ISE-TR-86-58

A SIMPLE REALIZATION OF LR PARSERS FOR REGULAR RIGHT PART GRAMMARS

by

Masataka Sassa

and

Ikuo Nakata

July 8, 1986

INSTITUTE
OF
INFORMATION SCIENCES AND ELECTRONICS

UNIVERSITY OF TSUKUBA

A simple realization of LR parsers for regular right part grammars

Masataka Sassa and Ikuo Nakata

Institute of Information Sciences
and Electronics
University of Tsukuba

Abstract

A Regular Right Part Grammar (RRPG) is a context free grammar in which regular expressions of grammar symbols are allowed in the right sides of productions. In this note, a simple method for generating LR parsers for RRPG's is presented.

The idea of the LR parser is to store so-called count values for counting the length of grammar symbols generated by the right side of productions.

Although the parsing efficiency of the method is not the best, the generation of the LR parser is simple and can be done with a slight refinement of the usual LR parser generation techniques. No grammar transformation nor computation of lookback states is necessary.

1. Introduction

A regular right part grammar (RRPG) (or an extended context free grammar) is a context free grammar in which regular expressions of grammar symbols are allowed in the right sides of productions [1-6]. RRPGs are useful for representing the syntax of programming languages naturally and briefly, and are widely used to specify programming languages.

An RRPG is called an ELR(k) grammar if its sentences can be analyzed from left to right by the LR parsing method with a lookahead of k symbols. More precisely, an RRPG is an ELR(k) grammar if (i) $S \overset{+}{\Rightarrow} S$ is impossible and (ii) if $S \overset{*}{\Rightarrow} \alpha Az \Rightarrow \alpha\beta z$, $S \overset{*}{\Rightarrow} \gamma Bx \Rightarrow \alpha\beta y$, and $FIRST_k(z) = FIRST_k(y)$ implies $A = B$, $\alpha = \gamma$, and $x = y$, where all derivations are rightmost [6]. The corresponding parser is called an ELR parser in this paper. In the following, we deal with the case $k=1$.

The main problem with ELR parsing of ELR grammars is in the "reduce" action when the right side of a production is recognized. The problem is that in ELR grammars the length of the sentential form generated by the regular expression of the right side of a production is generally not fixed and therefore, extra work is required to identify the left end of a handle to be reduced.

Three approaches have been proposed so far for ELR parsing of ELR grammars: (1) Transform the ELR grammar to an equivalent LR grammar and apply standard techniques for constructing the LR parser [1,2]. (2) Build the ELR parser directly from the ELR grammar [another method of 1,3,4,5]. (3) A method similar to (2), but transformation to another ELR grammar is necessary in

some cases [6].

In approaches (1) and (3), extra nonterminals are added to the transformed grammar and the correspondence of semantic rules with syntax rules is broken off.

In this paper, we present a simple method based on approach (2). No grammar transformation is necessary. In previous methods based on approach (2), the addition of readback machines [3,4,5] or the investigation of the lookback state [5,8] at reduction time was necessary. The algorithms for these methods were rather complicated. In our method, an ELR parser can be realized with a slight refinement of the usual LR parser technique, by storing so-called count values for counting the length of grammar symbols generated by the right side of productions. Although the parsing efficiency of the method is not the best, the generation of the LR parser is simple and practical.

2. Outline of the method

The outline of our method is explained using the following grammar [3,6].

Example Grammar

G1: #0: S' -> S $

#1: S -> {a} b

#2: S -> a A c

#3: A -> {a}

where {a} means that a is repeated 0 or more times.

Suppose that the input "aaab$" (input 1) is given. This is derived by

$$S' \overset{\#0}{\Longrightarrow} S\$ \overset{\#1}{\Longrightarrow} \{a\}b\$$$

In order to perform correct reductions for such input, it suffices to count the position of each grammar symbol in the right side of the corresponding production. We will save the count values into the usual parsing stack as follows.

parsing stack: a 1 a 2 a 3 b 4 remaining input: $

At the above situation, the parser will reduce by production #1. The number of symbols to be popped from the parsing stack is 4 which can be found at the top of count values.

