JOURNAL OF COMPUTER AND SYSTEM SCIENCES 22, 243–259 (1981)

# Extending Lookahead for LR Parsers

THEODORE P. BAKER*

*Florida State University, Tallahassee, Florida 32306*

Received February 26, 1980; revised November 10, 1980

A practical method is presented for extending the lookahead of LR parsers, by the addition of "reduce-arcs." Applied to an LR(0) parser, this gives a machine which is close in size to the corresponding LALR(1) machine, but is capable of making use of unbounded lookahead. The class of grammars parsable by this method is a subset of the LR-regular grammars which is shown to be properly included in the LALR($k$) grammars, if limited to $k$ symbols of lookahead, but also includes non-LR grammars if no such limit is imposed. Application is foreseen to error recovery in LALR(1) parsers, as well as the handling of occasional non-LALR(1) situations in normal parsing.

## 1. INTRODUCTION

The LR parsing method, introduced by Knuth in [9], has been studied and refined to a point where it is accepted as feasible and practical for use in compilers. In particular, the SLR(1) [5] and LALR(1) [4, 10, 1] techniques of DeRemer have proven useful. However, both of these techniques take a step backward in generality from LR(1), in order to reduce the parser to a manageable size. For larger $k$ than 1 the size and complexity of computing LR($k$) parsing tables grows out of proportion to the additional benefits, so that the method cannot be considered practical.

There are, however, instances where limitation of lookahead to one symbol forces rewriting the grammar. For example, the published grammar for the programming language Ada [14] makes use of the keywood "**is**" in several contexts where one symbol of lookahead is not sufficient. The nonterminals "subprogram_body," "body_stub," and "generic_subprogram_instantiation" all are derivable from "declarative_part," and so are indistinguishable by context. All of the following three derivations can produce a string beginning "**procedure** identifier **is**...":

subprogram_body $\Rightarrow^*$ subprogram_specification **is** declarative_part $\cdots$
body_stub $\Rightarrow^*$ subprogram_specification **is separate**;
generic-subprogram_instantiation $\Rightarrow^*$ **procedure** identifier **is**
generic_instantiation;

* This work supported in part by National Science Foundation Grants MCS-7724098 and MCS-7924583.



0022-0000/81/020243-17$02.00/0

The decision to reduce "**procedure** identifier" to "subprogram_specification" versus to shift "**is**" cannot be made without looking past the "**is**." A similar problem arises distinguishing use of "**and**" from "**and then**."

Such problems, once discovered, can be solved by modifying the grammar, but not without cost. Modifying the grammar takes effort and is subject to error. It is complicated by considerations of readability, parser size, and limitations imposed by semantic actions which are keyed to specific reductions. Thus, an efficently implementable parsing method that can make use of extended lookahead information, reducing the need for grammar modifications, could be of practical convenience.

While the ability to utilize more than one symbol of lookahead in normal parsing is a convenience rather than a necessity, it can be of far greater value in error recovery. When an LR(1) parser discovers an error it has excellent information available about the context to the left of the error, but no effective means of utilizing the right context. Obtaining enough right context information, in a useable form, to overcome this "left prejudice" has been the central goal of much of the work done on LR error recovery.

In this paper, we describe how to transform an LR parser's state-transition system, by adding a fixed set of "reduce-arcs," so that "optimal" use can be made of the first lookahead symbol, and limited use can be made of an unbounded number of symbols following. Applied to an LR(0) machine, this gives a parser that is equivalent to the LALR(1) if use is made of only the first lookahead symbol, but which actually can make use of a regular set of lookahead strings[1] (in contrast to the finite set used by LR($k$)).

All LR parsing methods share an underlying structure. The parse is performed left-right, bottom-up, by successive reductions of the leftmost simple phrase, or "handle," to a nonterminal symbol. The handle is located by means of a finite state machine that scans the intermediate sentential form from left to right up to, and perhaps slightly beyond, the handle. The symbols to the right of the handle are called "lookahead" symbols, and are always terminal symbols, due to the left-right order of the parse.

In general, the machine is assumed to work with a stack, on which previous states may be stored. With this model, the operations of the machine are limited to two types:

shift: push the current state onto the stack, change state to new state, determined by the current input symbol and the current state, and advance to the next input symbol;

reduce: remove a string of states from the top of the stack, insert a nonterminal symbol at the head of the input, reset the state to the value at the top of the stack, and perform a shift operation on the new nonterminal.

[1] The notion of using a regular set of lookahead strings is not new. Earlier theoretical work by Čulik and Cohen on LR-regular grammars is discussed in Section 4.

Thus, except for reductions, the machine operates as a finite state sequential machine. By adding "reduce-arcs" corresponding to all the possible transitions effected by reductions, we can produce a finite state sequential machine that approximates the actions of the LR parser (as best it can without the benefit of a stack). We call this the *extended* LR (XLR) *machine.*

A reduce-arc can be viewed as a null-move of the finite state machine, and might be implemented that way if parser size were of first importance. Instead, we choose to implement a reduce arc by a set of transitions on terminal symbols, corresponding to all the shift transitions on terminal symbols that may follow a chain of reductions. Thus, being able to short-cut chains of possible reductions, the extended LR finite state machine can scan a grammatical string of terminals left-to-right, performing only shifts of terminal symbols.

Of course, the new machine is finite, and so can only recognize a regular superset of the language parsed by the original LR machine, but our intent is not to parse with it directly. Instead we propose to use it to resolve conflicting choices of parsing or error recovery actions. For this purpose, even a regular approximation to the set of lookahead strings can be a vast improvement over the sets of one-symbol lookaheads used by other practical methods.

The new machine is also nondeterministic. This is a drawback, but not as serious as it first seems. Although the nondeterminism can be eliminated, we feel the increased cost in table size may rule this out in practice. In cases where one symbol of lookahead is sufficient, the machine will be deterministic. In other cases, we might expect to pay a price, but expect that it will neither be large nor payed frequently.

In summary, the advantage of the XLR machine over the LALR(1) machine is that

(1) extended lookahead information can be utilized when needed.

The advantages over LR($k$) machines for $k$ greater than 1 are that:

(2) table size is only slightly larger than for the LALR(1) machine (1801 reduce-arcs vs 1663 reduce actions (unoptimized), for a Pascal grammar);

(3) lookahead exacts no overhead except where it is needed;

(4) the extended machine can be computed practically. (A test program required 135 sec and used 24,500 words (60 bit) of memory on a CDC CYBER 70 to process a Pascal grammar.)

In the next section, we formally describe the XLR machine and characterize its lookahead information. Section 3 explains how the XLR machine can be used in parsing, and Section 4 relates its power to that of other LR variants, including the LR-regular method of Čulik and Cohen. The problem of determining whether a grammar is parsable by this method, and of determining how many symbols of lookahead will be needed, is the subject of Section 5. Section 6 deals with the computation of reduce-arcs, and Section 7 outlines the direction of our future research.

