

TRAINING AIDS FOR TRANSLATOR DESIGN

by

JAMES R. MEYER

B.S., Benedictine College, KS, 1971

A MASTER'S REPORT

submitted in partial fulfillment of the

requirements for the degree

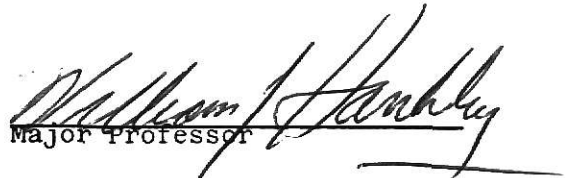
MASTER OF SCIENCE

Department of Computer Science

KANSAS STATE UNIVERSITY
Manhattan, Kansas

1977

Approved by:


Major Professor

LD
2068
R4
1977
M49
C.2

Document

TABLE OF CONTENTS

	<u>Page No.</u>
CHAPTER 1 - Introduction	1
CHAPTER 2 - Lexical Scanning	3
2.1 General	3
2.2 Separation and Identification of Input	3
2.3 Producing Tokens	4
2.4 Producing Symbol Tables	10
2.5 Example of a Scanned Procedure	13
2.6 Implementing Algorithm for Scanning	16
2.7 Error Detection	24
CHAPTER 3 - Parsing	25
3.1 General	25
3.2 Syntax Analysis	25
3.3 Top-Down Parsing	27
3.4 Bottom-Up Parsing	51
3.5 Code Generation	63
REFERENCE NOTES	77

FIGURES

	<u>Page No.</u>
1. Lexical Scanning	4
2. Data in Token Fields	6
3. Reserved Word Table	7
4. Operator Table	8
5. Delimiter Table	9
6. Developing Tokens for Source Code Line	10
7. Symbol Table	11
8. Symbol Table After Scanning Phase	12
9. A Scanned Procedure	14
10. Scanning Algorithm	16
11. Recursive Decent Parser	27
12. Rules for Recursive Decent Parser	32
13. Explicit Stack Parsing Algorithm	33
14. Rules for Explicit Stack Parser	37
15. Parsing an Input String from Grammar (A) by an Explicit Stack Parser	38
16. Example of Explicit Stack Parsing	40
17. Grammar for CS-700 Interpreter	43
18. Rules for CS-700 Production	44
19. Example of Parsing in CS-700 Interpreter	47
20. Simple Grammar for a Bottom-Up Parser	53
21. Rules for Parsing Grammar (E)	54
22. Bottom-Up Parse Using Grammar (E)	55
23. Grammar for CS-700 Bottom-Up Parser	57
24. Rules for CS-700 Interpreter Bottom-Up Parser	58

FIGURES (continued)

	<u>Page No.</u>
25. Bottom-Up Parse in CS-700 Interpreter	59
26. Algorithm for Bottom-Up Parser in CS-700 Interpreter	61
27. Semantic Action During Parsing	65
28. Code Generation During Parsing	66
29. Example of Parsing in CS-700 Interpreter	70
30. Code Generated During Bottom-Up Parsing	76
31. Example of Code Generation During Bottom-Up Parsing	77

CHAPTER 1

Introduction

The Interpreter Design Course, CS 286-700, is a first graduate level course in the concepts and design of compilers for computer systems. The course is designed to study the concepts, algorithms, and data structures of interpreters and compilers. The primary reference for the course is a text on compiler design. However, the text is used only as a supplement to the classroom presentations in the teaching of general concepts. The classroom presentations concentrate on teaching the course objectives using a student-developed interpreter as a model.

The model, the CS-700 Interpreter, was developed by students in the summer of 1975. The draft code covers the basic components of the interpreter, but requires further testing, debugging, and optimizing. In addition, the documentation is in varying degrees of perfection, and requires expansion and refinement. These discrepancies become individual and group projects for the students during the course.

