

Acknowledgments. We are very grateful to Rao Kosaraju, whose mastery of the theoretical results in structured programming is impeccable, and to Robert Taylor, for his helpful readings of the conclusions.

Received October 1974; revised December 1974

References

1. Allen, F.E., and Cocke, J. A catalogue of optimizing transformations. In Randall Rustin (Ed.), *Compiler Optimization*. 5th Courant Computer Science Symposium, Prentice-Hall, Englewood Cliffs, N.J., 1972, (pp. 1-30).
2. Ashcroft, E. and Manna, Z. The translation of "GOTO" programs to "WHILE" programs. Rep. No. STAN-CS-71-188, Comput. Sci. Dep., Stanford U., 1971.
3. Bochmann, G. V. Multiple exits from a loop without the GOTO. *Comm. ACM* 16, 7 (July, 1973) 443-444.
4. Böhm, C., and Jacopini, G. Flow diagrams, Turing machines and languages with only two formation rules. *Comm. ACM*, 9, 5 (May 1966), 366-371.
5. Bruno, J., and Steiglitz, K. The expression of algorithms by charts. *J. ACM*, 19, 3 (July 1972), 517-525.
6. Cooper, D.C. Some transformations and standard forms of graphs with applications to computer programs. In E. Dale and D. Michie (Eds.), *Machine Intelligence 2*. American Elsevier, New York, 1968, pp. 21-32.
7. Dijkstra, E. W. Notes on structured programming. In *Structured Programming*. W.J. Dahl, E.W. Dijkstra, and C.A.H. Hoare, Academic Press, New York, 1972, pp. 1-82.
8. Friedman, D., and Shapiro, S. A case for the while-until. *SIGPLAN Notices* (ACM newsletter) 9, 7 (July 1974), 7-14.
9. Gross, J.L., and Brainerd, W.S. *Fundamental Programming Concepts*. Harper and Row, New York, 1972.
10. Henderson, P., and Snowdon, R. An experiment in structured programming. *BIT* 12 (1972), 38-53.
11. Hoare, C.A.H., and Wirth, N. An axiomatic definition of the programming language PASCAL. *Acta Informatica* 2 (1973), 335-355.
12. Kernighan, B.W., and Plauger, P.J. *The Elements of Programming Style*. McGraw-Hill, New York, 1974.
13. Knuth, D.E. Structured programming with GOTO statements. *Computing Surveys*, 6 (Dec. 1974) 261-301.
14. Knuth, D.E., and Floyd, R.W. Notes on avoiding GO TO statements. Rep. No. CS-148, Comput. Sci. Dep. Stanford U., 1970.
15. Kosaraju, R. Analysis of structured programs *J. Comput. and Syst. Sci.*, 9, 3 (Dec. 1974), 232-255. Tech. Rep. No. 72-11, Elect. Eng. Dep., Johns Hopkins U. 1972.
16. Leavenworth, B. M. Programming with(out) the GOTO. *Proc. ACM Nat. Conf.*, 1972, pp. 782-786.
17. Ledgard, H. F. *Programming Proverbs*. Hayden Publishing Co., Rochelle Park, N.J., 1975.
18. McKeeman, W.M., Horning, J.J., and Wortman, D.B. *A Compiler Generator*. Prentice-Hall, Englewood Cliffs, N.J., 1970.
19. Mills, H.D. Mathematical foundations for structured programming. FSC 72-6012 Federal System Division, IBM Corp., Gaithersburg, Md., 1972.
20. Neely, P.M. On program control structure. *Proc. ACM Nat. Conf.* 1973, pp. 119-125.
21. Peterson, W.W., Kasami, T., and Tokura, N. On the capabilities of while, repeat, and exit statements. *Comm. ACM*, 16, 8 (Aug. 1973), 503-512.
22. Sites, R.L. Proving that computer programs terminate cleanly. Rep. No. STAN-CS-74-418, Comput. Sci. Dep., Stanford U., 1974.
23. Wirth, N. The programming language PASCAL. Revised report, Eidgenössische Technische Hochschule, Zurich, 1972.
24. Wirth, N. Program development by stepwise refinement. *Comm. ACM*, 14, 4 (Apr. 1971), 221-227.
25. Wulf, W.A., Russell, D.B., and Habermann, A.N. BLISS: A language for systems programming. *Comm. ACM*, 14, 12 (Dec. 1971), 780-790.
26. Zahn, C.T., A control statement for natural top-down structured programming. *Symp. on Prog. Lang.*, Paris, 1974.

Programming
Languages

B. Wegbreit
Editor

Practical Syntactic Error Recovery

Susan L. Graham
University of California, Berkeley
Steven P. Rhodes
Bell Laboratories, Greensboro

This paper describes a recovery scheme for syntax errors which provides automatically-generated high quality recovery with good diagnostic information at relatively low cost. Previous recovery techniques are summarized and empirical comparisons are made. Suggestions for further research on this topic conclude the paper.

Key Words and Phrases: syntax errors, error recovery, error correction, parsing, simple precedence, compilers, debugging

CR Categories: 4.12, 4.42, 5.23

1. Introduction

A substantial portion of any programmer's time is spent in debugging. One of the major services of every compiler ought to be to provide as much information as possible about compile-time errors in order to minimize the time required for debugging. Ideally, at compile time, the compiler should discover and report all syntax errors and those semantic errors detectable at that time. At the same time, no error messages should be generated for nonexistent errors. These

Copyright © 1975, Association for Computing Machinery, Inc. General permission to republish, but not for profit, all or part of this material is granted 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.

This research was supported in part by the National Science Foundation under grants GJ-474 and GJ-43318. A preliminary description of this work was presented at the ACM Symposium on Principles of Programming Languages [6].

Authors' addresses: Susan L. Graham, Computer Science Division, 577 Evans Hall, University of California, Berkeley, CA 94720; Steven P. Rhodes, Bell Laboratories, P.O. Box 21447, Greensboro, NC 27420.

spurious error detections and their associated error messages are usually engendered by an inappropriate action taken by the compiler to remove a previous error. Additionally, it is desirable to have diagnostic information about what seems to be wrong, rather than simply the information that an error occurred.

In this paper we consider the problem of recovering from syntax errors in a way that detects almost all errors, provides diagnostic information, and reports few nonexistent errors. We have attempted to provide a solution which depends to a great extent solely on the form of the syntax. Just as formal methods for syntax analysis have led to automatic parser construction (via parser-generators), formal methods for syntax error recovery enable us to have automatic construction of recovery capabilities. Not only does this free the implementor from the extra task of designing and programming recovery routines, but it also minimizes the biases which cause handwritten recovery routines to falter badly on unanticipated errors.

In order that the recovery techniques be usable in production compilers, we consider it important that the error recovery not degrade the parsing speed intolerably. The approach to be presented has the property that no overhead for error recovery is incurred in parsing correct programs or correct portions of programs with errors. The recovery routines are invoked only when an error is detected. They restore the parser to a valid configuration and return control to the parser. The recovery techniques can be carried out reasonably rapidly, permitting a small amount of recovery time per error.

In our view, the most meaningful way to evaluate the techniques presented here was to try them. Our experiments indicate that our recovery techniques work quite well in practice, particularly as compared with the recovery actions taken by other compilers whose designers claim high-quality recovery as one of their achievements. Many errors can be corrected locally, enabling our recovery techniques to handle densely occurring errors without skipping portions of the source text.

