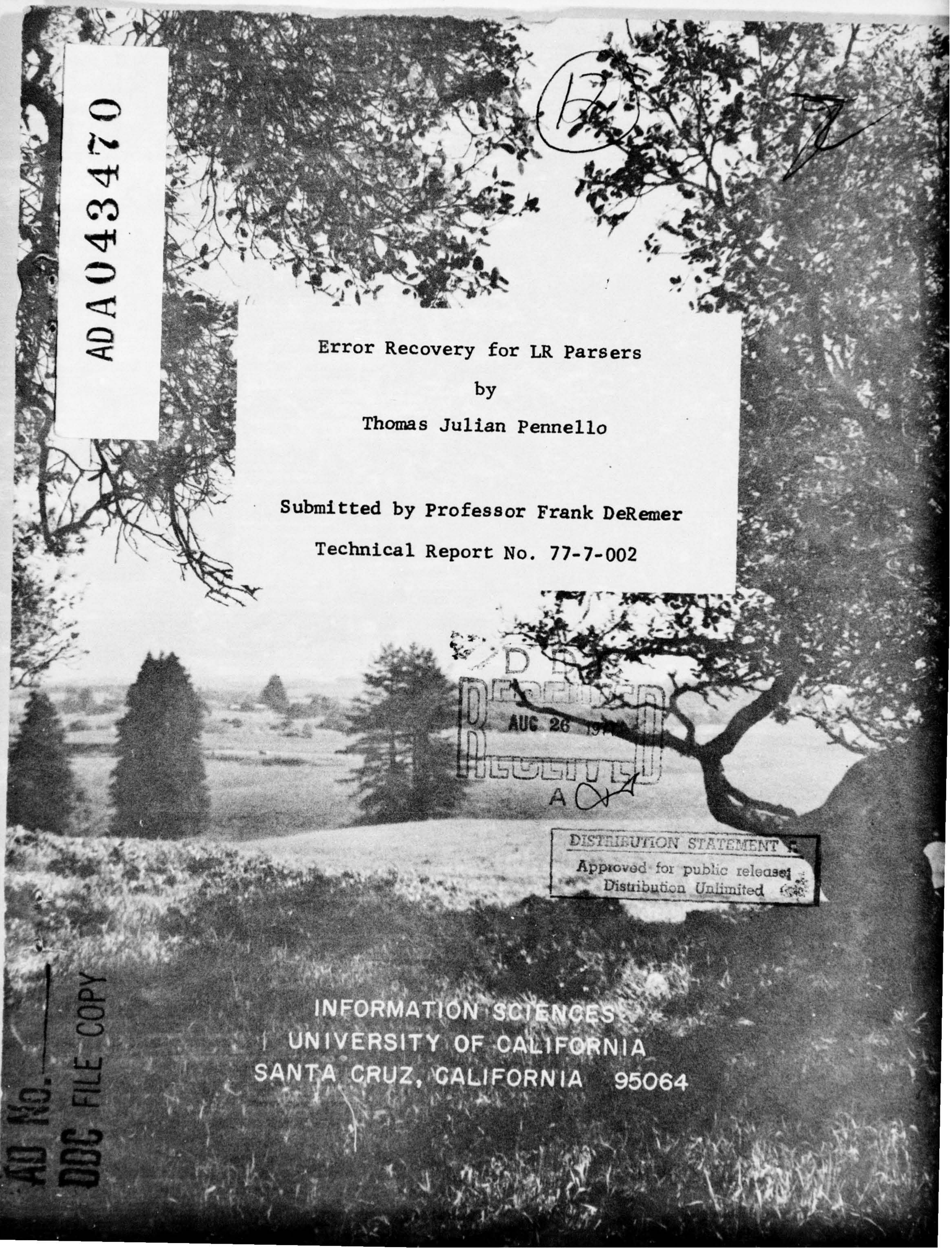

Error Recovery for LR Parsers

by

Thomas Julian Pennello

Submitted by Professor Frank DeRemer

Technical Report No. 77-7-002

INFORMATION SCIENCES
UNIVERSITY OF CALIFORNIA
SANTA CRUZ, CALIFORNIA 95064

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER 77-7-002 ✓	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) Error Recovery for LR Parsers ✓		5. TYPE OF REPORT & PERIOD COVERED
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Thomas Julian Pennello		8. CONTRACT OR GRANT NUMBER(s) ONR N00014-76-C-0682
9. PERFORMING ORGANIZATION NAME AND ADDRESS William M. McKeeman 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 June, 1977
		13. NUMBER OF PAGES 54
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)
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

DISTRIBUTION STATEMENT A
Approved for public release;
Distribution 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 practical algorithm is described that allows an LR parser to parse past the point at which an error was detected. By thus parsing, context beyond the point of error detection is gathered. ~~We prove~~ several important properties about this "forward context" are proven and ~~demonstrate~~ its usefulness in the selection and evaluation of error repairs is demonstrated. At first specifically restricting ~~our~~ the consideration to single occurrences of errors of → next Page

20. ABSTRACT (continued)

insertion, deletion, or replacement of a single terminal symbol, ~~we show how to use~~ the algorithm is given with ~~and~~ suggested possible error repair strategies. Then ~~we suggest~~ a generalization to encompass recovery from any number and type of error is given.

Our work is related to the similar work of Graham and Rhodes for simple precedence parsers. We not only extend their concept to LR parsers but derive properties about forward context that can significantly assist an error repair strategy.

(12)

UNIVERSITY OF CALIFORNIA

SANTA CRUZ

(6) Error Recovery for LR Parsers.

A Thesis submitted in partial satisfaction
of the requirements for the degree of

MASTER OF SCIENCE

in

INFORMATION SCIENCES

by

(10) Thomas Julian Pennello

(11) June, 1977

(9) Master's thesis,

(12) 69p.

(14) TR-77-7-ØØ2

(15) NØØØ14-76-C-Ø682

The thesis of
Thomas Julian Pennello
is approved:

____

____

____
Committee Chairman

____
Dean of the Graduate Division

410 350 New

mt

Error Recovery for LR Parsers

by Thomas Julian Pennello

ABSTRACT

A practical algorithm is described that allows an LR parser to parse past the point at which an error was detected. By thus parsing, context beyond the point of error detection is gathered. We prove several important properties about this "forward context" and demonstrate its usefulness in the selection and evaluation of error repairs. At first specifically restricting our consideration to single occurrences of errors of insertion, deletion, or replacement of a single terminal symbol, we show how to use the algorithm and suggest possible error repair strategies. Then we suggest a generalization to encompass recovery from any number and type of error.

Our work is related to the similar work of Graham and Rhodes for simple precedence parsers. We not only extend their concept to LR parsers but derive properties about forward context that can significantly assist an error repair strategy.

ACKNOWLEDGEMENTS

Frank DeRemer has been of invaluable help in both the exchange of ideas and the reading of this paper. The research effort culminating in this report was instigated by an idea of his similar to FMA' for carrying out a forward move (see [OHa 76]). Interesting discussions with Bill McKeeman have also contributed. Thanks to UC Santa Cruz for free access to the very useful B5700 facility on which implementation and tests were conducted.

Work reported herein was supported in part by the Office of Naval Research under contract N00014-76-C-0682.

CONTENTS

Chapter 1.
INTRODUCTION

Graham and Rhodes [G&R 75] have proposed an error recovery scheme for bottom-up deterministic parsers that involves "condensing" context about the point at which an error was detected. A "backward move" condenses the context to the left of the error point, and a "forward move" gathers context to the right of the error point. Such context is valuable input to an error repair strategy. In their paper they show how the condensation is done for simple precedence parsers, and give an error repair strategy that uses the condensed context.

We investigate the condensation problem for LR parsers (by which we mean to include LR(k) and all its variants -- SLR(k), LALR(k), etc.). We give a practical algorithm that allows an LR parser to perform the forward move, prove several properties about the algorithm relevant to error repair, and suggest ways that the "forward context" may be used in an error repair strategy. We do not treat the backward move since we are not convinced of its usefulness in LR error recovery.

Chapter 2 introduces terminology, both standard and nonstandard, to describe the concepts involved in LR parsing. Chapter 3 gives a preliminary version of the forward move algorithm. The algorithm works by carrying along in parallel all possible parses of the input text following the error point, halting when the parses do not agree as to the next move the parser should make, when the parser must make reference to the context to the left of the error point in order to proceed, or when another error occurs. The halting conditions give the algorithm important properties that can substantially assist an error repair strategy in the selection and evaluation of repairs. These properties we prove in Chapter 4. The most important is that the forward context produced by the forward move algorithm can be used to efficiently verify that a repair attempt is in a sense "consistent" with the input text consumed by the forward move.

In Chapter 5 we give a framework for error recovery: error recovery algorithm = forward move + error repair strategy. Limiting ourselves initially to the consideration of a few (but the most common) types of errors: errors of insertion, deletion, or replacement of a single terminal symbol, we show how to use the forward move algorithm to gather forward context. We suggest ways that the forward context may be used to assist an error repair strategy, based upon the properties proved in Chapter 4.

Finally we convert the algorithm in Chapter 3 to an equivalent but practical algorithm. The algorithm in Chapter 3 explicitly carries along the parallel parses; in Chapter 6 we recode the algorithm in terms of additional states and transitions between them, in essentially the same way a nondeterministic finite-state machine is converted to a deterministic finite-state machine. The recoded algorithm carries the parallel parses implicitly, and is about as efficient as the LR parsing algorithm.

Chapter 7 summarizes and lists further areas of research.

