The University of Warwick

COMPUTER SCIENCE REPORTS

REPORT No. 73

Error Recovery for Yacc Parsers

by

Julia Dain

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

October 1985

Error Recovery for Yacc Parsers

Julia Dain

Dept. of Computer Science
University of Warwick
Coventry CV4 7AL
UK

ABSTRACT

We aim to improve error recovery in parsers generated by the LALR parser-generator *Yacc*. We describe an error recovery scheme which a new version of *Yacc* automatically builds into its parsers. The scheme uses state information to attempt to repair input which is syntactically incorrect. Repair by alteration of a single token is attempted first, followed by replacement of a phrase of the input. A parser for the C language is generated from existing specifications and tested on a collection of student programs. The quality of error recovery and diagnostic messages is found to be higher than that of the existing portable C compiler. The new version of *Yacc* may be used by any current user of *Yacc*, with minor modifications to their existing specifications, to produce systems with enhanced syntax error recovery.

August 21, 1985

Error Recovery for Yacc Parsers

Julia Dain

Dept. of Computer Science
University of Warwick
Coventry CV4 7AL
UK

1. Introduction

The portable C compiler *pcc* [Johnson78b] is widely used in UNIX† environments but its diagnostic messages are poor. The parser for *pcc* is built by the LALR parser-generator *Yacc* [Johnson78a] which automatically generates error recovery routines. Many other popular UNIX utilities contain syntax analysers built by *Yacc*, such as the pattern matchers *lex*, *awk* and *grep* and the FORTRAN 77 and C++ compilers, and these utilities would also be easier to use if they had improved diagnostics. The aim of the work presented here is to improve the error recovery scheme which *Yacc* builds into its parsers and thus to improve the error handling in *pcc*.

This paper describes the old method for error recovery in parsers built by *Yacc*, and a new general-purpose method which is independent of source language and which may be used with existing *Yacc* input specifications with minor changes. We present tests on the resulting C compiler which show an improvement in error handling. We assume familiarity with LR parsing as described in [Aho77] for example.

2. The portable C compiler and *Yacc*

In some computing environments, for example a university where many students are learning to use a new language, the quality of error diagnostics produced by a compiler is at least as important as the efficiency of generated code. Students using a UNIX environment who learn C after Pascal often ask why the portable C compiler is so poor compared with the Berkeley Pascal system [Joy80]. A reason for their dissatisfaction is that *pcc* is unable to diagnose many simple syntax errors and produces misleading error messages, whereas the authors of the Berkeley Pascal compiler paid particular attention to developing a good error recovery scheme, presented in [Graham79].

Yacc produces LALR(1) parsers from a set of grammar rules (productions) and actions. The parsers contain default reductions, that is any state of the parser which has a unique reduction in its actions is given that reduction as entry for all symbols which cannot be shifted. To make use of the existing automatic error recovery scheme, described in [Aho74], the productions of the grammar should contain error productions of the form $A \rightarrow \alpha$ **error** β, where A denotes a non-terminal, α, β denote strings of grammar symbols, and **error** denotes the token reserved by *Yacc* for error handling. When the parser is presented with an input token which is not a legal symbol for the current state, it enters error recovery mode and inserts the **error** token on the input. The parser pops states from the stack until the top state is one which can shift

†UNIX is a Trademark of Bell Laboratories.

error. Parsing then continues as dictated by the parse tables, except that any token for which there is no parsing action is deleted from the input. When three input tokens have been shifted, the parser assumes recovery is complete and leaves error recovery mode. In effect the parser assumes that an error has occurred while looking for a derivation of a non-terminal A and that a series of tokens approximating to a derivation of A has been consumed from the input.

Yacc allows the user some control over error recovery actions by permitting error productions to have semantic actions associated with them. These can be used to specify actions to be taken in particular cases. *Yacc* also allows the user to force the parser out of error recovery mode before three tokens have been shifted, and to clear the lookahead token.

The grammar for *pcc* contains eight error productions, one for the external definition construct (the highest-level block of which C programs are composed, that is function and data definitions), five for various forms of declarations and two for the statement construct. Only three of these productions have semantic actions, and these only change local variables. The productions for the statement construct are

statement → **error** ';' | **error** '}'

These productions mean that if the parser detects an error in a statement it will skip all input to the next semi-colon or right curly bracket. All the other error productions have the form

declaration → **error**

These cause the parser to skip input to anything which can follow the declaration. No use is made of the facilities to force the parser out of error recovery mode or clear the lookahead token.

In general, the method for error recovery in *Yacc* has some disadvantages. The user has to write error productions which will control error recovery to an extent which the user may not realise. These productions may introduct ambiguities into the grammar. During recovery, input and stack states are deleted silently. No information about the nature of an error is available. The advantages of the method are that it is simple to implement and efficient to run. In the particular case of *pcc*, the main disadvantage is the poor quality of diagnostic messages, which is a result of the lack of information about errors.

