To the Graduate Council:

I am submitting herewith a thesis written by Jane Goldberger Widler entitled "A Table-Driven Parser for the TICK System." 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 M. Davis, Major Professor

We have read this thesis
and recommend its acceptance:

Dan Smyth

Michael G Thomason

Accepted for the Council:

L Evans Peck

Vice Chancellor
Graduate Studies and Research

Thesis
77
.W534
cop.2

# A TABLE-DRIVEN PARSER FOR THE TICK SYSTEM

A Thesis

Presented for the

Master of Science

Degree

The University of Tennessee, Knoxville

Jane Goldberger Widler

August 1977

1325620

## DEDICATION

Dedicated to Wid for all he has done and given toward completion of this work.

## ACKNOWLEDGMENTS

The author wishes to express her sincere appreciation and thanks to Professor Pinkert for all the time and effort he gave so unselfishly. Without his aid and encouragement, this project might not have been completed. Special thanks also go to Ms. Beverly Button, Mr. Jim Moore, and other members of the TICK development group who gave aid and assistance to this author during the past year. Additionally, the author would like to thank Professors Davis, Straight, and Thomason for all their time and service on the committee.

I would also like to express my deepest appreciation to my husband, Wid, for putting up with all the frustrations and problems related to the completion of this project. His patience and constant faith in me are my most valuable assets.

Also, I would like to thank my parents and family for their strong support and willingness to help me in any way they can.

## ABSTRACT

The purpose of this study was to develop a table-driven parser for the TICK system which would facilitate language implementation as well as providing excellent student-oriented features. The problems considered fell in two general areas: (1) the difficulties and inconveniences encountered by language implementors as they design and develop new language compilers, and (2) the limitations and frustrations experienced by students due to the unavailability of an interactive compiler.

As a solution, the transition diagram method of parsing was decided upon. Specifically, the parser used in the CAPS system from the University of Illinois provided the basic model. This approach allows the language designer to input the language specifications in a program format rather than a tabular format, as many other models require. Not only is this easier for the language implementor, but it also saves space as well. This method of language implementation also automatically provides information to the error analysis system directly from the language specifications indicated above. The TICK system also enables a student who uses the system to obtain clear, readable, and understandable error messages immediately, as soon as the line which includes the error is keyed in. In addition to a description of the problem that generated the error, most of the time the system will also be able to suggest to the student possible ways to correct the error.

The TICK system is effective in performing the above mentioned functions. This system provides a relatively easy form for language

implementors to write, debug, and maintain language specifications. It also provides a truly interactive compiler (not just a batch compiler running on an on-line system), which includes an unusually useful error diagnostic system. These features are directly related to the parsing method used.

There are problems that exist at this time, though. Most of the problems are related to the speed of the TICK system. With some minor reprogramming and reworking of the TICK system, it is possible that these remaining problems can be solved making TICK an outstanding system.

TABLE OF CONTENTS


| CHAPTER | | PAGE |
|---|---|---|
| 1. | INTRODUCTION | 1 |
| 2. | SURVEY OF PARSING METHODS USED IN TABLE-DRIVEN SYSTEMS | 6 |
| | Introduction | 6 |
| | Precedence Techniques | 7 |
| | Production Language Method | 26 |
| | Transition Matrix Method | 33 |
| | LR(k) Method | 37 |
| | Conclusion | 44 |
| 3. | TRANSITION DIAGRAM METHOD OF PARSING | 46 |
| | Introduction | 46 |
| | Conway's Work | 46 |
| | The CAPS Parser | 49 |
| | Conclusion | 63 |
| 4. | THE TICK SYSTEM | 66 |
| | Introduction | 66 |
| | Goals and Features of TICK | 66 |
| | Why Tindall's Method Was Chosen | 69 |
| | The TICK Parser | 70 |
| | Conclusion | 77 |
| 5. | CONCLUSION | 86 |
| BIBLIOGRAPHY | | 89 |
| APPENDIX | | 94 |
| VITA | | 98 |

LIST OF TABLES


| TABLE | | PAGE |
|---|---|---|
| 2.1. | Operator Precedence Matrix for Example Grammar 1 | 12 |
| 2.2. | Example Parse Using the Operator Precedence Technique | 14 |
| 2.3. | Simple Precedence Matrix for Example Grammar 2 | 17 |
| 2.4. | Example Parse Using the Simple Precedence Technique | 18 |
| 2.5. | Extended Precedence Matrix for Example Grammar 1 (Find Tail of Handle) | 21 |
| 2.6. | Extended Precedence Matrix 2 for Example Grammar 1 (To Find Tail of Handle) | 23 |
| 2.7. | Example Parse Using the Extended Precedence Technique | 25 |
| 2.8. | Example Parse Using the Production Language Technique | 29 |
| 2.9. | Transition Matrix for Example Grammar 3 | 35 |
| 2.10. | Example Parse Using the Transition Matrix Technique | 36 |
| 2.11. | GO-TO Table | 39 |
| 2.12. | PARSER ACTION Table | 41 |
| 2.13. | Example Parse Using LR(k) Technique | 42 |
| 3.1. | Relational Operators Available with the BC Instruction | 55 |
| 3.2. | ASSIGN, MASKON, MASKOFF, ADDIT, and SUBIT Instructions | 56 |
| 3.3. | Meaning of Format 1 Instructions | 61 |
| 3.4. | The OPER Field, Format 2 | 62 |
| 3.5. | Use of MISC Field | 63 |
| 3.6. | Meaning of PARMOP Field (Format 2 Instructions) | 64 |
| 4.1. | Example Parse Using the Transition Diagram Technique | 82 |
| 4.2. | Example Parse of an Incorrect String | 84 |

LIST OF FIGURES


| FIGURE | | PAGE |
|---|---|---|
| 1.1 | Compilation Using a TWS | 3 |
| 1.2. | Compiler Generator Stage of a CC | 4 |
| 1.3. | Compiler Stage Using a CC | 4 |
| 2.1. | Shift-Reduce Parser | 9 |
| 2.2. | Example Grammar 1 | 10 |
| 2.3. | Example Grammar 2 | 16 |
| 2.4. | Ternary Functions Used to Find Head of Handle | 24 |
| 2.5. | FPL Program for Example Grammar 1 | 28 |
| 2.6. | Original FPL Program (Before Reduction) | 32 |
| 2.7. | Reduced FPL Program Using Ichibiah and Morse's Reduction Scheme | 32 |
| 2.8. | Augmented Operator Grammar, Example Grammar 3 | 35 |
| 2.9. | Example Grammar 1 With Each Production Rule Numbered | 40 |
| 2.10. | Inclusion Tree | 45 |
| 3.1. | Example Transition Diagram | 47 |
| 3.2. | Algorithm for Traversing the Transition Diagrams | 48 |
| 3.3. | Overall Program Organization | 52 |
| 3.4. | Form of PROC-END Blocks | 52 |
| 3.5. | Breakdown of Format 1 Word | 60 |
| 3.6. | Breakdown of Format 2 Word | 61 |
| 3.7. | Addressing Modes | 64 |
| 4.1. | Block Diagram of Levels of Interpretation in CAPS-PLATO System | 75 |

| FIGURE | | PAGE |
|---|---|---|
| 4.2. | Major Modules of the TICK System Parser . . . . . | 76 |
| 4.3. | Inside Parser's "Domain" . . . . . | 76 |
| 4.4. | Example Grammar . . . . . | 78 |
| 4.5. | Transition Diagrams Generated for Grammar in Figure 4.4 . | 78 |
| 4.6. | Tindallian Source Code for an EXPRESSION . . . . . | 79 |
| 4.7. | Tindallian Source Code for TERM . . . . . | 80 |
| 4.8. | Tindallian Source Code for FACTOR . . . . . | 81 |

# CHAPTER 1

# INTRODUCTION

This paper is concerned with a parser for the TICK system. TICK (Table-Driven Interactive Compiler from Knoxville) is a programming system which consists of an editor, compiler, run-time package, and debugger. The parser is table-driven and is based on the transition diagram approach introduced by Conway (9) and later enhanced by Tindall (40).

A compiler is a meta-program that translates a program from one language to another. A compiler accepts a program written in a high level language as input and translates it into object code. Generally, this process is broken down into a number of passes. The lexical analyzer takes characters from the input text and outputs tokens (the smallest logical group in that language). The syntax analyzer, or parser, checks these tokens to make sure the syntax of the program being compiled conforms with the language. The semantic analyzer checks to make sure the semantics of the program conform to the design of the language. The code generator performs the actual translation of the source language into the object language. Often the last pass is code optimization which transforms the code generated in the previous step into more efficient code.

Writing a compiler is generally a long and expensive process. In an attempt to simplify and clarify the process, some computer scientists in the late 1950s and early 1960s developed the idea of a compiler-

compiler (CC). A CC is a program which accepts the specifications of a language as input, either in the form of tables or a program, and outputs a compiler for that particular language. In this way each time a new compiler is desired, only the specifications for the new language must be developed. One need not recode all the modules necessary to implement a new compiler. Producing a compiler in this manner saves the compiler-writer time and effort and alleviates frustration. Although the CC itself must be built, this major project needs to be undertaken just once.

Another approach to simplifying the writing of compilers is called a Translator Writing System (TWS). This approach also requires the person writing the compiler to input the specifications of the language using a form such as tables or a program. When someone wants to compile a program, he/she calls the TWS and chooses a language. The TWS then compiles the program using the appropriate language tables (see Figures 1.1, 1.2, and 1.3). TICK is an example of a TWS.

Compiling a program is usually slower with a TWS than with a compiler produced by a CC because a CC produces one compiler for each language. A TWS must be more general--not only must a TWS contain general routines for a variety of languages, but it also has the burden of interpreting the language definition tables while compiling. However, a TWS is more efficient in terms of space than many compilers because numerous languages may share identical code. CC and TWS are two ways of making a compiler table-driven, and thus multilingual (able to compile more than one language). After the major job of writing the CC or TWS itself has been completed, with a minimum amount of work, a compiler

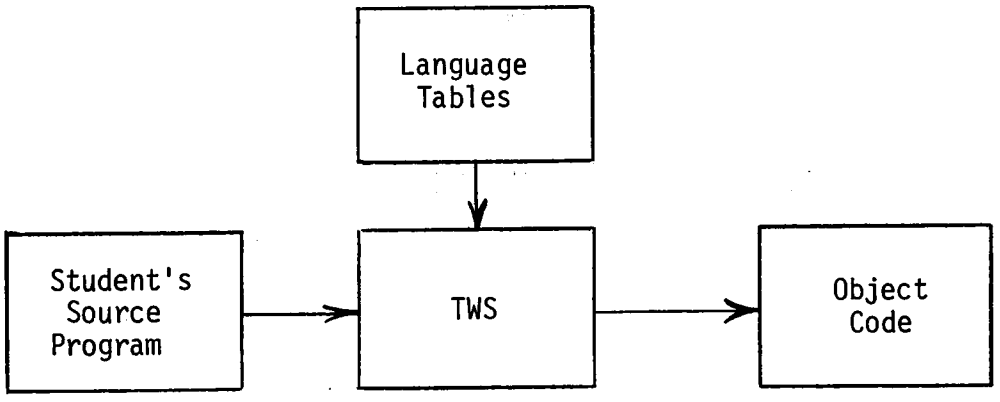


Figure 1.1. Compilation using a TWS.

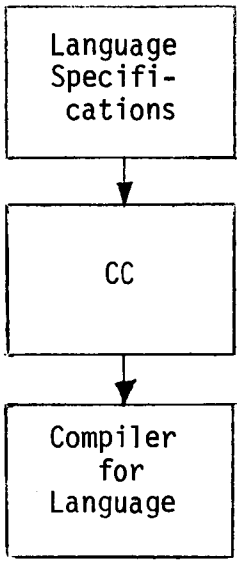


Figure 1.2. Compiler generator stage of a CC.

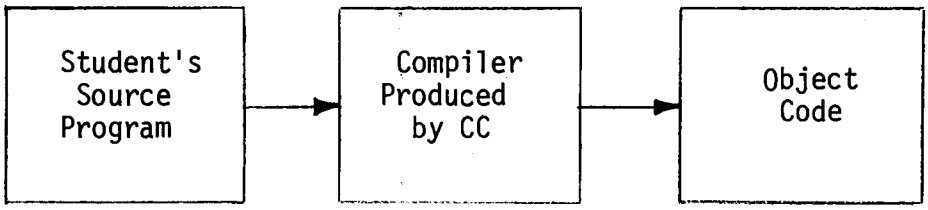


Figure 1.3. Compiler stage using a CC.

becomes available for any language which satisfies the constraints of the system.

The next chapter of this paper is a survey of parsing techniques used in table-driven systems. Most of those syntax analyzers use a variety of precedence methods which employ matrices or tables to decide when to get the next token from the lexical analyzer and when to reduce the tokens already obtained. Other methods are also discussed.

Chapter 3 discusses the transition diagram method of parsing in detail. Conway's work is introduced first. Following that is an in-depth discussion of Tindall's parser for the CAPS system. Tindall's parser is the basic model used for the parser in the TICK system.

The TICK system is discussed in Chapter 4 along with a detailed description of its parser and the extensions to Tindall's model. Chapter 5 presents the author's suggestions for further work on the TICK project.

It is the purpose of this thesis to describe the method of parsing used in the TICK system. The TICK system includes numerous features to assist programmers as well as language designers. Some of those features depend heavily upon a particular parsing method, so the parser was designed to facilitate their implementation.

# CHAPTER 2

# SURVEY OF PARSING METHODS USED IN TABLE-DRIVEN SYSTEMS

## 2.1 INTRODUCTION

This chapter contains a survey of a number of parsing techniques which have been used in compiler-compilers and translator writing systems. All parsing techniques described here use a coded version of the source language as input, either in matrix form or some form of BNF rules. A parser or a set of tables to drive a parser is the output in each case.

The first three parsing methods described are precedence methods. The initial work in this field was done by Floyd in 1963 (18). He called his system the operator precedence method. Later, in 1966, extensions and improvements were made to his original system by Wirth and Weber (43). They called theirs the precedence or simple precedence method. McKeeman (31) took another approach to improving Floyd's operator precedence method by developing the extended precedence technique. In each of the above mentioned precedence methods, there is a table which defines relationships between the input symbol(s) and the one(s) on the top of the stack.

In the early 1960s Floyd (17) worked on a novel method to describe the reduction process. Rather than using a matrix, he used an analytical syntactic metalanguage; that is, the grammar was defined by a program rather than a matrix. This is called the production language technique and was later modified by Evans (15). Floyd's work will be the fourth method discussed here.

The fifth method discussed is the transition matrix technique introduced by Samuelson and Bauer (37). It is similar to the precedence methods but uses subroutine names as entries in the matrix rather than precedence relationships. This alteration increases the power and efficiency of the parser.

The last method discussed is Knuth's LR(k) method (25), which usually uses two tables. This method employs the state transition concept in which the state is determined by the previous input symbols. The state and the incoming input string determine the appropriate action.

Some of the compilers mentioned later in this paper actually produce a parse tree in memory which is "walked through" at compile time. This approach was studied by Brooker, Morris, and Rohl (3, 4) and Warshall and Shapiro (42).

## 2.2 PRECEDENCE TECHNIQUES

### 2.2.1 Introduction

Precedence methods were originally introduced by Floyd (18) and have been used in table-driven systems since the early 1960s. Later improvements were made by Wirth and Weber (43) and McKeeman (31).

