数理解析研究所講究録
第586巻 1986年 65-90

A simple realization of LR parsers for regular right part grammars

Masataka Sassa and Ikuo Nakata

（佐々 政孝　　中田 育男）

Institute of Information Sciences and Electronics
University of Tsukuba

（筑波大学 電子・情報工学系）

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 use stacks for counting the length of grammar symbols generated by the right side of a production. The stacks behave synchronously with the parsing stack.

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 little refinement of the standard 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 note. In the following, we deal with the case k=1, and ELR(1)/LR(1) grammars and parsers may simply be called ELR/LR grammars and parsers.

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 there are cases where this method does not work. In these cases transformation to another ELR grammar is necessary [6]. (Note. This does not mean that the grammar class which can be handled by this method is greater than that of approach (2).)

In approaches (1) and (3), extra nonterminals are added to the transformed grammar and the structure of the grammar becomes more complex than the original. In compiler generation, the correspondence of semantic rules with syntax rules is destroyed. From these points, approach (2) seems preferable.

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 allocating so-called count stacks for counting the length of grammar symbols generated by the right side of productions. The stacks behave synchronously with the parsing stack. 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 allocate stack(s) called count stack(s) in addition to the usual parsing stack and save the count values in them as follows.

parsing stack: a a a b remaining input: $

count stack: 1 2 3 4

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 the count stack.

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, the input must be first reduced as

a a a c $

↓

A

In this case, the count values are different from those for input 1. They must proceed as follows.

parsing stack: a a a remaining input: c $

count stack: 1 1 2

Since there is no distinction between input 1 and 2 during the parsing of "aaa", multiple cases many arise for count values. Thus, we will allocate the necessary number of count stacks to handle each case. The present example is processed using two count stacks as follows.

parsing stack: a a a remaining input: b $

count stack 1: 1 2 3 or c $

count stack 2: 1 1 2

If the next input symbol is "b", the parsing proceeds as shown before for input 1. If the next input symbol is "c", the parser will decide that reduction by production #3 should take place at this point. Since the count value corresponding to production #3 is stored in count stack 2 and the top element of this stack is 2, the parser will reduce after popping 2 symbols from the parsing stack.

Cases where multiple count stacks are necessary correspond to the stacking conflict of [6] and [8]. It will be shown later that we need not worry ourselves about a combinatorial explosion of the number of count stacks.

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] for making LR items and LR states (sets of LR items). The detailed formalization of LR items and LR states can be made similarly to [7,6,8]. In the following, this is explained using the example grammar G1.

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. The finite state automaton is called the right part automaton. In this note, 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] is represented as "3" since the LR marker "·" is at state 3 of the right part automaton of Fig. 1. (Here, for the convenience of explanation, we often show only the core [7] of LR items, eliminating the lookahead part.)