3. The new method

The new method for error recovery in parsers generated by *Yacc* uses two techniques, local correction with a forward move, and phrase-level recovery as presented in [Leinius70]. When the parser meets an illegal input token, it first tries to make a local correction to the input string by changes of a single token. If no local correction is successful, where success is judged by the number of moves which can then be made by the parser, a phrase-level recovery is made by replacing a part of the input with a non-terminal. Both already parsed input and input still remaining may be replaced.

3.1. Local correction

The set of tokens which are legal shift symbols for the current configuration is determined by the current state. The parser attempts to repair the input by actions in the following order: inserting a token from this set on the input before the next token, deleting the next token, or replacing it with one from the set of legal tokens. In order to determine whether a repair is "good" the parser runs a forward move on the repaired input. This is achieved by copying some of the parse stack onto an error stack, buffering the input and turning off the semantic actions. The parser then restarts from the error state (the state in which it detected error), with the altered input. If the parser can continue to make moves without detecting a further error before five input tokens are shifted, or before accepting, the alteration is taken to be a good repair. The parser is returned to the error state and the parse stack, the input is backed up to the chosen alteration, and semantic actions are turned on again. If an alteration does not allow the parser to run a forward move which consumes five tokens from the input, a forward move is run with the next altered configuration from the set above.

3.2. Phrase-level recovery

If no local correction succeeds, the parser is restored to the error state and the input is backed up to the illegal token. The parser chooses a goal non-terminal from the set of kernel items for the current state. Its item has the form

$A \rightarrow v_1 \cdots v_m \,.\, v_{m+1} \cdots v_n$

where the v_i are grammar symbols. The phrase to be replaced by the goal non-terminal A is $v_1 \cdots v_n$. $v_1 \cdots v_m$ have been parsed, so the parser pops m states from the stack and pushes the goto state for the new top of stack and A. Further reductions may now take place. To complete the recovery, input is discarded until the next input token is legal for the current state. In effect, a reduction by the production $A \rightarrow v_1 \cdots v_m v_{m+1} \cdots v_n$ has taken place.

A heuristic rule is used to choose the goal phrase from the kernel items of the error state, namely the last item to have been added to the kernel during construction of the item sets, except for the special case of state 0, where the first item is chosen.

The scheme is guaranteed to terminate, because it always consumes input tokens during a successful repair.

3.3. Changes to *Yacc* input specifications

Error productions are no longer required for error recovery and so may be deleted from grammar rules. The user must supply a routine ***yyerrlval*** as part of the *Yacc* environment. The purpose of this routine is to supply a default semantic value which is required for tokens inserted during local correction and for non-terminals used as goals for phrase-level recovery. This semantic value will typically be a leaf of the syntax tree, suitably tagged.

3.4. Error messages

The parser synthesises an error message from the recovery action taken in each case. It use the terminal and non-terminal names from the input grammar to *Yacc*. Examples of messages are

SEMICOLON inserted before RIGHTCURLY

for a successful local repair and

e ASSIGN IF NAME replaced by e

for replacement of a phrase.

3.5. Space requirements

Parsers generated by the new *Yacc* require extra space for information for phrase-level recovery and diagnostic messages. No extra space is required for local recovery, as the information required, the valid shift symbols for each state, is present in the existing tables. For phrase-level recovery, two extra words are required for each state, the goal non-terminal and the number of symbols to pop from the stack. Tables of strings are needed for synthesizing diagnostic messages; one string is required for a meaningful name for each grammar symbol, excluding literal tokens.

The parser generated by the new *Yacc* will have fewer states than the equivalent parser generated by the old *Yacc*, because there are no error productions in its grammar. The space-saving device of default reductions for all states with a single reduction is still used.

4. The C compiler

The existing C compiler *pcc* contains a syntax analyzer which is generated by *Yacc*. We took the source of this compiler, removed the error productions from the *Yacc* specifications and included a new function *yyerrlval* which returns a semantic value for inserted tokens and non-terminals. This value is a new leaf of the syntax tree. The only other changes made were to the names of some of the terminals, such as changing SM to SEMICOLON and RC to RIGHTCURLY, to improve the error messages.

The relative sizes of the old and new C compilers are shown in Figure 1.

	pcc	new version
Size in bytes of binary (ccom)	86776	98312
Parser only:		
Number of grammar rules	187	179
Number of states	312	303
Size in chars of source (y.tab.c)	42980	54617

Figure 1. Space required by the C compilers

It is obvious that the compiler performs identically to *pcc* on C programs which are syntactically correct. Error recovery for incorrect programs consists of repairs to the input and error messages. For the new compiler, repairs may be simple changes of one token or replacement of a phrase, and error messages describe the repairs. For the old compiler, error messages do not describe the action taken by the parser to recover, but are either uninformative ("Syntax error") or indicate what the parser finds incorrect.