Precedence methods are so named because a precedence or hierarchy is set up among the symbols in the grammar; that is, a relation "greater than," "less than," "equal to," or "conflict" may exist between two symbols, two groups of symbols, or a single symbol and a group of symbols. Not all precedence techniques use all four relations. For each precedence technique a matrix is used to define relationships that exist between finite groups of symbols in the language being parsed. This matrix

determines the actions of an associated shift-reduce parser (see Figure 2.1).

A shift-reduce parser is a machine which uses a stack, the input string being parsed, and the matrix defined above. When parsing begins, the stack is empty and the initial element(s) from the input string is (are) available to the parser (referred to as being "in the window"). For the duration of the process, the parser consults the matrix to determine the relationship between the finite set of symbols in the window and a finite number of symbols on the top of the stack. One of the following actions is taken, depending on that relationship: (1) A shift, which means that one symbol in the window is pushed onto the stack and the next input symbol added to the window. (NOTE: If there is more than one symbol in the window, the one that corresponds to the leftmost input symbol, of the ones in the window, is the one that is pushed on the stack.); (2) A reduction, which means a group of elements from the stack are popped off and reduced to some nonterminal; (3) A syntactical error is detected. This process continues until all the input symbols have been scanned or there is termination with an error.

### 2.2.2 Operator Precedence Method

2.2.2.1 Introduction. In 1963 Floyd (18) developed the operator precedence technique, which was the first and simplest of the precedence techniques. However, this technique is also the most limited with respect to the grammars that can be used.

In the operator precedence method the row and column headings in the matrix are just the terminals in the grammar. The nonterminals are

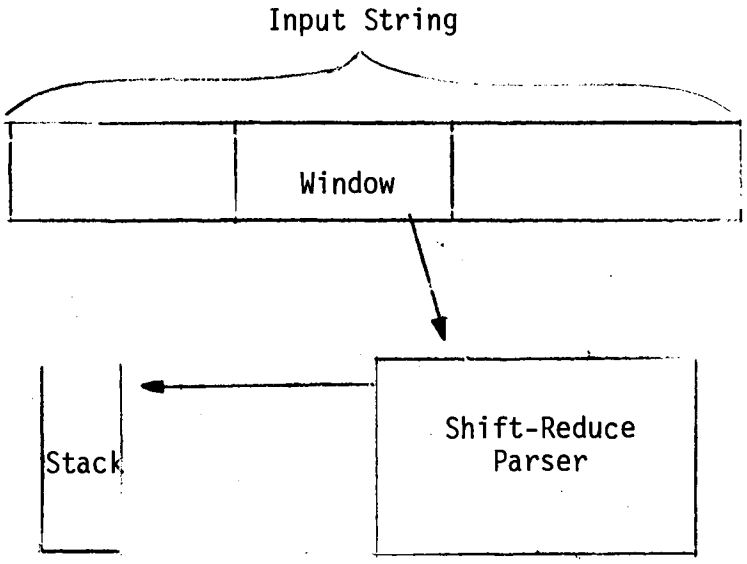


Figure 2.1. Shift-reduce parser.

not involved in the parsing process.

2.2.2.2 Discussion of operator precedence. In order to use the operator precedence technique, the grammar must be an operator grammar. An operator grammar is one in which no production is of the form $N_1$ ::= . . . $N_2N_3$ . . . In other words, no sentential form may contain two adjacent nonterminals. For example, the grammar shown in Figure 2.2 is an operator grammar. (NOTE: eol represents end-of-line.)

```
Prog ::= E eol
   E ::= E + T | T
   T ::= T * F | F
   F ::= (E) | i
```

Figure 2.2. Example grammar 1

Three possible relations, less than (<), greater than (>), and equal to (=), are used in this method. The following rules define these relations. Given the terminals, $T_1$ and $T_2$, and the nonterminals, $N_1$, $N_2$, and $N_3$, then the three basic rules are:

1. $T_1 = T_2$, if and only if there exists a production $N_1$ ::= . . . $T_1T_2$ . . . or $N_1$ ::= . . . $T_1N_2T_2$ . . . .
2. $T_1 < T_2$, if and only if there exists a production $N_1$ ::= . . . $T_1N_2$ . . . where $N_2$ ==>* $T_2$ . . . or $N_2$ ==>* $N_3T_2$ . . . .

3. $T_1 > T_2$, if and only if there exists a production

   $N_1 ::= \ldots N_2T_2 \ldots$ where $N_2 ==>* \ldots T_1$ or

   $N_1 ==>* \ldots T_1N_2$.

If, at most, one of these relations exists between any two terminals in an operator grammar, it is an operator precedence grammar and can be parsed by the operator precedence technique.

The matrix for the grammar in Figure 2.2 is shown in Table 2.1. (NOTE: e.s. represents empty stack.)

A terminal is shifted from the window to the stack if the terminal in the window is greater than or equal to the one on top of the stack. The "greater than" relation indicates that the terminal being pushed on the stack is the head of a phrase which will later be reduced. The "equal to" relation indicates that the terminal being pushed on the stack is not the head of such a phrase. A reduction occurs if the terminal in the window is less than the one on top of the stack. That is, the "less than" relation indicates that the element presently on the top of the stack is the tail of the phrase to be reduced.

In the operator precedence technique, only terminal symbols enter into the decision-making process. At each step of the parse, the left-most simple phrase containing at least one terminal must be found. This phrase is called the prime phrase. More formally, the prime phrase is a phrase which (1) contains no prime phrases other than itself and (2) contains at least one terminal. For instance, the prime phrases for the sentential form T + T * F * i of the grammar in Figure 2.2 are T * F and i. The other phrases are (1) T + T * F + i, (2) T + T * F, and (3) T. The first two phrases are not prime phrases because they include the

Table 2.1. Operator Precedence Matrix for Example Grammar 1

| Stack Symbol | Window Symbol | | | | | |
|---|---|---|---|---|---|---|
| | + | * | ( | ) | i | eol |
| + | > | < | < | > | < | > |
| * | > | > | < | > | < | > |
| ( | > | < | < | = | < | |
| ) | > | > | | > | | > |
| i | > | > | | > | | > |
| e.s. | < | < | < | | < | |

prime phrase T * F. The leftmost T is rejected because it does not contain a terminal.

Parsing is done by finding the entry in the matrix corresponding to the input terminal and the element on the top of the stack and then performing the indicated action. As an example, consider the parse of the string F + T * (T * i) of the grammar shown in Figure 2.2 demonstrated in Table 2.2. (NOTE: The cursor in the input text column indicates the element in the window.)

2.2.2.3 Conclusion. Operator precedence was used in Galler and Perlis' ALGOL compiler (19) and the ALGOL Illinois 7090 compiler (20). This technique is easy to understand, efficient, and flexible; however, it has several drawbacks--notably, that the class of grammars that can be used is relatively small. The rest of the sections in this chapter deal with improvements on the operator precedence technique.

## 2.2.3 Simple Precedence Method

2.2.3.1 Introduction. In an effort to produce a technique which would apply to a wider class of grammars, Wirth and Weber (43) modified Floyd's operator precedence scheme. The new method was called simple precedence (sometimes merely referred to as precedence).

In a simple precedence parser, relationships are defined not just between terminals, but between all terminals and nonterminals. Of course, this significantly enlarges the size of the tables or matrices.

Table 2.2. Example Parse Using the Operator Precedence Technique

| Step | Sentential Form | Input Text | Stack Bottom Top | Action | Prime Phrase | Reduced To |
|---|---|---|---|---|---|---|
| 1 | F + T * (T * i) | F + T * (T * i) eol | Empty | Shift | | |
| 2 | F + T * (T * i) | F + T * (T * i) eol | + | Shift | | |
| 3 | F + T * (T * i) | F + T * (T * i) eol | + * | Shift | | |
| 4 | F + T * (T * i) | F + T * (T * i) eol | + * ( | Shift | | |
| 5 | F + T * (T * i) | F + T * (T * i) eol | + * ( * | Reduce | i | F |
| 6 | F + T * (T * F) | F + T * (T * i) eol | + * ( * | Reduce | T * F | T |
| 7 | F + T * (T) | F + T * (T * i) eol | + * ( | Shift | | |
| 8 | F + T * (T) | F + T * (T * i) eol | + * ( ) | Reduce | (E) | F |
| 9 | F + T * F | F + T * (T * i) eol | + * | Reduce | T * F | T |
| 10 | F + T | F + T * (T * i) eol | + | Reduce | E + T | E |
| 11 | E | F + T * (T * i) eol | Empty | Reduce | E eol | prog |
| 12 | prog | F + T * (T * i) eol | Empty | Accept | | |

2.2.3.2 Discussion of simple precedence. In the simple precedence technique, the grammar is not restricted to an operator grammar. In order to use this parser, two restrictions are made on the grammar. One is that the grammar is required to be uniquely invertible; that is, no two distinct productions can have the same right side. The second is that at most one of the following relationships may hold between any pair of symbols in the grammar.

Given the symbols (terminals or nonterminals), $S_1$ and $S_2$, and the nonterminals, $N_1$, $N_2$, and $N_3$, the three rules defining the simple precedence relationships are:

1. $S_1 = S_2$, if and only if there exists a production $N_i ::= \ldots S_1 S_2 \ldots$
2. $S_1 < S_2$, if and only if there exists a production $N_1 ::= \ldots S_1 N_2 \ldots$ where $N_2 ::= S_2 \ldots$
3. $S_1 > S_2$, if and only if there exists a production $N_1 ::= \ldots N_2 N_3 \ldots$ where $N_2 ==>* \ldots S_1$ and $N_3 ==>* S_2 \ldots$ or $N_1 ::= N_2 S_2 \ldots$ where $N_2 ==>* \ldots S_1$.

Since relations may exist between any pair of symbols (terminals and nonterminals), many more conflicts can arise. In fact, three conflicts are found using the above rules and the grammar in Figure 2.2, page 10. These three conflicts are (1) + = T and + < T), (2) eol = E and eol < E, and (3) ( = E and ( < E. Modifications to the grammar in Figure 2.2 can be made to resolve these conflicts. The grammar in Figure 2.3 defines the same language but has introduced intermediate productions to remove all conflicts.

```
prog ::= E' eol
   E'::= E
   E ::= T' | E + T'
   T'::= T
   T ::= F | T * F
   F ::= (E') | i
```

Figure 2.3. Example grammar 2.

The matrix for the grammar in Figure 2.3 is shown in Table 2.3. Notice that the matrix is considerably larger than the previous one. Since all symbols are involved in the decision-making process in the simple precedence method, one searches for the handle, not the prime phrase.

A full example of parsing the sentential form F + T * (T * i) using the precedence method and the grammar shown in Figure 2.2, page 10, is shown in Table 2.4.

2.2.3.3 Conclusion. The simple precedence technique was implemented by Wirth and Weber (43) in an ALGOL compiler in 1966. The advantage of this method over the operator precedence method is that the grammar is not restricted to an operator grammar. The simple precedence method may be used for a larger class of grammars than the operator precedence method.

Three disadvantages stem from the restrictions placed on the grammar in simple precedence method. First, a suitable grammar might be difficult to construct. Feldman (16) reports the grammar of an average programming language must be manipulated considerably before it is a

Table 2.3. Simple Precedence Matrix for Example Grammar 2

| Stack Symbol | Window Symbol | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | E' | E | T' | T | F | ( | i | * | + | ) | eol |
| E' | | | | | | | | | | = | = |
| E | | | | | | | | | = | > | > |
| T' | | | | | | | | | > | > | > |
| T | | | | | | | | = | > | > | > |
| F | | | | | | | | > | > | > | > |
| ) | | | | | | | | > | > | > | > |
| i | | | | | | | | > | > | > | > |
| * | | | | | = | < | < | | | | |
| + | | | = | < | < | < | < | | | | |
| ( | = | < | < | < | < | < | < | | | | |
| e.s. | = | < | < | < | < | < | < | | | | |

Table 2.4. Example Parse Using the Simple Precedence Technique

| Step | Sentential Form | Input String | Stack Top | Action | Handle | Reduce To |
|---|---|---|---|---|---|---|
| 1 | F + T * (T * i) | F + T * (T * i) eol | Empty | Shift | | |
| 2 | F + T * (T * i) | F + T * (T * i) eol | F | Reduce | F | T |
| 3 | T + T * (T * i) | F + T * (T * i) eol | T | Reduce | T | T' |
| 4 | T' + T * (T * i) | F + T * (T * i) eol | T' | Reduce | T' | E |
| 5 | E + T * (T * i) | F + T * (T * i) eol | E | Shift | | |
| 6 | E + T * (T * i) | F + T * (T * i) eol | E + | Shift | | |
| 7 | E + T * (T * i) | F + T * (T * i) eol | E + T | Shift | | |
| 8 | E + T * (T * i) | F + T * (T * i) eol | E + T * | Shift | | |
| 9 | E + T * (T * i) | F + T * (T * i) eol | E + T * ( | Shift | | |
| 10 | E + T * (T * i) | F + T * (T * i) eol | E + T * (T | Shift | | |
| 11 | E + T * (T * i) | F + T * (T * i) eol | E + T * (T * | Shift | | |
| 12 | E + T * (T * i) | F + T * (T * i) eol | E + T * (T * i | Reduce | i | F |
| 13 | E + T * (T * F) | F + T * (T * i) eol | E + T * (T * F | Reduce | T * F | T |
| 14 | E + T * (T) | F + T * (T * i) eol | E + T * (T | Reduce | T | T' |
| 15 | E + T * (T') | F + T * (T * i) eol | E + T * (T' | Reduce | T' | E |
| 16 | E + T * (E) | F + T * (T * i) eol | E + T * (E | Reduce | E | E' |
| 17 | E + T * (E') | F + T * (T * i) eol | E + T * (E' | Shift | | |
| 18 | E + T * (E') | F + T * (T * i) eol | E + T * (E') | Reduce | (E') | F |
| 19 | E + T * F | F + T * (T * i) eol | E + T * F | Reduce | T * F | T |
| 20 | E + T | F + T * (T * i) eol | E + T | Reduce | T | T' |
| 21 | E + T' | F + T * (T * i) eol | E + T' | Reduce | E + T' | E |
| 22 | E | F + T * (T * i) eol | E | Reduce | E | E' |
| 23 | E' | F + T * (T * i) eol | E' | Reduce | E' eol | prog |
| 24 | prog | | | ACCEPT | | |

precedence grammar. Second, the precedence grammar can be very large. In general, the precedence grammar for a language is larger than the operator grammar for that language. Wirth and Weber (43) report that the matrix for their ALGOL compiler was 70 X 70. Third, the simple precedence method is slower than the operator precedence method.

### 2.2.4 Extended Precedence Method

2.2.4.1 Introduction. McKeeman developed the extended precedence method for his XPL compiler (32). His objective was to extend the class of grammars for which the precedence methods could be applied. In the extended precedence approach, two matrices are used rather than one. The first matrix aids in finding the head and the second in finding the tail of the handle. Two matrices are useful because the more context seen while the decisions are made, the fewer precedence conflicts arise. Among other things, this means more grammars can be used with this technique than with the simple precedence method.

2.2.4.2 Discussion of extended precedence. In the extended precedence method, as in the other methods discussed, the window holds exactly one symbol. Each of the tables is used to compare two elements, the one on the top of the stack and the symbol in the window. If a conflict develops, ternary functions which compare the top two elements on the stack and the symbol in the window are used to resolve the conflict.

