

Automatic Error Recovery for LR Parsers

M. Dennis Mickunas and John A. Modry
University of Illinois at Urbana-Champaign

In this paper we present a scheme for detecting and recovering from syntax errors in programs. The scheme, which is based on LR parsing, is driven by information which is directly and automatically obtainable from the information that is already present in an LR parser. The approach, which is patterned after that of Levy and Graham and Rhodes, appears to provide error recovery which is both simple and powerful.

Key Words and Phrases: programming languages, error correction, automatic correction, parsing, LR, syntax errors, compilers

CR Categories: 4.1.2, 4.4.2, 5.2.3

Introduction

There are few things more frustrating than spending a great deal of time debugging syntax errors in a program. Often, an error causes a compiler to make an incorrect assumption which leads to a confusing error message as well as to the generation of many additional error messages. The final recovery often involves the skipping of large portions of the program. This means that any additional errors that were skipped over will go undetected until future runs of the program.

General permission to make fair use in teaching or research of all or part of this material is granted to individual readers and to nonprofit libraries acting for them provided that ACM's copyright notice is given and that reference is made to the publication, to its date of issue, and to the fact that reprinting privileges were granted by permission of the Association for Computing Machinery. To otherwise reprint a figure, table, other substantial excerpt, or the entire work requires specific permission as does republication, or systematic or multiple reproduction.

This work was supported by the Research Board of the University of Illinois.

Authors' addresses: M. Dennis Mickunas, 297 Digital Computer Lab., Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, Illinois 61801. John A. Modry, 807 7th Ave., S.E., Rochester, Minnesota 55901.

© 1978 ACM 0001-0782/78/0600-0459 \$00.75

A good syntactic error recovery method should detect and give an accurate and meaningful error message for each error in a program. This means completely recovering, and resuming the parse at the point of each error, so as not to miss detecting any subsequent errors. The advantages of a compiler having a good error recovery method are obvious. The disadvantages are that it may be either very costly to develop or very inefficient to use. An automatically generated error recovery method solves the first of these two problems.

In this paper we present an error recovery scheme which is based on LR parsing [1, 9]. The mechanism is driven by information which is directly and automatically obtainable from the information that is already present in an LR parser. The approach is motivated by the previous work of Levy [12] and of Graham and Rhodes [4, 5, 15].

2. Relation to Previous Work

Surprisingly little work has been done on automatic error recovery. The first attempt was by Irons in 1963 [6]. Subsequent results have been obtained by others [11, 10, 12, 7, 14, 17, 4, 5, 15]. Of these, only [7, 10, 14, 17] and [4, 5, 15] involved implementations. Detailed surveys may be found in [10] and [15].

Our work was motivated by that of Graham and Rhodes, and the present paper deals with the extension of their technique to LR parsers. The main contribution of Graham and Rhodes lies in the development and implementation of a scheme which combines the simplicity of [11] with the powerful correction ability of [12]. The methods of both [11] and [4, 5, 15] are based on the observation that once an error is detected, it is possible to continue parsing in the vicinity of the error, thus "condensing" contextual information which aids in the subsequent correction attempt. The condensation phase of the Graham-Rhodes method consists of a "backward move" in which as many reductions as possible are made on the top elements of the parsing stack, and a "forward move" in which the input just beyond the error point is parsed. Graham and Rhodes were able to implement their method using a simple precedence parsing scheme [18]. With such a parsing method, the correction phase is greatly simplified, and amounts to little more than matching the right-parts of the production rules against one of three patterns.

Our attempts to directly apply the Graham-Rhodes method to LR parsers met with some problems in the correction phase. In particular, the left end of the patterns against which the right-parts of production rules are matched, are clearly delineated in the simple precedence case by the simple precedence relations, whereas no such delineation is present in the LR case. Moreover, unlike the simple precedence grammars, LR grammars need not be uniquely invertible, thus allowing that more than one production rule may match the condensed stack.

There are even subtler problems with the LR case, including the possibility that one right-part is a prefix or a suffix of another, and that some incorrect reductions might have been previously performed. These and other difficulties, together with the observation that the stacked states of an LR parser contain more information about the input than simply the symbols on which they were entered, led us to abandon the pattern matching approach. Instead, our method attempts to use the information contained in the states to directly construct a parse whose sequence of states is "continuous" across the error point.