The purpose of this project is to produce algorithms and training aids to be used in the classroom and as homework problems to support the teaching objectives. Specifically, they aid in teaching lexical scanning, top-down parsing, bottom-up parsing, and code generation. Another master's report, Models for Translator Design, Kansas State University, by Miles Tipton Clements Jr. concentrates on the execution of the CS-700 Interpreter. The abstract tutorial cases are designed to teach the concepts of a stack oriented sequential processor. The

narrative preceding the algorithms and training aids is designed to provide an introductory framework only, and no attempt is made to explain theories presented in the text. The specific implementation cases are designed to illustrate the concepts and prepare the students for the CS-700 Interpreter Projects. These specific implementation cases are also abstracts of the actual implementation. For example, the abstract may refer to an operator as "IPLUS", whereas in the real implementation an operator would have a numeric value. However, this level of abstraction lends clarity to details that are developing a concept.

CHAPTER 2

Lexical Scanning

2.1 General. Lexical scanning is logically the first function accomplished by a translator. In this phase the lines of source code are scanned and source code atoms are separated and identified. After an atom is separated and identified, a token is produced to represent the atom. In addition to producing tokens, some translators accomplish other actions during the lexical scanning phase. These actions may include semantic analysis and symbol table production. A symbol table performs a dictionary function and further describes the identifiers used in a source program. In the CS-700 Interpreter a symbol table is produced during lexical scanning to further describe identifiers in the source program.

2.2 Separation and Identification of Input. The scanning algorithm in Figure 1 demonstrates a typical scanner logic for separating and identifying source code atoms. This scanner is much like the scanner in the CS-700 Interpreter. In this algorithm an atom's type is determined by its first character. An atom may be typed as an identifier, an integer, a real number, an operator, a delimiter, a label, or a string. The right end of a variable length atom is determined when the scanner detects a blank, delimiter, or operator. If an atom is determined to be an identifier, then another algorithm matches it with a table of reserved words to make this differentiation.

Lexical Scanning

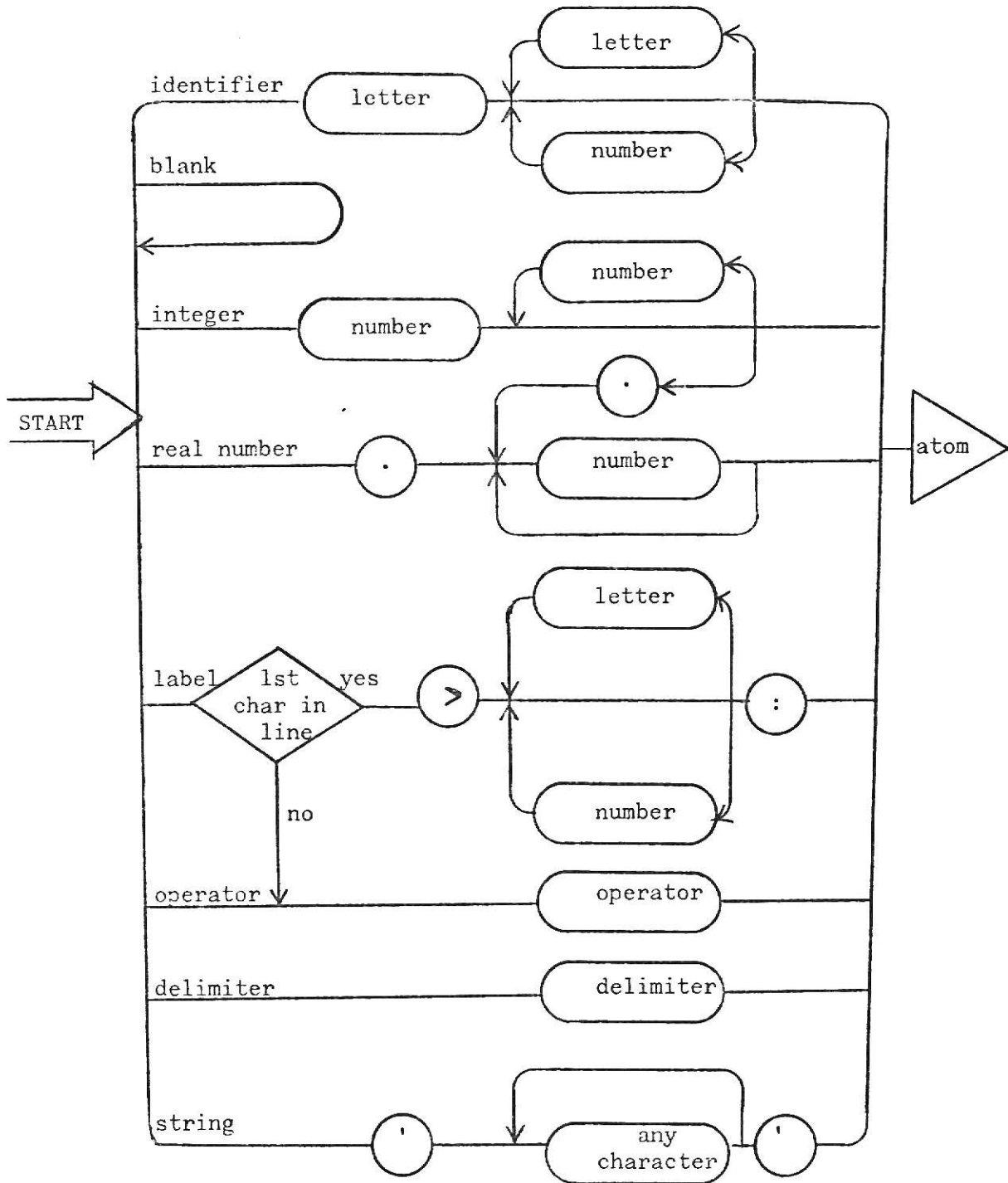
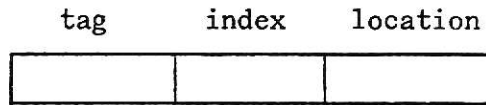