Of course, the recovery action taken does not necessarily correspond to the programmer's intention. Any given piece of syntactically invalid source text may mean different things for different computations. For example, consider the incorrect Algol statement

$$I := 1 - (((N * M) - (I * J)) / 2).$$

Among the many equally plausible correct versions of the above statement are the following two (different) statements.

$$I := 1 - (((N * M) - (I * J)) / 2)$$

$$I := 1 - ((N * M) - (I * J)) / 2$$

Furthermore, different classes of programmers (for instance, novice programmers versus experienced programmers) may make different kinds of errors. Consider

the syntactically invalid Fortran IV statement

READ532ABLE

given by E. James and Partridge [9]. A former Fortran II programmer would probably have meant

READ 532,ABLE

whereas a Fortran IV programmer would probably have meant

READ(5,32)ABLE

However, our techniques usually make plausible changes to incorrect programs. Even if the corrections are not what the programmer intended, they usually serve to show the programmer what error has been made.

The paper is organized as follows. After presenting our notation and giving some general definitions, we briefly survey some of the other approaches to error recovery that have been proposed and, in some cases, used. We then give an overall explanation of our recovery technique, followed by a more detailed description of the version we implemented and a summary of our empirical results. This is followed by a discussion of possible extensions to the scheme and some concluding remarks.

Many of the results present in this paper are contained in the second author's Ph.D. dissertation [17].

2. Definitions and Notation

A (*context-free*) *grammar* is a 4-tuple $G = (V, \Sigma, P, S)$, where $\Sigma \subseteq V$ is a finite set of *terminal symbols*, $N = V - \Sigma$ is a finite nonempty set of *nonterminal symbols*, P is a finite set of *rules* or *productions*, $X \rightarrow x$, where $X \in N$ and $x \in V^*$,¹ and $S \in N$ is the *initial symbol*. For any rule $X \rightarrow x$, X is termed the left-hand side (abbreviated LHS) and x is the right-hand side (RHS).

As usual, with respect to a grammar $G = (V, \Sigma, P, S)$ we define the relation \Rightarrow on $V^* \times V^*$ such that for any $a \in V^*$, $b \in V^*$, $a \Rightarrow b$ if and only if there exist $U \in N$, $\sigma, \pi, u \in V^*$, and $U \rightarrow u$ in P such that $a = \sigma U \pi$ and $b = \sigma u \pi$. We represent by \Rightarrow^+ (\Rightarrow^*) the transitive closure (reflexive-transitive closure) of \Rightarrow . For any $n \geq 0$, $a_i \in V^*$ where $0 \leq i \leq n$, we say that the sequence $a_0 \Rightarrow a_1 \Rightarrow \dots \Rightarrow a_n$ is a *derivation of a_n from a_0 of length n in G* . If $a_0 \Rightarrow a_1 \Rightarrow \dots \Rightarrow a_n$ where for $0 \leq i < n$, $a_i = \sigma_i U_i \pi_i$ and $a_{i+1} = \sigma_i u_i \pi_i$ for some $\sigma_i \in V^*$, $\pi_i \in \Sigma^*$, and $U_i \rightarrow u_i$ in P , the derivation is a *rightmost derivation*. A sequence of symbols u is a *sentential form* if $S \Rightarrow^* u$. The sentential form is *rightmost* if there is a rightmost derivation such that $S \Rightarrow^* u$. If $u \in \Sigma^*$ then u is a *sentence*. The *language* $L(G)$

¹ For any set of symbols V , V^* denotes the set of all finite length sequences of symbols from V and $V^+ = V^* - \{\lambda\}$ where λ is the empty sequence.

defined (or generated) by G is the set of all sentences. Thus $L(G) = \{u \in \Sigma^* \mid S \Rightarrow^* u\}$.

We refer to the process of reconstructing a derivation, given a sequence of terminal symbols and a grammar, as *parsing*. Given a sentential form $\sigma u \pi$, $\sigma, u, \pi \in V^*$, and a rule $U \rightarrow u$, the transition from $\sigma u \pi$ to $\sigma U \pi$ obtained by substituting U for u is called a *reduction*.

We restrict consideration to the parsing of deterministic context-free languages. A *bottom-up parsing method* is an algorithm which reconstructs derivations in reverse order by making a sequence of reductions from the *input*—a sequence of symbols from Σ^* —to the initial symbol. (That is, if $a_0 \Rightarrow a_1 \Rightarrow \dots \Rightarrow a_n$ is a derivation, and $a_n \in \Sigma^*$, then a_n is the input and the parser reduces a_n to a_{n-1} , a_{n-1} to a_{n-2} , etc. until it reaches a_0 .) The input is scanned symbol by symbol. We refer to the symbol being scanned at any given time as the *current input symbol*. The (bottom-up) parser uses a pushdown store called the *parsing stack* which we represent as a sequence of symbols with the base of the stack at the left and the top of the stack at the right. There is a base of the stack symbol designated ϵ and an end of file symbol designated $\$$, where ϵ , $\$$ are not in the vocabulary of the grammar. The input is terminated by $\$$. The parsing stack contains symbols of the vocabulary and possibly additional information, depending on the parsing method being used. Parsing actions consist either of stacking the current input symbol and advancing the input, or of making a reduction of symbols at the top of the parsing stack.

Following Levy [13], we say that a parsing method has the *correct prefix property* if the existence of a syntax error is detected (in a single deterministic left-to-right scan of the input) as soon as the input scanned no longer forms a prefix of a sentence in the language being parsed. $LR(k)$ parsing and its variants ($SLR(k)$, $LALR(k)$, etc.) and top-down recursive descent have this property; the various precedence and bounded right context methods do not.²

3. Previous Methods of Error Recovery

Compiler writers have always had to deal with the problem of error recovery. Many of the techniques in use depend on the details of the programming language being compiled. Often, the compiler writer includes in the syntax analysis portion of the compiler, routines to take particular actions if particular error situations occur. Error recovery can be handled in this way even when parsing is table driven. The approach taken by Gries [7] and the PL/C implementers [4] is to include in the parsing table error actions based on the implementers' knowledge of common programming errors and appropriate recovery actions. This technique re-

quires a substantial amount of programming effort for the error recovery portion of the compiler. Furthermore, although such a system handles the expected errors reasonably well, it can fail badly on unanticipated errors. Another kind of language dependent error recovery, which is easier to implement in syntax directed compilers, is to augment the grammar by "error productions" (Wirth [20]). Again, unanticipated errors cause trouble. A more local approach is to have a list of possible local modifications and to use the first one that works (see, for example, Wirth [20]). Additionally one can have an implementer designed table of possible local modifications for each symbol or pair of symbols in error. (See, for example, Bauer et al. [2], and Peterson [16].) However, these techniques will not always succeed.