## 2. The XLR Machine

The transformation of an LR machine to an XLR machine is most easily understood as the closure of the state graph under a graph rewriting scheme. The basic scheme and its generalization are illustrated in Figs. 1 and 2. The dotted arc indicates the reduce arc that should be added if the other nodes and arcs of the scheme can be matched with nodes and arcs of the original LR machine's state graph. (A complete example can be found in Fig. 3). However, to be able to prove properties of the XLR machine, we will need a definition in the formalism of parsing theory. The notation we use is similar to that in [2] and [8].

### 2.1. *Definitions*

Let $G$ be a fixed context free grammar, $G = (N, \Sigma, P, S)$, where $N$ is a finite set of *nonterminal symbols*, $\Sigma$ is a finite set of *terminal symbols*, $P$ is a finite set of *rules*, $P \subset N \times (N \cup \Sigma)^*$, and $S$ is the *start symbol* of $G$.

Let $M$ be a fixed *shift-reduce machine* for $G$, consisting of a tuple $M = (K, N \cup \Sigma, \delta, q_0, F, \rho)$, where $K$ is a finite set of *states*; $F$ is a set of *reduce* states, $F \subseteq K$; $q_0$ is the *start state*, $q_0 \in K$; $\Sigma$ is the *input alphabet*, the terminals of $G$; $\delta$ is the *shift transition* function $\delta: K \times \Sigma \to K$; $\rho$ is the *reduction function* $\rho: F \to \mathcal{P}(P)$, giving a set of applicable reductions for each reduce state.

In what follows, we shall use naming conventions that implicitly qualify the meaning of certain variables, except when specified otherwise.

$\alpha, \beta, \zeta, u, v, w \in (N \cup \Sigma)^*$, $\quad v \Rightarrow^* \Lambda$.

$t \in \Sigma$ or $t \in N \cup \Sigma$.

$\gamma \in \Sigma^*$.

$A, B, C \in N$.

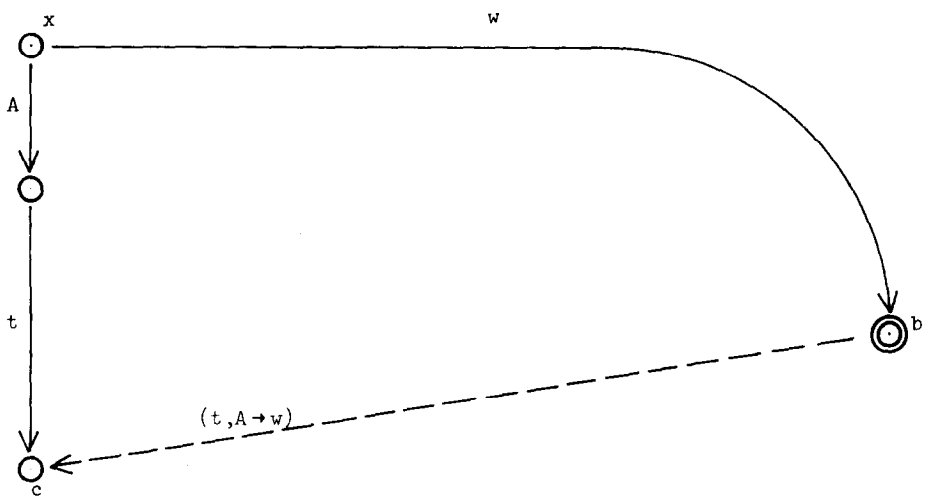


FIG. 1. Basic scheme: $A \Rightarrow w$.

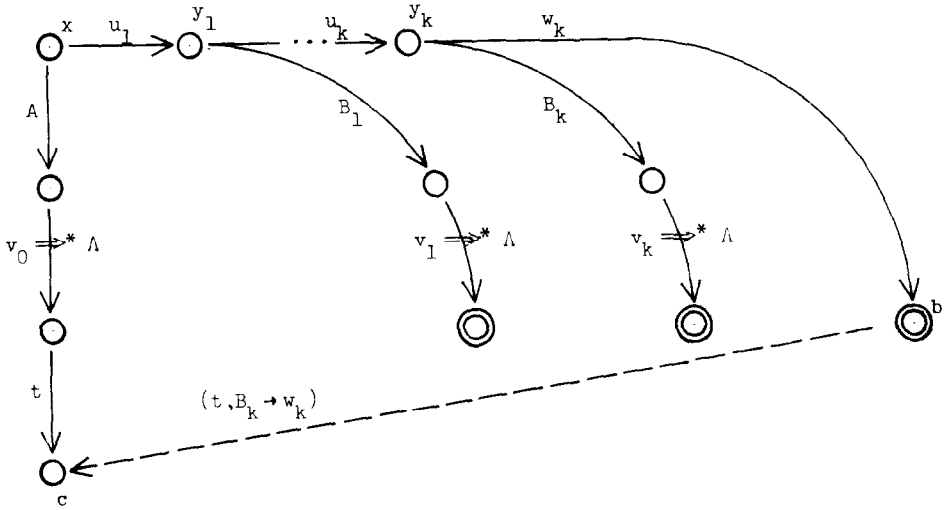


FIG. 2. General scheme: $Av_0 \Rightarrow^*_{rm} u_1 \cdots u_k B_k v_k \Rightarrow^*_{rm} u_1 \cdots u_k w_k$.

$r \in P \cup \{shift\}$.

$q_f$ is a new *final* state, not in $K$.

$a, b, c, d, s, x, y, z \in K \cup \{q_f\}$.

$\Lambda$ is the null string.

Let $\alpha \Rightarrow_{rm} \beta$ iff $\alpha = uBv$, $\beta = uwv$, $v \in \Sigma^*$ and $(B \rightarrow w)$ is a rule of $G$, i.e., $\beta$ is *directly derivable* from $\alpha$ via the rightmost nonterminal of $\alpha$. The transitive reflexive closure of $\Rightarrow_{rm}$ is written $\Rightarrow^*_{rm}$.

*Since the only derivations of interest in this paper will be rightmost derivations, we shall omit the* "rm" *hereafter.*

$\alpha$ is a *right sentential form* iff $S \Rightarrow^* \alpha$. If $S \Rightarrow^* \alpha B\gamma \Rightarrow \alpha w\gamma$, then $(B \rightarrow w)$ is the *handle* of $\alpha w\gamma$, with respect to this derivation, and $\gamma$ is the *lookahead string*. In this case, any prefix of $\alpha w$ is called a *viable* prefix, if $\alpha w\gamma \Rightarrow^* \beta$, $\beta \in \Sigma^*$.