Figure 1

2.3 Producing Tokens. After the scanner has separated and identified an atom, it produces a token to represent the atom. A token in the CS-700 Interpreter is composed of three fixed length fields; the tag, the index, and the location field.



The tag identifies the type of token. In the CS-700 Interpreter there are nine types allowed; integers (INT), real numbers (REAL), identifiers (ID), labels (LAB), strings (STR), reserved words (RES), operators (OPR), delimiters (DEL), and start-of-line symbols (SOL). The start-of-line symbol is actually not produced from an atom, but is generated by the scanner before scanning a line of source code. The start-of-line symbol acts as a pointer to the source code line and is used by the interpreter as a reference point in the source code. This reference point is necessary to later identify incorrect source code and to identify the beginning position of strings.

The index is used to supplement information about the token and its contents depends on the type of tag. The information in Figure 2 presents the contents of the index for the various types of tags.

The location is used to point to the relative position of the first character of an atom within a line of code. The one exception to this rule is for start-of-line symbols. Since the start-of-line symbol is not an element of the source code line, this field is used to depict the number of tokens in the source code line.

Data in Token Fields

Type of Token	Contents of		
	Tag	Index	location
Integer	INT	value of integer	beginning location in source code line
real number	REAL	value of real number	beginning location in source code line
identifier	ID	index number of symbol table	beginning location in source code line
label	LAB	index number of symbol table	beginning location in source code line
string	STR	number of characters in string	beginning location in source code line
reserved word	RES	number representing reserved word	beginning location in source code line
operator	OPR	word representing * type of operator	location in source code line
delimiter	DEL	word representing * type of delimiter	location in source code line
start-of-line	SOL	line number count	number of tokens in source code line

* For abstract purposes this is a word, but in the real implementation this is a number.

Figure 2

As shown in Figure 2, reserved words, operators, and delimiters are represented by either a number or a word. The table in Figure 3 depicts the reserved words in the CS-700 Interpreter and the rules for the algorithm that develops the representing number.

Reserved Word Table

Column							
Row	1	2	3	4	5	6	7
1	DO	END	CASE	BEGIN	ACCESS	ENDPROC	ENDWRITE
2	IF	OUT	ELSE	FALSE	EXPORT		EXTERNAL
3	IN		EXIT	WHILE	GLOBAL		
4	FI		GOTO	WRITE	RETURN		
5			PROC				
6			READ				
7			THEN				
8			TRUE				
9			QUIT				
10			CALL				

- Column - Reserved words were placed in columns based on the total number of characters in the word. Limitation was placed on the language in that a reserved word may not exceed eight but must have a minimum of two characters.
- Row - The total number of rows of the table is dependent on the largest number of reserved words with the same number of characters.
- Index - The index value of a reserved word is determined by multiplying the column number by a quantity and adding the column number.

