THE UNIVERSITY OF
WARWICK

Original citation:
Dain, J. A. (1984) Error recovery schemes in LR parsers. University of Warwick. Department of Computer Science. (Department of Computer Science Research Report). (Unpublished) CS-RR-071

Permanent WRAP url:
http://wrap.warwick.ac.uk/60768

Copyright and reuse:
The Warwick Research Archive Portal (WRAP) makes this work by researchers of the University of Warwick available open access under the following conditions. Copyright © and all moral rights to the version of the paper presented here belong to the individual author(s) and/or other copyright owners. To the extent reasonable and practicable the material made available in WRAP has been checked for eligibility before being made available.

Copies of full items can be used for personal research or study, educational, or not-for-profit purposes without prior permission or charge. Provided that the authors, title and full bibliographic details are credited, a hyperlink and/or URL is given for the original metadata page and the content is not changed in any way.

A note on versions:
The version presented in WRAP is the published version or, version of record, and may be cited as it appears here.For more information, please contact the WRAP Team at:
publications@warwick.ac.uk

warwickpublicationswrap
highlight your research

http://wrap.warwick.ac.uk/

The University of Warwick

THEORY OF COMPUTATION

REPORT NO. 71

ERROR RECOVERY SCHEMES IN LR PARSERS

J. A. Dain

Department of Computer Science
University of Warwick
Coventry
CV4 7AL
England

December 1984

Error Recovery Schemes in LR Parsers

J. A. Dain

Dept of Computer Science
University of Warwick
Coventry

1. Introduction

An important function of a compiler is to provide information about errors in the source program, in order to reduce the time a programmer spends on removing those errors. An ideal compiler detects all syntax errors, and all semantic errors detectable at compile-time, gives an accurate and helpful message for each, and gives no other error messages [Horning74]. For syntactic errors, this implies that the parser should recover from each error and continue parsing from the point of error.

Good error recovery may be expensive to implement and inefficient or impractical to use. Automatic construction of error recovery schemes, using formal methods developed from formal methods for syntax analysis, frees the human implementer from this task. Practical schemes must limit time spent in recovery and space occupied by the parser.

LR parsers belong to the class of shift-reduce parsing algorithms which includes algorithms for operator precedence grammars, simple precedence grammars and uniquely invertible weak precedence grammars [Aho72b]. The LR(k) grammars are the context-free grammars that can be parsed bottom-up using a deterministic push-down automaton with k-symbol lookahead. The advantages of using LR parsers are that the LR(k) class of grammars includes all other grammars which can be parsed by shift-reduce algorithms; LR parsers are efficient practical tools and can be generated automatically; they detect syntax

errors as soon as possible in a left-to-right scan of the input string, i.e. at the point at which the scanned input forms a prefix of a sentence but its concatenation with the next input symbol does not (the correct prefix property); LR parsers parse a string of length n in time O(n).

2. Notation

We assume familiarity with the operation and construction of LR parsers as in [Aho77]. We use the following notations and conventions. *Grammar symbols* are terminals and non-terminals. A, B, C denote single non-terminals; a, b, c denote single terminals; α, β, γ denote strings of grammar symbols. A *context-free grammar* G is a tuple (T, N, S, P) where T is the set of terminals, N the set of non-terminals, $S \in N$ the start symbol, P the set of productions. The *input vocabulary* $T \cup N$ is denoted by V. A production is a pair (A, α), written $A \rightarrow \alpha$, where $A \in N$, $\alpha \in V^*$. An LR parser M is a tuple (K, q_0, Σ, F, V) where K is the set of parser states, q_0 the start state, Σ the state transition function : $K \times V \rightarrow K$, F the set of final states, V the input vocabulary. If $\alpha A \gamma$ is a string and $A \rightarrow \beta$ is a production, we write $\alpha A \gamma \Rightarrow \alpha\beta\gamma$ and say $\alpha A \gamma$ *directly derives* $\alpha\beta\gamma$. We use $\overset{*}{\Rightarrow}$ and $\overset{+}{\Rightarrow}$ to denote the reflexive and transitive closure and transitive closure of this relation. A sequence of strings $\alpha_0, \alpha_1, \ldots, \alpha_n$ such that $\alpha_{i-1} \Rightarrow \alpha_i$ for $1 \leq i \leq n$ is a *derivation* of α_n from α_0. A *rightmost derivation* is a derivation in which at each stage the rightmost non-terminal is replaced. A *sentential form* is the start symbol or a string derivable from the start symbol. A *right-sentential form* is a sentential form in a rightmost derivation. A *sentence* generated by the grammar is a sentential form containing terminals only. The *language* L(G) generated by a grammar G is the set of sentences generated by G.

3. Overview

This paper reviews some recent work on practical error recovery schemes for LR parsers, and some important older work. A simple method for error recovery in LR parsers [Aho74], implemented in Yacc [Johnson78a], augments the grammar with productions involving a special **error** token. On error, the parser pops its stack until **error** is a legal token, and restarts. A more sophisticated error recovery scheme for all bottom-up parsers makes a backward move and a forward move from the point of error detection, to establish more context [Graham75]. In an implementation in a simple precedence parser, correction by replacement of sequences of input tokens is made, choosing the appropriate correction by a weighted minimum distance measure.

Various adaptations of the Graham and Rhodes scheme are made for LR parsers. An extension to Simple LR parsers adds states to the parser to obtain more context beyond an error; this context is used to restart the parser, without making any correction to the parse stack [Druseikis76]. Recovery states that obtain context are also added to general LR parsers in [Pennello78], the context then being used to make a repair. The repair is a local correction involving a single terminal deletion, insertion or replacement in the input. Local corrections are also used in schemes for LALR(1) parsers by Graham, Haley and Joy [Graham79] and Burke and Fisher [Burke82]. These schemes have to choose between alternative corrections, the former using a cost measure and the latter making an arbitrary choice. When local corrections fail to repair the input satisfactorily, a second strategy for error recovery is employed involving stack reconfiguration and deletion of input tokens. To control this, error productions and "beacon" tokens are used in [Graham79] and tokens which close scope are used in [Burke82]. A similar strategy using keywords is given by Pai and Kieburtz [Pai80].

These strategies for secondary error recovery are forms of *phrase-level recovery*, in which recovery actions are taken that have the effect of reducing the parse stack [Leinius70]. The parser replaces a string of tokens containing an error with a locally-correct non-terminal, that is one which enables parsing to proceed on the remaining input string. Phrase-level recovery forms the basis for an automatic technique implemented in the compiler-writing system HLP [Sippu83]; the basic strategy is generalized to include local corrections.

