Regular Paper

Generation of Path Directed LALR(k) Parsers for Regular Right Part Grammars

YOUPU ZHANG* and IKUO NAKATA**

A regular right part grammar (RRPG) is a context-free grammar in which regular expressions of grammar symbols are allowed in the right part of productions. An RRPG is called an ELR grammar if its sentences can be analyzed by the LR-parsing method. An efficient method for building ELR parsers is given in this paper.

In order to identify the left end of a handle at reduction time, an LR state symbol is pushed onto the parser stack only when the corresponding transition is from a nonkernel item of the state. In the case of stacking conflict, that is, if there are two corresponding transitions, one from a nonkernel item and the other from a kernel item, the LR state symbol is pushed onto the parser stack. Furthermore, the path number that records the transition path is pushed onto the parser stack in order to solve the stacking conflict.

The grammar class of our method is larger than that of [2], [7] and ESLR(k). We discuss the relations between these methods and present some comments on a previous method [7]. Our method is simple and efficient, and no grammar transformation or computation of lookback/readback states is necessary.

1. Introduction

A Regular Right Part Grammar (RRPG) (or an extended context-free grammar) is obtained by allowing regular expressions on the right part of productions of a context-free grammar. An RRPG is called an ELR(k) grammar if $S \Rightarrow {}^{+}S$ is impossible and if $S \Rightarrow {}^{*}\alpha Az \Rightarrow \alpha\beta z$, $S \Rightarrow {}^{*}\gamma Bx \Rightarrow \alpha\beta y$, and $first_k(z)=first_k(y)$ implies $A=B$, $\alpha=\gamma$, and $x=y$, where S is the start symbol and all derivations are rightmost [4, 9]. According to Heilbrunner [4, 5], there are two conditions equivalent to the above definition, which are (1) an RRPG is called an ELR(k) grammar if some unambiguous right linear transformation yields an LR(k) grammar, and (2) an RRPG is called an ELR(k) grammar if its ELR(k) automaton is consistent. RRPGs have many advantages over ordinary context-free grammars: the specification of the syntax of a programming language is shorter, easier to construct, and easier to understand; the corresponding syntax-directed parser may also be shorter and more efficient. The main problem with ELR parsing of ELR grammars is to identify the left end of a handle at reduction time. Several approaches to this problem have been proposed [2, 4, 6, 7, 9, 10]. Heilbrunner's [4] and Madsen's [6] methods require a transformation of an ELR(k) grammar into an equivalent LR(k) grammar. Purdom's method [9] requires a transformation of an ELR(k) grammar into an equivalent ELR(k) grammar to avoid stacking conflicts. The grammar transformation adds extra nonterminals to the grammar, and therefore makes the resultant parser inefficient. Furthermore, it often destroys the semantic structure (such as the priority and associativity of operators). Several methods for building the ELR parsers directly from ELR grammars have been given. Chapman's method [2] is based on adding readback states to usual LR parsers. Nakata's method [7] is based on lookback states. Sassa's method [10] is based on counter stacks.

The subject of this paper is the parser generation for RRPGs. Our method does not require the transformation of grammars. It uses so-called path numbers instead of readback states, lookback states, or counter stacks. When the end of the right part of a production is found, exactly one state entry is popped from the parser stack, except when stacking conflicts have occurred while reading the right part of the production, in which case several entries may be popped until the correct path number for the reduction appears. The grammar class for which our method is applicable is larger than that of [2], [7] and ESLR(k) (extended simple LR(k)) grammars.

The path method parser for a given grammar is constructed first by building an LR(0) automaton, which is augmented with lookahead symbols, and finally by adding some path actions to the pushdown machine if necessary. Therefore, it is possible to take advantage of the efficient lookahead algorithm of Park [8] in this method.

Section 2 presents our terminology, some basic definitions, and algorithms. The path method and the algorithms for parser construction are presented in Sec-

*School of Computer Science, McGill University, McConnell Engineering Building, 3480 University Street Montréal Québec Canada H3A 2A7.
**Institute of Information Science and Electronics, University of Tsukuba, Tsukuba, Ibaraki 305, Japan.

tion 3. Section 4 discusses the grammar class that corresponds to the path method. Section 5 presents an optimization algorithm and a parser generation algorithm for the path method. Finally, in Section 6, the advantages and efficiency of the path method are discussed.

2. Terminology, Definitions, and Basic Algorithms

We assume that the readers are familiar with the basic notation and definitions of the LR(k) parsing theory such as *first*$_k$, *follow*$_k$, lookahead, LR(k), LALR(k), SLR(k) and rightmost derivation [1, 5]. In this paper, all derivations are assumed to be rightmost derivations.

We use the definitions and notations of Chapman [2] with minor modifications. In the following, we represent regular expressions by deterministic finite state machines. A regular right part grammar is written 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 the start symbol, Q is a finite set of right part states, ∂: $Q\times V\to Q$ is the transition function (where $V=V_N\cup V_T$), $F\subset Q$ is a set of final states, and $P\subset V_N\times Q$ is a set of productions. A production is denoted by (A, p), where $A\in V_N$ is the left part and $p\in Q$ is the initial state of the right part deterministic automaton of the production. This is equivalent to the conventional notation $A\to\alpha$, where α is a regular expression and p is the initial state of the deterministic automaton for α. ∂ is extended to the mapping ∂': $Q\times V^*\to Q$ as follows:

$$\partial'(q, \varepsilon)=q$$
$$\partial'(q, \beta X)=\partial(\partial'(q, \beta), X) \text{ where } X\in V, \beta\in V^*$$

If $(A, p)\in P$, we write $L(A, p)=\{\alpha\in V^*\mid\partial'(p, \alpha)\in F\}$. For each $p\in Q$ such that $\exists A$: $(A, p)\in P$, we define the set of states accessible from p by $Q_p=\{q\mid\exists\alpha\in V^*: q=\partial'(p, \alpha)\}$. We assume that

1. the sets Q_p and $Q_{p'}$ are disjoint if $p\neq p'$;
2. grammars are reduced and in augmented form, that is, there is a single state $q_0\in Q$ such that $(S, q_0)\in P$ and $L(S, q_0)=\{S'\$\}$ for some $S'\in V_N$ and $\$\in V_T$, and that S' and $\$$ do not appear in any other places in the grammar.

An LR(0) automaton for an RRP grammar G is a 6-tuple ($\mathbf{Q}$, V, P, $\mathbf{q}_0$, Next, Reduce), where V, P are as in G, and $\mathbf{Q}$ is the set of states of the LR(0) automaton (boldface letters are used to distinguish the LR automaton states from the right part states), $\mathbf{q}_0\in\mathbf{Q}$ is the start state, Next: $\mathbf{Q}\times V\to\mathbf{Q}$ is the transition function and Reduce: $\mathbf{Q}\to 2^P$ is the reduce function. In the following, an LR(0) item is defined to be a right part state $q\in Q$, and an LR(0) state is defined to be a set of items.

We will use the following standard conventions:

$a, b, \cdots\in V_T$ $\quad$ $S, A, B, \cdots\in V_N$

$\cdots, x, y, z\in V_T^*$ $\quad$ $\cdots, X, Y, Z\in V$

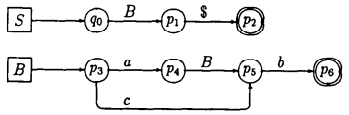


Fig. 1 Representation of the regular right part grammar G1.

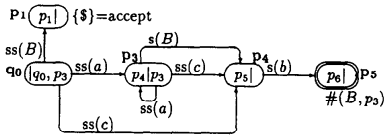


Fig. 2 LR automaton for grammar G1.

$t\in\bigcup_{i=1}^{k}V_T^i$ $\quad$ $\alpha, \beta, \cdots\in V^*$

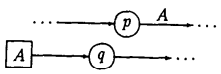
$\mathbf{p}, \mathbf{q}, \mathbf{r}, \cdots\in\mathbf{Q}$ $\quad$ $p, q, r, \cdots\in Q$

ϕ=empty set $\quad$ $\varepsilon\in V^0$

Define a relation $\downarrow$ on items by

$p\downarrow q$ iff $\exists A$: $\partial(p, A)$ is defined and $(A, q)\in P$.

In other words, "$p\downarrow$q" means that there exist two production in P as follows:



The above notation is the same as that used in Fig. 1.