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

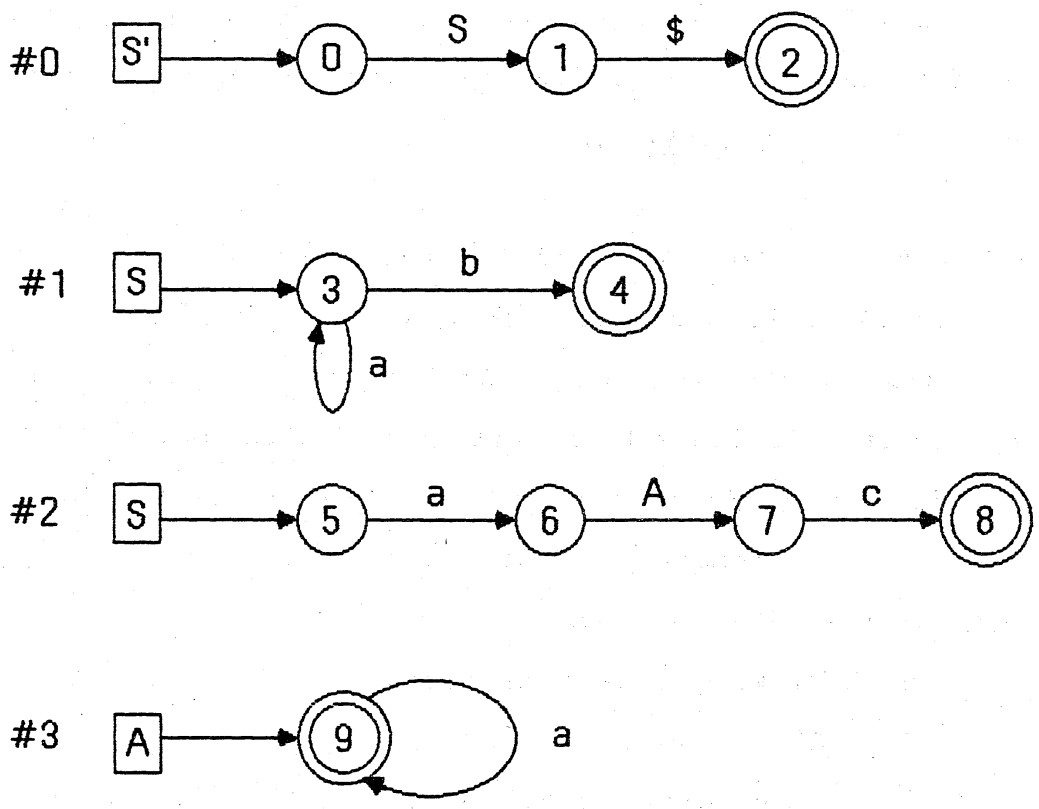


Fig. 1 Representation of regular right part grammar *G1* by right part automata

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, ...}" added to LR states denotes the reduce action by production #p when the parser is in that LR state and 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), δ: Q × V -> 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 new relation called goto1 which is a refinement of the goto relation. A goto1 relation

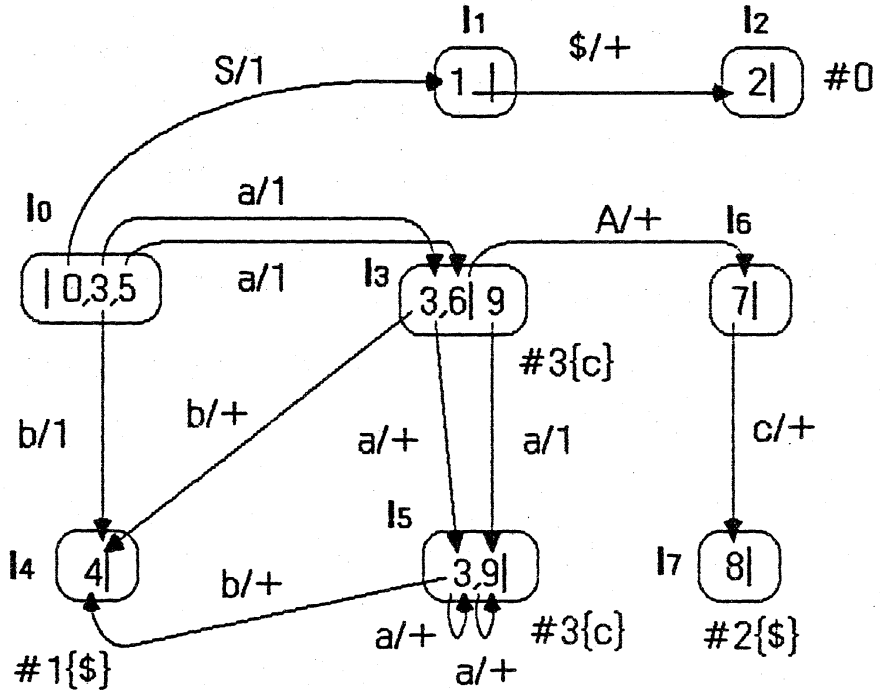


Fig. 2 LR automaton for *G1*

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 $\underline{I}_1$, $\underline{I}_2$ be LR states, and $\underline{X}$ be a grammar symbol satisfying $\underline{goto}(\underline{I}_1, \underline{X}) = \underline{I}_2$. From the definition of goto [7], there should exist LR items $\underline{t}_1 \in \underline{I}_1$ and $\underline{t}_2 \in$ kernel of $\underline{I}_2$ such that