On the other hand, suppose that the input "aaac$" (input 2) is given. This is derived by

$$S' \overset{\#0}{\Longrightarrow} S\$ \overset{\#2}{\Longrightarrow} aAc\$ \overset{\#3}{\Longrightarrow} a\{a\}c\$$$

Thus, "aa" must be first reduced to "A". In this case, the count values are different from those for input 1. They must proceed as follows.

parsing stack: a 1 a 1 a 2 remaining input: c $

Since there is no distinction between input 1 and 2 during the parsing of "aaa", multiple cases many arise for count values. Thus, we save the multiple count values to handle both cases as follows.

parsing stack: a (1,1) a (2,1) a (3,2)

remaining input: b $ or c $

If the next input symbol is "c", the parser reduces by production #3. Since the count value corresponding to production #3 is the second value, 2, of the array of count values, (3,2), the parser will reduce after popping 2 symbols from the parsing

stack.

3. The proposed ELR parser

The proposed ELR parser for ELR grammars can be organized with a slight refinement of the usual LR parser generation techniques [7].

3.1 Constructing LR states and LR automaton

First, according to the convention of ELR grammars, the regular expression in the right side of a production is represented by the corresponding finite state automaton [6,8]. The finite state automaton is called the right part automaton. In this paper, we assume that it is a deterministic finite state automaton. The right part automata of grammar G1 are shown in Fig. 1.

Similarly, an LR item is represented using the states of the right part automaton. For example, the LR item [S -> · {a} b] of G1 is represented as "3".

A set of LR items or an LR state of an ELR grammar is defined as usual. An LR state is defined as the closure of a set of LR items (kernel of the state). The set of items which are included into the state by the closure operation is called the nonkernel of the state. For example, if LR items "3" ([S -> · {a} b]) and "6" ([S -> a · A c]) are in the kernel of an LR state, LR item "9" ([A -> · {a}]) is included in the nonkernel of this LR state by the closure operation. The LR state is also represented using the states of the right part automaton, e.g. "{ 3, 6 | 9 }". We use "|" to separate the kernel and the

nonkernel of an LR state.

Next, we build an LR automaton as usual using the goto relation and the closure operation. The LR automaton for grammar G1 is shown in Fig. 2. (Fig. 2 contains some refinements as described below.) In this figure, the annotation "#p {l_1, l_2, ...}" denotes the reduce action by production #p when the next input symbol is in the set {l_1, l_2, ...}.

3.2 Refinement of the usual LR automaton

In the following, we borrow some notations from [5] and [7] with slight modifications: Concerning right part automata, Q represents a finite set of right part states (states of the right part automata), $\delta: Q \times V \rightarrow Q$ is the transition function where V is the set of grammar symbols (nonterminals and terminals), and $F \subset Q$ are the final states.

The following notational conventions are used: A, B, C, ... and S, S' are nonterminals; a, b, c, ... are terminals; X, Y, Z, ... are grammar symbols; α, β, γ, ... are strings of grammar symbols, $q_i \in Q$ is a right part state; t_i is an LR item of the form $[q_i, a]$ where a is a lookahead terminal; I_i is a set of LR items; s_i is an LR state. We often identify a set of LR items with an LR state.

Now, we make some refinements of the goto relation. A goto relation usually represents a transition from an LR state to another LR state. Here, in order to deal with multiple possible cases for count values, we introduce a refinement of the goto relation called goto1. It represents a transition from a pair (LR state, LR item) to another pair (LR state, LR item), together

with the information whether the source LR item of the transition is from the kernel or the nonkernel.

Definition (goto1 relation)

Let I_1, I_2 be LR states, and X be a grammar symbol satisfying goto(I_1, X) = I_2. From the definition of goto [7], there should exist LR items $t_1 \in I_1$ and $t_2 \in$ kernel of I_2 such that

$$t_1 = [q_1, a_1] \text{ and } t_2 = [\delta(q_1, X), a_2] \qquad (q_1 \in Q)$$

for some lookahead terminals a_1 and a_2. In this situation, we say that there is a transition by X from (I_1, t_1) to (I_2, t_2).