One matrix is used to determine if the stack contains the tail of the handle. The relations $<$ and $=$ have been combined to form one

relation, <, and a new third relation, ><, is defined to indicate a conflict situation has arisen. The relationship > indicates the top of the stack is the tail of the handle. The relationship < indicates push the window symbol on the stack and get the next symbol from the lexical analyzer.

Given the symbols (terminals or nonterminals), $S_1$ and $S_2$, and the nonterminals, $N_1$, $N_2$, and $N_3$, the rules to determine the relationships are:

1. $S_1 > S_2$, if and only if there is a production
   $N_1$ ::= . . . $N_2N_3$ . . . where
   $N_2$ ==>* . . . $S_1$ and $N_3$ ==>* $S_2$ . . . or
   $N_1$ ::= $N_2S_2$ . . . where $N_2$ ==>* . . . $S_1$.
2. $S_1 < S_2$, if and only if there is a production
   $N_1$ ::= . . . $S_1N_2$ . . . where
   $N_2$ ==>* $S_2$ . . . or $N_1$ ::= . . . $S_1S_2$ . . . .
3. $S_1 >< S_2$, if both rules 1 and 2 above are satisfied.

The matrix to find the tail of the handle for the grammar in Figure 2.2, page 10, is shown in Table 2.5. Note this matrix only has the entries < and >. For this case, no conflicts arise.

If the tail of the handle is found, then the other matrix is consulted to find the head of the handle. This second matrix is also based on three relationships. The rules that define these relations are the following:

Given the symbols, $S_1$ and $S_2$, and the nonterminals, $N_1$ and $N_2$:

1. $S_1 > S_2$, if and only if $S_1 > S_2$ in the first matrix.

Table 2.5. Extended Precedence Matrix for Example Grammar 1 (Find Tail of Handle)

| Stack Symbol | Window Symbol | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | E | T | F | ( | i | * | + | ) | eol |
| E | | | | | | | < | < | < |
| T | | | | | | < | > | > | > |
| F | | | | | | > | > | > | > |
| ) | | | | | | > | > | > | > |
| i | | | | | | > | > | > | > |
| * | | | < | < | < | | | | |
| + | | < | < | < | < | | | | |
| ( | < | < | < | < | < | | | | |
| e.s. | < | < | < | < | < | | | | |

2. $S_1 < S_2$, if and only if there exists a production $N_1 ::= \ldots S_1 N_2 \ldots$ where $N_2 ==>^* S_2 \ldots$.

3. $S_1 >< S_2$, if either of the following conditions is true: Both 1 and 2 above are satisfied or there exist two production rules $N_1 ::= \ldots S_1 S_2 \ldots$ and $S_2 ::= S_2 \ldots$.

If this new symbol, ><, is encountered, three elements must be consulted. These relationships are represented via ternary functions. The second matrix (the one to find the head of the handle) for the grammar in Figure 2.2, page 10, is shown in Table 2.6. The three element "conflict" functions for the same grammar are presented in Figure 2.4.

In the extended precedence method, all symbols enter into the decision-making process. Therefore, one is searching for the handle rather than the prime phrase.

A full example of parsing using the extended precedence technique and the sentential form F + T * (T * i) of the grammar shown in Figure 2.2 is shown in Table 2.7.

2.2.4.3 Conclusion. The extended precedence method was used in McKeeman's XPL compiler (31, 32). The clear advantage of the extended precedence system over the other precedence methods is that a wider class of grammars can be processed. The disadvantages stem mainly from the fact that now there is not just one large table, but two. According to McKeeman (31), a PL/I compiler's tables were 90 X 45 and 90 X 90. These tables have many empty entries, so some kind of compacting scheme must be used. The large size of the tables makes them hard to produce and makes access slow. In addition, when the tail of the handle is found,

Table 2.6. Extended Precedence Matrix 2 for Example Grammar 1 (To Find Head of Handle)

| Stack Symbol | Window Symbol | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | E | T | F | ( | i | * | + | ) | eol |
| E | | | | | | | > | > | > |
| T | | | | | | > | > | > | > |
| F | | | | | | > | > | > | > |
| ) | | | | | | > | > | > | > |
| i | | | | | | > | > | > | > |
| * | | | > | < | < | | | | |
| + | | >< | < | < | < | | | | |
| ( | >< | < | < | < | < | | | | |
| e.s. | >< | < | < | < | < | | | | |

| | |
|---|---|
| F[e.s., E, +] | = E is head of handle |
| F[e.s., E, eol] | = E is head of handle |
| F[(, E, +] | = E is head of handle |
| F[+, T, *] | = T is head of handle |
| F[(, E, )] | = E is not head of handle |
| F[+, T, +] | = T is not head of handle |
| F[+, T, )] | = T is not head of handle |
| F[+, T, eol] | = T is not head of handle |

Figure 2.4. Ternary functions used to find head of handle.

Table 2.7. Example Parse Using the Extended Precedence Technique

| Step | Sentential Form | Input String | Stack Top | Action | Handle | Reduce To |
|---|---|---|---|---|---|---|
| 1 | F + T * (T * i) | F + T * (T * i) eol | Empty | Shift | | |
| 2 | F + T * (T * i) | F + T * (T * i) eol | F | Reduce | F | T |
| 3 | T + T * (T * i) | F + T * (T * i) eol | T | Reduce | T | E |
| 4 | E + T * (T * i) | F + T * (T * i) eol | E | Shift | | |
| 5 | E + T * (T * i) | F + T * (T * i) eol | E + | Shift | | |
| 6 | E + T * (T * i) | F + T * (T * i) eol | E + T | Shift | | |
| 7 | E + T * (T * i) | F + T * (T * i) eol | E + T * | Shift | | |
| 8 | E + T * (T * i) | F + T * (T * i) eol | E + T * ( | Shift | | |
| 9 | E + T * (T * i) | F + T * (T * i) eol | E + T * (T | Shift | | |
| 10 | E + T * (T * i) | F + T * (T * i) eol | E + T * (T * | Shift | | |
| 11 | E + T * (T * i) | F + T * (T * i) eol | E + T * (T * i | Reduce | i | F |
| 12 | E + T * (T * F) | F + T * (T * i) eol | E + T * (T * F | Reduce | T * F | T |
| 13 | E + T * (T) | F + T * (T * i) eol | E + T * (T | Reduce | T | E |
| 14 | E + T * (E) | F + T * (T * i) eol | E + T * (E | Shift | | |
| 15 | E + T * (E) | F + T * (T * i) eol | E + T * (E) | Reduce | (E) | F |
| 16 | E + T * F | F + T * (T * i) eol | E + T * F | Reduce | T * F | T |
| 17 | E + T | F + T * (T * i) eol | E + T | Reduce | E + T | E |
| 18 | E | F + T * (T * i) eol | E | Reduce | E eol | prog |
| 19 | prog | | | ACCEPT | | |

both matrices must be consulted. Obviously, this is slower than consulting just one table. In the situations where a conflict arises, the ternary functions must also be consulted. This compounds the slowness problem even more.

## 2.3 PRODUCTION LANGUAGE METHOD

### 2.3.1 Introduction

Floyd originally developed the production language technique in 1961 (17). In 1964 and 1965 the method was improved by Evans and Early (15). In the production language method a specially devised metalanguage is used. This means that the specifications are described by a program rather than by a matrix. Pattern matching rather than a matrix table lookup is used to determine the appropriate action the parsing machine should take. As with the previous methods, this one also uses the input symbols and the stack symbols to make decisions.

### 2.3.2 Discussion of Production Language

The production language method can use the largest class of grammars discussed so far because more context is available when the decision is being made than in the methods previously discussed. A number of different languages were developed to implement compilers that used the production language technique. The FPL language will be used for purposes of illustration (17).

The first and most famous of the metalanguages is the FPL (Floyd Production Language). It was later modified and improved by Evans, and, hence, it is sometimes referred to as the Floyd-Evans

production language (FEP) (15). In contrast to the methods in which the parser is driven by a set of tables, the production language method is driven by a program written in FPL. Each statement in FPL describes a reduction as Backus-Naur Form rules do. A shift-reduce parser is used.

Each statement in FPL has the following form: an optional label placed on the far left; then a number of symbols (terminals or non-terminals); a vertical bar denoting the end of the list; an optional arrow meaning replace the previous symbols with the ones that follow; another string of symbols; another vertical bar; and optional star, meaning "shift the current input symbol to the stack and get the next input symbol"; an optional label indicating a jump. For example:

LABEL1 $T_1N_2T_3$ | ==> $N_1T_3$ | *LAB6

means that if $T_3$ is the input symbol, $N_2$ is on the top of the stack and $T_1$ is next to the top, then pop $N_2$ and $T_1$ off the stack, replace them by the nonterminal $N_1$, shift and go to rule LAB6. If the match was not made, ignore that rule and try the one that follows. The symbol '#' matches any symbol. Thus, the rule

$T_1$# | | LAB4

says if the top of the stack is $T_1$, go to LAB4.

The FPL program, generated from the grammar in Figure 2.2., page 10, is shown in Figure 2.5. (The line numbers are included for easier reference in Table 2.8.)

A different production language called CABAL was developed by Dove (13). The main difference between CABAL and FPL is that CABAL includes not only the GO-TO feature but a "call subroutine" feature as well. The other parsers that used precedence methods were strictly

```
PROGO    ( |                                  | *PROGO
         i | ==>                F             | *F1
         # |                                  | ERROR EXIT
E1:      E eol | ==>            PROG          | SUCCESS EXIT
         (E) | ==>              F             | *F1
         E + |                                | *PROGO
         # |                                  | ERROR EXIT
T1:      T * |                                | *PROGO
         E + T # | ==>          E#            | E1
         T # | ==>              E#            | E1
         # |                                  | ERROR EXIT
F1:      T * F# | ==>           T#            | T1
         F# | ==>               T#            | T1
         # | ==>                               | ERROR EXIT
```

Figure 2.5. FPL program for example grammar 1.

Table 2.8. Example Parse Using the Production Language Technique

| Step | On Line # | Symbol in Window | Stack Top | Match? | Reduction | Jump To | Scan? |
|---|---|---|---|---|---|---|---|
| 1 | 1 | i | Empty | No | | | |
| 2 | 2 | i | Empty | Yes | F | 12 | Yes |
| 3 | 12 | + | F | No | | | |
| 4 | 13 | + | F | Yes | T | 8 | No |
| 5 | 8 | + | T | No | | | |
| 6 | 9 | + | T | No | | | |
| 7 | 10 | + | T | Yes | E | 4 | No |
| 8 | 4 | + | E | No | | | |
| 9 | 5 | + | E | No | | | |
| 10 | 6 | + | E | Yes | - | 1 | Yes |
| 11 | 1 | i | E + | No | | | |
| 12 | 2 | i | E + | Yes | F | 12 | Yes |
| 13 | 12 | * | E + F | No | | | |
| 14 | 13 | * | E + F | Yes | T | 8 | No |
| 15 | 8 | * | E + T | Yes | | 1 | Yes |
| 16 | 1 | ( | E + T * | Yes | | 1 | Yes |
| 17 | 1 | i | E + T * ( | No | | | |
| 18 | 2 | i | E + T * ( | Yes | F | 12 | Yes |
| 19 | 12 | * | E + T * (F | No | | | |
| 20 | 13 | * | E + T * (F | Yes | T | 8 | No |
| 21 | 8 | * | E + T * (T | Yes | | 1 | Yes |
| 22 | 1 | i | E + T * (T * | No | | | |
| 23 | 2 | i | E + T * (T * | Yes | F | 12 | Yes |
| 24 | 12 | ) | E + T * (T * F | Yes | T | 8 | No |
| 25 | 8 | ) | E + T * (T | No | | | |
| 26 | 9 | ) | E + T * (T | No | | | |
| 27 | 10 | ) | E + T * (T | Yes | E | 4 | No |
| 28 | 4 | ) | E + T * (E | No | | | |
| 29 | 5 | ) | E + T * (E | Yes | F | 12 | Yes |
| 30 | 12 | eol | E + T * F | Yes | T | 8 | No |
| 31 | 8 | eol | E + T | No | | | |
| 32 | 9 | eol | E + T | Yes | E | 4 | No |
| 33 | 4 | eol | E | Yes | prog | SUCCESS | |

bottom-up parsers, but Dove's uses a combination of top-down (in calling subroutines) and bottom-up (the actual parsing).

A third production language, TGS II, was developed by Cheatham (6). It is similar to FPL, but is more flexible and requires more detailed information from the compiler writer.

A fourth production language system, META PI, was developed at the RCA laboratories in New Jersey (35). META PI is an interactive compiler-compiler. The input to this CC is a list of BNF rules. META PI uses the BNF rules and translates them into its own metalanguage (also called META PI). The output is a recursive descent top-down parser written in META PI. A novel and extremely useful feature of this system is that it allows the compiler writer to use any features from other languages available on the system (FORTRAN, PL/I, etc.). The compiler writer also can write new subroutines, new operators, etc., to include in the production language META PI. The amount of effort required to write the BASIC language compiler (the second language implemented) is impressive (35): It took 96 person hours of which 33 hours were spent at the console. The entire time elapsed between the time the project was started and the language was available was only 3 weeks. The total processor time for the implementation of BASIC was less than six minutes.

An example of a parse using the FPL metalanguage and the sentential form i + i * (i * i) of the grammar shown in Figure 2.2, page 10, is shown in Table 2.8.

### 2.3.3 Conclusion

The production language method is a popular one. As mentioned earlier, it was used by Dove, Cheatham, and O'Neil, each of whom developed his own production language. Ichbiah and Morse (24), Cohen (8), and DeRemer (11) wrote parsers that employed Floyd's production language.

The production language technique has two basic advantages over the previously mentioned precedence techniques. The first is that it can be used with a wider variety of grammars. The second is that once the production language has been learned, it is much easier to write a program describing the syntax than with previous methods. The program-writing approach is less tedious and, hence, less error-prone than the table-writing approach.

A disadvantage of the FPLs in Floyd's original design is that they took up a great deal of space. Attempts were made by Ichbiah and Morse and DeRemer to solve this problem. Ichbiah and Morse (24) developed a technique to merge some of the productions to make a more efficient program using less core. An example they present in (24) of Floyd's productions is reproduced in Figure 2.6. The reduction using Ichbiah and Morse's system is reproduced from (24) in Figure 2.7.

Ichbiah and Morse's technique for merging the productions is performed in two steps; the first applies the algorithm and actually causes the merge. The second checks what was done to make sure the sentences are still correct.

Floyd-Evans productions were successfully applied by Cohen et al. (8) at Brandeis University on their compiler-compiler. The specifications

| | | |
|---|---|---|
| Z( | (( | \|*Z( |
| | (i | \|*Zi |
| | (+ | \|*Z+ |
| | # | \|ERROR |
| Z+ | +( | \|*Z( |
| | +i | \|*Zi |
| | # | \|ERROR |

Figure 2.6. Original FPL program (before reduction).

| | | |
|---|---|---|
| Z( | + | \|*Z+ |
| Z+ | ( | \|*Z( |
| | i | \|*Zi |
| | # | \|ERROR |

Figure 2.7. Reduced FPL program using Ichibiah and Morse's reduction scheme.

are input in BNF. The CC generates the FEP to produce a parser. Cohen states that during the 3 years the system has been in operation, a number of languages have been developed. Two of them were as syntactically as rich as ALGOL. Also, several other simpler languages were generated.