$$\underline{t}_1 = [\underline{q}_1, \underline{a}_1] \text{ and } \underline{t}_2 = [\delta(\underline{q}_1, \underline{X}), \underline{a}_2] \qquad (\underline{q}_1 \in \underline{Q})$$

for some lookahead terminals $\underline{a}_1$ and $\underline{a}_2$. In this situation, we say that there is a transition by $\underline{X}$ from $(\underline{I}_1, \underline{t}_1)$ to $(\underline{I}_2, \underline{t}_2)$.

- If $\underline{t}_1$ is in the kernel of $\underline{I}_1$, it is denoted by

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

  $(\underline{I}_1, \underline{t}_1) \xrightarrow{\underline{X}/+} (\underline{I}_2, \underline{t}_2)$.

- If $\underline{t}_1$ is in the nonkernel of $\underline{I}_1$, it is denoted by

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

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

Example The arcs in Fig. 2 represent the goto1 relation. If the parser makes a transition by "a" from LR state $\underline{I}_3 = \{ 3, 6 \mid 9 \}$ to LR state $\underline{I}_5$, the LR item $(\underline{I}_3, 3)$ in the kernel of $\underline{I}_3$ "goes to" $(\underline{I}_5, 3)$, and the LR item $(\underline{I}_3, 9)$ in the nonkernel of $\underline{I}_3$ goes to $(\underline{I}_5, 9)$. Thus,

$\underline{goto1}((\underline{I}_3, 3), \underline{a}/+) = (\underline{I}_5, 3)$ or $(\underline{I}_3, 3) \xrightarrow{\underline{a}/+} (\underline{I}_5, 3)$

and

$\underline{goto1}((\underline{I}_3, 9), \underline{a}/1) = (\underline{I}_5, 9)$ or $(\underline{I}_3, 9) \xrightarrow{\underline{a}/1} (\underline{I}_5, 9)$.

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 $\underline{I}_0$ = { | 0, 3, 5 } of Fig. 2.

The following is another example of the ELR automaton and the goto1 relation.

Example Consider an RRPG G2 (the same as in Fig. 1 of [5])

S -> A$; A -> [xAy | B] ; B -> b { ab }

where [ ] and | means option and alternation, respectively. This grammar can be represented with right part automata as in Fig. 1b. The ELR automaton and the goto1 relation is shown in Fig. 2b.

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 and the last pair corresponds to a final state in a right part automaton. 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 coresponds to an initial state in a right part automaton. The remaining items in a path are kernel items. This can be schematically shown as