Figure 3

The table in Figure 4 depicts the operators in the CS-700 Interpreter and the representing word that is inserted in the index of the token. (In the real implementation this is a number.)

Operator Table

Operator	Replacement
←	IASG
+	IPLUS
-	IMINUS
*	IMULT
/	IDIV
=	IEQ
>	IGT
<	ILT
≠	INE
¬	INOT
≤	ILE
≥	IGE
TRUE	ITRUE
FALSE	IFALSE

Figure 4

The table in Figure 5 depicts the delimiters in the CS-700 Interpreter and the representing word that is inserted in the index of the token.

Delimiter Table

Delimiter	Replacement
(LPAREN
)	RPAREN
;	SCOLON
:	COLON
,	COMMA
[LBRAK
]	RBRAK

Figure 5

The following example illustrates the use of all three fields in developing tokens for a source code line.

<u>source code line-</u>			DO WHILE A = 0;
<u>Tokens</u>			
tag	index	location	<u>Explanation</u>
SOL	1	6	Start-of-Line symbol for first line
RES	11	1	DO is reserved word number 11
RES	43	4	WHILE is reserved word number 43
ID	1	10	First identifier in symbol table *
OPR	IEQ	12	IEQ replaces the equal sign
INT	0	14	Value is inserted into the index
DEL	SCOLON	15	SCOLON replaces the semicolon

* symbol tables are further explained in the next section

Figure 6

2.4 Symbol Tables. Since computer programs make frequent use of symbolic identifiers, the translator must know more about identifiers than other atoms. One way to do this is to store the additional information in a symbol table which acts as a dictionary of the symbols used in the program. The symbol table in Figure 7 is used by the CS-700 Interpreter to store additional information on identifiers used in a program.

Symbol Table

index number	number of characters	literal name	scope	type	value or address

Figure 7

The index number is a sequential number assigned to the entry associated with the identifier. This index number is unique and specifies the relative location of the identifier in the symbol table. (This is also the value in the index field of a token representing the identifier.)

The number of characters field contains the value for the number of characters in the identifier.

The scope field explains how the variable is used during the program's execution. The following are the types of scope information in the CS-700 Interpreter:

- IN - This is a value that can be passed to the program as an argument in the CALL statement.
- OUT - This is a value that can be passed back as a result of this program.
- LOC - This is a value that is generated and used only within this program. This is the default if scope is not specified.
- EXT - Not used, but intended to be the same as the EXTERNAL scope in Algol.
- GLOB- This is a value that is common to all programs.
- VARY- This is a value that can be passed to the program as an argument in the CALL statement and can be changed and returned.

The type field identifies the mode of the value; real, integer, etc.

The value or address field has three uses. If the identifier represents a scalar value, then this field contains the value. If the identifier represents a string, then this field contains an address in the source code of the beginning of the string. If the identifier represents a label, then this field contains the address of the associated instruction.

During the scanning phase, however, most of the information needed for the symbol table is not available. Some information will be added during the parsing, and some will be added or changed during the execution of the program. In the CS-700 Interpreter the number of characters and literal name are completed during the scanning phase. The scope and address (for labels and strings) are completed during the parsing phase. The type and value are completed during the execution phase.

The following is an example of a completed symbol table after the scanning phase.