There is yet another aspect of our method which differs significantly from that of Graham and Rhodes. As will become evident, the extension to LR parsing introduces a great deal of activity on the parsing stack. We must often "backup" the parse, thus retrieving previous, uncondensed stack configurations. This could be accomplished by either maintaining a complete history of the parse or keeping on hand the tokenized input string, from which previous parse configurations may be reconstructed. We have chosen the latter approach augmented by pointers from the parsing stack to the corresponding portions of the input string. This approach is suggested by Graham and Rhodes, although the pointers are needed only for computational efficiency.

3. Definitions and Notations

We shall be concerned with the basic activities of an "LR parser," [1, 9] which is a particular style of the so-called "shift-reduce parser" [1, 2]. Informally, such an LR parser consists of some finite set of *states*, Q , together with two finite sets of symbols, the vocabulary of *nonterminal symbols*, V_N , and the vocabulary of *terminal symbols*, V_T . Elements of Q will be denoted by q , q_i , p , or p_i ; subsets of Q by script letters; elements of V_N by A_i or as $\langle \text{text} \rangle$; elements of $V = V_T \cup V_N$ by X_i ; elements¹ of V^* by Greek letters; elements of V_T by anything other than the above. As the parser proceeds, it "reads" the input (which is a string of terminal symbols) symbol by symbol (shifts), and performs certain transformations (reductions) on the symbols that have been read so far. We use a "cursor" (\uparrow) to indicate the present position of the parser's "read head" and given that the parser is in some state, q , we may schematically represent the *configuration* of the parser as

$$X_1 \dots X_j X_{j+1} \dots X_{j+l} \quad \uparrow \quad x_k x_{k+1} \dots \quad (1)$$

where $X_1 \dots X_j X_{j+1} \dots X_{j+l}$ represents the transformed input $x_1 \dots x_{k-1}$, and $x_k x_{k+1} \dots$ is the input that has not yet been read. We may omit the \uparrow in cases where it is not important.

¹ V^* denotes the set of all *strings* formed by concatenating symbols of V , including the empty string which we denote by Λ .

A *shift* move specifies that the parser, when in a configuration like (1), may shift the cursor past x_k and move to a new state, p . The new state, p , is a function of both q and x_k ; schematically we shall write $q \xrightarrow{x_k} p$, or, using (1),

$$X_1 \dots X_j X_{j+1} \dots X_{j+l} \quad x_k \quad x_{k+1} \dots \quad q \rightarrow p$$

For technical reasons, we shall extend the shift move to permit shifts on nonterminal symbols as well. Moreover, we shall write $p_1 \xrightarrow{\alpha} p_n$ if either $\alpha = \Lambda$ and $p_1 = p_n$, or $\alpha = X_1 X_2 \dots X_{n-1}$ and

$$p_i \xrightarrow{X_i} p_{i+1} \quad \text{for } i = 1, 2, \dots, n-1.$$

A *reduce* move specifies that the parser, when in a configuration like (1), may replace the symbols $X_{j+1} \dots X_{j+l}$ by a single nonterminal symbol, A_{j+1} , and may move to a new state, p :

$$X_1 \dots X_j A_{j+1} \quad \uparrow \quad x_k x_{k+1} \dots \quad p$$

The new state, p , is precisely that state at which the parser arrives by starting in its initial state, q_0 , and performing shifts on $X_1 \dots X_j A_{j+1}$.² We say that the parser has performed the reduction³

$$A_{j+1} ::= X_{j+1} \dots X_{j+l}.$$

An LR parser is a shift-reduce parser, as described above, for which all "conflicts" in choosing which move to make are resolvable by "looking ahead" some fixed number of symbols in the input. For languages of practical interest, this look-ahead can usually be limited to one symbol. Thus, with the configuration (1), the parser need look only at its present state, q , and its incoming input symbol, x_k , in order to uniquely determine its next move (either a shift on x_k , or one of possibly many reductions). The details of just when and how such parsers may be produced are nicely explained in [1]. Further technical details may be found in [1, 2, 3, 9].

Given a state, q , we shall want to know precisely which input symbols admit a shift move, and which admit a reduce move. Thus, for each state, q , we define $S(q)$ and $R(q)$ to be the sets of symbols which admit,

² In practice, the parser "remembers" the states q_0, q_1, \dots which arose in arriving at configuration (1):

$$q_0 \xrightarrow{X_1 \dots X_j} q_j \xrightarrow{X_{j+1}} q_{j+1} \xrightarrow{\dots X_{j+l}} q$$

and upon performing the reduction, the parser can resume in state q_j from the configuration

$$q_0 \xrightarrow{X_1 \dots X_j} q_j \xrightarrow{A_{j+1} X_k X_{k+1} \dots}$$

³ Such a reduction corresponds exactly to a production rule of the formal grammar which defines the language in question, and which gives rise to the LR parser itself. We ignore the technical extension which permits so-called Λ -rules, $A ::= \Lambda$.

respectively, the shift moves and the reduce moves for state q .