The closure set of an item set is defined by the reflexive transitive closure $\downarrow^*$ as follows. For an item set R

$$\text{closure}(R)=\{q\mid p\in R: p\downarrow^* q\}.$$

Thus an LR(0) automaton is given by putting

$$\mathbf{q}_0=\text{closure}(\{q_0\})$$
$$\text{Next}(\mathbf{q}, X)=\text{closure}(\text{succ}(\mathbf{q}, X))$$
$$\text{succ}(\mathbf{q}, X)=\{r\mid q\in\mathbf{q}: r=\partial(q, X)\}$$
$$\text{Reduce}(\mathbf{q})=\{\text{reduce}(q)\mid q\in\mathbf{q}\cap F\}$$
$$\text{reduce}(q)=(A, p) \text{ where } (A, p)\in P, \exists\alpha\in V^*: q=\partial'(p, \alpha).$$

Next: $\mathbf{Q}\times V\to\mathbf{Q}$ can be extended to the mapping Next': $\mathbf{Q}\times V^*\to\mathbf{Q}$ as follows:

$$\text{Nexr}'(\mathbf{q}, \varepsilon)=\mathbf{q}$$
$$\text{Next}'(\mathbf{q}, \beta X)=\text{Next}(\text{Next}'(\mathbf{q}, \beta), X)$$
$$\text{where } X\in V, \beta\in V^*$$

Each LR state has two parts: the kernel and the nonkernel. These are defined as follows:

$$\text{kernel}(\mathbf{q}_0)=\phi$$
$$\text{nonkernel}(\mathbf{q}_0)=\mathbf{q}_0$$
$$\text{kernel}(\text{Next}(\mathbf{q}, X))=\text{succ}(\mathbf{q}, X)$$
$$\text{nonkernel}(\mathbf{q})=\{q\mid p\in\text{kernel}(\mathbf{q}), p\downarrow^+ q\}$$

Note that $\text{kernel}(\mathbf{q})\cap\text{nonkernel}(\mathbf{q})\neq\phi$ is possible.

Example. A grammar G1 is defined with the set of

productions $\{S \to B\$,\ B \to (aB|c)b\}$. The right part automaton of grammar G1 and its LR automaton are shown in Fig. 1 and Fig. 2, respectively. We use "|" to separate the kernel and the nonkernel of an LR state. In Fig. 2, the annotations "ss(a)" and "s(b)" denote "stack and shift a" and "shift b," respectively, which will be defined later in this section. "#(B, p_3)" means reduction by the production (B, p_3). "{$}=accept" in Fig. 2 means the transition by $ from $\mathbf{p}_1$ to the special state "accept".

The behavior and properties of an LR(0) automaton can be understood in terms of *transitions*. If $\mathbf{q}=\text{Next}(\mathbf{p}, X)$, the transition from $\mathbf{p}$ to $\mathbf{q}$ by X is represented by $\mathbf{p} \to^X \mathbf{q}$. We often write: $\mathbf{p} \to \cdots \to \mathbf{q}$ if $\exists \alpha \in V^*$: Next'($\mathbf{p}$, α)$=\mathbf{q}$.

In order to deal with multiple possible cases involved in a transition, we introduce a refinement of the Next relation called Goto. It represents a transition from a pair (a state, an item) to another pair (a state, an item).

Definition 2.1 Goto: $\mathbf{Q} \times Q \times V \to \mathbf{Q} \times Q$ is defined as follows:

$$\text{Goto}(\mathbf{p}, p, X)=(\mathbf{q}, q), \text{ if } \mathbf{q}=\text{Next}(\mathbf{p}, X),\ p \in \mathbf{p},$$
$$\text{and}\quad q=\partial(p, X). \quad \square$$

The transition $\text{Goto}(\mathbf{p}, p, X)=(\mathbf{q}, q)$ is represented by $(\mathbf{p}, p) \to^X (\mathbf{q}, q)$. In order to show whether the transition is from a kernel item or a nonkernel one, we write $(\mathbf{p}, p) \to^{X/K} (\mathbf{q}, q)$ if $p \in \text{kernel}(\mathbf{p})$, and $(\mathbf{p}, p) \to^{X/N} (\mathbf{q}, q)$ if $p \in \text{nonkernel}(\mathbf{p})$. We often write:

$(\mathbf{p}, p) \to^K (\mathbf{q}, q)$ if $\exists X \in V$: $(\mathbf{p}, p) \to^{X/K} (\mathbf{q}, q)$;

$(\mathbf{p}, p) \to^N (\mathbf{q}, q)$ if $\exists X \in V$: $(\mathbf{p}, p) \to^{X/N} (\mathbf{q}, q)$;

$(\mathbf{p}, p) \to (\mathbf{q}, q)$ if $\exists X \in V$: $(\mathbf{p}, p) \to^X (\mathbf{q}, q)$.

Here, K means kernel and N means nonkernel.

Definition 2.2 We write $\mathbf{p} \to^{X/K} \mathbf{q}$ if all the transitions from $\mathbf{p}$ to $\mathbf{q}$ by X are from the kernel items. $\square$

Definition 2.3 We write $\mathbf{p} \to^{X/N} \mathbf{q}$ if $\exists p \in$ nonkernel($\mathbf{p}$): $\text{Goto}(\mathbf{p}, p, X)=(\mathbf{q}, q)$ $\square$

Example. In Fig. 2, we have $(\mathbf{q}_0, p_3) \to^{a/N} (\mathbf{p}_3, p_4) \to^{B/K} (\mathbf{p}_4, p_5) \to^{b/K} (\mathbf{p}_5, p_6)$.

In general, a transition from a nonkernel item is caused by reading the first vocabulary symbol of the right part of a production, and a transition from a kernel item is caused by reading the rest. Simple and efficient algorithms for identifying the left end of a handle at reduction time were given in Purdom [9] and Nakata [7], but the grammar class for which this method is applicable is smaller than that of our method.

Basic ELR Parser: Parsing actions of a basic ELR automaton are described in terms of a relation $\vdash$ (moves to) defined on configurations. A configuration is a member of $(\mathbf{Q} \cup V)^* \times \mathbf{Q} \times (VV_T^*)$, consisting of the parser stack, the current state and the input string: $(\mathbf{p}_0\alpha_0 \ldots \mathbf{p}_{n-1}\alpha_{n-1},\ \mathbf{p}_n,\ Xz)$. When a reduction to a nonterminal A occurs, it can be considered that, after popping up the handle, A is placed temporarily in front of the input string and then shifted onto the stack.

Table 1 Example of parsing for G1.

$S \Rightarrow B\$ \Rightarrow aBb\$ \Rightarrow acbb\$$

parser stack	current state	input string	action
	q_0	$acbb\$$	stack shift a
q_0a	p_3	$cbb\$$	stack shift c
q_0ap_3c	p_4	$bb\$$	shift b
q_0ap_3cb	p_5	$b\$$	reduce by $\#(B, p_3)$ (reduce cb to B)
q_0a	p_3	$B\$$	shift B
q_0aB	p_4	$b\$$	shift b
q_0aBb	p_5	$\$$	reduce by $\#(B, p_3)$ (reduce aBb to B)
	q_0	$B\$$	stack shift B
q_0B	p_1	$\$$	accept

There are four different kinds of $\vdash$ moves:

$$(\mathbf{p}_0\alpha_0 \ldots \mathbf{p}_{n-1}\alpha_{n-1},\ \mathbf{p}_n,\ Xz) \vdash_{shift} (\mathbf{p}_0\alpha_0 \ldots \mathbf{p}_{n-1}\alpha_{n-1}X,\ \mathbf{p}_{n+1},\ z) \quad \text{if } \mathbf{p}_n \to^{X/K} \mathbf{p}_{n+1}$$

$$(\mathbf{p}_0\alpha_0 \ldots \mathbf{p}_{n-1}\alpha_{n-1},\ \mathbf{p}_n,\ Xz) \vdash_{stack\ shift} (\mathbf{p}_0\alpha_0 \ldots \mathbf{p}_{n-1}\alpha_{n-1}\mathbf{p}_nX,\ \mathbf{p}_{n+1},\ z) \quad \text{if } \mathbf{p}_n \to^{X/N} \mathbf{p}_{n+1}$$

$$(\mathbf{p}_0\alpha_0 \ldots \mathbf{p}_{n-1}\alpha_{n-1},\ \mathbf{P}_n,\ z) \vdash_{reduce} (\mathbf{p}_0\alpha_0 \ldots \mathbf{p}_0\alpha_0 \ldots \mathbf{p}_{n-2}\alpha_{n-2},\ \mathbf{p}_{n-1},\ Az)$$
$$\text{if } \exists (A, p)=\text{reduce}(q) \in \text{Reduce}(\mathbf{p}_n) \text{ and } q \in \text{kernel}(\mathbf{p}_n)$$

$$(\mathbf{p}_0\alpha_0 \ldots \mathbf{p}_{n-1}\alpha_{n-1},\ \mathbf{p}_n,\ z) \vdash_{\varepsilon\text{-reduce}} (\mathbf{p}_0\alpha_9 \ldots \mathbf{p}_{n-1}\alpha_{n-1},\ \mathbf{p}_n,\ Az)$$
$$\text{if } \exists (A, p)=\text{reduce}(p) \in \text{Reduce}(\mathbf{p}_n) \text{ and } p \in \text{nonkernel}(\mathbf{p}_n) \quad \square$$

The above parser is similar to the basic ELR parser in [7]. A parsing example is shown in Table 1.