In certain contexts, when $\alpha \Rightarrow^* \beta$, we will wish to consider a *fixed* (rightmost) derivation of $\beta$ from $\alpha$. In such cases we will abuse the $\Rightarrow^*$ notation slightly, implicitly restricting it to *subderivations* of the fixed derivation under consideration. For example, in the context of $S \Rightarrow^* \alpha\beta\gamma$, $S \Rightarrow^* \alpha A\gamma$ means that $\alpha A\gamma$ is an intermediate sentential form in the *particular derivation* of $\alpha\beta\gamma$ from $S$ that is under consideration.

$M$ is an *LR machine* for $G$ iff

(1) $\forall b \in K$ $\exists$ viable prefix $\alpha$ such that $\delta(q_0, \alpha) = b$ (i.e., every state is reachable from $q_0$ via a viable prefix), and

(2) $\forall \alpha$ $\delta(q_0, \alpha)$ is defined only if $\alpha$ is a viable prefix (i.e., only viable prefixes reach states).

We can now define the extended LR (XLR) machine for $M$,

$$M' = (K \cup \{q_f\}, N \cup \Sigma \cup \{\#\}, \delta', q_0, F, \rho),$$

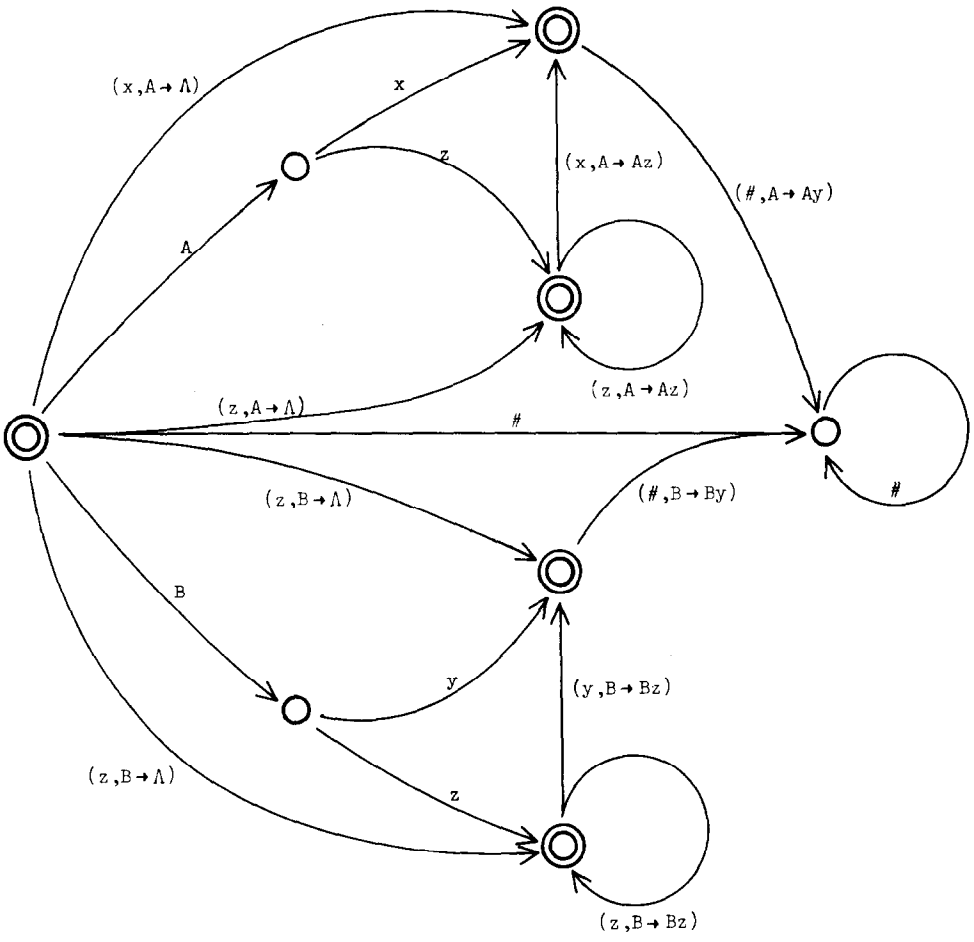


FIG. 3. XLR machine for grammar $G_3$.

where

$$\delta' : (K \cup \{q_f\}) \times (N \cup \Sigma \cup \{\neq\}) \to \mathcal{P}((K \cup \{q_f\}) \times (P \cup \{\textbf{shift}\}));$$

(1) $\forall\, b \in K, t \in N \cup \Sigma$, $(\delta(b, t), \textbf{shift}) \in \delta'(b, t)$ iff $\delta(b, t)$ is defined: (These are the **shift**-transitions of $M$, preserved in $M'$. The rest of what follows defines the new "reduce-arcs.")

(2) $\forall\, b \in K,\ t \in \Sigma,\ (s, (B \to w)) \in \delta'(b, t)$ iff $\exists\, \alpha,\ \gamma,\ A,\ u,\ v$ (Recall that $v \Rightarrow^* \Lambda$.) such that

$$S \Rightarrow^* \alpha A v t \gamma,$$

$$A \Rightarrow^* uB \Rightarrow uw,$$

$$b = \delta(q_0, \alpha u w),$$

and

$$s = \delta(q_0, \alpha A v t);$$

(3) $\forall\, b \in K$, $(q_f, (B \to w)) \in \delta'(b, \#)$ iff $\exists \alpha$

$$S \Rightarrow^* \alpha B \Rightarrow \alpha w,$$

and

$$b = \delta(q_0, \alpha w);$$

(4) $\delta'(q_f, \#) = \{(q_f, \textbf{shift})\}$.

An equivalent, but more algorithmic, definition of $M'$ can be found in Section 6. The present definition was chosen for its characterization of the lookahead information available in $M'$, which is key to the proofs of the results that follow.

### 2.2. *Characterization of* XLR *Lookahead Information*

$M'$ encodes an approximation to complete lookahead information for $G$. For any reduce state $b$ with applicable reduction $(B \to w)$, if $\alpha w \gamma$ is a right sentential form and $\alpha w$ takes $M$ to state $b$, then $\gamma\#$ takes $M'$ from $b$ to $q_f$, along a path beginning with a step labeled $(B \to w)$. This is stated formally in Theorem 2.1.

THEOREM 2.1. *If* $S \Rightarrow^+ \alpha w \gamma$ *by some fixed rightmost derivation with handle* $(B \to w)$, *then there exist* $r, s$ *such that* $(s, (B \to w)) \in \delta'(\delta(q_0, \alpha w), t)$ *and*

$$(q_f, r) \in \delta'(s, \beta),$$

*where* $\gamma\# = t\beta$, *or* $\gamma\# = t$ *and* $s = q_f$.

To prove this theorem, we first develop two lemmas.