The oldest and simplest recovery technique that is essentially language independent is the so-called *panic mode*. In this scheme, when an error is detected, the input is advanced until one of a class of special symbols, such as a ";" or an *end* is located. (The specification of this class of symbols is language dependent but can be determined rather easily.) The parsing stack is then erased until the special symbol can legitimately follow the top of the parsing stack. This method is fast and requires a small amount of code, but the errors contained in that portion of the text which is skipped are not detected, thus possibly necessitating many additional computer runs to detect all of the errors in the user's program. In addition, little information is available about the nature of the error. If the parsing method being used is predictive, that is, it is possible to determine easily all the possible valid continuation symbols for the input read so far, then the input can be advanced until one of these symbols is encountered. The recovery scheme for LR parsing described by Leinius [12] and Peterson [16] and implemented by L. James [10] is a more sophisticated version of this technique. Alternatively, the predictive capability can be used to insert one or more symbols so that the next input symbol can legitimately follow. This technique is presented in Irons [8].

Another early automatic technique is that of spelling correction, which appears in CORC [5] and was later developed further by Morgan [15]. It is normally used in conjunction with other recovery techniques.

An approach taken by Levy [13] and La France [11] is to choose one of an automatically generated set of corrections by simultaneously carrying out parses for each possibility. If, as in the Levy method, one continues multiple parsing for an unbounded number of steps, the ensuing combinatorial explosion in space and time makes the technique very impractical. Consequently, La France bounds the amount of multiplicity. This improves efficiency, but can yield insufficient information in some cases.

Another language-independent recovery technique investigated recently is to determine the minimum

² Descriptions of these parsing methods can be found in [1].

number of insertions, deletions, and substitutions of symbols which transform the entire incorrect input into a valid sentence of the language being analyzed. Studies by Lyon [14], Peterson [16], and Teitelbaum [18] have shown that for general context-free parsing, the "minimal distance" of an input sequence can be determined in time of order n^3 where n is the length of the input. However, these techniques do not provide linear-time minimal distance correction for linear-time parsing methods. More important, as pointed out by Levy [13], minimal distance corrections may not be the best corrections according to the programmer.

The previous work closest to our own is that of Leinius [12] and Levy [13]. Leinius proposes an error recovery scheme for simple precedence parsing.³ He introduces the notion of "phrase-level" recovery; namely, recovery actions that have the effect of reducing the parsing stack. He recognizes that if the parsing stack followed by the current input symbol contains any sequence of symbols $ab\alpha cd$, where a, b, c, d are single symbols, α is a sequence of symbols, $a < b$, $c > d$, and bac contains an error, then bac can be replaced by any "locally correct" nonterminal symbol as a recovery action. His recovery algorithm finds successively larger sequences enclosed by $<$ and $>$ (by looking back in the stack or parsing ahead in the input) until one is found for which there is a unique locally correct nonterminal to replace it.⁴

Levy attempts to find a theoretical basis for error correction in all deterministic context-free parsing methods having the correct prefix property. His method includes a backward move on the input to determine the entire left context of the error discovery point that could contain the error and then parallel parses from the beginning of the left context to pursue all possible minimal distance corrections of a fixed bounded distance. This method is admittedly impractical and Levy proposes some heuristics to improve its efficiency.

Unfortunately, neither Leinius' simple precedence recovery nor Levy's method appears to have been tested empirically. However, both methods contain important ideas, especially the attempt to provide better automatic language-independent recovery or correction and the use of context in this process.

4. General Description of Graham-Rhodes Method

We first describe our recovery method as incorporated in any bottom-up, no back-up parser, suppressing for the moment the details of how the parser works (i.e. how parsing decisions are made and how errors are detected).

³ See Section 5 for an explanation of this parsing method.

⁴ The sequence enclosed by $<$ and $>$ is not examined.

⁵ A second diagnostic message is issued only if there appears to be a second error in the source text.

The error recovery routines are invoked when a syntax error is detected by the parser. Control is returned to the parser when the error state has been removed. In view of the efficiency constraints placed on the recovery method, input which has been scanned is not retained by the parsing program. Consequently, the error state is removed by making modifications to the parsing stack and possibly to the remaining input, but not to the already scanned input.

Immediately after an error is detected, most recovery strategies consider what change to the parsing stack and/or the input would recover from the error. In our method, we first attempt to analyze the context in which the error occurs. That is, the *correction phase* of the recovery is preceded by a *condensation phase* which condenses the surrounding context.

In the condensation phase, an attempt is first made to make further reductions on the stack, preceding the point of error detection. We refer to this attempt as the *backward move*. The *forward move* is an attempt to parse the input just beyond the point of error detection. The forward move will terminate either because a second error is detected further on in the input or, more likely, because the only possible next parsing action is a reduction involving that part of the stack containing the detected error.⁵

We illustrate the condensation phase by several examples. All the examples use the grammar given in the Appendix for an Algol-like syntax. The point in the program at which the error is detected is indicated by " \uparrow "; the corresponding point in the parsing stack is designated by "?".

Example 1.

```

:
M := Q - 3;
I = 2*(M-P) then K := 1 else M := 1;
↑

```

The most probable error is that there is a missing **if** preceding I . The error is detected when the $=$ is "seen" by the parser. At that point, the stack has the form

$\notin \langle \text{blockbody} \rangle ? \langle \text{variable} \rangle$

and the current input symbol is $=$.

No reductions are made by the backward move. The forward move reduces $I = 2*(M-P)$ to $\langle \text{expression} \rangle$ and then terminates because no further reductions which include $\langle \text{expression} \rangle$ or **then** are possible. In our experimental recovery system, the stack then has the form

$\notin \langle \text{blockbody} \rangle ? \langle \text{expression} \rangle \text{ then}$

and the current input symbol is K . (In some other bottom-up parsing methods, the current input symbol would be **then**.) Notice that there is no fixed a priori bound on the amount of input read during the forward move. By allowing the parser to determine the look-ahead, the recovery routine can "see" the symbol **then**

before any correction is made. This contributes considerably to the power of the method.

Example 2.

```

:
X := I J;
      ↑

```

The most probable error is a missing operator between *I* and *J*. The error is detected when the *J* is “seen” by the parser. At that point, the parsing stack has the form

⊢ *<blockbody>* *<variable>* := *<identifier>*?

and the current input symbol is *J*. In this example, the backward move reduces *<variable>* := *<identifier>* to *<statement>* and the forward move reduces *J* to *<expression>*, leaving the parsing stack in the form

⊢ *<blockbody>* *<statement>* ? *<expression>*

where the current input is ; .

Example 3.

```

:
write (begin I := 3 end);
      ↑

```

This is legal in so-called expression languages but is invalid in our Algol dialect. The error is detected when the **begin** is “seen” by the parser. The parsing stack has the form

⊢ *<blockbody>* *<procedure id>* (?

and the current input symbol is **begin**. The backward move reduces *<procedure id>* (to *<procedure head>*. The forward move reduces **begin I := 3 end** to *<blockbody>* *<statement>* **end**, leaving the parsing stack in the form

⊢ *<blockbody>* *<procedure head>*? *<blockbody>* *<statement>* **end**

where the current input symbol is “)”. Since there appears to be another error,⁶ a second backward move reduces *<blockbody>* *<statement>* **end** to *<statement>*.

The backward and forward moves are an attempt to summarize the context surrounding the point of error detection. The forward move provides, in effect, an unbounded lookahead. The purpose of the correction phase is to change the condensed parsing stack so that the error situation is corrected and the parsing stack contains a sequence of symbols that could occur in the parse of a sentence in the language. (In fact, depending on the parsing method, the correction phase may insure only that in the vicinity of the error the parsing stack is legal.)

