for RRPGs without self conflicts differ from the usual LR parsing only with respect to the following points:

– At stack shift actions, the production symbol is pushed to the stack.
– At reduce actions, the stack is popped up to the topmost position corresponding to the production to be reduced.

Therefore, no extra work is required for grammars without self conflicts. Note that self conflicts are quite rare in real RRPGs. For instance, no self conflicts arise in the regular right part grammar of PASCAL-S in [11], and examples of stacking conflicts in previous papers do not contain self conflicts except for the example in [13], which was given as a counter example for the method of [7]. Therefore, almost all RRPGs can be parsed by the first method as simply as usual LALR grammars.

The second method improves on the first method in the amount of extra work at reduction for some cases, as stated in Sect. 4.1.

This method can also be regarded as an improvement of an existing method that uses the number of conflicts to recognize the handle. The novelty of this method resides in the fact that the number of self conflicts need not be counted during parsing in most cases. This concept is based on the following idea.

- The number of self conflicts is defined for the handle path.
- The number for the nonloop part of the handle path is determined in before parsing.
- The number for each loop of the loop part of the handle path is determined in before parsing.

Therefore, most of an evaluation concerning the number of self conflicts can be done at parser construction time.

We think the above idea shows another direction for the solution to parsing problems in general, that is, doing as much of the work as possible at parser construction time. Although this idea has been pointed out several times (e.g. subset construction in transforming nondeterministic finite automata to deterministic ones, and creating states in LR parsing) we believe its merits are substantial. This idea is widely applicable not only to RRPGs with attributes such as the number of self conflicts, but also to RRPGs and CFGs with attributes that satisfy the conditions similar to those, in reducing attribute evaluation during parsing. For example, as the length of the handle satisfies the above conditions, the method of [9] can be improved so that it is similar to our second method.

Acknowledgements. The authors thank two anonymous referees for their many helpful comments that improved the quality of this paper.

References

1. Chapman, N.P: LALR(1,1) parser generation for regular right part grammars, Acta Inf., **21** 29–45 (1984)
2. Galvez, J.F.: A note on a proposed LALR parser for extended context-free grammars, Inf. Process. Lett. **50** 303–305 (1994)
3. Heilbrunner,S. On the definition of ELR(k) and ELL(k) grammars. Acta Inf. **11** 169–176 (1979)
4. LaLonde, W.R.: Constructing LR parsers for regular right part grammars. Acta Inf. **11** 177–193 (1979)
5. Lee, G.-O., Kim, D.-H.: Characterization of extended LR(k) grammars, Inf. Process. Lett. **64** 75–82 (1997)
6. Madsen, O.L., Kristensen, B.B.: LR-parsing of extended context free grammars, Acta Inf. **7** 61–73 (1976)
7. Nakata, I., Sassa,M.: Generation of efficient LALR parsers for regular right part grammars, Acta Inf. **23** 149–162 (1986)

8. Purdom, P.W., Brown, C.A.: Parsing extended LR(k) grammars. Acta Inf. **15** 115–127 (1981)
9. Sassa, M., Nakata, I.: A simple realization of LR parsers for regular right part grammars, Inf. Process. Lett. **24** 113–120 (1987)
10. Shin, H.-C., Choe, K.-M.: An improved LALR(k) parser generation for regular right part grammars, Inf. Process. Lett. **47** 123–129 (1993)
11. Wirth, N.: Pascal S: A Subset and its Implementation, In: Barron,D.W. (ed): Pascal-The Language and its Implementation, pp. 199–259, New York: Wiley 1981.
12. Zhang, Y., Nakata, I., Sassa, M.: A simple realization of efficient LR parsers for regular right part grammars, Trans. IPS Japan, **8**,11(1987) 1162–1167 (in Japanese).
13. Zhang, Y., Nakata, I.: Generation of path directed LALR(k) parsers for regular right part grammars, J. Inf. Process. **14**(3) 325–334 (1991)

A Appendix

In this appendix, we prove the following proposition.

Proposition 1 For a configuration ($\mathbf{q_0}\overline{P}_0P_0\beta_0\cdots\mathbf{q_n}\overline{P}_nP_n\beta_n$, **q**, z) that satisfies $\exists$<q,m>$\in$**q**, $\exists p\in$P s.t. tail(q)$\in F_p$ and for i s.t. $N_p(i)\geq 0$, $N_p(i)$ is the number of self conflicts with p occurring in the nonloop part of a path from $\mathbf{q_I}$ to $\mathbf{q_i}$, where $\mathbf{q_I}$ is the head state for **q** and $\mathbf{q_i}$=**q** if $i = n+1$.