A basic ELR parser constructed as above may be nondeterministic if there exist states $\mathbf{p}, \mathbf{q} \in \mathbf{Q}$, $p_1, p_2 \in \mathbf{p}$ and $X \in V$ such that both $(\mathbf{p}, p_1) \to^{X/N} (\mathbf{q}, q_1)$ and $(\mathbf{p}, p_2) \to^{X/K} (\mathbf{q}, q_2)$ exist. Such a case is called a stacking conflict.

3. Resolution of Stacking Conflicts by the Path Method

There remain two problems: the resolution of stacking conflicts of parsing actions and the resolution of parsing conflicts in inadequate states. A state is called inadequate if it has reduce/reduce conflicts or shift/reduce conflicts [1]. The latter problem can be solved by lookahead sets as in usual LR(k) parsers. Park, Choe, and Chang [8] have given an efficient algorithm for finding lookahead sets from an LR(0) automaton, and this can be applied to our LR(0) automaton. Therefore, we will not discuss this problem further in this paper.

Stacking conflicts can be resolved by conflict-removing grammar transformation [9]. In this paper a new method is proposed in which stacking conflicts are resolved indirectly at reduction time by using path numbers. We will explain and define the term path number later.

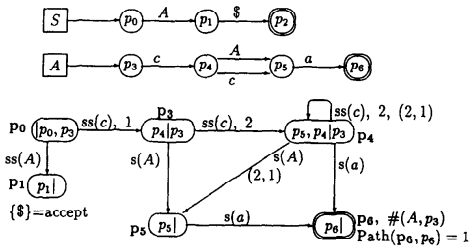


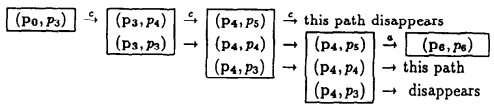
Fig. 3 Representation of G2 and the LR Automaton for G2.

Example. The right part automaton of the grammar G2 and its LR automaton is shown in Fig. 3. There exist both transitions $(\mathbf{p}_3, p_4)\rightarrow^{c/K}(\mathbf{p}_4, p_5)$ and $(\mathbf{p}_3, p_3)\rightarrow^{c/N}(\mathbf{p}_4, p_4)$, and therefore there is a stacking conflict. The numbers on transitions are path numbers, which will be explained in the following section.

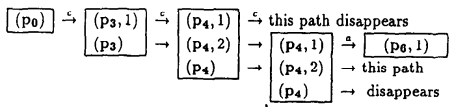
3.1 Outline of the Path Method

The first idea for resolving stacking conflicts is as follows. At shift time, if a stacking conflict occurs, select the stack shift action from the two possible actions, as in [7]. For example, we select ss(*c*) for the transition $\mathbf{p}_3\rightarrow^{c}\mathbf{p}_4$ in Fig. 3 from the two possible actions, s(*c*) for $(\mathbf{p}_3, p_4)\rightarrow^{c/K}(\mathbf{p}_4, p_5)$ and ss(*c*) for $(\mathbf{p}_3, p_3)\rightarrow^{c/N}(\mathbf{p}_4, p_4)$. By this ss(*c*) action, an irrelevant state symbol would be pushed onto the parser stack for transition $(\mathbf{p}_3, p_4)\rightarrow^{c/K}(\mathbf{p}_4, p_5)$. Therefore the remaining problem is to identify these irrelevant state symbols at reduction time.

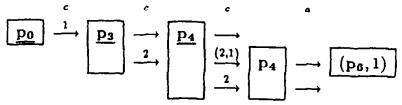
We use a parsing process for "$S\Rightarrow A\$\Rightarrow cAa\$\Rightarrow ccaa\$$" of G2 in Table 2 to explain the outline of our path method. The transitions from $\mathbf{p}_0$ to $\mathbf{p}_6$ are as "$\mathbf{p}_0\rightarrow^{c}\mathbf{p}_3\rightarrow^{c}\mathbf{p}_4\rightarrow^{a}\mathbf{p}_6$" and the configuration is $(\mathbf{p}_0c\mathbf{p}_3c\mathbf{p}_4ca, \mathbf{p}_6, a\$)$ when the transition reaches to $\mathbf{p}_6$. The above transitions in detail are as follows:



It is clear that the state which indicates the left end of the handle "*cca*" is $\mathbf{p}_3$ and the transition path for "*cca*" is the second of the above paths. $\mathbf{p}_4$ in the stack is the relevant state for the third path (which may indicate the left end of a handle), and is irrelevant for the second path. In order to identify the irrelevant state $\mathbf{p}_4$ for the current handle *cca*, the parser has to distinguish these transition paths. We use a number to show the path information. If p is the j^{th} kernel item in $\mathbf{p}$ we say that the *path number* of $(\mathbf{p}, p)$ is j. For example, p_5 and p_4 are the first and second kernel items in $\mathbf{p}_4$, respectively. We replace all kernel items with their path numbers and delete all nonkernel items for the above transitions as follows:



The second path, which begins as the second path, is changed from the second to the first path and ends as the first path. We rewrite the above transitions as follows. Here the underlined state symbols are pushed onto the stack. The "$\xrightarrow{2}$" means that a new handle begins as the second path. The "$\xrightarrow{(2,1)}$" means that the path number has been changed from 2 to 1. The "$(\mathbf{p}_6, 1)$" means that reduction takes place along the first path.



The result of the above transitions is shown in line 5 of Table 2, where the reduce action is triggered with the path number=1. In line 6, state $\mathbf{p}_4$ is popped because the attached number (=2) is not equal to 1. Then the parser changes the current path number from 1 to 2 in line 7. Finally in line 8, $\mathbf{p}_3$ indicates the left end of the handle since the attached path number (=2) matches the current path number.

3.2 Formalization of the Path Method

In this subsection, we summarize the above discussions and formally present the path method.

Table 2 An example of parsing for G2.

$S \Rightarrow A\$ \Rightarrow cAa\$ \Rightarrow cccaa\$$

	parser stack	current state or path number	input string	action
1		$\mathbf{p}_0$	*cccaa*$	stack shift $\mathbf{p}_0$, "1" and *c*
2	$\mathbf{p}_0c$ 1	$\mathbf{p}_3$	*ccaa*$	stack shift $\mathbf{p}_3$, "2" and *c*
3	$\mathbf{p}_0c\mathbf{p}_3c$ 1 2	$\mathbf{p}_4$	*caa*$	stack shift $\mathbf{p}_4$, (2,1), "2" and *c*
4	$\mathbf{p}_0c\mathbf{p}_3c\mathbf{p}_4$ *c* 1 2 2,(2,1)	$\mathbf{p}_4$	*aa*$	shift *a*
5	$\mathbf{p}_0c\mathbf{p}_3c\mathbf{p}_4$ *ca* 1 2 2,(2,1)	$\mathbf{p}_6$	*a*$	reduce by $(A, \mathbf{p}_3)$ current path=1
6	$\mathbf{p}_0c\mathbf{p}_3c\mathbf{p}_4$ *ca* 1 2 2,(2,1)	1	*Aa*$	pop $\mathbf{p}_4$ and "2", because the path number of $\mathbf{p}_4$ is 2 ($\neq 1$)
7	$\mathbf{p}_0c\mathbf{p}_3c$ *ca* 1 2 (2,1)	1	*Aa*$	change current path from 1 to 2
8	$\mathbf{p}_0c\mathbf{p}_3cca$ 1 2	2	*Aa*$	$\mathbf{p}_3$ indicates the left end, because the path number of $\mathbf{p}_3$ is 2. Pop *cca*.
9	$\mathbf{p}_0c$ 1	$\mathbf{p}_3$	*Aa*$	shift *A*
10	$\mathbf{p}_0cA$ 1	$\mathbf{p}_5$	*a*$	shift *a*
11	$\mathbf{p}_0cAa$ 1	$\mathbf{p}_6$	$	reduce by $(A, \mathbf{p}_3)$ current path=1
12	$\mathbf{p}_0cAa$ 1	1	*A*$	$\mathbf{p}_0$ indicates the left end, because the path number of $\mathbf{p}_0$ is 1. Pop *cAa*.
13		$\mathbf{p}_0$	*A*$	shift *A*
14	$\mathbf{p}_0$ *A*	$\mathbf{p}_1$	$	accept

The path transition functions are defined as follows:

Definition 3.1 Path: $\mathbf{Q}\times Q\rightarrow$ Integer.

Path($\mathbf{p}$, p)$=j$, if the item p is j^{th} kernel item in $\mathbf{p}$. □

Definition 3.2 PathBegin: $\mathbf{Q}\times V\rightarrow 2^{\text{Integer}}$.