Druseikis and Ripley [D&R 76] have solved the forward move problem for SLR parsers; we contrast our technique to theirs.

Chapter 2.
DEFINITIONS AND TERMINOLOGY

We assume the reader is familiar with LR parsers and their construction. We establish terminology for them, both standard and nonstandard. By "LR" we mean to include LR(k) and all its variants -- SLR(k), LALR(k), etc. Those unfamiliar with LR parsers should consult [DeR 69,71].

A context free grammar (CFG) is a quadruple $G = (N,T,S,P)$ where N, T, S, and P represent the terminals, nonterminals, start symbol, and productions, respectively. We define $V = N \cup T$ and, unless we otherwise specify, adhere to the following conventions for Latin letters:

w, y	$\varepsilon\ V^*$
u, v	$\varepsilon\ T^*$
A, B	$\varepsilon\ N$
s, t	$\varepsilon\ T$

We use $\rightarrow$ for the "generates" relation, $\rightarrow^*$ for its reflexive-transitive closure, and $\rightarrow^+$ for its transitive closure. Productions are elements of this relation. Thus, define $\rightarrow$ on $V^* \times V^*$ as

$$w_1 \rightarrow w_2 \text{ iff for some } A \in N,\ v \in T^*,\ w, y \in V^*,$$
$$w_1 = yAv \text{ and } w_2 = ywv \text{ and } A \rightarrow w \in P.$$

This is the <u>rightmost</u> <u>derivation</u>; for the purpose of LR parsing we are not interested in any other definition of derivation. Further, we assume that the grammar contains a production of the form $S \rightarrow S'\perp$, where S and $\perp$ appear in no other production, $S' \in N$, and $\perp \in T$. A (rightmost) <u>sentential</u> <u>form</u> of G is a string $y \in V^*$ such that $S \rightarrow^+ y$. A <u>sentence</u> of G is a sentential form consisting entirely of terminals.

Associate with each production $A \rightarrow w \in P$ a special symbol $\#_{A\rightarrow w}$ not in V. If, for some $A \in N$, $y,w \in V^*$ and $v \in T^*$, $S \rightarrow^* yAv \rightarrow ywv$, we define $yw\#_{A\rightarrow w}$ to be the <u>characteristic</u> <u>string</u> of the sentential form ywv, and any prefix of yw is called a <u>valid</u> <u>prefix</u> of G. Each sentential form of an unambiguous grammar has a unique characteristic string, and the set of all characteristic strings of a grammar is a regular set. A <u>characteristic</u> <u>finite-state</u> <u>machine</u> (CFSM) of G is a deterministic finite-state machine that recognizes the characteristic strings of G [DeR 69].

A <u>finite-state</u> <u>machine</u> (FSM) is a 5-tuple (K,START, SIGMA,V,F) where K is a finite set of states, START $\in K$ is the <u>start</u> <u>state</u>, $F \subseteq K$ is the set of <u>final</u> <u>states</u>, V the vocabulary, and SIGMA the <u>transition</u> <u>function</u> mapping

$K \times V$ into K. Let $G=(N,T,S,P)$. A CFSM of G is the FSM (K,START,SIGMA,V',F) where $V' = N \cup T \cup \{\#_p \mid p \in P\}$ and the states of K are sets of <u>items</u>, marked productions of the form $A \rightarrow x.y$ ('.' is the marker) where $A \rightarrow xy \in P$. START contains the item $S \rightarrow .S'\perp$, among others. Each nonempty state q in K has one or more successors under SIGMA. START has the successor state $\{S \rightarrow S'.\perp\}$, among others. In general, a state q has an s-successor for each symbol s in $N \cup T$ that is preceded by the marker dot in one of q's items. If q contains an item $A \rightarrow w.$ with a marker to the right of all symbols in the right part of the production (such an item is called a <u>final item</u>), q has a $\#_{A\rightarrow w}$ -successor that is the empty set, which is the only final state (i.e. $F = \{ \{\} \}$). The s-successor of q is called a <u>terminal read successor</u> if $s \in T$, <u>non-terminal read successor</u> if $s \in N$, or <u>reduce successor</u> if $s \in \{\#_p \mid p \in P\}$. The reader should consult [DeR 69] for the details of the computation of K. We express the fact that SIGMA(q,s) = q' by the <u>transition</u> $q \xrightarrow{s} q'$. All nonempty states have a unique <u>accessing symbol</u> defined as follows: if a state q is the s-successor of a state q', then the accessing symbol of q is s. This definition does not cover the state START, to which we assign the accessing symbol $\perp$.

A CFSM state having only read successors is called a <u>read state</u>. Any state having one reduce successor and zero or one nonterminal read successors is called a <u>reduce state</u>. States having two or more reduce successors or having one or more reduce successors and one or more terminal read successors are called <u>inadequate states</u>. All states in K are covered by these three definitions except the final state {}.

A <u>path</u> of the CFSM is a sequence of states $q_0, q_1, \ldots, q_n$ such that there exist transitions $q_0 \xrightarrow{w_1} q_1$, $q_1 \xrightarrow{w_2} q_2, \ldots, q_{n-1} \xrightarrow{w_n} q_n$ in the CFSM, and $w = w_1 w_2 \ldots w_n$ is the string <u>spelled out by</u> the path. $w \in V'^*$ <u>describes a path from</u> q_0 <u>to</u> q_n in the CFSM iff there exists a path $q_0, \ldots, q_n$ and the path spells out w. For brevity we say "q_0 <u>gets to</u> q_n <u>by</u> w". For any path P, <u>Top</u> P indicates the last state in the sequence, i.e. if $P = q_0, q_1, \ldots, q_n$ then Top P = q_n. If q_0 gets to q_n by w, then $[q_0:w]$ is the sequence of states $q_0, q_1, \ldots, q_n$ that is the path from q_0 that spells out w (in a CFSM this path is unique). w <u>accesses</u> q if START gets to q by w. We abbreviate [START:w] by [w]. The <u>concatenation</u> of two paths [q:y] and [q':y'], where Top [q:y] = q', is written [q:y][q':y'] and designates [q:yy'] (that is, we do not repeat the state q' in the concatenation of the paths).

For parsers with 1-symbol look-ahead a look-ahead set of terminal symbols is attached to each final item in the states of the CFSM. (Computation of the look-ahead sets may or may not affect the construction of the CFSM.) We use function $LA(q, A \to w)$ to represent the look-ahead set for final item $A \to w.$ in state q. The LR parser for G is the CFSM of G plus a parser decision function PD mapping $K \times V$ into $2^{P \cup \{read\} \cup \{accept\}}$. $PD(q,s) = \{read \mid q \xrightarrow{s} q'$ and $s \in T-\{\perp\}\} \cup \{A \to w \mid q \xrightarrow{\#A \to w} q'$ and $s \in LA(q, A \to w)\} \cup \{accept \mid q = \{S \to S'.\perp\}$ and $s = \perp\}$. The grammar G is LR iff $|PD(q,s)| \leq 1$ for all $q \in K$, $s \in V$. Equivalently, for each inadequate state, the 1-symbol look-ahead sets for final items are disjoint, and if the state has an s-successor, then s is in no look-ahead set.

For later reference we present the LR parsing algorithm, which uses the CFSM, PD, and a pushdown store called the <u>state stack</u>. By "reading a symbol" we mean that the parser strips the input text of its first terminal symbol, exposing the next symbol to be read. We assume that the last symbol, and only the last symbol, of the input is $\perp$. Parsing is accomplished by the following:

<u>LR parsing algorithm</u> (<u>LRPA</u>).

Push START on the (empty) state stack

```
Repeatedly parse according to the following:
Let  h = head of input,  q = state on top of
state stack.
do case  PD(q,h):
   case {read}:  Read the symbol  h  and
      push  SIGMA(q,h)  on the stack.
   case {$A \rightarrow w$}:  Pop  |w|  items off the stack.
      Let  q  be the new top of stack.
      Push  SIGMA(q,A)  on the stack.
   case {}:  Halt, signalling an error
      and rejecting the input.
   case {accept}:  Halt, accepting the input.
   case otherwise (i.e.  |PD(q,h)| > 1):
      Halt, confused; the parser cannot decide
      between the actions presented it.  If  G
      is  LR,  this step will never be
      encountered.
end  LRPA
```

We refer to a configuration of the parser as a pair (Z,R) where Z is the state stack and R is the remaining (unread) portion of the input. Thus the parser starts out in the configuration (START,R) where R is the input. The parser makes transitions from one configuration to another via moves, members of $P \cup \{read\} \cup \{accept\}$. PD maps $K \times V$ into a set of moves. We use $\vdash$ to indicate the parser's transitions from one configuration to another, and $\overset{*}{\vdash}$ and $\overset{+}{\vdash}$ as the reflexive-transitive and transitive closure of $\vdash$, respectively. Thus case {read} of LRPA can be stated as (Zq,hR) $\vdash$ (Zqq',R) where q' = SIGMA(q,h), and case {A → w} as $(Zqq_1q_2 \ldots q_{|w|}, hR \vdash (Zqq_A, hR)$ where $PD(q_{|w|},h) =$ {A → w} and q_A = SIGMA(q,A). The parser accepts iff (START,R) $\overset{*}{\vdash}$ ([S'],$\perp$); we use the synonym <u>accept</u> for ([S'],$\perp$). We define the relation <u>reduces to</u> as follows: (Z,hR) <u>reduces to</u> (Z',hR) iff PD(Top Z',h) is either {read} or {accept}, i.e. all possible reductions on Z with h as the next of input have been carried out, and the parser is prepared to read or accept.

