To the Graduate Council:

I am submitting herewith a thesis written by Beverly Karen Button entitled "A Compile-Time Error Diagnostic System for an Interactive, Table-Driven Compiler." I recommend that it be accepted in partial fulfillment of the requirements for the degree of Master of Science, with a major in Computer Science.

Alan Davis

Alan Davis, Major Professor

We have read this thesis
and recommend its acceptance:

Michael G. Thomason

Ronald P Leirina

Accepted for the Council:

L Evans Roth

Vice Chancellor
Graduate Studies and Research

Thesis
77
.B877
cop. 2

A COMPILE-TIME ERROR DIAGNOSTIC SYSTEM FOR AN INTERACTIVE, TABLE-DRIVEN COMPILER

A Thesis

Presented for the

Master of Science

Degree

The University of Tennessee, Knoxville

Beverly Karen Button

December 1977

1345654

DEDICATION

This thesis is dedicated to the memory of my father and in anticipation of Spud (and other nieces and nephews to follow).

ACKNOWLEDGEMENTS

The author wishes to thank the University of Tennessee Graduate School for the Fellowship she received as financial support for her first year of graduate study. A heartfelt thanks is also given to the Department of Computer Science, for its financial support in awarding the author an appointment as a Graduate Teaching Assistant, for its support of the TICK project, and for its continuous endeavors to create for its students an environment that is both academically challenging and supportive of individual needs.

The members of the Computer Science faculty have contributed a wealth of knowledge, experience, and personal interest toward my professional career. To each of them, and especially to Professor Charles Hughes, I owe a great deal.

Professor Michael Thomason and Dr. Ronald Leinius, as members of my Thesis Committee, have given valuable criticism and contributed much time to the completion of this paper.

Professor Alan Davis has been not only a professionally demanding mentor but also a valuable friend. To him I owe most sincere thanks.

My family has been a source of encouragement and support in many times of need. I want to thank each of them: my mother; my sisters, Lynn, Barbara, and Cathy; my brothers-in-law, Steve and Walt; and my grandparents.

Finally, I want to express my love and dedication to Jim, who has given me everything.

ABSTRACT

Many student-oriented compilers currently in use were designed to produce good error diagnostics. Most of these, however, are batch-oriented and fall far short of their two secondary goals: (1) to provide the user consistently with an accurate diagnosis of his problem, and (2) to detect all syntax errors in a given input.

The only method of correctly diagnosing a user's error with consistently accurate results is for the error system to interact directly with the user and rely on human feedback to guide it to the "correct" solution of the problem. The compile-time diagnostic system of CAPS (Computer Assisted Programming System) at the University of Illinois has been the only such highly interactive error system designed. Tindall's table-driven, language-independent system examines only the left context of the error, a syntax description of the language, and the parser's configuration to determine possible corrections for the error.

This thesis describes a modified version of Tindall's model and its implementation in the TICK (Table-driven Interactive Compiler from Knoxville) system at the University of Tennessee. By incorporating the right context of the error into the investigation, this work significantly strengthens Tindall's model.

Key words and phrases: table driven compilers, error correction interactive programming.

TABLE OF CONTENTS

CHAPTER		PAGE
1.	INTRODUCTION	1
2.	SURVEY OF AUTOMATIC CORRECTION TECHNIQUES	4
3.	A TRANSITION DIAGRAM PARSING MODEL	10
	Introduction	10
	Transition Diagram Systems	10
	Implementation of Transition Diagram Systems	13
	Parser Environment When An Error Is Detected	16
4.	TINDALL'S ERROR DIAGNOSTIC SYSTEM	21
	Caps Compiling Environment	21
	Design of Tindall's System	22
	Discussion of Tindall's Algorithms	27
5.	MODIFIED MODEL AND ITS IMPLEMENTATION	35
	TICK Compiling Environment	35
	Modification of Tindall's Model	38
	Changes in Implementation	43
6.	CONCLUSION	51
	Evaluation of TICK Error System	51
	Future Research in Error Diagnostic Systems	51
	Conclusions	54
LIST OF REFERENCES		55
APPENDICES		58
APPENDIX A:	TINDALL'S ALGORITHMS AND NOTATIONS	59
APPENDIX B:	BUTTON'S ALGORITHMS	67
VITA		74

LIST OF FIGURES

FIGURE		PAGE
3.1.	Simple transition diagram	12
3.2.	FORTRAN "Go To" statement transition diagram, example of general option	12
3.3.	FORTRAN logical IF statement transition diagram, example of one TO invoking another	12
3.4.	Arithmetic and logical IF statements and logical expression diagrams, example of multiple exit extension	14
3.5.	General organization of Syntax Program	15
3.6.	Form of a PROC-END block	15
3.7.	Simplified procedure in Tindallian to parse an IF statement	17
3.8.	Examples of initial error messages which might be supplied by a FORTRAN language designer	20
4.1.	Example of PLATO graphics capability as used by CAPS execution-time error system	23
4.2.	Example of CAPS compile-time error system dialogue	25
4.3.	Example of Tindallian representation for simple transition diagram	28
4.4.	A system of transition diagrams	30
4.5.	Tindallian representation of system of transition diagrams in Figure 4.4	31
4.6.	Example of Fortran diagram with prunable arcs	33
5.1.	Examples of confusing error messages produced by CAPS	40
5.2.	Example of TICK suggestion message which reconstructs the invalid FORTRAN line shown in Figure 5.1(b)	42
5.3.	Comparison of CAPS and TICK error messages	42
5.4.	Sample of error system dialogue using FORTRAN example	49
5.5.	Example of detailed dialogue using FORTRAN example	50

CHAPTER 1

INTRODUCTION

The area of error diagnostics has been the focus of much research in the past few years, and many compilers have been designed with diagnostic capability as a primary goal. To experienced programmers, the improvement in compile-time diagnostics has been astounding. But one limitation of almost all highly diagnostic compilers which have been developed is their orientation toward a batch environment. The major goals of such compilers have been (1) to provide a clear description of the error situation, and (2) to signal as many of the existing errors as possible during the current compilation. In order to detect additional errors in the input string which occur beyond the originally detected error, a repair must be made to the input string, either at or before the point of the original error. It may also be necessary to adjust the parser's configuration to a "safe state" to allow the current compilation to continue. Early correction attempts consisted of skipping over a portion of the input string and leaving unflagged errors to be encountered during the next compilation. Later efforts modified the error's immediate environment by local insertion and replacement of symbols, in addition to the earlier "panic mode" deletion of symbols. These systems could occasionally make valid suggestions, but much more often the repaired section of code was not what the programmer had intended [Livecky, 1976]. The diagnostic error of these batch compilers

was limited because of the lack of interaction with the student during compilation.

Tindall [Tindall, 1975b] was the first to develop an automatic error system which was also interactive. His system actively involved the user in an investigation of the error by making suggestions as to possible ways to "fix" it, and prompted the user to select from perhaps many plausible suggestions the one which most closely corresponded to his original intentions. The student was forced to correct the error before compilation could proceed, thus both eliminating any need for any automatic, artificial "repair" of the error, and reinforcing the correct syntactical form in the student's repertoire. As a result, the occurrence of multiple errors of the same type within the same program was lessened in the following way: assuming the student is typing from a handwritten copy of his program and correcting mistakes as they are typed in, he could also be editing the rest of the handwritten program to reflect his new knowledge. This not only strengthens the student's grasp of the syntax rules, but also minimizes duplication of effort by the error system and the compiler.

Tindall's work in this area was a great breakthrough in providing meaningful on-line consulting service to beginning programmers, but his implementation was severely hampered by restrictions placed on the CAPS system by the PLATO environment [Tindall, 1975b]. The power of his analysis was also significantly limited by the fact that he could not take advantage of the information contained in the right context of the error.

This thesis describes the design and implementation of an error diagnostics system based on Tindall's, but with changes in his algorithms and differences in implementation which make his basic model

much more powerful as a highly interactive, diagnostic consultant. Chapter 1 has presented arguments for the implementation of a highly interactive and accurate consultant for the novice programmer. Chapter 2 will give a survey of previous automatic error correction and analysis techniques. Chapter 3 will provide a description of the parsing model used in both Tindall's and the author's implementations. Chapter 4 will describe Tindall's error system model, and Chapter 5 will present the author's adaptations to the model and its implementation. Concluding remarks will be given in Chapter 6.

CHAPTER 2

SURVEY OF AUTOMATIC CORRECTION TECHNIQUES

The following discussion will present a brief survey of recent developments in the area of automatic error correction. It is assumed that the reader has some background in formal language theory and an understanding of the basic parsing methods frequently used in compiling. A detailed description of these areas may be found in Aho and Ullman [Aho, 1972], and Gries [Gries, 1971].

The earliest form of automatic error correction was the "panic mode" approach. The goal of this method is merely to adjust the parser to a "safe state" so that compilation can proceed. When an error is detected, the system skips ahead in the input until one of a class of special symbols is encountered, and adjusts the parsing stack accordingly. This may cause the compiler to miss errors in the skipped portion of the input, but will allow the detection of some errors in the rest of the program.

Some other early attempts at automatic correction used tables of "error rules" provided by the language designer which specified the correction to be taken for all possible syntax errors. As pointed out by Leinius [Leinius, 1970] in a thorough discussion of a system by Wirth [Wirth, 1966], it would be quite difficult for a language designer to specify all the information necessary for the correction of all possible syntax errors using this method. However, an extension of the model to an adaptive system could be one method of possible improvement.

A more language-independent method was proposed by Irons [Irons, 1963]. His system requires a multiple parse of the input string until none of the "active" parallel parses can continue, thus signalling a detection of an error. A list is then formed of all terminal and nonterminal symbols which would allow the previously active parses to continue, and the input is skipped until one of the terminal symbols (or a terminal symbol which could begin one of the nonterminal symbols) is found. The input string is modified (by insertion of terminal symbols) in the simplest way which will allow the parse to continue at least one input symbol past the modified string.

Leinius [Leinius, 1970] based his error correcting method on a simple precedence parsing algorithm, and emphasized the importance of "phrase-level" correction. His system attempts to find the smallest substring in the input containing the error which can be reduced to a unique nonterminal. The "recovery" phase then forces the appropriate stack reduction, adjusts the input location pointer, and proceeds. He concludes his discussion with a description of how his method could be applied to the class of LR k language.