PathBegin($\mathbf{p}$, X)$=\{$Path($\mathbf{q}$, q)$\mid \exists p\in \mathbf{p}$: ($\mathbf{p}$, p) $\rightarrow^{X/N}(\mathbf{q}, q)\}$ □

Definition 3.3 Path Change: $\mathbf{Q}\times V\rightarrow 2^{\text{Integer}\times\text{Integer}}$.

PathChange($\mathbf{p}$, X)$=\{$(Path($\mathbf{p}$, p), Path($\mathbf{q}$, q))$\mid \exists p\in \mathbf{p}$: ($\mathbf{p}$, p)$\rightarrow^{X/K}(\mathbf{q}, q)$, and Path($\mathbf{p}$, p)$\neq$Path($\mathbf{q}$, q)$\}$ □

Definition 3.4: PB$=\{$PathBegin($\mathbf{p}$, X)$\mid \mathbf{p}\in \mathbf{Q}$, $X\in V\}$

PC$=\{$PathChange($\mathbf{p}$, X)$\mid \mathbf{p}\in \mathbf{Q}$, $X\in V\}$

PT$=\{$Path($\mathbf{p}$, p)$\mid \mathbf{p}\in \mathbf{Q}$, $p\in$ kernel($\mathbf{p}$)$\}$ □

We will use the following conventions:

$$\mathrm{Pb}\in \mathrm{PB} \qquad \mathrm{Pc}\in \mathrm{PC}$$
$$\mathrm{H}\in ((\mathbf{Q}\times \mathrm{PB}\times \mathrm{PC})\cup \mathrm{PC}\cup V)^*$$

Definition 3.5: Right: PC$\rightarrow 2^{\text{Integer}}$.

Right(Pc)$=\{i\mid \exists j$: $(j, i)\in$ Pc$\}$ □

According to [1, 2, 7, 8], we can define the LALR(k) lookahead set la_k: $\mathbf{Q}\times Q\rightarrow \bigcup_{i=1}^{k} V_T^i$ as follows:

Definition 3.6:

$$\mathrm{la}_k(\mathbf{q}, q)=\bigcup_{\mathbf{p}\in \mathrm{LB}(\mathbf{q}, q),\,(A, p)=\mathrm{reduce}(q)} \mathit{Follow}_k(\mathbf{p}, A)$$

Here, LB($\mathbf{q}$, q)$=\{\mathbf{p}\mid p\in$ nonkernel($\mathbf{p}$), ($\mathbf{p}$, p)$\rightarrow\cdots\rightarrow$($\mathbf{q}$, q)$\}$ (see [7] p. 154), and $\mathit{Follow}_k(\mathbf{p}, A)=\{t\mid \exists \alpha\in V^*$, $z\in V_T^*$, $S\Rightarrow^*\alpha Atz$, Next'($\mathbf{q}_0$, α)$=\mathbf{p}\}$ (see [2] p. 38). □

Part, Choe, and Chang have given an efficient algorithm for computing the lookahead set in [8] (p. 163).

Definition 3.7: $\mathrm{LA}_k=\{\mathrm{la}_k(\mathbf{q}, q)\mid \mathbf{q}\in \mathbf{Q}$, $q\in \mathbf{q}\cap F\}$.

$$\mathrm{La}_k(\mathbf{q})=\bigcup_{q\in \mathbf{q}\cap F} \mathrm{la}_k(\mathbf{q}, q)$$ □

It is now possible to describe a PELALR(k) automaton (a Path directed parser for an ELALR(k) grammar), which is usually shown by a parsing table, and is interpreted by the PELALR driver routine.

A PELALR(k) automaton is a 10-tuple ($\mathbf{Q}$, V, P, $\mathbf{q}_0$, Goto, Reduce, PB, PC, PT, LA_k), where $\mathbf{Q}$, V, P, $\mathbf{q}_0$ and Reduce are as in the LR(0) automaton and Goto, PB, PC, PT, and LA_k are defined in Definition 2.1, 3.4, and 3.7, respectively.

We will propose in Section 5 an algorithm for generating the PELALR(k) automaton from an RRP grammar.

Parsing actions of a PELALR parser are described in terms of a relation $\vdash$ (moves to) defined on configurations. A configuration is a member of $((\mathbf{Q}\times \mathrm{PB}\times \mathrm{PC})\cup \mathrm{PC}\cup V)^*\times(\mathbf{Q}\cup \mathrm{PT})\times(VV_T^*)$, consisting of the parser stack, the current state or path number, and the input string. There are ten different kinds of $\vdash$ moves:

1. (H, $\mathbf{p}$, Xz)$\vdash_{\mathit{shift}}$(H Pc X, $\mathbf{q}$, z)
if $\mathbf{p}\rightarrow^{X/K}\mathbf{q}$, where Pc$=$PathChange($\mathbf{p}$, X). Here $\mathit{first}_k(Xz)\notin \mathrm{La}_k(\mathbf{p})$

2. (H, $\mathbf{p}$, Xz)$\vdash_{\mathit{stack\ shift}}$(H ($\mathbf{p}$, Pb, Pc) X, $\mathbf{q}$, z)
if $\mathbf{p}\rightarrow^{X/N}\mathbf{q}$, where, Pb$=$PathBegin($\mathbf{p}$, X) and Pc$=$PathChange($\mathbf{p}$, X). Here $\mathit{first}_k(Xz)\notin \mathrm{La}_k(\mathbf{p})$

3. (H, $\mathbf{q}$, z)$\vdash_{\#(A,p)}$(H, i, Az)
if there is $q\in$ kernel($\mathbf{q}$) such that $\mathit{first}_k(z)\in \mathrm{la}_k(\mathbf{q}, q)$, $i=$Path($\mathbf{q}$, q), where $(A, p)=$reduce(q).

4. (H, $\mathbf{q}$, z)$\vdash_{\#(A,p)}$(H, $\mathbf{q}$, Az)
if there is $p\in$ *nonkernel*($\mathbf{q}$) such that $\mathit{first}_k(z)\in \mathrm{la}_k(\mathbf{q}, p)$, where $(A, p)=$reduce(p).

5. (H Pc α, i, Az)$\vdash_{\mathit{PathChange}}$(H α, i, Az)
if $i\notin$ Right(Pc).

6. (H Pc α, i, Az)$\vdash_{\mathit{PathChange}}$(H α, j, Az)
if $i\in$ right(Pc), in other words $\exists(j, i)\in$ Pc.

7. (H ($\mathbf{p}$, Pb, Pc) α, i, Az)$\vdash_{\mathit{pop}}$(H Pc α, i, Az)
if $i\notin$ Pb.

8. (H ($\mathbf{p}$, Pb, Pc) α, i, Az)$\vdash_{\mathit{reduce}}$(H, $\mathbf{p}$, Az)
if $i\in$ Pb and $i\notin$ Right(Pc).

9. (H, $\mathbf{q}$, \$)$\vdash_{\mathit{acc}}$ accept
if $\exists q\in \mathbf{q}$ such that $\$\in \mathrm{la}_k(\mathbf{q}, q)$, where $(S, q_0)=$reduce(q).

10. otherwise error. □

A PELALR(k) machine constructed as above may be nondeterministic since (1) move 6 is nondeterministic if $\exists(j_1, i)$, $(j_2, i)\in$ Pc for $j_1\neq j_2$, and (2) move 8 is nondeterministic if $i\in$ Pb$\cap$Right(Pc). We will give a condition in Section 4 to guarantee that the above moves are deterministic.

The language recognized by the PELALR parser for a grammar G is

$$\mathrm{L(G)}=\{z\in V_T^+\mid (\varepsilon, \mathbf{q}_0, z)\vdash^+ \text{accept}\}.$$

In the following, the action of pushing Pc($\in$ PC) onto the stack is called the "path-change" action, and the action of pushing Pb($\in$ PB) is called the "path-begin" action.

4. The Grammar Class

Several methods (including our path method) that build a parser directly from an ELR(k) grammar have been proposed. However, the grammar classes for which these methods are applicable are smaller than the class of ELALR(k) grammars. We can define ELALR(k) grammars as follows, using the same method as that for ELR(k) and LALR(k) grammars found in Heilbrunner [4, 5]. An RRPG is called an ELALR(k) grammar if its ELALR(k) automaton is consistent, and its ELALR(k) automaton is obtained from the ELR(k) automaton by merging states with equal cores. In other words, the parsing conflicts in inadequate LR(0) states can be resolved by using LALR(k) lookahead symbols if the RRP grammar is an ELALR(k) grammar. In this section, we show that the grammar class of our method is larger than those of [2], [7] and ESLR(k), as indicated in Fig. 4.

A PELALR parser (a Path-directed parser for an ELALR(k) grammar) can be constructed from any RRP grammars if the following conditions are satisfied:

Conditions of a Path-directed parser (CP for short):

(i) It is an ELALR(k) grammar, and

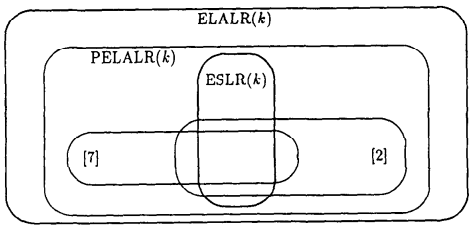


Fig. 4 The relation of several grammar classes.