4. A simple scheme

LR parsers could adopt many different strategies for error recovery by replacing each error entry in each state with an error correction routine [Aho77]. The context supplied by the set of items for each state may enable construction of sophisticated error recovery routines. One simple method for error recovery uses a terminal symbol **error** and error productions of the form A → **error** for selected non-terminals [Aho74]. When the parser detects an error, it announces this and replaces the current input token with **error**. States are removed from the stack until the top state is one which can shift the **error** token. The parser then shifts **error** and reduces by the appropriate production A → **error**.

The implementer of the parser may choose to write a routine to be associated with this production, such as a routine which skips input until the next input token is a valid last symbol for a string generated by A or a valid first symbol for a string that can follow A in a sentential form.

A parser-generator can be implemented which generates automatically the error recovery productions for a specified set of non-terminals. The parser proceeds as above. The handwritten routines associated with error productions are replaced by actions where the parser automatically skips input until it meets an input symbol which enables a legal parsing action. In effect, the

parser assumes that the error occurred while it was looking for a phrase to reduce to non-terminal A; the recovery action implies that the parser has found such a phrase.

Some schemes may produce a flood of spurious error messages, and a possible solution is for the parser to suppress these until a number of successful shift actions have been performed.

A scheme based on this simple method, using a terminal symbol **error** in some productions, is implemented in the popular parser-generator Yacc [Johnson78a]. In a grammar supplied as input to Yacc, **error** can be used in productions to suggest where errors are expected and recovery may occur. When the parser produced by Yacc encounters an error, it pops the parse stack until it reaches a state where **error** is a legal token. It behaves as if **error** is the next input token and takes the action associated with the appropriate production. It then sets the next input token to the one which caused the error. If no special error rules have been included in the grammar, parsing halts. In order to suppress a flood of error messages, when the parser has detected an error it enters a special mode called error mode. It remains in error mode until a certain number of input tokens has been read and shifted (three tokens in Yacc). While in error mode, any input token causing a further error is deleted from the input and no message is issued for it. The parser can be forced back into normal mode by code in the semantic actions supplied as input to Yacc.

Suppose productions for a statement denoted by non-terminal *stat* are included in some input to Yacc. A possible error production for *stat* is

```
stat: error
```

On detecting an error, the parser attempts to skip over the statement by scanning ahead in the input looking for three tokens that can follow a statement. Alternatively the error production

stat: error ';'

causes the parser on error to search for a semi-colon in the input, discarding any token between the point of error detection and the next semi-colon.

Let us take a closer look at the actions of the parser on detecting an error. The parser generated by Yacc for the grammar

S → A
S → SA
A → c := d ;

with the error production

A → **error** ;

where N = { S, A } and T = { c, d, :=, ; }, has tables

State	Action						Goto	
	c	:=	d	;	**error**	$	S	A
0	S3				S4		1	2
1	S3				S4	accept		5
2	R S → A				R S → A	R S → A		
3		S6						
4			S7					
5	R S → SA				R S → SA	R S → SA		
6			S8					
7	R A → **error**				R A → **error**	R A → **error**		
8				S9				
9	R A → c := d;				R A → c := d;	R A → c := d;		

Suppose the parser is given the erroneous input

c := d;
c := d
c := d;

The resulting configurations of the parser are as follows, where the symbol $

denotes the bottom of the stack and the end of the input, S*n* means shift to state *n*, and R*p* means reduce by production *p*.

Stack	Input	Action
$0	c := d; c := d c := d;$	S3
$0 c 3	:= d; c := d c := d;$	S6
$0 c 3 := 6	d; c := d c := d;$	S8
$0 c 3 := 6 d 8	; c := d c := d;$	S9
$0 c 3 := 6 d 8 ; 9	c := d c := d;$	R A → c := d;
$0 A 2	c := d c := d;$	R S → A
$0 S 1	c := d c := d;$	S3
$0 S 1 c 3	:= d c := d;$	S6
$0 S 1 c 3 := 6	d c := d;$	S8
$0 S 1 c 3 := 6 d 8	c := d;$	insert **error** and pop stack until **error** is legal token
$0 S 1	**error** c := d;$	S4
$0 S 1 **error** 4	c := d;$	further error - delete token
$0 S 1 **error** 4	:= d;$	further error - delete token
$0 S 1 **error** 4	d;$	further error - delete token
$0 S 1 **error** 4	;$	S7
$0 S 1 **error** 4 ; 7	$	R A → **error**;
$0 S 1 A 5	$	R S → SA
$0 S 1	$	accept

which yields a parse tree with frontier

c := d ; **error** ;

Note the silent deletion of three tokens from the stack and a further three tokens from the input.

5. Condensation and Correction

Phrase-level recovery is used in a two-phase error recovery scheme implemented in a simple precedence parser by Graham and Rhodes [Graham75]. The first phase, a *condensation phase*, attempts to summarize the context surrounding the point of error detection. Secondly, a *correction phase* changes the parse stack and remaining input so that the parser may proceed normally. No changes are made to the scanned input.

The condensation phase starts by making a *backward move*, an attempt to make reductions on the stack before the point of error detection; then it makes a *forward move*, an attempt to parse input after the point of error detection. The forward move terminates when a subsequent error is detected, or when the parser calls for a reduction involving the error on the stack. No limit is placed upon the amount of input which can be read during the forward move.

The correction phase aims to change the parse stack to contain a sequence of symbols which could occur while parsing a sentence in the language. Changes to sequences of symbols are considered which will correct the error "locally", i.e. which will change the contents of the stack at the point of error to be the right hand side of a production or its prefix which is legal in context. To choose between changes, a weighted minimum distance measure is used, so the chosen correction is the one requiring minimum modification to the stack. Cost vectors are used to compute the measure: for each grammar symbol, vectors give the cost of inserting and deleting the symbol on the stack. A cost function is also used to give the cost of replacing one symbol by another. If the total cost exceeds some maximum, the parser resorts to a form of "panic mode", not detailed by Graham and Rhodes. A panic mode scheme is a simple language-independent scheme which makes use of a (language-dependent) class of special tokens, sometimes called "beacons", such as ; or **end**. After an error is

detected, the parser skips input symbols until one of the beacons is encountered; the stack is then deleted until this token is legal for the top of stack.

We use the grammar of the previous section, without the error production, to illustrate the method. The table of simple precedence relations for this grammar is as follows.

	S	A	c	:=	d	;	$
S		=	<				
A			>				>
c				=			
:=					=		
d						=	
;			>				>
$	<	<	<				

On input

```
c := d;
c := d
c := d;
```

the configurations of the parser are

Stack	Input	Action
$	c := d; c := d c := d;$	S
$ c	:= d; c := d c := d;$	S
$ c :=	d; c := d c := d;$	S
$ c := d	; c := d c := d;$	S
$ c := d;	c := d c := d;$	R A → c := d;
$ A	c := d c := d;$	R S → A
$ S	c := d c := d;$	S
$ S c	:= d c := d;$	S

$ S c :=	d c := d;$	S
$ S c := d	c := d;$	error

A character-pair error is detected - there is no relation between the tokens d and c. In the condensation phase of error recovery, there is no backward move as no reductions are possible. The forward move marks the error point, stacks the input token and continues, assuming the relation < or = between the symbol preceding the error point and the symbol following it.