James [James, 1972] discusses an error recovery system which uses a bottom-up LALR k parser described by DeRemer [DeRemer, 1971]. During the parse, a "configuration set" is maintained which includes all possible right-hand-sides of production rules in the grammar. Each member of the configuration set includes a marker which indicates the current state of its parse. An error is detected when the next input symbol is invalid for all members of the active configuration set (no marker can be advanced). The recovery scheme involves popping

the parse stack and skipping the input until a configuration set is found which allows a reduction to a nonterminal that is valid at the updated input location. The configuration set then becomes the "current" one, is updated to reflect a parse of the nonterminal, and parsing proceeds.

Graham and Rhodes [Graham, 1975] have developed an error correction scheme for a bottom-up, no back-up parser which recovers from an error situation in two phases: (1) the "condensation" phase involves an analysis of the context of the error, and (2) the "correction" phase changes the condensed parsing stack so that the error situation is corrected.

The "condensation" phase has two "moves." The "backward move" makes all possible reductions to the stack preceding the point of error detection, and the "forward move" attempts to parse the input located beyond the point of error detection. The "forward move" terminates either when a second error is detected, or when the only possible parsing action involves a reduction of the substring containing the error. An additional backward move may be performed if necessary.

The "correction phase" follows, and involves a modification which combines the insertion, deletion, and replacement of stack symbols. The modification with the least "cost" (determined by vectors which specify the cost of inserting, deleting, and replacing each symbol in the language) is chosen as the correction. If the minimum cost is above a fixed a priori maximum, the system resorts to a form of "panic mode."

The article by Graham and Rhodes also re-classifies the errors detected by simple precedence parsing methods. In an attempt to detect an error at a point in the input string which is closer to its cause, they incorporate a check for a "nonvalid right-hand-side error" into the parser (this is similar to the maintenance of configuration sets as proposed by DeRemer). They state that errors detected by simple precedence parsers fall into two categories: (1) stackability errors which occur when a reduction of the stack results in the top elements not having a $\doteq$ relation between them, and (2) nonvalid right hand side errors which occur when the top of the stack ceases to form a valid prefix for a RHS included in the production rules (this corresponds to the inability of DeRemer's configuration set markers to the advanced). Character pair errors are stackability errors.

Another system which maintains a set of all possible parses is the least-errors recognizer of Lyon [Lyon, 1974]. A least-errors recognizer attempts to determine the minimum number of insertions, deletions, and replacements necessary to transform an incorrect input into a valid sentence of the language being analyzed. To do this, not only are legal parallel parses constructed, but also all parses which involve deletions of input tokens. For each parse, a count is kept of the number of errors that have occurred for that parse. The parse with the smallest error count is selected as the final parse. As noted by Levy [Levy, 1971], a minimal distance correction may not be the best correction from the programmer's standpoint.

An adaptive system has been produced by James and Partridge [James and Partridge, 1973] which views the syntax definition of a correct program as a binary tree structure, where nodes are representations of either single characters or groups of characters invoked as a subtree. The system can use its parsing history to restructure the tree so that the commonly used parsing paths are tried first. An error is detected when a "dead end" is reached, or when the current input symbol has been compared to each of the possible characters on the current limb of the tree.

Upon detection of an error, a "strategy tree" is used to match the current configuration to a known error situation. Once completely specified, this tree can also be adapted so that common error situations are tested for before less common ones.

La France [La France, 1971] proposed an error system based on a Floyd Production System parser [Floyd, 1963]. His system generates a list of all two- or three-symbol strings which could be accepted at the point of the error, and matches each of these strings to the next four symbols in the input. Each comparison assumes that a particular type of error has occurred in the input. If a match is found between a possible valid substring and the modified form of the input, the correction is made and parsing resumes. If no match is found, the system resorts to a "panic mode" form of error recovery.

The error diagnostic system discussed in this thesis is a modification of one implemented by Tindall [Tindall, 1975b] which was based on the work of Levy [Levy, 1971]. Levy attempted to find a theoretical basis for error correction by all deterministic context-free parsing methods having the correct prefix property. His approach

was to (1) back up in the input until the beacon location is reached (the beacon location is the left most input location beyond which the actual syntactic error can not occur); (2) generate all interpretations of the input string requiring k or fewer changes, until the right context location is reached; (3) choose the interpretation which requires the least number of changes as the "correction." To generate possible interpretations, three operations are performed: "R" (replacement of the input token with a valid token), "A" (addition of a valid token at the input location), and "S" (suppression of the input token).

Levy proves that all syntax errors can be corrected by the use of these three operations. Tindall adapted the system proposed by Levy to an interactive system, and increased its efficiency by generating the possible interpretations only as required by the user. He also reversed the order in which the interpretations are constructed by first generating the interpretations for the error location, and then backing up, token by token, until the beacon location is reached. At each input location, various interpretations are generated as necessary.

Tindall extended Levy's system of operations to nonterminal symbols represented by strings within the input. This allowed Tindall's system to produce error messages which referenced constructs in the language as well as tokens, and significantly increased the power of a single operation.

CHAPTER 3

A TRANSITION DIAGRAM PARSING MODEL

3.1 INTRODUCTION

The parser design used in both the CAPS and TICK compilers is a top-down parser using recursive descent. An important modification of the recursive descent model, however, is that a single recursive procedure may be written to recognize the instance of one of a set of nonterminals. The particular nonterminal recognized by such a procedure is passed back to the calling procedure and used to determine which of a number of alternatives should be chosen. This modified model is equivalent to the separable transition diagrams of Conway [Conway, 1963] and has been shown to be sufficiently powerful to recognize the set of deterministic context free languages [Lomet, 1973].

3.2 TRANSITION DIAGRAM SYSTEMS

The transition diagram as described by Conway may be viewed as a finite state machine with stack capabilities. A finite state machine traverses its representative graph by examining the current input symbol and comparing it to the set of labeled arcs which originate at its current state. If the input symbol matches one of the arcs leading from that state, then the arc is traversed, the current state pointer is updated, and the next input symbol is examined.

Similarly, a transition diagram is a graph which may represent the possible parsing paths for a given language construct. The set of arcs originating at the current state then define the possible valid input symbols for the parser at that state. Figure 3.1 shows a simple transition diagram which could be used to parse the set of strings "abcd", "acd", and "be".

Figure 3.2 gives an example of a transition diagram which might be used to parse a FORTRAN "GO TO" statement. Note that the arc originating from state S2 does not define a particular input symbol. Rather, it describes a general class of symbols, any one of which is valid at the current state. Note also that each arc represents a "token" in the language, such as produced by lexical analysis of the program.

For the examples shown, a finite state model has been sufficient. The expansion from this model to Conway's system of transition diagrams necessitates the creation of another type of label for the arcs of the graph--an invokation of a complete transition diagram. That is, a label may represent either a terminal or a nonterminal in the source language. Figure 3.3 shows an example of this type of transition. It should be noted at this point that an arc of a given transition diagram may also invoke itself, the only restriction being that "left recursion" (for example, the arc from S0 being labeled as <logical if statement>) is not allowed.

A final extension to the model will enable a single transition diagram to return to the invoking diagram after having found one of a set of possible constructs. This is necessary, for example, in

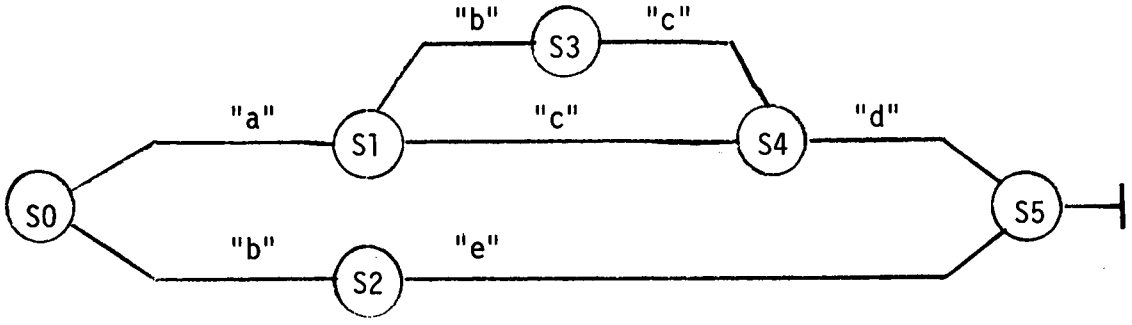


Figure 3.1. Simple transition diagram.

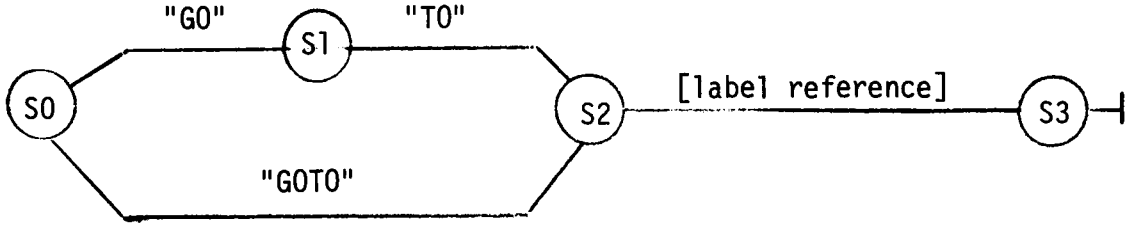


Figure 3.2. FORTRAN "GO TO" statement transition diagram, example of general option.

<Logical IF statement>

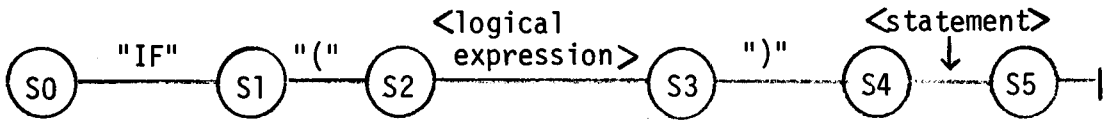


Figure 3.3. FORTRAN logical IF statement transition diagram, example of one TO invoking another.

parsing FORTRAN Logical and Arithmetic IF statements. It is impossible to determine which form of the IF is being used until the parser examines the input following the first left parenthesis. For an Arithmetic IF, that input should form an arithmetic expression and be terminated by a closing right parenthesis. On the other hand, an arithmetic expression is also a valid beginning for one simple form of the logical expression (<arithmetic expression><relational operator><arithmetic expression>) required in a Logical IF. For these cases, diagrams with "multiple returns" which may invoke diagrams with "multiple exits" are used. An example of the use of multiple exits is shown in Figure 3.4.