(ii) there does not exist a transition $\mathbf{p}\rightarrow^{X}\mathbf{q}$ such that there exist $p_1, p_2\in\mathbf{p}$, $q\in\mathbf{q}$ and Goto($\mathbf{p}$, p_1, X)=Goto($\mathbf{p}$, p_2, X)=($\mathbf{q}$, q), where (*a*) $p_1\neq p_2$ or (*b*) $p_1=p_2$, $p_1\in$ kernel($\mathbf{p}$) and $p_2\in$ nonkernel($\mathbf{p}$). □

Lemma 4.1: For all (j, i)$\in$ PathChange($\mathbf{p}$, X), there does not exist (j', i)$\in$ PathChange($\mathbf{p}$, X) for $j'\neq j$, if CP(ii) holds. Therefore, the move $\vdash_{PathChange}$ for case 6 can be uniquely defined.

Proof: From Definition 3.3 and CP(ii). □

Lemma 4.2: For all $\mathbf{p}\in\mathbf{Q}$, $X\in V$: PathBegin($\mathbf{p}$, X)$\cap$ Right(PathChange($\mathbf{p}$, X))=ϕ, if CP(ii) holds. Therefore, the move $\vdash_{reduce}$ for case 8 can be uniquely defined.

Proof: From Definitions 3.2, 3.3, 3.5 and CP(ii). □

Theorem 4.1: The PELALR parser for an RRPG grammar G parses all sentences of the grammar correctly if, and only if, the condition CP holds for G.

Proof: First, let us assume that the condition CP holds.

Let $(\mathbf{p}_0, p_0)\rightarrow^{X_1/N}(\mathbf{p}_1, p_1)\rightarrow^{X_2/K}\cdots\rightarrow^{X_n/K}(\mathbf{p}_n, p_n)$, where $p_n\in F$ and reduce(p_n)=(A, p_0). If $n=0$, then the final state $p_n\in$ nonkernel($\mathbf{p}_0$). Therefore there would not be any problem in determining the left end of the handle by the move $\vdash_{\#(A,\,p)}$ of case 4. Thus assuming $n>0$, which means that $p_n\in$ kernel($\mathbf{p}_n$), we will prove in the following that the left end $\mathbf{p}_0$ can be uniquely decided by the PELALR parser.

1. Since the given RRP grammar G is an ELALR(k) grammar, the parsing conflicts in inadequate states can be resolved by the lookahead set $la_k(\mathbf{p}_n, p_n)$. In other words, when $\mathbf{p}_n$ appears as the current state, the move $\vdash_{\#(A,\,p_0)}$ can be accurately defined, therefore the resultant configuration is (H, Path($\mathbf{p}_n$, p_n), Az).

2. The current path will always keep the value to Path($\mathbf{p}_i$, p_i) ($0<i\leq n$) until ($\mathbf{p}_0$, PathBegin($\mathbf{p}_0$, X_1), PathChange($\mathbf{p}_0$, X_1)) appears at the top of the stack. We will prove this using an inductive method as follows:

Basis: When $i=n$, the configuration and the current path are (H, Path($\mathbf{p}_n$, p_n), Az) and Path($\mathbf{p}_n$, p_n), respectively.

Inductive step: When $j=i$, $0<i\leq n$, we can assume that the configuration and the current path could be replaced by (H′, Path($\mathbf{p}_j$, p_j), Az) and Path($\mathbf{p}_j$, p_j), respectively. Therefore, we will check four cases for moves 5, 6, 7, and 8.

(a) Let H′=H″ PathChange($\mathbf{p}_{j-1}$, X_j) α, and Path($\mathbf{p}_j$, p_j)$\notin$ Right(PathChange($\mathbf{p}_{j-1}$, X_j)).
Referring to Definition 3.3, Path($\mathbf{p}_j$, p_j) =Path($\mathbf{p}_{j-1}$, p_{j-1}), with move 5, (H′, Path($\mathbf{p}_j$, p_j), Az) $\vdash_{PathChange}$ (H″α, Path($\mathbf{P}_{j-1}$, p_{j-1}), Az).

(b) Let H′=H″ PathChange($\mathbf{p}_{j-1}$, X_j)α, and Path($\mathbf{p}_j$, p_j)$\in$ Right(PathChange($\mathbf{p}_{j-1}$, X_j)).
According to Definition 3.3, to our assumption, and to Lemma 4.1, (Path($\mathbf{p}_{j-1}$, p_{j-1}), Path($\mathbf{p}_j$, p_j))$\in$ PathChange($\mathbf{p}_{j-1}$, X_j), hence, for any Path($\mathbf{q}$, q)$\neq$Path($\mathbf{p}_{j-1}$, p_{j-1}), (Path($\mathbf{q}$, q), Path($\mathbf{p}_j$, p_j))$\in$ PathChange($\mathbf{p}_{j-1}$, X_j) does not exist.
Following move 6, (H′, Path($\mathbf{p}_j$, p_j), Az) $\vdash_{PathChange}$ (H″α, Path($\mathbf{p}_{j-1}$, p_{j-1}), Az).

(c) Let H′=H″ ($\mathbf{p}_{j-1}$, PathBegin($\mathbf{p}_{j-1}$, X_j), PathChange($\mathbf{p}_{j-1}$, X_j))α, and Path($\mathbf{p}_j$, p_j) $\notin$ PathBegin($\mathbf{p}_{j-1}$, X_j).
Along with move 7, (H′, Path($\mathbf{p}_j$, p_j), Az) $\vdash_{pop}$(H″ PathChange ($\mathbf{p}_{j-1}$, X_j)α, Path ($\mathbf{p}_j$, p_j), Az). Hence the next step will be (a) or (b).

(d) Let H′=H″ ($\mathbf{p}_{j-1}$, PathBegin($\mathbf{p}_{j-1}$, X_j), PathChange($\mathbf{p}_{j-1}$, X_j))α, and Path($\mathbf{p}_j$, p_j) $\in$ PathBegin($\mathbf{p}_{j-1}$, X_j).
According to Lemma 4.2, Pathbegin($\mathbf{p}_{j-1}$, X_j)$\cap$Right (PathChange ($\mathbf{p}_{j-1}$, X_j))=ϕ.
If $j>1$, and in accordance with our assumption, $(\mathbf{p}_{j-1}, p_{j-1})\rightarrow^{X_j/K}(\mathbf{p}_j, p_j)$. Since Path($\mathbf{p}_j$, p_j) $\in$ PathBegin($\mathbf{p}_{j-1}$, X_j), $(\mathbf{p}_{j-1}, q)\rightarrow^{X_j/N}(\mathbf{p}_j, p_j)$ exists. This is contrary to Lemma 4.2.
If $j=1$, ($\mathbf{p}_0$, PathBegin($\mathbf{p}_0$, X_1), PathChange($\mathbf{p}_0$, X_1)) will appear at the top of the stack. Thus our recursive proof ends correctly.

3. When ($\mathbf{p}_0$, PathBegin($\mathbf{p}_0$, X_1), PathChange($\mathbf{p}_0$, X_1)) appears at the top of the stack, the current Path number will be set to path($\mathbf{p}_1$, p_1) ($\in$ PathBegin($\mathbf{p}_0$, X_1)), which means that $\mathbf{p}_0$ indicates the left end of the handle and the move $\vdash_{reduce}$ is taken.

Second, let us assume that the condition CP does not hold.

The parsing conflicts in inadequate states cannot be resolved by the lookahead sets if CP(i) does not hold, and the move $\vdash_{PathChange}$ or $\vdash_{reduce}$ cannot be uniquely defined if CP(ii) does not hold. □

An RRP grammar is called a PELALR(k) grammar if and only if the condition CP holds. Let us compare our condition with those of [2] and [7].

[2]'s conditions are as follows:

1. It is an ELALR(k) grammar.

2. There does not exist a transition $\mathbf{p}\rightarrow^{X}\mathbf{q}$ such that there exist $p_1, p_2\in\mathbf{p}$, $q\in\mathbf{q}$ and ($\mathbf{p}$, p_1)T($\mathbf{q}$, q), ($\mathbf{p}$, p_2) T($\mathbf{q}$, q) where (a) $p_1\neq p_2$ or (b) $p_1=p_2$, $p_1\in$ kernel($\mathbf{p}$) and $p_2\in$ nonkernel($\mathbf{p}$). Here ($\mathbf{p}$, p)T($\mathbf{q}$, q) means $\exists X\in V$: Goto($\mathbf{p}$, p, X)=($\mathbf{q}$, q). □

The transition relation T is similar to our Goto relation. However, T ignores the vocabulary information.