- If t_1 is in the kernel of I_1, it is denoted by

  goto1$(I_1, t_1), X/+) = (I_2, t_2)$ or

  $(I_1, t_1) \xrightarrow{X/+} (I_2, t_2)$.

- If t_1 is in the nonkernel of I_1, it is denoted by

  goto1$(I_1, t_1), X/1) = (I_2, t_2)$ or

  $(I_1, t_1) \xrightarrow{X/1} (I_2, t_2)$.

Example The arcs in Fig. 2 represent the goto1 relation.

Another modification of the usual LR parsing method is that all LR items in the initial LR state are assumed to be in the nonkernel instead of the kernel. This is for satisfying the property to be described below. An example is the initial state I_0 = { | 0, 3, 5 } of Fig. 2.

With the above refinement, we can consider a sequence of (LR state, LR item) pairs connected by the goto1 relation such that the first pair in the sequence is really the first one connected

by the goto1 relation. Let call it a path [5]. The following property holds for a path.

Property (of a path) A path corresponds to an occurrence of the right side of a production. The first item in a path is a nonkernel item and it corresponds to an initial state in a right part automaton. The remaining items in a path are kernel items. This can be schematically shown as

$$(\underline{I}_1,\underline{t}_1) \xrightarrow{\underline{X}_1/1} (\underline{I}_2,\underline{t}_2) \xrightarrow{\underline{X}_2/+} \cdots (\underline{X}_{n-1},\underline{t}_{n-1}) \xrightarrow{\underline{X}_{n-1}/+} (\underline{I}_n,\underline{t}_n)$$

nonkernel kernel kernel kernel

Example In Fig. 2, the path

$$(\underline{I}_0,3) \xrightarrow{\underline{a}/1} (\underline{I}_3,3) \xrightarrow{\underline{a}/+} (\underline{I}_5,3) \xrightarrow{\underline{a}/+} (\underline{I}_5,3) \xrightarrow{\underline{b}/+} (\underline{I}_4,4)$$

nonkernel kernel kernel kernel kernel

corresponds to an occurrence of the right side of production #1.

3.3 The handling of count stacks in the ELR parser

From the above "Property", we can see that when the parser makes a transition by $\underline{X}$ corresponding to the goto1 relation

$$\xrightarrow{\underline{X}/1}$$

we must initialize the count value to 1 since the first symbol of the right side of a production is read, and when it makes a transition corresponding to

$$\xrightarrow{\underline{X}/+}$$

we must increment the count value by 1 since the parser is processing the middle part of the right side of a production.

In the following, we use a configuration of the ELR parser to denote the status of the parser. Our configuration is

similar to the usual one [7] but is augmented with count values:

$(\underline{s}_0\ \underline{C}_0\ \underline{X}_1\ \underline{s}_1\ \underline{C}_1\ \underline{X}_2\ \underline{s}_2\ \underline{C}_2\ \cdots\ \underline{X}_m\ \underline{s}_m\ \underline{C}_m,\ \ \underline{a}_i\ \underline{a}_{i+1}\ \cdots\ \underline{a}_1\ \$)$ (*)

Here, $\underline{s}_i$ is an LR state, $\underline{C}_i$ is an array of count values, $\underline{X}_i$ is a grammar symbol, and $\underline{a}_i \ldots \underline{a}_1\$$ is the remaining input. $\underline{C}_0$ is actually empty.

A count value in $\underline{C}_i$ designates the length of a path to $\underline{s}_i$ (ending with $\underline{X}_i$). If $\underline{s}_i$ has $\underline{k}_i$ kernel items, then the number of possible paths to $\underline{s}_i$ is $\underline{k}_i$. The count values in $\underline{C}_i$ is stored in a way such that

- $\underline{C}_i[\underline{j}]$, the $\underline{j}$-th element of $\underline{C}_i$, contains the count value of the path to the $\underline{j}$-th LR item in the kernel of the corresponding LR state $\underline{s}_i$.