(Many LR parser implementations do not attach look-ahead sets to final items in reduce states, but only to final items in inadequate states. This allows somewhat smaller parse tables, a slightly faster parser, and perhaps less look-ahead set computation time. We regret that the forward move algorithm precludes the use of this efficiency technique. However, the payoff is earlier detection of errors and better error recovery than when the efficiency technique is employed.)

Chapter 3.
FORWARD MOVE ALGORITHM

When an error occurs during parsing (case {} of LRPA), we would like to invoke a mechanism that performs the "forward move" of Graham and Rhodes, i.e. parses some of the remaining input without regard to the text already parsed. In an LR parser, this means that the forward move proceeds without referencing the left context already developed on the state stack. For example, the Algol symbol "do" can appear in two contexts: in a "for" or "while" statement. If "do" is unexpectedly encountered by LRPA, the forward move would resume parsing without knowing which of these two contexts the "do" actually appears in (if either). We would parse ahead as far as we could without referencing the context to the left of the error point, halting when we can no longer parse independent of that context, and ending up with a fragment of a sentential form representing the text we parsed. A grammar for an Algol-like language appears in Figure 1. Consider the would-be program in this language

```
begin integer  X, J; J := 0;
for  X := 1 step 1 until   do begin  J := X  end
end.
```

where we omitted the limiting value in the "for" statement. Upon detecting the error, LRPA's state stack (writing only the accessing symbols of the states) would appear as

$\perp$ begin Stmt ; Stmt ; for Id := Exp step exp until

where we have capitalized nonterminals and left terminals uncapitalized. Now, mark the top of the stack with the symbol ?, and attempt a forward move. We might read as far as the penultimate "end", resulting in the new stack

$\perp$ begin Stmt ; Stmt ; for Id := Exp step Exp
until ? do Stmt

The forward move halts presumably because the appearance of the last "end" indicates that we should reduce either with the production "Stmt $\rightarrow$ for Id := Exp step Exp until Exp do Stmt" or with the production "Stmt $\rightarrow$ while Exp do Stmt", and we do not know which is applicable without looking at the stack to the left of the ?. Reducing the text "do begin X := J end" to "do Stmt" did not require reference to the context to the left of ?; no matter whether a "for" or a "while" appears earlier on the stack, "do begin X := J end" should always be reduced to "do Stmt". We call the text read during the forward move the _forward text_ and that phrase fragment to which the text is reduced the _forward context_.

We describe an algorithm that achieves this forward move by carrying along in parallel all possible parses of the forward text, as long as all parses agree as to the next move to make, and no parse refers to context to the left of the error point. For this algorithm we have not states but sets of states appearing on the stack. (In Chapter 6 we convert the sets of states to states themselves and recode the algorithm so that it is practical.) The algorithm has two initialization steps, followed by repeated parse steps.

```
Forward Move Algorithm (FMA)
Push?:  Push ? = K on an empty stack.
Readh:  Let h = head of input.
   Push $\{q' \mid q \xrightarrow{h} q'$  and  $q \in ?\}$
   on the stack.  Read h.
Parse repeatedly according to the following rules:
Let h = head of input, Q = state set on top of stack.
Let PD = $\bigcup_{q \in Q}$ PD(q,h).
do case PD:
   case {read}:  Read h and push
      $\{q' \mid q \xrightarrow{h} q'$  and  $q \in Q\}$.
   case $\{A \to w\}$:  Perform a reduction:
      Ensure that there are at least $|w|$ state
      sets on the stack following the ? (i.e.
      ensure that the entire right hand side w
      resides on the top of the stack).
      If not, halt.
```

```
            Otherwise, pop $|w|$ state sets off the stack.
            Let $Q$ be the new top of stack.
            Push $\{q' \mid q \xrightarrow{A} q'$ and $q \in Q\}$.
        case {}: Halt, signalling an error.
        case {accept}: Halt; we have consumed all but
            the $\perp$.
        case otherwise (i.e. $|PD| > 1$): Halt.
    end FMA
```

FMA essentially follows all paths starting at any state in the CFSM that allow the parsing of the input text, halting (1) when two different paths end up in states that disagree as to how to continue the parse (this difference is caught in case "otherwise" of FMA), (2) when all paths end up in states requiring a reduction over the ? (case $\{A \to w\}$), (3) when we read the entire input (case {accept}), or (4) when we encounter another error (case {}), i.e. no path can be continued.

We illustrate the halts of case $\{A \to w\}$ and case "otherwise" by Examples 1 and 2 below, where the grammar involved is a simple arithmetic expression grammar. Figure 2 contains the grammar and its CFSM augmented with LALR(1) look-ahead sets.

Example 1. Let the erroneous input string be

$i\ (\ i\ )\ \perp$

LRPA 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
Push?	?	(i) $\perp$
Readh	? {$(_0$}	i) $\perp$
{read}	? {$(_0$} {i_0}	) $\perp$
{P → i}	? {$(_0$} {P_0}	) $\perp$
{T → P}	? {$(_0$} {T_0}	) $\perp$
{E → T}	? {$(_0$} {E_1}	) $\perp$
{read}	? {$(_0$} {E_1} {$)_0$}	$\perp$
{P → (E)}	? {P_0}	$\perp$
{T → P}	? {T_0, T_1, T_2}	$\perp$

The algorithm halts here because

$$PD(T_0,\perp) \cup PD(T_1,\perp) \cup PD(T_2,\perp)$$
$$= \{E \to E + T,\ T \to P ** T,\ E \to T\}$$

Example 2. Input is () $\perp$.

LRPA halts with state stack [(].

FMA step	Stack	Rest of input
Push?	?	) $\perp$
Readh	? {$)_0$}	$\perp$

Halt: $PD()_0,\perp) = \{P \to (E)\}$, 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 the start state; $E \rightarrow E + T$ 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; the ? to $\{T_0, T_1, T_2\}$'s left indicates no knowledge of what precedes the T. Thus we cannot continue parsing without making a guess, and must halt. In effect the three different places in the CFSM in which a T can be read 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 provides useful information for an error recovery algorithm, as we shall see later.

The second initialization step Readh of FMA guarantees that the algorithm produces a forward context of length at least one. If we did not force FMA to read the first symbol, then it might also consider reductions that have the first input 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 Readh.)

FMA computes state sets dynamically; there is no reason why these state sets and the transitions between them cannot be precomputed, resulting in an FSM. This is formalized in Chapter 6. Meanwhile, we can use Chapter 6's results to extend the concepts of transitions and paths to FMA's state sets. Hence, if FMA consumes forward text u from string uv and produces forward context U, we may write $(?,uv) \overset{*}{\vdash} ([?:U],v)$. U represents a "condensed" or "partially parsed" version of u: $U \rightarrow^{+} u$ (we may write $U \rightarrow^{+} u$ instead of $U \rightarrow^{*} u$ since $|u| \geq 1$). To prevent confusion between LRPA and FMA, we prefix moves of FMA by "FMA:", as in FMA:$(?,uv) \overset{*}{\vdash} ([?:U],v)$.

Chapter 4.
THE WEAK VALID FRAGMENT PROPERTY AND FMA

Suppose FMA:$(?,uv) \overset{*}{\vdash} ([?:U],v)$. Relative to the string uv from which FMA reads u, the forward context satisfies an important property called the "weak valid fragment property." First, we define the "valid fragment property" and then weaken it. Informally, for some suffix uv of a sentence, $U \in V^*$ is a "valid fragment" of uv iff $U \to^* u$ and for every y such that $S \to^* yuv$,

$$S \to^* yUv \to^* yuv,$$

and yU is a proper prefix of the characteristic string of yUv. That is, if $S \to^* yuv$ not only must u be derived from U in the generation of yuv (if it is not, then the grammar is ambiguous), but the derivation step deriving yUv must involve the last symbol of U. We define this formally in terms of parser actions:

<u>Definition</u>. For some suffix uv of a sentence, $U \in V^*$ is a <u>valid</u> <u>fragment</u> of uv iff $U \to^* u$ and for every valid prefix y such that $([y],uv) \overset{*}{\vdash}$ <u>accept</u>,

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

In other words, any state stack [y] satisfying the conditions of the definition must cause LRPA to read all of u and develop the valid fragment U on its state stack, i.e. reduce u to U.

In the context of error recovery, this concept has the following significance: Suppose LRPA encounters an error and halts in configuration (Z,uv), with uv a suffix of a sentence. (We deal with the case where uv is not a suffix in Chapter 5.) Let us propose that by substituting [y] for Z we could cause LRPA to accept. How could we verify this proposition? By running LRPA, to be sure. But if we had many such strings [y] to try, running LRPA could be costly. Now, suppose that we had some valid fragment U of uv. A <u>necessary</u> (not sufficient) condition that $([y],uv) \mid\overset{*}{-}$ <u>accept</u> is that a path starting at Top [y] spells out U, i.e. there exists some path [Top[y]:U], implying that (since $U \rightarrow^{*} u$) $([y],uv) \mid\overset{*}{-} ([y][\mathrm{Top}[y]:U],v) = ([yU],v)$. Thus, valid fragments give us a useful tool with which to limit our selection of [y]'s.

