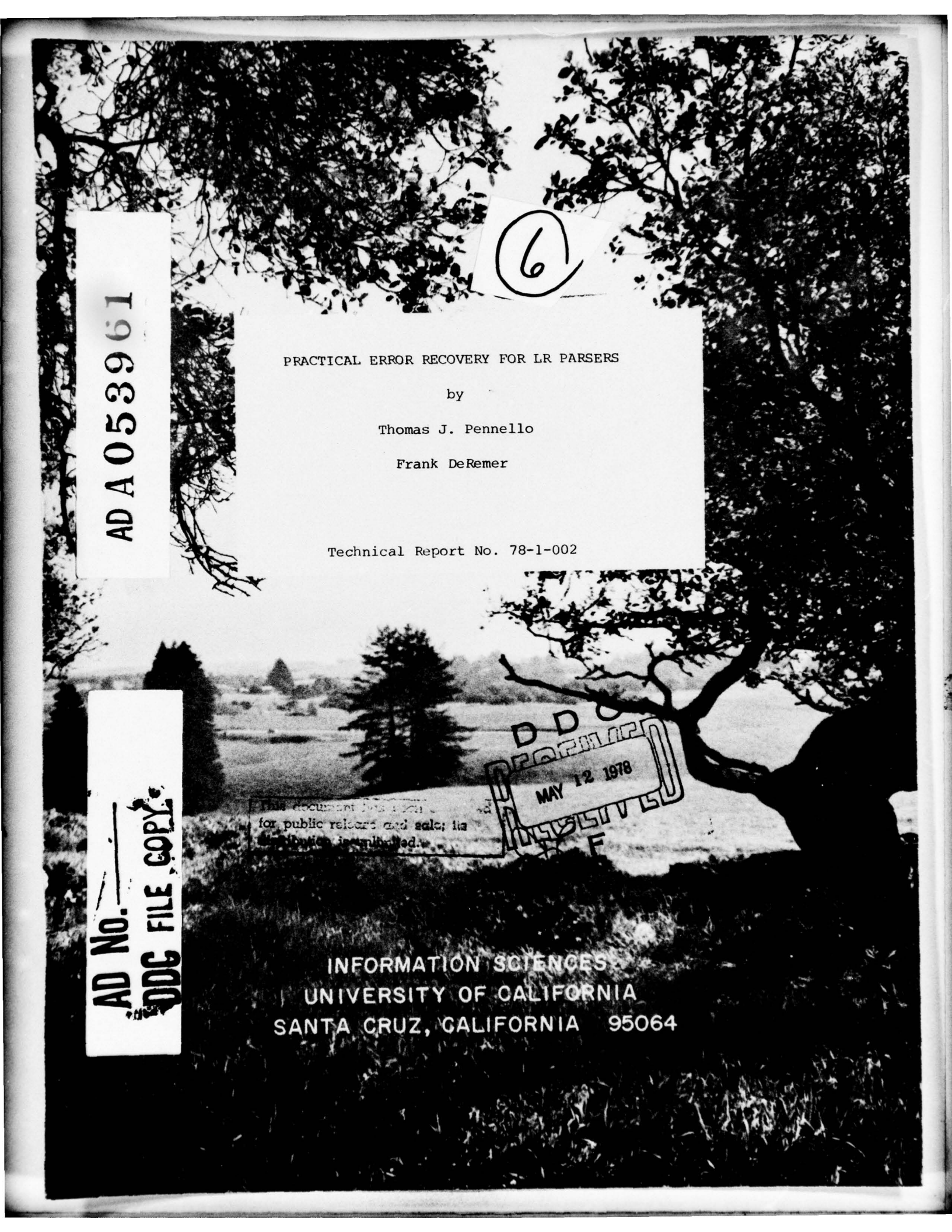


(6)

PRACTICAL ERROR RECOVERY FOR LR PARSERS

by

Thomas J. Pennello

Frank DeRemer

Technical Report No. 78-1-002

INFORMATION SCIENCES
UNIVERSITY OF CALIFORNIA
SANTA CRUZ, CALIFORNIA 95064

NR 049-391
Code 437

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER: 78-1-002 (14 TR-78-1-112)
2. GOVT ACCESSION NO.
3. RECIPIENT'S CATALOG NUMBER

4. TITLE (and Subtitle): PRACTICAL ERROR RECOVERY FOR LR PARSERS (6)
5. TYPE OF REPORT & PERIOD COVERED: Technical rept. (9)
6. PERFORMING ORG. REPORT NUMBER

7. AUTHOR(s): Thomas J. Pennello (10), Frank DeRemer
8. CONTRACT OR GRANT NUMBER(s): (15) ONR N00014-76-C-0682

9. PERFORMING ORGANIZATION NAME AND ADDRESS
William M. McKeeman and Sharon Sickel
Information Sciences, UCSC, Rm. 239 AS
Santa Cruz, Ca. 95064

10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS

11. CONTROLLING OFFICE NAME AND ADDRESS
Office of Naval Research
Arlington, Virginia 22217

12. REPORT DATE: December 10, 1977
13. NUMBER OF PAGES: 43 (12) 48 p.

14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)
Office of Naval Research
University of California
553 Evans Hall
Berkeley, California 94720

15. SECURITY CLASS. (of this report): (11) 19 Dec 77
15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)
This document has been approved for public release and sale; its distribution is unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

A "forward move algorithm" and some of its formal properties are presented for use in a practical syntactic error recovery scheme for LR parsers. The algorithm finds a "valid fragment" (comparable to a valid prefix) just to the right of a point of error detection. For expositional purposes the algorithm is presented as parsing arbitrarily far beyond the point of error detection in a "parallel" mode, as long as all parses agree on the read or reduce action to be taken at each parse step. (OVER)

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102 LF 014 6601

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

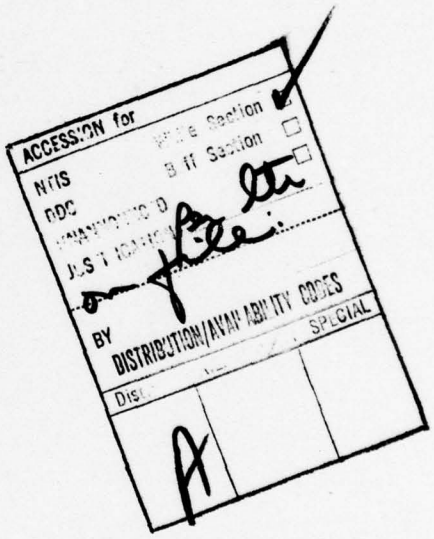
410 359

20. ABSTRACT (continued)

In practice the forward move is achieved serially by adding "recovery states" to the LR machine. Based on the formal properties of the forward move we propose a practical error recovery algorithm that uses the "right context" accumulated by the forward move. The performance of the recovery algorithm is illustrated in a specific case and discussed in general.

Key words and phrases: syntax errors, error recovery, parsing, LR(k), SLR(k), LALR(k).

CR categories: 4.12, 4.42, 5.23.



PRACTICAL ERROR RECOVERY FOR LR PARSERS

by

Thomas J. Pennello

Frank DeRemer

December 10, 1977

Board of Studies in Information Sciences
University of California at Santa Cruz
Santa Cruz, California 95064

Portions of this paper were presented at the Fifth ACM Symposium on the Principles of Programming Languages, sponsored jointly by ACM SIGPLAN and SIGACT, January 23-25, 1978, Tucson, Arizona, under the title, "A Forward Move Algorithm for LR Error Recovery." Research was supported in part by the Office of Naval Research under Contract N00014-76-C-0682. Submitted for publication to the Communications of the ACM.

CONTENTS

ABSTRACT

A "forward move algorithm" and some of its formal properties are presented for use in a practical syntactic error recovery scheme for LR parsers. The algorithm finds a "valid fragment" (comparable to a valid prefix) just to the right of a point of error detection. For expositional purposes the algorithm is presented as parsing arbitrarily far beyond the point of error detection in a "parallel" mode, as long as all parses agree on the read or reduce action to be taken at each parse step. In practice the forward move is achieved serially by adding "recovery states" to the LR machine. Based on the formal properties of the forward move we propose a practical error recovery algorithm that uses the "right context" accumulated by the forward move. The performance of the recovery algorithm is illustrated in a specific case and discussed in general.

Key words and phrases: *syntax* errors, error recovery, parsing, LR(k), SLR(k), LALR(k).

CR categories: 4.12, 4.42, 5.23.

0. INTRODUCTION

Over the past twenty years much effort has been invested into the science of deterministic parsing; that is, determining the phrase structure of a sentence generated by a context-free grammar [H&U 69] during a single scan, usually from left to right. The two pinnacles of this research are the LL(k) and LR(k) grammars and their parsers, respectively top-down and bottom-up techniques [A&U 72].

Unfortunately, the more adept parsing techniques have gotten, the more difficult it has seemed to achieve flexible error recovery. It seems that the more the parser knows about the input possibilities and specializes itself via state transitions to restricted parts of itself, the more difficult it is for it, in the face of a detected error, to back out and get global information necessary for good error recovery. In the words of Graham and Rhodes [G&R 75]: "The fact that the next move of the parser can depend on the entire correct prefix already analyzed makes it difficult or impossible to start up the parser after the error [detection] point."

This paper is a contribution toward giving LR parsers some such global capabilities. Indeed, we show that it is easy to extend them to start up after an error is detected and to parse arbitrarily far ahead, gathering right context. This context can then be used to guide the selection and evaluation of repair attempts. Thus, we decompose the notion of error recovery into (1) gathering right context and (2) a repair strategy.

History. Graham and Rhodes [G&R 75] proposed an error recovery scheme for deterministic bottom-up parsers that involves "condensing" context about the point at which an error was detected. A "backward move" condenses context to the left and a "forward move" gathers context to the right. Such

context is valuable input to an error repair strategy. Graham and Rhodes show how the condensation is done for simple precedence parsers and give an error repair strategy that uses the condensed context.