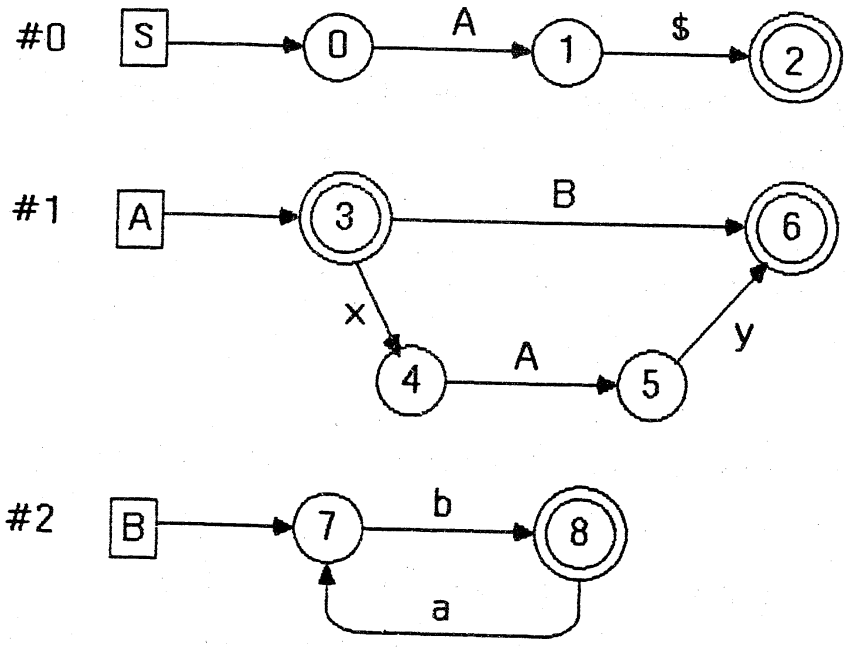


Fig. 1 b Right part automata for *G2*

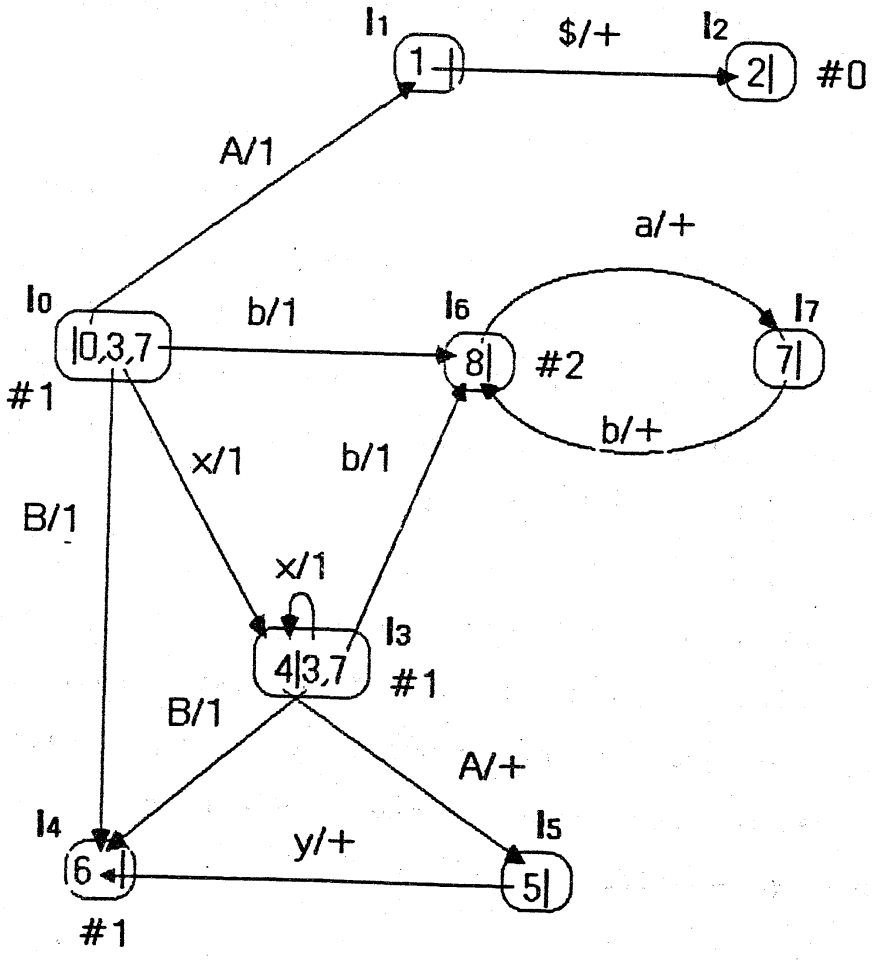


Fig. 2 b LR automaton for *G2*

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

nonkernel kernel kernel kernel

To meet the above property, we have assumed LR items in the initial LR state to be in the nonkernel.