It turns out that since FMA <u>reads</u> as its first step, the forward context U that it provides does not quite satisfy the valid fragment property. It is, however, a "weak valid fragment" and can be used in a testing procedure similar to that described above. Informally, for some suffix uv of a sentence, $U \in V^{*}$ is a "weak valid fragment" of uv iff $U \rightarrow^{*} u$ and for every y such that $S \rightarrow^* yuv$, there exists $y' \varepsilon V^*$ such that

$$S \rightarrow^* y'Uv \rightarrow^* y'uv \rightarrow^* yuv,$$

and $y'U$ is a proper prefix of the characteristic string of $y'Uv$. That is, if $S \rightarrow^* yuv$, not only must u be derived from U, but there exists a y' such that $y' \rightarrow^* y$ and the derivation step producing $y'Uv$ involves the rightmost symbol of U. Formally:

Definition. For some suffix uv of a sentence, $U \varepsilon V^*$ is a weak valid fragment (WVF) of uv iff $U \rightarrow^* u$ and for every valid prefix y such that $([y],uv) \mid\!\overset{*}{-}$ accept, there exists a $y' \varepsilon V^*$ such that

$$([y],uv) \mid\!\overset{*}{-} ([y'],uv) \mid\!\overset{*}{-} ([y'U],v).$$

In other words, any state stack $[y]$ that causes LRPA to accept uv must cause it to reduce $[y]$ to some $[y']$, read all of u and develop the weak valid fragment U on its state stack. We shall prove that the forward context returned by FMA satisfies the WVF property. The reason for the complication of reducing $[y]$ to $[y']$ is because FMA does not consider reducing as its first move.

Suppose now that LRPA encounters an error in configuration (Z,uv), and that uv is a suffix of a sentence. If we propose that replacing Z by $[y]$ could cause LRPA to accept, the forward context U of uv

provided by FMA gives us a necessary condition on the validity of [y] as a replacement. $([y],uv) \vdash^{*}$ accept only if there exists y' such that $([y],uv) \vdash^{*} ([y'],uv)$ (by a series of reductions), and there exists a path from Top[y'] that spells out U, i.e. $([y'],uv) \vdash^{*} ([y'][\mathrm{Top}[y']:U],v) = ([y'U],v)$.

We shall now show that the U returned by FMA satisfies the WVF property. In Lemma 1 we explore the nature of the state sets manipulated by FMA. We use this lemma to prove Theorem 1, which establishes the WVF property as a corollary. Theorem 2 gives us the additional result that FMA in some sense tries as hard as it can by consuming the longest possible forward text. Theorem 2 is not essential to our error recovery techniques but reassures us that the techniques perform as well as they can.

Lemma 1 captures the fact that if LRPA starting with any left context on its stack makes the same series of moves as FMA does in parsing string uv, then FMA has kept track of LRPA's state stack in its state sets.

Lemma 1. Suppose FMA:$(?,uv) \vdash_{M_1} \cdots \vdash_{M_r} (?Q_1\ Q_2 \ldots Q_m,\ v)$. If $([y'],uv) \vdash_{M_1} \cdots \vdash_{M_r} (Z,v)$, then $Z = [y']\ q_1\ q_2 \ldots q_m$, where $q_i \in Q_i$, $1 \le i \le m$.

<u>Proof</u>. By induction on r. For $r = 1$: M_1 = read by step Readh of FMA, and FMA has stack ? Q_1. LRPA, after making move M_1, has stack [y'] q_1, where q_1 = SIGMA(Top[y'],u_1). Now $Q_1 = \{q' \mid q \xrightarrow{u_1} q'$ and $q \in K\}$ by Readh, hence $q_1 \in Q_1$.

Assume the hypothesis true for $r = k$; thus FMA has halted with stack ? $Q_1\ Q_2 \ldots Q_m$, and LRPA has stack [y'] $q_1\ q_2 \ldots q_m$. Consider move M_{k+1}.

(1) M_{k+1} = read; let the symbol to be read be s. Then LRPA pushes state q_{m+1} = SIGMA(q_m,s) by case {read} of the parsing algorithm. FMA pushes state set $Q_{m+1} = \{q' \mid q \xrightarrow{s} q'$ and $q \in Q_m\}$. But since $q_m \in Q_m$, $q_{m+1} \in Q_{m+1}$.

(2) $M_{k+1} = A \to w$

FMA pops $|w|$ state sets, leaving stack

? $Q_1\ Q_2 \ldots Q_{m-|w|}$

where $m - |w| \geq 0$ (since there are at least $|w|$ state sets above ? on the stack). It then pushes $Q'_{m-|w|+1} = \{q' \mid q \xrightarrow{A} q'$ and $q \in Q_{m-|w|}\}$. LRPA pushes state $q'_{m-|w|+1}$ = SIGMA($q_{m-|w|}$,A) on the stack. Since $q_{m-|w|} \in Q_{m-|w|}$ by the inductive hypothesis, $q'_{m-|w|+1} \in Q'_{m-|w|+1}$.

(3) M_{k+1} = accept; the stacks remain the same for both FMA and LRPA.

Theorem 1. Suppose FMA:(?,uv) $|\overline{M_1} \ldots |\overline{M_r}$ ([?:U],v). Let h = head(v). Then for every y and Z such that ([y],uv) $|\overset{*}{-}$ (Z,v) and PD(Top Z,h) is either {read} or {accept}, there exists y' such that ([y],uv) $|\overset{*}{-}$ ([y'],uv) $|\overset{*}{-}$ ([y'U],v).

Proof. Choose some y and Z such that ([y],uv) $|\overset{*}{-}$ (Z,v) and PD(Top Z,h) is either {read} or {accept}. We let [y'] be such that ([y],uv) reduces to ([y'],uv). Thus, ([y],uv) $|\overset{*}{-}$ ([y'],uv) and the first move LRPA takes out of configuration ([y'],uv) is read (=M_1). We now prove by induction on r that LRPA's next r moves from configuration ([y'],uv) are $M_1 \ldots M_r$.

For r = 1: M_1 = read by step Readh of FMA. We know that LRPA must read as its first move from configuration ([y'],uv), by our definition of y'. Now let the theorem hold for r = k. By Lemma 1, FMA's stack after move M_k is

$$? \; Q_1 \; Q_2 \ldots Q_m$$

and LRPA's stack after move M_k is

$$[y'] \; q_1 \; q_2 \ldots q_m$$

where $q_i \in Q_i$, $1 \leq i \leq m$. Let the next symbol in the input be s (s is either in u or is the first symbol of v). FMA now makes move M_{k+1}. Consider LRPA's possible actions:

(1) It makes no move at all.

If s is in u, then this case is impossible since $([y],uv) \vdash^{*} (Z,v)$. If $s = h$, then if $s \neq \perp$, then LRPA must be able to move since $PD(\text{Top } Z,h) = \{\text{read}\}$ or $\{\text{accept}\}$ implies that LRPA eventually accepts or reads h; if $s = \perp$, then the only way LRPA cannot move is if its previous move was accept; but then FMA's previous move (by induction) would have been accept, and it cannot then make move M_{k+1}.

(2) It makes move $M'_{k+1} \neq M_{k+1}$.

Then $M'_{k+1} = PD(q_m,s)$.

But then $\bigcup_{q \in Q_m} PD(q,s)$ would contain both M_{k+1} and M'_{k+1} since $q_m \in Q_m$. Hence by case "otherwise" of FMA, FMA would not make move M_{k+1}. This contradicts the fact that FMA makes move M_{k+1}.

Thus we have shown that the next r moves LRPA makes from configuration $([y'],uv)$ are $M_1 \ldots M_r$. But by Lemma 1,

$$\text{FMA}:(?,uv) \vdash_{\overline{M}_1} \cdots \vdash_{\overline{M}_r} (?Q_1\, Q_2 \ldots Q_m,\, v)$$

and

$$([y'],uv) \mid\!\frac{}{M_1} \cdots \mid\!\frac{}{M_r} ([y']\ q_1\ q_2 \cdots q_m,\ v)$$

where $q_i \in Q_i$, $1 \leq i \leq m$. Since $?\ Q_1\ Q_2 \cdots Q_m =$ $[?:U]$, $[y']\ q_1\ q_2 \cdots q_m = [y']\ [\text{Top}[y']:U] = [y'U]$.

<u>Corollary</u>. If $([y],uv) \mid\!\overset{*}{-}$ <u>accept</u>, then there exists y' such that $([y],uv) \mid\!\overset{*}{-} ([y'],uv) \mid\!\overset{*}{-} ([y'U],v)$ (the WVF property for U).

<u>Proof</u>. If $([y],uv) \mid\!\overset{*}{-}$ <u>accept</u>, then there exists Z such that $([y],uv) \mid\!\overset{*}{-} (Z,v)$ and $\text{PD}(\text{Top } Z,\text{head}(v)) =$ $\{\text{read}\}$ or $\{\text{accept}\}$. The corollary now follows.

The next theorem is not essential to the correct fragment property, but reassures us that FMA goes as far as it can without making a decision based on context to the left of the ? state set.

<u>Theorem 2</u>. Consider suffix uv of a sentence. If there exists a sequence of moves $M_1 \ldots M_r$ $(r > 1)$ such that

(i) $M_1 = \text{read}$,

(ii) there exists a valid prefix y such that

$$([y],uv) \mid\!\frac{}{M_1} \cdots \mid\!\frac{}{M_r} ([yU],v)$$

and LRPA never pops any of $[y]$ from the state stack,

(iii) there do not exist valid prefixes y, y' and $k < r$ such that

$$([y],uv) \mid\!\frac{}{M_1} \cdots \mid\!\frac{}{M_k} (Z,R) \mid\!\frac{}{M_{k+1}} (Z',R')$$

$$([y'],uv) \mid\!\frac{}{M_1} \cdots \mid\!\frac{}{M_k} (Y,R) \mid\!\frac{}{M'_{k+1}} (Y',R'')$$

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

then $\text{FMA}:(?,uv) \mid\!\frac{}{M_1} \cdots \mid\!\frac{}{M_r} ([?:U],v)$.