Definition 28 *If q_i,$q\prime_i\in\mathbf{q_i}$, $X_i\in V(1\leq i\leq n)$ satisfy the following conditions,*

- *(Item($\mathbf{q_1}$),q_1)$\rightarrow^{X_1/K}$(Item($\mathbf{q_2}$),q_2)*
- *(Item($\mathbf{q_1}$),$q\prime_1$)$\rightarrow^{X_1/N}$(Item($\mathbf{q_2}$),$q\prime_2$)*
- *(Item($\mathbf{q_i}$),q_i)$\rightarrow^{X_i}$(Item($\mathbf{q_{i+1}}$),q_{i+1}) ($1<i<n$)*
- *(Item($\mathbf{q_i}$),$q\prime_i$)$\rightarrow^{X_i}$(Item($\mathbf{q_{i+1}}$),$q\prime_{i+1}$) ($1<i<n-1$)*
- *∂(tail($q\prime_{n-1}$),X_{n-1})=ϕ*

$\mathbf{q_n}$ *is called a* **resolution state** *for the conflict of* $\mathbf{q_1}$.

Lemma 1 If $q_i\in\mathbf{q_i}$, $X_i\in V(1\leq i\leq n)$ satisfy the following conditions,

- (Item($\mathbf{q_i}$),q_i,X_i)($1\leq i\leq n$) constitute a loop
- (Item($\mathbf{q_n}$),q_n,X_n) is the loop tail
- a self conflict occurs at $\mathbf{q_k}$ for the input X_k

$\exists j(1\leq j\leq n)$ s.t. $\mathbf{q_j}$ is the resolution state for the conflict of $\mathbf{q_k}$.

Proof. Assume that all $\mathbf{q_i}(1\leq i\leq n)$ are not resolution states. Let the self conflict that occurs at $\mathbf{q_k}$ be a self conflict with a production p.

By our assumption, there is no resolution state in the loop $\mathbf{q_k q_{k+1}}\cdots\mathbf{q_n}\cdot\mathbf{q_1}\cdots\mathbf{q_{k-1}}$, which is a sequence of states for $X_kX_{k+1}\cdots X_nX_1\cdots X_{k-1}$ from $\mathbf{q_k}$. Therefore there exists a string β_3 such that $X_k\beta_2\beta_1\beta_3$ and $X_k\beta_2\beta_1\cdot X_k\beta_2\beta_1\beta_3$ are handles of p, where β_1=$X_1\cdots X_{k-1}$ and β_2=$X_{k+1}\cdots X_n$. Let A be a left part symbol of p, then the string $X_k\beta_2\beta_1X_k\beta_2\beta_1\beta_3$ can be reduced to both A and $X_k\beta_2\beta_1$A, which contradicts the condition for the ELALR grammars. □

Proof of Proposition 1

Hereafter the path $\mathbf{q_j}\cdots\mathbf{q_k}$ means the nonloop part of $\mathbf{q_j}\cdots\mathbf{q_k}$, unless otherwise stated and the conflict means the self conflict with p. Let $S_p(i)$ be the number of self conflicts with p occurring in the nonloop part of a path from $\mathbf{q_I}$ to $\mathbf{q_i}$.

We prove

$$N_p(i) = S_p(i) \qquad (*)$$

by induction from $i = n + 1$.

From the definition of state elements of the parser automaton, $S_p(n{+}1){=}$ m and from the definition of N_p, $N_p(n{+}1){=}\, m$.
Therefore (*) holds for $i = n + 1$.

We assume that (*) holds for $i = k{+}1$, which means

$$N_p(k + 1) = S_p(k + 1) \qquad (X)$$

and prove

$$N_p(k) = S_p(k) \qquad (Y)$$

A) Case in which $\mathbf{q_k}$ is not a loop tail for a self conflict loop

In this case,

$S_p(k{+}1){=}S_p(k){+}1$ (if p$\in P_k$) and
$S_p(k{+}1){=}S_p(k)$ (if p$\notin P_k$),

which means

$$S_p(k + 1) = S_p(k) + C_p(k) \qquad (1)$$

Alternatively,

$$N_p(k) = N_p(k + 1) - C_p(k) \qquad (2)$$

because $L_p(k){=}0$ in this case.