Symbol Table After Scanning Phase

source code line- PAY = GROSS - TAXES;

Symbol Table

index number	number of characters	literal name	scope	type	value or address
1	3	PAY			
2	5	GROSS			
3	5	TAXES			

Figure 8

2.5 Example of a Scanned Procedure. The preceding paragraphs have detailed the basic functions of the scanner in the CS-700 Interpreter. The example in Figure 9 combines these functions to illustrate the scanning of a simple procedure.

```
procedure-      PROC NEWBALANCE;  
                  NEWBAL = OLDBAL - PAYMENT;      .  
                  INTEREST = NEWBAL * .015;        -  
                  ACCTBAL = NEWBAL + INTEREST;  
                  RETURN;END;
```

A Scanned Procedure

<u>Source Statements</u>	Tokens		
	Tag	Index	Location
PROC NEWBALANCE;	SOL	1	3
	RES	35	1
	ID	1	6
	DEL	SCOLON	16
NEWBAL = OLDBAL - PAYMENT;	SOL	2	6
	ID	2	1
	OPR	IASG	8
	ID	3	10
	OPR	IMINUS	17
	ID	4	19
	DEL	SCOLON	26
INTEREST = NEWBAL * .015;	SOL	3	6
	ID	5	1
	OPR	IASG	10
	ID	2	12
	OPR	IMULT	19
	REAL	.015	21
	DEL	SCOLON	25
ACCTBAL = NEWBAL + INTEREST;	SOL	4	6
	ID	6	1
	OPR	IASG	9
	ID	2	11
	OPR	IPLUS	18

Figure 9

Source Statements

RETURN;END;

Tokens

Tag	Index	Location
ID	5	20
DEL	SCOLON	28
SOL	5	4
RES	54	1
DEL	SCOLON	7
RES	21	8
DEL	SCOLON	11

Symbol Table

index number	number of characters	literal name	scope	type	value or address
1	10	NEWBALANCE			
2	6	NEWBAL			
3	6	OLDBAL			
4	7	PAYMENT			
5	8	INTEREST			
6	8	ACCTBAL			

(Figure 9 continued)

2.6 Implementing Algorithm for Scanning. The scanning phase of a translator is the easiest phase to implement. Figure 10 is a structured algorithm that would implement the pictorial algorithm in Figure 1.

Scanning Algorithm

PROCEDURE SCAN

COMMENT: THIS PROCEDURE SCANS INPUT STATEMENTS AND PRODUCES
TOKENS AND A SYMBOL TABLE

SYMBOL-TABLE.INDEX = 0

LINE-COUNT = 0

DO WHILE MORE-TO-BE-READ = .TRUE

BEGIN

READ INPUT

LINE-COUNT = LINE-COUNT + 1

TOKEN-COUNT = 0

IF LAST-INPUT THEN CALL PARSER

ELSE BEGIN

COMMENT: I IS THE LEFT END OF AN ATOM, J CLOCKS THROUGH
TO END OF AN ATOM

J = 1

I = 1

COMMENT: PUSH START-OF-LINE SYMBOL ON TOKEN STACK

TAG = "SOL"

INDEX = "Ø"

LOCATION = "Ø"

PUSH TOKEN ON TOKEN-STACK

NO = ADDRESS-OF-TOKEN

DO WHILE J LESS-THAN-OR-EQUAL-TO 80

BEGIN

COMMENT: WHEN I = J A NEW ATOM IS BEING SCANNED

IF I = J THEN CALL IDENTIFY-ATOM

ELSE NULL

CALL LOOK-FOR-ATOM-END

END-DO

Figure 10


```

        COMMENT:  COMPLETE THE INDEX AND LOCATION FIELDS OF SOL TAKEN
        MOVE LINE-COUNT TO TOKEN-STACK(NO).INDEX
        MOVE TOKEN-COUNT TO TOKEN-STACK(NO).LOCATION
    END-ELSE
END-DO
END-PROC

PROCEDURE IDENTIFY-ATOM
COMMENT:  AN ATOM IS IDENTIFIED BY ITS FIRST CHARACTER, HOWEVER THE
          IDENTIFICATION CAN CHANGE FOR IDENTIFIERS THAT PROVE TO BE
          RESERVED WORDS AND INTEGERS THAT PROVE TO BE REAL NUMBERS
CASE INPUT(J) = " "
    ATOM = BLANK
CASE INPUT(J) = "A THRU Z"
    ATOM = IDENTIFIER
    COMMENT:  "COUNT" IS USED TO COUNT CHARACTERS IN IDENTIFIERS AND
          LABELS
    COUNT = 0
CASE INPUT(J) = 0 THRU 9"
    ATOM = NUMBER
CASE INPUT(J) = ">" AND I = 1
    ATOM = LABEL
    COUNT = 0
CASE INPUT(J) = ""
    ATOM = STRING
    COMMENT:  J IS SET TO FIRST CHARACTER IN STRING
    J = J + 1
CASE INPUT(J) = "+ OR - OR * OR /"
    ATOM = OPERATOR
CASE INPUT(J) = "( OR ) OR ;"
    ATOM = DELIMITER
END-CASE
END-PROC