Example 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 "$\underline{S}$ -> {$\underline{a}$} $\underline{b}$".

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

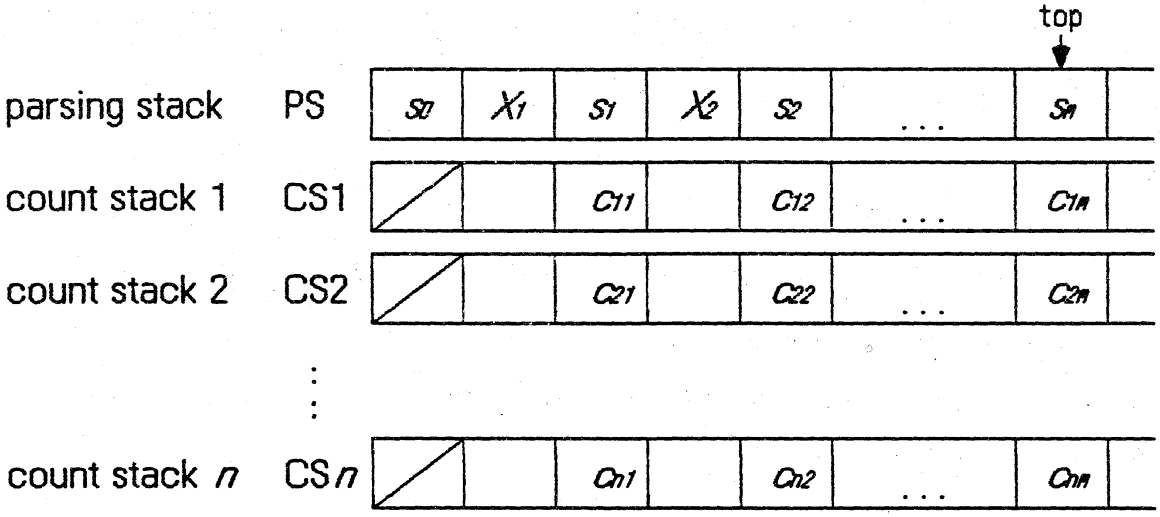
$\xrightarrow{\underline{X}/1}$

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 practical implementation, the organization of the stacks of the ELR parser will be as shown in Fig. 3. The count stacks CSi's behave synchronously with the parsing stack. In Fig. 3, LR states $\underline{s}_i$'s and grammar symbols $\underline{X}_i$'s are stored alternately in the parsing stack PS following the usual convention, and thus count values $\underline{c}_{ij}$'s are stored in every other element of the count



s_i is an LR state, X_i is a grammar symbol, and

c_{ij} is a count value, which may be empy.

n is the maximum number of LR items in the kernel of LR states.

Fig. 3 Stack configuration for an ELR parser

stacks.

The count stacks are used in a way such that the element of the i-th count stack CSi contains the count value of the i-th LR item in the kernel of the corresponding LR state. Thus, the action of the parser concerning the count stacks is as follows.

(1) Action at state transition time

Assume we are at the moment when the parser makes a transition from LR state I_1 to LR state I_2 by pushing the symbol X to the parsing stack. The handling of the count stacks at this moment is as follows.

- If there exist (one or more) t_1 and t_2 such that

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

  where t_1 is the i-th LR item in the kernel of I_1 and

  t_2 is the j-th LR item in the kernel of I_2,

  then perform

  CSj[top] := CSi[top-2] + 1.

- If there exist (one or more) t_1 and t_2 such that

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

  where t_2 is the j-th LR item in the kernel of I_2,

  (t_1 should be in the nonkernel of I_1),

  then perform

  CSj[top] := 1.

Example If the parser for G1 (Fig. 2) is at LR state I_3 and the next input symbol is "a", the parser goes to LR state I_5 and performs

CS1[top] := CS1[top-2]+1 ; CS2[top] := 1

As can be seen from the above action, it is natural to consider that the count values synchronize with the LR states in the parsing stack rather than the grammar symbols.