[7]'s conditions are the same as CP. It is, however, pointed out by Grass [3] that the method fails to build correct parsers for some kinds of PELALR(k) grammars. Therefore, [7]'s conditions have to be modified as

follows:

Corrected conditions for the method of [7]:

1. The condition CP.
2. Stacking conflicts can be resolved by using lookback states $LB(\mathbf{q}, q)=\{\mathbf{p} \mid \mathbf{p}\rightarrow^{N}\rightarrow\cdots\rightarrow\mathbf{q}\}$ at **q** if there is no **p** and **r** such that both **p** and **r** belong to $LB(\mathbf{q}, q)$ and $\mathbf{p}\rightarrow^{N}\cdots\rightarrow\mathbf{r}\rightarrow^{K}\cdots\rightarrow\mathbf{q}$. □

Our condition is actually less restrictive than the above conditions. For example, G1 cannot be parsed by [2]'s method, and G2 cannot be parsed by [7]'s method.

According to Fig. 2 and [2], $(\mathbf{p}_3, p_3)T(\mathbf{p}_4, p_5)$, $(\mathbf{p}_3, p_4)T(\mathbf{p}_4, p_5)$, and $p_3\neq p_4$. This is contrary to the second of [2]'s conditions.

According to Fig. 3 and [7], $LB(\mathbf{p}_6, p_6)=\{\mathbf{p}_0, \mathbf{p}_3, \mathbf{p}_6\}$ because

$$(\mathbf{p}_0, p_3)\rightarrow^{c/N}(\mathbf{p}_3, p_4)\rightarrow^{c/K}(\mathbf{p}_4, p_5)\rightarrow^{a/K}(\mathbf{p}_6, p_6)$$
$$(\mathbf{p}_3, p_3)\rightarrow^{c/N}(\mathbf{p}_4, p_4)\rightarrow^{c/K}(\mathbf{p}_4, p_5)\rightarrow^{a/K}(\mathbf{p}_6, p_6)$$
$$(\mathbf{p}_4, p_3)\rightarrow^{c/N}(\mathbf{p}_4, p_4)\rightarrow^{c/K}(\mathbf{p}_4, p_5)\rightarrow^{a/K}(\mathbf{p}_6, p_6)$$

In other words, $\mathbf{p}_0\rightarrow^{N}\mathbf{p}_3\rightarrow^{K}\mathbf{p}_4\rightarrow^{K}\mathbf{p}_6$, and $\mathbf{p}_0$, $\mathbf{p}_3$, $\mathbf{p}_4\in LB(\mathbf{p}_6, p_6)$. This is contrary to the second of [7]'s corrected conditions.

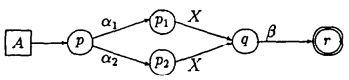
Next, we will show that the grammar class of the PELALR(k) grammars is larger than that of ESLR(k). We can define ESLR(k) grammars as follows, using the same method as that for ELR(k) and SLR(k) grammars found in Heilbrunner [4, 5]. An RRPG is called an ESLR(k) grammar if $S\Rightarrow^{+}S$ is impossible and if $S\Rightarrow^{*}\alpha Az\Rightarrow\alpha\beta z$, $S\Rightarrow^{*}\gamma Bx\Rightarrow\alpha\beta y$, and $follow_k(A)\cap follow_k(B)\neq\phi$ implies $A=B$, $\alpha=\gamma$, and $x=y$.

Theorem 4.2: An RRP grammar G is not an ESLR(k) grammar if the condition CP does not hold.

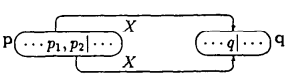
Proof: If G is not an ELALR(k) grammar, it clearly is not an ESLR(k) grammar. Therefore we assume that the grammar G is an ELALR(k) grammar but CP(ii) does not hold. In other words, $\exists\mathbf{p}$, $\mathbf{q}\in\mathbf{Q}$, $X\in V$, p_1, $p_2\in\mathbf{p}$, $q\in\mathbf{q}$ satisfying $(\mathbf{p}, p_1)\rightarrow^{X}(\mathbf{q}, q)$ and $(\mathbf{p}, p_2)\rightarrow^{X}(\mathbf{q}, q)$. We will prove that G is not an ESLR(k) grammar by checking three cases for p_1 and p_2 as follows:

1. p_1, $p_2\in$ kernel(**p**)

In this case, $p_1\neq p_2$. Since $q=\partial(p_1, X)=\partial(p_2, X)$, according to assumption 1 of Section 2, $\exists(A, p)\in P$: p_1, p_2, $q\in Q_p$. We assume $\exists r\in Q_p\cap F$ and $\exists\beta\in V^{*}$: $r=\partial'(q, \beta)$. There is a production in Q of the form



satisfying $\alpha_1\neq\alpha_2$, and there is a transition in **Q** as follows:



Since p_1 and p_2 are in the same LR state **p**, the following derivations exist:

$S\Rightarrow^{*}\alpha Aw\Rightarrow\alpha\alpha_1X\beta w$

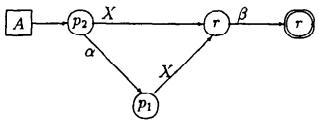
$S\Rightarrow^{*}\gamma Av\Rightarrow\gamma\alpha_2X\beta v=\alpha\alpha_1X\beta v$

and $follow_k(A)\cap follow_k(A)\neq\phi$

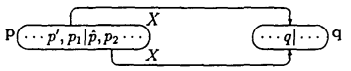
If G were an ESLR(k) grammar, then it would be true that $\alpha=\gamma$, and hence that $\alpha_1X\beta=\alpha_2X\beta$, which contradicts $\alpha_1\neq\alpha_2$.

2. $p_1\in$ kernel(**p**), $p_2\in$ nonkernel(**p**)

In this case, p_1 may be p_2. Since $p_2\in$ nonkernel(**p**), p_2 is an initial state in Q for some production (A, p_2) and this production is of the form



satisfying $\alpha\neq\varepsilon$, and there is a transition in **Q** as



satisfying $p'\downarrow^{+}p_2$ (p' may be p_1). According to the definition of $\downarrow$, $\exists(B, \hat{p})\in P$: $\partial(p', B)\in Q$, and $\hat{p}\downarrow^{*}p_2$ ($(B, \hat{p})$ may be (A, p_2)). Since p', p_1, p_2, and $\hat{p}$ are in the same LR state **p**, the following derivations exist:

$S\Rightarrow^{*}\gamma Aw\Rightarrow\gamma\alpha X\beta w$

$S\Rightarrow^{*}\delta Bv\Rightarrow^{*}\delta Av_1v\Rightarrow\delta X\beta v_1v=\gamma\alpha X\beta v_1v$

and $follow_k(A)\cap follow_k(A)\neq\phi$

If G were an ESLR(k) grammar, then it would be true that $\gamma=\delta$ and hence that $\alpha X\beta=X\beta$, which contradicts $\alpha\neq\varepsilon$.

3. p_1, $p_2\in$ nonkernel(**p**)

In this case, $p_1\neq p_2$. According to assumption 1 of Section 2, this case is impossible. □

Furthermore, the grammar G3 shown in Fig. 5 is a PELALR(1) grammar but is not an ESLR(k) grammar for any $k>0$; the grammar G4 shown in Fig. 6 is an ELALR(1) grammar, but is not a PELALR(k) grammar for any $k>0$.

It is clear that PB can be recorded compactly and efficiently, whereas PC cannot. In Section 5, we will propose an optimization algorithm to remove some of these "path-change" actions.

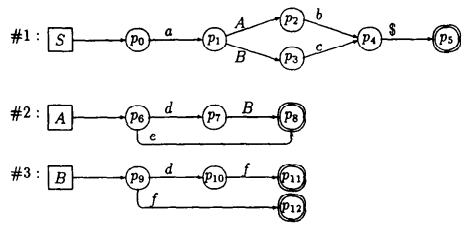


Fig. 5 Representation of the regular right part grammar G3.

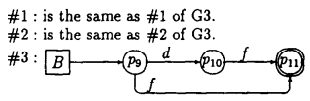


Fig. 6 Representation of the regular right part grammar G4.

5. Optimization and Parser Generation

In this section, we use the grammar G5 in Figs. 7 and 8 to explain our optimization algorithm.

In a PELALR parser, path actions must be executed even if there is no stacking conflict. It is therefore inefficient. If there is no stacking conflict, the basic ELR parser is the most efficient. We want to optimize our path method in which the path actions are executed only when stacking conflicts occur. We mark all state-item pairs in the transition paths that contain stacking conflicts, and we only add path actions for these transitions. For example, we do not add path actions to the transition $(\mathbf{p}_0, p_3)\rightarrow^{b}(\mathbf{p}_5, p_6)$ in Fig. 8.