DeRemer (11) also tried the production language approach. He describes an algorithm to translate from BNF grammars to FPL. DeRemer's system was used on the ILLIAC IV machine. It differs from the previously described technique in that it only allows reductions from the top of the stack; therefore, there are fewer rules, and, hence, a diminishing of the space problem.

## 2.4 TRANSITION MATRIX METHOD

### 2.4.1 Introduction

The transition matrix technique was developed in 1960 by Samelson and Bauer (37) and improved in 1967 by Gries (21). It is similar to the operator precedence discussed earlier. Two stacks are used, one for terminals and one for the nonterminals. Recall that in the operator precedence method the terminal in the window is compared to the terminal on the top of the stack. If a reduction is to be made, all rules have to be scanned to find the correct one to apply. The transition matrix technique is similar in that the terminal in the window is compared to the symbol on the top of the terminal stack. If no reduction can be made, the window symbol is pushed on the stack and compared to the next symbol from the source program. However, if a reduction can be performed, the matrix provides the name of the subroutine which performs the correct reduction. To accomplish this, the parser removes the appropriate symbols

from both the terminal and nonterminal stacks and pushes the new nonterminal on the appropriate stack.

### 2.4.2 Discussion of the Transition Matrix Method

In order to parse using the transition matrix approach, one must use an augmented operator grammar. An augmented operator grammar is a grammar in which each derivation of one nonterminal from another is unique. That is, if $N_1$ ==>* $N_2$, then there is only one such derivation.

To produce an augmented operator grammar, one starts with an operator grammar. The grammar in Figure 2.2, page 10, is an operator grammar. It must also be true that no production has more than three symbols on the right-hand side. This condition is also satisfied by the grammar in Figure 2.2. However, in the grammar in Figure 2.2, there are two cases where the derivation from one nonterminal to another is not unique. They are: (1) E ::= E + T and E ::= T, and (2) T ::= T * F and T ::= F. The augmented operator grammar for this language needs two more nonterminals than are in the operator grammar. They are: E' and T'. These new nonterminals are generally referred to as starred nonterminals (SNTS). Because the operator precedence technique is being used but the grammar is no longer an operator precedence grammar, the SNTS, i.e., E' and T', must be stacked on the "terminal" rather than "nonterminal" stack. The augmented operator grammar is shown in Figure 2.8. (NOTE: $\varepsilon$ represents the empty string.)

The matrix for this technique has numbers as entries rather than relations. These numbers represent the subroutine to call. The column headings for the matrix are the terminals in the grammar and the row

```
prog ::= E eol
   E ::= E' T
  E' ::= E + | ε
   T ::= T' F
  T' ::= T * | ε
   F ::= (E) | i
```

Figure 2.8. Augmented operator grammar, example grammar 3.

headings are the characters that will be stacked. An example of the matrix for the grammar in Figure 2.8 is shown in Table 2.9.

Table 2.9. Transition Matrix for Example Grammar 3.

| Stack Symbol | Window Symbol | | | | | |
|---|---|---|---|---|---|---|
| | eol | + | * | ( | ) | i |
| e.s. | 1 | 4 | 5 | 6 | | 8 |
| E' | 2 | 2 | 5 | 6 | 2 | 8 |
| T' | 3 | 3 | 3 | 6 | 3 | 8 |
| ( | | 4 | 5 | 6 | 7 | 8 |

In the transition matrix technique, the parser is searching for the leftmost augmented operator grammar (AOG) prime phrase. An AOG prime phrase is a phrase which (1) contains no AOG prime phrases other than itself and (2) contains at least one terminal or SNTS.

A full example of parsing the same string that was parsed in the other examples, F + T * (T * i), of the grammar shown in Figure 2.8, is shown in Table 2.10.

Table 2.10. Example Parse Using the Transition Matrix Technique

| Step | Sentential Form | Input String | Stack Top | Subroutine # | AOG Prime Phrase | Reduced To |
|---|---|---|---|---|---|---|
| 1 | F + T * (T * i) | F + T * (T * i) eol | Empty | 4 | E + | E' |
| 2 | E' T * (T * i) | F + T * (T * i) eol | E' | 5 | T * | T' |
| 3 | E' T' (T * i) | F + T * (T * i) eol | E' T' | 6 | | |
| 4 | E' T' (T *i) | F + T * (T * i) eol | E' T' ( | 5 | T * | T' |
| 5 | E' T' (T' i) | F + T * (T * i) eol | E' T' (T' | 8 | i | F |
| 6 | E' T' (T' F) | F + T * (T * i) eol | E' T' (T' | 3 | T' F | T |
| 7 | E' T' (T) | F + T * (T * i) eol | E' T' ( | 7 | (E) | F |
| 8 | E' T' F | F + T * (T * i) eol | E' T' | 3 | T' F | T |
| 9 | E' T | F + T * (T * i) eol | E' | 2 | E' T | E |
| 10 | E eol | F + T * (T * i) eol | Empty | 1 | E eol | prog |
| | | Successful Parse | | | | |

### 2.4.3 Conclusion

The transition matrix approach was used by Gries (20) in his ALGOL compiler. Samelson and Bauer (37) also worked with this technique.

One major advantage of using this technique is its speed. The matrix not only identifies what type of operation is to be performed (i.e., reduce, shift, or error), but, by using subroutine numbers, it also identifies exactly what elements are involved and what should be done with them. This feature enables code-generation to be implemented much more easily. The time consuming search process required for other techniques is not necessary in this one.

Another advantage of this technique is that the matrix is machine-generated. Gries has written a program to generate the matrix from an augmented operator grammar in BNF form. Having the matrix machine-generated relieves the language implementor of a great deal of work.

The drawback to using the transition matrix method is that this technique uses a large amount of space. In Gries' ALGOL compiler (20) the matrix is 100 X 75 and requires 500 subroutines.

## 2.5 LR(k) METHOD

### 2.5.1 Introduction

The last technique to be presented in this chapter is the LR(k) approach, a concept developed by Knuth (25). In this method all the symbols in the stack as well as the next k input symbols are used to determine what actions should be taken. Two tables and two stacks are used in this method. One stack holds previous input symbols, and the other holds the present state number. (Optionally, one stack can be

used to hold both the symbols and the states.)

### 2.5.2 Discussion of LR(k)

The technique requires an LR(k) grammar. An LR(k) grammar is one in which the handle is uniquely determined by the entire string to its left and k symbols to the right.

The grammar in Figure 2.2, page 10, is LR(k) for k = 1. Knuth (25) proves that every LR(k) language can also be represented by an LR(1) grammar.

Two tables are needed to parse using the LR(k) method. The first, called the PARSER ACTION table, has the state numbers for rows and terminals for columns. The machine starts in state 0 and changes state according to the input. When the entry corresponding to the current state and the input string is found, the action indicated in the table is performed. As before, this can be a shift (indicated by an S), a reduction (indicated by an R), or an error (indicated by a blank). If a reduction is the appropriate action to take at a certain time, the table will indicate what rule to apply. The rule specifies the number of symbols to be popped off the stack and the new nonterminal to be pushed on. If an error condition is indicated, the machine will stop and output diagnostic information. For both a shift and a reduce, the appropriate next state must be determined. The next state is determined from the second table, called the GO-TO table. Its two dimensions are the present state and the symbol on the top of the stack. The GO-TO table for the grammar in Figure 2.2 is shown in Table 2.11.

Table 2.11. GO-TO Table.

| State | Element on Top of Stack | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | E | T | F | i | + | * | ( | ) |
| 0 | 1 | 2 | 3 | 4 | | | 5 | |
| 1 | | | | | 6 | | | |
| 2 | | | | | | 7 | | |
| 3 | | | | | | | | |
| 4 | | | | | | | | |
| 5 | 8 | 9 | 10 | 11 | | | 12 | |
| 6 | | 13 | 3 | 4 | | | 5 | |
| 7 | | | 14 | 4 | | | 5 | |
| 8 | | | | | 16 | | | 15 |
| 9 | | | | | | 17 | | |
| 10 | | | | | | | | |
| 11 | | | | | | | | |
| 12 | 18 | 9 | 10 | 11 | | | 12 | |
| 13 | | | | | | 7 | | |
| 14 | | | | | | | | |
| 15 | | | | | | | | |
| 16 | | 19 | 10 | 11 | | | 12 | |
| 17 | | | 20 | 11 | | | 12 | |
| 18 | | | | | 16 | | | 21 |
| 19 | | | | | | 17 | | |
| 20 | | | | | | | | |
| 21 | | | | | | | | |

Before a PARSER ACTION table is constructed for grammar in Figure 2.2, page 10, each production rule must be numbered for purposes of reference. This is done in Figure 2.9.

```
0        prog ::= E eol
1           E ::= T
2           E ::= E + T
3           T ::= F
4           T ::= T * F
5           F ::= i
6           F ::= (E)
```

Figure 2.9. Example grammar 1 with each production rule numbered.

The PARSER ACTION table for the grammar in Figure 2.9 (the same grammar as shown in Figure 2.2) is shown in Table 2.12. (NOTE: X indicates inaccessible entries.)

A full parse of the string i + i * (i *i) of the grammar shown in Figure 2.9 using the LR(1) method is shown in Table 2.13.

### 2.5.3 Conclusion

The advantage of the LR(k) method is that the class of LR(k) grammars is large. The drawbacks, however, are the usual technical problems of space and time.

A number of people have worked on answers to the problems of space and time. Korenjack's (26) method involves breaking the language into parts, making tables for each part, and then merging the tables.

Table 2.12. PARSER ACTION Table

| State | Window Symbol | | | | | |
|---|---|---|---|---|---|---|
| | i | + | * | ( | ) | eol |
| 0 | S | | | S | | |
| 1 | X | S | X | X | X | A |
| 2 | X | R,1 | S | X | X | R,1 |
| 3 | X | R,3 | R,3 | X | X | R,3 |
| 4 | | R,5 | R,5 | | | R,5 |
| 5 | S | | | S | | |
| 6 | S | | | S | | |
| 7 | S | | | S | | |
| 8 | X | S | X | X | S | X |
| 9 | X | R,1 | S | X | R,1 | X |
| 10 | X | R,3 | R,3 | X | R,3 | X |
| 11 | | R,5 | R,5 | | R,5 | |
| 12 | S | | | S | | |
| 13 | X | R,2 | S | X | X | R,2 |
| 14 | X | R,4 | R,4 | X | X | R,4 |
| 15 | | R,6 | R,6 | | | R,6 |
| 16 | S | | | S | | |
| 17 | S | | | S | | |
| 18 | X | S | X | X | S | X |
| 19 | X | R,2 | S | X | R,2 | X |
| 20 | X | R,4 | R,4 | X | R,4 | X |
| 21 | | R,6 | R,6 | | R,6 | |

Table 2.13. Example Parse Using LR(k) Technique

| Step | Stack | String Left | Action | Reduction Used |
|---|---|---|---|---|
| 1 | 0[a] | i + i * (i * i) eol | Shift | |
| 2 | 0 i 4 | + i * (i * i) eol | Reduce | 5 |
| 3 | 0 F 3 | + i * (i * i) eol | Reduce | 3 |
| 4 | 0 T 2 | + i * (i * i) eol | Reduce | 1 |
| 5 | 0 E 1 | + i * (i * i) eol | Shift | |
| 6 | 0 E 1 + 6 | i * (i * i) eol | Shift | |
| 7 | 0 E 1 + 6 i 4 | * (i * i) eol | Reduce | 5 |
| 8 | 0 E 1 + 6 F 3 | * (i * i) eol | Reduce | 3 |
| 9 | 0 E 1 + 6 T 13 | * (i * i) eol | Shift | |
| 10 | 0 E 1 + 6 T 13 * 7 | (i * i) eol | Shift | |
| 11 | 0 E 1 + 6 T 13 * 7 ( 5 | i * i) eol | Shift | |
| 12 | 0 E 1 + 6 T 13 * 7 ( 5 i 11 | * i ) eol | Reduce | 5 |
| 13 | 0 E 1 + 6 T 13 * 7 ( 5 F 10 | * i ) eol | Reduce | 3 |
| 14 | 0 E 1 + 6 T 13 * 7 ( 5 T 9 | * i ) eol | Shift | |
| 15 | 0 E 1 + 6 T 13 * 7 ( 5 T 9 * 17 | i ) eol | Shift | |
| 16 | 0 E 1 + 6 T 13 * 7 ( 5 T 9 * 17 i 11 | ) eol | Reduce | 5 |
| 17 | 0 E 1 + 6 T 13 * 7 ( 5 T 9 * 17 F 20 | ) eol | Reduce | 4 |
| 18 | 0 E 1 + 6 T 13 * 7 ( 5 T 9 | ) eol | Reduce | 1 |
| 19 | 0 E 1 + 6 T 13 * 7 ( 5 E 8 | ) eol | Shift | |
| 20 | 0 E 1 + 6 T 13 * 7 ( 5 E 8 ) 15 | eol | Reduce | 6 |
| 21 | 0 E 1 + 6 T 13 * 7 F 14 | eol | Reduce | 4 |
| 22 | 0 E 1 + 6 T 13 | eol | Reduce | 2 |
| 23 | 0 E 1 | eol | ACCEPT | |

[a]NOTE: Double underlines indicate state numbers.

He claims this produces more efficient tables.

Aho and Ullman (1) also present a reduction scheme for the tables. Their method merges states in which every corresponding entry is identical. They also define two states to be "compatible" if each pair of corresponding entries in the two states is either identical or inaccessible for one state. Each pair of compatible states is also merged. Note in Tables 2.11 and 2.12 their technique would merge compatible states 3 and 10 as well as states 14 and 20. They also discuss a method to postpone error detection in certain cases in order to create more compatible pairs. Using that method, the following pairs of states in Tables 2.11 and 2.12 could be merged: 4 and 11, 15 and 21, 8 and 18, 5 and 12, 7 and 17, 13 and 19, 2 and 9, and 6 and 16. After performing the merges indicated above, the tables would have only 12 states as compared to the original tables which had 22.

DeRemer (12) also developed a method for producing compacted LR(k) tables. The savings in his approach arises from the fact that lookahead is used to resolve parsing action conflicts (and, therefore, states can be merged), but lookahead is only used where it is needed. However, this method only works for some LR(k) grammars, the subclass DeRemer refers to as simple LR(k) (SLR(k)) grammars.

Altman (2) wrote a system that can use comprehensive LR(k), called CLR(k) grammars. Altman's claim is that "they include virtually all of the 'practical' grammars with which [they] have come in contact." The CLR(k) grammars may define traditional programming languages such as ALGOL, PL/I, or COBOL. However, they also can describe other languages used for man-machine communications: languages such as job control

languages, and interactive management information decision system languages.

## 2.6 CONCLUSION

This chapter has presented a number of parsing techniques used in table-driven compilers. Each of the techniques is restricted to certain classes of the grammars. Figure 2.10 shows an inclusion tree which pictorically displays the relationships of the grammars that have been presented in this chapter. Note that LR(1) grammars include operator precedence, precedence, extended precedence, and augmented operator grammars.

The operator precedence technique is easy to use, but applies to a very restricted class of grammars. The simple precedence and extended precedence techniques were designed to enlarge the class of grammars that could be employed. The goal was reached, but in the process the parsers became slower and the tables larger.