LEMMA 2.2. *If* $S \Rightarrow^* \alpha w \beta t \gamma$ *by some fixed rightmost derivation with handle* $(B \to w)$, *then* $\delta'(\delta(q_0, \alpha w), \beta t)$ *includes a pair* $(\delta(q_0, \zeta t), r)$ *for each viable prefix* $\zeta$ *such that* $S \Rightarrow^* \zeta t \gamma$ *is a stage of the derivation* $S \Rightarrow^* \alpha w \beta t \gamma$.

*Proof.* The argument is by induction on the length of $\beta$. If $\beta = \Lambda$ then $S \Rightarrow^* \alpha B t \gamma \Rightarrow \alpha w t \gamma$. If $S \Rightarrow^* \zeta t \gamma \Rightarrow^* \alpha B t \gamma$, we know $\zeta \Rightarrow^* \alpha B$, and so either

(i) $\zeta = \alpha B$ or

(ii) we can partition $\zeta = u_1 A v$, where $A \Rightarrow^* u_2 B$, $v \Rightarrow^* \Lambda$, and $u_1 u_2 = \alpha$. In case (i), $S \Rightarrow^* \alpha B t \gamma$ and $B \Rightarrow^* B \Rightarrow w$, so that by case (2) of the definition of $\delta'$,

$$(\delta(q_0, \zeta t), (B \to w)) \in \delta'(\delta(q_0, \alpha w), t).$$

In case (ii), $S \Rightarrow^* u_1 A v t \gamma$ and $A \Rightarrow^* u_2 B \Rightarrow u_2 w$, and case (2) of the definition of $\delta'$ can be applied to achieve the same result as in case (i). If $\beta \neq \Lambda$, let $\beta = \beta' t$. In this

case, let $\gamma' = t\gamma$, so that $S \Rightarrow^* \alpha w\beta' t'\gamma'$. By induction, $\delta'(\delta(q_0, \alpha w), \beta)$ includes a pair $(\delta(q_0, \zeta' t'), r)$ for each viable prefix $\zeta'$ such that $S \Rightarrow^* \zeta' t'\gamma'$ is a stage of the derivation $S \Rightarrow^* \alpha w\beta' t' t\gamma$. If $\zeta$ is a viable prefix of $\zeta t\gamma$ as in the hypothesis, then $\zeta \Rightarrow^* \zeta' t'$. Thus either

(i) $\zeta = \zeta' t'$ or

(ii) $\zeta = u_1 B v_1$, $B \Rightarrow^+ u_2 t'$, $v_1 \Rightarrow^* \Lambda$, and $u_1 u_2 = \zeta'$.

In case (i), there is an $r$ such that $(\delta(q_0, \zeta), r) \in \delta'(\delta(q_0, \alpha w), \beta)$. Since $\zeta t$ is a viable prefix, $\delta(q_0, \zeta t)$ is defined, and so by case (1) of the definition of $\delta'$, $(\delta(q_0, \zeta t),$ **shift**$) \in \delta'(\delta(q_0, \alpha w), \beta t)$. In case (ii), $S \Rightarrow^* u_1 B v_1 t\gamma$, and for some $A \to w_2$, $B \Rightarrow^* u_3 A \Rightarrow u_3 w_2 = u_2 t'$. By case (2) of the definition of $\delta'$, $(\delta(q_0, \zeta t), (A \to w_2)) \in \delta'(\delta(q_0, \zeta' t'), t)$. By induction, there is an $r$ such that $(\delta(q_0, \zeta' t'), r) \in \delta'(\delta(q_0, \alpha w), \beta)$. Thus $(\delta(q_0, \zeta t), (A \to w_2)) \in \delta'(\delta(q_0, \alpha w), \beta t)$, by functional composition. ∎

LEMMA 2.3. *If $S \Rightarrow^+ \alpha w\gamma$ by some fixed rightmost derivation with handle $(A \to w)$, then there is an $r$ such that*

$$(q_f, r) \in \delta'(\delta(q_0, \alpha w), \gamma\not\in).$$

*Proof.* If $\gamma = \Lambda$, then case (3) of the definition of $\delta'$ applies directly. Otherwise, let $\gamma = \beta t$. By Lemma 2.2, $\delta'(\delta(q_0, \alpha w), \gamma)$ includes a pair $(\delta(q_0, \zeta t), r)$ for each viable prefix $\zeta$ such that $S \Rightarrow^* \zeta t$ is a stage in the derivation $S \Rightarrow^+ \alpha w\beta t$. Since the derivation $S \Rightarrow^+ \alpha w\gamma$ is nontrivial, there is at least one such $\zeta$. Consider the last step in the derivation $S \Rightarrow^* \zeta t$. Either

(i) $S \Rightarrow^* u_1 B$, $B \Rightarrow u_2 t$, and $u_1 u_2 = \zeta$, or

(ii) $S \Rightarrow^* \zeta t B$, $B \Rightarrow \Lambda$.

In case (i), case (3) of the definition of $\delta'$ gives

$$(q_f, (B \to u_2 t)) \in \delta'(\delta(q_0, \zeta t), \not\in),$$

and so, by functionl composition,

$$(q_f, (B \to u_2 t)) \in \delta'(\delta(q_0, \alpha w), \gamma\not\in).$$

Case (ii) is similar to case (i). ∎

*Proof of Theorem* 2.1. If $\gamma = \Lambda$, then the conclusion follows directly from the definition of $\delta'$, case (3). Otherwise, $\gamma = t\gamma'$. In this case there must be a stage in the derivation where this $t$ first appears to the right of the rightmost nonterminal: $S \Rightarrow^* u_1 C v t\gamma' \Rightarrow^* \alpha B t\gamma'$, where $C \Rightarrow^* u_2 B$, $u_1 u_2 = \alpha$, and $v \Rightarrow^* \Lambda$. By case (2) of the definition of $\delta'$, since $S \Rightarrow^* u_1 C v t\gamma'$ and $C \Rightarrow^* u_2 B \Rightarrow u_2 w$,

$$(\delta(q_0, u_1 C v t), (B \to w)) \in \delta'(\delta(q_0, \alpha w), t).$$

Thus $s = \delta(q_0, u_1 C v t)$ satisfies the first part of the conclusion. Since this is the first

stage at which $t$ appears to the right of the rightmost nonterminal, $t$ must be in the handle of $u_1 Cvt\gamma'$, or to the left of it. Partition $\gamma'$ into $w_1 w_2$, where $w_1$ is the portion of $\gamma'$ included in or to the left of the handle. Since $u_1 Cvtw_1$ is a viable prefix, $\delta(s, w_1)$ is defined, and by case (1) of the definition of $\delta'$, $(\delta(s, w_1), \textbf{shift}) \in \delta'(\delta(q_0, \alpha w), tw_1)$. By Lemma 2.2, there is an $r$ such that $(q_f, r) \in \delta'(\delta(s, w_1), w_2 \#)$, and so, by functional composition, $(q_f, r) \in \delta'(\delta(q_0, \alpha w), t\beta)$. ∎

THEOREM 2.4. *If* $(s, (B \to w)) \in \delta'(b, t)$ *then either*

(a) *there is a right sentential form* $\alpha wt\gamma$, *with handle* $(B \to w)$, *such that* $\delta(q_0, \alpha w) = b$, *or*

(b) $t = \#$ *and there is a right sentential form* $\alpha w$, *with handle* $(B \to w)$, *such that* $\delta(q_0, \alpha w) = b$.