Proof. By induction on r. For $r = 1$: FMA makes move M_1 = read by step Readh. Let the theorem hold for $r = k$, and let y be the valid prefix of hypothesis (ii). By Lemma 1,

$$\text{FMA}:(?,uv) \mid\!\frac{}{M_1} \cdots \mid\!\frac{}{M_k} (?\ Q_1Q_2 \cdots Q_m,\ R)$$

and

$$([y],uv) \mid\!\frac{}{M_1} \cdots \mid\!\frac{}{M_k} ([y]q_1\ q_2 \cdots q_m,\ R)$$

where $q_i \in Q_i$. Let the next symbol of input be s (s is either in u or is the first symbol of v). We show that FMA makes move M_{k+1}. Consider the possible actions of FMA.

(1) M_{k+1} is $A \to w$, but FMA cannot make that move because there are less than $|w|$ state sets following ? on the stack. This contradicts hypothesis (ii): LRPA would then have to pop some of $[y]$ from the state stack.

(2) FMA makes some move $M'_{k+1} \neq M_{k+1}$. But by Lemma 1, $M_{k+1} \in \bigcup_{q \in Q_m} PD(q,s)$ and thus FMA has a choice of at least two moves to make. Thus FMA cannot make move M'_{k+1}.

(3) FMA halts due to another error, i.e. $\bigcup_{q \in Q_m} PD(q,s) = \{\}$. This cannot occur, since by Lemma 1 $M_{k+1} \in \bigcup_{q \in Q_m} PD(q,s)$.

(4) FMA halts in case "otherwise" because it has a choice between two or more moves (one of them M_{k+1}). Let one of the moves, different from M_{k+1}, be M'_{k+1}. Then there is some path $q_0, q_1, \ldots, q_m$ such that $q_i \in Q_i$, $1 \leq i \leq m$, $PD(q_m,s) = \{M'_{k+1}\}$, and $q_0 \in ?$. Let y' access q_0. Then for some Y, Y', and R'',

$$([y'],uv) \mid\!\overline{}_{M_1} \cdots \mid\!\overline{}_{M_k} (Y,R) \mid\!\overline{}_{M'_{k+1}} (Y',R'').$$

This contradicts hypothesis (iii).

We have shown possibilities (1) through (4) to be contradictory, thus the only possibility left for FMA is to make move M_{k+1}, and the inductive step is proved.

Theorem 2 is somewhat tedious, but proves that FMA simulates LRPA in all the (possibly infinite) situations in which LRPA has already parsed some valid prefix y that causes LRPA to read head(u). Thus the ? state set can really be regarded as representing the set of all such valid prefixes.

Parts 1 and 4 of the case analysis demonstrate how FMA proceeds without any knowledge of left context. In part 1, reducing would cause FMA to interrogate context to the left of ? to determine what state to go to on nonterminal A. In part 4, we have 2 or more choices for FMA; the correct choice depends on left context. Parts 2 and 3 capture the fact that the choices FMA is presented with contain all choices that LRPA would ever consider.

In summary, if LRPA encounters an error in configuration (Z,uv), and FMA reads u from uv producing forward context U, we know that FMA makes as many moves as possible and U satisfies the WVF property. We can verify that some proposed replacement of [y] of Z satisfies a necessary condition for $([y],uv) \overset{*}{\vdash} \underline{accept}$ by the following process, which we call CHECK_VALID and which takes as arguments [y], uv, and U:

<u>CHECK_VALID</u>

Determine y' such that ([y],uv) reduces to ([y'],uv). (Note: there may not be any such y', in which case we fail, i.e. [y] is unsatisfactory.) Determine that a path [Top[y']:U] exists. This can be accomplished in the following fashion:

```
Let U = $a_1 a_2 \ldots a_m$.
Let Stack = [y'].
for i := 1 to m do
   if Top Stack $\xrightarrow{a_i}$ q  exists for some  q
   then push  SIGMA(Top Stack,$a_i$)  on Stack
   else we fail
We succeed if the for loop runs to completion.
end of CHECK_VALID
```

CHECK_VALID is a simple, efficient test to check the viability of a proposed stack repair. The essential tactic that guarantees this result is that forward moves never proceed after FMA encounters an inadequacy, a reduction over ?, or another error. Making some arbitrary choice between the alternatives in an inadequate transition in an attempt to continue parsing is a serious mistake; it makes an unwarranted assumption about the context to the left of the error point. The assumption has no foundation and is just a guess that may be wrong.

The WVF property of a forward context U gives us the CHECK_VALID procedure, but there is still another property of U that can aid error repair. If uv is the suffix of some sentence, and FMA:(?,uv) $\vdash^*$ ([?:U],v), then FMA cannot halt in case {} (the error step): the possible set of moves PD will never be empty. The moves in PD can give us information relating to the class of valid prefixes y such that $S \to^* yuv$. We elaborate on the use of this information in the subsequent chapter, but prove a property about it here. Theorem 3 states the property and needs Lemma 2 for its proof.

Lemma 2. Let $FMA:(?,uv) \overset{*}{\vdash} ([?:U],v) = (?Q_1 \ldots Q_m, v)$. For any path $[y'U]$ in the CFSM where $p = Top[y']$, $[Top[y']:U] = p\ q_1 \ldots q_m$ and $q_i \in Q_i$, $1 \leq i \leq m$.

Proof. Let $p = Top[y']$ and $U = a_1 \ldots a_m$. $p \in ?$, hence by step Readh or case $\{A \to w\}$ of FMA, $q_1 = SIGMA(p,a_1) \in Q_1 = \{q' \mid q \xrightarrow{a_1} q' \text{ and } q \in ?\}$. By a simple induction on m, we have the result.

Theorem 3. Let uv be a suffix of a sentence, $FMA:(?,uv) \overset{*}{\vdash} ([?:U],v)$, and $PD = \bigcup_{q \in Top[?:U]} PD(q, head(v))$. Then

(1) $|PD| \geq 1$

(2) For every y', Z', v', M,

$([y'U],v) \vdash_{\overline{M}} (Z',v')$ implies that $M \in PD$.

Proof. If PD were empty, then there could be no y such that $S \to^* yuv$, and hence uv would not be a suffix of a sentence. Hence conclusion (1). Consider now $[y'U]$. $Top[y'U] \in Top[?:U]$ by Lemma 2, hence $PD(Top[y'U],head(v)) \in PD$. Hence conclusion (2).

Thus if LRPA halts in some configuration (Z,uv), and uv is a suffix of a sentence, applying FMA to uv yields a set of moves PD such that if we propose some substitution [y] for Z, there must exist some y' such that $([y],uv) \overset{*}{\vdash} ([y'U],v) \vdash_{M} (Z',v')$ and $M \in PD$. Suppose, for example, that $PD = \{A \rightarrow w\}$ and that $|w| \geq |U|$. Then $M = A \rightarrow w$, and some suffix y'' of y' must be such that y''U = w. Hence we know something explicit about y'. We delay application of this until the next chapter. We call the property guaranteed by Theorem 3 the "next move" property.

In summary, we have shown the following three properties to hold of FMA when applied to a sentence suffix: the returned forward context is a WVF; it parses ahead as far as possible; and it halts with a non-empty set of moves, one of which must be taken next. We have seen how the first property yields an efficient algorithm for validating proposed error repairs, and have hinted at the value of the next move property. In the next chapter we learn how to use FMA to gather forward context in particular error situations and how to use the next move property as an aid to error repair.

We emphasize finally that the results of this chapter do not define any error recovery strategy, but merely provide useful tools that any strategy may use.

Chapter 5.
USING FMA IN AN ERROR REPAIR ALGORITHM

In this section we concern ourselves with determining the best way to use FMA to gather forward context in conjunction with some error repair strategy. As mentioned in the introduction, we restrict ourselves at first to considering only a single occurrence of one of three types of errors: insertion, deletion, and replacement of a single terminal symbol. We note that the errors in the sample test program of Graham and Rhodes [G&R 75] are all of this type. We call this assumption the "simple error assumption."

We can describe these three situations in the following manner $(x, z \in T^*,$ and $s, t \in T)$:

Insertion error: $S \rightarrow^* xz$ but $S \not\rightarrow^* xtz$.

Deletion error: $S \rightarrow^* xtz$ but $S \not\rightarrow^* xz$.

Replacement error: $S \rightarrow^* xsz$ but $S \not\rightarrow^* xtz$.

We view an error recovery algorithm as being composed of two phases: (1) the gathering of forward context, and (2) the application of an error repair strategy (which uses the forward context). Given the simple error assumption, we first investigate how use FMA to acquire forward context. Then we show how an error recovery strategy might use the forward context, leaving the error recovery strategy itself unspecified, but providing general hints as to how it might work.

Gathering forward context. We investigate the different situations LRPA encounters when it detects an error, determine how best to gather forward context in each case, and develop an overall strategy based upon the case analysis.

In the insertion case, LRPA may detect the error before or after reading t, i.e. it may halt in configuration (Z,tz) or in (Z,z') (z' is a suffix of z). In the latter case, the inserted symbol t has been absorbed into the left context z. The possibilities are the same for the replacement case. In the deletion case, LRPA halts in (Z,z') (again, z' is a suffix of z). We consider halting configurations (Z,tz) and (Z,z') separately.