We have adapted the general idea of Graham and Rhodes to LR parsers [A&J 74], by which we mean LR(k) parsers and all their variants: LALR(k), SLR(k), etc. Some of the investigation has already been reported in Pennello's Master's Thesis [Pen 77] (see also [O'H 76]). The present paper refines the theoretical results developed in [Pen 77] and adds some algorithms and some empirical results from a recent implementation.

Briefly, we found the "backward move" to be detrimental in enough cases that we abandoned it, in favor of a philosophy of trying to do only what is consistent with every context for as long as possible, resorting to guesses only when we know of no other way to proceed. Pennello developed a "parallel parse" exposition of the "forward move" for LR parsers that facilitates understanding and proof of results, and we show how to "serialize" it so that in practice the parser simply has some extra "recovery states" that work just as the other states do, but are entered only in recovery mode. Several theorems were developed, primarily relating to the "derived valid fragment" accumulated during the forward move. Druseikis and Ripley [D&R 77] have reported some similar results which we note as the issues arise.

Preview. We first present the forward move algorithm (FMA) in its parallel form. Then we state several properties of FMA that are relevant to error repair. These properties suggest ways that the "forward context" may be used in a repair strategy. Perhaps the most important of these indicates that the forward context may be used to efficiently verify that a repair attempt is "consistent" with the input text parsed by FMA.

Next we present several algorithms that are used in the repair strategy. The analysis there starts with the simplifying assumption of but a single error of insertion, replacement, or deletion of a single terminal symbol. The effect of delayed versus immediate detection is discussed, and then multiple errors are treated. For clarity and simplicity the algorithms are presented without regard to certain practicalities, which are then discussed in the text.

Finally, we show how to serialize FMA for a practical implementation. We present some statistics on how many extra states are needed for some well known programming languages. Then we demonstrate how our error recovery performed on the example Algol program of Graham and Rhodes [G&R 75].

1. TERMINOLOGY

We review basic notation and terminology for strings, grammars, and parsers. A vocabulary (or alphabet) V is a finite set of symbols. V^* denotes the set of all strings of symbols from V. V^+ denotes V^* less Λ, the empty string. If x is a nonempty string, First x denotes the first symbol of x and Rest x denotes x stripped of its first symbol. (We typically do not put parentheses around arguments to functions when the meaning is clear, as above.)

A context-free grammar G is a quadruple (N,T,S,P), i.e. the nonterminals, terminals, start symbol, and productions, respectively; we define $V = N \cup T$. Each production is a pair (A,w), left part and right part, written $A \rightarrow w$, where $A \in N$ and $w \in V^*$. $\rightarrow$ is the rightmost derivation relation, in which the rightmost nonterminal is replaced at each step; $\rightarrow^+$ is its transitive closure and $\rightarrow^*$ is its transitive-reflexive closure. We assume a production $S \rightarrow S'\perp \in P$, where $S' \in N$, $\perp \in T$, and neither S nor $\perp$ appear in any other production. The language generated by G is $L(G) = \{w \in T^* \mid S \rightarrow^+ w\}$.

An LR parser, i.e. an LR(1), LALR(1), SLR(1), or other such parser [A&J 74], for G=(N,T,S,P) is a sextuple (K,V,P,START,SIGMA,REDUCE) where K is a finite set of states, START $\in$ K is the start state, SIGMA is the transition function mapping K x V into K, and REDUCE maps K x V into 2^P. If SIGMA(q,h) = p, we also write this as the transition $q \xrightarrow{h} p$. From SIGMA and REDUCE we derive the parser decision function PD, mapping K x V into 2^M, where M = {read, accept} $\cup$ P; PD indicates, for a given state and input symbol, all possible actions the parser may take; if the grammar G is LR, PD always yields at most a singleton set. PD is defined as PD(q,h) = {read | $q \xrightarrow{h} q'$ for some $q' \in K$} $\cup$ {accept | $h = \perp$ and $q \xrightarrow{h} q'$ for

some $q' \in K\} \cup$ REDUCE(q,h).

In figures we represent LR parsers as state diagrams in which states are connected by arcs labelled with elements of V, according to SIGMA. For each state q in which REDUCE indicates a possible choice of a reduction by production p, we list p and its 1-symbol look-ahead set, $\{h \in V \mid p \in$ REDUCE$(q,h)\}$. Figure 1 depicts a state diagram for the LALR(1) parser for a common arithmetic expression grammar; in this figure, for example, PD$(**_0,i)$ = {read}, SIGMA$(**_0,i) = i_0$ and PD$(i_0,+)$ = $\{P \rightarrow i\}$.

Note that REDUCE and PD may take a nonterminal as a second argument. LR parser constructor algorithms easily generalize to include nonterminals in look-ahead sets. We assume their inclusion, but also give alternate means of implementing the results of this paper should their inclusion in some implementation be too difficult. Also note that in Figure 1, there happen to be *no nonterminals* in look-ahead sets due to the nature of the grammar.

A path P in an LR parser is a sequence of states $q_0, q_1, \ldots, q_n$ such that $q_0 \xrightarrow{w_1} q_1$, $q_1 \xrightarrow{w_2} q_2$, ..., $q_{n-1} \xrightarrow{w_n} q_n$. We define Top P = q_n. We say that P spells $w = w_1 w_2 \ldots w_n$; we define Spelling P = w. An alternate notation for P is $[q_0:w]$, given the parser or its state diagram. We abbreviate [START:w] by [w]; thus [] denotes START alone. We say that w accesses q iff Top [w] = q. The concatenation of two paths [q:y] and [q':y'] where Top [q:y] = q' is written [q:y][q':y'] and denotes [q:yy']. If, for some q, q', and h, SIGMA(q',h) = q, then Accessing_symbol q = h (the accessing symbol for each state is unique, except that START has no accessing symbol).

An LR parser configuration is a pair (Z,R) where Z is a path and $R \in T^+$.

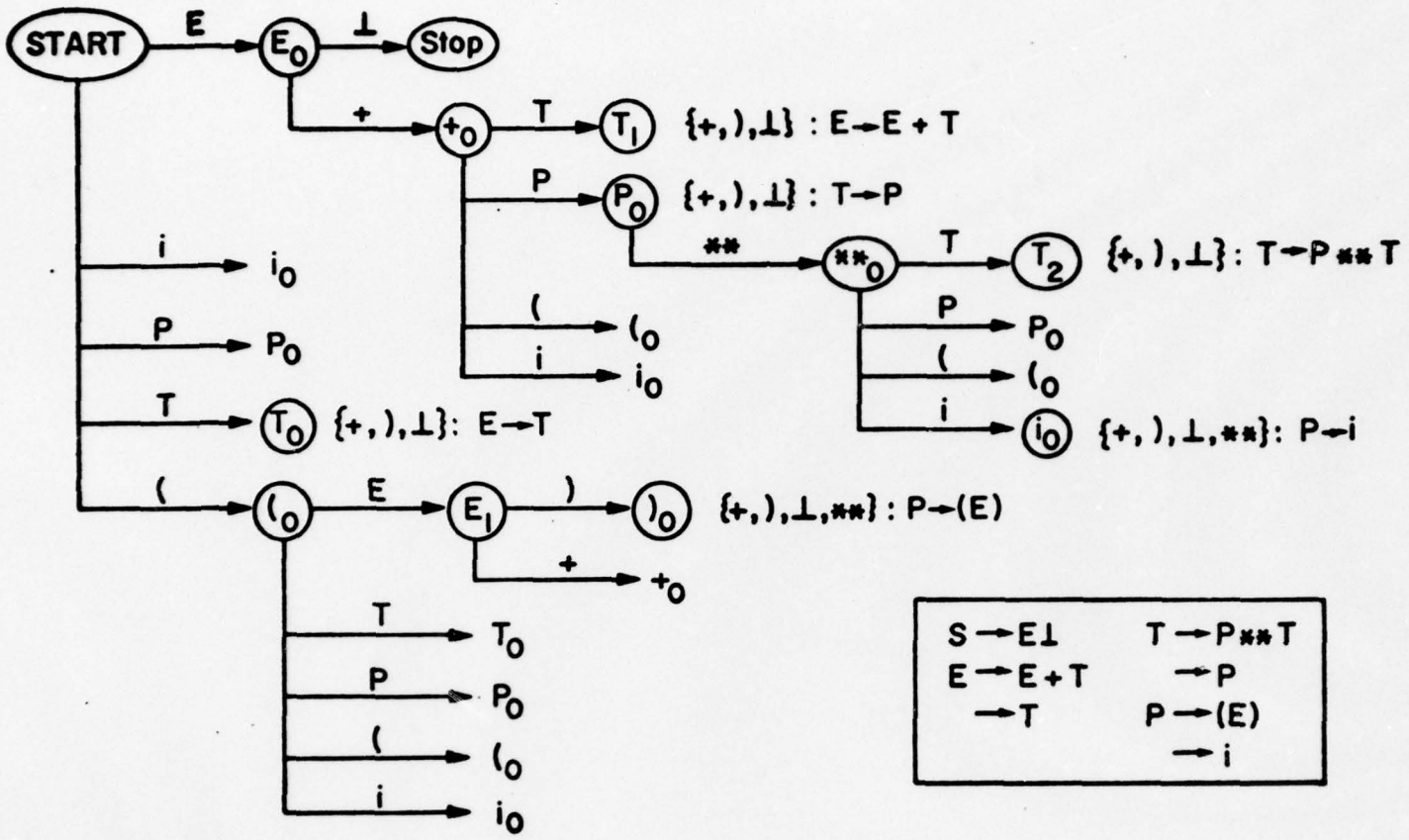


Fig. 1. A simple arithmetic expression grammar and its LALR(1) parser.

For example, when the first three symbols of the sentence i+i+⊣ have been reduced to E+T by the parser of Figure 1, it is in the configuration ([E+T], +i⊣) = ((START,E_0,$+_0$,T_0), +i⊣).