From (1),(2),(X)

$$S_p(k){=}S_p(k{+}1){-}C_p(k){=}N_p(k{+}1){-}C_p(k){=}N_p(k)$$

which means that (Y) holds.

B) Case in which $\mathbf{q_k}$ is the loop tail for a self conflict loop L

Let $\mathbf{q'_i}(1{\le}i{\le}r)$ be states that constitute L and $\mathbf{q'_r}$ be the loop tail. In this case $\mathbf{q_k}{=}\mathbf{q'_r}$ and $\mathbf{q_{k+1}}{=}\mathbf{q'_1}$

B$-$1) Case in which Item($\mathbf{q_k}$) does not contain an inloop prefix for p

In this case, $\mathbf{q_I}$ is not a state that constitutes L.

As $\mathbf{q_k}$ is the loop tail of L and $\mathbf{q_k}{\to}\mathbf{q'_1}$, a path $\mathbf{q_I}\cdots\mathbf{q_k}$ consists of a path $\mathbf{q_I}\cdots\mathbf{q'_1}$ and a path $\mathbf{q'_1}\cdots\mathbf{q_k}$. Therefore

$$S_p(k) = S_1 + S_2 \qquad (3)$$

where S_1 is the number of conflicts in $\mathbf{q_I} \cdots \mathbf{q'_1}$ and S_2 is the number of conflicts in $\mathbf{q'_1} \cdots \mathbf{q_k}$. Therefore,

$$S_1 = S_p(k+1) \tag{4}$$

because $\mathbf{q'_1}$=$\mathbf{q_{k+1}}$.

As $\mathbf{q'_1} \cdots \mathbf{q_k}$ constitute L,

l =S_2+1 (if p$\in P_k$)
l =S_2 (if p$\notin P_k$)

which means that

$$l = S_2 + C_p(k) \tag{5}$$

As no inloop prefix exists in L, $L_p(k)$=l. By this and (5),

$$S_2 = L_p(k) + C_p(k) \tag{6}$$

holds.

From (3),(4),(5),(6),(X)

$S_p(k)$=$S_1 + S_2$=$S_p(k+1)$+$L_p(k)$−$C_p(k)$
=$N_p(k+1)$+$L_p(k)$−$C_p(k)$
=$N_p(k)$

which means that (Y) holds.

B−2) Case in which Item($\mathbf{q_k}$) contains an inloop prefix for p

Let q' be the inloop prefix contained in Item($\mathbf{q_k}$) and $\mathbf{q'_s}$ be the head state of $\mathbf{q_k}$ for q'.

From the definition of an inloop prefix, there exists an element q'_1 of $\mathbf{q'_1}$ whose head state is $\mathbf{q'_s}$. Furthermore, there exists a state $\mathbf{q''_s}$ that does not constitute L and $\mathbf{q''_s}$ is the head state for q'_1 because there exists a transition to $\mathbf{q'_1}$ from a state that does not constitute L. Therefore there exist $\mathbf{q''_t}(s \le t \le r)$ that satisfy the following conditions

- $\mathbf{q'_t} \rightarrow^{X_t} \mathbf{q'_{t+1}}$, $\mathbf{q''_t} \rightarrow^{X_t} \mathbf{q''_{t+1}}(s \le t < r)$
- $\mathbf{q'_r} \rightarrow^{X_r} \mathbf{q'_1}$, $\mathbf{q''_r} \rightarrow^{X_r} \mathbf{q'_1}$
- $\mathbf{q'_r} = \mathbf{q_k}$
- $\mathbf{q''_t}(s \le t \le r)$ does not constitute L

$\mathbf{q_k}$, $\mathbf{q'_s}$, $\mathbf{q''_s}$ are shown in Fig. 24.

B−2−a) Case in which $\mathbf{q_I}$=$\mathbf{q'_i}$ for some $i(1 \le i < r)$(Fig. 25)

We can make $\mathbf{q_I} = \mathbf{q'_s}$.

In this case,

$$S_p(k+1) = S_p(k) + C_p(k) \tag{7}$$

because the handle for this reduction is an inloop handle.