Since we wish to use as much of the context of the error as can be efficiently exploited, the correction phase considers changes to sequences of symbols, rather than isolated changes to single symbols. The

⁶ **end** cannot be followed by **)** in a legal input.

⁷ In the case of a tie, either some tie-breaking rule can be invoked or the selection can be made arbitrarily.

implementer can trade quality of recovery for efficiency in determining how this correction is done. The idea is to change the parsing stack, at the point of error, to an RHS of the grammar, or to one or more prefixes of RHS's, which “fit in” in the sense that they can legitimately occur in the given context.

In general, there will be more than one possible change that appears locally to correct the error. For instance, in Example 1, after the condensation phase, the stack has the form

⊢ *<blockbody>* ? *<expression>* **then**

and the current input symbol is *K*. One correction that might be made would be to replace the symbols *<expression>* **then** by *<statement>* ; . This change appears locally to work, since

<blockbody> *<statement>* ;

is the RHS of the production, and could be followed by *<identifier>* in a legal input. Alternatively, **if** could be inserted before *<expression>*, since **if** can follow *<blockbody>* and

if *<expression>* **then**

is the RHS of a rule.

In order to provide helpful diagnostic information to the programmer, as well as to increase the likelihood that, in the absence of a more global analysis, the change really corrects the error, it is necessary to make some effort to choose the “best” correction. The way this is done is to determine which of the possible locally correct changes has the “closest fit”; that is, which change requires a minimum of symbol by symbol modification of the parsing stack. A *weighted* minimum distance measure is used. In order to compute how close a given RHS is to one of the candidates for change, two vectors *I* and *D* are used. For each symbol in the grammar, the *I* vector contains the cost of inserting that symbol anywhere in the stack and the *D* vector gives the cost of deleting that symbol anywhere in the stack. The closest fit is then defined to be the match with the minimum cost. As an example of the cost computation, consider yet again Example 1. The cost of changing *<expression>* **then** to *<statement>* ; is *D(<expression>)* + *D(then)* + *I(<statement>)* + *I(;*, whereas the cost of inserting **if** before *<expression>* is *I(if)*. Notice that Example 2 can be corrected at a cost of *D(<expression>)*.

In Example 3, the form of the stack after the condensation phase is

⊢ *<blockbody>* *<procedure head>* ? *<statement>*

and the current input symbol is “)”. One way to correct this error is to replace *<statement>* by *<expression>*, at a cost of *D(<statement>)* + *I(<expression>)*.

After the pattern matching process has determined the cost of the changes by using the cost vectors, the minimum cost change is made.⁷ Control then returns

to the parser. However, in the unlikely event that the minimum cost is greater than a fixed a priori maximum, a form of the panic mode is used. The assumption in that case is that the change, although locally correct, is so bizarre that it is probably wrong.

Clearly, the change which is selected depends on the values of I and D . These cost functions, in effect, indicate the relative likelihood that each grammatical symbol is intentional if it occurs in the input text and is unintended if it does not occur. There are a variety of heuristics which can be used in selecting the costs in order to improve the quality of the recovery. For example, brackets (**begin**, **end**, $(,)$, etc.) and the non-terminals generating them ($\langle \text{blockhead} \rangle$, $\langle \text{blockbody} \rangle$, etc.) should have relatively high I and D values and long "reserved words" should have high deletion costs. Using these rules, the cost vectors can be generated mechanically. Alternatively, values for I and D can be supplied by the implementer. This allows him or her to incorporate language-dependent criteria about the use of the implemented language and to "tune" the recovery system to a particular user community.

An addition to the cost computation which we found to be very useful is to include a cost function R which assigns costs to the replacement of one symbol by another. (The function values are normally lower than the corresponding I and D costs.) The replacement function can be used by the implementer for additional tuning and for introducing such factors as lexical similarity. For example, there can be a relatively low replacement cost of $=$ by $:=$ or of certain reserved words by $\langle \text{identifier} \rangle$. Of course, introducing R in the cost computation increases both the time to compute costs and the amount of compiler code for recovery.

5. Error Recovery for Precedence Parsers

In order to explain in more detail how the recovery method works, it is necessary to specify, for a given parsing method, how errors are detected and what is known about the parsing stack when an error is discovered, how the condensation phase is carried out, and how the set of possible changes is determined prior to the cost computation.

We initially developed this recovery method for simple precedence parsing, for which it is particularly well suited. We now discuss more precisely the way in which the various aspects of the recovery are carried out in that parsing method. Subsequently, we consider the incorporation of this approach to error recovery in other parsing methods.

First, we briefly review simple precedence parsing for the reader.

For any grammar $G = (V, \Sigma, P, S)$, simple precedence relations $<, \doteq, >$ are defined for all $(A, B) \in V \times V$ by:

$A \doteq B$ if for some $\sigma, \pi \in V^*$, P contains a rule $U \rightarrow \sigma AB\pi$

$A < B$ if for some $\sigma, \pi, \alpha \in V^*$, $Y \in N$, P contains a rule $U \rightarrow \sigma AY\pi$ and $Y \Rightarrow^+ B\alpha$

$A > B$ if for some $\sigma, \pi, \alpha, \gamma \in V^*$, $X \in N$, $y \in V$, P contains a rule $U \rightarrow \sigma XY\pi$ and $X \Rightarrow^+ \gamma A$ and $Y \Rightarrow^* B\alpha$.

A grammar $G = (V, \Sigma, P, S)$ is a *simple precedence grammar* [19] if: (a) For all $(A, B) \in V \times V$, at most one precedence relation is satisfied. (b) P contains no rule with RHS λ . (c) No two rules in P have the same RHS. (d) With respect to G , there is no rightmost derivation $S \Rightarrow^+ S$.

We extend the precedence relations to the end-markers ϕ and $\$$ by the rules that for every $X \in V$, if there is some $\sigma \in V^*$ such that $S \Rightarrow^+ X\sigma$, then $\phi < X$ and if there is some $\sigma \in V^*$ such that $S \Rightarrow^+ \sigma X$, then $X > \$$.

Initially, the parsing stack contains only ϕ . For an input string which is contained in $L(G)$, the parser works in the following way (excluding the output steps, semantic routines, etc.):

Step 1. Read the next input symbol.

Step 2. If the precedence relation between the symbol at the top of the parsing stack and the input symbol is $<$ or \doteq then stack the input symbol and go to Step 1.

Step 3. If the input symbol is $\$$ and the contents of the stack are ϕS then exit.

Step 4. (Otherwise the precedence relation between the top stack symbol and the input symbol is $>$.) Scan the stack from right to left until the first instance in which a symbol (call it A) and the symbol to the right of it in the stack have the precedence relation $<$.

Step 5. Find the rule having as RHS the sequence of symbols to the right of A on the parsing stack. Replace the symbols to the right of A by the LHS of that rule and go to Step 2.

In the usual precedence parser, errors are detected in one of two ways. The first occurs when there is no precedence relation between the top of the parsing stack and the incoming symbol (Step 2); this situation is usually referred to as a *character pair error*. In a typical Algol grammar, for instance, the string

$A := I \quad J := K;$

would have a character pair error between the I and the J since an identifier can never be followed by an identifier.