When a stacking conflict occurs, the "path-begin" action records the start position of the handle for the corresponding transition from the nonkernel item. Therefore, it is not an eliminable action. A "path-change" action is necessary only when $\text{Path}(\mathbf{p}, p) \neq \text{Path}(\mathbf{q}, q)$ for a transition $(\mathbf{p}, p)\rightarrow^{X/K}(\mathbf{q}, q)$. We can eliminate this action if p becomes the $\text{Path}(\mathbf{q}, q)^{th}$ item of kernel(**p**) as a result of a rearrangement of the kernel items in kernel(**p**). For example, we set p_7, p_8, p_9 to the first kernel items in $\mathbf{p}_6$, $\mathbf{p}_7$, $\mathbf{p}_8$, $\mathbf{p}_{10}$, and set p_4, p_5 to the second kernel items in $\mathbf{p}_1$, $\mathbf{p}_3$, $\mathbf{p}_4$, $\mathbf{p}_7$, $\mathbf{p}_8$, respectively, as in Fig. 8.

We summarize the above discussion and show an optimization algorithm in Algorithm 5.1. The set Conflict_pair is used to mark state-item pairs in some transition paths containing stacking conflicts. $\text{Path}(\mathbf{p}, p):=i$ means that the kernel item p is set to the i^{th} kernel item in **p**. $\text{Path}(\mathbf{p}, p):=0$ means that the kernel item p can be set anywhere in **p**. We define the function **Path** as follows:

Definition 5.1: Path: $\mathbf{Q}\rightarrow 2^{\text{Integer}}$.

$\mathbf{Path}(\mathbf{p})=\{\text{Path}(\mathbf{p}, p) \mid p\in \text{kernel}(\mathbf{p}),\ \text{Path}(\mathbf{p}, p)\neq 0\}$ □

Algorithm 5.1. Optimization of PELALR parsers

Input: The LR(0) automaton for an RRP grammar G

Output: The optimized PELALR(0) automaton for the LR(0) automaton

Method:

1. Mark all state-item pairs in some transition paths containing stacking conflicts.

(a) Conflict_pair:=ϕ;

(b) For all stacking conflicts $(\mathbf{p}, p_1)\rightarrow^{X/N}(\mathbf{q}, q_1)$ and $(\mathbf{p}, p_2)\rightarrow^{X/K}(\mathbf{q}, q_2)$, add $(\mathbf{p}, p_2)$ and $(\mathbf{q}, q_2)$ to Conflict_pair.

(c) $\forall(\mathbf{p}, p)\in$ Conflict_pair, add all $(\mathbf{r}, r)$'s such that $(\mathbf{r}, r)\rightarrow^{K}(\mathbf{p}, p)$ or $(\mathbf{p}, p)\rightarrow^{K}(\mathbf{r}, r)$, to Conflict_pair.

2. Set different numbers for all marked final paris.

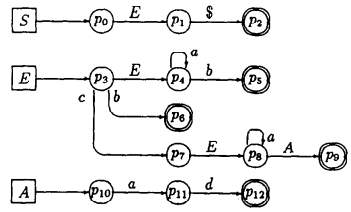


Fig. 7 Representation of the regular right part grammar G5.

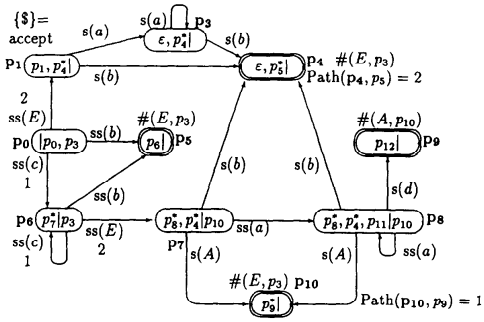


Fig. 8 PELALR(0) automaton for G5 based on Algorithm 5.1.

(a) path_number:=1;

(b) **For** all $(\mathbf{q}, q)\in$ Conflict_pair, **if** $q\in F$ **then** $\text{Path}(\mathbf{q}, q):=$path_number++; **else** $\text{Path}(\mathbf{q}, q):=0$;

3. Adjust the order of kernel items that have been marked.

for $\forall(\mathbf{p}, p)$, $(\mathbf{q}, q)\in$ Conflict_pair, $X\in V$, $(\mathbf{p}, p)\rightarrow^{X/K}(\mathbf{q}, q)$

if $\text{Path}(\mathbf{q}, q)\notin$ **Path(p) then**

if $\text{Path}(\mathbf{p}, p)=0$ **then** $\text{Path}(\mathbf{p}, p):=\text{Path}(\mathbf{q}, q)$

else if $\text{Path}(\mathbf{p}, p)=0$ **then** $\text{Path}(\mathbf{p}, p):=$path_number++;

4. Compute PB.

for $\forall\mathbf{p}\in\mathbf{Q}$, $\forall X\in V$, $\text{PathBegin}(\mathbf{p}, X)=\{\text{Path}(\mathbf{q}, q) \mid p\in\mathbf{p}: (\mathbf{p}, p)\rightarrow^{X/N}(\mathbf{q}, q), (\mathbf{q}, q)\in$ Conflict_pair$\}$

5. Compute PC.

for $\forall\mathbf{p}\in\mathbf{Q}$, $\forall X\in V$, $\text{PathChange}(\mathbf{p}, X)=\{(\text{Path}(\mathbf{p}, p), \text{Path}(\mathbf{q}, q)) \mid p\in\mathbf{p}$: $(\mathbf{p}, p)\rightarrow^{X/K}(\mathbf{q}, q)$, $\text{Path}(\mathbf{p}, p)\neq \text{Path}(\mathbf{q}, q)$, and $(\mathbf{q}, q)\in$ Conflict_pair$\}$ □

The optimized PELALR(0) automaton for the grammar G5 is shown in Fig. 8. The marked state-item pairs are indicated by asterisks. In Fig. 8, there are no "path-change" actions at all. However, the "path-change" actions of G2 cannot be removed by Algorithm 5.1.

In order to remove irrelevant path actions, we slightly modify the moves of PELALR parsers as follows:

1. Add "Pc is not pushed if Pc=ϕ" to the move$\vdash_{shift}$.

2. Add "Pb is not pushed if Pb=ϕ" to the move$\vdash_{stack\ shift}$.

3. Rewrite step 3 of the move$\vdash_{\#(A,p)}$ as follows:

(a) 3.1: $(\text{H}, \mathbf{q}, z)\vdash_{\#(A,p)}(\text{H}, i, Az)$

if there is $q\in$ kernel(**q**) such that $first_k(z)\in \text{la}_k(\mathbf{q}, q)$, $(\mathbf{q}, q)\in$ Conflict_pair and $i=\text{Path}(\mathbf{q}, q)$, where (A, p)

$=$reduce(q).

(b) 3.2: (H (**p**, Pb, Pc) α, **q**, z)$\vdash_{\#(A,p)}$(H, **p**, Az) if there is $q\in$ kernel(**q**) such that $first_k(z)\in la_k$(**q**, q) and (**q**, q)$\notin$ Conflict_pair, where (A, p)=reduce(q).

In order to simplify the representation of PELALR parsers, we leave the form (**p**, Pb, Pc) even if there is no Pb or Pc.

It is now possible to describe a PELALR(k) parser generating system based on the preceding algorithms and definitions.

Algorithm 5.2: PELALR(k) Parser Generation

Input: An RRP grammar G.

Output: The PELALR(k) automaton for G.

Method:

1. Build the LR(0) automaton.
2. Build the PELALR(0) automaton by Algorithm 5.1, check the condition CP(ii).
3. Compute the lookahead sets by using the method in Park [8], and check the condition CP(i).
4. Write out the PELALR(k) automaton. □

6. Conclusion

Nowadays, the advantages of regular right part grammars are well known. This paper shows a method of conceiving ELR parsers for regular right part grammars. In our method, the ELR parser is directly built from a given ELR grammar, and no grammar transformation is necessary. Therefore, it is not necessary to rewrite the semantic rules attached to each production. The grammar class of PELALR(k) grammars is larger than that of ESLR(k) grammars, that of [2]'s method, or that of [7]'s method, and is the same as that of [10]'s method.

Generally speaking, when a single production derives a long string, the path method is more efficient than [10] 's method. Two practical examples, G6 (Figs. 9 and 10) and G7 (Figs. 11 and 12), are given in order to compare the efficiency of the two methods. We will compare the number of push/pop operations while the two different parsers run through the long string "**if** x **then** x **elsif** x **then** x **else** x" and the short string "$i+i*i$". We assume that all the operations of the following list have the same unit cost.

- Pushing an LR state symbol;
- Pushing a counter;
- Pushing less than eight "path-begin" symbols;
- Pushing a "path-change" symbol;
- Reducing . . . by [10]'s method;
- Moves 5, 6, 7, and 8 of the path method;

Tables 3, 4, 5, and 6 give details of the process of the two parsing methods applied to the two above-mentioned strings. In order to parse the long string, [10]'s method has to push/pop the parser stack 32 times (Table 3), while the path method only has to push/pop the parser stack 9 times (Table 4). Even in the case of the short string, [10]'s method has to push/pop the parser stack 31 times (Table 5), while the path method only has to push/pop parser stack 22 times (Table 6). These results seem to show that the path method generates more efficient parsers than [10]'s method does. Unfortunately, we have not been able to prove this conjecture.