*Proof.* By the definition of $\delta'$, either

(i) $S \Rightarrow^* u_1 Avt\gamma \Rightarrow^* u_1 u_2 wt\gamma$, *where* $A \Rightarrow u_2 B \Rightarrow u_2 w$, $b = \delta(q_0, u_1 u_2 w)$, and $s = \delta(q_0, u_1 u_2 Bvt)$, or

(ii) $S \Rightarrow^* \alpha B \Rightarrow \alpha w$, $t = \#$, and $b = \delta(q_0, \alpha w)$. ∎

Theorem 2.4 says that the 1-lookahead information of $M'$ is "optimal" for an LR machine with $M$'s states. Thus, in particular, if $M$ is the canonical LR(0) machine for $G$, $M'$ has the 1-lookahead information of the LALR(1) machine for $G$, and if $M'$ is LR(1), $M'$ has full LR(1) 1-lookahead information. (The effectiveness of $M'$'s lookahead information beyond the first symbol is considered in Section 4.)

## 3. THE PARSING ALGORITHM

The basic parse loop is as follows:

```
t := first input symbol;
s := q0;
while s ≠ qf do
begin if δ'(s, t) is multi-valued
   then (q, r) := resolve (s, t)
   else let {(q, r)} = δ'(s, t);
   if r = shift
   then begin push(s);
            s := q;
            t := next input symbol
        end
   else begin perform reduction by rule r,
            including "shift" of the left-hand side of r
        end
end
```

Function "resolve" runs finite state machine $M'$ on the input until all but one of the actions in $\delta'(s, t)$ has been ruled out.

```
function resolve (s, t);
begin x := t;
  S_0 := {q|(q, shift) ∈ δ'(s, t)};
  r_0 := shift;
  let {r_1,..., r_n} be the rules such that
    (q, r_i) ∈ δ'(s, t);
  for i := 1 to n do
    S_i := {q|(q, r_i) ∈ δ'(s, t)}
  while (more than one S_i ≠ ∅) do
  begin x := next input symbol;
    for i := 0 to n do
      S_i := {p|(p, r) ∈ δ'(q, x), q ∈ S_i}
  end;
  if S_i ≠ ∅ for some i
  then resolve := (q, r_i), where (q, r_i) ∈ δ'(s, t)
end;
```

There are two ways in which this function can fail. If the input is ungrammatical, either

(1) the end of the input will be encountered in the **while** loop, or

(2) upon exit from the **while** loop, all $S_i$'s will be empty.

The former will also happen for grammatical inputs if the grammar is not parsable by our method. (Section 4 deals with the problem of detecting such potentially unresolvable conflicts).

## 4. Comparisons to Other Parsing Methods

As shown by Theorem 2.4 in Section 2, the XLR(1) grammars are the same as the LALR(1) grammars, assuming we start with an LR(0) machine $M$. For $k > 1$, this relationship does not extend. Clearly, the XLR($k$) grammars are always a subset of the LALR($k$) grammars. To see that they form a proper subset, consider the following grammar $G_1$, which is SLR(2), and therefore LALR(2), but not XLR($\infty$):

$$
\begin{aligned}
G_1 : \; & S \to xAx \mid yAy \mid xCAy \mid xDwx, \\
& A \to w, \\
& C \to z, \\
& D \to z.
\end{aligned}
$$

In the previous example the weakness of the XLR machine is apparent. Information is lost when the state structure of $M'$ forces the joining of two distinct lookahead paths. On the other hand, for this to happen the grammar must be rather perverse, as the example also illustrates. It is simpler to find an example of a grammar that is XLR(2) but not SLR($k$) for any $k$:

$$\begin{aligned} G_2 : S &\to zAx \mid Ay \mid Cy, \\ A &\to b, \\ C &\to bx. \end{aligned}$$

This example shows that XLR shares some of the strength of the LR and LALR methods in the way that the uses of left and right context are coordinated.

If we are willing to allow unbounded lookahead, it is even possible to parse using non-LR grammars. Grammar $G_3$ (below) is XLR($\infty$) but not LR($k$) for any $k$:

$$\begin{aligned} G_3 : S &\to Ax \mid By, \\ A &\to Az \mid \Lambda, \\ B &\to Bz \mid \Lambda. \end{aligned}$$

The XLR machine for $G_3$ is shown in Fig. 3. The LR($k$) conflict is in the starting state, on input $z^k \cdots$. XLR($\infty$) resolves this conflict by the lookahead sets $z^*x\#$ for $(A \to \Lambda)$ and $z^*y\#$ for $(B \to \Lambda)$.

The XLR grammars form a proper subset of the LR-regular grammars of Čulik and Cohen. In their 1973 paper [3], they formally define this class of grammars and describe a parsing method which makes use of a *right-to-left* scanning finite state preprocessor to compress right context information, which is then used by a conventional left-to-right scanning pushdown automaton. The method is impractical, for two reasons. First, the finite state preprocessor is, in essence, assumed to be given with the grammar. No method is given for constructing a suitable machine; moreover, the problem of determining whether such a machine exists is stated to be undecidable. Second, at the present time, right-to-left scanning of large text files appears difficult, due to the unidirectional nature of most auxiliary storage devices. While the XLR method cannot handle all LR-regular grammars, it does not suffer from either of the above limitations, and so is more likely to be of practical value.

Although we have assumed the XLR machine is constructed from an LR(0) characteristic finite state machine, and so we obtained an extension of the LALR(1) method, this is not an inherent limitation of the construction. The XLR machine may just as well be constructed from an LR(1) machine, in which case it would yield a proper extension of the LR(1) method. Since it is based on a state transition graph, rather than a grammar, there should not even be any difficulty applying the construction to LR parsers obtained from regular right-part grammars [11], or via "state-splitting" techniques.

## 5. Detecting Unresolvable Conflicts

Of course, there are grammars for which the XLR($k$) or XLR($\infty$) parsing methods are inadequate. We present two algorithms, one for detecting conflicts not resolvable by XLR($\infty$), and another for determining which conflicts are resolvable by XLR($k$) for specific $k$. These algorithms are based on a flow-analysis approach, working on the XLR state graph.