$ S c := d ? c	:= d;$	S
$ S c := d ? c :=	d;$	S
$ S c := d ? c := d	;$	S
$ S c := d ? c := d ;	$	R a → c := d;
$ S c := d ? A	$	R S → A

Unfortunately the relation < at the error point now calls for the reduction by S → A to be made, leaving the parser in configuration

$ S c := d ? S	$

The forward move terminates because of a character-pair error. The sequences to be considered for correction are "c := d", "c := d S" and "S". There is no production right-hand side to replace "c := d" or "S" which is locally correct.

Replacements of "c := d S" by "A", "SA" or "c := d;" are all locally correct, and the one with least cost would probably be "c := d;" depending on the cost vectors. We obtain the parse tree with frontier

c := d; c:= d;

so recovery has in effect deleted the "incorrect" statement.

6. SLR Parsers Extended with Forward Moves

Context around the point of error detection is useful in determining recovery action and such context can be established by a forward move and a backward move. LR parsers, unlike simple precedence parsers, do not need a backward move as they contain all left context information in the parse stack. However, this fact can make restarting an LR parser difficult, as its next move may depend crucially on the entire correct prefix already analyzed. A scheme for Simple LR parsers [Druseikis76] constructs states which are used to restart after error. The parser chooses a recovery state which makes a "forward move", obtaining context which is used to determine subsequent recovery action, with no correction made to the parse stack.

Let M_0 denote the original SLR parser. For each grammar symbol $v \in V$ an error recovery state is constructed consisting of all states in the parser that can accept v regardless of left context. A parser M_v is constructed from this error recovery state and its successors; M_v is an extension of M_0 that can carry out a forward move for the symbol v. The *extended SLR parser* M_{Ext} is defined to be the union of all such parsers M_v

$$M_{Ext} = \bigcup_{v \in V} M_v$$

M_{Ext} is not in general an SLR parser but it contains M_0 which is SLR. M_0 may contain inadequate states, that is states which contain a shift/reduce or reduce/reduce conflict. In M_0, one symbol of lookahead is sufficient to resolve

the conflicts in all inadequate states (M_0 is SLR(1)); but M_{Ext} may contain inadequate states with conflicts which cannot in general be resolved by any amount of lookahead. Such states, termed *trapped states*, arise because the parsers M_v do not use all the left context available to states of M_0.

M_{Ext} works as an SLR(1) parser, halting on error, with the following extension for its inadequate states. Let q denote the current (inadequate) state and x the next input token. The algorithm for the parser is:

```
if x is a member of one of q's lookahead sets
        then reduce
else if q contains an x-successor
        then shift
else begin
        signal error
        if there is only one possible reduction for q
                then reduce and continue parse
                else halt
     end
```

(The reduction performed by an inadequate state is referred to as the "backward move").

When the parser halts on error, whether from an adequate or inadequate state, the *forward move* commences. The current state q_e is marked on the stack as containing the point of error detection and parsing restarts with parser M_x, where x is the current input token. M_x can always proceed as it can shift x by virtue of its construction.

The forward move terminates when one of the following occurs:

C1 A further error is detected in a read state or an inadequate state that cannot reduce.

C2 A reduction is called for that involves reducing over the point of error, i.e. popping state q_e from the stack.

C3 The parser enters a trapped state.

When the forward move halts, *simple recovery* involving no correction to the

parse stack or scanned input is made. The strategy is to use for recovery only context obtained from the forward move.

SR1 If the forward move terminates because of C1, with input symbol x, restart with parser M_x.

SR2 If the forward move terminates because of C2, with reduction by production $A \rightarrow \alpha$, restart with parser M_A.

SR3 If the forward move terminates because of C3, in trapped state q with input symbol x, there are three possibilities.

(a) If q has an x-successor and x is not in any lookahead set for q, restart M_x by taking the x-transition out of q.

(b) If q has no x-successor and x is in the lookahead set for exactly one symbol A (where [$A \rightarrow \alpha.$] is an item for q), restart with parser M_A.

(c) In all other cases, restart with parser M_x, first signalling an error if q has no x-successor or x is not in any lookahead set.

The backward move in inadequate states and the resolution of trapped states could be replaced with a new forward move, simplifying the operation of the parser, but are included for better error messages.

An appropriate error message for a parser to make which halts in a read state, with no further reductions possible, is a list of the successors of the state as expected legal symbols. If the parser halts in an inadequate state, the message should not include a list of the lookahead symbols for the state, as that would mislead, but take the form "unexpected symbol".

As a simple example, let us construct the SLR parser for the grammar of the previous sections. We augment the grammar with the production $S' \rightarrow S$ and obtain the canonical collection of LR(0) items

I_0: $S' \rightarrow .S$ $S \rightarrow .A$ $S \rightarrow .SA$ $A \rightarrow .c := d ;$

I_1 = GOTO(I_0, S): S′ → S. S → S.A A → .c := d ;

I_2 = GOTO(I_0, A): S → A.

I_3 = GOTO(I_0, c): A → c. := d ;

I_4 = GOTO(I_1, A): S → SA.

GOTO(I_1, c) = I_3

I_5 = GOTO(I_3, :=): A → c := .d ;

I_6 = GOTO(I_5, d): A → c := d .;

I_7 = GOTO(I_6, ;): A → c := d ;.

The SLR parser has tables

	Action					Goto	
State	c	:=	d	;	$	S	A
0	S3					1	2
1	S3				accept		4
2	R S → A				R S → A		
3		S5					
4	R S → SA				R S → SA		
5			S6				
6				S7			
7	R A → c := d;				R A → c := d;		

To construct the error recovery state for the terminal c we compute the set of all states of the original parser which can shift c, which is the set of states {0, 1}. The basis for state 0 is the set consisting of one item { S′ → .S} and the basis

for state 1 is the set of two items { S′ → S. , S → S.A }. The basis Z_c for the error recovery state for c consists of those items with a non-empty string to the right of the dot, i.e. { S′ → .S , S → S.A }. The closure of this set and its successor states are as follows.

I_8 = Closure(Z_c):
S′ → .S S → S.A
S → .A
S → .SA
A → .c := d ;
(I_8 is not a state of M_0)

GOTO(I_8, S) = I_1 in M_0

I_9 = GOTO(I_8, A):
S → A.
S → SA.