3.3 IMPLEMENTATION OF TRANSITION DIAGRAM SYSTEMS

To facilitate the specification of language syntax using transition diagram systems as a model, Tindall developed an assembler-level language (hereafter referred to as Tindallian) in which the syntax-defining tables are written, and which the parser interprets during compilation. Only those instructions which are necessary to the description of the error handler's implementation will be discussed in this thesis. A more detailed description may be found in reports by Tindall [Tindall, 1975a] and Widler [Widler, 1977].

The general organization of a syntax description program is given in Figure 3.5. Within the global "main program," many PROC-END blocks may be defined. Each of these procedures represents a transition diagram within the parser's framework. The general form of a procedure is given in Figure 3.6. Within a given procedure is a sequence of SCAN instructions (requests for the next token in the input string), each

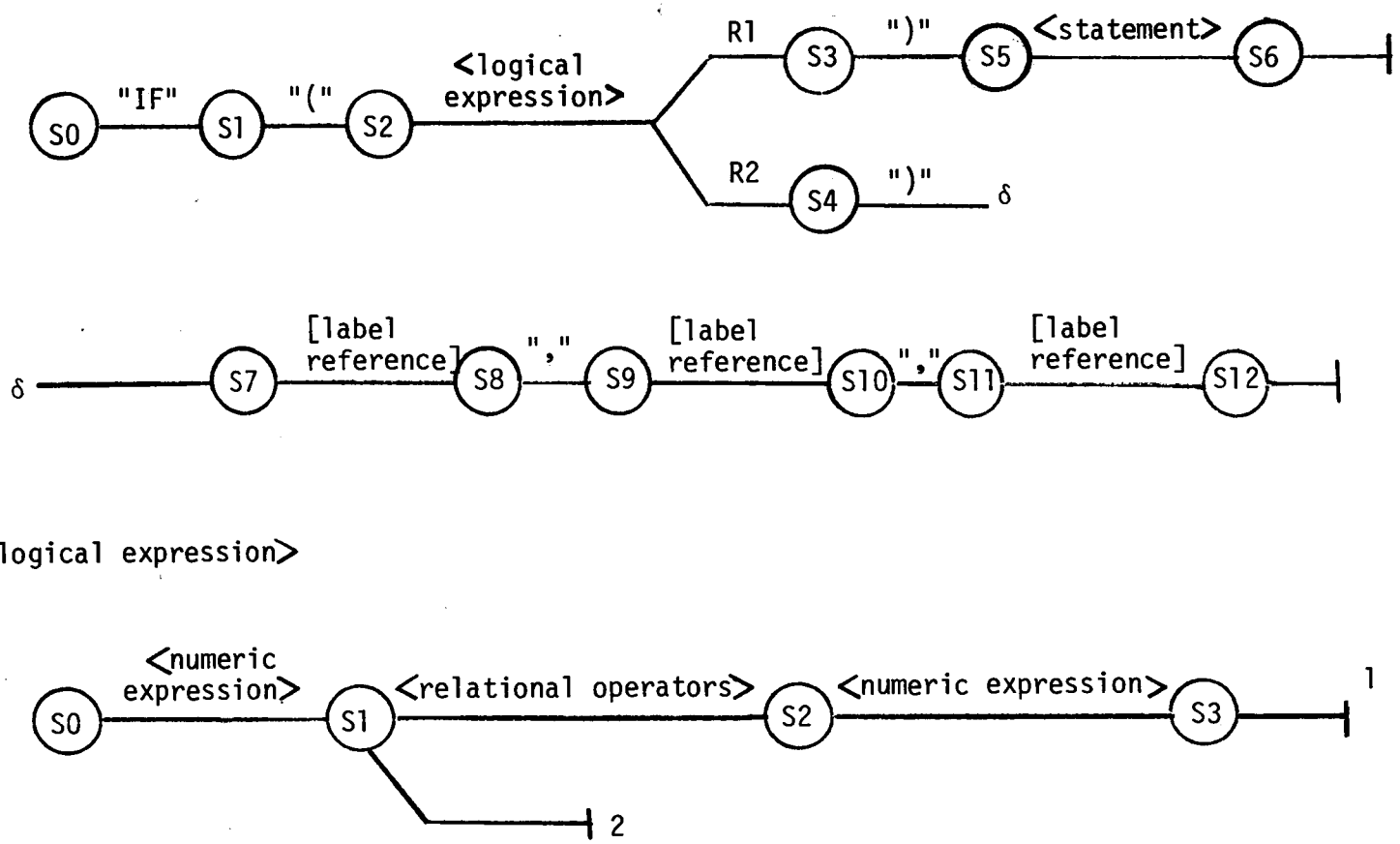


Figure 3.4. Arithmetic and logical IF statements and logical expression diagrams, example of multiple exit extension.

```
{ Definition of mnemonics for constants used }
{ Allocation of global variables             }
{ Definition of texts for error system       }
  Main parsing procedure
{ PROC - END blocks                          }
  END SYNA
```

Figure 3.5. General organization of Syntax Program.

```
  PROC <proc name> [(<arg>{,<arg>} )] [RETURN <#rets>]
[NAME (<error text>)]
{ Local variables allocated                  }
  Proc body procedure
  END PROC
```

Figure 3.6. Form of a PROC-END block.

followed by code which tests certain information concerning the state of the parser and the current input token in a determination of whether the current input symbol is valid. If so, the parser's current state pointer is updated to the next SCAN, and parsing continues. If the series of tests on available information reveals that the current input symbol is invalid a BRANCH is made to the error diagnostic system. Necessary information about the input token is provided primarily by system variables which give the current token's location in the symbol table (STP), its CLASS, and its SUBCLASS. A sample of the tests which might be made and the method of invokation which might be used during the procedure which parses a FORTRAN IF statement (as defined in Figure 3.4) is given in Figure 3.7. The syntax of the instructions has been slightly modified in order to improve readability; the actions of the instructions are as defined in Tindall's work.

3.4 PARSER ENVIRONMENT WHEN AN ERROR IS DETECTED

When an error is detected, a BRANCH in the syntax tables causes the parser to give control to the error diagnostic system. The parser, as well as the rest of the compiling system, assumes that all necessary adjustments of the parser's configuration will be handled by the error system, and that upon return from the system, editing will continue as normal. For this reason, all data structures which define the compiler's current state are made available to the error system, and are modified by it as necessary.

In order to provide immediate diagnostics, especially in the case when the student is editing a previously compiled line, "decompiling"

```
          PROC IF_STMT NAME (IF STATEMENT)
             |
             |
             | Local definitions of variables
             |
          BRANCH      STP ≠ IF_STP, ERROR
          SCAN
          BRANCH      STP ≠ LEFTP_STP, ERROR
          SCAN
          CALL        LOG_EXP THEN GOT_A_LOGEXP,  GOT_ARITH_EXP
GOT_A_LOGEXP BRANCH   STP ≠ RIGHTP_STP, ERROR
          SCAN
          CALL        STMT
          RETURN
GOT_ARITH_EXP BRANCH  STP ≠ RIGHTP_STP  ERROR
          SCAN
          BRANCH      CLASS ≠ LABEL_REF, ERROR
             |
          SCAN
          BRANCH      STP ≠ COMMA_STP,  ERROR
             |
             |        Parsing of 2 more label references
             |
          RETURN
          END PROC
```

Figure 3.7. Simplified procedure in Tindallian to parse an IF statement.

must be performed whenever the student moves the cursor backwards through the program. To facilitate this process of decompiling, a parser history is maintained which records all changes made to the compiling environment for each token. This provides the editor, compiler, and error system with a convenient method of backing up through the parser tables. In particular, the error system must back up in order to search for possible corrections to the error. This history trace can become quite large; remedies for this problem differ between the CAPS and TICK implementations, and will be described in Chapter 5.

Another important tool for the error system is a set of language-dependent string texts which may be provided by the language designer. These texts are related to four concepts common to different languages. The first and most important of these is the construct represented by a single transition diagram within the syntax description. Each procedure can have a corresponding "name" associated with it for use by the error system. An example of this is shown in Figure 3.7, where the procedure which parsed FORTRAN IF statements is referenced within the Tindallian code as the mnemonic IF_STMT, but is associated with the string text "IF STATEMENT ".

Another important concept which may be described by a language-dependent text is the syntactic class of a given token in the language. When defining a language's lexical tables, the language designer dictates the syntactic class which should be assigned to each token recognized by the lexical analyzer. A one-to-one correspondence may then be maintained between each class of tokens and a text which could be used to describe any member of it, such as "DEFINED VARIABLE NAME"

or "PUNCTUATION SYMBOL". The set of texts which describe subclasses is used in conjunction with the set of class texts. These may serve to further define members of the same class, such as allowing both "REAL" and "INTEGER" members of the class "DEFINED VARIABLE NAME".

The fourth type of text available to the error system is used in languages which place restrictions on the use of specific columns in the source program line. The field defined by columns 1 through 5 in FORTRAN is an example of this, and may be associated with the text "LABEL FIELD". COBOL also uses field descriptions, such as "AREA A".

An initial message for any particular BRANCH to the error system may also be supplied. These are primarily used when the cause of the error is global in nature or, in the language designer's opinion, the error system's automatically generated initial message would be significantly less informative than one which he could supply. Examples of some error messages which a FORTRAN language designer might supply, and examples of their related error situations, are given in Figure 3.8.

```
      GO TO 15
      IF (A .EQ. B) GO TO 10
```

"A label is necessary on the statement following an unconditional GOTO."

```
      IF (X. NE. Z) DO 10 I = 1, 5
```

"A DO statement may not be the executable statement with an IF."

```
      INTEGER I, J
      I = 4
      REAL X, Y, Z
```

"All specification statements must precede the first executable statement."

Figure 3.8. Examples of initial error messages which might be supplied by a FORTRAN language designer.

CHAPTER 4

TINDALL'S ERROR DIAGNOSTIC SYSTEM

4.1 CAPS COMPILING ENVIRONMENT

The CAPS compiler is incremental, that is, it compiles each character as it is typed by the student. It is also multilingual, requiring tables which specify the lexical and syntactic requirements of the source language being compiled. Designed for use by beginning programmers, it provides excellent compile- and run-time diagnostics in a highly interactive, "consulting" atmosphere.

The PLATO IV system (Alpert and Bitzer, [Alpert, 1970] on which CAPS is implemented imposes severe limitations on and provides extraordinary capabilities to an interactive compiler [Tindall, 1975b]. Because it was designed as a large-scale computer-aided instruction system able to support as many as 1000 simultaneous users, the amount of central processor computing time allowed per user is very restricted. For that reason, all algorithms for lexical, syntactic, and error analysis in CAPS were required to be fairly efficient.