A state $s$ has a potentially unresolvable XLR($\infty$) *conflict* if

$$\exists\, t, r_1 \neq r_2,\ p_1,\ p_2,\ w \text{ s.t.}$$
$$\{(p_1, r_1), (p_2, r_2)\} \subseteq \delta'(s, t),$$
$$tw \in \Sigma^* \#,$$

and

$$q_f \in \delta_1'(p_1, w) \cap \delta_1'(p_2, w),$$

where $\delta_1'(p, w)$ denotes $\{q \mid \exists\, r\ (q, r) \in \delta'(p, w)\}$. That is, there is a terminal string ending in $\#$ that appears to be a viable XLR lookahead for both $r_1$ and $r_2$. These conflicts can be detected by computing the relation

$$F = \{(p_1, p_2) \mid p_1 \neq p_2 \text{ and } \exists\, w \in \Sigma^* \text{ s.t. } q_f \in \delta_1'(p_1, w) \cap \delta_1'(p_2, w)\}.$$

This can be computed as the limit of $F_i$, where

$$F_1 = \{(p_1, p_2) \mid p_1 \neq p_2 \text{ and } \exists\, t \in \Sigma \text{ s.t. } \delta_1'(p_1, t) \cap \delta_1'(p_2, t) \neq \varnothing\}$$

and

$$F_{i+1} = F_i \cup \{(p_1, p_2) \mid p_1 \neq p_2 \text{ and } \exists\, t \in \Sigma,\ q_1, q_2 \text{ s.t.}$$
$$(q_1, q_2) \in F_i,\ q_1 \in \delta_1'(p_1, t) \text{ and } q_2 \in \delta_1'(p_2, t)\}.$$

Since this relation is sparse, it can be computed more efficiently than might immediately be apparent, using a hashing technique.

A state $s$ has a potentially unresolvable XLR($k$) *conflict* if

$$\exists\, t, r_1 \neq r_2,\ p_1,\ p_2, w \text{ s.t.}$$
$$\{(p_1, r_1), (p_2, r_2)\} \subseteq \delta'(s, t),$$
$$w \in \Sigma^{k-1},$$
$$\delta'(p_1, w) \neq \varnothing \text{ and } \delta'(p_2, w) \neq \varnothing.$$

That is, there is a terminal string of length $k$ that appears to be a viable XLR

lookahead for both $r_1$ and $r_2$. If we define $L_k(s)$ to be the set of "live" conflict-pairs for $k$-symbol lookahead from state $s$,

$$L_1(s) = \{(p_1, p_2) \mid \exists p_1, p_2, t, r_1 \neq r_2, \{(p_1, r_1), (p_2, r_2)\} \in \delta'(s, t)\}$$
$$L_{i+1}(s) = \{(q_1, q_2) \mid \exists p_1, p_2 \in L_i(s), t \in \Sigma \text{ s.t.}$$
$$p_1 \neq p_2,$$
$$q_1 \in \delta_1'(p_1, t) \text{ and } q_2 \in \delta_1'(p_2, t)\},$$

then state $s$ has an XLR($k$) conflict if and only if $L_k(s) \neq \varnothing$ or, for some $i < k$ and some $p$, $(p, p) \in L_i(s)$.

Since, for grammars of practical interest, we expect there to be few conflicts not resolvable by LALR(1) lookahead, there would be few states for which $L_1(s)$ would be nonempty, and in general, $L_k(s)$ should be small.

## 6. Computing the Reduce-Arcs

It is not immediately obvious from the definition of $\delta'$ that the reduce-arcs can be computed efficiently. In this section, we describe a method we have found practical, and prove that it produces the desired result—the same machine as defined by $\delta'$.

Our algorithm is based on a flow-analysis approach, where each pair $(x, A)$ of a state and a nonterminal symbol has an associated set $IN(x, A)$ of states. These sets are "pushed" around the state transition graph until the process converges. Let $x$ be any state and $A$ be any nonterminal symbol such that $\delta(x, A)$ is defined.

$$IN_0(x, A) = \{c \mid \exists v, t \ \ \delta(x, Avt) = c, v \Rightarrow^* \Lambda \text{ and } t \in \Sigma\},$$
$$IN_0(q_0, S) = \{q_f\},$$
$$IN_{i+1}(x, A) = IN_i(x, A) \cup \{c \mid \exists y, b, u, v \ \ c \in IN_i(y, B), v \Rightarrow^* \Lambda, B \Rightarrow uAv \text{ and } \delta(y, u) = x\},$$
$$IN(x, A) = \bigcup_{i=0}^{\infty} IN_i(x, A).$$

For simplicity in proving the following results, we introduce a nonstandard definition:

$$w_1 \Rightarrow^0 w_2 \quad \text{if} \quad w_1 = w_2 v, \quad \text{where} \quad v \Rightarrow^* \Lambda,$$
$$w_1 \Rightarrow^{i+1} w_2 \quad \text{if} \quad w_2 = u w_3 w_4, \quad \text{where} \quad w_1 \Rightarrow^i uAvw_4,$$
$$v \Rightarrow^* \Lambda, w_4 \in \Sigma^* \quad \text{and} \quad A \Rightarrow w_3.$$

It follows that $w_1 \Rightarrow^*_{(rm)} w_2$ if and only if $w_1 \Rightarrow^i w_2$ for some $i \geqslant 0$.

LEMMA 6.1. *$c \in IN_i(x, A)$ if and only if either*

(a) $(\exists v_1, v_2, B, t, u, y \quad \delta(y, Bv_1t) = c, \quad v_1, v_2 \Rightarrow^* \Lambda, \quad t \in \Sigma, \quad B \Rightarrow^i uAv_2$ *and* $\delta(y, u) = x)$ *or*

(b) $(c = q_f$ *and* $\exists u \; S \Rightarrow^i uA$ *and* $\delta(q_0, u) = x)$.

*Proof.* (only if) The argument is by induction on $i$. $c \in IN_i(x, A)$. If $i = 0$, unless $(x, A) = (q_0, S)$, it follows from the definition of $IN_0$ that there exist $v_1, t$ such that $\delta(x, Bv_1t) = c$, $v_1 \Rightarrow^* \Lambda$ and $t \in \Sigma$. Taking $u = v_2 = \Lambda$, $y = x$, and $B = A$, one gets $B \Rightarrow^0 uAv_2$, $\delta(y, u) = x$, and $v_2 \Rightarrow^* \Lambda$, so that (a) is satisfied. In the case that $i = 0$ and $(x, A) = (q_0, S)$, it may be that $c = q_f$, but in this case $S \Rightarrow^0 S$, $\delta(q_0, \Lambda) = q_0$ and so (b) is satisfied.