We distinguish between the concepts *error* and *error symptom*. When LRPA encounters an unexpected symbol (case {} of the LR algorithm), we say that it *detects (the existence of) an error* and that the *symptom of the error* is that LRPA fails on the symbol. It is the goal of error recovery to eliminate the symptom.

Case (Z,tz). (We have an insertion or replacement error).

Where Graham and Rhodes and Druseikis and Ripley resume the parse by immediately performing a forward move, we do not. Since the symbol t heads the input, such an action would necessarily start us off in the wrong context. We instead delete the t from the input, and then invoke FMA. Since our simple error assumption guarantees that z is a sentence suffix, the forward context developed is both a WVF and satisfies the next move property.

<u>Case</u> <u>(Z,z')</u>. Either LRPA has absorbed t on its stack (in the replacement/insertion case), or a deletion error occurred. LRPA halts in configuration (Z,z').

Since t has been absorbed onto the stack, z' is a sentence suffix and we merely submit z' to FMA.

<u>Combining the case analyses</u>. Since we cannot know a priori which case is the actual circumstance when an error is detected, we must combine case strategies into one. This combination works as follows: Not knowing whether the unexpected symbol is in error or not, we always initially skip over it, then perform the forward move. By the assumption that the program is mutilated by only a single error, this forward context is derived from a sentence suffix. Then we determine if the unexpected symbol can be attached to the front of the

already developed forward context. If it can, it is most likely not in error (we are thus in case (Z,z')); if it cannot, then with an exception (case "otherwise" of RCA below) we are most likely in case (Z,tz). Therefore, assume LRPA halts in configuration (Z,suv), where $s \in T$, $u, v \in T+$ (uv is a sentence suffix). Compute the forward context by the following algorithm:

```
Right Context Algorithm (RCA).
Determine U such that FMA:(?,uv) $\overset{*}{\vdash}$ ([?:U],v).
Then, try to attach s to the front of U as
follows:
    Determine s' such that (?,suv) $\vdash$ ([?:s],uv)
    $\overset{*}{\vdash}$ ([?:s'],uv) where FMA has made as many
    moves as it can without reading head(u).
    Let PD = $\bigcup_{q \in Top[?:s']}$ PD(q,head(u)).
    do case PD:
        case {read}: Determine if path [?:s'U] exists.
            suv is a sentence suffix only if the path
            exists. If it does not, then discard s;
            we are in case (Z,tz), with s = t. If
            it does, then try to continue the forward
            move farther from configuration ([?:s'U],
            v), i.e. determine U' such that
               ([?:s'U],v) $\overset{*}{\vdash}$ ([?:U'],v').
            It is likely (but not certain) that we
            are in case  (Z,z').
        case  {}:
            We may conclude that  s  is a bad symbol,
            and discard it;  we are in case  (Z,tz),
            with  s = t.
        case otherwise  (i.e. |PD| ≥ 1):
            We cannot conclude anything definite about
            s.  We then end up with two forward con-
            texts,  [s']  and  [U].
    end  RCA
```

RCA sometimes can tell us whether we are in case (Z,tz) or (Z,z'), and in most situations produce a single forward context with which to validate potential error repairs. The single exception is case "otherwise" where we have two. But in all situations we have a forward context (U or U') that is a WVF and satisfies the next move property, since it is the result of applying **FMA** to a suffix of some sentence.

This completes the discussion of how to gather right context.

Error repair suggestions. We do not intend to provide a complete error repair strategy. Rather, we offer only some suggestions and indicate how the forward context might be used to aid the strategy.

In case (Z,tz), the obvious thing to try is the deletion of the unexpected symbol and the replacement of it with all other terminals; the former is achieved by applying CHECK_VALID to the existing stack Z and the forward context U of RCA, the latter by applying it to Z modified by appending to it all possible terminals. Given the simple error assumption we must be able to hit upon the proper correction.

In case (Z,z'), since t (or its absence) is buried in the stack Z, complex stack modifications may be required to repair the error. We illustrate this with the following, where a deletion causes LRPA to erroneously reduce the stack into a "higher context." Suppose that the text "if I=K then I=J else I=L" were altered to "I=K then I=J else I=L". Rather than detecting the deletion of "if", LRPA assumes it is parsing an assignment statement, and halts when the unexpected "then" is encountered, in configuration ([Left_part = Exp], then I=J else I=L$\perp$). Were the "if" not omitted, upon encountering the "then" LRPA would be in configuration ([if Bexp], then I=J else I=L$\perp$). We need a stack modification to transform [Left_part = Exp] to [if Bexp] -- a tall order. This example illustrates how erroneous reductions greatly complicate error repair.

(The occurrence of such erroneous reductions is the reason that we are not convinced of the efficacy of the backward move espoused by Graham and Rhodes and Druseikis and Ripley. The backward move seeks to cause just such reductions.)

To aid the invention of stack repairs, we suggest the use of the next move property, which says that after FMA halts, we have a set PD of moves, one of which must eventually be made. (Although if erroneous reductions have occurred, the "easiest" repair may not include any of those moves.) If FMA terminates in case $\{A \to w\}$ with an attempted reduction over ?, then (given our simple error assumption) we <u>know</u> what phrase was intended and what move we must make (viz., the reduction). In the previous example, suppose the forward context U' computed in case {read} of RCA was such that (?, then I=J else I= L$\perp$) $\overset{*}{\vdash}$ ([then Stmt else Stmt],$\perp$), with FMA halting in case {If_stmt $\to$ if Bexp then Stmt else Stmt}; we then know that, in this example, [Left_part = Exp] must be modified to "if Bexp". After thus modifying the stack we can effect the reduction and resume normal parsing. As an approximation to this we could simply search for some state preceding the ? that reads the nonterminal If_stmt; after finding such a state q we delete all states on top of it and push SIGMA(q,If_stmt) on the stack. In the example, we would delete the top three states, leaving only the start state (=q) on the stack, and resume parsing in the new configuration ([If_stmt],⊥). While not correcting the actual error, we in effect modify LRPA's stack so that it behaves as if the error were corrected. We call this technique SF for "stack forcing", because it tries to "force" a production to fit the stack.

If FMA terminates in case "otherwise", we are given a choice of one or more productions to try to use or possibly a read transition. Only some of these choices are of practical use in improving a repair strategy, as follows. Classify the productions as either "long" or "short" depending upon whether reduction by them would consume the ? state. Long productions give us an indication of what the stack preceding the ? should be; we can submit each of these to SF, in the hope that at least one can be forced to fit. Short productions can also give us some information with regard to this portion of the stack; this information is not explicit but is buried within the CFSM transitions and the items in the states. A way to extract it is to perform the reduction and continue parsing, awaiting a long production that can tell us something explicit. We believe such an approach may be too cumbersome to be useful.

If a read transition is among our choices, let the items associated with the read transition be $A_i \to x_i.ty_i$ where t is the symbol to be read. If we choose to read t, then one of the strings x_i must match the top of the stack, and we can verify this before reading t. There are both long and short such strings x_i, and the long strings can give us information about the stack preceding ?. Unfortunately, to use the read items we must keep the actual items around during parse time, a requirement that is uneconomical in space.

Among all of the possibilities presented when FMA halts with an inadequate transition, the next move property tells us that one must be the "correct" choice. As we have noted, long productions may be of immediate use, but we do not see obvious or simple ways of using the other choices.

Summary. Thus, an error recovery algorithm incorporates (1) the gathering of right context, which RCA outlines how to do, and (2) the application of an error repair strategy, which we have not specified, but for which we have made some suggestions. We further suggest that if the strategy fails to succeed, then we apply the algorithm recursively, again gathering right context and attempting error repair in the hope that some later correction can repair more than one error. The recursive approach ensures that we never stop trying to parse the input, therefore preventing the algorithm from totally failing when we cannot correct some error, or when there are multiple errors in the input.

Chapter 6.
MAKING FMA PRACTICAL

In this section we show how to convert the state sets manipulated by FMA into other states and precompute the transitions between these new states.

We have described FMA as an algorithm that manipulates sets of states in an attempt to keep track of many state stacks at once. FMA computes state sets dynamically by referring to the CFSM; e.g., cases {read} and $\{A \rightarrow w\}$ compute the next state from the previous state Q by calculating $\{q' \mid q \xrightarrow{s} q' \text{ and } q \in Q\}$ (s is h or A). There is no reason why we cannot precompute these state sets and the transitions between them; this gives rise to an <u>error recovery FSM</u> (ERFSM). For a grammar G, let (K,START,SIGMA,V',F) be its CFSM. The ERFSM of G is the 6-tuple (K',?,ERSIGMA,V',F') where ? = K and $F' = \{\{f\} \mid f \in F\} = \{\ \{\ \{\}\ \}\ \}$. K' is computed as follows: Begin with K' = {?}. Repeatedly add to K' the successors of state sets in K', where if $s \in V'$, the s-successor of $Q \in K'$ is $\{q' \mid q \xrightarrow{s} q' \text{ and } q \in Q\}$. Thus for $Q, Q' \in K'$, $s \in V'$, ERSIGMA(Q,s) = Q' iff Q' is the s-successor of Q in the ERFSM. We can in a simple way specify the look-ahead function $LA(Q, A \rightarrow w)$

for elements of K'; $LA(Q,A \rightarrow w) = \bigcup_{q \varepsilon Q} LA(q,A \rightarrow w)$.
The parsing decision function can now be computed as for the CFSM; due to the construction of the ERFSM, we can show that for $Q \varepsilon K'$, $s \varepsilon V'$, $PD(Q,s) = \bigcup_{q \varepsilon Q} PD(q,s)$.
Figure 3 shows the ERFSM for the CFSM of Figure 2.

