effort required to implement the proposed method are comparable to those required by Fishman's, Wallace's, and Tadikamalla's methods. The proposed method is faster than Fishman's and Tadikamalla's method for all values of α, $\alpha > 2$. The average CPU time required per variate for the proposed method remains fairly constant for medium and large values of α. The proposed method is simpler and faster than Ahrens and Dieter's method for all values of α.

Received October 1977; revised April 1978

References

1. Ahrens, J.H., and Dieter, U. Computer methods for sampling from gamma, beta, Poisson and binomial distributions. *Computing 12* (1974), 223–246.
2. Atkinson, A.C., and Pearce, M.C. The computer generation of beta gamma and normal random variables. *J. Royal Statist. Soc. Ser. A, 139* (1976), 431–461.
3. Box, G.E.P., and Muller, M.E. A note on the generation of normal deviates. *Annals Math. Statist. 29* (1958), 610–611.
4. Dudewicz, E.J. Speed and quality of random numbers. Annual Tech. Conf. Trans. ASQC, Vol. 29, 1975, pp. 170–180.
5. Fishman, G.S. Sampling from the gamma distribution on a computer. *Comm. ACM. 19*, 7 (July 1976), 407–409.
6. Greenwood, A.J. A fast generator for gamma distributed random variables. In *CompStat*, G. Bruckman et al., Eds., Physica Verlag, Vienna, 1974, pp. 19–27.
7. Jöhnk, M.D. Erzeugung Von Betavesteilten Und Gamma Vesteilten Zufellszahlen. *Metrika 8* (1964), 5–15.
8. Kinderman, A.J., and Ramage, J.G. Computer generation of normal random variables. *J. Amer. Statist. Assoc. 17* (1976), 893–896.
9. Lurie, D., and Mason, R.L. Empirical investigation of several techniques for computer generation of order statistics. *Comm. Statist. 2* (1973), 363–371.
10. Marsaglia, G. Random variables and computer. Trans. Third Prague Conf. Inform. Theory, Statist. Decision Functions, Random Processes, June 1962, Prague: Czechoslovak Acad. of Sciences, Prague, 1964, pp. 499–512.
11. Odell, P.L., and Newman, T.G. *The Generation of Random Variates.* Charles Griffin, London, 1972.
12. Tadikamalla, P.R. Computer generation of gamma random variables. *Comm. ACM 21*, 5 (May 1978), 419–422.
13. Tadikamalla, P.R. FORTRAN programs for computer generation of gamma random variables. Tech. Rep., Dept. Business Admin., Eastern Kentucky U., Richmond, Ky., 1977.
14. Tadikamalla, P.R. The factors that may affect the speed of normal variate generators. Tech. Rep., Dept. Business Admin., Eastern Kentucky U., Richmond, Ky., 1978.
15. Tadikamalla, P.R., and Johnson, M.E. Some simple rejection methods for sampling from the normal distribution. Proc. First Int. Conf. Math. Modeling, St. Louis, Mo., 1977, 573–578.
16. Wallace, N.D. Computer generation of gamma random variates with non-integral shape parameters. *Comm. ACM 17*, 12 (Dec. 1974), 691–695.
17. Whittekar, J. Generating gamma and beta random variables with nonintegral shape parameters. *App. Statist. 23* (1974), 210–213.

Programming Languages

J.J.Horning Editor

A Simple Recovery-Only Procedure For Simple Precedence Parsers

G. David Ripley
RCA Laboratories

A simple method is described enabling simple precedence parsers to recover from syntax errors. No attempt to repair errors is made, yet parsing and most semantic processing can continue. The result is a good "first approximation" to syntax error handling with negligible increase in parsing time, space, and complexity of both the parser and its table generator.

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

CR Categories: 4.12, 4.42, 5.23

1. Introduction

Syntax error handling procedures for formal parsing methods, such as simple precedence [12], LL(k) [9], and LR(k) [8], usually suffer from one or more drawbacks. Some of these procedures skip the rest of a sentence after detecting an error [10]. Others are complex and require a substantial implementation effort [4–6], and may require tailoring to a particular language [2]. A procedure is described here which enables simple precedence parsers to recover from syntax errors. With this method, no input text is skipped, the procedure is very simple and easy to implement, and it is independent of source language.

The method, called Simple Recovery (SR) after a similar method for SLR parsers [3], is essentially the "forward move" described by Graham and Rhodes in their paper on error recovery and repair for simple precedence parsers [6]. Unlike the method of Graham

Author's present address: RCA Laboratories, Princeton, NJ 08540.

and Rhodes, SR makes no "backward move," nor is any repair of the error attempted. Rather, upon detection of any type of syntax error, a new forward move is initiated. In spite of its simple nature, SR is quite successful at enabling the parse, and hence error detection, to continue. Most semantic error checking can also continue.