GOTO(I_8, c) = I_3 in M_0

The additional tables for the parser M_c are

	Action					Goto	
State	c	:=	d	;	$	S	A
8	S3					1	9
9					accept		

If the parser is given input

c := d;
c := d
c := d;

it detects error after shifting the second d, when it is in configuration

Stack	Input	Action
$0 S 1 c 3 := 5 d 6	c := d;$	

No backward move (reduction) is possible in state 6. State 6 is marked as the point of error detection and the forward move starts with parser M_c, i.e. state 8. The remaining configurations of the parser are

$0 S 1 c 3 := 5 d 6 ? 8	c := d;$	S3
$0 S 1 c 3 := 5 d 6 ? 8 c 3	:= d;$	S5
$0 S 1 c 3 := 5 d 6 ? 8 c 3 := 5	d;$	S6
$0 S 1 c 3 := 5 d 6 ? 8 c 3 := 5 d 6	;$	S7
$0 S 1 c 3 := 5 d 6 ? 8 c 3 := 5 d 6 ; 7	$	R A → c := d;
$0 S 1 c 3 := 5 d 6 ? 8 A 9	$	accept

yielding two parse trees for the derivations

$$S \Rightarrow A \Rightarrow c := d; \text{ and } A \Rightarrow c := d;$$

Recovery has had the effect of deleting the "incorrect" statement from the input.

7. LR Parsers Extended with Forward Moves

LR (as opposed to Simple LR) parsers may also be extended with recovery states which parse ahead to obtain forward context. The forward move performed by such states is developed as a parallel parse by Pennello and DeRemer [Pennello78]. The parallelism is serialized for implementation, in the form of extra recovery states which return forward context that is used in selecting a repair to the input and parse stack.

The forward move algorithm, called FMA, starts with no left context, i.e. no reference to the parse stack, and proceeds until it must refer to the "missing" left context, when it halts. It carries out all possible parses of the input in

parallel, as long as they agree on the next move, and do not refer to the missing left context.

FMA maintains a stack of sets of states which keep track of the parallel parses. The first step of FMA is to push the set of all parser states onto the stack. The set of all successor states for the current input token is computed and pushed on the stack, and the input is advanced by one token. At each subsequent step, each state in the set on top of the stack is inspected to see what move it would make on the next input token. If all the states in the set that can accept the token agree on the move, and it does not refer to the missing left context, FMA makes that move. Thus FMA follows all paths that allow parsing of the input. It halts if two different paths end in states that disagree on their next move, if all paths end in states which call for a reduction over the error point, if the entire input is accepted, or if another error is detected. When FMA halts, it returns the forward context it has established, where the context is given by the string of grammar symbols $v_1 \ldots v_n$ that give the transitions for the current stack $Q_0 \ldots Q_n$, i.e.

$$\Sigma(q_{j-1}, v_i) = q_j \text{ for all } q_j \in Q_i,\ 1 \leq i \leq n$$

The v_i are uniquely defined by the stack because of the method of construction of LR parser states.

FMA is readily converted into an algorithm for practical implementation which manipulates states rather than sets of states in a similar way to the conversion of an NFA into a DFA.

FMA gives the parser the ability to restart after detecting an error. The scheme also attempts to repair the input; it assumes that a simple error of a single terminal deletion, insertion or replacement has been made, and that the parser detects such an error where it occurs. The parser runs trials of all possible repairs of the form of a single terminal deletion, insertion or substitution. If

no trial leads to the accept state, for an LR(k) parser either the error occurred before the parser detected it or the simple error assumption was incorrect. For an SLR(k) or LALR(k) parser, the parser may have made reductions before detecting error.

Before making any trials, in order to limit the repeated parsing of the remaining input FMA is applied recursively to the remaining input, excluding the token at the point of error detection, which is assumed to be in error. This process, called FMA^+, consumes all the remaining input returning a sequence of forward contexts (sequence of strings of grammar symbols). The extended forward context is used in trials of all possible repairs. If no trial succeeds, the recovery scheme backs down the stack one symbol at a time, trying deletions and replacements of each symbol followed by insertions before that symbol. It may be worth using nonterminals as replacement and insertion symbols, as well as terminals. The technique of "stack forcing", where the stack is searched for a state that can read the LHS of a production called for by a reduce state in the top set of the stack, appears to have potential when considering recovery from errors which have been detected late and caused many erroneous reductions.

It is impractical to parse all remaining input and FMA^+ is stopped after producing a convenient number (seven is suggested) of symbols of forward context. A trial is then deemed successful if all that forward context can be parsed. Further errors in the input may cause FMA^+ to halt before producing the required number of symbols, in which case the first repair to reach the new error point is chosen.

This method of error recovery means that parsing proceeds in non-canonical order (the parser does not produce a rightmost derivation in reverse of the input string), so it may not be worth performing semantic actions which generate code or check semantics, as such actions may occur in incorrect

order. But it is worth continuing to build an abstract syntax tree as simple repairs will build the tree appropriately. FMA can be made practical by pre-computing the state sets and transitions between them which are computed dynamically in FMA. The algorithm for FMA can be modified to manipulate the pre-computed sets of states. The only overhead in speed which the modified FMA imposes on the LR parsing algorithm is in the case where a reduction is called for, where it must check whether the reduction extends over the error point.

Continuing our simple example of a parser for the grammar of Section 4, the LR(1) parse table (and the LALR(1) table) for this grammar is the same as the SLR table given in Section 6. Using the same input of

```
c := d;
c := d
c := d;
```

the parser detects error at the same point, after shifting the second d in state 6. FMA starts by pushing the set of all parser states on its stack of sets of states, and then computes the set of all successor states for the next input token c, pushing that set on its stack and advancing the input. There is only one successor state for c which is state 3. The parse then proceeds, the configurations being as follows (where K denotes the set of all states):

Stack	Input	Action
\$$K$ c {3}	:= d;\$	S5
\$$K$ c {3} := {5}	d;\$	S6
\$$K$ c {3} := {5} d {6}	;\$	S7
\$$K$ c {3} := {5} d {6} ; {7}	\$	R A → c:= d;

The reduction called for, by production A → c := d; , is permitted as there are enough sets of states on the stack. So FMA reduces, popping four sets of states, and pushes onto the stack the set of all A-successor states for the states in the set now on top of the stack.