Thus, the array size of $\underline{C}_i$ is limited by the number of kernel items in $\underline{s}_i$. Note also that cases where multiple count values are necessary correspond to the stacking conflict of [6] and [8].

4. Formalization of the ELR parser and its construction

In this section, we summarize the above discussions and present formally the ELR parser and its construction.

Move (of the ELR parser)

The move of this parser is similar to the usual one [7]. To include the handling of count values, we divide the usual "shift $\underline{s}$" operation into a simple shift and a goto to $\underline{s}$ handled with the count values. Assume that the present configuration is (*) above.

1. If $\underline{action}[\underline{s}_m, \underline{a}_i]$ = "shift" and

$\underline{goto}[\underline{s}_m, \underline{a}_i]$ = "state $\underline{s}$,

incr($(\underline{i}_1, \underline{j}_1)$, $(\underline{i}_2, \underline{j}_2)$ $\cdots$),

init($\underline{k}_1, \underline{k}_2, \cdots$)"

("incr" and "init" indicate the operation to increment and initialize count values, and are constructed by the algorithm shown below. They may be empty),

then the parser enters the configuration

$(\underline{s}_0\ \underline{C}_0\ \underline{X}_1\ \underline{s}_1\ \underline{C}_1\ \underline{X}_2\ \underline{s}_2\ \underline{C}_2\ \cdots\ \underline{X}_m\ \underline{s}_m\ \underline{C}_m\ \underline{a}_i\ \underline{s}\ \underline{C},\ \underline{a}_{i+1}\ \cdots\ \underline{a}_1\$)$

where

$\underline{C}[\underline{j}_1] = \underline{C}_m[\underline{i}_1] + 1,\ \underline{C}[\underline{j}_2] = \underline{C}_m[\underline{i}_2] + 1, \cdots$, and (a)

$\underline{C}[\underline{k}_1] = 1,\ \underline{C}[\underline{k}_2] = 1, \cdots$.

2. If $\underline{action}[\underline{s}_m, \underline{a}_i]$ = "reduce #$\underline{p}$,$\underline{j}$", and

$\underline{goto}[\underline{s}_{m-r}, \underline{A}]$ = "state $\underline{s}$,

incr($(\underline{i}_1, \underline{j}_1), (\underline{i}_2, \underline{j}_2), \cdots$),

init($\underline{k}_1, \underline{k}_2, \cdots$)",

then the parser enters the configuration

$(\underline{s}_0\ \underline{C}_0\ \underline{X}_1\ \underline{s}_1\ \underline{C}_1\ \underline{X}_2\ \underline{s}_2\ \underline{C}_2\ \cdots\ \underline{X}_{m-r}\ \underline{s}_{m-r}\ \underline{C}_{m-r}\ \underline{A}\ \underline{s}\ \underline{C},$

$\underline{a}_i\ \underline{a}_{i+1}\ \cdots\ \underline{a}_1\ \$)$

where

$\underline{r} = \underline{C}_m[\underline{j}]$ if $\underline{j} \neq \varepsilon$ ($\underline{r} = 0$ if $\underline{j} = \varepsilon$. This is an ε-rule),

$\underline{A}$ is the left side of production #$\underline{p}$,

$\underline{C}[\underline{j}_1] = \underline{C}_{m-r}[\underline{i}_1] + 1,\ \underline{C}[\underline{j}_2] = \underline{C}_{m-r}[\underline{i}_2] + 1, \cdots$, and (b)

$\underline{C}[\underline{k}_1] = 1,\ \underline{C}[\underline{k}_2] = 1, \cdots$.

3. If $\underline{action}[\underline{s}_m, \underline{a}_i]$ = "accept", parsing is completed.

4. If $\underline{action}[\underline{s}_m, \underline{a}_i]$ = "error", the parser reports an error.

The construction of the ELR parser is also similar to the usual one [7] except for the handling of count values.