Finally, we must introduce the notion of “ancestral” states. In the case of a reduction as detailed above, where the parser was in state q before the reduction and in state p afterwards, we say that q is an *ancestor* of p . Moreover, we wish that all of q ’s ancestors be considered ancestors of p , and that p be considered its own ancestor as well. Likewise, we say that p is a *descendent* of each of its ancestors. For each state $q \in Q$, we compute its set of ancestors (descendents) and denote that set by $\mathcal{A}(q)$ ($\mathcal{D}(q)$).

4. Overview of the Error Recovery Scheme

An LR parser proceeds from configuration to configuration by performing shifts and reductions while altering its state. An error is detected in the configuration

$$\begin{array}{c} \alpha \quad x_1 x_2 \dots \\ \uparrow \\ q \end{array} \quad (1)$$

whenever there are no moves permitted at that point, i.e., whenever x_1 admits neither a shift nor a reduce move for state q ($x_1 \notin S(q) \cup R(q)$). It is at this point that the Error Recovery Scheme is invoked. The Error Recovery Scheme is provided with a copy of the configuration (1) and will return with either an indication of failure or a modified configuration⁴ obtained from (1) by inserting and deleting some number of terminal symbols and partially reparsing the altered portions. In addition, the total cost of the insertions and deletions is returned. In case the Error Recovery Scheme fails to repair the error, then it is presumed that x_1 is a spurious symbol. The action then taken is to delete x_1 and reinvoke the Error Recovery Scheme.

The Error Recovery Scheme consists of two phases: the Condensation Phase and the Correction Phase. The Condensation Phase is identical in spirit to that of the precedence parser, viz. we simply continue parsing from the error point until either 1) a second error is encountered, or 2) the parser attempts to make a reduction which extends past the error point on the stack. (We call this “reducing over the error point.”) There are usually many ways of condensing the configuration (1), and we must consider all of them. Encountering a second error indicates that either 1) there are truly two errors in the vicinity, or 2) the attempted condensation is not the correct choice. The only time that we accept the possibility of a second error is when *all* attempted condensations result in error, and in such a situation, the Error Recovery Scheme is recursively reinvented in an attempt to independently repair and the second error. However,

if any condensation results in an attempt to reduce over the error point, then the corresponding configuration, termed a correction candidate, is passed on to the Correction Phase.

In the Correction Phase, we have a sequence of states which lead from the initial state up to the error point as well as a sequence of states resulting from the Condensation Phase which lead beyond the error point. We attempt to link the two together first by inserting a terminal symbol. If that fails, then we attempt to backup the error point essentially by conceding that just prior to the error point some of the parser’s actions might have been incorrect. Little by little we force the parse to retreat, thus freeing symbols for consumption by an extension of the sequence of Condensation Phase states. After each retreat, we again attempt to link the two sequences together by inserting a terminal symbol. If, at any time after a retreat of the parse, the sequence of the Condensation Phase states cannot be back-extended to include the freed symbol, then that symbol is deleted. Failure occurs if either the parse retreats to the initial state or excessive deletion occurs. We present the Condensation Phase as developing the correction candidates in parallel and subsequently passing them on to the Correction Phase in parallel. In practice, however, this is done serially.

5. The Error Recovery Scheme

The Error Recovery Scheme is called whenever the parser, having arrived at the state q with α as the stack configuration, finds an error with $x_1 x_2 \dots$ as input:

$$\xrightarrow{\alpha_*} q \quad ? \quad x_1 x_2 \dots \quad (1)$$

where $x_1 \notin S(q) \cup R(q)$

5.1 Condensation Phase

Given the error configuration (1), we perform the following.

CD1. Compute the set of states which may shift on x_1 :

$$\mathcal{S} = \{p \in Q \mid x_1 \in S(p)\}$$

CD2. For each $p \in \mathcal{S}$ continue the parse:

$$\xrightarrow{\alpha_*} q \quad ? \quad p \xrightarrow{x_1} p_1 \xrightarrow{x_2} \dots$$

For each $p \in \mathcal{S}$, step CD2 terminates in one of two ways.