The second type of error is found when a potential RHS is detected using the precedence relations (Step 4), but it does not match any RHS of the grammar (Step 5). This type of error is normally referred to as a *reduction error*. A reduction error can arise in the following way. The following simple precedence grammar:

$S \rightarrow N\#$

$N \rightarrow D - D$

$D \rightarrow 1$

generates only the one sentence $1 - 1\#$. If the parser

for this grammar is given as input $1 - 1 - 1$, eventually Step 4 finds a potential RHS $D - D - D$, yet no rule of the grammar has this RHS.

The error detection capability of a simple precedence parser can be significantly improved, at the cost of a small increase in the running time of the parser. First, as suggested by Leinius [12], when a reduction is performed, a check can be made (Step 5) to see that A and the LHS to be stacked have precedence relation $<$ or \doteq , otherwise a *stackability error* occurs.

The second error detection extension is an improvement in the detection of reduction errors. In this second extension, which is original as far as we know, the system continually checks the top of the stack for prefixes of RHS's of rules of the grammar before it puts a symbol onto the stack. This can be done, for example, by having the production table sorted lexicographically by RHS's and having a pointer into this table which is advanced before each symbol of an RHS is stacked. All the RHS's with a common prefix will then be grouped together. When a new RHS is begun (i.e. when the top symbol of the stack and the symbol to be stacked have precedence relation $<$), the previous pointer value is saved and the pointer is set to the first production such that the leftmost symbol of the RHS is the symbol to be stacked. When the prefix at the top of the stack is to be continued (i.e. when the top symbol of the stack and the symbol to be stacked have precedence relation \doteq), the pointer is set to the first RHS having that prefix followed by the symbol to be stacked. When the prefix at the top of the stack should be an RHS (i.e. when the top stack symbol and the input symbol have precedence relation $>$), the pointer should be pointing to the rule with that RHS.

In the latter two cases, if there is no such RHS, a *nonvalid RHS error* is said to have occurred. Since the traditional parsing method must also search through the RHS's (Step 5), our method entails no increase in parsing time except for the inability to hash-address the production table in certain ways. The difference in our method is that it does the searching incrementally, whereas the usual method performs it all at one time.

Consider again the example $1 - 1 - 1$ given in the discussion of the usual precedence parser. In the system described in this paper, the parser detects a nonvalid RHS error on the second “-” since there is no production whose RHS begins with $D - D -$. Notice that a character pair error is just another kind of stackability error and a reduction error is one kind of nonvalid RHS error.

The advantage of the added detection capability for our error recovery method is that we can more accurately determine the likely location of the error. At the commencement of the condensation phase of recovery, that location is assumed to be either the point at which no precedence relation holds (possibly after condensation) or the point at which the previous contents of the stack do not form a prefix of an RHS (possibly after

condensation). More specifically, the error location is the point immediately preceding an RHS if the corresponding LHS causes a stackability error; the top of the stack if the corresponding LHS causes a nonvalid RHS error or in the case of a character-pair error. If the current input symbol causes a nonvalid RHS error, the incoming symbol is stacked, the input is advanced, and the new top of the stack is designated the location of the error.

The condensation phase is carried out easily. For the backward move, it is assumed that there is a precedence relation $>$ between the symbol immediately preceding and the symbol immediately following the point of error. Control is transferred to the parser, which makes all possible reductions (possibly none) preceding the point of error. For the forward move, the state of the parser is adjusted, by stacking the current input symbol if necessary, so that the location of the error is one symbol below the top of the stack. It is assumed that there is a precedence relation $<$ or \doteq between the symbol immediately preceding and the symbol immediately following the point of error, the nonvalid RHS check in the parser is turned off, and control is again returned to the parser. The forward move terminates either because of a new stackability error, or, more likely, because a $>$ is encountered, but the stack does not contain a valid RHS (because of the error which necessitated the forward move). If necessary, a second backward move is done from the second point of error by again assuming $>$ and returning control to the parser.

In the correction phase, we exploit the properties of the precedence relations. For the sake of efficiency, possible changes are restricted to replacing a portion of the condensed parsing stack by the RHS of a rule (not by a prefix). There are three sequences of symbols that are considered for correction. They are the sequence of symbols from the nearest $<$ to the left of the point of error up to the point of error, the sequence from that $<$ to the top of the stack, and the sequence from the point of error to the top of the stack. The restriction on possible replacements is not unreasonable, since it corresponds to the possibilities (1) that the precedence relation at point ? is $>$, (2) that the precedence relation at point ? is \doteq , and (3) that the precedence relation at point ? is $<$. (There is also an implicit assumption that the precedence relation at the top of the stack is $>$. In practice, this is very often the case. One can modify the correction phase so that prefixes of RHS's are also possible replacements, but the increase in computation is significant when measured against the empirical percentage of instances when such replacements are necessary.) Additionally, the possibility that any of the three stack sequences be deleted is considered. Thus if the number of rules in the grammar is n , at most $3(n + 1)$ changes are considered.

The next step is to reduce the set of possible changes to those which would enable the parser to continue;

that is, those which, in the immediate context, appear to correct the error. Essentially, a modification is locally correct if it does not create a stackability error or a nonvalid RHS error. More precisely,

Definition. Let $G = (V, \Sigma, P, S)$ be a simple precedence grammar. A production $X \rightarrow x$ is *locally correct* in the context⁸ (yA, B) ; $x, y \in V^*$; $X \in N$; $A, B \in V$ if the following three conditions are satisfied.

P1. $A < X$ or $A \doteq X$.

P2. $X < B$ or $X \doteq B$ or $X > B$.

P3. If $A \doteq X$ and $X > B$, then yAX is the RHS of some rule in P ; if $A < X$ and $X > B$ then X is the RHS of some rule in P ; otherwise, if $A \doteq X$ then yAX is a proper prefix of the RHS of some rule in P .

The *deletion* of a string x is *locally correct* in the context (yA, B) ; $y \in V^*$; $A, B \in V$ if the following two conditions are satisfied.

D1. $A < B$ or $A \doteq B$ or $A > B$.

D2. If $A \doteq B$ then yAB is the prefix of the RHS of some rule in P ; if $A > B$ then yA is the RHS of some rule in P .

Notice that since we consider replacing sequences of symbols only by RHS's, not by prefixes of RHS's, the tests for local correctness can be carried out rapidly, since it is the set of LHS's or nonterminals which are tested and the error checks are those done by the parser. In most cases, these tests eliminate a substantial portion of the possibilities (90–95 percent in our experiments).

The cost computations are made on those possible changes which are locally correct, as described in Section 4, and the minimum cost change is then made.

6. Experimental Results

We programmed a simple precedence parser in which we incorporated the described recovery techniques, together with a variety of experimental modifications that could be independently enabled or disabled. Our test of these recovery techniques was to compare our recovery with that of other compilers in general use. We implemented both an Algol subset and the full syntax of PASCAL [21]. We prepared a set of test programs with a wide variety of syntax errors. (The source of most of these errors was student programs; a few errors were deliberately designed to challenge any recovery system.) We then compared the results on our Algol programs with the result of submitting PL/I-equivalent programs to the PL/C compiler [3, 4]. This seemed particularly appropriate in view of the PL/C design objective of providing a maximum degree of diagnostic assistance. Output for the PASCAL programs was compared with the results of running the same programs on the PASCAL compiler produced by Wirth's group

⁸ In considering a candidate sequence for replacement or deletion, the left context is always the sequence of symbols starting with the nearest $<$ to the left.