In what follows, it is assumed that the reader is familiar with basic parsing notions and simple precedence parsers in particular, for example as described in [1] or [7].

2. Simple Recovery

A simple precedence parser detects errors in several situations. A *character pair error* is detected when the parser's top stack symbol and the next input symbol have no precedence relation. A *right hand side error* has occurred when a potential right-hand side on the stack has been identified by the precedence relations yet no grammar rule has such a right-hand side. Finally, a *stackability error* occurs when the stack symbol to the left of the handle is neither less than nor equal in precedence to the left hand side of the rule whose right-hand side is the handle. Regardless of the type of error, it is assumed that the parser calls a recovery routine for the purposes of producing an error message and resetting the parser state to permit continuation of the parse.

The recovery action of SR is to simply discard the current stack entries and push onto the (now empty) stack the symbol "??", a unique error symbol that is less in precedence than all other symbols in the grammar. (The current input symbol has not as yet been pushed onto the stack.) Control is then returned to the parser.

Error messages are produced by SR prior to recovery based on the type of error. For character pair errors and stackability errors, the two symbols in question are printed along with an appropriate explanation. Right-hand side errors are messaged by listing the alleged right-hand side as well as the symbol immediately preceding and the symbol immediately following the alleged right-hand side (i.e. the stack symbol below the alleged right-hand side and the current input symbol).

An exception to the messaging of right-hand side errors occurs when the stack contains only the symbol "??" followed by the alleged right-hand side. Experience with the method indicates that most of the time this situation is really an attempt by the parser to reduce a right-hand side whose prefix is missing due to a previous error and the subsequent discarding of the stack contents. In this situation, referred to as a probable attempt to "reduce across the error point" [4], no error message is issued.

Implementation of SR follows directly from the above description. The only modification to the simple precedence parser tables is the addition of a row to the precedence matrix for the error symbol "??", which is set "less in precedence" to all other symbols in the grammar. Of course, the grammar symbols in symbolic form must be available for use in the error messages. In addition, to improve error pinpointing the recovery routine used in the examples given below also lists the first terminal symbol that begins a stacked nonterminal symbol. These terminal symbols are maintained in a stack paralleling the syntax stack.

Figure 1 contains the results of using SR on an example similar to that given in [6] (Graham and Rhodes used a subset of Algol; the examples here are written in a language called L/1, used in a course on compiler construction). Figure 2 is included to illustrate several additional error situations.

3. Discussion

Although it is not clear what a "representative" collection of syntax errors consists of, a recent study of errors made by programmers indicates that most errors are either single missing, extra, or wrong symbols [11]. That is, most errors involve only one symbol. Figure 1 contains examples of all three of these types of errors. All the syntax errors in this program were detected, including the four errors in the assignment to an element of the array "a". The erroneous symbol "." at the end of the program was deleted by the lexical analyzer. Note that SR tends to "bracket" wrong symbol errors, such as "IS" in place of ":=", by giving a message immediately before and after the incorrect symbol.

An important reason for error recovery is to enable not only continued syntax error checking but also semantic error checking. This is particularly true for strongly typed languages, where many semantic errors can be detected at compile time. Even though SR makes no attempt at repairing syntax errors, it has been found in practice that most semantic processing can still take place. This is particularly true when semantic processing is synchronized with syntax analysis. Since a handle that contains a syntax error is never properly reduced, the semantic routines deal only with correct handles and their associated semantic information.

Certain semantic processing in a compiler relies heavily on action taken during prior reductions, although this seems to cause little problem. For example in Figure 2 the "loopheader" statement beginning with the keyword "LOOP" contains an error (a missing semicolon), enough to prevent semantic processing of the loop header. The semantic action for the corresponding ENDLOOP statement depends on actions that should have been taken upon reduction of the loop header. However since a loop header was never recognized, the grammar rule

⟨statement⟩ ::= ⟨loop header⟩ ⟨statements⟩ ENDLOOP

cannot be applied upon recognition of "ENDLOOP", hence the dependent semantic processing will not be attempted.

Of course semantic cascading and even internal con-

Fig. 1. An example of Simple Recovery.

```
  PROC test( );
    LOCAL a[5] INT (5:0), b[10] INT (10:0);
    LOCAL i INT(0), j INT(0), k INT(0), l INT(0);
    CALL sub(1+5 1+10);
***** Error: 5 may not be followed by 1
  up: i+j > k+1*4 THEN BREAKTO; ELSE k IS 2;
***** Error: the label beginning with up
        may not be followed by
        the arith.expr. beginning with i
***** Error: BREAKTO may not be followed by ;
***** Error: k may not be followed by IS
***** Error: IS may not be followed by 2
  a 2 := b[3*(i+4,j*/k]
***** Error: a may not be followed by 2
***** Error: 2 may not be followed by :=
***** Error: The components left paren arith.expr. ,
            beginning with (
            when preceded by *
            and followed by j
            do not form part of a sentence
***** Error: * may not be followed by /
  IF i=1 THEN THEN REPEATAT up;
***** Error: ] may not be followed by IF
***** Error: THEN may not be followed by THEN
  12: RETURN;
END test.
***** Error: "." is illegal—deleted
***** Error: test may not be followed by end-file
```