Case 1. An attempt is made to reduce over the error point, thus yielding a configuration

$$\xrightarrow{\alpha_*} q \quad ? \quad p \xrightarrow{\beta_*} \quad (2)$$

which is termed a “correction candidate.”

Case 2. Another error occurs, thus yielding a configuration.

⁴ Depending upon the kind of LR parser being used, the new configuration may or may not be suitable for immediate resumption of normal parsing.

$$\alpha_* \rightarrow q \ ? \ p \xrightarrow{\beta_*} p_{n-1} \ ? \ x_n \dots \quad (2')$$

which is termed a "holding candidate."

The Error Recovery Scheme continues with the Correction Phase for each of the correction candidates. If the Correction Phase succeeds in repairing any correction candidate(s), then the Error Recovery Scheme chooses the one with least cost and fewest exits. It is only if the Correction Phase fails to repair any of the correction candidates that the holding candidates are revived. For each holding candidate, the Error Recovery Scheme is recursively reinvoked in an attempt to independently repair the presumed second error which has been encountered.

5.2 Correction Phase

Each correction candidate (2) found in the Condensation Phase is passed on to the Correction Phase which attempts to break the ?-barrier by searching for some terminal symbol, x , for which $q \xrightarrow{x} p$. If such an x is found, then the insertion of x in place of ? apparently provides a correction.

More generally, the correction candidate will take the form⁵

$$\alpha_* \rightarrow q' \xrightarrow{X} q \ ? \ p \xrightarrow{\beta_*} \dots \quad (3)$$

where \mathcal{P} is a set of states, called *rightstates*. Initially, \mathcal{P} contains only p as in (2), but as backup occurs, \mathcal{P} will be expanded. Moreover, we shall want to find more than simply all $x \in V_T$ for which $q \xrightarrow{x} \mathcal{P}$. For it might be that a reduction of some suffix of $\alpha'X$ had been suppressed because the incoming input symbol (the first terminal symbol derivable from β) did not admit the reduction. However, if some x will permit the reduction, its insertion might very well provide the needed repair. The repaired configuration after the reduction(s) (of some suffix of $\alpha'X$ to, say, α'') would be $\alpha''_* \rightarrow q'' \xrightarrow{\beta_*} \dots$. Clearly, we determine x by demanding not merely $q \xrightarrow{x} \mathcal{P}$, but rather $q'' \xrightarrow{x} \mathcal{P}$ or, more generally, $\mathcal{D}(q) \xrightarrow{x} \mathcal{P}$.

Similarly, if the insertion of some x permits the reduction of some suffix of $\alpha'Xx$, we have not $q \xrightarrow{x} \mathcal{P}$ but $q \xrightarrow{x} \mathcal{A}(\mathcal{P})$. Recalling that each state is an ancestor as well as a descendent of itself, we have the following: given the correction candidate (3) and the repair cost accumulated thus far (COST).

CR1. Find all $x \in V_T$ for which $\mathcal{D}(q) \xrightarrow{x} \mathcal{A}(\mathcal{P})$. For each such x , return $\alpha'Xx\beta \dots$ as a repair with associated cost, (COST + insert cost of x). If no insertion repair is found, then we attempt to backup the ?-barrier. It is clear how the parse retreats from q to

q' , but a number of cases arise in attempting to backup the rightstates. In the first case, we attempt to backup \mathcal{P} over X :

CR2. Compute $\mathcal{P}' = \{p \in Q | p \xrightarrow{x} \mathcal{A}(\mathcal{P})\}$. If $\mathcal{P}' \neq \Phi$, then repeat the Correction Phase, beginning at CR1, using the configuration

$$\alpha'_* \rightarrow q' \ ? \ p' \xrightarrow{X\beta_*} \dots \quad (3')$$

instead of (3).

No additional cost is incurred by such a backup.

If, however, $\mathcal{P}' = \Phi$, i.e., none of the rightstates can backup over X , then we must somehow eliminate X ; the method depends on whether X is a terminal or a nonterminal symbol.

CR3. If $X \in V_T$, then delete X , adding to COST the deletion cost of X . We then have a configuration

$$\alpha'_* \rightarrow q \ ? \ p \xrightarrow{\beta_*} \dots \quad (3'')$$

CR3.1 If $\mathcal{D}(q) \cap \mathcal{P} \neq \Phi$, then return (3'') as a repair with associated cost, COST.