(hereafter referred to as the Zurich compiler). Again we compared with an implementation designed to provide good recovery. To quote Wirth [22, p. 320],

... It was also recognized that one of the major challenges in developing a processing system for a language is its capability to meaningfully diagnose syntactic errors and to continue processing of subsequent text with a reasonably large probability of correct diagnosis. If the system is to be used successfully in an environment of programming novices, this capability must be assigned no less than highest priority. The problem of syntax analysis thereby obtains entirely new aspects; the compiler must not only process the defined language, but virtually all sequences of symbols of the basic vocabulary. ...

Such comparisons are necessarily somewhat subjective. However, it appears that our error recovery techniques are qualitatively better than those of the Zurich and PL/C implementations. The errors which those compilers handle well are also dealt with appropriately by our recovery scheme. In addition, we handle well a variety of errors that are improperly dealt with in the other compilers. The PASCAL compiler tends to find only the first of a set of dense errors and to skip arbitrarily large portions of text in getting "back on the track." (In one of our tests, an error in a declaration caused the PASCAL compiler to skip all of the subsequent declarations, causing a plethora of undefined symbols in the remainder of the program.) The PL/C compiler treats errors substantially more locally, thereby detecting more errors than the PASCAL compiler, although fewer than our programs find. However, the PL/C compiler can sometimes correct too locally, thereby failing to use context information. Additionally, it tends to have a left-to-right bias (that is, it assumes that any text already parsed must have been parsed correctly) and an inflexibility with regard to misuse of key words.

For lack of space we present only a representative example. Figure 1 contains a program run on our Algol recovery parser; Figure 2 contains a PL/C-equivalent program run on the PL/C compiler; Figure 3 contains a PASCAL-equivalent program run on our PASCAL recovery parser; and Figure 4 contains the same PASCAL program run on the PASCAL compiler produced in Zurich.

[In the research reported here, we were concerned primarily with the recovery actions taken by the compiler. Little emphasis was placed on the wording of the messages. The first author has subsequently considered this problem; the examples shown represent a transitional stage in improving the wording.]

In the first declaration of each program, the comma in the bounds list of the array declaration has been omitted. Both the PL/C compiler and our Algol parser recover by inserting the comma. Our PASCAL parser recovers by deleting the subrange 1..10. This is an equally good recovery action but is less likely to correspond to the programmer's intention. Notice that since the recovery actions are based on the form of the grammars, the Algol and PASCAL parsers do not necessarily take the same action. The lack of lookahead in the

Fig. 1. Algol subset recovery parser.

```

ALGOL W SUBSET COMPILER - SEPT 1973 VERSION

1 BEGIN
2   INTEGER ARRAY A[1..5 1..10];
3   0
4   0
5   0
6   0
7   0
8   0
9   0
10  0
11  0
12  0
13  0
14  0
15  0
16  0
17  0
18  0
19  0
20  0
21  0
22  0
23  0
24  0
25  0
26  0
27  0
28  0
29  0
30  0
31  0
32  0
33  0
34  0
35  0
36  0
37  0
38  0
39  0
40  0
41  0
42  0
43  0
44  0
45  0
46  0
47  0
48  0
49  0
50  0
51  0
52  0
53  0
54  0
55  0
56  0
57  0
58  0
59  0
60  0
61  0
62  0
63  0
64  0
65  0
66  0
67  0
68  0
69  0
70  0
71  0
72  0
73  0
74  0
75  0
76  0
77  0
78  0
79  0
80  0
81  0
82  0
83  0
84  0
85  0
86  0
87  0
88  0
89  0
90  0
91  0
92  0
93  0
94  0
95  0
96  0
97  0
98  0
99  0
100 0

```

Fig. 2. PL/C compiler.

```

TEST: PROCEDURE OPTIONS(MAIN);
STMT LEVEL NEST BLOCK SOURCE STATEMENT PL/C-R6.6000

1 1 1 TEST: PROCEDURE OPTIONS(MAIN);
2 1 1 DECLARE(A,B) (1:5 1:10) FIXED;
3 1 1 ERROR SY0 MISSING COMMA
4 1 1 DECLARE (A,B) (1: 5,1: 10) FIXED;
5 1 1 DECLARE (I,J,K,L) FIXED;
6 1 1 UP: I+J>K+L*4 THEN GO L1 ELSE K IS L2;
7 1 1 ERROR SY11 IMPROPER ELEMENT
8 1 1 ERROR SY34 IMPROPER THEN OR ELSE
9 1 1 ERROR SY0F MISSING KEYWORD
10 1 1 ERROR SY08 MISSING SEMI-COLON
11 1 1 UP: GOTO L1;
12 1 1 ERROR SY34 IMPROPER THEN OR ELSE
13 1 1 ERROR SY27 MULTIPLE DECLARATION
14 1 1 ERROR SY09 MISSING :
15 1 1 ERROR SY00 MISPELLED KEYWORD
16 1 1 ERROR SY07 EXTRA SEMI-COLON
17 1 1 $L001$: IF L2
18 1 1 A 1,2 := 8(3*(I+4, J+K))
19 1 1 ERROR SY27 MULTIPLE DECLARATION
20 1 1 ERROR SY09 MISSING :
21 1 1 ERROR SY11 IMPROPER ELEMENT
22 1 1 ERROR SY10 INCOMPLETE EXPRESSION
23 1 1 ERROR SYED COMMENT RUNS ACROSS CARD BOUNDARY
24 1 1 ERROR SY10 INCOMPLETE EXPRESSION
25 1 1 THEN $L002$: IF I=1
26 1 1 IF I = 1 THEN THEN GO TO UP;
27 1 1 ERROR SY38 IMPROPER PREFIX ORDER
28 1 1 L2: END;
29 1 1 ERROR SYES ILLEGAL USE OF COLUMN 1 ON CARD
30 1 1 L2: END;
31 1 1 IN STMT
32 1 1 PL/C USES 4 ERROR SH#1 WRONG TYPE FOR EXPRESSION
33 1 1 UP: GOTO $L001$;
34 1 1 IN STMT
35 1 1 PL/C USES 5 ERROR SH#1 WRONG TYPE FOR EXPRESSION
36 1 1 $L001$: IF '1'B

```

Zurich compiler and its predictive parsing algorithm cause it to choose "]" instead of "," as the expected symbol.