The transition matrix method is a faster technique and is easy to use. However, this method also takes up a great deal of space.

The LR(k) class of grammars is the largest of all the classes mentioned. This method's main drawbacks are the size of the tables and the difficulty in constructing them. A number of researchers have attacked these problems by developing parsers using the LR(k) basic model and including some reduction schemes.

Each of the above techniques has the disadvantage of requiring large amounts of space. The technique to be discussed in the next chapter, the transition diagram approach, presents a solution to this problem.

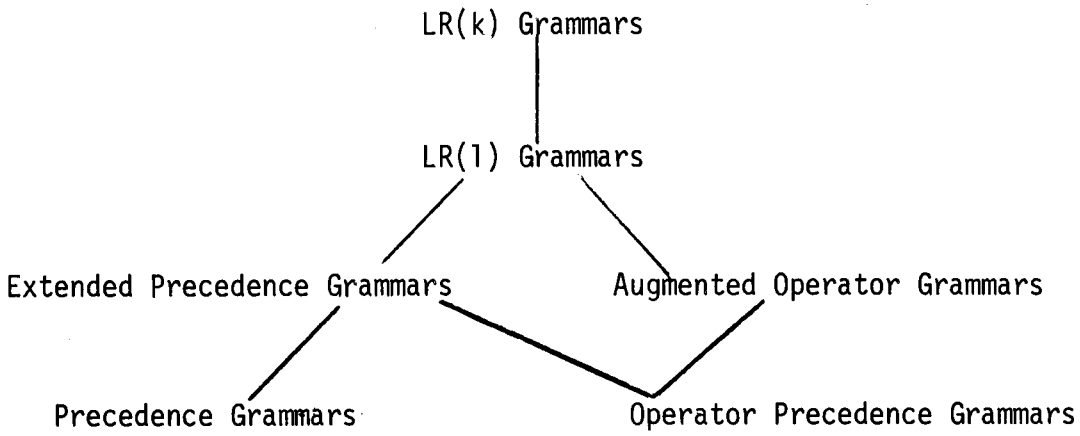


Figure 2.10. Inclusion tree.

# CHAPTER 3

# TRANSITION DIAGRAM METHOD OF PARSING

## 3.1 INTRODUCTION

In this chapter the author presents the last of the table-driven parsing techniques--the transition diagram approach. The original theoretical work on the subject is discussed in section 3.2. Section 3.3 deals with Tindall's (40) implementation of the technique. The last section encompasses the positive and negative aspects of this design and compares it to the techniques discussed in Chapter 2.

## 3.2 CONWAY'S WORK

### 3.2.1 Introduction

In 1963 Conway (9) developed the transition diagram parsing method. In this technique transition diagrams are used to describe the syntax of the language. At parsing time a simulator "walks through" the diagrams to insure that the string being parsed is syntactically correct.

### 3.2.2 Discussion of Transition Diagram Approach

The transition diagram approach requires that the grammar be LR(1). (For an explanation of LR(k) grammars, see section 2.5.2.)

Diagrams form the basis of this system. For each nonterminal in the grammar, a diagram is constructed with numbered nodes connected by directed paths (arrows). Each arrow is labeled with a string or a blank (see Figure 3.1). Each diagram also conforms to the following rules:

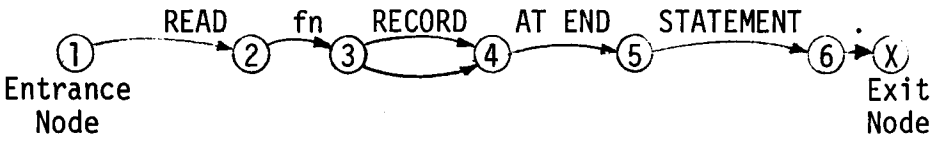


Figure 3.1. Example transition diagram.

1. There is exactly one entrance node and at least one exit node. (Exit nodes are indicated by an "X.")

2. A node may have at most one blank arrow leaving it.

3. A blank path cannot be the only path leaving a node.

4. No blank path goes from the entrance node to the exit node.

5. No two arrows coming out of the same node may be labeled with the same symbol.

6. No two paths P1 and P2 leaving the same node may exist such that the label on P1 is a nonterminal, $N_1$, and the label on P2 is a symbol, $S_1$, where $N_1$ ==>* $S_1$ . . . .

7. No transition diagram may require a jump to itself without first scanning at least one input symbol.

As an example, Figure 3.1 shows the diagram for a COBOL read statement. In that example "fn" stands for a file name and "Statement" represents the nonterminal "STATEMENT." All other labels represent terminals. Notice that node 3 has two possible branches leaving it. Either the word "RECORD" or a blank is acceptable.

An algorithm for scanning the transition diagrams is presented in Figure 3.2. The global variable POINTER_TO_INPUT_SYMBOL holds the pointer to the symbol in the window. The parameter POINTER_TO_PRESENT_NOD points to the node which the parser has reached. The parameter SUCCFL can

```
PROCEDURE PARSE (POINTER_TO_PRESENT_NODE, SUCCFL)
    1.  Do while (node pointed to by POINTER_TO_PRESENT_NODE is not
        an exit node)
        1.1 For all arcs leaving the node pointed to by
            POINTER_TO_PRESENT_NODE which are labeled with terminals do:
                if the label matches the symbol pointed to by
                POINTER_TO_INPUT_SYMBOL then
                    SUCCFL ← "SUCCESS"
                    POINTER_TO_PRESENT_NODE ← node at the end of the
                                              arc which satisfied the
                                              above condition
                    RETURN

        1.2 For all arcs leaving the node pointed to by
            POINTER_TO_PRESENT_NODE which are labeled with nonterminals
            do:
                SUB_POINTER_TO_PRESENT_NODE ← entrance node of the
                                              diagram for this
                                              nonterminal
                SUB_SUCCFL ← "FAIL"
                CALL PARSE (SUB_POINTER_TO_PRESENT_NODE, SUB_SUCCFL)
                    If SUB_SUCCFL = "SUCCESS" then SUCCFL ← "SUCCESS"
                    and Return

        1.3 If the node pointed to by POINTER_TO_PRESENT_NODE has a
            blank arc leaving it:
                SUCCFL ← "SUCCESS"
                POINTER_TO_PRESENT_NODE ← node reached by transversing
                                          the blank arc
                Return

        1.4 SUCCFL ← "FAIL"
            Return

    2.  SUCCFL ← "SUCCESS"
```

Figure 3.2. Algorithm for traversing the transition diagrams.

have either the value SUCCESS or FAIL. The SUCCFL parameter is initially set to FAIL. SUB_SUCCFL and SUB_POINTER_TO_PRESENT_NODE are local variables.

### 3.2.3 Conclusion

The transition diagram approach was used by Tindall (40) in 1975 at the University of Illinois. That implementation is discussed in the remainder of this chapter.

There are three advantages of this system over the systems presented in Chapter 2. First, this system breaks the syntax into small, simple parts which are more manageable for the language implementor. The second advantage of the transition diagram approach is a saving of space. In the matrix methods, many of the entries are empty, thereby wasting space. The transition diagram method describes the syntax specifications of a language in a more efficient manner because it specifies only the nonerror transitions. Finally, the transition diagram approach can use a wide variety of grammars. The class of grammars that can be used with this method is comparable to the class that can be used with the LR(k) method.

However, this method does require the parser to perform recursive descent top-down parsing. Recursive descent parsing is not as efficient as either precedence or LR(k) parsing.

## 3.3 THE CAPS PARSER

### 3.3.1 Introduction

CAPS (Computer-Assisted Programming System) (34) is a translator

writing system developed at the University of Illinois at Urbana-Champaign. It is run on the University of Illinois' PLATO IV computer-aided instruction system.

The CAPS parser (40) uses the transition diagram approach. The actual coding of the diagrams is the language implementor's job. To make that job as easy as possible, Tindall designed a special language with which the diagrams are defined. Each diagram is coded as a sub-routine in an assembler-like language. Henceforth, this language shall be referred to as the "source language" or "Tindallian source language" and will be described in depth in section 3.3.2.

The implementor codes a program that describes his/her language's syntax. This program is processed by the assembler which checks the syntax of the Tindallian source code, reports any errors found, and translates it into object code. The object language will be discussed in depth in section 3.3.3.

### 3.3.2 The Source Language

3.3.2.1 Introduction. The discussion of the source language is divided into three sections: (1) overall organization, (2) parser action instructions, and (3) unparsing. The organization section describes the basic structure of the source language. It includes specification of the overall syntax of the main procedure and subroutines.

The parser action section describes the actual statements that describe the transition diagrams. The need for reverse compilation and the requirements it puts on the parser are the last topics discussed in this section.

3.3.2.2 Overall organization. Only the syntax of the main program and procedures will be discussed here; for other information regarding procedures, the reader is referred to section 3.3.2.3.

The normal form of a Tindallian source program is the main procedure, followed by a sequence of subroutine procedures (also called PROC-END blocks). The main procedure begins with the mnemonic definitions (DEFINE instructions), followed by the global declarations (ALLOCATE instructions), and the error system instructions (CLASS NAME, MASK NAME, FIELDPDN, and ERROR MESSAGE). After these statements comes the body of the main parsing procedure, the PROC-END blocks and the END SYNA, indicating the physical end of the source program (see Figure 3.3).

Each PROC-END block must start with a PROC statement followed by a list of local variable declarations using the ALLOCATE instruction. The body of the procedure follows the ALLOCATE instructions. The last instruction in a procedure is END PROC (see Figure 3.4 for summary).

3.3.2.3 Parser action statements.

3.3.2.3.1 Non-executable statements. The non-executable statements are: DEFINE, ALLOCATE, END SYNA, and the ERROR MESSAGE instructions CLASS NAME, MASK NAME, FIELDPDN, and ERROR MESSAGE. The DEFINE statement permits a permanent assignment of a value to a variable name. The number assigned to the name can be either a decimal or a binary constant. A binary number is indicated by enclosing it in parentheses and preceding it by the word MASK. The ALLOCATE instruction performs the job of a declaration. When it appears in the main procedure, the variable specified is global to the entire program. If the ALLOCATE

```
DEFINE instructions

ALLOCATE instructions

CLASS NAME instructions

MASK NAME instructions

ERROR MESSAGE instructions

FIELDPDN instructions

       Main Parsing Procedure

PROC-END blocks

END SYNA
```

Figure 3.3. Overall program organization.

```
PROC<proc name>[(<arg>{,<arg>})][RETURN<#ret>][NAME(<error print name>)]

{Local variable ALLOCATE instructions}

       PROC Parsing Procedure

END PROC
```

Figure 3.4. Form of PROC-END blocks.

instruction appears at the beginning of a PROC-END block, the variable ALLOCATED is local to that block.

The purpose of CLASS NAME and MASK NAME is to provide the automatic error system (41) with information to convey to a user when an error is encountered. These NAME statements provide a correlation between tokens being parsed and terms in the English language with which the student is accustomed. For example:

NAME DECLARED VARIABLE, 12

NAME ARITHMETIC OPERATOR, 13

would correlate syntactic class 12 with the words "declared variable," and syntactic class 13 with the words "arithmetic operator." Two examples of error messages using these names are:

REPLACE THE UNDERLINED TEXT WITH AN ARITHMETIC OPERATOR

REPLACE THE UNDERLINED TEXT WITH A DECLARED VARIABLE

The FIELDPDN instruction informs the error system which predefined number (PDN) represents "field tokens." PDN's are presented in the section on parser variables. The ERROR MESSAGE instruction is used to override the automatic error analysis system. If an error is encountered and that error matches one given in an ERROR MESSAGE instruction, then the text associated with that instruction is displayed and no further error analysis is performed.

3.3.2.3.2 Executable statements. The first group of instructions to be discussed in this section are SCAN, SEMA, BC, ASSIGN, MASKOFF, MASKON, ADDIT, SUBIT, and FINAL PARSE STATE. The SCAN instruction causes the lexical analyzer to get the next token. SEMA results in a

call to the indicated semantic routine. This semantic routine must be coded by the language implementor in the TUTOR programming language (39). BC is the conditional branch instruction. If the relation indicated is "TRUE," then the branch is taken unconditionally. If the relation indicated is one other than "TRUE," then the two parameters given are compared and if the relation holds between them, the indicated action is taken. The relation can be any one of the following: EQ, NE, LT, LE, GT, GE, ANY, ALL, NOTANY, NOTALL (see Table 3.1). If the relation is satisfied, the resulting action can be either a jump to another location in the current procedure or a call to the error system.

ASSIGN, MASKOFF, MASKON, ADDIT, and SUBIT instructions require two parameters each. ASSIGN sets the value of the first parameter equal to the value of the second parameter. MASKOFF assigns the first parameter the value of the first parameter ADDed with the bitwise complement of the second parameter. MASKON sets the value of the first parameter to the bitwise OR of the two variables. ADDIT assigns the first parameter to the value of the sum of the two parameters. SUBIT subtracts the second parameter from the first: the result becomes the new value of the first parameter. See Table 3.2 for the above information in chart form.

In the main program the FINAL PARSE STATE instruction is used once and only once. The parser should be in this state when execution is requested. If a user requests execution of his/her program and the parser is not in this state, then an error message is given to indicate that the program is incomplete. This check is necessary because the interactive nature of the compiler allows a programmer to request

Table 3.1. Relational Operators Available with the BC Instruction

| Condition | Condition True if |
|---|---|
| EQ | Var 1 = Var 2 |
| NE | Var 1 ≠ Var 2 |
| LT | Var 1 < Var 2 |
| LE | Var 1 ≤ Var 2 |
| GT | Var 1 > Var 2 |
| GE | Var 1 ≥ Var 2 |
| NOTALL | (Var 1 AND Var 2) ≠ Var 2 |
| NOTANY | (Var 1 AND Var 2) = 0 |
| ALL | (Var 1 AND Var 2) = Var 2 |
| ANY | (Var 1 AND Var 2) ≠ 0 |

Where AND is the bitwise AND, Var 1 is the first parameter, and Var 2 is the second parameter.

Table 3.2. ASSIGN, MASKON, MASKOFF, ADDIT, and SUBIT Instructions

| Instruction | Value of First Parameter Becomes |
|---|---|
| ASSIGN | Val (Var 2) |
| MASKON | (Var 1 OR Var 2) |
| MASKOFF | (Var 1 AND (-Var 2)) |
| ADDIT | Var 1 + Var 2 |
| SUBIT | Var 1 - Var 2 |

Where AND is the bitwise AND, OR is the bitwise OR, -Var 2 is the bitwise complement of Var 2.

execution of a program at any time.

The rest of the executable statements deal with linkage to procedures. There are two forms of the CALL instruction and two forms of the RET (return). The CALLI (call immediate) instruction is a subroutine call to the named procedure. The CALL form will scan first and then call the named procedure. Similarly, RETI performs the return without the initial scan, while RET causes a scan and then a return.

Zero to five parameters may be passed when calling a procedure. Each procedure has a fixed number of arguments, and, hence, whenever the procedure is called, the specified number of parameters must be passed. There may be more than one return from a procedure. As with arguments, the number of returns is set by the procedure. If a procedure has multiple returns, all call instructions invoking it must list exactly the same number of return labels as the procedure has returns. This multiple return feature enables the parser to act without backup.