Algorithm (Construction of the ELR parsing table)

Input: A grammar G augmented by production "S' -> S"

Output: ELR parsing table functions action and goto

Method:

1. Construct $\{I_0, I_1, \cdots, I_n\}$, the collection of LR states for G.
2. The parsing actions for state I_i are determined as follows:
   a) If LR item [q,b] is in I_i and
      there is a transition by a from q in the right part automaton (This means that q roughly corresponds to the form "A -> α · a β"),
      then set action[I_i,a] to "shift".
   b) If [q,a] is in I_i and q ∈ F (final states)
      (except for case c)),
      then set action[I_i,a] to "reduce #p,j"
      where production #p has q as a final state in the corresponding right part automaton
      and j is the index of [q,a] in the kernel of I_i.
      (If q is in the nonkernel, then set action[I_i,a] to "reduce #p,ε". It is an ε-rule.)
      (Note. LR items with the same core [7] are collected together.)
   c) If [S'->S$·,] is in I_i, then set action[I_i,] to "accept".
3. The goto transitions for state I_i are determined as follows:
   If goto(I_i,X) = I_j,
      and there exist t_{i1}, t_{i2}, ··· ∈ I_i, t_{j1}, t_{j2}, ··· ∈ I_j
      such that
         goto1((I_i,t_{ik}), X/+) = (I_j,t_{jk})
      where i_1,i_2, ··· and j_1, j_2 ··· are indices of t_{i1}, t_{i2}, ···

and $\underline{t}_{j1}$, $\underline{t}_{j2}$, $\cdots$ in the kernel of $\underline{I}_i$ and $\underline{I}_j$, respectively

and there exist $\underline{t}_{i1}'$, $\underline{t}_{i2}'$, $\cdots \in \underline{I}_i$, $\underline{t}_{j1}'$, $\underline{t}_{j2}'$, $\cdots \in \underline{I}_j$ such that

$\underline{goto1}((\underline{I}_i,\underline{t}_{ik}'), \underline{X}/1) = (\underline{I}_j,\underline{t}_{jk}')$

where j_1', j_2' $\cdots$ are indices of $\underline{t}_{j1}'$, $\underline{t}_{j2}'$, $\cdots$ in the kernel of $\underline{I}_j$ ($\underline{t}_{ik}'$ is in the nonkernel of $\underline{I}_i$),

then set

$\underline{goto}[\underline{I}_i,\underline{X}]$ = "state $\underline{I}_j$,
incr($(\underline{i}_1,\underline{j}_1),(\underline{i}_2,\underline{j}_2), \cdots$)
init($j_1', j_2', \cdots$)"

Example The ELR parsing table for G1 is shown in Fig. 3. In the figure, the action table and the goto table are merged. Depending on the implementation, this may allow a reduction in table space. An example parsing for grammar G1 is shown in Fig. 4.

5. Grammar class

The following holds concerning grammars which can be dealt with by this method.

Theorem An RRPG G can be parsed by the proposed ELR parser if (i) parsing conflicts in inadequate LR states can be resolved by using lookahead symbols, and (ii) for $\underline{I}_1$, $\underline{I}_2$ and $\underline{X}$ satisfying $\underline{goto}(\underline{I}_1, \underline{X}) = \underline{I}_2$, and for each $\underline{t}_2 \in$ kernel of $\underline{I}_2$, there is a unique $\underline{t}_1 \in \underline{I}_1$ which satisfies

$\underline{goto1}((\underline{I}_1,\underline{t}_1), \underline{X}/+) = (\underline{I}_2, \underline{t}_2)$ or

$\underline{goto1}((\underline{I}_1, \underline{t}_1), \underline{X}/1) = (\underline{I}_2, \underline{t}_2)$.