In the first executable statement, the symbol **if** is missing, following the label. The statement following **then** is a mutilated branch statement; the statement following **else** is perhaps an assignment. Both of our parsers insert the **if** and recover locally from each conditionally executed statement, although again the ac-

Fig. 3. PASCAL recovery parser.

```

PASCAL COMPILER - 16 SEPT 1973 VERSION

1 VAR A,B:ARRAY[1..5 1..10] OF INTEGER;
2   0
3   0
4   0
5   0
6   0
7   0
8   0
9   0
10  0
11  0
12  0
13  0
14  0
15  0
16  0
17  0
18  0
19  0
20  0
21  0
22  0
23  0
24  0
25  0
26  0
27  0
28  0
29  0
30  0
31  0
32  0
33  0
34  0
35  0
36  0
37  0
38  0
39  0
40  0
41  0
42  0
43  0
44  0
45  0
46  0
47  0
48  0
49  0
50  0
51  0
52  0
53  0
54  0
55  0
56  0
57  0
58  0
59  0
60  0
61  0
62  0
63  0
64  0
65  0
66  0
67  0
68  0
69  0
70  0
71  0
72  0
73  0
74  0
75  0
76  0
77  0
78  0
79  0
80  0
81  0
82  0
83  0
84  0
85  0
86  0
87  0
88  0
89  0
90  0
91  0
92  0
93  0
94  0
95  0
96  0
97  0
98  0
99  0
100 0

```

Fig. 4. Zurich PASCHAL compiler.

```

005001 VAR A,B:ARRAY[1..5 1..10] OF INTEGER;
005001 1,J,K,L: INTEGER;
005005 BEGIN
005071 3: I + J > K + L * 4 THEN GO 1 ELSE K IS 2;
005071 A 1,2 := B [ 3 * ( I+4,J+K)
005074 IF I=1 THEN THEN GOTO 3;
005077 2: END.

COMPILER ERROR MESSAGES:
*****
17: E:E EXPECTED.
42: ILLEGAL SYMBOL IN EXPRESSION.
48: E:E EXPECTED.
52: E:E EXPECTED.
54: ILLEGAL SYMBOL IN STATEMENT.
58: E:E EXPECTED.

```

tions may or may not conform to the programmer's intentions. (If the branch were backward and the label already "seen," the recovery might well be different.) Both the PL/C compiler and the Zurich compiler exhibit a left-to-right bias, thereby failing to insert the **if**, causing subsequent error messages concerning the structure of the statement. PL/C reconstructs the branch statement but converts "is" to **if**. The Zurich compiler never detects the error following **then**, but it signals a missing **:=**.

In the assignment statement, the brackets around the subscripts of *A* are missing and the subscripts of *B* are mutilated expressions. There is also a missing ";" following the statement. Our parsers recover satisfactorily from all these errors, although not always in the same way. PL/C is unable to analyze the statement at all. The Zurich compiler skips forward to the **:=** in an attempt to fix the previous error. It then inserts the missing ")" and ";;".

The **if** statement contains an extra **then**. Both our parsers delete the **then** and analyze the rest of the statement. The PL/C compiler gives an unintelligible message for the entire statement. The Zurich compiler objects to the second **then** and skips to the “;”.

In our empirical studies, we tried a number of optional features, some more successfully than others. We found, for example, that one could reduce the amount of computation in the correction phase and get satisfactory, although lower quality, recovery. One could consider additional replacements for candidate substrings—for instance, replacement by any non-terminal in the grammar (rather than only RHS's). Our studies on these variations are, for the most part, inconclusive. (Most of these experiments are described in [17].)

7. Discussion

A distinction can be made between those error-handling schemes which concentrate solely on getting the parser “back on the track” and those which transform incorrect input sequences into syntactically correct programs.⁹ The work of Leinius is an example of the first approach; Levy's work exemplifies the second. Our methods lie somewhere in between. Our weighted minimum distance correction phase is an attempt to determine a probable explanation for the error, not only as a more powerful and more local method than Leinius' for choosing among locally correct changes but also as an aid to the programmer in correcting his program. However, we do not attempt to produce a “corrected” program text. It seems likely that our approach could be extended in this way at the cost of some compile-time overhead for correct programs (at the very least, retaining the input text and keeping track of the correspondence between elements of the parsing stack and locations in the input text). Additionally there are some instances in which the recovery actions taken clearly do not correspond to the programmer's intention. Consider again Example 2 of Section 4, in which

... $X := I \ J;$

is parsed as

$\notin \langle \text{blockbody} \rangle \langle \text{statement} \rangle ? \langle \text{expression} \rangle$

with current input symbol “;”. The recovery action is to delete $\langle \text{expression} \rangle$; namely J . The same parsing stack and subsequent action would ensue if the erroneous statement were

$X := I \ J + K * L;$

⁹ Levy refers to the former approach as “recovery” and the latter as “correction.” As our remarks in the introduction indicate, we find this terminology inappropriate.

¹⁰ The first author is experimenting with a technique that solves this problem.

although it seems clear that some binary operator is missing between I and J .¹⁰ However, notice that the expression following I is analyzed, thereby discovering and recovering from subsequent errors in the expression.

It appears that certain aspects of our approach to error recovery are easily incorporated in other bottom-up parsing methods. For example, the use of weightings in choosing among alternatives is a general technique. (Interestingly enough, a probabilistic approach was used for spelling correction in CORC [5] more than ten years ago.) Many of the error recovery techniques in use suffer from a left-to-right bias; namely, it is assumed that the text prior to the point at which an error is discovered must be correct because it can be parsed. This bias can prevent insertion of missing **if**'s, **begin**'s, and the like. In some instances this bias is simply myopia on the part of the implementer. As Peterson points out [16], this left-to-right bias can be avoided in *LR* parsing by considering changes to entries further back in the parsing stack, rather than restricting consideration to the element at the top. A change below the top of the *LR* parsing stack requires recomputing all the elements above the one that is changed; this computation can be bounded by the depth of the stack at which changes are considered, and, in any case, is substantially smaller than reparsing the input text.

However, correct prefix parsing methods appear to be at a disadvantage when it comes to using context to determine recovery actions. Left context information is available in abundance. In fact, it is one of the strong points of *LR* parsing that much important structural information about the text already parsed is contained on the parsing stack. However, that strength becomes a weakness when it comes to analyzing the right context of an error detection point. The fact that the next move of the parser can depend on the entire correct prefix already analyzed makes it difficult or impossible to start up the parser after the error point. On the other hand, it is quite straightforward to extend the context analysis to other more local parsing methods, such as the wide variety of mixed strategy and bounded right context methods.

Although our approach to error recovery appears very promising, there are still many issues to be investigated. Details need to be worked out for the extension to other parsing methods and the techniques should be implemented and tested. It may turn out that if parsing methods are used which put weaker constraints on grammar form, more computation will be required in the correction phase. A related question is in what ways the characteristics of a grammar affect the quality of recovery.

As the reader can observe, a number of heuristics have been used in order to impose rather stringent efficiency constraints. It would be interesting to know what the tradeoffs are. How much better might the recovery be if more context were used? What if the

correction phase were made "smarter"?)

Another issue of importance, particularly for those interested in producing "corrected" source text, is the incorporation in the recovery method of language-dependent¹¹ features and of information from other parts of the compiling process. In the method described, language-dependent information is introduced by the implementer only in the form of modifications to the cost functions. If, as is usual, the lexical analysis is a separate compiler phase which produces tokens or symbols for the parser, incorporation of lexically based recovery can be achieved by the addition of Morgan-style spelling correction [15] whenever a detected error involves an identifier and lexically determined low-cost replacements (such as ":", for " "). Since our emphasis was initially on automatically generated error recovery techniques, we have not explored the introduction of semantics (in the compiler sense, thereby including non-context-free syntax issues) into the recovery process. For instance, when it is determined that an identifier is to be inserted, no attempt is made to infer which identifier. At present we know of no alternative to hand-coded recovery routines to handle this particular problem. On the other hand, if the correction phase of recovery succeeds in replacing part of the sentential form by the RHS of a rule, the compiler could then execute the semantics associated with that rule, although it might also be necessary to do semantic error recovery in that case. In order to preserve semantic integrity, the semantic rules must also be designed so they can be executed "out of order" by the forward move.