A hindrance to efficiency imposed on CAPS due to its status as a user program on PLATO was that it had to be written in TUTOR, a high-level language simular to FORTRAN. Unlike FORTRAN, TUTOR is interpreted; thus, the CAPS compiling system, which interprets lexical and syntactic tables, is bogged down in at least two levels of interpretation during compilation of the source text.

The PLATO student terminals provided graphic capabilities to the compiler and error systems which greatly enhanced their ability to communicate complex ideas to the student. An example of the type of run-time consulting which is provided by a system developed by Davis [Davis, 1975] is shown in Figure 4.1 (figure taken from Wilcox, Davis, and Tindall [Wilcox, 1976]). This level of automatic system communication is quite impossible without graphics capabilities.

The CAPS system has a remarkable improvement in diagnostic capability over previous compilers. However, due to the excessive slowness of the system (from causes described above) and the resultant degrading of interaction, the CAPS system is no longer in use.

4.2 DESIGN OF TINDALL'S SYSTEM

The error system developed by Tindall was designed to provide beginning programmers with the best diagnostic aid possible under the constraints of PLATO. The fact that the compiling system was multilingual required a general algorithm for the generation of error messages, while the goal of being as helpful as possible indicated the use of language-specific terms. In order to take advantage of the system's interactive nature, a high degree of flexibility was needed to allow partial guidance of the system's actions by the programmer. At the same time, the system needed to be easy enough for beginning students to use.

Tindall's resulting error system is one which incorporates all of these design goals, system capabilities and constraints into a powerful automatic consultant. The type of system-student interaction chosen by Tindall will now be described.

EXECUTION-TIME ERROR ANALYSIS

```
EXAMPL2:  PROCEDURE OPTIONS (MAIN);
          DECLARE A FIXED:
          GET LIST (A):
          A = [1/A] ;
          A =  1/[A];
          END:
```

POSITION OF ERROR

This statement gave A an incorrect value of Ø.

This division operation performed here was integer division. In this case, the resulting value was zero because the numerator was less than the denominator.

Press [BACK] to edit your program; [HELP] for more help.

Figure 4.1. Example of PLATO graphics capability as used by CAPS execution-time error system.

Upon detection of an existing error, the system displays a blinking box around the invalid token. If the student (who is currently editing the program) tries to move the cursor beyond the error, his editing command is ignored. If he tries to move the cursor backwards, or to change the offending token, the blinking box disappears and his command is processed. Otherwise, he may press a key labeled "HELP" to receive aid from the automatic system. If help is requested, the system begins to generate a series of suggestion messages which describe possible corrections for the error. After each message is displayed, the student may choose between returning to the editing process and receiving a different suggestion. When all correction possibilities have been exhausted, the system displays only the last suggestion made (erasing the "help available" line), and returns to the editor when a valid editing command is issued by the student. Figure 4.2 gives a sample of the discussion which takes place (Wilcox, Davis, and Tindall, [Wilcox, 1976]).

Tindall's error correction algorithm is an adaptation of one proposed by Levy [Levy, 1971]. As mentioned in Chapter 2, Levy proved that all syntax errors can be corrected by some combination of the following three operations on an input string: "R" (replacing the input token with one which is acceptable at that point), "A" (adding an acceptable token in front of the input token), and "S" (suppressing the input token from the input string). Levy's algorithm for error correction is as follows: upon notification of an error, the system backs up through the input string until the beacon location (discussed in Chapter 2) is found; then a list is formed of all possible

```
E:      PROC;
        DCL I, A (1Ø);
        GET LIST (A);
        I = A ( 1  ;

**********ERROR**********                    BACK or ERASE to fix.
This punctuation symbol is not permitted here.
-------------------------------------------------------------------
 Press HELP for more information.


E:      PROC;
        DCL I, A (1Ø);
        GET LIST (A);
        I = A (1  ;

**********Possible Correction**********     BACK or ERASE to fix.
Replace [      ] with an arithmetic operator.
-------------------------------------------------------------------
 Press  HELP  for a different suggestion.


E:      PROC;
        DCL I, A (1Ø);
        GET LIST (A);
        I = A (1  ;

**********Possible Correction**********     BACK or ERASE to fix.
Insert ")" in front of  [      ] .
```

Figure 4.2. Example of CAPS compile-time error system dialogue.

interpretations of the input string with k fewer changes (for some fixed k, made between the beacon location and the right context location); the interpretation with the least number of modifications is then assumed to be the desired correction.

The development of Tindall's system involved a number of modifications to Levy's model. Taking advantage of his system's interactive capabilities, Tindall decided to avoid generating a complete list of all interpretations by choosing instead to generate each interpretation only as requested by the user, thus both substantially reducing central processor usage and increasing the student's active participation in the diagnostic process.

Another important modification is that Tindall begins building his list of corrections at the point of the error's detection rather than at the beacon location. He does this by systematically backing up through the input string, token by token. At each input location, the list of possible interpretations is updated with those which involve one of the Add, Replace or Suppress operations at the backed-up position. This method also orders the suggestion messages by giving "local" corrections to the student first, followed by those which involve changes farther away from the error.

Tindall's final change to Levy's model was dictated by the character-by-character incremental compiler of CAPS. Since the student is in the process of editing the current line when an error is signalled, any residual right context information may have little constructive information for the error

system. If an analysis had been made of the incomplete right context, its effect on the error system could easily have been detrimental to producing an accurate diagnosis. As will be discussed in Chapter 5, this absence of right context information is the major weakness of Tindall's implementation.

4.3 DISCUSSION OF TINDALL'S ALGORITHMS

Appendix A contains the algorithms (as given in [Tindall, 1975b]) which defined Tindall's error system model. A brief description of Tindall's algorithmic notation is also provided. The purpose of the following discussion is not to describe in detail the algorithms given (a full description may be found in [Tindall, 1975b]), but to give the reader the level of understanding of the model which is necessary to appreciate Tindall's implementation.

As described previously, Tindall's system begins at the error location and generates all necessary interpretations of the input involving k or fewer changes. An important part of this process is the formation of a list of all tokens acceptable to the parser at the given state.

Chapter 3 included a description of the Tindallian language and its use in implementing transition diagram systems. Figure 4.3 presents a sample diagram (a) and its Tindallian representation (b). Table 4.1 lists all "token options" for each state within the diagram.

The role of nonterminals in diagram systems was discussed in Chapter 3 during presentation of the parsing method. An example of a

S:

(a) Sample transition diagram

```
        PROC S
        BRANCH TOKEN ≠ "a", IS.IT.B
        SCAN
        BRANCH TOKEN ≠ "c", ERROR
        SCAN
        RETURN
IS.IT.B BRANCH TOKEN ≠ "b", ERROR
        SCAN
        BRANCH TOKEN ≠ "d", ERROR
        SCAN
        RETURN
        END PROC
```

(b) A Tindallian representation for (a)

Figure 4.3. Example of Tindallian representation for simple transition diagram.

TABLE 4.1

TOKEN OPTION LISTS FOR EACH STATE WITHIN S (FIGURE 4.3)

State in S	Option List
S1	"a", "b"
S2	"c"
S3	"d"

transition diagram system, its implementation in Tindallian, and the corresponding "general option" lists for each state are given in Figures 4.4 and 4.5, and Table 4.2, respectively. Notice that the options for a diagram's exit states are determined by the state from which the current diagram was invoked; thus, the fact that the diagram R may only be invoked at P1 in P determines that its option list is equivalent to that of P6 in P. On the other hand, Q may be invoked from both states P1 and P3 in diagram P. Thus, the "general option list" for state Q2 depends on its history, the state from which Q was invoked during the parse.

The option list for a particular state is not only affected by the configuration of its diagram's invoking state--the list of acceptable tokens as defined by Tindallian code may also be affected by any information relevant to the complete history of parser actions taken. In other words, although a given state within a transition diagram may have a non-empty list of options (or arcs leading from it), during the actual parse it is often the case that some (or all) of the options