In order to test the compiler's performance on incorrect programs, we made a collection of all C programs submitted by undergraduate students in the Department of Computer Science at the University of Warwick to *pcc* for compilation over three twenty-four hour periods, October 9, 10 and 16, 1984. Duplicate programs were removed and the programs were run through *pcc* and the new compiler. The code generated for syntactically correct programs was identical. Error recovery was evaluated according to the criteria used by Sippu [Sippu83], rating a correction as excellent if it was the same as a competent programmer might make, good if it introduced no spurious errors and missed no actual errors, fair if it introduced one spurious error or if it missed one error, and poor otherwise, or if the error message generated was meaningless. Missed errors, that is syntax errors that were present in the source code but not reported by the compiler, were counted. Also counted was the number of extra messages, that is messages about errors introduced into the source by incorrect recovery action taken by the compiler. A comparison of the performances of the two compilers, evaluated according to these criteria, is shown in Figure 2. Figure 3 shows a sample C program and its diagnostics.

	pcc		new version	
Quality of recovery action:				
Excellent	1%	(1)	54%	(64)
Good	3%	(3)	11%	(13)
Fair	54%	(64)	13%	(15)
Poor	27%	(32)	19%	(22)
Missed errors	15%	(18)	3%	(4)
Total number of errors	118		118	
Extra messages	127		82	

Figure 2. Comparison of the performance of the C compilers

```
 1   /* Kernighan and Ritchie p. 102 - mutilated */
 2
 3   strcmp(s, t)
 4   char *s, *t
 5   {
 6      for ( ; *s == *t; s++, t++)
 7         if *s == ' '
 8            return(0)!
 9      return(*s - *t);
10   ? }
```

Diagnostics from pcc:
   Line 5: Syntax error
Diagnostics from new version:
   Line 6: SEMICOLON inserted before LCURLY
   Line 7: LPAREN inserted before MUL
   Line 8: RPAREN inserted before RETURN
   Line 9: UNOP replaced by SEMICOLON
   Line 11: QUEST deleted

Figure 3. A sample C program

5. Discussion

The C compiler generated by the new *Yacc* performed better on the collection of incorrect programs than *pcc*. The majority of the errors were simple ones which occurred sparsely, and were therefore amenable to repair by the local recovery tactic. This pattern of occurrence of simple errors concurs with Ripley and Druseikis' analysis of syntax errors in Pascal programs [Ripley78], which showed that the majority of these are single-token errors and occur infrequently. Clusters of errors and complicated errors were not handled so well by the phrase-level recovery, and these were responsible for the large number of extra messages generated.

The diagnostic messages produced depend on the names for the terminals and non-terminals, which should be carefully chosen by the grammar-writer. Ideally, messages should be at source level rather than lexical token level, as the user will understand a message of the form

```
Line 16:          x = y
Semi-colon inserted ...... |
```

better than one of the form

```
Line 16: SEMICOLON inserted after ID
```

More communication between the lexical analyzer and the parser may be needed for this sort of message. Line numbers at present are occasionally out by one because of buffering of the lexical tokens.

A disadvantage is that the scheme shows bias towards assuming correctness of the left context. Local recovery assumes a single error in the current input token, and secondary recovery makes an arbitrary choice of item from the error state which takes no account of the right context.

Several other error recovery schemes for LR(1) parsers have been described. [Sippu81] and [Dain84] contain recent reviews of the literature. The scheme presented here bears resemblance to that devised by Graham, Haley and Joy for a Pascal compiler [Graham79], in that a two-stage recovery is attempted. There are several differences to note. Firstly, our scheme is a general-purpose recovery scheme incorporated in a new version of *Yacc*, and is used by any parser generated by the new *Yacc*. Graham requires special purpose error recovery routines and cost vectors to be supplied for use by their parser generator *Eyacc* which contains no error recovery scheme. Secondly, *Eyacc* produces parsers with certain states calling for reductions having their lookahead tokens enumerated, i.e. some default reductions are not made. Our *Yacc* has the usual default reductions. Thirdly, Graham requires the user to supply error productions in the grammar, to control secondary recovery. These are not required for our scheme.

The error recovery scheme for the compiler-writing system HLP [Raiha83] incorporates a local recovery tactic into the phrase-level recovery scheme [Sippu83]. No forward move is made on the input and there is less check on the "correctness" of a local correction; the user must supply costs for deletion and insertion of each terminal in local correction. Different criteria are used for identifying and replacing the error phrase in phrase-level recovery.

A two-stage recovery scheme for LL(1) and LALR(1) parsers which uses the concept of *scope recovery* is implemented by Burke and Fisher [Burke82]. The scheme cannot be used however in LR parsers with default reductions. The user must supply additional language information such as constructs which open and close scope in the language, lists of tokens which cannot be inserted between a given pair of tokens, and lists of tokens which cannot be substituted for a given token. Pai and Kieburtz [Pai80] use *fiducial* (trustworthy) symbols, typically reserved words, in a scheme for LL(1) parsers which they suggest as suitable for extending to LR parsers.

Requiring the user of a parser-generator to supply information to aid error recovery in addition to the grammar may result in recovery which is more tailored to the language, but imposes an extra burden on the user, who may not have a full understanding of the mechanism of the parser and its error handling. The scheme which we have implemented in *Yacc* makes few demands of this nature on its users, yet improves the quality of error recovery in its parsers.

6. References

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

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

Wesley, 1977.

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

[Dain84] Dain, J. A. Error recovery schemes in LR parsers. Theory of Computation Report No. 71, Dept. of Computer Science, University of Warwick, Coventry, December 1984.

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

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

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

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

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

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

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

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

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

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