Fig. 2. Another example of Simple Recovery.

```
  PROC samples (i INT,,s STR);
***** Error: , may not be followed by ,
  s:= 'sam';
        LOCAL x[2] INT (0,0), y INT(1;;
***** Error: The statement(s) beginning with s
        may not be followed by LOCAL
***** Error: The components decl.header arith.expr.
        beginning with y
        when preceded by the declarations
        beginning with LOCAL
        and followed by ;
        do not form part of a sentence
***** Error: ; may not be followed by ;
  LOOP i := 1 BY 1 TO 10
    x[] := x[]+y+1;
***** Error: 10 may not be followed by x
***** Error: The components x [ ]
        when preceded by :=
        and followed by +
        do not form part of a sentence
  ENDLOOP;
  RETURN;
  END samples;
```

fusion may occur unless certain semantic processes are done in a "fail-safe" manner. The point is that experience suggests this may not in general be difficult to achieve.

Figure 2 illustrates the benefit of continued semantic processing in several places. For example, even though the formal parameter list contains an error, following recovery the parameter "s" was processed both syntactically and semantically, resulting in no cascaded semantic errors for this parameter. Panic mode recovery, in comparison, by skipping to the end of an erroneous sentence or other large construct before restarting the parse, would most likely not have fared so well. Figure 2 also contains a declaration out of order (declarations must precede executable statements in L/1). After detection of the ordering problem, syntactic and semantic processing of the rest of the declaration continued.

It is possible for SR to miss certain right-hand side errors, but only when such an error occurs immediately following another error, with no intervening symbols. This is due to the assumption by SR that in such a situation an attempt is being made to reduce across the error point. Figure 2 illustrates this point. The first erroneous "x[ ]" is not reported, as the missing semicolon error occurred immediately before it. However, the symbol ":=" appearing between the two errors "x[ ]" was sufficient to enable messaging of the second such error. Fortunately, due to the sparse nature of most syntax errors, this property of SR seems to be of little concern in practice.

Of course effective error repair in general is more desirable than simple recovery in situations where the associated complexity is considered worth the effort. The paper by Graham and Rhodes in fact illustrates this. The point of SR is that for negligible increase in time, space, and, most important, complexity of the parser (and also of the parser generator), a reasonable "first approximation" to error handling is possible.

Received April 1977

References

1. Aho, A.V., and Ullman, J.D. *The Theory of Parsing, Translation, and Compiling, Vol. 1.* Prentice-Hall, Englewood Cliffs, N.J., 1972.
2. Conway, R.W., and Wilcox, T.R. Design and implementation of a diagnostic compiler for PL/I. *Comm. ACM 16*, 3 (March 1973), 169–179.
3. Druseikis, F.C., and Ripley, G.D. Error recovery for simple LR(k) parsers. Proc. ACM Annual Conf., Houston, Tex., 1976, pp. 396–400.
4. Druseikis, F.C., and Ripley, G.D. Extended SLR(k) parsers for error recovery and repair. Tech. Rep., Comptr. Sci. Dept., U. of Arizona, Tucson, 1977.
5. Fischer, C.N., Milton, D.R., and Quiring, S.B. An efficient insertion-only error-corrector for LL(1) parsers. Proc. Fourth ACM Symp. on Principles of Programming Languages, Santa Monica, Calif., 1977, pp. 97–103.
6. Graham, S.L., and Rhodes, S.P. Practical syntax error recovery. *Comm. ACM 18*, 11 (Nov. 1975), 639–642.
7. Gries, D. *Compiler Construction for Digital Computers.* Wiley, New York, 1971.
8. Knuth, D.E. On the translation of languages from left to right. *Inform. Contr. 8* (Oct. 1965), 607–639.
9. Lewis, P.M., and Stearns, R.E. Syntax directed transduction. *J. ACM 15*, 3 (March 1968), 464–488.
10. McKeeman, W.M., Horning, J.J., and Wortman, D.B. *A Compiler Generator.* Prentice-Hall, Englewood Cliffs, N.J., 1970.
11. Ripley, G.D., and Druseikis, F.C. A statistical analysis of syntax errors. *J. Computer Languages.* To appear.
12. Wirth, N., and Weber, H. EULER: A generalization of Algol and its formal definition, Pts. I, II. *Comm. ACM 9*, 1-2 (Jan., Feb. 1966), 13–35, 89–99.