CR3.2 If $\mathcal{D}(q) \cap \mathcal{P} = \Phi$, then repeat the Correction Phase beginning at CR1, using the configuration (3'') instead of (3).

CR4. If $X \in V_N$, then X resulted from the reduction of some portion of input, say $y_1 \dots y_{m-1}y_m$. First reparse $y_1 \dots y_{m-1}y_m$, stopping just prior to the reduction of $\dots y_m$, yielding the new configuration

$$\alpha'_* \rightarrow q' \xrightarrow{\alpha''_*} q'' \xrightarrow{y_m} q''' \ ? \ p \xrightarrow{\beta_*} \dots \quad (3''')$$

Then repeat the Correction Phase, beginning at CR1, using the configuration (3''') instead of (3).

6. Implementation and Examples

The Error Recovery Scheme was implemented with an LR(1) parser for a small language whose partial syntax, G1, is given in Appendix A. The parser has 356 states. The sample programs were all run with a constant cost of 2 for the insertion or deletion of any terminal symbol. The threshold for insertions and deletions was set to 5.

In the remainder of this section, we briefly summarize some examples. The examples are illustrative of the behavior of the Error Recovery Scheme and demonstrate its powerful correction ability. We discuss the examples further in the succeeding section. Additional examples and details may be found in [12].

Example 1. The program

... READ $A \ B[20]$ WRITE A ; GOTO ...

is indicated in the configuration.

... READ (input-list)(identifier)[(expression)]

? WRITE A ; GOTO ...

⁵ Let $\mathcal{Q} \subseteq Q$, $\mathcal{P} \subseteq Q$, $Z \in V$. By $\mathcal{Q} \xrightarrow{Z} \mathcal{P}$, we mean $q \xrightarrow{Z} p$ for some $q \in \mathcal{Q}$, $p \in \mathcal{P}$. In addition we define $\mathcal{A}(\mathcal{P}) = \bigcup_{p \in \mathcal{P}} \mathcal{A}(p)$.

The Condensation Phase produces the correction candidate

```
... READ <input-list><identifier>[(expression)]
                                     (1)
                                     ? <statement>; GOTO ...
```

on account of the attempted reductions

```
<statement> ::= <identifier>:<statement>
```

and

```
<statement-list> ::= <statement-list><statement>;
```

In the first case, the Correction Phase proceeds: CR1, CR2, CR3 (delete], COST = 2), CR3.2, CR1, CR2, CR4 (expand <expression> to 20), CR1, CR2, CR3 (delete 20, COST = 4), CR3.2, CR1, CR2, CR3 (delete [, COST = 6), resulting in failure due to excessive deletion. In the second case, the Correction Phase proceeds CR1 (insert ;, COST = 2) providing the repaired configuration

```
... READ <input-list><identifier>[(expression)];
                                     <statement>; GOTO ...
```

Example 2. The program segment

```
... X := Y THEN GOTO L ELSE Z := 1; GOTO ...
```

is indicated in the configuration

```
... <statement-list><leftpart><identifier>
                                     ? THEN GOTO L ELSE Z := 1; GOTO ...
```

The Condensation Phase develops the correction candidate

```
... <statement-list><leftpart><identifier>
                                     ? THEN <statement> ELSE <statement>; GOTO ...
```

on account of the attempted reduction

```
<statement> ::= IF <boolean-expr>
                                     THEN <statement> ELSE <statement>
```

The Correction Phase proceeds: CR1, CR2 (backup over <identifier>), CR1, CR2, CR4 (expand <leftpart> to <identifier> :=), CR1, CR2 (backup over =), CR1, CR2, CR3 (delete :, COST = 2), CR3.2, CR1, CR2 (backup over <identifier>), CR1 (insert IF, COST = 4) providing the repaired configuration

```
... <statement-list> IF <identifier>
    = <identifier> THEN <statement> ELSE <statement>; GOTO ...
```

Example 3. The program segment

```
... BEGIN X := Y; Y := Z; WRITE X Y;
```

is indicated in the configuration

```
... <statement-list> BEGIN <statement-list><statement>; ?.
```

The Condensation Phase attempts the reduction

```
<program> ::= <declaration-list><statement-list>.
```

yielding a correction candidate. The Correction Phase proceeds: CR1 (the insertion of END is insufficient, since the pair END; is needed), CR2 (backup over ;), CR1, CR2 (backup over <statement>), CR1, CR2 (backup

over <statement-list>), CR1, CR2, CR3 (delete BEGIN, COST = 2), CR3.1, providing the repaired configuration

```
... <statement-list><statement-list><statement>;
```