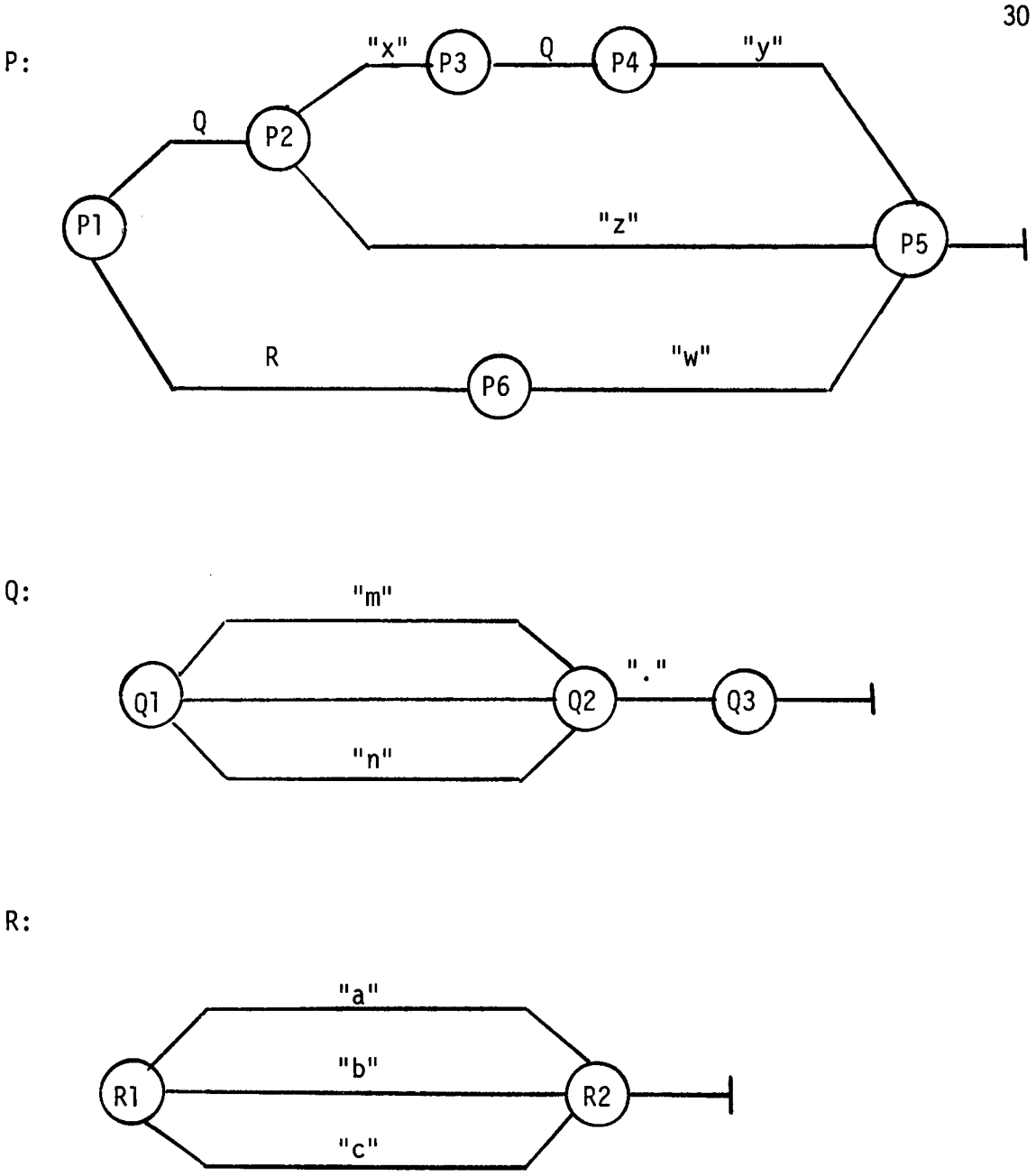


Figure 4.4. A system of transition diagrams.

```
            SCAN
            BRANCH      TOKEN = "m", CALLQ
            BRANCH      TOKEN = "n", CALLQ
            CALL  R
            BRANCH      TOKEN ≠ "w", ERROR
DONE        FINAL PARSE STATE
            BRANCH      ERROR   *IF MORE INPUT, INVALID

CALL Q
            BRANCH      TOKEN = "z", DONE
            BRANCH      TOKEN ≠ "x", ERROR
            SCAN
            CALL Q
            BRANCH      TOKEN ≠ "y", ERROR
            BRANCH      DONE

            PROC Q

            BRANCH      TOKEN ≠ "m", IS.IT.N    SCAN
            RETURN
ISITN       BRANCH      TOKEN ≠ "n", ERROR
            SCAN
            RETURN
            END PROC

            PROC R

            BRANCH      TOKEN = "a", GOT.IT
            BRANCH      TOKEN = "b", GOT.IT
            BRANCH      TOKEN = "c", GOT.IT
            BRANCH      ERROR
GOT.IT      SCAN
            RETURN
            END PROC

            END SYNA
```

Figure 4.5. Tindallian representation of system of transition diagrams in Figure 4.4.

TABLE 4.2

OPTION LISTS FOR STATES IN SYSTEM FROM FIGURE 4.4

State	Invoking State	Option List
P1	---	<Q> < "m", "n"
		<R> < "a", "b", "c"
P2	---	"x", "z"
P3	---	<Q> < "m", "n"
P4	---	"y"
P5	---	---
P6	---	"w"
Q1	P1	"m" "n"
	P3	"m" "n"
Q2	P1	"o"
	P3	"o"
Q3	P1	"x" "z"
	P3	"y"
R1	P1	"a" "b" "c"
R2	P1	"w"

must be pruned from the list because of the parser's history. An example of a familiar instance in which this is the case is shown in Figure 4.6. Within a FORTRAN program, all specification statements must precede the first DATA statement, and all DATA statements must precede the first executable statement. Thus, in the transition diagram which parses a <statement>, the arc labeled <specification statement> is pruned from the option list of state S1 after the first DATA or executable statement has been seen, and the arc labeled <DATA statement> is pruned after the first executable statement has been seen.

The storage locations used by the language designer to record such information may be any of a number of possible variable types accessible within the Tindallian code. Thus, the error system restricts the option list to acceptable tokens by interpreting all test instructions within the Tindallian code, which may refer to not only characteristics of the input token, but also any other information within the parser's domain.

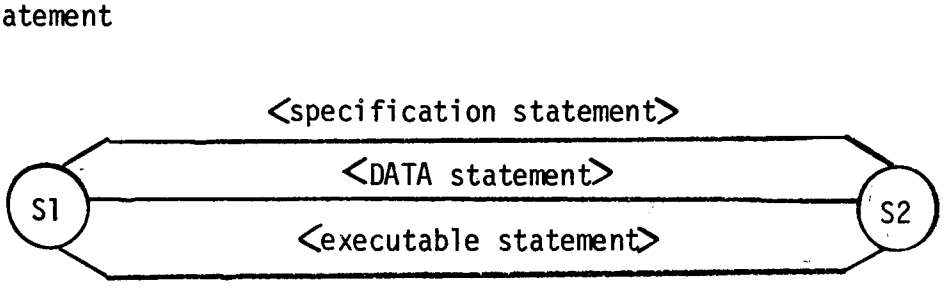


Figure 4.6. Example of Fortran diagram with prunable arcs.

The preceding discussion has described the struggles involved in building a list of parsable token options. The relationship of nonterminals to this effort has also been mentioned. What remains to be described is the way in which nonterminals are used during the actual Add, Replace, and Suppress operations.

It is possible to view the input string which has been parsed as either the string of terminal input symbols, or as a string which may represent any of the strings of terminals and nonterminals which are sentential forms in the input sentence derivation. This allows the Add, Replace, and Suppress operations to apply not only to the actual input token at any current location, but also to the nonterminal of which the input token is the initial symbol. For example, the diagram P in Figure 4.4 would find an erroneous token ("c") while parsing the string "nocw.". After the last valid character has been backed-up to, the substring beyond the input pointer has only the representation "ocw". After backing up another token, however, the expanded substring may be viewed as either "nocw" or "Qcw". Thus, when Tindall's system has backed up to the token "n", the "A", "r" and "S" operations may be tried for both the terminal symbol "n" and the nonterminal "Q". The correction involving a single deletion of "Q" yields a much more understandable suggestion than would the suggestion of the combined deletions of "n" and "o", and is made at the k = 1 level.

This concludes a discussion of Tindall's system model. Some details of its implementation will be included in Chapter 5.

CHAPTER 5

MODIFIED MODEL AND ITS IMPLEMENTATION

5.1 TICK COMPILING ENVIRONMENT

The TICK (Table-driven Interactive Compiler from Knoxville) system being developed at the University of Tennessee is a translator writing system whose design is based on CAPS. Like CAPS, it is table-driven, multilingual, and highly interactive. Its compiler uses the top-down, recursive descent parsing model as described in Chapter 2. The TICK system provides an editing/compiling environment as well as a language design facility.

There are several differences between the TICK and CAPS systems. TICK is implemented on a DECsystem-10 and interacts with the student via a standard CRT terminal; CAPS is implemented on the PLATO IV system, connected to plasma graphics terminals. The differences in resources which are available to users on the PLATO and DEC-10 systems are summarized in Table 5.1. Note that the only significant limitation on the DEC-10 system is CPU usage due to timesharing. Yet this figure is quite generous compared to the severe limitation of CPU time on PLATO, and because the CPU power of the DEC-10 is greater than that of PLATO, this difference is even more substantial. The combined restrictions placed on program size and disk accesses per hour prevented any design modifications in CAPS which would expand program size to increase efficiency.

TABLE 5.1

SUMMARY OF RESOURCES AVAILABLE ON PLATO IV AND DECSYSTEM-10*

Maximum Resource Available	PLATO IV	DEC-10
Core		
Program Size	8K ┐	48K
Data Area	3K ┘	
Disk		
File Size	8K	128K+
Accesses/hr.	5	1800+
CPU usage (msecs./sec.)	2	25

*Word size is 60-bit for PLATO, 36-bit for DEC-10.

Another restriction placed upon CAPS was its required implementation in the TUTOR language. As mentioned in Chapter 4, the interpreted compiler in CAPS is very slow. The TICK compiler (written in ALGOL with some machine-language components), once translated into DEC-10 machine code, is executed directly.

Whereas CAPS generated an intermediate text which was interpreted at run-time, TICK generates DEC-10 machine code. Although it makes no attempts at optimization of the code produced, execution of the student's program is also directed rather than interpreted. A drawback of this code generation is that no run-time error handler has yet been devised for TICK. CAPS, on the other hand, has an excellent one.

Two differences between CAPS and TICK have provided the TICK compile-time error diagnostician with a resource which was totally

unavailable to Tindall. The CAPS compiler translated the student's program character-by-character into a tokenized form. As discussed in Chapter 4, this prevented Tindall's system from referencing any of the right context of the error. In TICK, on the other hand, the student's program is compiled line-by-line, stored internally in its original form, and is immediately available to the error system.

The result of this available information on the diagnostic capabilities of TICK will be discussed in the next section; however, the line-by-line compilation allowed a significant reduction in the size of the parser history trace in the following way: changes to the parser's environment were traced only once per line (rather than tracing all changes made for each token, as in CAPS); for decompilation, an entire line was reverse-compiled at once by simply undoing the changes for that line which were recorded in the parser history; analysis by the error system required only a special "trace per token rather than per line" mode to be indicated to the parser during a re-compilation of the line containing the current input location. This modification of the tracing method decreased the amount of trace information placed on the stack per line and delayed the invokation of a compaction scheme used when the stack becomes full.

The compaction scheme used in TICK was developed and implemented in the CAPS system, and is briefly described in (Wilcox, Davis, and Tindall [Wilcox, 1976]). The general method is as follows: on the parser trace stack is a sequence of state vectors which describe the changes made to the parsing environment for each token (line); when this stack becomes full, simply compact it by eliminating every other

vector in the stack; when the student moves the program cursor to the left (up one line) in the input, return the parser to the state associated with the last token (line) whose state vector has been saved; then, re-compile the program to the desired input location, tracing normally. This scheme results in a dense population of tokens (lines) whose state information is near the top of the stack, and a sparse population at the bottom (after repeated compaction has taken place). Wilcox et al. reported that "the density of tokens for which state information is saved is proportional to 1/log n, where n is the distance (in number of tokens) from the cursor" [Wilcox, 1976].

5.2 MODIFICATION OF TINDALL'S MODEL

The major modification which the author has made to the model describing Tindall's system is the addition of right context analysis. A description of its effects on diagnostic accuracy will constitute the greater portion of this section. Another change is a slight reordering of the modifications tried for any given input location: Tindall lists the order in which the three operations are tried as Add, Replace, and then Suppress. The order Suppress, Add, and Replace, on the other hand, would prevent unnecessary analysis of the syntax tables for those cases where an "extra" token occurs (due to either a typing error or a mis-matching of parentheses), and thus is worth re-ordering the algorithm.