As mentioned earlier, each proc has a name. To call that procedure

CALL <PROCNAME>

is used. However, there is an additional feature which allows multiple entry points into a procedure. At each point where the programmer wants to permit entry, the statement

ENTRY <ENTRY NAME>

is used. If a

CALL <ENTRY NAME>

is encountered, then instead of starting the procedure from the beginning, the parser jumps to the instruction with that entry name. None of the

attributes associated with a proc--number of arguments, number of multiple returns, or local variables--can be respecified at an entry point. Hence, the call must contain the same number of arguments and multiple returns for the entry as it would for the procedure containing the entry.

3.3.2.3.3 Predefined parser variables. There are a large number of predefined parser variables. For a full discussion of these, the reader is referred to (40). A number of the more frequently used ones will be mentioned here.

The language implementors have available to them information regarding the present token being parsed. PDSTP and UDSTP hold the predefined symbol table pointer and user-defined symbol table pointer, respectively, for the given token: CLASS holds its syntactic class, PDN (predefined number), and UDN (user-defined number) hold the predefined and user-defined token's syntactic subclass, respectively. The ITPTP (intermediate text parser token pointer) contains the current location of the intermediate text pointer. The ITPTP is useful for saving information about locations in the intermediate code needed later (such as label locations). There are five temporary locations (TEMP1 to TEMP5) available for the language implementor's individual use.

There are a large number of variables available to make references to the symbol table easier. These contain many items of information regarding the token, such as name, dope vector, links to other symbol table entries, etc.. For further information on this topic, see (40).

3.3.2.4 Unparsing. In the CAPS system, as well as the TICK system, reverse compilation is necessary because compilation is going on while the student is keying in his/her program. That is, these systems do not wait for the whole program to be keyed in before checking it. This feature distinguishes them from batch compilers running on on-line systems. Each time an end-of-line is received, the compiler tries to compile that line. If any errors are detected by the compiler, error analysis is invoked and the user may not continue with program writing until the indicated error has been corrected. Sometimes the student or the error analysis system will discover that the syntax can best be corrected by a modification to a previous line. To solve this problem, the student must return to an earlier state in his/her editing process. Since the system must still be able to supply diagnostic assistance at this new location, the parser, as well as the entire compiler, must be reversed in order to return the entire system to the state it was in when the user originally edited that line.

The way the parser keeps track of all the information it needs to unparse is through the use of a stack. Anytime any location in the parser's domain is changed, this change is recorded by pushing the relevant change onto a special trace stack. In this way, the parser maintains a history of all its previous actions and, thus, unparsing, or backing up, is possible.

### 3.3.3 Object Language

The parser for the CAPS compiler system is a simulator for a 12-bit machine. The machine is driven by an object program. Each of

the source language instructions is translated by the assembler into one or more words to be executed later by the parser. Some instructions, such as SCAN, never use more than one word. Other instructions may need a different number of words to represent them each time they are employed.

There are two basic forms in the object language. The first (referred to as format 1) is used for the instructions that do not use parameters. These are SCAN, ALLOCATE, DEALLOCATE, CALL, RET, SEMA, and BC TRUE. The 12 bits are broken up into two fields. The left seven bits are used for "information" (INFO) and the right five for the OPCODE (see Figure 3.5). For a breakdown of these fields and their possible values, see Table 3.3.




Figure 3.5. Breakdown of format 1 word.

The remaining instructions (format 2) divide the 12 bits into three fields (see Figure 3.6). The format 2 instructions are divided into two groups: branch and general instructions. Both have specifications for two parameters following the first word of the instruction. In format 2 instructions, the leftmost two bits are the miscellaneous (MISC) field. The next five are the operator (OPER) field, and the five on the right represent the parameter's operator (PARMOP) field. The OPER field determines the type of instruction (see Table 3.4). For general instructions the MISC field is zero. For the branch instructions, the MISC field contains the TRUE OPTION (what to do if the

Table 3.3. Meaning of Format 1 Instructions

| OPCODE | Instruction | Meaning of INFO Field |
|---|---|---|
| 0 | SCAN | |
| 1 | ALLOCATE | # of variables to ALLOCATE |
| 2 | DEALLOCATE | # of variables to DEALLOCATE |
| 3 | CALL | # of multiple returns |
| 4 | RET | # of return to take |
| 5 | SEMA | # of semantic routine to call |
| 6 | BC TRUE | See page 63. |




Figure 3.6. Breakdown of format 2 word.

Table 3.4. The OPER Field, Format 2

| OPER | Type of Instruction |
|---|---|
| 0 | BC EQ |
| 1 | BC NE |
| 2 | BC GT |
| 3 | BC GE |
| 4 | BC LT |
| 5 | BC LE |
| 6 | BC NOTALL |
| 7 | BC NOTANY |
| 8 | BC ALL |
| 9 | BC ANY[a] |
| 10 | ASSIGN |
| 11 | PARMASSIGN |
| 12 | MASKON |
| 13 | MASKOFF |
| 14 | ADDIT |
| 15 | SUBIT[b] |

[a]OPER 0-9: branch instructions use MISC field. See Table 3.5.

[b]OPER 10-15: general instructions (MISC field set to 0).

condition is satisfied) (see Table 3.5).

Table 3.5. Use of MISC Field

| MISC | Destination Address[a] |
|---|---|
| 0 | Address constant (jump to) |
| 1 | ERROR constant |
| 2 | ERROR direct address |
| 3 | ERROR indexed address |

[a]For all OPER 0-9 shown in Table 3.4, if the relation is satisfied, then the destination address is as shown above.

The PARMOP field indicates the addressing mode of the two parameters (see Table 3.6). The first parameter can be direct, indexed, or double indexed (see Figure 3.7). The second parameter can be a constant or any of the addressing modes available for the first parameter (see Figure 3.7). For each parameter, one, two, or three words follow the instruction and describe the parameter's exact value.

## 3.4 CONCLUSION

The transition diagram approach has a number of advantages over the methods presented in Chapter 2.

Perhaps the most important advantage of the transition diagram approach is the ease of implementation. Writing the specifications in the Tindallian source language is considerably easier than creating

Table 3.6. Meaning of PARMOP Field (Format 2 Instructions)

| PARMOP | ADDRESSING MODES Var 1 | Var 2 |
|---|---|---|
| 7 | Direct | Direct |
| 8 | Direct | Indexed |
| 9 | Direct | Double Indexed |
| 10 | Direct | Constant |
| 11 | Indexed | Direct |
| 12 | Indexed | Indexed |
| 13 | Indexed | Double Indexed |
| 14 | Indexed | Constant |
| 15 | Double Indexed | Direct |
| 16 | Double Indexed | Indexed |
| 17 | Double Indexed | Double Indexed |
| 18 | Double Indexed | Constant |

| Instruction |
|---|
| A |
| B |
| C |

| If the addressing mode is: | The value of the parameter is: |
|---|---|
| Constant | A |
| Direct | Val (A) |
| Indexed | Val (A + val (B)) |
| Double Indexed | Val (A + val (B + val (C))) |

Figure 3.7. Addressing modes.

tables for the matrix method. The same arguments can be used here as are used for arguing programming in a higher level rather than assembler language. The source program form is easier to write, easier to debug, and easier to maintain than tabular formats. Also, an assembler checks the source program to find errors: none of the implementations of the methods presented in Chapter 2 that this author is aware of have compilers available for checking the tables.

The methods presented in Chapter 2 are bottom-up parsers. The transition diagram method uses top-down recursive descent parsing, a method which does not produce as efficient code as the bottom-up parsers presented in Chapter 2. This is probably its primary disadvantage.

# CHAPTER 4

# THE TICK SYSTEM

## 4.1 INTRODUCTION

In this chapter the author discusses the TICK system in general and its parser in particular. In the first section goals and features of the system are discussed. This discussion leads directly into the explanation of why Tindall's model was chosen as the model for the TICK parser. Following that explanation is a discussion of this parser, including: (1) a comparison of the parsers used in the CAPS and the TICK systems and (2) information regarding the implementation of TICK. The conclusion presents an overall evaluation of TICK and its parser.

## 4.2 GOALS AND FEATURES OF TICK

There were three major goals originally chosen for TICK: (1) to make the system student-oriented, (2) to establish a language lab, and (3) to make the system easy for all to use. Each of these major goals will now be discussed in detail.

In an effort to achieve the first goal, a number of particular subgoals were specified. First, the compiler system is intended to be used in The University of Tennessee courses. The compiler is not completed at this time, so this goal has not yet been met. However, by fall, 1977, the Computer Science Department expects some courses to be using the TICK system for machine problems. Eventually, it is planned

that many courses will make use of this system.

In order to use TICK in courses, the system would obviously have to run on one of The University's large computers. Hence, the choice of machine was narrowed to the DEC-10 or the IBM-360. The DEC-10 was chosen because it was (at the time) considerably underutilized (not running even close to capacity). Also, the compilers available with it were limited (e.g., no PL/I). Probably the most convincing argument in favor of the DEC-10 is that it is highly interactive. The compilers that are available on the 10 now are simply batch compilers running on an on-line system. The TICK compiler is a truly interactive one.

Being highly interactive has a number of positive features. The first is simply that using an on-line system is more convenient and enjoyable than a batch system. One of the Computer Science Department's goals is to encourage student interest and use of the facilities; hence, making the job of writing and debugging programs easier is an appropriate goal. The TICK compiler is interactive in that it compiles each line of code as it is received. In consequence, a second positive feature is that error messages are given to the student immediately, rather than at the end of program input.

This real-time interaction makes it necessary for the TICK system to contain an editor so a student can make changes in his/her source program. This editor will be used when one is designing a language (using the TWS aspect of TICK), writing a program, or just setting up a file. The editor uses function keys for commands. The function keys move the cursor and enable the user to move from one of TICK's parts (implementing a language, compiling a program, etc.) to another. When

the system is fully operational, all terminals will have a copy of these commands attached to them.

The last, and possibly most important, student-oriented feature is that good diagnostics are available. Some other systems produce compiler error messages that students cannot decipher. A message that does not inform the student what and where the problems are is of little use. As mentioned earlier, the TICK compiler tries to compile each line after it has been keyed in. When an error has been detected, the TICK error system will print out an easily understood message explaining what and where the problem is. For most problems the error system will also make suggestions to the student indicating possible solutions. This feature should aid the neophyte programmer considerably.

One of the greatest advantages of the TICK error diagnostic system is that it is automatic. By defining the syntax of the language, the language designer has given the system all the information it needs to generate language-dependent understandable messages. For more detailed information on the TICK error diagnostic system, the reader is referred to Button (5).

The second basic goal of the TICK project is to establish a language laboratory. Since the compiler in the TICK system is part of a translator writing system, new languages can be implemented quite easily (compared to writing a full compiler from scratch). Therefore, students in language design courses will actually be able to implement languages or parts of languages as a concrete exercise. The TICK system will enable students to get experience in a subject with which most students are able to deal only on an abstract level.

It is estimated that once individuals are familiar with the entire TICK system including Tindallian source language, it will take only a half-person year of work to bring up a new language. As mentioned earlier, the language implementor has the standard TICK editor available to him/her during the implementation stage. This should also ease the implementation process. Once a person is familiar with the editor, he/she does not have to learn new commands.

The third major goal of the TICK system is to make all phases of the system easy for people to use. The TICK designers have included two features especially to aid users: (1) Choices are presented in "menu" style; (2) On-line help is available at all times. Whenever a choice must be made by the user regarding which parts of the TICK system he/she wants to employ next, all the possible choices are displayed in a "menu." This enables the user to see all the choices available. Whenever possible, the same code is used for a command (e.g., "Z" always means leave TICK, "H" always means help, "E" always means edit). On-line help is always available. This will prove extremely useful for the neophyte and could be beneficial for the more experienced user as well.

## 4.3 WHY TINDALL'S METHOD WAS CHOSEN

There are two basic reasons why Tindall's method was chosen as the model for TICK's parser: (1) It was table-driven; (2) It facilitated language implementation.

Since TICK is to be used as a language lab, it was appropriate to make TICK's compiler system table-driven. This required that the compiler be a TWS or a CC. Of all the methods considered, Tindall's

made the language implementor's job easiest for the reasons cited in section 3.4.

## 4.4 THE TICK PARSER

### 4.4.1 Introduction

In this section the author discusses the TICK parser. The parsers on the TICK and CAPS systems are so similar that most of the material presented in section 3.3 is true for the TICK parser as well. That material will not be repeated here. This section will first describe the differences between the CAPS and TICK parsers and then will discuss the implementation of the TICK parser and present an extensive example.

### 4.4.2 Differences Between the CAPS and TICK Parsers

There are five major differences between the parsers for the two systems: (1) the semantic routine processing, (2) code generation, (3) the parser variables, (4) error analysis system, and (5) implementation.

In the CAPS system's parser each team of language designers is required to code each and every semantic routine they need. The TICK system recognizes that many languages can utilize some of the same routines, and, therefore, a library of frequently used routines has been made available to all language designers. Of course, each language being implemented on the system will require some of its own individual, unique semantic routines. The language implementors have available both the library of semantic routines and the facilities for writing their own routines. The language implementors may choose to write such routines in

any DEC-10 system-supported language (including ALGOL, ASSEMBLER, FORTRAN, etc.). On the CAPS system, such implementation is limited to the TUTOR language.

The way the TICK system generates code and executes programs is considerably different from the way the CAPS system does. The CAPS system never generates any machine code; rather, it produces intermediate text which must be interpreted later. The intermediate text it generates is simply a tokenized version of the source text. This produces two problems: (1) Since no universal intermediate text is generated, one execution package cannot execute all languages; and (2) The execution supervisors must essentially reparse the entire program, thus, slowing down the process considerably. In order to avoid these problems, the TICK compiler generates object code. The language implementor has available a set of macros. Utilizing the CODEGEN parser instruction, the language implementor indicates when one of these macros is to be generated. The macros are later expanded into DEC-10 object code. The object code is executed when the student runs his/her program. This means that the execution phase will be a good deal quicker in the TICK system than it is in the CAPS system. Even more important, the language implementor in the CAPS system is burdened with perhaps an additional half-person year of work in order to write and debug an interpreter for his/her language. For the TICK implementor, this burden has been eliminated.

The differences between the parser variables available in CAPS and TICK are minor. All the variables discussed in section 3.3.2.3.3 are available in TICK except PDN and UDN. These were found not to apply

in the TICK version.

The main difference between the TICK and CAPS error systems is the amount of context each utilizes when suggesting possible corrections. As a direct result of that, the two TWS' differ in the number of suggestions that they will make when an error is detected.

In the CAPS system, when an error is detected in the middle of the current line, a change is suggested to the student if the change enables the line to be parsed from the beginning of the program through the location where the error is detected. In the TICK system, the change will only be suggested if it will enable the line to be parsed from the beginning of the program to the end of the line. Thus, the full context around the error is taken into account. An example of how these would differ is shown using the ALGOL string

If (A = B, C = D) THEN Z := 1;.

Upon the entrance of the line, an error would be detected by either system when parsing the ",". The CAPS error system would suggest two alternatives: (1) replace the "," with a ")" and (2) replace the "," with a boolean operator such as "AND" or "OR." The first suggestion would enable the string to be parsed through IF (A = B), but then another error would be encountered. This might tend to cause the neophyte programmer considerable confusion. TICK's error system would only suggest the second choice given above, which would probably be closer to the student's intention.