Example 4. The program segment

```
... X := Y: A := B; GOTO ...
```

is indicated in the configuration

```
... <statement-list><leftpart><identifier>: ? A := B; GOTO ...
```

The Condensation Phase produces the correction candidate

```
... <statement-list><leftpart><identifier>: ? <assignment>; GOTO ...
```

on account of the attempted reduction

```
<assignment> ::= <leftpart><assignment>
```

as well as the correction candidate

```
... <statement-list><leftpart><identifier>: ? <statement>; GOTO ...
```

on account of the attempted reductions

```
<statement> ::= IF <boolean-expr>
                                     THEN <statement> ELSE <statement>
```

and

```
<statement> ::= <identifier>:<statement>
```

In the first case, the Correction Phase proceeds: CR1 (insert =, COST = 2) providing the repaired configuration

```
... <statement-list><leftpart><identifier> := <assignment>; GOTO ...
```

In the second case, the Correction Phase quickly exceeds the threshold cost with deletions.

In the third case, the Correction Phase proceeds: CR1, CR2 (backup over :), CR1, CR2 (backup over <identifier>), CR1, CR2, CR4 (expand <leftpart> to <identifier> :=), CR1, CR2, CR3 (delete =, COST = 2), CR3.1, providing the repaired configuration

```
... <statement-list><identifier>:<identifier>:<statement>; GOTO ...
```

Example 5. The program segment

```
... READ A X := Y ...
```

is indicated in the configuration

```
... <statement-list> READ <input-list><identifier> ? := Y ...
```

The Condensation Phase considers this a correction candidate on account of the attempted reductions

```
<leftpart> ::= <identifier> :=
```

and

```
<leftpart> ::= <subscripted-var> :=
```

The second alternative quickly dies out, but the first proceeds: CR1, CR2 (backup over <identifier>), CR1 (insert ;, COST = 2) providing the repaired configuration

```
... <statement-list> READ <input-list>; <identifier> := Y ...
```

7. Discussion

The Error Recovery Scheme has been presented in a generalized form for the sake of notational clarity, with little attention paid to details of the space and time required. But, with so much condensation and reparsing indicated, it is natural to wonder whether the scheme requires, for example, additional pushdown stacks, additional copies of the remaining input, or additional copies of the existing stack. It is clear that since the Correction Phase, given (2), may destroy α ,⁶ it will be necessary to save a copy of α for subsequent Correction Phase attempts. (Note that α is the same for all Correction Candidates.) All we need do to provide such a backup is duplicate α on the parsing stack. Next, we notice that it is not really necessary to pass the entire correction candidate (2) on to the Correction Phase, but only $\xrightarrow{\alpha_*} q^? p$. Similarly, the only identification needed for holding candidate (2') is p , from which (2') can be reconstructed, if needed. This results in some duplicate parsing in case it becomes necessary to revive the holding candidates, but the hope is that such revival will be unnecessary. These observations suggest that the strategy should be to continue each of the indicated parses in turn, and record not the resulting configurations (2) or (2'), but merely which $p \in \mathcal{P}$ led to correction candidates and which led to holding candidates. The parsing can be done using the existing stack, and no extra copies of the remaining input are needed. The resulting recording of each $p \in \mathcal{P}$ requires slight additional space (which can be sharply bounded by a constant function of the LR parser). We have found it useful to prioritize the correction candidates based on the number of terminal symbols consumed. The expectation is that the parse which successfully consumes many symbols before becoming a correction candidate is likely to provide a low-cost repair, thus bounding the cost of subsequent repair attempts. In the same way, we prioritize the holding candidates, expecting that the holding candidates whose second errors occur very close to the first are probably the result of incorrect condensation attempts and should be the last ones considered as apparent multiple error configurations.

An interesting part of the algorithm is the expansion and reparsing that occurs in step CR4. A natural question is why stop the reparse just prior to reducing the terminal symbol y_m , rather than just prior to the reduction to X . The reason is that if X were expanded to a string which ended in a nonterminal symbol, Y , it is possible that backup across Y could not be accomplished, and that CR4 would then find it necessary to expand Y . This repeated expansion of the rightmost nonterminal would continue until either a nonterminal is obtained which admits backup, or the final nonterminal is expanded to $\dots y_m$ (yielding exactly the configuration (3''') of CR4).