The parser moves from one configuration to the next by reading or reducing; thus we define a move as an element of the set {read} ∪ P. $\vdash$ denotes a move from one configuration to another; $\vdash^+$ is its transitive closure and $\vdash^*$ is its transitive-reflexive closure. We sometimes use $\vdash \ldots \vdash$ for $\vdash^+$. If C $\vdash$ C' by move M, we sometimes write C $\vdash_M$ C'. We define $\vdash$ as follows: Given some ([q:y],R), consider the possible values of M = PD(Top[q:y],First R):

case {read}: Let h = First R. Then

([q:y],R) $\vdash_{read}$ ([q:yh],Rest R).

case {A → w}: If [q:y] = [q:y'w] for some y', then

([q:y],R) $\vdash_{A \to w}$ ([q:y'A],R).

case {accept} or {} or |M| > 1: There is no C such that ([q:y],R) $\vdash$ C. (I.e. the parser makes no move but instead halts, accepting or rejecting the input, or unable to proceed deterministically.)

The language recognized by an LR parser is {w ε T^+ | ([],w) $\vdash^+$ accept} where we abbreviate ([S'],⊣) by accept. (Note that PD(Top[S'],⊣) = {accept}.) The usual LR parsing algorithm is easily deduced from the $\vdash$ relation.

A final note: We present algorithms that may return results in two different ways: via a return value and/or via so-called "result" parameters. E.g., the phrase "if F(x,y,z) gives A, B ..." tests the boolean return value of F which is called with the three input expressions (actual parameters) x, y, and z, and with the two result parameters, A and B; some side-effect will

happen to A and B according to the definition of F. Returning from such a function is indicated by a statement such as "return True giving u, v."

2. FORWARD MOVE ALGORITHM

When an error is detected, we wish to perform a "forward move" that parses the input after the point of the error detection. The parse cannot depend upon the left context already developed on the stack to proceed, since it is precisely that left context that causes the parser to detect the error. Thus, we devise an algorithm that parses ahead starting with no left context. Our formulation of the "forward move algorithm" keeps parsing input text until it must refer to the "missing" left context to proceed. At that point it halts, and we use the developed "forward context" in an error repair strategy.

Consider an Algol-like language in which the symbol "do" can appear in a "for" or "while" construct. Suppose the two productions involving these constructs are

```
Stmt → for Id := Exp step Exp
           until Exp do Stmt          (1)
Stmt → while Exp do Stmt              (2)
```

where we capitalized nonterminals and left terminals uncapitalized. Now consider the erroneous phrase

```
for X := 1 step 1 until     do begin J := X end
```

where we have omitted the third Exp in the "for" construct. The forward move, if started with its input head at the symbol "do," reduces "do begin J := X end" to "do Stmt." It goes no further than this because it needs to know left context to determine whether at this point it must reduce by production (1) or (2) (each of which end in "do Stmt," not coincidentally).

The reason the forward move can parse this much of the input is because in both places that "do" appears in the grammar, it is followed immediately

by Stmt. Thus, context to the left of the "do" is not necessary to reduce "do begin J := X end" to "do Stmt." This situation occurs often enough in programming languages that it is not uncommon for the forward move to make quite some progress in the input text before it needs to refer to left context.

The essential idea of our algorithm is to carry out all possible parses of the input text, as long as all parses agree as to the next move to make (i.e. they must all manipulate the stack in the same way at each parse step) and no parse refers to nonexistent left context. We present the algorithm first as having a stack upon which we push sets of states rather than states; these sets of states keep track of the parallel parses. At each step of the forward move we inquire of each state in the set on the top of the stack what its decision is with regard to the next symbol in the input. If all the states in that top state set that accept the next input symbol agree as to the next move, and this next move does not refer to nonexistent left context, then we make that next move. For example, in the case of the "read" move, we push on the stack the set of all the states that can be reached from any state in the top state set by taking a transition on the next input symbol.

Of course, manipulating sets of states is not practical, but we show how the forward move algorithm can be easily converted into an algorithm that manipulates states only, essentially like the conversion of a nondeterministic finite state machine to a deterministic one. The converted algorithm is as fast as the LR parsing algorithm.

Let ? be the set K of all the parser states. We now present the forward move algorithm. The algorithm has an initialization step that causes it to consume at least one symbol of the input, followed by repeated parse steps.

```
algorithm Forward_move (FMA)
input R - the remaining input
output the forward context developed
   and the input not consumed
Init:
   let Z be the stack consisting only of ?
   Push {q' | q --First R--> q', q ε ?} on Z
   R ← Rest R
repeat
   let h = First R, Q = Top Z,
      and MOVES = ⋃ PD(q,h)
                  q ε Q
   select MOVES:
      case {read}:
         Push {q' | q --h--> q' and q ε Q} on Z
         R ← Rest R
      case {A → w}: # Reduce, if possible:
         if |Z| > |w| then
            Pop |w| state sets off of Z
            Push {q' | q --A--> q' and q ε Top Z} on Z
         else return (Spelling Z,R) fi
            # w does not reside on the stack
      case {}          # We hit an error
         or {accept}   # R is ⊥
         or otherwise: # |PD| > 1
            return (Spelling Z,R)
   end repeat
end FMA
```

Notice that the repeated parse steps of FMA are identical to those that the parser normally follows, save the "otherwise" case, the manipulation of sets of states instead of states, and the check, just prior to a reduction, that the entire right part resides on the stack.

FMA essentially follows all paths that allow the parsing of the input text. It halts in case "otherwise" when two different paths end up in states that disagree as to how to continue the parse, or in case {A → w} when all paths end up in states requiring a reduction over the ?, or in case {accept} when we read the entire input, or in case {} when we encounter another error, i.e. no path can be continued. The set MOVES computed by FMA represents all the possible ways that the states in the top state set Q wish to treat the

input symbol h. Note that states q ε Q that cannot accept h (i.e. for which PD(q,h) = {}) have no effect on the parsing decision unless all states in Q cannot accept h (case {}); we extend each path as far as we can, even though other paths terminate.

We illustrate the halts of cases {A → w} and "otherwise" by Examples 1 and 2 below, where the parser of concern is that of Figure 1.

Example 1. Let the erroneous input string be i(i)⊥. The parser stops with state stack [i]. The following displays the execution of FMA on the remainder of the input.

FMA step just made	Stack after FMA step	Rest of input
Init	? $\{(_0\}$	i) ⊥
{read}	? $\{(_0\}$ $\{i_0\}$	) ⊥
{P → i}	? $\{(_0\}$ $\{P_0\}$	) ⊥
{T → P}	? $\{(_0\}$ $\{T_0\}$	) ⊥
{E → T}	? $\{(_0\}$ $\{E_1\}$	) ⊥
{read}	? $\{(_0\}$ $\{E_1\}$ $\{)_0\}$	⊥
{P → (E)}	? $\{P_0\}$	⊥
{T → P}	? $\{T_0,T_1,T_2\}$	⊥

The algorithm halts here because $PD(T_0,\perp) \cup PD(T_1,\perp) \cup PD(T_2,\perp)$ = {E → E + T, T → P ** T, E → T}. Of course, the expression between the parentheses could have been arbitrarily long with the same result.

Example 2. Input is ()⊥. The parser halts with state stack [(].

FMA step	Stack	Rest
Init	? $\{)_0\}$	⊥

Halt: $PD()_0,\perp)$ = {P → (E)}, and there are less than three items on the stack above the ?.

In Example 1, we face the possibilities of reducing by three different productions. $E \rightarrow T$ is the proper reduction only if what immediately precedes the T is a "(" or nothing; $E \rightarrow E+T$ is the proper reduction only if what immediately precedes the T is "E+"; and $T \rightarrow P**T$ is correct only if "P**" precedes the T; but no context exists to the left of $\{T_0, T_1, T_2\}$. Thus, we cannot continue parsing without making a guess, and must halt. In effect, the three different situations in the parser in which it can read a T yield three different decisions as to what to do with the T.

In Example 2, we attempt to reduce with $P \rightarrow (E)$, but find that "(E" does not precede ")" on the stack. The attempted reduction gives us an indication of what the user intended, however, and may provide useful information for an error repair strategy called "stack forcing," as we explain in the next section.

The initialization step Init of FMA guarantees that the algorithm produces a forward context of length at least one. If we did not cause FMA to read the first symbol, then it would consider all reductions that have the first symbol in their look-ahead sets; possible choices between a read and some reductions might have caused FMA to halt immediately in case "otherwise," making no progress whatsoever. (We assume also for the remainder of this paper that we never invoke FMA on the input consisting only of $\perp$, otherwise we would immediately read $\perp$ in step Init.)

In Section 5 we precompute the state sets of FMA as states. This allows us to extend the concepts of transitions and paths to FMA's state sets. Hence, if FMA consumes text u from string uv and produces forward context U, we may write $FMA:(?,uv) \vdash^{*} ([?:U],v)$. The relation $\vdash$ that can be deduced from FMA is exactly the same as that of the LR parsing algorithm, but to prevent confusion between the LR parsing algorithm and FMA, we prefix moves of FMA by "FMA:", as above.

3. FORMAL PROPERTIES OF FMA

Suppose FMA:(?,uv) $\vdash^*$ ([?:U],V). U satisfies important properties that we explore in this section. Essentially, U is such that during a parse of any sentence ending in uv, u must be reduced to U. We formalize and indicate the significance of this property in this section. To do so, we define some new terminology.

A valid prefix is any prefix of yw, where $S \rightarrow^* yAv \rightarrow ywv$ for some $y \in V^*$, $A \rightarrow w \in P$, and $v \in T^*$. The string spelled by the stack at any point during LR parsing is a valid prefix. A valid fragment is a suffix of a valid prefix; i.e. valid fragments are suffixes of the strings spelled by the parser's stack. For example, for the grammar of Figure 1, $E \rightarrow^* E+P**i \rightarrow E+(E)**i$ so any prefix of E+(E) is a valid prefix, e.g. E+(, and any suffix of E+(is a valid fragment, e.g. +(. We now define the concept central to this paper.