In the CAPS' error system, the algorithm used to find possible suggestions may not find all applicable ones. A goal of the designer of the TICK error analysis system is to define a set of rules for language

implementors which guarantees that the algorithm used by the parser will find every possible suggestion. Therefore, without the language implementor's aid, the algorithm the TICK system uses should find more valid suggestions than the CAPS algorithm. Also, this algorithm should eliminate making invalid suggestions the CAPS system makes due to its failure to reparse the entire current line. With the language implementor's aid, TICK's error analysis system should find all the possible suggestions. For more information on the error systems, the reader is referred to Button (5) and Tindall (41).

The remaining major differences are the ones stemming from the machine each compiler employs. The CAPS system was written on the PLATO system; TICK's on a PDP-10. Since at this time only three installations use the PLATO IV system, the CAPS system is not as portable as the TICK system.

At the present time, the CAPS system is not being used. Some major problems developed, all due to the fact that CAPS was being run on the PLATO system. The PLATO system does not allow execution of user programs in object code form. All user programs run on PLATO must be interpreted, including CAPS itself, and this interpretation of CAPS slows down the process considerably. As the DEC-10 allows user programs to be executed in object code, TICK will obviously run faster.

A more serious speed problem is encountered when a student tries to run a program which was compiled using the CAPS system. As mentioned earlier, the output from the CAPS compiler is a tokenized version of the student's original program. That output must be processed through an interpreter specifically written for the high-level language the student

is using. The program must then be processed by PLATO's interpreter. That is, when a student executes a program compiled on the CAPS system, it must go through two levels of interpretation which causes execution of a program to be extremely slow (see Figure 4.1).

The third major problem contributing to the CAPS system being temporarily taken out of use is the resource limitations the PLATO system management imposes on users. Each user is severely restricted in the amount of space available to him/her. These include the following limitations:

1. All programs limited in size to 8,000 words.
2. Total addressable memory limited to 1,500 words per program.
3. Total amount of additional storage allocation available limited to 1,500 words per user.
4. Storage allocated limited to 1,000 words per terminal.
5. CPU usage limited to 2-3 msecs per real-time second.
6. Disk accesses limited to only a few per half hour.

None of these limitations is nearly as severe when the DEC-10 is used as the host machine.

### 4.4.3 Implementation

The TICK parser is written in ALGOL. It is a simulator which executes the program generated by the language implementor's source code. This code is first checked for errors and then translated into object code by the assembler. Figure 4.2 shows a block diagram of the major modules of the TICK parser. Figure 4.3 shows the information which is included in the parser's "domain." Notice that both the symbol table

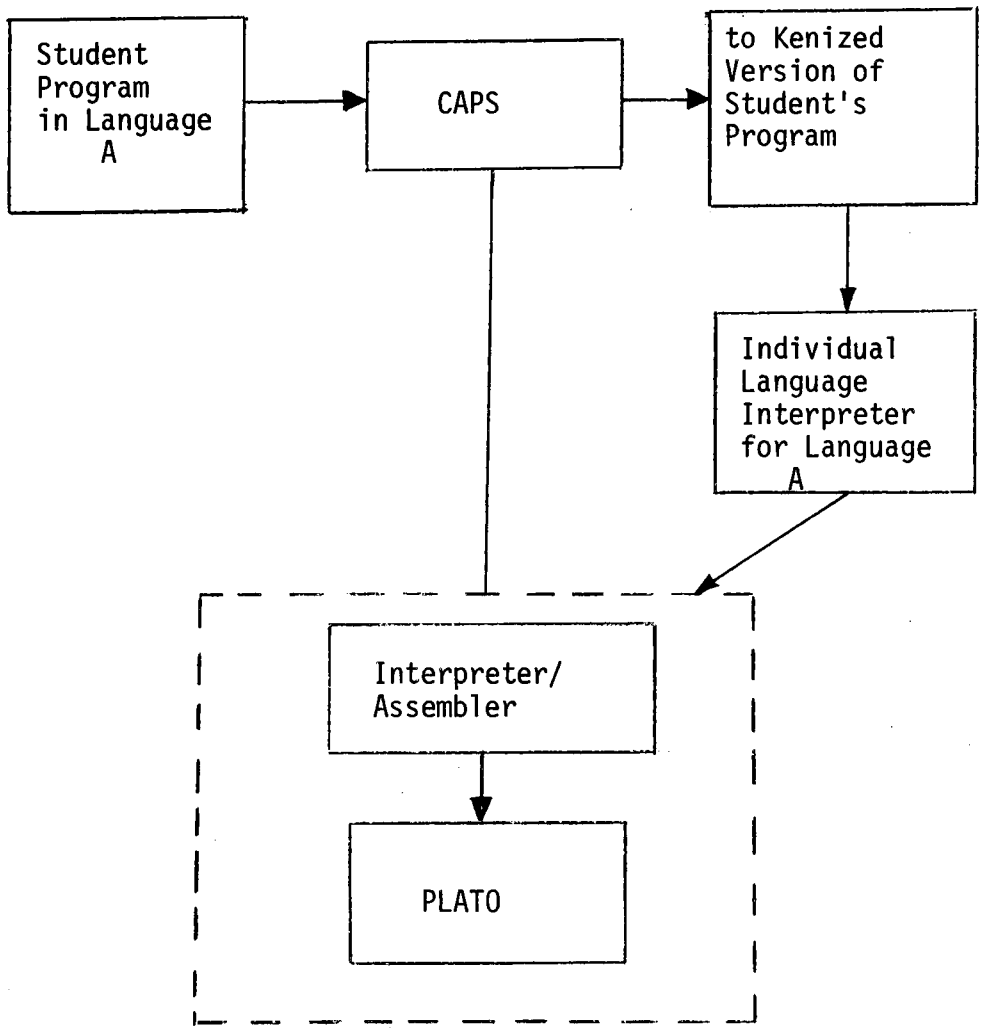


Figure 4.1. Block diagram of levels of interpretation in CAPS-PLATO system.

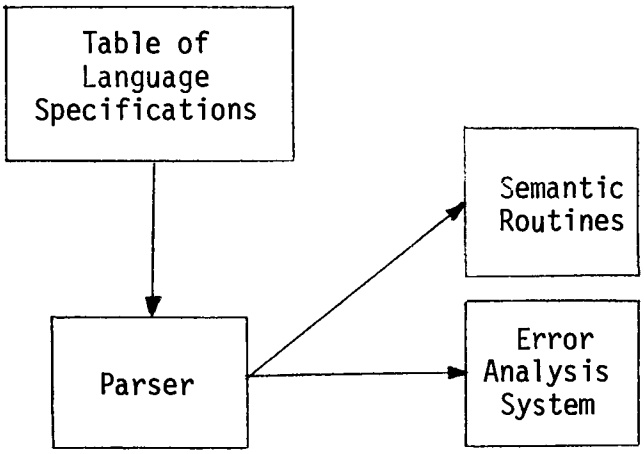


Figure 4.2. Major modules of the TICK system parser.

| Object Code Form Of Language Specifications | Trace Stack | Code Generation Stack | Parser Stack | Symbol Table | Parser Program |
|---|---|---|---|---|---|

Figure 4.3. Inside parser's "domain."

and the object code form of the language specifications are in that area at compile-time.

This section closes with an extensive example. This example includes (1) the BNF description of a grammar (see Figure 4.4), (2) the transition diagrams corresponding to the grammar in Figure 4.4 (see Figure 4.5), (3) the Tindallian source code for Figure 4.4 (see Figures 4.6, 4.7, and 4.8), and (4) two example parses (see Tables 4.1 and 4.2).

Regarding the procedures in Figures 4.6, 4.7, and 4.8, note the lines starting with the character "*" represent comments. Also, only the first PROC has a name. The error system will utilize this name when communicating with the student programmer. The numbers appearing on the far left are for reference only (used in Figures 4.1 and 4.2).

As an example, the expression, i + i * (i * i), is parsed in Table 4.1. As an example of an expression containing an error, an attempt is made to parse the string i * ) i * i) in Table 4.2. Note that after a "*" there must be a TERM. Since a term must start with a FACTOR, then the next character after a "*" must be the first token of a FACTOR. The ")" does not satisfy that condition so an error is found.

## 4.5 CONCLUSION

The TICK system has an excellent chance of being a highly successful software package. Almost all of its original goals have been met. A sound original model (CAPS) was chosen. (NOTE: In this chapter many problems in the CAPS system have been cited. Virtually all these problems are related to the machine on which the system was run. None

```
E ::= T + E | T

T ::= F * T | F

F ::= i | (E)
```

Figure 4.4. Example grammar.

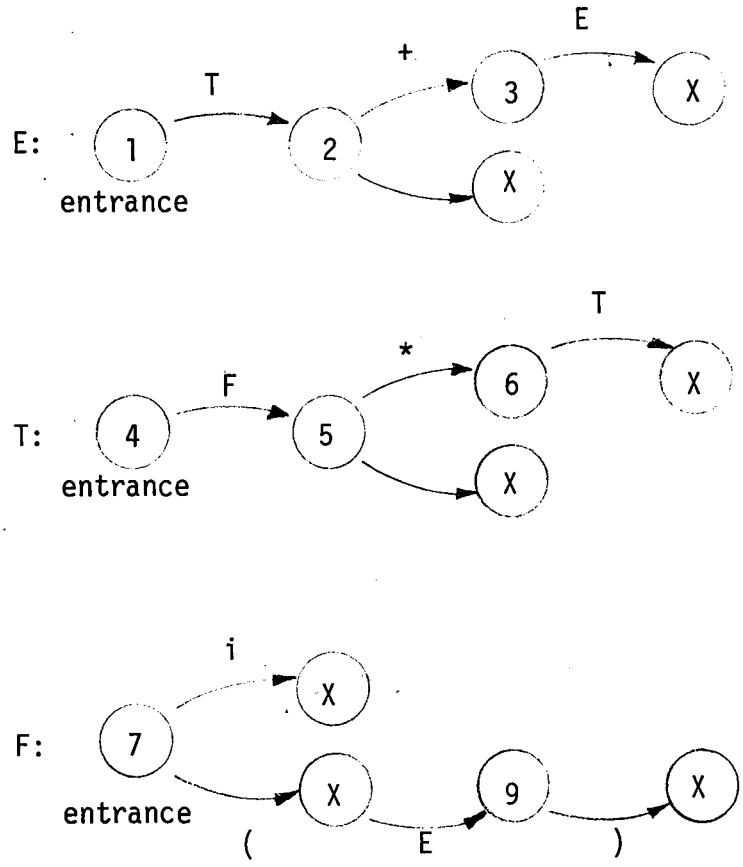


Figure 4.5. Transition diagrams generated for grammar in Figure 4.4.

```
PROC   EXPR   RETURN 2  NAME (EXPRESSION)

*      Call immediate (scan not initiated first) followed by the labels
*      to jump for a Return 1 and Return 2.

1      CALLI       TERM, THEN LOOP, RETE2

*      Checking for a plus "+" sign, if found, continue, if not found,
*      RETE1 (expression found).

2      LOOP BC             NE, STP, PLUS, RETE1

*      If a plus sign was found, above, then to make an expression
*      the "+" must be followed by a term.

3      CALL                TERM, THEN LOOP2, ERRETE

*      If another term was found, generate macro to add and see if the
*      next token is also a "+."

4      LOOP2       CODEGEN       PLUS
5                  BC            TRUE, LOOP

*      The following are possible returns:  RETE1 (expression found);
*      RETE2 (expression not found); for both--do not scan first.

6      RETE1   RETI         1
7      RETE2   RETI         2

*      Calling error system.

8      ERRETE  BC           TRUE, ERROR  68
               END PROC
```

Figure 4.6. Tindallian source code for an EXPRESSION.

```
PROC    TERM    RETURN 2

*       A term must start with a factor, so that PROC is called.  The
*       PROC FACTOR also has 2 return points so 2 return labels are
*       required.

1       CALLI       FACTOR, THEN LOOPT, RETT2

*       Looking for multiplication sign, "*," if found, continue, if
*       not found, RETT1 (term found).

2       LOOPT  BC       NE, STP, MULT, RETT1

*       If a multiplication sign was found, another factor must follow,
*       so call factor again.

3       CALL      FACTOR, THEN LOOPT2, ERRETT

*       If a factor was found, generate the macro to multiply and loop
*       back to LOOPT.

4       LOOPT2      CODEGEN      MULT
5                   BC           TRUE, LOOPT

*       Returns are:  RETT1 (term found); RETT2 (term not found); for both
*       returns:  do not scan before returning.

6       RETT1       RETI      1
7       RETT2       RETI      2

*       Calling error system.

8       ERRETT  BC          TRUE, ERROR 69
                END PROC
```

Figure 4.7. Tindallian source code for TERM.

```
PROC      FACTOR      RETURN 2

*       See first if we have F ::= (E)--check for "("
*       If don't have an "(" then to be a factor, the token must be i.

1       BC      EQ, STP, LEFTP, PARENS
2       BC      NE, CLASS, VAR, RETF2

*       If an i was found, generate the appropriate macro and
*       RET1 (factor found).

3       CODEGEN     PUSH,STP
4       RET         1

*       We end up here if the first token was a "(."  So, now
*       find an "E," then ")," otherwise, we cannot find a factor.

5       PARENS  CALL        EXPR, THEN CONTN, ERRETF
6       CONTN   BC          NE, STP, RIGHTP, ERROR 42
7               RET         1

8       RETF2    RET        2
9       ERRETF   BC         TRUE, ERROR 43
                 END PROC
```

Figure 4.8. Tindallian source code for FACTOR.