⁶ We remind the reader that in practice the stack does not really contain α , but the sequence of states q_0, \dots, q for which $q_0 \xrightarrow{\alpha_*} q$.

We make the following observations. If there is some intermediate string, γ , between X and $\alpha''y_m$ (which of necessity ends in a nonterminal) which admits backup, then the string $\alpha''y_m$ will also admit backup. Since $\alpha''y_m$ may be longer than γ , backup through γ may proceed faster than backup through $\alpha''y_m$. If it were possible to directly expand X by one step, then the alternate approach would indeed be quite attractive. However, there may be many choices for expanding X , and although examination of $y_1 \dots y_m$ might narrow the possibilities,⁷ we must generally start with $y_1 \dots y_m$ and work backwards to X . The disadvantage of the approach is that if γ is derived from X in k steps, then the steps "CR1, CR2, CR4 (expand one step by reparsing)" are repeated k times before γ is obtained. Moreover, if the correct repair involves the deletion of y_m , then clearly much time is wasted in examining the strings at each step from X to $\alpha''y_m$. These problems are, of course, aggravated in the case of an SLR parser, in which many incorrect reductions may have to be undone.

Although our technique thus far appears promising, there still remain a number of questions to be answered. For example, it is clear that in step CR1, locating $x \in V_T$ for which $\mathcal{D}(q) \xrightarrow{x} \mathcal{A}(\mathcal{P})$ can be implemented (automatically) by expanding the so-called GOTO table of an LR parser. But the resulting table is quite large, and we wonder whether it can be compressed using standard techniques. We suspect that Joliat's technique [8] will yield good results.

Examination of Example 4 from the previous section provides another interesting question. Should the repaired program read

... $X := Y := A := B$; GOTO ...

or

... $X:Y:A := B$; GOTO ... ?

With the given insertion/deletion costs, each repair has $\text{COST} = 2$. The example indicates that semantic information can be usefully employed in choosing the repair. For example, if either X or Y were "declared variables," then the first repair might have been chosen. However, if either X or Y had previously appeared in a GOTO statement, then the second repair would be correct.

Example 5 raises an interesting question concerning when the Error Recovery Scheme should be halted. In the example, the scheme halts after having provided one correction:

... READ A ; $X := Y$...

However, were it allowed to search for another solution, it would proceed: CR1, CR2 (backup over (identifier)), CR1, and after presenting the above solution, continue: CR4 (expand (input-list) to A), CR1 (insert [, providing the repaired configuration

... (statement-list) READ $A[X := Y]$...

⁷ For example, if the grammar possessed an LL-like property.

Clearly, if the next input symbol is], this second repair is correct, whereas the first is not. It is interesting to follow the actions of the Error Recovery Scheme when the wrong repair is initially chosen. In one case, the Error Recovery Scheme retreats and undoes its wrong repair; in the other, it attempts to coverup its first error by making a second wrong repair. This coverup effect can be elided in a very interesting way. We can simply distinguish between programmer-supplied terminal symbols and Error Recovery Scheme-supplied terminal symbols.

Whereas the deletion costs for the former are positive, those for the latter would be negative, i.e., the scheme would encourage deletion of previously inserted symbols. We do not know whether the coverup effect is serious enough or widespread enough to justify such an approach. Note that in the present example, choice of the first repair may lead to coverup, whereas choice of the second repair will never lead to coverup. Thus, the potential coverup can be avoided in this case by assigning to [an insertion cost that is lower than that for ;. The determination of insertion and deletion costs is sometimes aided by considerations such as avoiding coverup. But there remains the more general problem of developing reliable guidelines for determining insertion and deletion costs.

Finally, we have noted that the repaired configuration may not be suitable for immediate resumption of normal parsing. Indeed, the resulting configuration is almost never a so-called canonical sentential form (obtainable in a strictly left-to-right parse). However, it is often a noncanonical sentential form. In such cases, the use of noncanonical parsing [16] would permit immediate resumption of normal operation. There are, however, cases in which the repaired configuration is not a sentential form at all (as ... <statement-list><statement-list><statement>; of Example 3). We are attempting to develop techniques for characterizing such forms, and hope to report on those developments soon.