Definition 1. $U \in V^*$ is a derived valid fragment (DVF) of sentence suffix x iff

(1) $U \rightarrow^* u$ and x = uv for some u, $v \in T^*$, and

(2) for every valid prefix y such that

$$([y],uv) \vdash_{read} \ldots \vdash \text{accept},$$

$$([y],uv) \vdash^* ([yU],v).$$

Thus, during a parse of any sentence ending in uv, at some point the parser must reduce u to the valid fragment U. (The requirement that the first $\vdash$ is $\vdash_{read}$ relates to the fact that FMA reads as its first move.)

In the context of error recovery, this concept has the following significance: Suppose the parser encounters an error and halts in configuration (Z,uv) with uv a suffix of a sentence, and that an error repair algorithm

suggests [y'] as a possible replacement for Z. We could verify that ([y'],uv) $\vdash^*$ accept by actually trying the parse, but if many such [y']s were to be tested, reparsing uv each time would be costly. The significance of having some DVF U of uv is that in U we have a "partially parsed" version of u and need not repeat this partial parse, for the DVF property states that u must be reduced to U no matter what the string to the left of uv.

A necessary (not sufficient) condition that ([y'],uv) $\vdash^*$ accept is as follows: Let y be such that ([y'],uv) $\vdash^*$ ([y],uv) and PD(Top [y],First uv) = {read}; then it must be the case that ([y],uv) $\vdash^*$ ([yU],v). This is by definition of a DVF and due to the fact that ([y'],uv) $\vdash^*$ accept only if ([y],uv) $\vdash^*$ accept. For any given [y'], then, this requires only that we compute [y] and determine whether a path exists from Top [y] spelling U. Determining the existence of the path [yU] is considerably cheaper than reparsing u if u is much longer than U, and gives us an inexpensive test to determine if the proposed stack repair [y'] is "good enough" to cause the parser to consume u.

For future convenience we present the algorithm Consume_DVF that performs the computation just described.

```
algorithm Consume_DVF
input [q:y] and U -- a path and a forward context.
output a boolean value -- indicating
   whether [q:y] can consume U --
   and giving either the successfully
   computed path or an error message.

# First, do reductions triggered by First U.
while ([q:y],U) |-_M (Z,U)
   for M ε P (the productions)
   and for some path Z
   do y ← Spelling Z od # i.e. reduce.
```

(cont.)

```
# After all possible reductions have been made, the
# the next parsing decision must be a read.
let MOVES = PD(Top [q:y], First U)
if MOVES ≠ {read} then return False giving [q:y] fi

# Now we must be able to find a path
if path [q:yU] exists then return True giving [q:yU]
else return False giving "path ended in error" fi
end Consume_DVF
```

Note that the effect is the same when the computation of [q:y] is based on First U rather than First u, namely, if path [q:yU] exists both methods produce the same result. However, if [q:yU] does not exist, using First U rather than First u may cause the situation to be detected earlier: MOVES might not be equal to {read}. We get this "earlier detection" capability because First U may be a nonterminal representing not only First u but also some text to its right, and hence First u may be a member of look-ahead sets of which First U is not a member. Parsers not having nonterminals in look-ahead sets must retain First u for any DVF U in order to use Consume_DVF. Note further that Consume_DVF takes any path as its first argument, rather than just paths beginning at START; this is because we eventually intend to use it with paths produced by FMA also.

Our main result is: For some sentence suffix uv, if FMA:(?,uv) $\vdash^{*}$ ([?:U],v), then U is a DVF of uv. This results, intuitively, from the fact that FMA parses uv with no assumptions about left context. Thus u must be reduced to U no matter what the left context of u.

But first we consider a generalization of the DVF concept, and prove our results in terms of it.

Definition 2. $U \in V^{*}$ is a restricted DVF (RDVF) of sentence suffix x with respect to $RQ \subseteq K$ iff

(1) $U \rightarrow^{*} u$, $x = uv$ for some $u, v \in T^{*}$, and

(2) for every valid prefix y accessing some $q \in RQ$

such that $([y],uv) \vdash_{read} \ldots \vdash \underline{accept}$,

$([y],uv) \vdash^{*} ([yU],v)$.

Note that a DVF is an RDVF with respect to ? = K.

To adapt FMA to the RDVF concept, we allow it to begin parsing with any $RQ \subseteq K$, not just ? = K. Thus we may write $FMA:(RQ,uv) \vdash^{*} ([RQ:U],v)$. This requires only that the Init step of FMA be altered to use RQ instead of ?. We shall prove that if $FMA:(RQ,uv) \vdash^{*} ([RQ:U],v)$, then U is an RDVF of uv with respect to RQ. Letting RQ = ? gives us our main result as a corollary to Theorem 1.

The reason for the RDVF concept is that sometimes we will want to apply FMA to x in a situation in which we do know something about the context to the left of x. In general, starting FMA with some restricted context RQ allows it to both get farther in the input text (there will be fewer inadequacies), and have better error detection capabilities. In essence, RQ defines the possible contexts in which to parse x; if RQ = ? then we essentially put no restriction on the left contexts.

First, we prove two lemmas concerning FMA. These demonstrate how FMA keeps track in parallel of paths that would be computed by the parser.

Lemma 1. Let $FMA:(RQ,uv) \vdash^{*} ([RQ:U],v) = (RQ\ Q_1 \ldots Q_m, v)$. For any path [yU] such that y accesses some $p \in RQ$, $[p:U] = p\ q_1 \ldots q_m$ and $q_i \in Q_i$, $1 \leq i \leq m$.

Proof. By induction on m. Let $U = a_1 \ldots a_m$. For m = 1: $p \in RQ$, hence by step Init (for $a_1 \in T$) or case $\{A \rightarrow w\}$ (for $a_1 \in N$) of FMA, $q_1 = SIGMA(p,a_1)$, $q_1 \in Q_1$, and $Q_1 = \{q' \mid q \xrightarrow{a_1} q' \text{ and } q \in RQ\}$. Now assume true for m = k; thus $[RQ:a_1 \ldots a_k] = RQ\ Q_1 \ldots Q_k$, $[p:a_1 \ldots a_k] = p\ q_1 \ldots q_k$ and

$q_i \in Q_i$, $1 \le i \le k$. By case {read} or case {$A \to w$} of FMA, q_{k+1} = SIGMA(q_k,a_{k+1}), $q_{k+1} \in Q_{k+1}$, and $Q_{k+1} = \{q' \mid q \xrightarrow{a_{k+1}} q' \text{ and } q \in Q_k\}$.

Lemma 2. Suppose FMA:(RQ,uv) $|\overline{}_{M_1} \ldots |\overline{}_{M_r}$ ([RQ:U],v) = (RQ $Q_1 \ldots Q_m$,v). If, for some y and y', ([y],uv) $|\overline{}_{M_1} \ldots |\overline{}_{M_r}$ ([y'],v) and Top [y] $\in$ RQ, then [y'] = [y] $q_1 \ldots q_m$, where $q_i \in Q_i$, $1 \le i \le m$.

Proof. Given that the parser and FMA make the same moves $M_1 \ldots M_r$, they stack the same symbols. Thus y' = yU, and the result follows from Lemma 1.

Theorem 1. For some sentence suffix uv, if FMA:(RQ,uv) $|\overline{}_{M_1} \ldots |\overline{}_{M_r}$ ([RQ:U],v), then U is an RDVF of uv with respect to RQ.

Proof. Let y be a valid prefix such that y accesses some $q \in RQ$, ([y],uv) $|\overline{}_{M'_1} \ldots |\overline{}_{M'_{r'}}$ accept and M'_1 = read. By induction we show property P(r) to hold, where P(k) is defined for $k \le r$ as

$r' \ge k$ and $M'_i = M_i$ for $1 \le i \le k$.

If P(r) holds, then ([y],uv) $|\overline{}_{M_1} \ldots |\overline{}_{M_r}$ ([yU],v), the desired conclusion.

For r = 1: M_1 = read = M'_1 by step Init of FMA. Let P(k) hold. By Lemma 2, FMA's stack after move M_k is

RQ Q_1 Q_2 ... Q_m = [RQ:U']

and the parser's stack after move $M'_k = M_k$ is

[y] q_1 q_2 ... q_m = [yU']

for some m, where $q_i \in Q_i$, $1 \le i \le m$. Let the next input symbol be h (h is in u or is First v). We prove by contradiction P(k+1), i.e. that in addition, $r' \ge k+1$ and $M_{k+1} = M'_{k+1}$.

(1) Assume $r' < k+1$; by the induction hypothesis, $k \le r'$, so this forces k = r', i.e. the parser's last move was $M_{r'}$. Then ([yU'],$\perp$) = accept, so that m = 1, [y] = [], and U' = S' (recall production $S \to S'\perp$). Since there is a unique state in the parser having

accessing symbol S', we have $\{q_1\} = Q_1$ and $PD(q_1, \perp) = \{accept\}$, so that for FMA, MOVES = {accept}. Thus FMA cannot make move M_{k+1}. Hence by contradiction, $r' \geq k+1$.

(2) Assume $r' \geq k+1$, but $M'_{k+1} \neq M_{k+1}$. Now $\{M'_{k+1}\} = PD(q_m, h)$. But since $q_m \in Q_m$, $\bigcup_{q \in Q_m} PD(q,h)$ would contain both M_{k+1} and M'_{k+1}. By case "otherwise" of FMA, FMA would not make move M_{k+1}. Hence $M'_{k+1} = M_{k+1}$.

Hence P(k+1) holds when P(k) holds, hence P(r) and our conclusion.

Corollary. If we let RQ = ?, we have immediately that U is a DVF of uv.

We claimed in section 2 that FMA parses as much as it can until it must refer to nonexistent left context. We formalize this intuition below.