$*K* A {2, 4} $

The moves called for now are Reduce by S → A and Reduce by S → SA, that is the states in the set do not agree on the next move, and so FMA terminates, returning the forward context "A". A trial with the insertion of a semi-colon in the input will succeed: from the configuration

$0 S 1 c 3 := 5 d 6 ; c := d;$

the parser continues to trace out the derivation for the "correct" input

S ⇒ SA ⇒ Sc := d; ⇒ SAc := d; ⇒ Sc := d; c := d; ⇒ Ac := d; c := d;
⇒ c := d; c := d; c := d;

8. A Forward Move for Yacc

The popular Berkeley Pascal system [Joy80] contains a parser which is generated by a version of Yacc that has been modified to incorporate an improved error recovery scheme [Graham79]. The goal of the scheme is to continue parsing after errors, giving as many useful diagnostic messages as possible, particularly important in a parser which is much used by students new to

programming. To achieve this, the scheme uses error productions in the grammar, a forward move to obtain right context, and a recovery choice based on weighted costs and on semantic information. Lexical and semantic analyses are used as well as syntactic analysis to give these high quality corrections. In order for the scheme to be practical in space and time, only techniques which can be implemented efficiently are used. Recovery actions are of two kinds, *first level* and *second level* recovery. In first level recovery, changes of a single token insertion, deletion or replacement are made to the input. Certain errors cannot be handled by such treatment and the scheme recovers gracefully by invoking second level recovery, in which an incorrect phrase approximating a string derivable from a non-terminal is identified and replaced.

The scheme is implemented in the LALR parser generator Yacc. In a parse table produced by Yacc, if a state has exactly one reduction, then the entries for all tokens which cannot be shifted give that reduction. These *default reductions* reduce the table size but may hinder error recovery, as reductions on the stack may be made in the presence of an illegal next input token. In this scheme, undoing reductions on the stack is considered infeasible; but default reductions are replaced with the actual lookahead symbols, for some designated states. The grammar is augmented with error productions containing the Yacc **error** token. States which shift the **error** token have no default reductions but have all their lookahead tokens enumerated. Error productions are given for the major non-terminals of the language only. The table size is kept practical as some default reductions remain.

On error, the parser inserts **error** before the next input token, and reduces the stack until a state is reached which inspects the next input token. The **error** token is seen and error recovery is invoked. This is equivalent to the backward move of Graham and Rhodes [Graham75]; it condenses the left context informa-

tion, the error productions controlling the amount of condensation.

For first level recovery, no preliminary forward move is made as it appears to be too costly. Instead a small set of changes to the input is computed and a forward move is run on each of these. The first level changes considered are single token changes involving the next input token at the point of error detection and the shift tokens for the current state. The changes are deletion of the input token, insertion of each shift token before it, and replacement of it by each shift token. In addition to changes involving the input token at the point of error detection, if the last action of the parser before detecting error was a shift, then that previous input token is on top of the parse stack: it is removed and placed back on the input and the same set of changes involving that input token and all shift tokens for the current state is considered. No semantic actions need to be undone as the backup involves a single input token. For each such change, a forward move is made on the input, halting when either a specified number of tokens has been consumed, another syntax error is detected, or a reduction is made which gives a semantic error.

The cost of each first level repair is then computed. Each terminal has a cost associated with it. The basic cost of the repair is the sum of the costs of terminals deleted, inserted or replaced in the repair. The basic cost is then multiplied by a factor indicating progress in the forward move. If the specified number of input tokens is consumed this factor is one, consuming fewer tokens gives larger factors. A repair may be eliminated before a forward move is made on it if its basic cost is higher than the total cost of a repair already found. Higher costs are assigned to repairs which involve tokens with semantic attributes. Lower costs are assigned to repairs which replace some keywords by identifiers. A forward move is ended if it calls for a reduction which gives another error, and such a repair is assigned a higher cost.

If no repair is found with a low enough cost, and the state of the parser which invoked the error recovery mechanism has only one shift token, then that token is inserted in the input. A second such insertion is not permitted.

Second level error recovery is invoked if the first level recovery fails to find a single terminal repair with low enough cost. The stack is popped until a state is reached which shifts the **error** token. The input is advanced until a token is reached which is either a legal shift for the state or one of a set of "beacon" symbols for the language. The latter are equivalent to the special tokens used in panic mode schemes.

The recovery scheme produces one short message for each error, stating the token involved and the recovery action taken.

Using the grammar and input from Section 4 as a simple example, the only legal shift token for the state which detects error is a semi-colon. Inserting this allows a forward move to consume the rest of the input; deleting the next token or replacing it with a semi-colon does not allow the forward move to proceed far. So the "obvious" repair is made at the first level of recovery.

9. Forward Moves and Scope Recovery

Another two-phase error recovery scheme whose first phase attempts local correction in steps, backing up the parse stack by one symbol at each step, and whose second phase performs phrase-level recovery, is implemented in LL(1) and LALR(1) parsers by Burke and Fisher [Burke82]. The error routine, invoked when the parser can make no legal action, requires the parse stack to be in the configuration obtaining when the token preceding the token at the point of error detection was shifted. SLR and LALR parsers and LR parsers with default reductions do not satisfy this requirement, and such parsers have to delay reductions until the next shift occurs.

Simple recovery of a single terminal insertion, deletion or replacement is tried first; if no such repair succeeds, insertion, deletion or replacement of a sequence of terminals is tried. The aim of inserted text is to close one or more open scopes, where a scope is a nested construct such as a procedure, block, control structure or bracketed expression. This form of recovery is called *scope recovery*.

The first phase of recovery, primary recovery, backs up the parse stack in stages, by removing a symbol from it and inserting the symbol on the input. At each stage trials of single token repairs are made. Repairs of merging, insertion, substitution, scope recovery and deletion are tried in that order. Merging is joining the previous token with the current token, or the current token with the next token. The tokens used for substitution and insertion are those legal in the current configuration.

A trial of a repair is deemed successful if it enables the parser to consume a specified number of input tokens. If more than one repair is successful an arbitrary repair is chosen. If no repair succeeds but one repair enables the parser to consume more tokens than any other, that repair is chosen providing the number of tokens consumed exceeds some minimum. If no repair succeeds at one stage, the parse stack is backed up one symbol and trials are made. The process is repeated until a repair succeeds or the stack is backed up to a scope opener token such as **begin,** when secondary recovery is invoked.

Secondary recovery restores the parse to the configuration when the error recovery routine was first called. It then deletes symbols one by one, starting with the input token at the point of error detection and continuing to delete symbols from the parse stack; it may insert sequences of tokens that close scope, then checking to see if parsing can continue.

The method uses language-specific maps, for specifying scope opener and

closer constructs, and for controlling primary and secondary recovery for common errors or errors which are difficult to handle. Examples are a map which lists for a given pair of tokens those tokens that should never be inserted between the pair, and a map which lists tokens that should never be substituted for a given token.

Diagnostic messages are synthesized from the error recovery action and the tokens involved, and state the chosen repair.