(Proof) We have to show that the "Move (of the ELR parser)" (section 4) is correctly and uniquely defined if (i) and (ii) hold. Since the ELR parser is an extension of the standard LR parser, no parsing conflict occurs if condition (i) holds. What is left is to show that the number of elements to be popped at a reduction is correctly and uniquely determined if (ii) holds. As can be seen from the "Move", possible count values are correctly stored in doing a transition by a grammar symbol. If (ii) holds, the incr or init operation in the goto table (3. of the "Algorithm (Construction of the ELR parsing table)") is determined uniquely for each $\underline{t}_2 \in$ kernel of $\underline{I}_2$. Thus, the operations for assigning values to each count value ((a) and (b) of the "Move") are uniquely defined since those operations are defined from the goto table. At reduction time, "$\underline{r}$" = "$\underline{C}_m[\underline{j}]$" elements of the stack are popped (2. of the "Move"). Since the count value $\underline{C}_m[\underline{j}]$ has been uniquely defined in the above discussion, the reduce operation is correct.

Condition (ii) of the Theorem is essentially the same as condition (ii) of the definition of LALR(1,1) grammar in [5] which says "the readback machines for all reductions are deterministic" and condition (ii) of the theorem for ELALR(1) grammar in [8]. This seems to be an essential condition of the class of ELR grammars for which LR parsers can be directly built.

6. Concluding remarks

We have described a simple realization of ELR parsers for

regular right part grammars. An early idea of this paper appeared in [9]. Other examples and suggestion to a possible optimization are given in [10].

The proposed method belongs to the second of the three approaches given in the introduction. The LR parser is directly built from the given ELR grammar, and no grammar transformation is necessary. The ELR parser can be easily built by a slight refinement of the usual techniques for building the LR parser.

In exchange for the easier parser generation in our method there is some overhead needed for handling count values at parsing time. Our method involves additional count value handling operations during state transitions to simplify the action at reduction time. This is in contrast to other methods [5,8] which do no additional operations during state transitions but which must investigate the readback machine or lookback states at reduction times. But on the whole, we think our method can be a favorable method due to its simplicity in formalization and implementation.

References

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

[2] S. Heilbrunner, On the definition of ELR(k) and ELL(k) grammars, Acta Inf., 11 (1979) 169-176.

[3] W.R. LaLonde, Regular right part grammars and their parsers, Comm. ACM, 20 (10) (1977) 731-741.

[4] W.R. LaLonde, Constructing LR parsers for regular right part grammars, Acta Inf., 11 (1979) 177-193.

[5] N.P. Chapman, LALR(1,1) parser generation for regular right part grammars, Acta Inf., 21 (1984) 29-45.

[6] P.W. Purdom and C.A. Brown, Parsing extended LR(k) grammars, Acta Inf., 15 (1981) 115-127.

[7] A.V. Aho, R. Sethi and J.D. Ullman, Compilers - Principles, Techniques, and Tools, (Addison-Wesley, 1986).

[8] I. Nakata and M. Sassa, Generation of efficient LALR parsers for regular right part grammars, to appear in Acta Inf. (1986).

[9] M. Sassa and I. Nakata, A simple realization of LR parsers for regular right part grammars (short note) (in Japanese), Trans. IPS Japan, 27 (1) (1986).

[10] M. Sassa and I. Nakata, A simple realization of LR parsers for regular right part grammars, Tech. Memo PL-9, Inst. of Inf. Sciences, Univ. of Tsukuba (1985), also to appear in Proc. Symp. on Software Science and Engineering, Res. Inst. for Math. Sci., Kyoto University (1986).