It is also clear that the number of count stacks CSi's is limited by the number of LR items in the largest kernel of the set of LR states, where largest kernel means the kernel containing the most items. Thus, some elements of the count stacks will not be used for LR states with fewer than the maximum number of kernel items.

(2) Action at reduction time

When the parser is at LR state I annotated with #p{l_1, l_2, ...}, and if the next input symbol belongs to the lookahead set {l_1, l_2, ...}, the parser makes a reduction by production #p. Elements of the parsing stack and the count stacks must be popped. The number of elements to be popped is CSi[top]×2 if the LR item in I corresponding to the final state of the right part automaton for production #p is the i-th LR item in the kernel. However, no popping of stacks is performed at a reduction by an e- rule.

Example If the parser of G1 (Fig. 2) is at LR state I_5, and the next input symbol is "c", the parser reduces by production #3. The number of elements to be popped from the stacks is CS2[top]×2. (This is because the LR item "9" which corresponds to production #3 is the 2nd item in the kernel.)

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.

A configuration of the ELR parser is similar to the usual one [Aho] but is augmented with count values:

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

where $\underline{s}_i$ is an LR state, $\underline{X}_i$ is a grammar symbol, $\underline{C}_i$ is an array[1..$\underline{n}$] of count values ($\underline{n}$ is the maximum number of LR items in the kernel of LR states) and $\underline{a}_i \ldots \underline{a}_1\$$ is the remaining input. $\underline{C}_0$ is empty. $\underline{C}_m[\underline{i}]$ in the configuration corresponds to $\underline{c}_{im}$ or CS$\underline{i}$[top] of Fig. 3.

Move (of the ELR parser)

The move of this parser is also 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 stacks. Assume that the present configuration is (*).

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

   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" or "init" is constructed below. It 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)

$C[k_1] = 1,\ C[k_2] = 1,\ \cdots$.

2. If $action[s_m, a_i]$ = "reduce #p",

   $r = C_m[i]$ (i is the index of the LR item in the kernel of s_m corresponding to the final state for production #p),

   and

   $goto[s_{m-r}, A]$ = "state s,

   $incr((i_1,j_1),(i_2,j_2),\ \cdots\ )$,

   $init(k_1,k_2,\ \cdots\ )$",

   then the parser enters the configuration

   $$(s_0\ C_0\ X_1\ s_1\ C_1\ X_2\ s_2\ C_2\ \cdots\ X_{m-r}\ s_{m-r}\ C_{m-r}\ A\ s\ C,\ a_i\ a_{i+1}\ \cdots\ a_1\ \$)$$

   where

   $C[j_1] = C_{m-r}[i_1] + 1,\ C[j_2] = C_{m-r}[i_2] + 1,\ \cdots$

   and (b)

   $C[k_1] = 1,\ C[k_2] = 1,\ \cdots$.

3. If $action[s_m, a_i]$ = "accept", parsing is completed.

4. If $action[s_m, a_i]$ = "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 stacks.

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 by production #p" where production #p corresponds to q.

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 t_{j1}, t_{j2}, ··· in the kernel of I_i and I_j, respectively

and there exist t_{i1}', t_{i2}', ··· ∈ I_i, t_{j1}', t_{j2}', ··· ∈ I_j such that

goto1((I_i,t_{ik}'), X/1) = (I_j,t_{jk}')

where j_1', j_2' ··· are indices of t_{j1}', t_{j2}', ··· in the kernel of I_j (t_{ik}' is in the nonkernel of I_i),

then set

goto[I_i,X] = "state I_j,
incr((i_1,j_1),(i_2,j_2), ···)
init(j_1',j_2', ···)"

Example The ELR parsing table for G1 is shown in Fig. 4. 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. 5.