As

- $N_p(k+1)$is the number of conflicts in $\mathbf{q'_s} \cdots \mathbf{q'_r q'_1}$, and

file

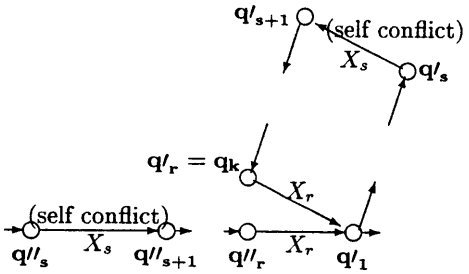

Fig. 24. Example for case B2)

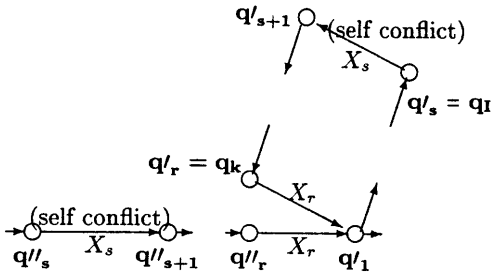

Fig. 25. Example for case B2a)

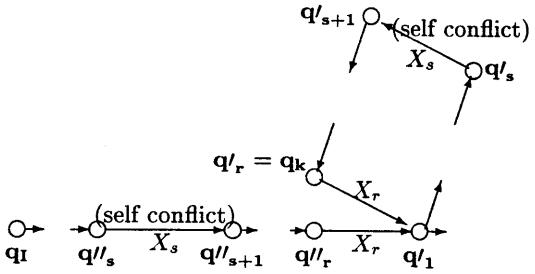

Fig. 26. Example for case B2b)

– l is the number of conflicts in $\mathbf{q'_1}\cdots\mathbf{q'_s}\cdots\mathbf{q'_r}\mathbf{q'_1}$, and
– a self conflict with p occurs at $\mathbf{q'_s}$ $N_p(k{+}1){<}l$

hold, which means

$$L_p(k) = 0 \tag{8}$$

From (7),(8),(X)

$$\begin{aligned} S_p(k){=}S_p(k{+}1){-}C_p(k){=}N_p(k{+}1){-}C_p(k) \\ {=}N_p(k{+}1){+}L_p(k){-}C_p(k) \\ {=}N_p(k) \end{aligned}$$

which means that (Y) holds.

B−2−b) Case in which $\mathbf{q_I}{\neq}\mathbf{q'_i}$ for all $i(1{\leq}i{<}r)$(Fig. 26)

From the lemma, the resolution state for the conflict in $\mathbf{q'_s}$ exists in L. This state exists in the path from $\mathbf{q'_s}$ to the sequence of states that corresponds to $X_s \cdots X_r X_1 \cdots X_{s-1}$.

Therefore there exists a resolution state for the conflict in $\mathbf{q''_s}$ in the path from $\mathbf{q''_s}$ to the sequence of states that correspond to $X_s \cdots X_r X_1 \cdots X_{s-1} \cdots X_k$.

This means that any elements whose head state is $\mathbf{q''_s}$ does not exist in Item($\mathbf{q_k}$). Therefore, the stack position for $\mathbf{q_I}$ is under the position for $\mathbf{q''_s}$.

As $\mathbf{q'_1}$=$\mathbf{q_{k+1}}$, N_p(k+1) is the number of conflicts in $\mathbf{q'_I} \cdots \mathbf{q'_s} \cdots \mathbf{q'_r q'_1}$ and l is the number of conflicts in $\mathbf{q'_1} \cdots \mathbf{q'_s} \cdots \mathbf{q'_r q'_1}$.

As the number of conflicts in $\mathbf{q'_1} \cdots \mathbf{q'_s} \cdots \mathbf{q'_r q'_1}$ equals the number of conflicts in $\mathbf{q''_s} \cdots \mathbf{q''_r q'_1}$, $N_p(k+1) \geq l$ holds, which means that

$$L_p(k) = l \tag{9}$$

Alternatively, as $\mathbf{q'_1}$ exists in the path from $\mathbf{q_I}$ to $\mathbf{q_k}$, a path $\mathbf{q_I} \cdots \mathbf{q_k}$ consists of a path $\mathbf{q_I} \cdots \mathbf{q'_1}$ and a path $\mathbf{q'_1} \cdots \mathbf{q_k}$ as in case B1).

In this case, (3),(4)and(5) also hold as in case B1).

From (3),(4),(5),(9),(X)

$$\begin{aligned} S_p(k) &= S_1 + S_2 = S_p(k+1) + l - C_p(k) \\ &= S_p(k+1) + L_p(k) - C_p(k) \\ &= N_p(k+1) + L_p(k) - C_p(k) = N_p(k) \end{aligned}$$

which means that (Y) holds. □