The modified versions of Tindall's algorithms are given as Appendix B. For ease of comparison, the same algorithmic notation is used and the lines in the author's version which are modifications of

the original lines contain a "+" at the left hand margin. The remainder of this section will discuss the results gained from addition of the right context to Tindall's model.

The elimination of right context analysis was imposed on Tindall's error system model by his implementation environment. The author, in extending Tindall's model, felt that the analysis of some portion of the error's right context would significantly improve the accuracy of the error diagnostics produced. Since the student had finished typing (or editing) a complete line of his program and had submitted it for compilation, it seems reasonable that the entire contents of the line should be useful to the error system in determining the student's original intention, in the event that an error is detected.

The decision to limit this analysis to the correct line was purely intuitive. At times, useful information may be available past the error, but occurs only upon following lines. In these cases, the limitation reduces the system's suggestion message accuracy to that of Tindall's. Many situations exist, however, where the small amount of text available to the right of the erroneous token prevent the extended system from making a somewhat confusing suggestion.

Some examples of error messages which would be produced by Tindall which are deficient are shown in Figure 5.1. In (a), the message shown is the only message produced. Since a Suppression of the error token trivially results in a "correct" interpretation in Tindall's system, he prunes that suggestion from his model (see algorithm MODIFY in Appendix A). Thus, the only valid correction ("Remove this

(a) GO TO 10 [IF]

This reserved word is not permitted here.

(b) IF (A [B]) 10, 20, 30

Insert a relational operator in front of [ ].

(c) If (A .EQ. B) [TO], 20, 30

Replace [ ] with an executable statement.

(d) IF (A. EQ. B) GO TO 10[,] 20, 30

This punctuation symbol is not permitted here.

Figure 5.1. Examples of confusing error messages produced by CAPS.

reserved word from the line.") is never suggested.

A confusing series of messages which could be produced by Tindall appear as (b), (c), and (d). The corrections suggested are not only inconsistent with the rest of the line; the student who follows the suggestions of (b) and (c) receives no diagnostic aid at all when his statement transforms into the one shown in (d). The user, in this case, has followed the suggestions produced by Tindall's system to a "dead end." In the TICK implementation, these suggestions will not be made because their corresponding modifications do not yield a correct interpretation of the complete line.

For the reader concerned that the student had, for example, intended the FORTRAN statement in (b) to be a Logical IF, the ability for the system to make up to k changes must be remembered! Eventually, the system will reconstruct the right hand portion of the line into a correct form of the Logical IF and suggest all the changes necessary for that correction. An example of this message is shown in Figure 5.2. Note that Tindall's system would not make this precise suggestion; his only approximation to the complete revision necessary is the message given in Figure 5.1(b).

Figure 5.3 gives a comparison of all suggestion messages requiring one change which will be produced by the TICK and CAPS systems for the sample input string given in Figure 5.1(b).

```
IF (A B) 10, 20, 30

    Insert a relational operator in front of "B" and
      Replace the label list with an executable statement.
```

Figure 5.2. Example of TICK suggestion message which reconstructs the invalid FORTRAN line shown in Figure 5.1(b).

IF (A B) 10, 20, 30

Produced by CAPS	Produced by TICK
	"B" could be removed from this line.
Insert a relational operator in front of "B".	
Insert a binary operator in front of "B".	Insert a binary operator in front of "B".
Replace "B" with ")".	
Remove "A" from this line.	Remove "A" from this line.
Replace "A" with "(".	
Replace "A" with a unary operator.	Replace "A" with a unary operator.

Figure 5.3. Comparison of CAPS and TICK error messages.

5.3 CHANGES IN IMPLEMENTATION

The modifications to Tindall's model as described in Section 5.2 greatly enhance its diagnostic capability. Some important differences in implementation between the TICK and CAPS systems have also been made, and will now be discussed.

The algorithms given in Appendices A and B imply that the complete "general-option-tree" for a given location is constructed before modification begins, but this is not the case for either system. The algorithm is stated in this form so that the actions performed which relate directly to the error correction method are not complicated by the somewhat confusing tree-generation procedure which is actually taking place at the same time.

As mentioned previously, the error system generates each interpretation only if requested by the student. Both TICK and CAPS use the following basic scheme: the system analyzes the syntax tables to determine a possibly valid "option"; the "A" and "R" operations are then performed for the option, and if either results in a correction, it is suggested; if the option did not yield a correct interpretation or if the user requests another suggestion, the analysis proceeds to search for a different option; when all options for the current location have been tried, the error system backs up one input location and starts the procedure again (In TICK, Suppression is tried immediately after backing up; in CAPS, immediately before). In practice, the error system actually starts searching for the next possible correction as soon as a suggestion is made. This allows an overlap of CPU time with the student's contemplation of the message and improves overall response time.

There are major differences in the analysis of the parser tables during the option search which give the TICK language designer more freedom in writing the syntax description. The CAPS system generates the option tree by actually interpreting the syntax tables, instruction by instruction. Whenever a reference is made to system information about the current input token, Tindall records any information which may be gained from that reference. For example, the statement

```
BRANCH TOKEN=="a", ERROR
```

tells the system exactly what it was looking for--that "a" is a valid input symbol (an option) at the current input location. Likewise, the statements

```
BRANCH CLASS=VARIABLE_NAME, ERROR
BRANCH SUBCLASS=REAL_TYPE, ERROR
SCAN
```

indicate that one valid option may be described by the phrase "variable name(real)".

To search for a different option, Tindall backs up through the interpreted code until he finds an instruction similar to

```
BRANCH TOKEN="b",TRY_C
```

When the system initially interpreted the instruction, it assumed that the token option was a "b". Other options, however, may be defined in the Tindallian code beginning at TRY_C. For example,

```
          BRANCH TOKEN≠"b",TRY_C
             *
             *
             *
TRY_C     BRANCH TOKEN≠"c",TRY_D
          SCAN
             *
             *
             *
TRY_D     BRANCH TOKEN≠"d",ERROR
```

Both "c" and "d" are valid options.

At first glance, this method seems to be the best solution to the problem of testing information referenced by the language designer during the parse of a given input token. The language designers of CAPS, however, were restricted by this method in the following way: whenever Tindall found a reference to system information which did not give him "exact" information (such as a BRANCH CLASS>6,ERROR ,) he "gave up" [Tindall, 1977]. That is, he stopped his forward interpretation of the code and backed up to find another option. This restricted the language designer's ability to write his syntax code more efficiently. Another instance in which Tindall "gave up" was in the case that the interpretation of the code forced a reference to an "illegal" value. The "illegal" values in Tindall's method were markers he left in the parser's domain to record what information had been "spoiled" during his interpretation. For example, when Tindall interpreted the instruction

```
ASSIGN      LOCAL_VAR:=CLASS        *set
```

local_var equal to CLASS

he marked the location defined by LOCAL_VAR as an "illegal" value if his system had not yet seen an instruction which defined the value of CLASS for the current token. This was done to prevent his interpretation from later making invalid tests on the value of LOCAL_VAR. This restriction also affected the language designer's coding.

TICK's implementation of the tree-generation/analysis scheme is based on the following interpretation of the syntax tables: view each BRANCH instruction as a node of a binary tree structure, where transfer of control to the label field of the instruction begins a traversal of the right subtree of the node, and execution of the instruction immediately following the BRANCH (and thus, the condition of the BRANCH begins "false") being a traversal of the left subtree (for a description of binary trees and methods of traversal, see Knuth [Knuth, 1968, pp. 305-328]); the binary tree whose root is the first BRANCH instruction following a given SCAN then fully describes the option tree for the parser state which issues the SCAN (arrival at a SCAN instruction is equivalent to the transition of an arc from one state to the next within the transition diagram--the input token has been parsed).

An unconditional BRANCH to the error system is translated into a "dead end" (all possible arcs from the current state have already been compared to the input token). A conditional BRANCH to the error system is treated as a node with a null right link, and an unconditional branch to a label is treated as a node with a null left link.

The CALL instructions must also be incorporated into the tree structure. To do this, a CALL instruction with a single return is viewed as a node with a null left link; a CALL with multiple returns is viewed as a non-binary tree which may be represented as a binary tree (see [Knuth, 1968, pp. 332-345]).

The determination of a valid option incorporates a three-pass approach. The first pass involves a pre-order traversal of the binary tree structure described. This traversal continues until a "possible option" is found. The recognition of a "possible option" is simple--if a SCAN is found during the traversal, then the sum of information which has been gleaned from the nodes of the current "limb" of the tree is assumed to define the characteristics of the "possible option" (this information should be roughly equivalent to the information represented by the label on one arc leaving the "current state"). If a CALL is found, the sum of the information gleaned serves to partially describe the first terminal symbol of the non-terminal represented by the procedure CALLed.

The second pass uses the compiler's parser to test the actual validity of the option at the input location. The parser interprets the code previously traversed by the error system using a dummy entry in the symbol table which includes all information about the option to insure that no "pruning" information has been overlooked. For a nonterminal, this pass involves only the testing of information from the SCAN down to the CALL--if this sequence of code executes properly, it is assumed that the nonterminal may occur in the given input location.

The final pass of this approach is the REPARSE procedure given in Appendix B. It consists of a trial parsing of the rest of the input

string (through the current line). It should be noted that it is possible for an error situation which arises during this phase to result in another modification to the input (for up to k changes).

Sample dialogues which might take place between TICK and a student during error diagnosis (for $k = 1$) are given as Figure 5.4 and 5.5. This concludes a discussion of the TICK error system implementation.

```
            INTEGER I,J,K
            I = J K

This variable name is not permitted here.
****ERROR SYSTEM****To Edit, press 'E'; for Help, press 'H'.        ?

            INTEGER I,J,K
            I = J K

This variable name could be removed from this line.
****ERROR SYSTEM****To Edit, press 'E'; for Help, press 'H'.        ?

            INTEGER I,J,K
            I = J K

A binary operator could be inserted in front of 'K'.
****ERROR SYSTEM****To Edit, press 'E'; for Help, press 'H'.        ?

            INTEGER I,J,K
            I = J K

This variable name could be removed from this line.
****ERROR SYSTEM****To Edit, press 'E'; for Help, press 'H'.        ?

            INTEGER I,J,K
            I = J K

This variable name could be replaced by a unary operator.
****ERROR SYSTEM****To Edit, press 'E'; for Help, press 'H'.        ?

            INTEGER I,J,K
            I = J K

Sorry, no more help is available from error system.  Press 'E' to edit?
```

Figure 5.4. Sample of error system dialogue using FORTRAN example.

```
          INTEGER I
          I = 5
          REAL X

This reserved word is not permitted here.
****ERROR SYSTEM****To Edit, press 'E'; for Help, press 'H'.        ?

          INTEGER I
          I = 5
          REAL X

This assignment statement could be replaced by a type statement.
****ERROR SYSTEM****To Edit, press 'E'; for Help, press 'H'.        ?
    To see what a type statement looks like, press 'D'.

          INTEGER I
          I = 5
          REAL X

This assignment statement could be replaced by a type statement.
****ERROR SYSTEM****To Edit, press 'E'; for Help, press 'H'.        ?
    A type statement may begin with 'INTEGER'.
    To see another example of a type statement, press 'D'.
```

Figure 5.5. Example of detailed dialogue using FORTRAN example.