10. Fiducial Symbols for Phrase-Level Recovery

The phrase-level recovery strategies we have seen so far employ different means of identifying the phrase which contains an error and different ways of choosing a replacement for that phrase. Yet another method is based on *fiducial* (trustworthy) symbols, syntactic tokens chosen for the extent to which such a token constrains a string following it in a sentential form [Pai80]. A *strong fiducial symbol* is one for which all sentential suffixes starting with that symbol can be derived from a single sentential suffix (the symbol fully constrains all suffixes starting with that symbol). A *weak fiducial symbol* is one for which all sentential suffixes of non-recursively defined sentential forms which start with that symbol can be derived from a sentential suffix.

Error recovery is based on weak fiducial symbols, as there are few strong fiducial symbols for programming language grammars, but appropriate grammars can be chosen which yield many reserved words as weak fiducial symbols. On detecting error, the parser scans the input until a fiducial symbol f is encountered. The stack is popped until the top of stack symbol is the fiducial symbol f or a non-terminal which derives a string containing f, or until the stack is empty. In the case that the top of stack is now a non-terminal, it is replaced with a sentential suffix starting with f; in the case that the stack is empty, the start symbol of the grammar is pushed onto it. Parsing can now restart.

Weak fiducial symbols are characterized as possessing *weak phrase level uniqueness:* a terminal a has weak phrase level uniqueness in a grammar G if G is empty, or if

(1) a occurs at most once in the right hand sides of productions of G, say in a production $A \rightarrow \alpha a \beta$, and

(2) the non-terminal A has weak phrase level uniqueness in the grammar obtained from G by deleting all productions for A and considering A to be a terminal.

This method has been used for second-level recovery in a LL(1) parser but could also be suitable for LR parsers.

11. Phrase-Level Recovery in LALR Parsers

The phrase-level strategy of Leinius [Leinius70] is used as the basis of an efficient automatic method in the compiler-writing system HLP (the Helsinki Language Processor) [Raiha83] [Sippu83]. Local corrections and use of left context are incorporated in the basic strategy. HLP, as Yacc, generates LALR parsers with default reductions, but the approach to error recovery does not require forward moves on the input or elimination or postponement of default reductions. Instead the strategy is to isolate, in the parser configuration which detects error, an *error phrase*, a sequence of parse states and input tokens containing the point of error detection, and replace it by a non-terminal whose insertion in the configuration allows parsing to continue. Thus in the parser configuration

$$q_0 \ldots q_i q_{i+1} \ldots q_m \mid a_0 \ldots a_{j-1} a_j \ldots a_n$$

where the q_r are parser states and the a_r are input tokens, the aim is to isolate an error phrase

$$q_{i+1} \ldots q_m \mid a_0 \ldots a_{j-1}$$

and replace it with a non-terminal A such that q_i has an A-successor state q_A which can accept the input token a_j, that is the configuration

$$q_0 \ldots q_i q_A \mid a_j \ldots a_n$$

is legal. The effect of this reconfiguration is the same as if a reduction by the error production

$$A \rightarrow v_{i+1} \ldots v_m a_0 \ldots a_{j-1}$$

had been made, where the v_r are the accessing symbols for the states q_r, i.e.

$$\Sigma (q_{r-1}, v_r) = q_r \text{ for } r = i+1, \ldots, m$$

The basic idea of the method is to select a candidate error phrase and find a reduction goal for it; if the reduction goal is not suitable, then the process is repeated with the next candidate error phrase. Given an error phrase there may be more than one reduction goal for it. For example, in Pascal the error phrase of a partially parsed erroneous statement has reduction goals the non-terminal *Statement* and all special cases including *Compound-Statement*, *Assignment-Statement* and *Unlabelled-Statement*. Whichever reduction goal is chosen, the parser enters a configuration with a state whose accessing symbol is *Statement* on top of the stack. A reduction goal A of an error phrase is said to be *important* if the error phrase has no reduction goal B that can non-trivially derive A using only productions whose right-hand sides are single non-terminals.

The method aims to find a *unique* important reduction goal for an error phrase; but also aims to use left context in the selection of reduction goals, i.e. input already correctly parsed. If the contents of the stack are assumed to be correct, then the stack states to be replaced by a reduction goal should correspond to a prefix of a string derivable from the reduction goal. A reduction goal A of an error phrase $q_{i+1} \ldots q_m \mid a_0 \ldots a_{j-1}$ is *feasible* if there is a string z of terminals such that

$$A \stackrel{*}{\Rightarrow} v_{i+1} \ldots v_m z$$

where the v_r are the accessing symbols of the q_r.

For recovery in practice it is usually enough to inspect the first state q_{i+1} in the stack part of the error phrase. The notion of feasibility is relaxed as follows: a reduction goal A of an error phrase is *weakly feasible* if A can non-trivially rightmost derive a string of the form $v_{i+1}z$. So a candidate error phrase is accepted as the error phrase for recovery if it has a unique important weakly feasible reduction goal.

The first candidate error phrase is the shortest possible one, i.e. containing no states and no tokens. Longer phrases are selected in an order chosen by the user of HLP, the default order being one in which stack states are consumed twice as fast as input tokens. It can happen that no candidate error phrase has a unique important weakly feasible reduction goal, for example if the parser has recognized a sentence but tokens remain on the input. In this case the top state on the stack is deleted and the parser restarts, usually in an error configuration.

Local corrections of a single terminal insertion, deletion or replacement are desirable, because phrase-level recovery depends heavily on the productions of the grammar. A simple error such as omitting a semicolon from a construction can cause the deletion of an entire statement. The productions

Statement-List → Statement | Statement-List ; Statement

cause Statement-List to be chosen as a reduction goal and statement delimiters are the only recovery symbols. Local corrections are computed as terminal or empty reduction goals. The cost of each possible local correction is computed from deletion and insertion costs assigned to each token by the implementer. The local correction with lowest cost is chosen, unless there is more than one such, or the cost is too high. In this case normal phrase-level recovery takes place.

Two messages are produced for each error detected by the parser, a message telling where and why the parser has detected error, and after recovery a message telling what action has been taken. For example, a Pascal program starting with

```
var l, n: real;
```

gives rise to messages

```
* No Program can start with this.
The recovery action was to insert Program-Heading.
```

12. Other Work

[Gries74] contains a short readable introduction to the literature on error recovery. There is a bibliography of error-handling schemes in all parsers in [Ciesinger79], and an indexed bibliography of LR parsers in [Burgess81]. A survey and bibliography of syntax error handling in compilers is given in [Sippu81]. [Gries71] gives an introduction to syntax error recovery, illustrating recovery for top-down parsers with Irons' technique using global context [Irons63] and recovery for bottom-up parsers with a panic-mode scheme as described in [Mckee70].