Definition 3. $U \in V^*$ is the maximal RDVF (MRDVF) of sentence suffix x with respect to $RQ \subseteq K$ iff the following three conditions imply that $([yU'],v') \vdash^* ([yU],v)$:

(1) U is an RDVF of x with respect to RQ where $U \rightarrow^* u$ and $x = uv$ for some $u, v \in T^*$,

(2) U' is any other RDVF of x with respect to RQ where $U' \rightarrow^* u'$ and $x = u'v'$ for some $u, v \in T^*$,

(3) there exists valid prefix y such that y accesses some $q \in RQ$ and

$$([y],uv) \vdash_{read} \ldots \vdash \underline{accept}.$$

Thus, by the definition of DVF's, $([y],uv) \vdash^* ([yU'],v') \vdash^* ([yU],v)$, so that an MRDVF U is "as far up" the derivation tree of yuv as possible. v must be a suffix of v', so that $|u| \geq |u'|$, i.e. U derives the longest possible prefix of x. If v = v' then we see that U is reduced as much as possible, since then $U \rightarrow^* U'$. An algorithm that produces the MRDVF would therefore read as far as it could into the input, and reduce as much as it

could. It is clear from the definition that the MRDVF is unique.

FMA, when started in state RQ, does not always compute the MRDVF. This is because FMA is restricted to using the same parsing technique as the parser and therefore to the same finite look-ahead. An algorithm superior to FMA might scan all of x, perhaps discovering some contextually significant symbol located towards the end of x that could help it parse earlier text. An FMA based on an SLR machine might be bested by one based on an LALR machine for the same grammar. But given the limitation of the base parser, FMA does as best it can. These restrictions are encoded in the following theorem that formalizes FMA's performance in terms of its base parser.

Theorem 2. Consider suffix uv of a sentence. If there exist $RQ \subseteq K$, integer $r \geq 1$ and a sequence of moves $M_1 \ldots M_r$ such that

(i) M_1 = read,

(ii) there exists some state $q \in RQ$ and $U \in V^*$ such that

$$(q,uv) \mid\!\overline{}_{M_1} \ldots \mid\!\overline{}_{M_r} ([q:U],v), \text{ and}$$

(iii) there exists no valid prefix y, integer $k < r$,

and configurations C and C' such that

y accesses some $q' \in RQ$,

$$([y],uv) \mid\!\overline{}_{M_1} \ldots \mid\!\overline{}_{M_k} C \mid\!\overline{}_{M'_{k+1}} C',$$

and $M_{k+1} \neq M'_{k+1}$,

then $\text{FMA}:(RQ,uv) \mid\!\overline{}_{M_1} \ldots \mid\!\overline{}_{M_r} ([RQ:U],v)$.

Proof. By induction. We prove P(r), where P(k) is defined for $k \leq r$ as

$$\text{FMA}:(RQ,uv) \mid\!\overline{}_{M_1} \ldots \mid\!\overline{}_{M_k} C''$$

for some configuration C''. P(1) holds since M_1 = read and FMA reads as its first move, by step Init of FMA. Let P(k) hold; we show that P(k+1) holds. Let q be as in hypothesis (ii). If P(k), then by Lemma 2 we have

$$\text{FMA:}(RQ,uv) \;|\overline{}_{M_1} \; \ldots \; |\overline{}_{M_k} \; (RQ\ Q_1 \ \ldots \ Q_m,\ R) \quad (= C'')$$

and

$$(q,\ uv) \;|\overline{}_{M_1} \; \ldots \; |\overline{}_{M_k} \; (q\ \ q_1 \ \ldots \ q_m,\ R)$$

for some m and R, where $q_i \in Q_i$, $1 \leq i \leq m$. The parser's next move is M_{k+1} from $(q\ q_1 \ \ldots \ q_m,R)$; we show by contradiction that FMA makes move M_{k+1} from $(RQ\ Q_1 \ \ldots \ Q_m,R)$. Assume instead that FMA does not make move M_{k+1}. Consider the possible ways in which this can happen. Let MOVES = $\bigcup_{q \in Q_m}$ PD(q,First R):

(1) MOVES = $\{M_{k+1}\}$ = $\{A \rightarrow w\}$, but FMA cannot make move $A \rightarrow w$ because $m < |w|$. Then neither can the parser make move $A \rightarrow w$ by the definition of $|\!-$; $m < |w|$ implies that there exists no y' such that $q\ q_1 \ \ldots \ q_m = [q{:}y'w]$.

(2) MOVES = $\{M'_{k+1}\}$ and $M'_{k+1} \neq M_{k+1}$. But since $q_m \in Q_m$, $M_{k+1} \in$ MOVES, so that MOVES $\neq \{M'_{k+1}\}$.

(3) MOVES = {}. But again since $q_m \in Q_m$, $M_{k+1} \in$ MOVES.

(4) |MOVES| > 1. Then let $\{M'_{k+1},\ M_{k+1}\} \subseteq$ MOVES. Let $q\ q_1 \ \ldots \ q_m = [q{:}y']$. There must exist some $q' \in RQ$, $q' \neq q$, such that $\{M'_{k+1}\}$ = PD(Top[q':y],First R). Let y access q'. Then

$$([y],uv) \;|\overline{}_{M_1} \; \ldots \; |\overline{}_{M_k} \; ([yy'],R) \;|\overline{}_{M'_{k+1}} \; C'$$

for some C'. But this contradicts hypothesis (iii).

We have shown all possibilities contradictory; hence P(k+1) and thus P(r):

$$\text{FMA:}(RQ,uv) \;|\overline{}_{M_1} \; \ldots \; |\overline{}_{M_r} \; C''.$$

for some C''. But since $(q,uv) \;|\overline{}_{M_1} \; \ldots \; |\overline{}_{M_r} \; ([q{:}U],v)$, C'' = ([RQ:U],v).

Corollary. FMA applied to a sentence suffix makes the greatest number r of moves possible, where r is as defined in Theorem 2.

Proof. Merely let the q of hypothesis (ii) be such that r is maximized.

Apart from the significance of DVF's in validating error repairs, DVF's satisfy other useful properties. The "next move" property is helpful in selecting error repairs. Let uv be a sentence suffix, and U be the DVF of uv returned by FMA (when started with ?). When FMA halts, the value of MOVES is such that

$$\text{MOVES} = \bigcup_{\substack{y \in V^* \\ [yU] \text{ a valid prefix}}} \{M \mid ([yU],v) \vdash_M C \text{ for some } C\}$$

In other words, MOVES contains all the moves that the parser may make from some configuration ([yU],v). Intuitively, since FMA parses without knowing y, at each step MOVES represents the set of moves for all possible y's.

The utility of the next move property is illustrated as follows. For the Algol example of the previous section, MOVES would contain the two moves indicating "reduce by production (1)" and "reduce by production (2)", where U = "do Stmt". The next move property says that if we find some y such that the parser makes move M from configuration ([yU],v), then M must be one of those two reductions. Either reduction puts a constraint on y; it must end in either "for Id := Exp step Exp until Exp" or "while Exp". We may thus sometimes use the elements of MOVES to guide us in the selection of y's. We call these reductions "long reductions" because if performed during the forward move, they would attempt to pop the ? state set (in the Algol example FMA halted in case "otherwise"). Such long reductions can sometimes provide "instant solutions" to some errors. In this example, a comparison of the stack with the set MOVES shows that we should patch up the stack by inserting the missing Exp and continue. In practice, we may simply search the stack preceding the point of error detection for some state that can read the left part of either production (we call this technique "stack forcing").

We mention this again in the next section. MOVES may contain elements that are not long reductions, such as "read" or a "short reduction," but we do not yet know how best to make use of this information. We formalize the next move property as follows:

Theorem 3. Let x be a sentence suffix, RQ $\subseteq$ K, U an RDVF of x with respect to RQ with U $\rightarrow^*$ u and x = uv, and

$$\text{MOVES} = \bigcup_{q \in \text{Top } [RQ:U]} PD(q, \text{First } v).$$

Then we have the following:

$$\text{Let } X = \bigcup_{\substack{y \in V^* \\ [yU] \text{ a valid prefix} \\ \text{Top } [y] \in RQ}} \{M \mid ([yU],v) \vdash_M C \text{ for some } C\}.$$

Then MOVES = X.

Proof. Let M $\in$ MOVES. By the definition of MOVES, there exists some q $\in$ TOP [RQ:U] such that M $\in$ PD(q,First v), and hence some q' $\in$ RQ such that Top [q':U] = q. Let y access q'; then M $\in$ PD(Top[yU],First v) so that there exists C such that ([yU],v) $\vdash_M$ C. Hence MOVES $\subseteq$ X.

Consider now M $\in$ X; corresponding to M there is a valid prefix yU such that M $\in$ PD(Top [yU],First v) and Top [y] $\in$ RQ. But Top [yU] $\in$ Top [RQ:U] by Lemma 1, so that by the definition of MOVES, M $\in$ MOVES. Hence X $\subseteq$ MOVES, and our conclusion.

Corollary. If RQ = ?, then |MOVES| $\geq$ 1.

Proof. Since x is a sentence suffix, there exists some valid prefix y such that S $\rightarrow^*$yx; thus ([y],x) $\vdash^*$ accept. Without loss of generality let read $\in$ PD(Top [y],x). Then by the RDVF definition, ([y],x) $\vdash^*$ ([yU],v). Let C be such that ([yU],v) $\vdash_M$ C for some M (there must be at least one since ([y],x) $\vdash^*$ accept); M $\in$ MOVES. Hence |MOVES| $\geq$ 1. (For a similar result where RQ = ? see [D&R 77].)

We can further use the next move property to help pinpoint errors. If FMA(?,uv) $\vdash^{*}$ ([?:U],v) but halts in case {}, i.e. MOVES = {}, then uv is not a sentence suffix. That is, an error has occurred somewhere in the text uv, because there exists no y such that $S \rightarrow^{*} yuv$. More specifically, since we are dealing with LR(1) parsers, the error has occurred in the "window" comprised of the first $|u|+1$ symbols of uv.