CHAPTER 6

CONCLUSION

6.1 EVALUATION OF TICK ERROR SYSTEM

The modifications made by Tindall to Levy's algorithm and its adaptation to an interactive system have led to the development of a powerful diagnostic tool. The error messages provided by TICK, due to the analysis of right context information, in general provide a more accurate diagnosis than Tindall's, and give very useful aid to beginning programmers. As a result of this diagnostic power, the TICK system itself is one of the most impressive translator writing systems which have been developed.

The TICK error system is able to restructure an invalid sequence of terminal symbols into a correct interpretation, and to describe the modifications necessary in a language-dependent vocabulary without much additional effort by the language designer. This facility allows a language designer to generate an interactive compiler with excellent error diagnostics with a minimum amount of effort.

6.2 FUTURE RESEARCH IN ERROR DIAGNOSTIC SYSTEMS

The dialogue which takes place between the TICK error system and its student user shows the potential of an automatic system in an interactive environment. In retrospect, however, the error dialogue produced by TICK is deficient in two respects: (1) it can be used by the lazy trial-and-error programmer as a crutch rather than a tool;

and (2) even the conscientious student trying to understand his error is required to do a complex mental analysis of the differences between the corrected form of the input and his original input.

A series of suggestions describing all the possible ways to correct the error situation does not seem to be the most pedagogically sound form that such a dialogue could take. The beginning student's error (as compared to that of an experienced programmer) is more likely to have been made due to a misunderstanding of syntactic form rather than carelessness. Since the student is in the process of learning proper syntax, perhaps a dialogue which would encourage a more active participation by the student during the analysis of the problem is needed.

A better dialogue for encouraging the student to think about the cause of his error (the syntactic form that he has misunderstood) might do the following: (1) initially determine the smallest substring containing the error which can be replaced by a unique nonterminal; (2) give the student a description of its proper syntactic form (perhaps in BNF notation, but in language-dependent terms) and indicate that he is to try to type a corrected version of the substring; (3) back up to the beginning of the substring and prompt the user for each syntactic element of the nonterminal, giving him a list of all options which could be taken at each point (this is similar to the program preparation system developed by Pinc and Schweppe [Pinc, 1973]); (4) make suggestions at each point in the trial correction, based on an analysis of the invalid substring.

Admittedly, this system could become quite tedious at times, especially for experienced programmers. However, the increase in student participation and analysis might substantially improve the value of the system in helping the beginning student to both learn syntactic forms and become less careless.

Another area of possible research would be a study of exactly what information is necessary to produce the best diagnostic assistance. That is, given the "best" error message which can be produced, what information is necessary for an automatic system to generate it? The need also exists for determining the "best" form of meaningful diagnostics for a beginning student.

To the author's knowledge, although a few studies have been made to determine the most common errors of specific languages, none have studied what might be called the "average distance" between a detected error and its cause. A method which might be used to do this is for a sample of error situations to be gathered from a population of beginning student programs, analyzed by a human "judge" who would determine the "best" correction for each error, and the distance computed between the error detection point and the first token modified by the correction.

This information could apply directly to the TICK implementation by allowing k modifications to be made to the invalid input string, where k is a function of (a) the distance between the current backed-up-to location and the error location, and (b) those variables found to be important in the study of "distance" (e.g., the experience of the user and the number of errors detected so far).

6.3 CONCLUSIONS

The area of automatic error diagnostics and correction is relatively new. The application of existing techniques to interactive systems and the development of new techniques specifically for these systems are fields in which many aspects of error analysis may be explored. It is hoped that this work will inspire further research in these areas.

LIST OF REFERENCES

LIST OF REFERENCES

Alpert, D. and Bitzer, D., "Advances in Computer Based Education," Science, Vol. 167 (March 20, 1970), pp. 1582-1590.

Aho, A. V., and Ullman, J. D., The Theory of Parsing, Translating, and Compiling, Prentice-Hall, Englewood, N. J., 1972.

Conway, M., "Design of a Separable Transition-Diagram Compiler," CACM, Vol. 6, No. 7 (July 1963), pp. 396-408.

Davis, A., An Interactive Analysis System For Execution-time Errors, Ph.D. thesis, Department of Computer Science, University of Illinois, 1975, Technical Report # UIUCDCS-R-695.

DeRemer, F. L., "Simple LR(K) Grammars," CACM, Vol. 14, No. 7 (July 1971), pp. 453-460.

Floyd, R. W., "Syntactic Analysis and Operator Precedence," JACM, Vol. 10 (July 1963), pp. 316-333.

Graham, S., and Rhodes, S., "Practical Syntactic Error Recovery," CACM, Vol. 18, No. 11 (November 1975), pp. 639-650.

Gries, D., Compiler Construction for Digital Computers, John Wiley, New York, 1971.

Irons, E. T., "An Error-Correcting Parse Algorithm," CACM, Vol. 6, No. 11 (November 1963), pp. 669-673.

James, L., A Syntax Directed Error Recovery Method, Masters thesis, Computer Systems Research Group, University of Toronto, May 1972, Technical Report # CSRG-13.

James, E., and Partridge, D., "Adaptive Correction of Program Statements," CACM, Vol. 16, No. 1 (January 1973), pp. 27-37.

Knuth, D., The Art of Computer Programming, Vol. 1, Addison-Wesley Publishing Company, Inc., Reading, Massachusetts, 1968.

La France, J., Syntax Directed Error Recovery for Compilers, PhD. thesis, Department of Computer Science, University of Illinois, 1971, Technical Report # 459.

Leinius, R., Error Detection and Recovery for Syntax-Directed Compiler Systems, Ph.D. thesis, Department of Computer Science, University of Wisconsin, 1970.

Levy, J., Automatic Correction of Syntax Errors in Programming Language, Ph.D thesis, Computer Science Department, Cornell University, December 1971, Technical Report # TR 71-116.

Litecky, C., and Davis, G., "A Study of Errors, Error Proneness, and Error Diagnosis in Cobol," CACM, Vol. 19, No. 1 (January 1976), pp. 33-37.

Lomet, D. B., "A Formalization of Transition Diagram Systems," JACM, Vol. 20, No. 2 (April 1973), pp. 235-257.

Lyon, G., "Syntax directed least-errors analysis for context free languages: a practical approach," CACM, Vol. 17, No. 1 (January 1974), pp. 3-14.

Nievergelt, J., et al, ACSES: The Automated Computer Science Education System at the University of Illinois, Department of Computer Science, University of Illinois, 1976.

Pinc, J. and Schweppe, E., "A Fortran Language Anticipation and Prompting System," Proceedings ACM73 Annual Conference, Association for Computing Machinery, N.Y., 1973.

Tindall, M., An Interactive Table Driven Parser System, Masters thesis, Department of Computer Science, University of Illinois, 1975, Technical Report # UIUCDCS-R-75-748.

Tindall, M., An Interactive Compile-time Diagnostic System, Ph.D. thesis, Department of Computer Science, University of Illinois, Urbana-Champaign, Illinois, 1975.

Tindall, M., Personal Communication, February 10, 1977.

Widler, T., A Table-Driven Parser for the TICK System, Masters thesis, Department of Computer Science, University of Tennessee, Knoxville, 1977.

Wilcox, T., Davis, A., and Tindall, M., "The Design and Implementation of a Table Driven, Interactive Diagnostic Programming System, CACM, Vol. 19, No. 11 (November 1976), pp. 609-616.

Wirth, N., and Weber, H., "EULER-A Generalization of ALGOL and its Formal Definition," CACM, Vol. 9, No. 1 (January 1966), pp. 13-25, and Vol. 9, No. 2 (February 1966), pp. 89-99.

APPENDICES

APPENDIX A

TINDALL'S ALGORITHMS AND NOTATION

Algorithmic Notation

The notation used in the following algorithms will be an Algol-like language (as used in [Tindall, 1975b] containing the following constructs: the left arrow "←" is the assignment operator; an "if - then - else" statement is terminated by a "fi"; a looping construct "loop . . . while . . . repeat"; and a "begin - end" construct.