```

Figure 10 (continued)

PROCEDURE LOOK-FOR-ATOM-END

COMMENT: THE END OF AN ATOM IS LOCATED WHEN A BLANK OR DELIMITER IS
IDENTIFIED, A TOKEN IS CREATED, AND FOR IDENTIFIERS AND LABELS
AN ENTRY IS MADE IN THE SYMBOL TABLE

CASE ATOM = IDENTIFIER

BEGIN

IF INPUT(J) = "% OR (OR) OR ;" THEN BEGIN

COMPARE ATOM TO RESERVED-WORD-TABLE.VALUES

IF COMPARE = .TRUE THEN BEGIN

TAG = "RES"

INDEX = RESERVE-WORD-TABLE.NO

LOCATION = I

ELSE BEGIN

COMMENT: SEE IF IDENTIFIER IS ALREADY IN SYMBOL TABLE

COMPARE INPUT(I THRU J) TO SYMBOL-TABLE.VALUES

IF COMPARE = .TRUE THEN INDEX = SYMBOL-TABLE.NO

ELSE BEGIN

SYMBOL-TABLE.INDEX = SYMBOL-TABLE.INDEX + 1

INDEX = SYMBOL-TABLE.INDEX

MOVE INPUT(I THRU J) TO SYMBOL-TABLE(INDEX).NAME

MOVE COUNT TO SYMBOL-TABLE(INDEX).NO-CHAR

TAG = "ID"

LOCATION = I

PUSH TOKEN ON TOKEN-STACK

TOKEN-COUNT = TOKEN-COUNT + 1

I = J

ELSE BEGIN

J = J + 1

COUNT = COUNT + 1

END-CASE

Figure 10 (continued)

```

CASE ATOM = NUMBER
  BEGIN
    IF INPUT(J) = "Ø OR ( OR ) OR ;" THEN BEGIN
      TAG = "INT"
      INDEX = INPUT(I THRU J)
      LOCATION = I
      PUSH TOKEN ON TOKEN-STACK
      TOKEN-COUNT = TOKEN-COUNT + 1
      I = J
    ELSE BEGIN
      IF INPUT(J) = "." THEN ATOM = REAL-NUMBER
      ELSE NULL
      J = J + 1
    END-CASE
  END-CASE

CASE ATOM = REAL-NUMBER
  BEGIN
    IF INPUT(J) = "Ø OR ( OR ) OR ;" THEN BEGIN
      TAG = "REAL"
      INDEX = INPUT(I THRU J)
      LOCATION = I
      PUSH TOKEN ON TOKEN-STACK
      TOKEN-COUNT = TOKEN-COUNT + 1
      I = J
    ELSE J = J + 1
  END-CASE

CASE ATOM = STRING
  BEGIN
    COMMENT: A SINGLE QUOTE MARK ENDS A STRING
    IF INPUT(J) = "!" THEN BEGIN
      TAG = "STR"
      INDEX = J - (I + 1)
    
```