An automatic error recovery scheme by Mickunas and Modry [Mickunas78] extends the technique of Graham and Rhodes [Graham75] to LR parsers. The scheme performs *condensation* by restarting the parser from the point of error detection and parsing until a second error is encountered or the parser tries to reduce over the error point, and *correction* by trying to link the sequence of parser states before error detection and the sequence of states obtained by condensation by inserting a single terminal. If correction is not possible the parse stack is backed up one state at a time. There may be several condensations, as the parser is restarted in any state that can shift the token that caused the error, and all are considered.

A theoretical model which guarantees the minimum number of changes to the entire input string is the minimum Hamming distance correction. Algorithms based on this model are given by Aho and Peterson [Aho72a], Wagner and Fischer [Wagner74], Lyon [Lyon74] and Backhouse [Backhouse79]. A collection of error transformations is defined, a simple example being insertion or deletion of single terminals. The shortest sequence of error transformations that maps any valid string into an erroneous string is the minimum distance correction. In general this method is too costly to be used in practice, as the best known algorithm for a minimum distance error correction parser is $O(n^3)$. Backhouse finds this impractical and implements a panic-mode scheme in a recursive descent parser, based on Wirth's own scheme for PL/0 [Wirth76]. Wirth uses error productions in the PL360 compiler [Wirth68] to enable the parser to take special action in constructs where errors are expected by the compiler designer.

Somewhere between the locally least-cost model which involves one token at a time, and the globally least-cost model of Aho and Peterson, is the idea of regionally least-cost repair, which selects some region and finds the least-cost repair to that region. This forms the basis for a forward move algorithm for LL and LR parsers [Mauney82].

13. Discussion

The simple scheme employed in Yacc has the advantages of being fast to run and simple to implement. However, diagnostic messages issued by a parser generated by Yacc are not of consistently good quality, because little information about the nature of an error is available; this fact also hinders recovery. No errors are detected in skipped text. Error productions as used in Yacc and [Graham79] have the disadvantages of affecting error recovery in ways which may not be obvious to the user of the parser generator (see the example in Section 4), and of introducing ambiguities in the grammar and making it unwieldy.

For example, if we include in the grammar

```
E → T | E + T
T → F | T * F
F → id
```

the error productions

```
E → T error | E + T error
T → F error | T * F error
```

Yacc reports two shift/reduce conflicts (these conflicts are resolved in favour of shift).

Druseikis and Ripley's method [Druseikis76] for continuing the parse after error provides the starting point for many schemes which seek to repair the input locally using tokens legal for the current state. But their paper gives the method only for SLR parsers, does not use left context and does not attempt to repair the input. When repair is not attempted it is reasonable to ignore left context, as the only aim is to parse the remaining input, and the probability of introducing spurious errors is lessened. But repair is desirable in production compilers and use of left context aids repair. Pennello and DeRemer [Pennello78] attempt repair, but first make a forward move using no left context and then make trials of repairs involving single terminals only. This appears impractical for two reasons: it is time-consuming and it uses an insufficiently rich set of repairs.

For schemes which use costs ([Graham75], [Graham79], [Mauney82], [Sippu83]), the quality of diagnostic messages and repair depends on the implementer's choice of cost vectors. Similarly the use of language-specific maps by [Burke82] requires the implementer to choose such maps carefully.

A surprising number of schemes require changes to be made to the parse stack ([Graham75], [Pennello78], [Mickunas78], [Burke82], [Pai80], [Sippu83]). Such changes are not always practical (see for example [Gries71] and [Graham79]), because symbols on the stack may have been processed and semantic

actions such as symbol table manipulation and intermediate code generation cannot easily be undone.

The strength of methods which have achieved practical success ([Graham79], [Sippu83], [Burke82] for example) lies in the combination of local corrections with phrase-level recovery and heuristic tricks.

The method of Burke and Fisher [Burke82] has certain weaknesses: the requirements of delayed reductions, the removal of symbols from the stack, and the use of various language-dependent maps. The notion of "merging" tokens is not explained adequately. Scope recovery is a form of panic-mode recovery and requires the input to contain enough nested constructs so that only a small portion of the stack is deleted.

A sample of students' Pascal programs has been collected by Ripley and Druseikis [Ripley78] and used by several authors ([Pennello78], [Pai80], [Graham79], [Sippu83], [Burke82]) to evaluate their recovery schemes. The sample contains 126 programs with 182 syntax errors. Evaluation criteria used have been slightly different; those used by Pennello and DeRemer [Pennello78] and Pai and Kieburtz [Pai80] were to rate a correction as excellent if it was the same as a competent programmer might make, good if it was a reasonable correction introducing no spurious errors, and poor otherwise. Sippu and Soisalon-Soininen [Sippu83] use a similar rating for excellent; a good correction is one which is not excellent but introduces no spurious errors and misses no actual errors; fair is one which either introduces one spurious error or misses one error; and poor otherwise. Graham, Haley and Joy [Graham79] "accurately diagnosed well over 80% of the errors". Burke and Fisher [Burke82] "diagnose more than 90% of the syntax errors accurately". Classifying Sippu and Soisalon-Soininen's fair and

poor errors together as poor we obtain a comparison in tabular form.

	[Pennello78]	[Pai82]	[Sippu83]
Missed errors	18%	0%	0%
Recovery action			
Excellent	42%	52%	52%
Good	28%	26%	23%
Poor	12%	22%	25%

14. Future Work

In a review article of 1980, Aho writes "Although there have been considerable improvements, automatic generation of good diagnostics and error recovery methods is still wanting" [Aho80]. We wish to improve error recovery in parsers generated by Yacc. Yacc is a very popular tool, yet the scheme for error recovery is quite old, based on [Aho74], and not easy for novices to handle. In addition to general improvements to Yacc's error recovery, there is a particularly suitable case for treatment. The portable C language compiler of Johnson [Johnson78b] is a striking example of a popular compiler containing a Yacc-produced parser whose diagnostics and repairs need improvement. This parser does not mention precisely where an error was detected, nor what action was taken to recover. For example, the C program

```
main()
{
        int x = 1;
        if x == 1 then x = 2; else x = 3;
}
```

supplied as input to the compiler, gives the unhelpful diagnostics

```
"ex.c", line 4: syntax error
```

and the program

```
struct {
        int real;
        int imag;
} complex

main()
{
        int x = 1;
        complex.imag = x;
}
```

gives the verbose and incomprehensible messages

```
"ex.c", line 6: warning: old-fashioned initialization: use =
"ex.c", line 7: illegal initialization
"ex.c", line 7: syntax error
"ex.c", line 7: warning: old-fashioned initialization: use =
"ex.c", line 8: warning: old-fashioned initialization: use =
"ex.c", line 8: warning: undeclared initializer name x
"ex.c", line 8: operands of = have incompatible types
"ex.c", line 9: redeclaration of complex
"ex.c", line 9: warning: old-fashioned initialization: use =
"ex.c", line 9: syntax error
"ex.c", line 9: warning: old-fashioned initialization: use =
"ex.c", line 10: syntax error
```

The work done by Graham, Haley and Joy has produced a Pascal compiler with diagnostics considerably better than those of the C compiler. We intend to build on the work of those authors who have implemented practical schemes employing local corrections followed by phrase-level recovery ([Graham79], [Pai80], [Sippu83]), but we hope to avoid the use of error productions or changes to the parse stack. A recovery method which on error computes a set of tokens for single token insertion, replacement or deletion, then determines whether the parse can continue, may be a starting point. This method seems likely to repair the input only if the error is a simple one and it would be interesting to know how often this assumption is reasonable in various contexts. A tactic to employ when this method fails to recover is needed.