In summary, we have shown that FMA (1) provides an inexpensive test for stack replacements, (2) sometimes points us directly to the repair we need to continue the parse, and (3) sometimes finds a "window" within which an error has occurred. We do not know how, in the general case, to come up with stack replacements. In a more specialized case in which we assume some knowledge of the types of errors, we have a chance of designing stack replacements.

4. REPAIR STRATEGIES USING FMA

Given FMA and its formal properties, we now proceed to develop an algorithm that finds a useable configuration in which to restart the parser. In our initial analysis we make the "simple error assumption" (SEA), viz. the non-sentence z in question resulted from a sentence via a single "mutilation": an insertion, a replacement, or a deletion of a single terminal symbol.

Insertion: $z = ytx$ and $S \rightarrow^* yx$ but not $S \rightarrow^* ytx$

Replacement: $z = ytx$ and $S \rightarrow^* yt'x$ but not $S \rightarrow^* ytx$

Deletion: $z = yx$ and $S \rightarrow^* ytx$ but not $S \rightarrow^* yx$

In the next few paragraphs we assume an LR(k), as opposed to SLR(k) or LALR(k), parser and we even assume that the parser detects the error at the point of mutilation. Then we generalize gradually and discuss the consequences.

Suppose the parser detects an error in configuration (Z,tx). Thus, t is an unexpected symbol in the left context spelled by Z. Suppose further that we have reason to believe that an insertion of t occurred. How could we confirm that suspicion? A straight-forward way is simply to determine if $(Z,x) \vdash^* \underline{\text{accept}}$; i.e. delete t and resume parsing. Similarly, if we thought the mutilation was the replacement of some terminal t' by t, we must resume with (Z,t'x), and if the deletion of some t' just prior to t, then (Z,t'tx).

Now, in the error recovery context we have no clue as to which of the above repairs may work, so we must try them all. Furthermore, if none of them work, we can conclude for an LR(k) parser and under SEA that the mutilation occurred left of the point of error detection, i.e. the parser somehow

incorporated the mutilation on its stack. In the case of an SLR(k) or LALR(k) parser, even if the correct "unmutilation" is found, the above trials may not work since the parser may have made reductions (by looking ahead at or ignoring the unexpected symbol) that the corresponding LR(k) parser would not have made. Repairs in these cases will involve some form of backing up the parser. But before considering those implications, let us consider the use of FMA to reduce the cost of trial parses.

To limit the repeated parsing of x we apply FMA to x recursively until it has been reduced to a sequence of DVF's U_1, ..., U_n. We call this process FMA+, which can be defined as follows:

FMA+(x) =

if x = $\perp$ then $\perp$ else

U such that FMA:(?,x) $\vdash^*$ ([?:U],v)

followed by FMA+(v)

Furthermore, before trying the insertions, not having found a deletion or replacement that will work, we should apply FMA+ to tx, thus producing some U'_1, ..., U'_m for some $m \leq n+1$. Let u_i be the terminal string that was reduced to U_i by applying FMA+ to x. We have $m = n+1$ and $U'_{i+1} = U_i$ for $1 \leq i \leq n$ if the first application of FMA to tx parses t and then halts; $m = n$ and $U'_i = U_i$ for $1 \leq i \leq n$ if FMA parses t and all of u_1 but then halts; $m = n-1$ and $U'_{i-1} = U_i$ for $2 \leq i \leq n$ if t, u_1, and u_2 are similarly combined by FMA. Taking advantage of the fact that u_1, ..., u_n have been reduced to U_1, ..., U_n by the previous application of FMA+ to x, we can avoid applying FMA+ to tx by instead using the following algorithm to "attach" t to the "extended forward context" U_1 ... U_n, producing the same result.

algorithm Attach
input h, C-- the symbol to be attached and
the sequence of DVFs to attach it to. (cont.)

```
output a Boolean value, and giving
   the resulting sequence of DVFs.
let P be a path such that Consume_DVF(?,h) gives P
while C is not null do
   let P' be a path variable
   if Consume_DVF(P, First C) gives P'
      then P ← P'; C ← Rest C
      elseif P' = "path ends in error"
      then return False
         # not giving anything; irrelevant.
      else return True
           giving  Augment(Spelling P, C)
      fi
   od
return True giving Spelling P
end Attach
```

In the above we have assumed the operation, Augment, on sequences that provides a sequence of length n+1 by adding a new element, the left operand, to the front (left) of a sequence of length n, the right operand.

Non-immediate detection. Suppose that none of the deletions, replacements, or insertions succeed. An easy way to proceed next is to start backing down the stack, one symbol at a time, trying deletions and replacements of each symbol h, then if none of these succeed, attaching h to the previous extended forward context and trying insertions in front of h, just as we did for the unexpected symbol, which has now been "exonerated" (for LR(k) parsers, at least). We summarize this entire strategy as follows.

```
algorithm Error_recovery
input (Z,R)--the erroneous configuration.
output the repaired configuration.
let h and h' = First R, Z' = Z,
   EFC and EFC' = FMA+(Rest R)
while Z' is not empty do
   # Try deletion, replacements,
   # attachment, then insertions.
   let C be a configuration variable
   if Try (Z', {Λ}, EFC) gives C then return C fi
   if Try (Z', T, EFC) gives C then return C fi
   if not Attach (h,EFC) gives EFC then
      exit fi # implies a "window"
   if Try (Z', T, EFC) gives C then return C fi
   h ← Accessing_symbol (Top Z'); Pop Z'                    (cont.)
```

```
   od
if Try_stack_forcing (Z,V,h',EFC') gives C
   then return C fi
return (S',⌊)  # i.e. give up by
               # returning accept.
end Error_recovery
```

Note that we return from Error_recovery, when we Try a repair that succeeds, with the repaired configuration C. However, if an error is detected by Attach, we exit from the while loop, having isolated the mutilation to within a "window" comprising the text from the leftmost token of the phrase associated with the symbol h up to the original unexpected symbol, inclusive. (A message should be printed to this effect.) Perhaps in this case we should just delete all the text in the window and call Error_recovery recursively. Instead we have indicated above to Try_stack_forcing as suggested in Section 3. At this point further investigation and development of the overall algorithm is needed.

Backing down the stack is, of course, an attempt at repairing damage caused by the mutilation. In some cases the mutilation will not have affected the phrases around it. A replaced symbol, for example, may still be on the stack, unreduced or simply reduced. Then a deletion, replacement, or insertion of a single symbol may precisely undo the mutilation. In fact, to increase the likelihood of a useful repair we have found it worthwhile to Try nonterminals as well as terminals, i.e. TRY(Z',V,EFC), as replacements and insertions. This pays off when, for example, a mutilation has affected only one phrase.

On the other hand, a mutilation only belatedly detected can have caused an arbitrarily large amount of "damage" to occur on the stack, in the sense that many reductions may have occurred that would not have on the unmutilated string. For example, inserting a semicolon before an operator in the right

part of an assignment statement, e.g. "X = X + Y; * Z;", typically results in the text to the left of the semicolon being reduced to a statement; but then we are left with an expression fragment to the right of the semicolon. Ideally, in such cases we would like to partially "unparse" some symbol(s) on the stack, then Try the repairs. Another example is the PL/I conditional statement, which when the if is deleted, may look like an assignment statement up to the then: "...; X = Y + Z then ... else ... ;". Here the unwanted reduction of X = Y + Z to statement might occur with a LALR(1) or SLR(1) parser but not with an LR(1) parser.

We are still investigating possible approaches to recovering from such potentially massive damage, approaches about which something formal can be said; and we are looking for grammatical restrictions that might limit such damage. However, since that research is incomplete, we refrain from discussing the ideas here, other than to note that "stack forcing" mentioned in Section 3 above appears to have good development potential. Ultimately, any scheme used must have a significantly greater potential for facilitating "upper level" parsing after making a repair (FMA will have done the "lower level" parsing) than it has potential for causing an avalanche of spurious error messages, e.g. if the repair discards several left bracket symbols.

Finally, note that we may give up by telling the parser it is done, i.e. by returning to it the accepting configuration. This is not unreasonable since we have already partially parsed the input beyond the point of error detection and we are only giving up the opportunity to parse the remaining upper level, thus losing the opportunity to detect some other errors. Now in practice we do not parse all of the remaining input, but rather we stop FMA+ at a convenient point after it has produced at least seven, say, symbols of

extended forward context. We consider a trial successful, then, if after the repair, all of the extended forward context can be parsed; then we return to the parser the resulting stack and the remaining input. For expositional purposes, however, we continue the presentation in terms of the simpler, if impractical, approach of partial parsing to the end of the program. The practical modifications are not difficult to make, but they would obscure the presentation.

```
algorithm Try
input Z, V, EFC -- a state stack,
   vocabulary, and a sequence of DVF's.
output a Boolean value, giving a configuration.
for s ε V do
   let Z' = Z
   let EFC' = if s = Λ then EFC
              else Augment(s,EFC) fi
   while EFC' is not empty and
      Consume_DVF(Z', First EFC') gives Z'
      do EFC' ← Rest EFC' od
   if EFC' is empty then
      return True giving (Z',|) fi
   od
return False # giving nothing; irrelevant.
end Try
```

Pragmatics. The algorithms presented above are idealistic in several ways. We have already mentioned limiting FMA+ rather than allowing it to proceed to the end of the program being parsed. Now we consider the possibility of several mutilations to the program. In this case FMA+ may end in error before accumulating the desired number of symbols; thus we simply accept the first repair that successfully reaches the subsequent error detection point.

However, it may happen that FMA+ will not detect a subsequent error, due to its lack of "upper level" parsing. Any such error will have to be detected after a repair is made. For example, suppose "...; X < Y + Z then then ... else ...;" was created by deleting an if and inserting a then, and

suppose one relevant production is: If_clause → if Bexp then. An error is detected at the "<", "Y" is skipped, and FMA+ is invoked. After the first then the parser may be ready to reduce without needing to look ahead; but no reduction can be made due to incomplete parsing to the left. So FMA+ calls FMA again, starting at the second then, and the inserted then is not reported as an error by FMA+. Again the solution of choice is to take the repair that results in getting the farthest into the extended forward context before detecting a subsequent error.