If $i > 0$ and $c \in IN_i(x, A)$ then by definition of $IN_i$, there exist $y, B, u, v$ such that $c \in IN_{i-1}(y, B)$, $v \Rightarrow^* \Lambda$, $B \Rightarrow uAv$, and $\delta(y, u) = x$. By induction, either (a) or (b) is satisfied for $c$ and $(y, B)$. If (a) holds, there exist $v_1, t, B_2, v_2, u_2, y_2$ such that $\delta(y_2, B_2v_1t) = c$, $v_1 \Rightarrow^* \Lambda$, $t \in \Sigma$, $B_2 \Rightarrow^{i-1} u_2Bv_2$, $\delta(y_2, u_2) = y$ and $v_2 \Rightarrow^* \Lambda$. Thus $\delta(y_2, u_2u) = x$, and $B_2 \Rightarrow^i u_2uAvv_2$, where $vv_2 \Rightarrow^* \Lambda$, while $\delta(y_2, B_2v_1t) = c$, so (a) is satisfied for $c$ and $(x, A)$. If (b) holds for $c$ and $(y, B)$, $c = q_f$, and there exists $u_2$ such that $S \Rightarrow^{i-1} u_2B$ and $\delta(q_0, u_2) = y$. Thus $S \Rightarrow^{i-1} u_2B \Rightarrow u_2uAv$, and $v \Rightarrow^* \Lambda$, so that $S \Rightarrow^i u_2uA$. Since $\delta(q_0, u_2u) = \delta(y, u) = x$, (b) is satisfied for $c$ and $(x, A)$.

(if) The argument is by induction on $i$. If (a) is satisfied and $i = 0$, then $B = A$, $u = v_2 = \Lambda$, and $x = y$, so that $\delta(x, Av_1t) = \delta(y, Av_1t) = c$, and therefore $c \in IN_0(x, A)$. If (b) is satisfied and $i = 0$, then $c = q_f$, $S = A$, and $x = q_0$, so that $c \in IN_0(x, A)$. If (a) is satisfied and $i > 0$, there must be an intermediate stage $\alpha Cv_3\gamma$ in the derivation $B \Rightarrow^i uAv_2$, so that $B \Rightarrow^{i-1} \alpha Cv_3\gamma \Rightarrow^* uAv_2$, where $\gamma \in \Sigma^*$, $C \Rightarrow \beta$, $v_3 \Rightarrow^* \Lambda$ and $uAv_2 = \alpha\beta\gamma$. Since $\alpha\beta$ must contain $A$ somewhere, two cases arise:

(i) $\alpha = uAv_4$, $v_4\beta\gamma = v_2 \Rightarrow^* \Lambda$, $B \Rightarrow^{i-1} \alpha Cv_3\gamma = uAv_4Cv_3\gamma$, $v_4Cv_3\gamma \Rightarrow^* \Lambda$, and so, by induction, $c \in IN_{i-1}(x, A)$, and we are done.

(ii) $u = \alpha u_4$, $\beta = u_4Av_4$, $v_4\gamma = v_2 \Rightarrow^* \Lambda$, $B \Rightarrow^{i-1} \alpha Cv_3\gamma$, $v_3\gamma \Rightarrow^* \Lambda$, and so, by induction $c \in IN_{i-1}(\delta(y, \alpha), C)$.

In the latter case, by definition of $IN_i$, since $C \Rightarrow \beta = u_4Av_4$, $\delta(y, u) = \delta(\delta(y, \alpha), u_4) = x$, and $v_4 \Rightarrow^* \Lambda$, it follows that $c \in IN_i(x, A)$.

If (b) is satisfied and $i > 0$, then there is an intermediate stage $\alpha Cv_3$ in the derivation $S \Rightarrow^i uA$, so that $S \Rightarrow^{i-1} \alpha Cv_3 \Rightarrow^* uA$, where $v_3 \Rightarrow^* \Lambda$, $C \Rightarrow \beta$, and $uA = \alpha\beta$. Again, there are two cases:

(i) $\alpha = uA$, in which case $S \Rightarrow^{i-1} uA$, and, by induction, $c \in IN_{i-1}(x, A) \subseteq IN_i(x, A)$.

(ii) $\beta = u_4A$, in which case $S \Rightarrow^{i-1} \alpha Cv_3$, $v_3 \Rightarrow^* \Lambda$, and, by induction, $c \in IN_{i-1}(\delta(q_0, \alpha), C)$. By definition of $IN_i$, since $C \Rightarrow \beta = u_4A$ and $\delta(\delta(q_0, \alpha), u_4) = \delta(q_0, u) = x$, $c \in IN_i(x, A)$.

THEOREM 6.2. *$(c, (B \to w)) \in \delta'(b, t)$ if and only if $\exists x \; c \in IN(x, B)$, $\delta(x, w) = b$, $c$ is a state entered on terminal symbol $t$, and $(B \to w)$ is a rule of $G$.*

*Proof.* (only if) There are two relevant cases in the definition of $\delta'$:

(2) Suppose case (2) of the definition holds for $b$, $(B \to w)$, $\alpha, \gamma, A, u, v, s, t$. Taking $y = \delta(q_0, \alpha)$ and $x = \delta(q_0, \alpha u)$, $\delta(y, Avt) = s$, $v \Rightarrow^* \Lambda$, $t \in \Sigma$, $A \Rightarrow^* uB$ and $\delta(y, u) = x$, so that $s \in IN(x, B)$ by Lemma 6.1.

(3) Suppose case (3) of the definition holds for $b$, $(B \to w)$, and $\alpha$. Taking $x = \delta(q_0, \alpha)$, $S \Rightarrow^* \alpha B$, so that, by Lemma 6.1, $q_f \in IN(x, B)$.

(if) Again there are two cases:

(a) Suppose case (a) of Lemma 6.1 holds for $c \in IN(x, B)$. Then $\exists v_1, v_2, A, t, u, y$ such that $\delta(y, Av_1t) = c$, $v_1, v_2 \Rightarrow^* \Lambda$, $t \in \Sigma$, $A \Rightarrow^* uBv_2 (\Rightarrow^* uB)$ and $\delta(y, u) = x$. Since $B \Rightarrow w$, $A \Rightarrow^* uB \Rightarrow uw$, and by the LR-ness of $M$, $\exists \alpha, \gamma$ such that $S \Rightarrow^* \alpha A v_1 t \gamma$, and $\delta(q_0, \alpha) = y$. Thus $\delta(q_0, \alpha uw) = b$, and $\delta(q_0, \alpha A v_1 t) = c$, so that part (2) of the definition of $\delta'$ is satisfied.

(b) Suppose case (b) of Lemma 6.1 holds for $c \in IN(x, B)$. Then $c = q_f$ and $\exists u$ $S \Rightarrow^i uA$ and $\delta(q_0, u) = x$. Since $\delta(x, w) = b$, so that $\delta(q_0, uw) = b$, and $B \Rightarrow w$, case (3) of the definition of $\delta'$ is satisfied.

Based on Theorem 6.2, we can write the following algorithm for computing reduce-arcs:

ALGORITHM 6.3.