5. Discussions

5.1 Grammar class

Since the proposed ELR parsing method is an extension of the usual LR parsing method in the handling of the count stacks, 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 I_1, I_2 and X satisfying $goto(I_1, X) = I_2$, and for each $t_2 \in$ kernel of I_2, there is a unique $t_1 \in I_1$ which satisfies

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

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

(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

state	*action* 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			I_6, +((2,1))
I_4				r#1		
I_5	s I_5, +((1,1),(2,2))	s I_4, +((1,1))	r#3			
I_6			s I_7, +((1,1))			
I_7				r#2		

s Ii : shift and goto Ii, Ii : goto Ii, r#p : reduce by production #p,

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

Fig. 4 The ELR parsing table for *G1*

stacks	remaining input	action
PS: I_0 CS1: CS2:	aaab$	shift
: :		
$I_0 a I_3 a I_5 a I_5$ 1 2 3 1 1 2	b$	shift
$I_0 a I_3 a I_5 a I_5 b I_4$ 1 2 3 4 1 1 2	$	reduce by #1. pop 4x2 elements(note) and push S
$I_0 S I_1$ 1	$	shift
$I_0 S I_1 \$ I_2$ 1 2		(accept)

(note) pop CS1[top] x 2 elements since the parser reduces by (I_4, 4)

which is the 1st item in the kernel.

(a) input = "aaab$"

Fig. 5 An example parsing of
a sentence generated by *G1*

stacks	remaining input	action
PS: I_0 CS1: CS2:	aaac$	shift
:		
$I_0 a I_3 a I_5 a I_5$ 1 2 3 1 1 2	c$	reduce by #3. pop 2x2 elements(note) and push A
$I_0 a I_3 A I_6$ 1 2 1	c$	shift
$I_0 a I_3 A I_6 c I_7$ 1 2 3 1	$	reduce by #2. pop 3x2 elements and push S
$I_0 S I_1$ 1	$	shift
$I_0 S I_1 \$ I_2$ 1 2		(accept)

(note) pop CS2[top] x 2 elements since the parser reduces by (I_5, 9) which is the 2nd item in the kernel.

(b) input = "aaac$"

Fig. 5 An example parsing of a sentence generated by *G1*

"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 stack, ((a) and (b) of the "Move") are uniquely defined since those operations are defined from the goto table. At reduction time, "$\underline{r}$" = "$\underline{C}_m[\underline{i}]$" elements of the stacks are popped (2. of the "Move"). Since the count stack number $\underline{i}$ is clearly uniquely determined and the count value $\underline{C}_m[\underline{i}]$ is 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.

So long as this condition is satisfied, the proposed method can be generally applied to regular right part grammars including productions of any form such as

$\underline{A} \rightarrow \alpha \{ \beta \} \gamma \{ \delta \} \eta$ or

$\underline{B} \rightarrow \xi (\mu \mid \nu) \zeta$

where α, β etc. are strings of grammar symbols.

5.2 Trivial optimization

For a state of a right part automaton whose distance from the start state of the right part automaton is fixed, the handling of count stacks can be dispensed with. Since this optimization is trivial, its description is omitted here.

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].

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 the count stacks at parsing time. Our method involves count stack operations during state transitions to simplify the action at reduction time. This may be compared 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.

What is the frequency of regular expressions in the productions in practical situations? An example is the description of Pascal by the compiler generator GAG [10]. Among 141 productions, 20 productions (14%) use regular expressions in the right sides. One possible reason for this rather small number seems to be that the techniques for semantic analysis of regular right part grammars are still not well established.

The proposed ELR parser is not the best in terms of parsing efficiency, but we think it can be a favorable method due to its simplicity in formalization and implementation.

Acknowledgements

The authors wish to thank David Duncan for polishing up the English in this paper.

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] U. Kastens, B. Hutt, and E. Zimmermann, GAG: A practical compiler generator, Lec. Notes in Comp. Sci. 141 (Springer, 1982).