We consider that parsers should have knowledge of common errors, if there are such things, and some means of using this knowledge. At present the implementer has to incorporate this into a parser by hand and we wish to formalize

this process.

15. Acknowledgement

The author wishes to thank Mike Paterson for advice and encouragement in writing this paper.

16. References

[Aho72a] Aho, A. V. and Peterson, T. G. A minimum-distance error-correcting parser for context-free languages. *SIAM J. Computing 1*, 4 (1972), 305-312.

[Aho72b] Aho, A. V. and Ullman, J. D. *The Theory of Parsing, Translating and Compiling*, Vol. 1. Prentice-Hall, 1972.

[Aho74] Aho, A. V. and Johnson, S. C. LR Parsing. *Computing Surveys 6*, 2 (June 1974), 99-124.

[Aho77] Aho, A. V. and Ullman, J. D. *Principles of Compiler Design.* Addison Wesley, 1977.

[Aho80] Aho, A. V. Translator writing systems: where do they now stand? *Computer 13*, 8 (August 1980), 9-14.

[Backhouse79] Backhouse, R. C. *Syntax of Programming Languages, Theory and Practice.* Prentice-Hall, 1979.

[Bauer74] Bauer, F. L. and Eickel, J. *Compiler Construction - An Advanced Course.* Springer-Verlag, 1974.

[Burgess81] Burgess, C. and James, L. A revised indexed bibliography for LR grammars and parsers. *ACM SIGPLAN Notices 16*, 8 (1981), 18-26.

[Burke82] Burke, M. and Fisher, G. A. A practical method for syntactic error diagnosis and recovery. *ACM SIGPLAN Notices 17*, 6 (June 1982), 67-78.

[Ciesinger79] Ciesinger, J. A bibliography of error-handling. *ACM SIGPLAN Notices 14*, 1 (January 1979), 16-26.

[Druseikis76] Druseikis, F. C. and Ripley, G. D. Error recovery for simple LR(k)

parsers. *Proceedings of the Annual Conference of the ACM*, October 1976, 396-400.

[Graham75] Graham, S. L. and Rhodes, S. P. Practical syntactic error recovery. *Comm. ACM 18*, 11 (November 1975), 639-650.

[Graham79] Graham, S. L., Haley, C. B. and Joy, W. N. Practical LR error recovery. *ACM SIGPLAN Notices 14*, 8 (August 1979), 168-175.

[Gries71] Gries, D. *Compiler Construction for Digital Computers*. Wiley, 1971.

[Gries74] Gries, D. Error recovery and correction - an introduction to the literature. In [Bauer74], 627-638.

[Horning74] Horning, J. J. What the compiler should tell the user. In [Bauer74], 525-548.

[Irons63] Irons, E. T. An error correcting parse algorithm. *Comm. ACM 6*, 11 (November 1963), 669-673.

[Johnson78a] Johnson, S. C. Yacc - Yet Another Compiler-Compiler. Bell Laboratories, Murray Hill, New Jersey, 1978.

[Johnson78b] Johnson, S. C. A portable compiler: theory and practice. *Proc. 5th ACM Symp. on Principles of Programming Languages* (January 1978), 97-104.

[Joy80] Joy, W. N., Graham, S. L. and Haley, C. B. Berkeley Pascal User's Manual. Computer Science Division, Dept. of Electrical Engineering and Computer Science, University of California, Berkeley, California, 1980.

[Leinius70] Leinius, R. P. Error detection and recovery for syntax directed compiler systems. Ph. D. Th., Computer Science Dept., University of Wisconsin, Madison, 1970.

[Lyon74] Lyon, G. Syntax-directed least-errors analysis for context-free languages, *Comm. ACM 17*, 1 (January 1974), 3-14.

[Mauney82] Mauney, J. and Fischer, C. N. A forward move algorithm for LL and

LR parsers. *ACM SIGPLAN Notices 17*, 6 (June 1982), 79-87.

[Mickunas78] Mickunas, M. D. and Modry, J. A. Automatic error recovery for LR parsers. *Comm. ACM 21*, 6 (June 1978), 459-465.

[Mckee70] McKeeman, W. M., Horning, J. J. and Wortman, D. B. *A Compiler Generator*. Prentice-Hall, 1970.

[Pai80] Pai, A. B. and Kieburtz, R. B. Global context recovery: a new strategy for parser recovery from syntax errors. *ACM Trans. on Programming Languages and Systems 2*, 1 (January 1980), 18-41.

[Pennello78] Pennello, T. J. and DeRemer, F. A. A forward move for LR error recovery. *Proc. 5th ACM Symp. on Principles of Programming Languages* (January 1978), 241-254.

[Raiha83] Raiha, K-J., Saarinen, M., Sarjakoski, M., Sippu, S., Soisalon-Soininen, E. and Tienari, M. Revised Report on the Compiler Writing System HLP78. Report A-1983-1, Dept. of Computer Science, University of Helsinki, Finland, January 1983.

[Ripley78] Ripley, G. D. and Druseikis, F. A statistical analysis of syntax errors. *Computer Languages 3*, 4 (1978), 227-240.

[Sippu81] Sippu, S. Syntax Error Handling in Compilers. Report A-1981-1, Dept. of Computer Science, University of Helsinki, Finland, March 1981.

[Sippu83] Sippu, S. and Soisalon-Soininen, E. A syntax-error-handling technique and its experimental analysis. *ACM Trans. on Programming Languages and Systems 5*, 4 (October 1983), 656-679.

[Wagner74] Wagner, R. A. and Fischer, M. J. The string-to-string correction problem. *J. ACM 21*, 4 (1974), 168-174.

[Wirth68] Wirth, N. A programming language for the 360 computers. *J. ACM 15*, 1 (January 1968), 37-74.

[Wirth76] Wirth, N. *Algorithms + Data Structures = Programs*. Prentice-Hall,

1976.