Other aspects of the error recovery problem which we are presently studying include the generation of error messages, the effect of recovery considerations on programming language design, and more sophisticated cost function determination. Some of these topics are discussed further in [17].

Despite the current activity in other areas of software development, we see a continuing need for error recovery techniques. Most syntax errors and many semantic errors result from incomplete knowledge of the programming language being used, transcription errors, and carelessness. These errors will continue to be in evidence. Our contribution is an attempt to provide recovery techniques that aid both the implementer and the users of his or her implementation.

Received December 1974; revised February 1975

¹¹ By "language-dependent" we really mean "not formally specified"; that is, knowledge of the language that must be explicitly incorporated by the implementer.

References

1. Aho, A.V., and Ullman, J.D. *The Theory of Parsing, Translating, and Compiling*. Prentice-Hall, Englewood, N.J., 1972.
2. Bauer, H.R., Becker, S., and Graham, S.L. Algol W implementation. Tech. Rep. CS 98, Computer Sci. Dep., Stanford U., Stanford, Ca., 1968.
3. Conway, R.W., Morgan, H.L., Wagner, R.A., and Wilcox, T.R. PL/C. A high performance subset of PL/I. Tech. Rep. 70-55, Computer Sci. Dep., Cornell U., Ithaca, N.Y., 1970.
4. Conway, R.W., and Wilcox, T.R. Design and implementation of a diagnostic compiler for PL/I. *Comm. ACM* 16, 3 (Mar. 1973), 169-179.
5. Freeman, D.N. Error correction in CORC, the Cornell computing language. *FJCC*, 1964, 15-34.
6. Graham, S.L., and Rhodes, S.P. Practical syntactic error recovery in compilers. ACM SIGACT-SIGPLAN Symp. on Principles of Prog. Lang., Boston, Oct. 1973, pp. 52-58.
7. Gries, D. The use of transition matrices in compiling. *Comm. ACM* 11, 1 (Jan. 1968), 26-34.
8. Irons, E.T. An error-correcting parse algorithm. *Comm. ACM* 6, 11 (Nov. 1963), 669-673.
9. James, E.G., and Partridge, D.P. Adaptive correction of program statements. *Comm. ACM* 16, 1 (Jan. 1973), 27-37.
10. James, L.R. A syntax directed error recovery method. Master's Th., U. of Toronto, Computer Systems Research Group Tech. Rep. CSRG-13, May 1972.
11. La France, J.E. Syntax directed error recovery for compilers. Ph.D. Th., U. of Illinois, Urbana, Computer Sci. Dep. ILLIAC IV Doc. 249, 1971.
12. Leinius, R.P. Error detection and recovery for syntax directed compiler systems. Ph.D. Th., Computer Sci. Dep., U. of Wisconsin, Madison, 1970.
13. Levy, J.P. Automatic correction of syntax errors in programming languages. Ph.D. Th., Cornell U., Computer Sci. Dep. Tech. Rep. TR71-116, Dec. 1971.
14. Lyon, G. Syntax-directed least-errors analysis for context-free languages: a practical approach. *Comm. ACM* 17, 1 (Jan. 1974), 3-14.
15. Morgan, H.L. Spelling correction in system programs. *Comm. ACM* 13, 2 (Feb. 1970), 90-94.
16. Peterson, T.G. Syntax error detection, corrections and recovery in parsers. Ph.D. Th., Stevens Institute of Technology, Hoboken, N.J., 1972.
17. Rhodes, S.P. Practical syntactic error recovery for programming languages. Ph.D. Th., U. of California, Berkeley, Dep. of Computer Sci. Tech. Rep. 15, June 1973.
18. Teitelbaum, R. Context-free error analysis by evaluation of algebraic power series. Proc. ACM-SIGACT Fifth Ann. Conf. on Theory of Computing, U. of Texas, Austin, Texas, 1973, pp. 196-199.
19. Wirth, H., and Weber, H. EULER: a generalization of Algol and its formal definition; Parts I and II. *Comm. ACM* 9, 1-2 (Jan.-Feb. 1966), 13-35, 89-99.
20. Wirth, N. A programming language for the 360 computers. *J. ACM* 15, 1 (Jan. 1968), 37-74.
21. Wirth, N. The programming language PASCAL. *Acta Informatica* 1, 1 (Jan. 1971), 35-63.
22. Wirth, N. The design of a Pascal compiler. Proceedings of the International Summer School of Program Structures and Fundamental Concepts of Programming, Munich, Germany, July 1971.

(Please turn the page for Appendix.)

Appendix: Syntax of Algol Subset

```

<program> ::= ENDMARKER <block> ENDMARKER
<block> ::= <blockbody> end
           ::= <blockbody> <statement> end
<blockbody> ::= <blockhead>
               ::= <blockbody> <label-definition>
               ::= <blockbody> <statement> ;
               ::= <blockbody> ;
<blockhead> ::= begin
               ::= <blockhead> <declaration> ;
<declaration> ::= <simple-declaration>
                 ::= <array-declaration>
<simple-declaration> ::= integer <identifier>
                       ::= <simple-declaration>, <identifier>
<array-declaration> ::= <boundslist> <expression> .. <expression> )
<boundslist> ::= <arrayhead> (
                 ::= <boundslist> <expression> .. <expression> ,
<arrayhead> ::= integer array <identifier>
                ::= <arrayhead>, <identifier>
<label-definition> ::= <identifier> :
<statement> ::= <simplestatement>
               ::= <if-then-cl> <statement>
               ::= <if-then-cl> <elseclause> <statement>
<elseclause> ::= <simplestatement> else
<if-then-cl> ::= if <expression> then
<simplestatement> ::= go to <identifier>
                   ::= <block>
                   ::= <variable> := <expression>
                   ::= <procedure-head> <expression> )
                   ::= stop

<procedure-head> ::= <procedure-identifier> (
                   ::= <procedure-head> <expression> ,
<expression> ::= <expr>
               ::= <expr> <relationop> <expr>
<expr> ::= <term>
           ::= + <term>
           ::= - <term>
           ::= <expr> + <term>
           ::= <expr> - <term>
           ::= <expr>  $\vee$  <term>
<term> ::= <factor>
          ::= <term> * <factor>
          ::= <term> / <factor>
          ::= <term>  $\wedge$  <factor>
<factor> ::= <secondary>
           ::=  $\neg$  <factor>
<secondary> ::= <primary>
              ::= <secondary> ** <primary>
<primary> ::= <number>
            ::= <variable>
            ::= ( <expression> )
<variable> ::= <simple-variable>
            ::= <arrayname> <expression> )
<simple-variable> ::= <identifier>
<arrayname> ::= <array-identifier> (
              ::= <arrayname> <expression> ,

```

The symbols <identifier>, <number>, <array-identifier>, <procedure-identifier>, and <relationop> are handled by the lexical scanner.

```

<array-identifier> ::= <identifier>
<procedure-identifier> ::= READ
                        ::= WRITE
                        ::= <identifier>
<relationop> ::= <
               ::=  $\leq$ 
               ::= =
               ::=  $\neq$ 
               ::=  $\geq$ 
               ::= >

```