An ELR grammar is more concise than the corresponding LR grammars. However, the parser generation for the ELR grammar is more complex because a single production can potentially derive a string of infinite length [2, 4]. Unfortunately, it is impossible for ELR parsers to avoid some overhead actions. For example, grammar transformations [4, 6], readback states [2], or counter stacks [10] are necessary. The basic ELR parser with no overhead action is the most efficient in terms of both time and space, even more efficient than the ordinary LR parser [9]. According to Algorithm 5.1, the efficiency of the path method is the same as that

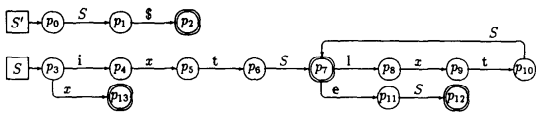


Fig. 9 Representation of the regular right part grammar G6.

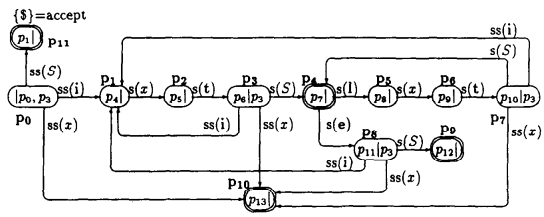


Fig. 10 PELALR(0) automaton for G6 based on Algorithm 5.1.

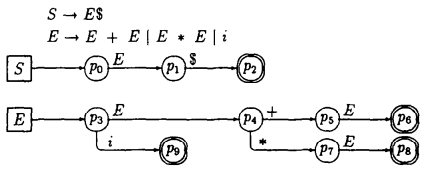


Fig. 11 representation of the regular right part grammar G7.

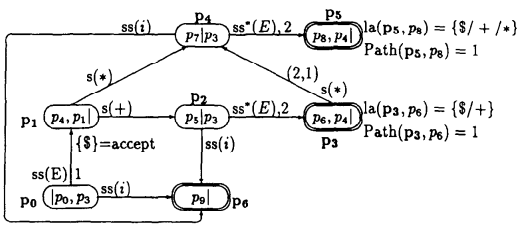


Fig. 12 PELALR(1) automaton for G7 based on Algorithm 5.1.

Table 3 A process for parsing G6 by [10]'s method.

if x then x elsif x then x else x $

	parser stack	current state	input string	number of shifts	number of reductions
1		p_0	$ixtxlxtxex$	2	
2	p_0i	p_1 1	$xtxlxtxex$	6	
3	$p_0ip_1xp_2tp_3x$ 1 2 3	p_{10} 1	$lxtxex$		1
4	$p_0ip_1xp_2t$ 1 2	p_3 3	$Slxtxex$	10	
5	$p_0ip_1xp_2tp_3Sp_4ip_5xp_6tp_7x$ 1 2 3 4 5 6 7	p_{10} 1	ex		1
6	$p_0ip_1xp_2tp_3Sp_4ip_5xp_6t$ 1 2 3 4 5 6	p_7 7	Sex	6	
7	$p_0ip_1xp_2tp_3Sp_4ip_5xp_6tp_7Sp_4ep_8x$ 1 2 3 4 5 6 7 8 9	p_{10} 1	$		1
8	$p_0ip_1xp_2tp_3Sp_4ip_5xp_6tp_7Sp_4e$ 1 2 3 4 5 6 7 8	p_8 9	S	2	
9	$p_0ip_1xp_2tp_3Sp_4ip_5xp_6tp_7Sp_4ep_8S$ 1 2 3 4 5 6 7 8 9	p_9 10	$		1
10		p_0	S	2	
11	p_0S	p_{11} 1	$	accept	
				28 +	4 = 32

Table 4 A process for parsing G6 by the path method.

if x then x elsif x then x else x

	parser stack	current state	input string	number of shifts	number of reductions
1		p_0	$ixtxlxtxex$	1	
2	p_0i	p_1	$xtxlxtxex$	1	
3	p_0ixtp_3x	p_{10}	$lxtxex$		1
4	p_0ixt	p_3	$Slxtxex$	1	
5	$p_0ixtSlxtp_7x$	p_{10}	ex		1
6	$p_0ixtSlxt$	p_7	Sex	1	
7	$p_0ixtSlxtSep_8x$	p_{10}	$		1
8	$p_0ixtSlxtSe$	p_8	S	0	
9	$p_0ixtSlxtSeS$	p_9	$		1
10		p_0	S	1	
11	p_0S	p_{11}	$	accept	
				5 +	4 = 9

of the basic ELR parser, if stacking conflicts do not occur. When stacking conflicts occur, some overhead actions are necessary. The latter actions of the path method are very few, because (1) the "path-begin" actions can be compactly and efficiently performed and (2) the "path-change" actions can be mostly removed by Algorithm 5.1. The reason is that we can use one number to represent information for several kernel items that in the same transition path.

Finally, the optimization algorithm of [7] is also applicable to our path method. As a result, further path actions can be removed. On the whole, we think our path method is useful because of its wide applicability and efficiency.

Table 5 A process for parsing G7 by [10]'s method.

$i + i * i $

	parser stack	current state	input string	number of shifts	number of reductions
1		p_0	$i + i * i$	2	
2	p_0i	p_6 1	$+i * i$		1
3		p_0	$E + i * i$	7	
4	$p_0Ep_1+p_2i$ 1,1 2	p_6 1	$*i$		1
5	p_0Ep_1+ 1,1	p_2 2	$E * i$	7	
6	$p_0Ep_1+p_2Ep_3*p_4i$ 1,1 2 3,1 2	p_6 1	$		1
7	$p_0Ep_1+p_2Ep_3*$ 1,1 2 3,1	p_4 2	E	3	
8	$p_0Ep_1+p_2Ep_3*p_4E$ 1,1 2 3,1 2	p_5 3,1	$		1
9	p_0Ep_1+ 1,1	p_2 2	E	3	
10	$p_0Ep_1+p_2E$ 1,1 2	p_3 3,1	$		1
11		p_0	E	3	
12	p_0E	p_1 1,1	$	accept	
				26 +	5 = 31

Table 6 A process for parsing G7 by the path method.

$i + i * i $

	parser stack	current state	input string	number of shifts	number of reductions
1		p_0	$i + i * i$	1	
2	p_0i	p_6	$+i * i$		1
3		p_0	$E + i * i$	3	
4	p_0E+p_2i 1	p_6	$*i$		1
5	p_0E+ 1	p_2	$E * i$	4	
6	$p_0E+p_2E*\ p_4i$ 1 2 (2,1)	p_6	$		1
7	p_0E+p_2E* 1 2 (2,1)	p_4	E	2	
8	$p_0E+p_2E*\ p_4E$ 1 2 (2,1)2	p_5	$		3
9	p_0E+ 1	p_2	E	2	
10	p_0E+p_2E 1 2	p_3	$		2
11		p_0	E	2	
12	p_0E 1	p_1	$	accept	
				14 +	8 = 22

References

1. AHO, A. V., SETHI, R. and ULLMAN, J. D. *Compilers Principles, Techniques, and Tools*, Addison Wesley, Reading, Massachusetts (1986).
2. CHAPMAN, N. P. LALR(1, 1) Parser Generation for Regular Right Part Grammars, *Acta Inf.*, **21** (1984), 29-45.
3. GRASS, J. E. personal communication (Sep. 1987).
4. HEILBRUNNER, S. On the Definition of ELR(k) and ELL(k) Grammars, *Acta Inf.*, **11** (1979), 169-176.
5. HEILBRUNNER, S. A Parsing Automata Approach to LR Theory, *Theoretical Computer Science*, **15** (1981), 117-157.
6. MADSEN, O. L. and KRISTENSEN, B. B. LR Parsing of Extended Context Free Grammars, *Acta Inf.*, **7** (1976), 61-73.
7. NAKATA, I. and SASSA, M. Generation of Efficient LALR Parsers for Regular Right Part Grammars, *Acta Inf.*, **23** (1986), 149-162.
8. PARK, J. C. H., CHOE, K. M. and CHANG, C. H. A New Analysis of LALR Formalisms, *ACM trans. Program. Lang. Syst.*, **7**, 1 (1985), 159-175.
9. PURDOM, P. W. and BROWN, C. A. Parsing Extended LR(k) Grammars, *Acta Inf.*, **15** (1981), 115-127.
10. SASSA, M. and NAKATA, I. A Simple Realization of LR Parsers for Regular Right Part Grammars, *Inf. Process. Lett.*, **24** (1987), 113-120.

(Received June 14, 1990; revised March 6, 1991)