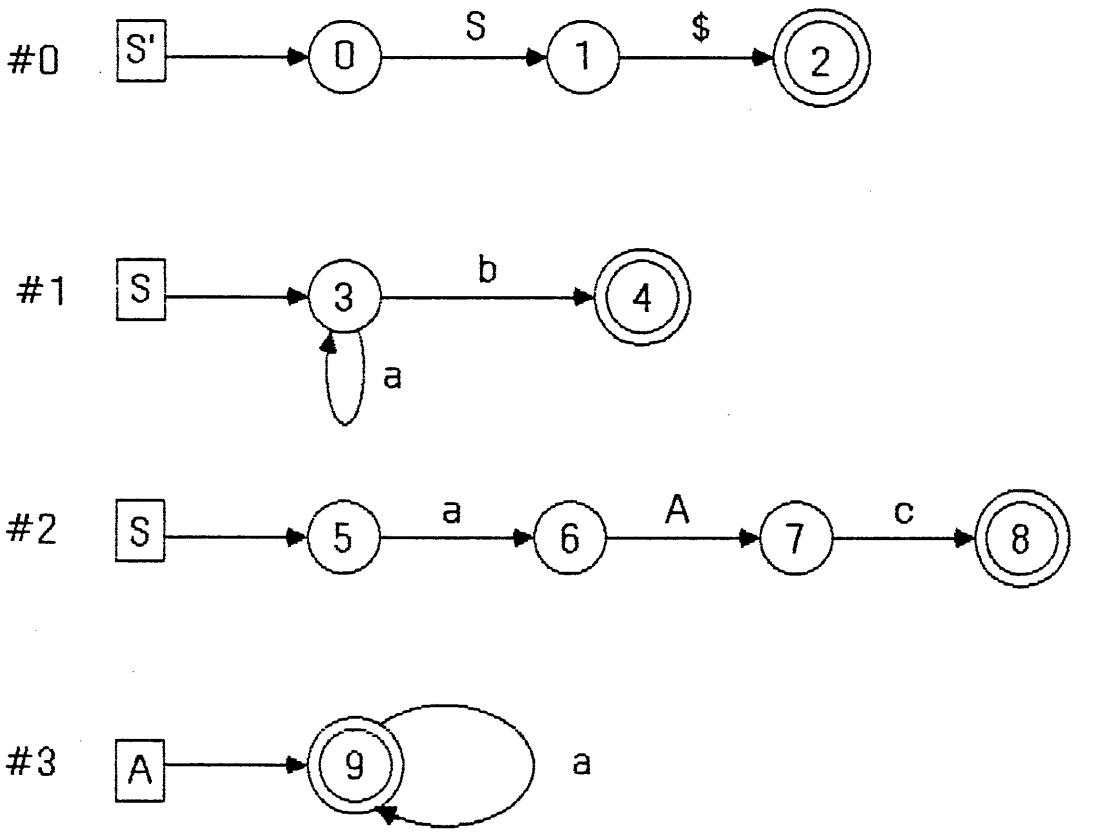

Fig. 1 Representation of regular right part grammar *G1* by right part automata

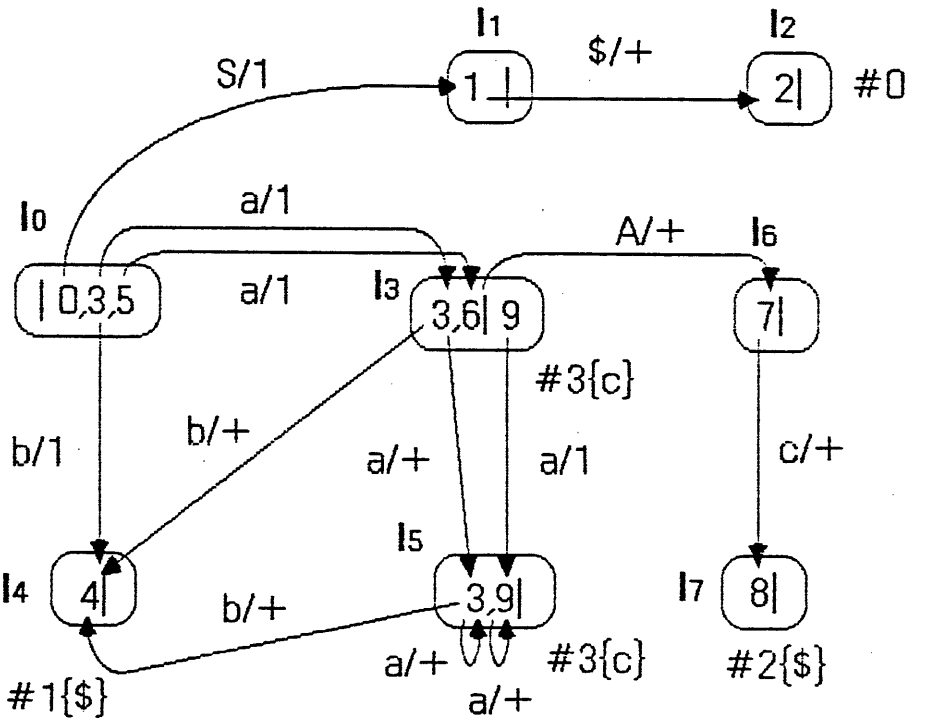

Fig. 2 LR automaton for *G1*

state	*action*				— *goto*	
	a	b	c	$	S	A
I_0	s I_3, i(1,2)	s I_4, i(1)			I_1, i(1)	
I_1				s I_2 +((1,1))		
I_2	(a c c e p t)					
I_3	s I_5, +((1,1)),i(2)	s I_4, +((1,1))	r#3,ϵ			I_6, +((2,1))
I_4				r#1,1		
I_5	s I_5, +((1,1),(2,2))	s I_4, +((1,1))	r#3,2			
I_6			s I_7, +((1,1))			
I_7				r#2,1		

s Ii : shift and goto Ii, Ii : goto Ii,

r#p,j : reduce by production #p for the j-th kernel item,

i(...) : init(...), +(...) : incr(...).

Fig. 3 The ELR parsing table for *G1*

configuration (stack (note 1) , remaining input)	action
(I_0Ø , aaac\$)	shift I_3
⋮	
(I_0Ø$a I_3$(1,1)$a I_5$(2,1)$a I_5$(3,2) , c\$)	reduce by #3, 2 (note 2)
(I_0Ø$a I_3$(1,1)$A I_6$(2) , c\$)	shift I_7
(I_0Ø$a I_3$(1,1)$A I_6$(2)$c I_7$(3) , \$)	reduce by #2, 1
(I_0Ø$S I_1$(1) , \$)	shift I_2
(I_0Ø$S I_1$(1)\$$I_2$(2) ,)	(accept)

(note 1) stack = $s_0C_0X_1s_1C_1X_2.....s_nC_n$ where $C_n = (C_n[1], C_n[2], ...)$

(note 2) pop 2 elements from the stack since $C_n[2] = 2$, and push A

Fig. 4 An example parsing of a sentence generated by *G1* (input = *aaac*\$)

INSTITUTE OF INFORMATION SCIENCES AND ELECTRONICS
UNIVERSITY OF TSUKUBA
SAKURA-MURA, NIIHARI-GUN, IBARAKI 305 JAPAN

REPORT DOCUMENTATION PAGE	REPORT NUMBER
	ISE - TR - 86 - 58

TITLE

A simple realization of LR parsers for regular right part grammars

AUTHOR(S)

Masataka Sassa and Ikuo Nakata

REPORT DATE	NUMBER OF PAGES
July 8th, 1986	abstract, 14p,4 figs.
MAIN CATEGORY	CR CATEGORIES
Parsing	

KEY WORDS

Regular right part grammars, LR parsers

ABSTRACT

A Regular Right Part Grammar (RRPG) is a context free grammar in which regular expressions of grammar symbols are allowed in the right sides of productions. In this note, a simple method for generating LR parsers for RRPG's is presented.

The idea of the LR parser is to store so-called count values for counting the length of grammar symbols generated by the right side of productions.

Although the parsing efficiency of the method is not the best, the generation of the LR parser is simple and can be done with a slight refinement of the usual LR parser generation techniques. No grammar transformation nor computation of lookback states is necessary.

SUPPLEMENTARY NOTES

to appear in Information Processing Letters