Semantics. Given this error recovery scheme it is unlikely to be worth trying to continue to drive "semantic routines" that perform static semantic analysis and/or code generation, even after the first error is encountered, since parsing proceeds in a non-canonical order while recovering. On the other hand, in a compiler whose parser builds an abstract-syntax tree, which is to be traversed subsequently to parsing for further analysis and code generation, we may continue tree-building during FMA+; and simple repairs will lead to knitting the subtrees together appropriately for subsequent processing. Of course, gross repairs will result in a mangled tree, but that in turn just presents an error detection and recovery problem to the subsequent processor. Presumably, if a formal technique such as an "affix grammar" [Kos 71] is adapted to describe the static semantics of languages based on abstract-syntax trees, then automatic techniques can be used to perform the analysis (see, e.g., [Wat 75] and [DeR 77]), and thus our recovery algorithm may prove useful there, too.

Improving FMA+. Having nonterminals in look-ahead sets allows us to construct an improved version of FMA+. When FMA is applied to a sentence suffix, it may halt by encountering an inadequacy (case "otherwise"); i.e.

the next (terminal) symbol is insufficient to resolve the parsing conflict. However, FMA+ immediately applies FMA to that symbol and what follows, resulting in a DVF that may begin with a nonterminal that is sufficient to resolve the conflict. This is because the nonterminal may represent an arbitrarily long look-ahead, i.e. the phrase that was reduced to it and perhaps one symbol beyond (due to the usual look-ahead). It behooves us then to review recursively the decision at the end of the prior DVF each time a new one is computed that begins with a nonterminal. This approaches the non-canonical parsing of the LR(k,t) style as suggested by Knuth [Knu 65].

```
algorithm Super_FMA+
input  x -- a sentence suffix.
output a sequence of DVFs derived from x.
if x = ⊥ then return ⊥
else
   let U, v be such that
      FMA:(?,x) |-* ([?:U],v)
   let S be an empty stack of paths
   Push [?:U] on S
   while v ≠ ⊥ do
      let U', v' be such that
         FMA:(?,v) |-* ([?:U'],v')
      v ← v'
      if First U' in N (the nonterminals) then
         while S is not empty do
            let Z be a path variable
            if Consume_DVF(Top S, U') gives Z
               then U' ← Spelling Z; Pop S
               else Push [?:U'] on S; exit
                  # exit from inner loop
               fi
            od
         fi
      od
   return the sequence of DVFs spelled by
      the paths on S, followed by ⊥
   fi
end Super_FMA+
```

Even this algorithm can be improved. Each time we "restart" FMA, at the beginning of the outermost while loop, we begin with "?", representing no knowledge of left context whatsoever. But we do know something about the

left context in this case, viz. the possibilities are restricted to those implied by the top state in the top path on S. Assume FMA has halted with Q on the top of its stack. Then instead of restarting with state set ?, we restart with state set $\bigcup_{q \in Q} RS(q)$, where

$$RS(q) = \{q' \mid S \rightarrow^* y'x \rightarrow^* yx,$$
$$y, y' \in V^*, x \in T^*,$$
$$y \text{ accesses } q, y' \text{ accesses } q'\}$$

The states from which First x can be read, after y is (optionally) reduced to some y', are in RS(q). The idea of using a restricted restart state has been suggested by Tai [Tai 77] for use in non-canonical SLR(1) parsing, although his restart states are different from ours and do not apply to LR(k) parsers in general.

We have implemented such restricted restart states and have observed that, while they do in fact improve error recovery, their expense in terms of the size of the parser + error recovery machine may be too great. For the statistics see the end of Chapter 5.

5. MAKING FMA PRACTICAL

In this section we re-describe FMA as an algorithm that manipulates not state sets but pre-computed states, thereby making it practical.

FMA computes state sets dynamically by referring to the parser's states -- see cases {read} and {A → w} of FMA. There is no reason why we cannot precompute these state sets and the transitions between them; this gives rise to a separate set of states for FMA.

We compute these states as follows. Let K be the set of parser states. The set K' of FMA states is computed by beginning with K' = {?}. Repeatedly add to K' the successors of state sets in K', where for s ε V, the s-successor of Q ε K' is $\{q' \mid q \xrightarrow{s} q' \text{ and } q \in Q\}$. We use ERSIGMA to mean the thus computed transition function for K'. Now we define the decision function PD(Q,h), where Q is a state in K' and h ε V, in terms of the states Q. Simply,

$$PD(Q,h) = \bigcup_{q \in Q} PD(q,h)$$

Observe that the computation of MOVES in FMA is just this same computation. Thus, algorithm FMA' below achieves the same effect as FMA.

```
algorithm FMA'
input  -- as in FMA
output -- as in FMA
Init:
   let Z be the stack consisting only of ?
   Push ERSIGMA(?,First R) on Z;   R ← Rest R
repeat
   let h = First R, Q = Top Z, and MOVES = PD(Q,h)
   select MOVES:
      case {read}:
         Push ERSIGMA(Q,h) on Z;  R ← Rest R
      case {A → w}:  # Reduce, if possible:
         if |Z| > |w| then
            Pop |w| states off of Z
            Push ERSIGMA(Top Z,A) on Z
         else return (Spelling Z,R) fi                    (cont.)
```

```
            case {}          # We hit an error
               or {accept}   # R is ⊥
               or otherwise: # |PD| > 1
                  return (Spelling Z,R)
         end repeat
      end FMA'
```

It should be evident that FMA and FMA' are equivalent. FMA' is as fast as the LR parsing algorithm save the check in case {A → w} that |w| states reside on the stack.

Now that FMA' manipulates states rather than state sets, we can suggest a space optimization. Suppose for some q ε K, {q} ε K' (this occurs often). If $q \xrightarrow{s} q'$ is a transition, then $\{q\} \xrightarrow{s} \{q'\}$ is also a transition. Once FMA' pushes a state {q} on its stack, and until it sometime later pops {q}, it will behave as if it had pushed state q on its stack. Thus we may "share" state {q} in K' with state q in K; states in K' having transitions into {q} can be modified to instead have the same transitions into q. Such sharing reduces the storage required for the parser/error recovery package.

The following state sharing criterion, satisfied by (but not only by) the singleton states in K', determines whether state sharing may occur: For any q ε K, Q ε K', Q may share with q iff for every y in V, if y spells a path from q to q' and a path from Q to Q' then PD(q',s) = PD(Q',s) for every s ε V. In other words, the parsing decisions that Q' and q' make must be the same. States in K' other than singleton sets satisfy this criterion. To see this, let t_0 = {A → t·} and t_1 = {A → t·, B → t·}, both members of K (A → t· and B → t· are final items, productions whose right part has been recognized; see [A&J 74] or [DeR 71]). Let $\{t_0,t_1\}$ ε K'. Note that $t_0 \cup t_1 = t_1$. Then if $PD(t_1,s) = PD(\{t_0,t_1\},s)$ for every s ε V, $\{t_0,t_1\}$ may be shared with t_1. This is the same as requiring that the look-ahead for production A → t in state t_0 be a subset of the look-ahead for production

$A \rightarrow t$ in state t_1. Non-singleton states that can be shared occur in practice, but they are non-trivial to determine. Singleton states are easy to find when generating K'. Figure 2 shows the state diagram for K' with singleton states shared with states in the state diagram of Figure 1.

Due to state sharing, the percentage of extra states needed over the original parser is only about 20-50%, and the percentage of extra transitions about 39%-78%, depending upon the grammar. For Pascal [Wir 73] we need increases of 48% and 78%, respectively, for XPL [MHW 70] 22% and 39%, and for PAL [A&U 72] 27% and 40%. With restricted restart states included, the percentages for PASCAL are 125% and 426%, for XPL 80% and 259%, and for PAL 74% and 384%.

The significant differences between our approach and that of Druseikis and Ripley [D&R 76,77] is that they compute the states K' via the LR(0) constructor algorithm, using actual sets of LR items (see [A&J 74]) and they do not show how to compute the look-ahead sets needed by FMA' for LALR(1) or LR(1) parsers. Our technique works equally well for SLR(1), LALR(1), or any other LR-style parser.

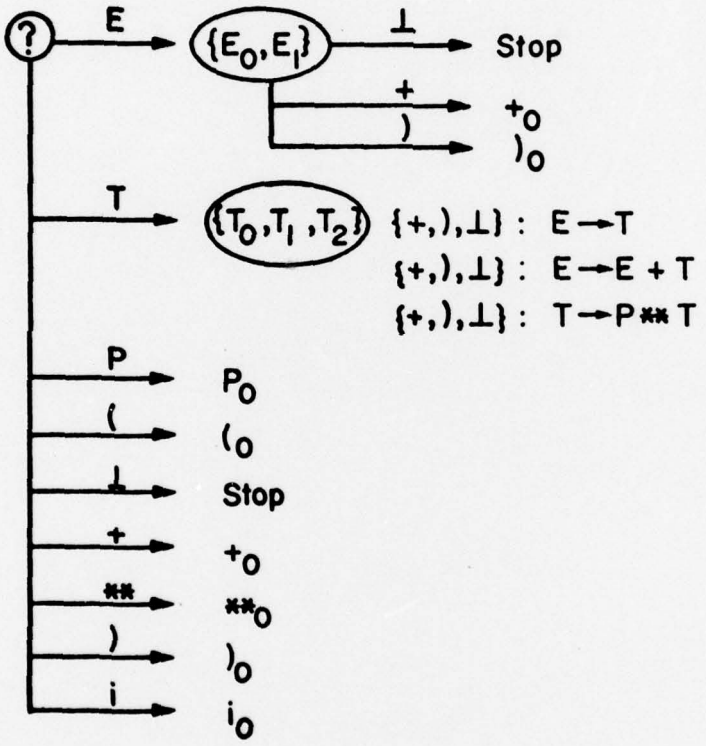


Fig. 2. State diagram representing transitions between states in K'. Singleton states in K' have been shared with the corresponding states in K. Reductions associated with ? have been omitted, since FMA' never considers them.