FMA now need not do dynamic state computation; we can use the ERFSM and algorithm FMA' below to achieve the same effect:

```
FMA'.
Push?:  Push  ?  on the stack.
Readh:  Let h = head of input.
   Push  ERSIGMA(?,h)  on the stack; read  h.
Parse repeatedly according to the following rules:
Let  h = head of input,  Q = state on top of stack.
Let  PD = PD(Q,h).
do case PD:
   case  {read}:  Read  h  and push  ERSIGMA(Q,h).
   case  {A → w}:  Ensure that  |w|  states reside
      on the top of the stack following the  ?
      state.  If not, halt.
      Otherwise, pop  |w|  states off the stack.
      Let  Q  be the new top of stack.
      Push  ERSIGMA(Q,A)  on the stack.
   case  {}:  Halt, signalling an error.
```

```
        case  {accept}:  Halt; we have consumed all
            but the  $\perp$.
        case  otherwise (i.e.  $|PD| > 1$):  Halt.
    end FMA'.
```

The fact that FMA and FMA' are equivalent should not be difficult to see based on the construction of the ERFSM. Note that FMA' is much like the normal parsing algorithm in that it manipulates only states.

Now that FMA' manipulates states rather than state sets, we can suggest a space optimization on the ERFSM. Suppose for some $q \in K$, $\{q\} \in K'$ (this occurs often; see Figure 3). If $q \xrightarrow{s} q'$ is a transition of the CFSM, then $\{q\} \xrightarrow{s} \{q'\}$ is a transition of the ERFSM. 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 in the final parser + error recovery package. The ERFSM may share every state $\{q\}$ with its corresponding state q in the CFSM. The following criterion, satisfied by (but not only by) the singleton states in K', determines whether an ERFSM state can be shared with a CFSM state: (<u>State sharing criterion</u>) for any $q \in K$, $Q \in K'$, Q may share with q iff for every $y \varepsilon V'^*$, if q gets to p by y and Q gets to P by y then $PD(p,h) = PD(P,h)$ for every $h \varepsilon V$. Phrased differently, if y describes a path from q to p in the CFSM and a path from Q to P in the ERFSM, the parsing decisions that P and p make must be the same. States in K' other than singleton sets satisfy this criterion. To see this, let $t_0 = \{A \rightarrow t.\}$ and $t_1 = \{A \rightarrow t., \; B \rightarrow t.\}$, both members of K. Let $\{t_0,t_1\} \varepsilon K'$. Note that $t_0 \cup t_1 = t_1$. Then if $PD(t_1,h) = PD(\{t_0,t_1\},h)$ for every $h \varepsilon V$, $\{t_0,t_1\}$ may be shared with t_1. This is the same as requiring that the look-ahead for production $A \rightarrow t$ 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 check for. Singleton states are very easy to check for when generating the ERFSM, and the LALR generator at UC Santa Cruz does this. Figure 4 shows the shared ERFSM for the ERFSM of Figure 3.

For grammars we have run, which include a grammar for PASCAL, from 60 to 80 percent of the ERFSM states may be shared, resulting in a substantial savings in space.

FMA' resembles the technique of Druseikis and Ripley. However, they (1) do not have a unique start state with which to begin the forward move, (2) do not consider states in the ERFSM to be sets of states in K but rather actual item sets (our research independently started out that way but study revealed that the item sets were unions of item sets of states in K, so that ERFSM states were conceptually better modelled and computed as sets of states in K), (3) handle the problem only for SLR grammars (they claim that the generalization to LALR is straightforward, but their paper does not indicate the greater difficulty in computing LALR look-ahead sets for the ERFSM; they merely attach SLR look-ahead sets to every production in the ERFSM, and SLR look-ahead sets are computed independently of the state in which the final item appears). Our technique works in general for LR parsers of any type, handling SLR as a special case. In addition, the number of states in our CFSM plus the number of states in our ERFSM can be up to $|V| - 1$ fewer than the number of states needed by Druseikis and Ripley to implement the parser and error recovery machine; this is due to the $|V|$ start states needed by their error recovery machine.

Chapter 7.
CONCLUSION

We have provided a method to do the forward move of Graham and Rhodes for LR parsers in a practical and efficient manner. We have shown that our algorithm FMA carries the forward move along as far as it possibly can before halting, and that the results of it are useful in selecting and validating error repairs. Given the simple error assumption we have described how FMA can be used to gather forward context, and have indicated how an error recovery strategy might employ the gathered context. At UC Santa Cruz an error recovery strategy using forward context is in development which so far has proven successful in practice.

Further research. We have left unexplored many areas related to FMA. In particular, some of them are

(1) How large is the ERFSM in comparison to the CFSM?
Are their sizes linearly related?
How is this related to the grammar?

(2) On "the average", how much forward text does FMA consume?

What circumstances permit FMA to consume a lot of forward text?

How are these circumstances related to language constructs?

(3) We define a grammar's "robustness" to be proportional to how much forward text FMA consumes on "the average".

Is there an algorithm that indicates weak spots in a grammar, i.e. where the grammar is not robust?

(4) What better or other ways may forward context be used in error repair?

REFERENCES

[A&U 75] Aho, Alfred V. and Jeffrey D. Ullman, The Theory of Parsing, Translation and Compiling, vols. I and II, Prentice-Hall, 1972.

[DeR 71] DeRemer, Frank, Simple LR(k) Languages, CACM, July 1971.

[DeR 69] DeRemer, Frank, Practical Translators for LR(k) Languages, PhD. thesis, Dept. of Electrical Engineering, MIT, Cambridge, Mass., 1969.

[D&R 76] Druseikis, Frederick C. and G. David Ripley, Error Recovery for Simple LR(k) Parsers, Dept. of Computer Science, Univ. of Arizona, Tucson, Az. 85721, 1976.

[G&R 75] Graham, Susan L. and Steven P. Rhodes, Practical Syntactic Error Recovery, CACM, Nov. 1975.

[OHa 76] O'Hare, Michael F., Modification of the LR(k) Parsing Technique to Include Automatic Syntactic Error Recovery, senior thesis, Univ. of Calif. at Santa Cruz, Santa Cruz, CA. 95064, 1976.

```
S → Program ⊥

Program → Stmt .

Stmt → integer Id list,
     → Id := Exp
     → for Id := Exp step Exp until Exp do Stmt
     → begin Stmt list ; end
     → while Exp do Stmt

Exp  → Id
     → Int

Id   → '<IDENTIFER>'
Int  → '<INTEGER>'
```

Figure 1. Grammar for a simple Algol-like language. '<IDENTIFER>' and '<INTEGER>' represent the generic classes of identifiers and integers respectively. "A list B" means a list of A's separated by B's. Capitalized strings are the only nonterminals.

$$
\begin{aligned}
S &\rightarrow E \perp \\
E &\rightarrow E + T \\
  &\rightarrow T \\
T &\rightarrow P ** T \\
  &\rightarrow P \\
P &\rightarrow (E) \\
  &\rightarrow i
\end{aligned}
$$

Figure 2a. A simple arithmetic expression grammar.