In certain instances, explicit definitions are not given for local operations. These are denoted by single slashes "/" around a description of the local operations performed. References to other procedures within the error system are denoted by double slashes "//" which are placed around a description of what operations are performed by the procedure. All comments begin with the word comment, and end with a period ".".

```
-   begin USER_INTERACTION (request)
    comment request is an output parameter.  It is set to either
         'successful-provide detail' or 'successful-restrict detail,'
         depending upon the programmer's keypress response.
    comment USER_INTERACTION displays a new suggestion message on the
         screen and then waits for the programmer to respond.  If the
         programmer requests another suggestion message, then "request"
         is set and the routine returns normally.  If the programmer
         responds with any other compiler editing key, then this
         routine forces the termination of the error analysis process,
         and exits directly into the compiler's editor system.
    comment Display new message.
-   //erase : current message on screen//
-   //display : new message on screen//
    comment "Read-edit-key" waits until an editing key or one of the
         "help request" keys is pressed; the key-type is returned in
         "Read-key."
-   /Read-edit-key/
-   if Read-key = Compiler-editing-key then
         /reestablish program editing environment/
         /exit from error system to the compiler's editor/
-   fi
    comment The programmer has requested another suggestion.  The normal
         request is the 'HELP' key, which requests that more detail be
         provided (if possible).  The alternate request is to restrict
         detail in the next message.
```

```
-   request ← if Read-key = 'HELP' then 'successful-provide detail'
        else 'successful-restrict detail' fi
-   end USER_INTERACTION

-   begin REPARSE (LR, QR, KR, user_request)
    comment LR is reparse start input location index,
            QR is reparse start state for location LR,
            KR is number of modifications made to input string,
            user_request is an output parameter that specifies
                whether the reparsing is successful or not.
-   /user_request ← 'unsuccessful'/
    comment error_token_location is input location of the originally
        detected error token.
    comment Is reparse attempt successful?
-   if LR > error_token_location then
-       if KR = N then
            comment Reparsing was successful.  Invoke routine
                    USER_INTERACTION, which will either return
                    with user_request set indicating the amount
                    of detail the programmer wants, or else
                    will cause the error system process to be
                    terminated without returning normally.
-           //USER_INTERACTION (user_request)//
-       fi
-   else
```

```
        comment Extend reparse attempt 1 token if next token
                is acceptable.
-       if transit (token (LR), QR) ≠ "null" then
-          //REPARSE (LR + 1, transit (token (LR), QR), KR,
                                        user_request)//
-       fi
        comment Make another modification at location LR, if possible,
                to build all legal interpretations.
-       if KR < N then
-          //MODIFY (LR, QR, KR)//
-       fi
-   fi
-   end REPARSE


-   begin MODIFY (L, S, K)
    comment L is the input location pointer (indexed from the
              beginning of the input string),
            Q is the parser state number for token at location L,
            K is the number of modifications that have been made
              prior to current activation.
    comment OPT routine constructs the general_option_tree.
        It is a tree structure, with descendents of a node repre-
        senting further refinements or detail for the node.
-   //OPT:  build general_option_tree for state Q with root 'root'//
```

```
-   /Lnt ← location in input following the nonterminal that
        begins with token at location L; if no nonterminal
        begins there, set Lnt ← 0/
    comment  Try Add, Replace operations for sons of root of tree.
-   //MOD_TREE (L, Q, K, Lnt, root)//
    comment  Prune special case for Suppress operation.
-   if L ≠ error_token_location then
        comment  Token Suppress operation.
-       //REPARSE (L + 1, W, K + 1, last_try)//
        comment Does nonterminal exist?
-       if Lnt ≠ 0 then
            comment  Nonterminal Suppress operation.
-           //REPARSE (Lnt, Q, K + 1, last_try)//
-       fi
-   fi
-   end MODIFY


-   begin  MOD_TREE (LD, QD, KD, LntD, Droot)
    comment  LD is the input location pointer,
             QD is the parser state number for token at location
                LD,
             KD is number of previous modifications.
             LntD is location in input following nonterminal or
                  0 if no nonterminal exists,
             Droot is the root of the (sub) tree in general_
                  option_tree.
```

```
    comment  MOD_TREE tries Add and Replace  (token and nonterminal)
             operations for each son (option) of Droot.
    comment  REPARSE routine's 4th parameter is set to
             1)  'unsuccessful' : if previous modification did
                      not completely "correct" input string, or
             2)  'successful-provide detail' or
                 'successful-restrict detail' : if previous
                      modification "corrected" input string, but
                      programmer requested another suggestion.
-   for all g ε sons (Droot) loop :
        comment  Add operation.
-       //REPARSE (LD, transit (g, QD), KD + 1, last_try)//
        comment  Is more detail requested?
-       if  last_try = 'successful-provide detail' then
            comment  Give more detail:  move depth-first through
                     tree.
-           //MOD_TREE (LD, QD, KD, LntD, g)//
-       fi
        comment Prune special case for Replace operation.
-       if (LD ≠ error_token_location or last-try = 'unsuccessful')
            and last_try ≠ 'successful-restrict detail'
            then
            comment  Token Replace operation.
-           //REPARSE (LD + 1, transit (g, QD), KD + 1,
                last_try)//
            comment  is more detail requested?
```

```
-           if last_try = 'successful-provide detail' then
                comment  Give more detail:  move depth-first
                         through tree.
-               //MOD_TREE (LD, QD, KD, LntD, g)//
-           fi
            comment  Does nonterminal exist?
-           if LntD ≠ 0 and last_try ≠ 'successful-restrict
              detail'
                then
                comment  Nonterminal Replace operation.
-               //REPARSE (LntD, transit (g, QD), KD + 1,
                    last_try)//
                comment  Is more detail requested?
-               if last_try = 'successful-provide detail' then
                    comment  Give more detail:  move depth-
                             first through tree.
-                   //MOD_TREE (LD, QD, KD, LntD, g)//
-               fi
-           fi
-       fi
-   repeat
-   end MOD_TREE
```

APPENDIX B

BUTTON'S ALGORITHMS

```
-   begin USER_INTERACTION (request)
    comment request is an output parameter.  It is set to either
        'successful-provide detail' or 'successful-restrict
        detail,' depending upon the programmer's keypress
        response.
    comment USER_INTERACTION displays a new suggestion message on
        the screen and then waits for the programmer to respond.
        If the programmer requests another suggestion message,
        then "request" is set and the routine returns normally.
        If the programmer responds with any other compiler edit-
        ing key, then this routine forces the termination of the
        error analysis process, and exits directly into the com-
        piler's editor system.
    comment Display new message.
-   //erase : current message on screen//
-   //display : new message on screen//
    comment "Read-edit-key" waits until an editing key or one of
        the "help request" keys is pressed; the key-type is
        returned in "Read-key."
-   /Read-edit-key/
-   if  Read-key = Compiler-editing-key then
-       /reestablish program editing environment/
        /exit from error system to the compiler's editor/
-   fi
```

```
-   comment  The programmer has requested another suggestion.
         The normal request is the 'HELP' key, which requests that
         more detail be provided (if possible).  The alternate
         request is to restrict detail in the next message.
-   request ← if Read-key = 'HELP' then 'successful-provide
         detail'
-        else 'successful-restrict detail'  fi
-   end  USER_INTERACTION

-   begin  REPARSE (LR, QR, KR, user_request)
    comment  LR is reparse start input location index,
             QR is reparse start state for location LR,
             KR is number of modifications made to input string,
             user_request is an output parameter that specifies
                  whether the reparsing is successful or not.
-   /user_request ← 'unsuccessful'/
    comment  error_token_location is input location of the
         originally detected error token.
    comment  Is reparse attempt successful?
-   if LR > error_token_location then
-        if  KR = N  then
                comment  Reparsing was successful.  Invoke routine
                         USER_INTERACTION, which will either return
                         with user_request set indicating the amount
                         of detail the programmer wants, or else
                         will cause the error system process to be
                         terminated without returning normally.
```

```
-           //USER_INTERACTION (user_request)//
-       fi
-   else
        comment  Extend reparse attempt 1 token if next token
                 is acceptable.
-       if  transit (token (LR), QR) ≠ "null" then
-           //REPARSE (LR + 1, transit (token (LR), QR), KR,
-                                        user_request)//
-       fi
        comment  Make another modification at location LR, if
                 possible, to build all legal interpretations.
-       if  KR < N then
-           //MODIFY (LR, QR, KR)//
-       fi
-   fi
-   end  REPARSE


-   begin  MODIFY (L, S, K)
    comment  L is the input location pointer (indexed from the
               beginning of the input string),
             Q is the parser state number for token at location L,
             K is the number of modifications that have been made
               prior to current activation.
    comment  OPT routine constructs the general_option_tree.
         It is a tree structure, with descendents of a node repre-
         senting further refinements or detail for the node.
```

```
-    //OPT:  build general_option_tree for state Q with root
          'root'//
-    /Lnt ← location in input following the nonterminal that
          begins with token at location L; if no nonterminal
          begins there, set Lnt ← 0/
+
+
+
+
          comment  Token Suppress operation.
-         //REPARSE (L + 1, Q, K + 1, last_try)//
          comment  Does nonterminal exist?
-         if  Lnt ≠ 0  then
               comment  Nonterminal Suppress operation.
               //REPARSE (Lnt, Q, K + 1, last_try)//
+         comment  Try Add, Replace operations for
                        sons of root of tree.
+         //MOD_TREE (L, Q, K, Lnt, root)//
-         fi
-    fi
-    end  MODIFY


-    begin  MOD_TREE (LD, QD, KD, LntD, Droot)
     comment  LD is the input location pointer,
              QD is the parser state number for token at location
                 LD,
```

```
            KD is number of previous modifications,
            LntD is location of input following nonterminal or
                 0 if no nonterminal exists,
            Droot is the root of the (sub) tree in general_
                 option_tree.
      comment  MOD_TREE tries Add and Replace (token and nonterminal)
            operations for each son (option) of Droot.
      comment  REPARSE routine's 4th parameter is set to
            1)  'unsuccessful' : if previous modification did
                      not completely "correct" input string, or
            2)  'successful-provide detail' or
                'successful-restrict detail' : if previous
                      modification "corrected" input string, but
                      programmer requested another suggestion.
-     for all g ε sons (Droot) loop :
            comment  Add operation.
-           //REPARSE (LD, transit (g, QD), KD + 1, last_try)//
            comment  Is more detail requested?
-           if last_try = 'successful-provide detail'  then
               comment  Give more detail:  move depth-first through
                        tree.
-              //MOD_TREE (LD, QD, KD, LntD, g)//
-           fi
+
+
+
+
```

```
                comment  Token Replace operation.
-               //REPARSE (LD + 1, transit (g, QD), KD + 1,
                    last_try)//
                comment  is more detail requested?
-               if last_try = 'successful-provide detail' then
                    comment  Give more detail:  move depth-first
                             through tree.
-                   //MOD_TREE (LD, QD, KD, LntD, g)//
-               fi
                comment  Does nonterminal exist?
-               if  LntD ≠ 0 and last_try ≠ 'successful-restrict
                    detail'
                    then
                    comment  Nonterminal Replace operation.
-                   //REPARSE (LntD, transit (g, QD), KD + 1,
                        last_try)//
                    comment  Is more detail requested?
-                   if last_try = 'successful-provide detail' then
                        comment  Give more detail:  move depth-
                                 first through tree.
-                       //MOD_TREE (LD, QD, KD, LntD, g)//
-                   fi
-               fi
-   fi
-   repeat
-   end MOD_TREE
```

VITA

Beverly Karen Button was born on December 28, 1954. Raised in Chattanooga, Tennessee, she graduated from Chattanooga Central High School in June, 1972. She entered the University of Tennessee as a freshman in September of 1972, and received her Bachelor's degree in Psychology in August of 1975. In September, she began graduate studies in Computer Science, and will receive her Master's degree in December of 1977.

She was president of the Student Chapter of the Association for Computing Machinery in 1976-1977, and is a member of Phi Kappa Phi.

After graduation, she plans to join the Technical Staff at Bell Laboratories in Holmdel, New Jersey.