6. CONCLUSIONS

The proof of an error recovery algorithm is in its performance in a practical environment, quite apart from any nice theoretical properties it might have. Druseikis and Ripley [D&R 76] were kind enough to share with us a tape containing erroneous student Pascal programs. We ran our preliminary implementation on some of them, given a Pascal grammar deduced more-or-less mechanically from the Pascal syntax diagram [Wir 73].

Each repair selected by the algorithm was rated "excellent" if it repaired the text as a human reader would have, "good" if not but it still resulted in a reasonable program and no spurious errors, "poor" if it resulted in one or more spurious errors (in fact, none resulted in more than one spurious error), and "unrepaired" if no repair was selected but we continued to parse via FMA+ rather than the parser. The results follow:

Excellent	Good	Poor	Unrepaired	Total
32 (42%)	21 (28%)	9 (12%)	14 (18%)	76 (100%)

We have counted spurious errors in these statistics. Note that 70% were good or excellent. With some tuning, we hope to reduce the poor and unrepaired responses in number. The unrepaired cases both rob us of upper-level parsing and sometimes adversely affect recovery from other nearby errors. We have no idea how much different these statistics might be for more "sophisticated" errors made by seasoned system programmers intimately familiar with the language.

As another concrete demonstration of the algorithm's performance, we present in Figure 3 the erroneous Algol-like sample program used by Graham and Rhodes to illustrate the performance of their error recovery algorithm

[G&R 75]. We used the same Algol subset grammar as they, and our repairs are identical to theirs, without the need for a weighting for symbols or a pattern matching algorithm. We used the algorithms presented in Chapter 5 except that we modified them to prefer insertions to replacements and replacements to deletions. We rate each repair as "excellent," except the insertion of the <identifier> between * and / on line 5, which we rate "good" since we have no idea what the human might have done.

We should note that backing down the stack infrequently resulted in a good repair, except in the critical case of the deleted if in the program above. Thus, there is some question as to whether that technique is worth its computational cost; it should at least be delayed until no more-productive techniques have succeeded. Clearly more research, trials, and errors (no pun intended) are in order. As yet we have not implemented Super_FMA+.

```
1     begin
2     integer array A, B  (1..5  1..10);
3     integer I, J, K, L;
4 Up: I + J > K + L - 4 then go L1 else K Is 2;
5     A 1,2 := B ( 3 * ( I+J, J * / K)
6     if I = 1 then then go to Up;
7 L2: end


ERROR       Line 2, token 10, unexpected "1"
FOLLOWS     Blockhead Bounds_list Expression .. "5"
FORWARD     ? .. Expression )  ? ;  ? Declaration :  ? Label_Definition
REPAIR      "," was inserted after "5" and before "1".

ERROR       Line 4, token 4, unexpected "+"
FOLLOWS     Blockbody "I"
FORWARD     ? Primary  ? Relationop Expr  ? then  ? go    (ERROR)
REPAIR      "if" was inserted after Blockbody and before "I".

ERROR       Line 4, token 14, unexpected "L1"
FOLLOWS     Blockbody If_then_cl go
FORWARD     ? else  ? "K"    (ERROR)
REPAIR      "to" was inserted after "go" and before "L1".

ERROR       Line 4, token 17, unexpected "Is"
FOLLOWS     Blockbody If_then_cl Else_clause "K"
FORWARD     ? Primary  ? ;  ? "A"    (ERROR)
REPAIR      "Is" was replaced with ":=" after "K" and before "Primary".

ERROR       Line 5, token 2, unexpected "1"
FOLLOWS     Blockbody "A"
FORWARD     ? ,  ? "2"   (ERROR)
REPAIR      "(" was inserted after "A" and before "1".

ERROR       Line 5, token 5, unexpected ":="
FOLLOWS     Blockbody "A" ( Expression , "2"
FORWARD     ? "B"  ? (  ? Primary  ? * ( Expression    (ERROR)
REPAIR      ")" was inserted after "2" and before ":=".

ERROR       Line 5, token 14, unexpected ","
FOLLOWS     Blockbody Variable := "B" ( Term * ( Expression
FORWARD     ? Primary  ? *     (ERROR)
REPAIR      ")" was inserted after "Expression" and before ",".

ERROR       Line 5, token 17, unexpected "/"
FOLLOWS     Blockbody Variable := "B" ( Expression , Term *
FORWARD     ? Primary  ? )     (ERROR)
REPAIR      "<Identifier>" was inserted after "*" and before "/".

ERROR       Line 6, token 1, unexpected "if"
FOLLOWS     Blockbody Variable := "B" ( Expression , Expression )
FORWARD     ? Primary  ? Relationop Expression  ? then     (ERROR)
REPAIR      ";" was inserted after ")" and before "if".

ERROR       Line 6, token 6, unexpected "then"
FOLLOWS     Blockbody if Expression then
FORWARD     ? Statement  ? ;  ? Label_definition  ? end
REPAIR      "then" was deleted, after "then" and before "Statement".

END OF PARSE.
```

Figure 3. Run of error recovery algorithm on program of Graham and Rhodes.

REFERENCES

[A&J 74] Aho, Alfred V. and Johnson, S. C. LR parsing. *Computing Surveys 6*, 2 (June 1974), 99-124.

[A&U 72] Aho, Alfred V. and Ullman, Jeffrey D. *The Theory of Parsing, Translation, and Compiling*. Vol. 1, Prentice-Hall, Inc., Englewood Cliffs, N.J., 1972.

[DeR 71] DeRemer, Frank. Simple LR(k) grammars. *Comm. ACM 14*, 7 (July 1971), 453-460.

[DeR 77] DeRemer, Frank. Tree-affix dendrogrammars. Research proposal to NSF, Information Sciences, University of California, Santa Cruz, CA. 95064, 1977.

[D&R 76] Druseikis, Frederick C. and Ripley, G. David. Error recovery for simple LR(k) parsers. Dept. of Computer Science, University of Arizona, Tucson, AZ. 85721, 1976.

[D&R 77] Druseikis, Frederick C. and Ripley, G. David. Extended SLR(k) parsers for error recovery and repair. Dept. of Computer Science, University of Arizona, Tucson, AZ. 85721, Feb. 1977.

[G&R 75] Graham, Susan and Rhodes, Steven. Practical syntactic error recovery. *Comm. ACM 18*, 11 (Nov. 1975), 639-650.

[H&U 69] Hopcroft, John E. and Ullman, Jeffrey D. *Formal Languages and Their Relation to Automata*. Addison-Wesley, Reading, Mass., 1969.

[Kos 71] Koster, C. H., A. Affix grammars. In Peck, J. E. L., *Algol 68 Implementation*. Amsterdam, North Holland, 1971.

[MHW 70] McKeeman, W. M., Horning, J. J., and Wortman, D. B. *A Compiler Generator*. Prentice-Hall, Inc., Englewood Cliffs, N.J., 1970.

[O'H 76] O'Hare, Michael F. Modification of the LR(k) parsing technique to include automatic syntactic error recovery. Senior thesis, University

of California at Santa Cruz, Santa Cruz, CA. 95064, June 1976.

[Pen 77] Pennello, Thomas J. Error recovery for LR parsers. M.Sc. Thesis, Information Sciences, University of California at Santa Cruz, Santa Cruz, CA. 95064, June 1977.

[Tai 77] Tai, Kuo-Chung. Noncanonical SLR(1) grammars. Computer Science Department, North Carolina State University, 1977.

[Wat 75] Watt, David. The parsing problem for affix grammars. Department of Computer Science, University of Glasgow, Glasgow, Scotland, 1975.

[Wir 73] Wirth, Niklaus. *Systematic Programming: An Introduction*. Prentice-Hall, Inc., Englewood Cliffs, N.J., 1973.

OFFICIAL DISTRIBUTION LIST

Contract N00014-76-C-0682

Defense Documentation Center
Cameron Station
Alexandria, VA 22314
12 copies

Office of Naval Research
Information Systems Program
Code 437
Arlington, VA 22217
2 copies

Office of Naval Research
Code 102IP
Arlington, VA 22217
6 copies

Office of Naval Research
Code 200
Arlington, VA 22217
1 copy

Office of Naval Research
Code 455
Arlington, VA 22217
1 copy

Office of Naval Research
Code 458
Arlington, VA 22217
1 copy

Office of Naval Research
Branch Office, Boston
495 Summer Street
Boston, MA 02210
1 copy

Office of Naval Research
Branch Office, Chicago
536 South Clark Street
Chicago, IL 60605
1 copy

Office of Naval Research
Branch Office, Pasadena
1030 East Green Street
Pasadena, CA 91106
1 copy

New York Area Office
715 Broadway - 5th Floor
New York, NY 10003
1 copy

Naval Research Laboratory
Technical Information Division
Code 2627
Washington, DC 20375
6 copies

Dr. A. L. Slafkosky
Scientific Advisor
Commandant of the Marine Corps (CodeRD-
Washington, D. C. 20380
1 copy

Naval Electronics Laboratory Center
Advanced Software Technology Division
Code 5200
San Diego, CA 92152
1 copy

Mr. E. H. Gleissner
Naval Ship Research & Development Cent
Computation and Mathematics Department
Bethesda, MD 20084
1 copy

Captain Grace M. Hopper
NAICOM/MIS Planning Branch (OP-916D)
Office of Chief of Naval Operations
Washington, D. C. 20350
1 copy

Mr. Kin B. Thompson
Technical Director
Information Systems Division (OP-911G)
Office of Chief of Naval Operations
Washington, D. C. 20350
1 copy