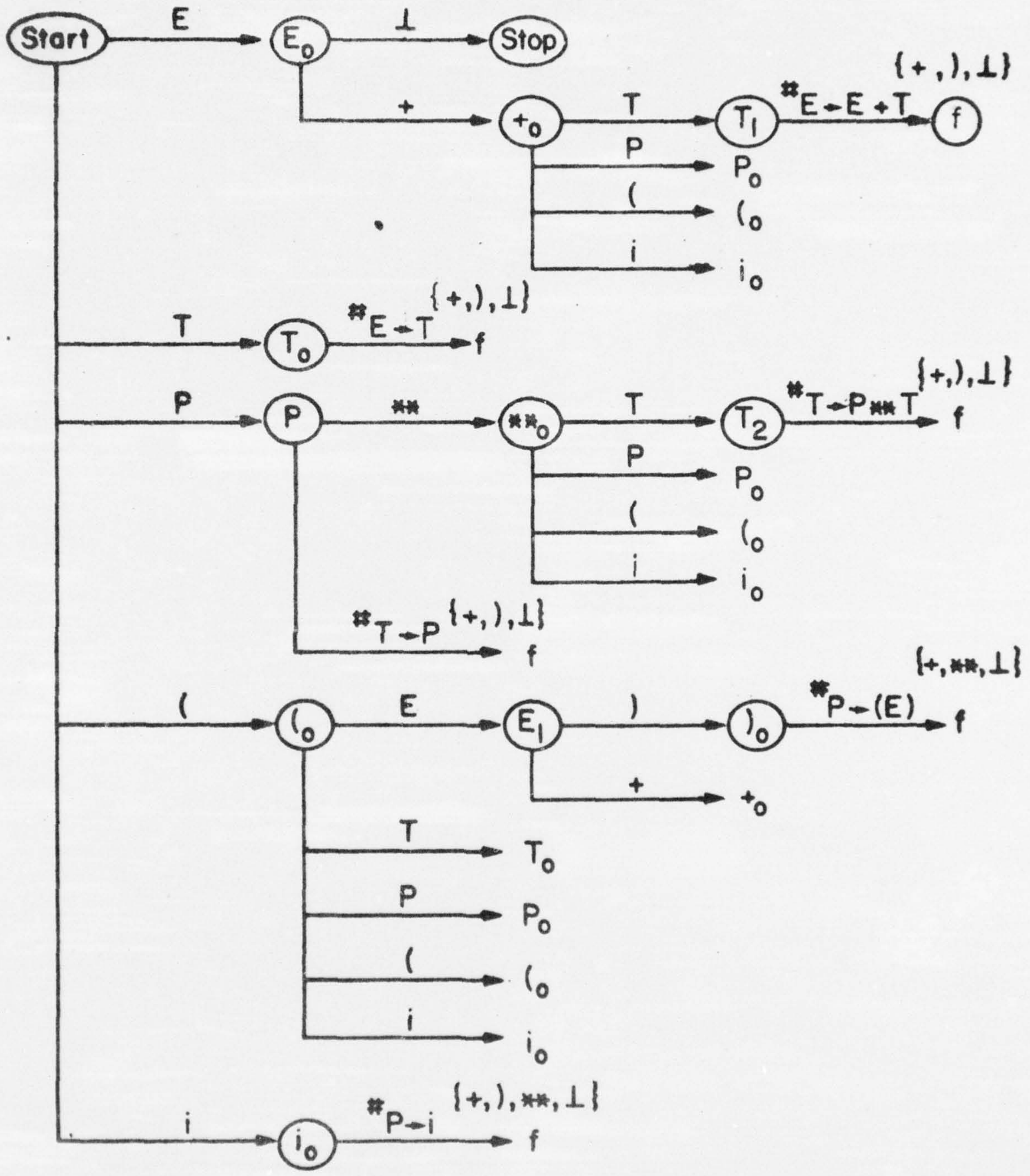

Figure 2b. CFSM for grammar of Figure 2a.

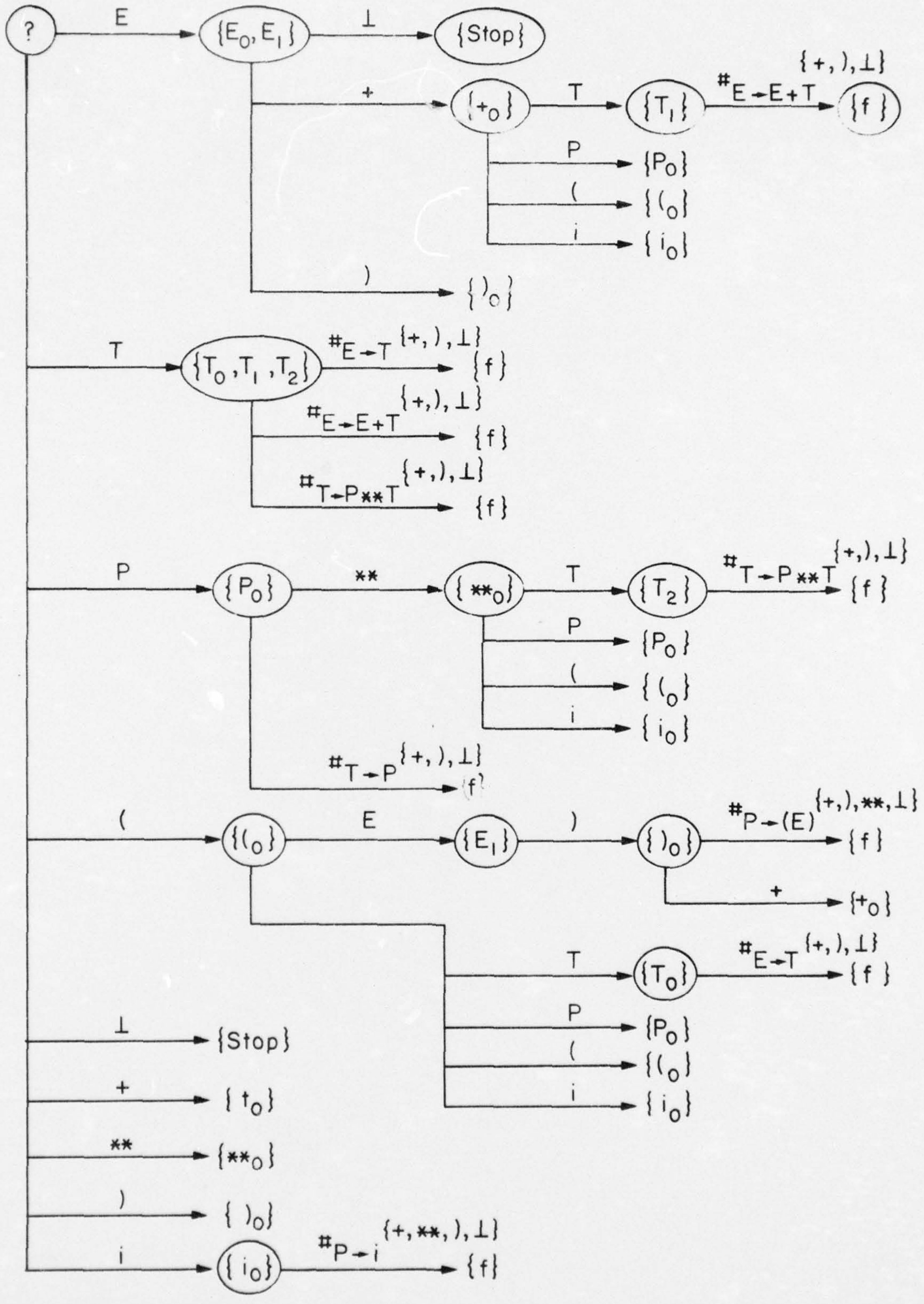

Figure 3. ERFSM for CFSM of Figure 2b. Reduce transitions from ? have been omitted.

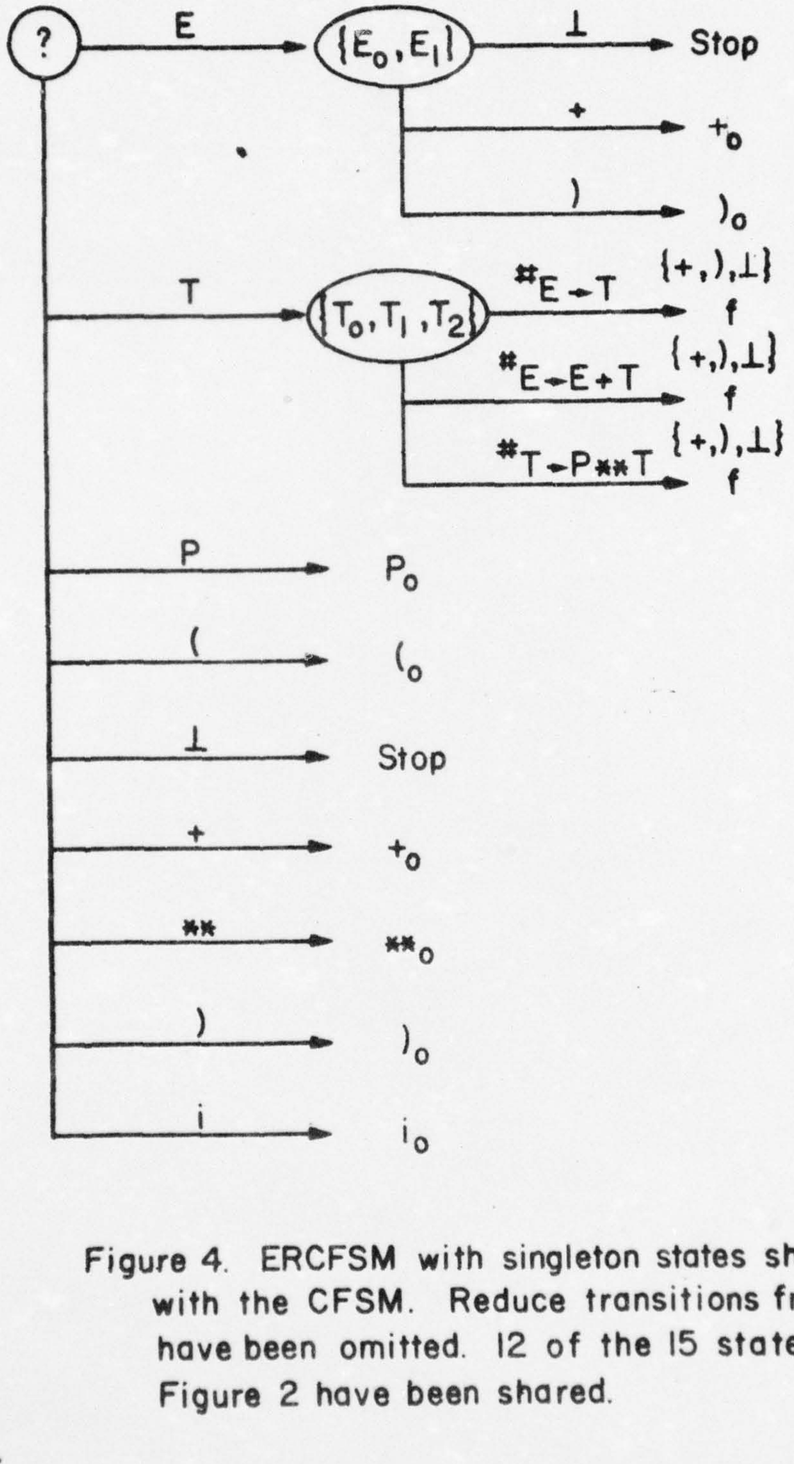

Figure 4. ERCFSM with singleton states shared with the CFSM. Reduce transitions from ? have been omitted. 12 of the 15 states in Figure 2 have been shared.

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-1)
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 Center
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