Table 4.1. Example Parse Using the Transition Diagram Technique

| Step | Input String | PROC In | Line # |
|---|---|---|---|
| 1 | i + i * (i * i) eol | EXPR | 1 |
| 2 | i + i * (i * i) eol | TERM | 1 |
| 3 | i + i * (i * i) eol | FACTOR | 1 |
| 4 | i + i * (i * i) eol | FACTOR | 2 |
| 5 | i + i * (i * i) eol | FACTOR | 3 |
| 6 | i + i * (i * i) eol | FACTOR | 4 |
| 7 | i + i * (i * i) eol | TERM | 2 |
| 8 | i + i * (i * i) eol | TERM | 6 |
| 9 | i + i * (i * i) eol | EXPR | 2 |
| 10 | i + i * (i * i) eol | EXPR | 3 |
| 11 | i + i * (i * i) eol | TERM | 1 |
| 12 | i + i * (i * i) eol | FACTOR | 1 |
| 13 | i + i * (i * i) eol | FACTOR | 2 |
| 14 | i + i * (i * i) eol | FACTOR | 3 |
| 15 | i + i * (i * i) eol | FACTOR | 4 |
| 16 | i + i * (i * i) eol | TERM | 2 |
| 17 | i + i * (i * i) eol | TERM | 3 |
| 18 | i + i * (i * i) eol | FACTOR | 1 |
| 19 | i + i * (i * i) eol | FACTOR | 5 |
| 20 | i + i * (i * i) eol | EXPR | 1 |
| 21 | i + i * (i * i) eol | TERM | 1 |
| 22 | i + i * (i * i) eol | FACTOR | 1 |
| 23 | i + i * (i * i) eol | FACTOR | 2 |
| 24 | i + i * (i * i) eol | FACTOR | 3 |
| 25 | i + i * (i * i) eol | FACTOR | 4 |
| 26 | i + i * (i * i) eol | TERM | 2 |
| 27 | i + i * (i * i) eol | TERM | 3 |
| 28 | i + i * (i * i) eol | FACTOR | 1 |
| 29 | i + i * (i * i) eol | FACTOR | 2 |
| 30 | i + i * (i * i) eol | FACTOR | 3 |
| 31 | i + i * (i * i) eol | FACTOR | 4 |
| 32 | i + i * (i * i) eol | TERM | 4 |
| 33 | i + i * (i * i) eol | TERM | 5 |
| 34 | i + i * (i * i) eol | TERM | 2 |
| 35 | i + i * (i * i) eol | TERM | 6 |
| 36 | i + i * (i * i) eol | EXPR | 2 |
| 37 | i + i * (i * i) eol | EXPR | 6 |
| 38 | i + i * (i * i) eol | FACTOR | 5 |
| 39 | i + i * (i * i) eol | FACTOR | 6 |
| 40 | i + i * (i * i) eol | FACTOR | 7 |
| 41 | i + i * (i * i) eol | TERM | 4 |
| 42 | i + i * (i * i) eol | TERM | 5 |

Table 4.1, Continued

| Step | Input String | PROC In | Line # |
|---|---|---|---|
| 43 | i + i * (i * i) eol | TERM | 2 |
| 44 | i + i * (i * i) eol | TERM | 6 |
| 45 | i + i * (i * i) eol | EXPR | 4 |
| 46 | i + i * (i * i) eol | EXPR | 5 |
| 47 | i + i * (i * i) eol | EXPR | 2 |
| 48 | i + i * (i * i) eol | EXPR | 6 |
| | Successful Parse | | |

Table 4.2. Example Parse of an Incorrect String

| Step | Input String | PROC In | Line # |
|---|---|---|---|
| 1 | i * ) i * i) eol | EXPR | 1 |
| 2 | i * ) i * i) eol | TERM | 1 |
| 3 | i * ) i * i) eol | FACTOR | 1 |
| 4 | i * ) i * i) eol | FACTOR | 2 |
| 5 | i * ) i * i) eol | FACTOR | 3 |
| 6 | i * ) i * i) eol | FACTOR | 4 |
| 7 | i * ) i * i) eol | TERM | 2 |
| 8 | i * ) i * i) eol | TERM | 3 |
| 9 | i * ) i * i) eol | FACTOR | 1 |
| 10 | i * ) i * i) eol | FACTOR | 2 |
| 11 | i * ) i * i) eol | FACTOR | 9<br>error detected |

of the problems is related to the design of CAPS itself.) A number of places where improvements to that model could be made were identified by the TICK system designers. These improvements were implemented and are included in the TICK version. Prior knowledge of the CAPS' system gave the TICK system implementors extra insight and an opportunity to produce a better system. The author's thoughts and suggestions regarding the project are presented in Chapter 5.

# CHAPTER 5

# CONCLUSION

In this chapter the author will discuss her ideas and suggestions regarding the TICK system in general and some possible improvements of it.

There are three primary suggestions for the TICK system. The first, and the most important, is that some better error system be available on the assembler (the program that translates from Tindallian source code to object code). The error system currently available with the assembler is inadequate. Since the TICK compiler has excellent on-line interactive debugging features, this author suggests making those features available to the language designers during the debugging stage of their language specifications. That would involve writing the language specifications for Tindallian source code and setting it up as a language capable of being parsed using TICK. Then the TICK system's compiler, with all its interactive features, could aid the language designer in debugging his/her language specification program. Once the program is debugged, the designer could use the present assembler to generate the object code.

The other two suggestions relate to the parser directly. In an attempt to make the parser more comprehensive, the TICK group might consider allowing a language designer to input the language specifications in BNF. This author feels that the Tindallian language is easy to learn; however, BNF is a much more common form of language specification. If

the TICK system is to be used elsewhere, it might prove easier for others to input BNF rather than Tindallian source code.

The last suggestion is a tentative one. If the top-down recursive descent parser now being used with TICK turns out to be too slow and inefficient, the TICK system's development group might consider switching to bottom-up parsing. If that were the case, the author would suggest considering both the production language method and the LR(k) method (using one of the condensing algorithms). These are considered the best candidates among the bottom-up methods presented in Chapter 2.

Two other comments to note are in the areas of documentation and speed of the host computer. After using the TWS aspect of the TICK system, this author has found that good hard-copy, off-line documentation is needed for the system to be of any use. If no more documentation in the form of usage manuals and other such reference material becomes available, the TICK system will not be utilized at anywhere near its full potential. The language designer, particularly, will need a good deal of information regarding what tables to use and how to fill them in. Also, a manual is needed to aid the language implementor in learning the Tindallian source code. This manual should include many examples of code.

It is the author's opinion that the major drawback of the TICK system will be its speed. Both the DEC-10 and TICK systems contribute to this problem. When the TICK system designers chose the DEC-10 to be the host machine for the TICK project, the DEC-10 was not close to saturation. At the present time, however, that situation has changed. It is now the case that at times the response time of the DEC-10 system

is extremely slow due to heavy use. Obviously, if the TICK system is used in many computer science courses, the number of students needing the DEC system will increase tremendously.

The heart of the speed problem, however, is the TICK system itself. TICK, much more than the DEC-10 system, will be responsible for the unacceptably slow speed. Two suggestions which would aid in alleviating this problem are: (1) microprogramming a machine to have Tindallian object language as its machine language and (2) reprogramming TICK's lexical analyzer and parser in DEC-10 assembler language (they are now programmed in ALGOL).

The author considers TICK to be an excellent system. If the slowness problem can be solved, this system will not only achieve all the goals previously mentioned, but also exceed all expectations.

# BIBLIOGRAPHY

# BIBLIOGRAPHY


1. Aho, Alfred V. and Jeffrey D. Ullman, The Theory of Parsing, Translation and Compiling. Vol. 1 and 2. Englewood Cliffs: Prentice-Hall, Inc., 1972.

2. Altman, Vernon E. A Language Implementation. Masters Thesis. Cambridge: Massachusetts Institute of Technology, 1974.

3. Brooker, R. A.; D. Morris; and J. S. Rohl. "Trees and Routines," The Computer Journal, Vol. 5, The British Computer Society, 1963, pp. 33-47.

4. Brooker, R. A.; D. Morris; and J. S. Rohl. "Experience with the Compiler Compiler," The Computer Journal, Vol. 9 (February, 1967), pp. 345-349.

5. Button, Beverly. Forthcoming Masters Thesis. Department of Computer Science, The University of Tennessee, Knoxville.

6. Cheatham, T. E. "The TGS-II Translator-Generator System," New York: PROC IFIP Congress, 1965, pp. 592-593.

7. Cocke, John and J. T. Schwartz. Programming Languages and Their Compilers. New York: Courant Institute of Mathematical Sciences, 1970.

8. Cohen, Jacques. "Experiences with a Conversational Parser Generating System," Software-Practice and Experience, Vol. 5, New York: John Wiley & Sons, Ltd., 1975, pp. 169-180.

9. Conway, M. E. "Design of a Separable Transition-Diagram Compiler," CACM, Vol. 6, July, 1963, pp. 396-408.

10. Davis, Alan M. An Interactive Analysis System for Execution-Time Errors. Report UIUCDCS-R-75-695. Ph.D. Dissertation. Department of Computer Science, University of Illinois at Urbana-Champaign, January, 1975.

11. DeRemer, Franklin L. "Generating Parsers From BNF Grammars," SJCC, 1969, pp. 793-799.

12. DeRemer, Franklin L. "Simple LR(k) Grammars," CAMC, Vol. 14, Number 7, pp. 453-459.

13. Dove, Richard K. "Design Highlights of CABAL--A Compiler-Compiler," FJCC, 1968, pp. 1321-1328.

14. Ershov, A. P. "A Multilanguage Programming System Oriented to Languages Description and Universal Optimization Algorithms," ALGOL 68 Implementations. J. E. L. Peck, Editor. Amsterdam: North-Holland Publishing Company, 1971, pp. 143-162.

15. Evans, Arthur. "An ALGOL 60 Compiler," Annual Review in Automatic Programming, Vol. 4, 1964, pp. 87-124.

16. Feldman, Jerome and David Gries. "Translator Writing Systems," CACM, Vol. 11, February, 1968, pp. 77-113.

17. Floyd, R. W. "A Descriptive Language for Symbol Manipulation," JACM, October, 1961, pp. 579-584.

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

19. Galler, B., and A. J. Perlis. " A Proposal for Definitions in ALGOL," CAMC, Vol. 10, April, 1967, pp. 204-219.

20. Gries, D.; M. Paul; and H. R. Niehle. "Some Techniques Used in the ALCOR Illinois 7090," CACM, Vol. 8, August, 1965, pp. 496-500.

21. Gries, David. "Use of Transition Matrices in Compiling," CACM, Vol. 11, January, 1968, pp. 26-34.

22. Gries, David. Compiler Construction for Digital Computers. New York: John Wiley & Sons, Inc., 1971.

23. Goos and Hartman, Editors. Lecture Notes in Computer Science. Compiler Construction, An Advanced Course. Berlin: Springer-Verlag, 1974.

24. Ichbiah, L. D., and S. P. Morse. "A Technique for Generating Almost Optimal Floyd-Evans Productions for Precedence Grammars," CACM, Vol. 13, August, 1970, pp. 501-508.

25. Knuth, D. E. "On the Translation of Languages from Left to Right," Inf. Contr., Vol. 8, October, 1965, pp. 607-639.

26. Korenjak, A. J. "A Practical Method for Construction LR(k) Processors," CACM, Vol. 12, November, 1969, pp. 613-623.

27. Lawson, Harold W., Jr., and David R. Doucette. "A Translation Machine with Automated Top-Down Parsing," Sigplan Notices, Vol. 11, February, 1976, pp. 20-29.

28. Lui, C. L.; G. D. Chang; and R. E. Marks. "The Design and Implementation of a Table-Driven Compiler System," SJCC, (AFIPS), 1967, pp. 691-697.

29. Lomet, David Bruce. "A Formalization of Transition Diagram Systems," JACM, Vol. 20, April, 1973, pp. 235-257.

30. McClure, R. M. "An Appraisal of Compiler Technology," Proc of SJCC, May, 1972, pp. 1-9.

31. McKeeman, W. M. An Approach to Computer Language Design. Technical Report CS48, Computer Science Department, Stanford University, Stanford, California, August, 1966.

32. McKeeman, W. M., et al. "The XPL Compiler Generator System," Proc FJCC, 1968, pp. 617-635.

33. Mickunas, M. D., and V. B. Schneider. "A Parser-Generating System for Construction Compressed Compilers," CACM, Vol. 16, November, 1973, pp. 669-675.

34. Nievergelt, J., et al. ACSES: The Automated Computer Science Education System at The University of Illinois. Report UIUCDCS-R-76-810, Department of Computer Science, University of Illinois at Urbana-Champaign, August, 1976.

35. O'Neil, J. T. "Meta Pi--An On-Line Interactive Compiler-Compiler," Proceedings, 1968, FJCC, pp. 201-218.

36. Richards, Martin. "BCPL: A Tool for Compiler Writing and System Programming," Proceedings, 1969, SJCC, pp. 557-566.

37. Samelson, K., and F. L. Bauer. "Sequential Formual Translation," CACM, Vol. 3, February, 1960, pp. 76-83.

38. Schneider, Victor. "A System for Designing Fast Programming Language Translators," Proceedings, 1969, SJCC, pp. 777-792.

39. Sherwood, Bruce Arne. The Tutor Language. University of Illinois at Urbana-Champaign, Urbana, Illinois, June, 1974.

40. Tindall, Michael Harry. An Interactive Table-Driven Parser System. Masters Thesis. University of Illinois, Urbana-Champaign, Urbana, Illinois, August, 1975.

41. Tindall, Michael Harry. An Interactive Compile-Time Diagnostic System. Report UIUCDCS-R-75-748. Ph.D. Dissertation. Department of Computer Science, University of Illinois, Urbana-Champaign, October, 1975.

42. Warshall, Stephen and Robert M. Shapiro. "General-Purpose Table-Driven Compiler," Proceedings, 1964, SJCC, pp. 59-65.

43. Wirth, N., and H. Weber. "EULER--A Generalization of ALGOL, and Its Formal Definition, Part I, Part II," CACM, Vol. 9, January, February, 1966, pp. 89-99.

# APPENDIX

# APPENDIX

## GLOSSARY

### BNF

The productions used in this paper are in the standard Backus-Naur form (BNF). $N_1$ ::= $T_1N_2T_2$ | $T_3N_3$ means the single nonterminal, $N_1$, can be rewritten as the string $T_1N_2T_2$ or the string $T_3N_3$.

### Bottom-Up Parsing

If parsing is done by starting with the string of input symbols and reducing these until only the distinguished symbol remains, then this procedure is called bottom-up parsing.

### Context-Free Grammar

A context-free grammar is a quadruple (T, N, S, P). Set T = $\{T_1, T_2, \ldots T_n\}$ is a finite set of symbols called terminals. Set N = $\{N_1, N_2, \ldots N_m\}$ is a finite set of symbols called nonterminals. These are selected so that $N \cap T = \emptyset$. S is an element of N called the distinguished symbol. P is a finite set of productions, each of the form $N_1$ ::= x in which the left part, $N_1$, is a nonterminal and the right part, x, is a string each element of which is a member of $(N \cup T)$.

### Derivation of

C is a derivation of a, and a produces c, written a ==> + c if there exists a sequence of direct productions

a ==> . . . ==> y ==> c.

If a ==> + c or a = c, then a ==> * c is written.

Distinguished Symbol

See definition of context-free grammar.

Handle

The handle of a sentential form is the leftmost simple phrase.

Language

The language of a grammar is the set of all sentences of that grammar.

Nonterminal

See context-free grammar.

Phrase

Given that the string caf is a sentential form, then a is a phrase of the sentential form if $S \Longrightarrow^{*} cN_1f$ and $N_1 \Longrightarrow^{+} a$.

Prime Phrase

The prime phrase of a sentential form is a phrase that contains at least one terminal but no prime phrases other than itself.

Productions

See context-free grammar.

Recursive Descent

The recursive descent technique is said to be used when the syntax analyzer has one recursive procedure for each nonterminal, and parsing is performed by calling these procedures to recognize goals (and subgoals) when necessary.

Sentential Form

If a string a is derivable from the distinguished symbol, S (i.e., S ==> * a), then a is a sentential form.

Sentence

If a sentential form contains only terminals, it is called a sentence.

Simple Phrase

If S ==> * $cN_1f$ and $N_1$ ::= a, then a is a simple phrase.

String

A sequence of symbols is referred to as a string and represented by lower case latin letters.

Terminals

See context-free grammar.

Top-Down Parsing

If the parser starts with the distinguished symbol and works until the original input string has been reconstructed, then it is performing top-down parsing.

# VITA

Jane (Tania) Goldberger Widler was born in New York City on August 16, 1950. She grew up and attended public school in Tenafly, New Jersey, graduating from Tenafly High School in June, 1968. The following September, she entered Goucher College, Baltimore, Maryland, and in May, 1972, she received a Bachelor of Arts degree in Mathematics.

From February, 1972, through April, 1975, Ms. Widler worked as a programmer, first in Baltimore, Maryland, and later in Sarasota, Florida. In the fall of 1975 she accepted a teaching assistantship at The University of Tennessee, Knoxville, Tennessee, and began study toward a Masters Degree.

Ms. Widler is married and plans to settle in Redwood Valley, California, upon the completion of the requirements for a Master of Science Degree in Computer Science.