1. **for** each state $x$ and nonterminal $A$ such that $\delta(x, A)$ is defined, or $x = q_0$ and $A = S$ **do** compute $IN_0(x, A)$ and put the pair $(x, A)$ into the queue $Q$.
2. **for** each pair $(x, A)$ discovered in step 1 **do** compute the set $\text{NEXT}(x, A) = \{(y, B) | \exists w = uBv,\ v \Rightarrow^* A,\ \delta(x, u) = y \text{ and } A \Rightarrow w\}$.
3. **while** $Q$ not empty **do**
   **begin** remove $(y, B)$ from head of $Q$;
   **for** each $(x, A) \in \text{NEXT}(y, B)$ **do**
   **if** $IN(y, B) - IN(x, A) \neq \varnothing$
   **then begin** $IN(y, B) := IN(y, B) \cup IN(x, A)$;
   **if** $(y, B) \notin Q$ **then** add $(y, B)$ to $Q$
   **end**
   **end**
4. **for** each pair $(y, B)$ such that $IN(y, B) \neq \varnothing$ **do**
   **for** each rule $(B \to w)$ **do**
   **for** each $c \in IN(y, B)$ **do**
   add$(c, (B \to w))$ to $\delta'(\delta(y, w), t)$, where state $t$ is the (unique) symbol on which state $c$ is entered.

A recent paper by DeRemer [6] on efficient algorithms for computing LALR(1) lookahead sets makes use of state relationships very similar to those behind the XLR construction. It appears likely that application of DeRemer's techniques to the XLR construction may produce a refined algorithm that is more efficient than naïve implementation of Algorithm 6.3, and perhaps even as fast as his LALR(1) constructor.

## 7. Conclusions

We have programmed and tested our algorithm for computing the XLR reduce-arcs, proving empirically that XLR parsers can be computed for real programming languages. Since the Pascal grammar we used in testing has only one LALR(1) conflict that does not trace back to an ambiguity, we would not expect much difference in performance between our parsing algorithm and the LALR(1) on this grammar. In further research, perhaps rewriting the Pascal grammar in a terser form, or by choosing a harder to parse language, it will be possible to introduce enough LALR(1) conflicts that some data on the speed/generality trade-off can be obtained.

While XLR($\infty$) grammars, such as $G_3$, are interesting from a theoretical standpoint, we do not seriously propose that parsers be permitted completely unbounded lookahead. Instead, it is reasonable to assume some fixed input buffer size $k$, and restrict the method to grammars for which this amount of lookahead is sufficient. We have programmed and tested the method described in Section 5 for detecting XLR($\infty$) conflicts, but we have not yet done the same for XLR($k$) conflicts. We intend to do this, as well as testing whether the parser size may be significantly reduced by "pruning" reduce-arcs that are not needed to resolve actual parsing conflicts.

The potential usefulness of the XLR machine is probably greater in error recovery and repair than it is in "straight" parsing. History has shown that programming language syntax and semantics can be tailored, regardless of the effort, to fit parsing methods even more constrained than LALR(1). When it comes to error recovery, however, the story is different. The LR methods are known to have a "left-bias." Since what has been parsed leading up to an error is a viable prefix of *some* program, it is presumed to be correct. For effective error recovery, some way of also utilizing information from the right context is needed.

Three solutions to this problem, very similar in nature, have been proposed [7, 12, 13]. The basic idea of these proposals is to restart the LR PDA, nondeterministically, in all of its states, and run it until it has partially reduced the right context, or until it encounters a choice of stack operations. Methods based on this idea must deal with two problems:

(1) the resulting parse is not necessarily canonical, and so may wreak havoc with semantic processing;

(2) when the nondeterministic PDA is faced with alternative stack operations, (as is likely to happen before the parse has progressed very far), it is forced to either give up, heuristically choose one of the alternatives, continue parsing, in parallel, with multiple stacks, or restart in the initial recovery state.

An additional problem is the simulation of the nondeterministic PDA, which may be made deterministic, but only at the cost of an increase in the number of states. While the authors of the three papers cited above have made significant progress toward solving these problems, each choosing different trade-offs, their methods still

pay a price in increased complexity and memory overhead, which is difficult to balance against the improvement in effectiveness of error recovery.

We propose a simpler approach, with lower overhead, that would not interfere with semantic processing: When an error is encountered and several recovery actions appear reasonable on the basis of left context, the action is chosen that allows the XLR machine, functioning as a finite automaton, to scan furthest ahead on the input buffer. While this method is also nondeterministic, and therefore subject to some of the same overhead costs as the methods based on a nondeterministic PDA, problems (1) and (2) are eliminated.

Whether the approach to error recovery proposed here is justified can only be determined empirically. We hope to do this in further research.

## REFERENCES


1. A. V. AHO AND S. C. JOHNSON, LR parsing. *Comput. Surv.* **6**, No. 4 (1974).
2. A. V. AHO AND J. C. ULLMAN, "The Theory of Parsing, Translation, and Compiling," Prentice–Hall, Englewood Cliffs, N.J., 1972.
3. K. ČULIK II AND R. COHEN, LR-regular grammars—An extension of LR($k$) grammars, *J. Comput. Syst. Sci.* **7** (1973), 66–96.
4. F. L. DEREMER, "Practical Translators for LR($k$) Languages," Ph.D. thesis, Dept. of Electrical Engineering, MIT, Cambridge, Mass., 1969.
5. F. L. DEREMER, Simple LR($k$) grammars, *Comm. ACM* **14**, No. 7 (1971), 453–460.
6. F. L. DEREMER, Efficient computation of LALR(1) look-ahead sets, Proceedings of the SIGPLAN Symposium on Compiler Construction, Assoc. Comput. Mach., New York, 1979.
7. F. C. DRUSEIKIS AND G. D. RIPLEY, "Error Recovery for Simple LR($k$) Parsers," Dept. of Computer Science, Univ. of Arizona, Tucson, 1976.
8. J. E. HOPCROFT AND J. D. ULLMAN, "Formal Languages and Their Relation to Automata," Addison–Wesley, Reading, Mass., 1969.
9. D. E. KNUTH, On the translation of languages from left to right, *Inform. Control.* **8** (Oct. 1965), 607–639.
10. W. R. LALONDE, E. S. LEE, AND J. J. HORNING, An LALR($k$) parser generator, Proc. IFIP Congress 71, TA-3, North-Holland, Amsterdam, 1971.
11. W. R. LALONDE, "Practical LR Analysis of Regular Right Part Grammars," Ph.D. thesis, University of Waterloo, Waterloo, Ont., 1975.
12. M. D. MICKUNAS AND J. A. MODRY, Automatic error recovery for LR parsers, *Comm. ACM* **21**, No. 6 (1978), 459–465.
13. T. J. PENNELLO AND F. L. DEREMER, A forward move algorithm for LR error recovery, "Conf. Record of the Fifth Annual ACM Symposium on POPL (1978)," pp. 231–240.
14. U. S. Department of Defense, "Reference Manual for the Ada Programming Language—Proposed Standard," July 1980.


*Printed in Belgium*