Appendix A: A Grammar, G1, for a minilanguage

<program>	::= <declaration-list><statement-list>.
<declaration-list>	::= <identifier>
	::= <identifier>[(digits)]
	::= <declaration-list><identifier>
	::= <declaration-list><identifier>[(digits)]
<statement-list>	::= <statement>;
	::= <statement-list><statement>;
<statement>	::= GOTO <identifier>
	::= READ <input-list>
	::= WRITE <output-list>
	::= IF <boolean-expr> THEN <statement>
	ELSE <statement>
	::= <identifier>:<statement>
	::= BEGIN <statement-list> END
	::= <assignment>
<input-list>	::= <variable>
	::= <input-list><variable>
<output-list>	::= <variable>
	::= <character>
	::= <output-list><variable>
	::= <output-list><character>

<boolean-expr>	::= <expression><relational-op><expression>
<relational-op>	::= <
	::= =
<assignment>	::= <left-part><assignment>
	::= <left-part><expression>
<left-part>	::= <identifier> :=
	::= <subscripted-var> :=
<subscripted-var>	::= <identifier>[(expression)]
	::= <identifier>[(assignment)]
<variable>	::= <identifier>
	::= <subscripted-var>

Acknowledgments. We are grateful to the referees for their helpful comments.

Received October, 1976; revised June, 1977

References

1. Aho, A.V., and Johnson, S.C. LR parsing. *Computing Surveys* 6, 2 (June 1974), 99-124.
2. Aho, A.V., and Ullman, J.D. *The Theory of Parsing Translation and Compiling, Vol. 1: Parsing*. Prentice-Hall, Englewood Cliffs, N.J., 1972.
3. DeRemer, F.L. Simple LR(k) Grammars. *Comm. ACM* 14, 7 (July 1971), 453-460.
4. Graham, S.L., and Rhodes, S.P. Practical syntactic error recovery in compilers. *Conf. Rec. ACM Symp. on the Principles of Programming Languages*, Boston, Mass., Oct. 1973, pp. 53-58.
5. Graham, S.L., and Rhodes, S.P. Practical syntactic error recovery. *Comm. ACM* 18, (Nov. 1975), 639-650.
6. Irons, E.T. An error-correcting parse algorithm. *Comm. ACM* 6, 11 (Nov. 1963), 660-673.
7. James, L.R. A syntax directed error recovery method. Master's Th., Tech. Rep. CSRG-13, Comptr. Syst. Res. Group, U. of Toronto, Toronto, Ont., Canada, May 1972.
8. Joliat, M.L. On the reduced matrix representation of LR(k) parser tables. Ph.D. Th. Tech. Rep. CSRG-28, Comptr. Syst. Res. Group, U. of Toronto, Toronto, Ont., Canada, Oct. 1973.
9. Knuth, D.E. On the translation of languages from left to right. *Inform. and Control* 8 (1965), 607-639.
10. LaFrance, J.E. Syntax-directed error recovery for compilers. Ph.D. Th. ILLIAC IV Doc. No. 249, Dept. Comptr. Sci., U. of Illinois, Urbana, Ill., 1971.
11. Leinius, R. Error detection and recovery for syntax directed compiler systems. Ph.D. Th., Comptr. Sci. Dept., U. of Wisconsin, Madison, Wis., 1970.
12. Levy, J.P. Automatic correction of syntax errors in programming languages. Ph.D. Th., Tech. Rep. TR71-116, Comptr. Sci. Dept., Cornell U., Ithaca, N.Y., 1971.
13. Modry, J.A. Syntactic error recovery for LR parsers. Master's Th. UIUCDCS-R-76-388, Dept. Comptr. Sci., U. of Illinois, Urbana, Ill., 1976.
14. Partridge, D. Heuristic methods in the analysis of program statements. Ph.D. Th., Dept. of Comptng. and Control, U. of London, London, England, Aug. 1972.
15. Rhodes S.P. Practical syntactic error recovery for programming languages. Ph.D. Th., Tech. Rep. No. 15, Comptr. Sci. Dept., U. of California, Berkeley, Calif., June 1973.
16. Szymanski, T.G., and Williams, J.H. Non-canonical parsing. *Proc. 14th Annual Symp. on Switching and Automata Theory*, Oct. 1973, pp. 122-129.
17. Tindall, M.H. An Interactive Compile-Time Diagnostic System, Ph.D. Thesis, UIUCDCS-R-75-748, Dept. Comptr. Sci., U. of Illinois, Urbana, Ill., Oct. 1975.
18. Wirth, N., and Weber, H. EULER, A generalization of ALGOL, and its formal definition, Pt I, *Comm. ACM* 9, 1 (Jan. 1966), 13-23; Pt II, 9, 2 (Feb. 1966), 89-99.