Figure 10 (continued)

```

        LOCATION = I + 1
        PUSH TOKEN ON TOKEN-STACK
        TOKEN-COUNT = TOKEN-COUNT + 1
        J = J + 1
        I = J
        ELSE J = J + 1
END-CASE

CASE ATOM = LABEL
    BEGIN
        COMMENT: LABELS ARE TERMINATED BY A COLON
        IF INPUT(J) = ":" THEN BEGIN
            COMMENT: SEE IF LABEL IS ALREADY IN SYMBOL TABLE
            COMPARE INPUT(I THRU J) TO SYMBOL-TABLE.VALUES
            IF COMPARE = .TRUE THEN INDEX = SYMBOL-TABLE.NO
            ELSE BEGIN
                COMMENT: LABEL IS NOT IN SYMBOL TABLE
                SYMBOL-TABLE.INDEX = .SYMBOL-TABLE.INDEX + 1
                INDEX = SYMBOL-TABLE.INDEX
                MOVE INPUT(I+1 THRU J-1) TO SYMBOL-TABLE(INDEX).NAME
                MOVE COUNT TO SYMBOL-TABLE(INDEX).NO-CHAR
            END
            TAG = "LAB"
            LOCATION = I + 1
            PUSH TOKEN ON TOKEN-STACK
            TOKEN-COUNT = TOKEN-COUNT + 1
            J = J + 1
            I = J
        ELSE BEGIN
            J = J + 1
            COUNT = COUNT + 1
        END
    END-CASE

```

Figure 10 (continued)

```

CASE ATOM = BLANK
    BEGIN
        COMMENT:  BLANKS ARE NOT STORED AS TOKENS BUT ARE SKIPPED
        IF INPUT(J) = " " THEN J = J + 1
        ELSE I = J
    END-CASE

CASE ATOM = DELIMITER
    BEGIN
        TAG = "DEL"
        CASE INPUT(J) = "("
            INDEX = "LPAREN"
        CASE INPUT(J) = ")"
            INDEX = "RPAREN"
        CASE INPUT(J) = ";"
            INDEX = "SCOLON"
        CASE INPUT(J) = ":"
            INDEX = "COLON"
        CASE INPUT(J) = ","
            INDEX = "COMMA"
        CASE INPUT(J) = "["
            INDEX = "LBRAK"
        CASE INPUT(J) = "]"
            INDEX = "RBRAK"
        LOCATION = I
        PUSH TOKEN ON TOKEN-STACK
        TOKEN-COUNT = TOKEN-COUNT + 1
        J = J + 1
        I = J
    END-CASE

```

Figure 10 (continued)

```

CASE ATOM = OPERATOR
  BEGIN
    TAG = "OPR"
    CASE INPUT(J) = "<"
      INDEX = "IASG"
    CASE INPUT(J) = "+"
      INDEX = "IPLUS"
    CASE INPUT(J) = "-"
      INDEX = "IMINUS"
    CASE INPUT(J) = "*"
      INDEX = "IMULT"
    CASE INPUT(J) = "/"
      INDEX = "IDIV"
    CASE INPUT(J) = "="
      INDEX = "IEQ"
    CASE INPUT(J) = ">"
      INDEX = "IGT"
    CASE INPUT(J) = "<"
      INDEX = "ILT"
    CASE INPUT(J) = "≠"
      INDEX = "INE"
    CASE INPUT(J) = "⊥"
      INDEX = "INOT"
    CASE INPUT(J) = "≤"
      INDEX = "ILE"
    CASE INPUT(J) = "≥"
      INDEX = "IGE"
    CASE INPUT(J) = "TRUE"
      INDEX = "ITRUE"
    CASE INPUT(J) = "FALSE"
      INDEX = "IFALSE"
    LOCATION = I
    PUSH TOKEN ON TOKEN-STACK

```

Figure 10 (continued)

```
TOKEN-COUNT = TOKEN-COUNT + 1  
J = J + 1  
I = J
```

END-CASE

END-PROC

Figure 10 (continued)

2.7 Error Detection. Errors in the source code can be detected by the translator in the scanning phase, parsing phase, and the execution phase. During the scanning phase the translator can detect the following errors:

- illegal characters (e.g. unprintable characters in a line)

- faulty strings (e.g. missing quote marks)

- number overflow (e.g. numbers longer than the space allowed in the index field)

- symbol overflow (e.g. identifiers longer than the space allowed in the symbol table)

CHAPTER 3

Parsing

3.1 General. Once the tokens and the symbol table have been generated by the scanner, the parsing phase is ready to begin. There are three functions that can be accomplished in the parsing phase. The first function is syntax analysis or checking the input to insure it conforms with the grammar of the language. The second function is semantic analysis or checking the operators and operands against the symbol table to insure the administrative rules of the grammar are followed. A grammar that requires checking against the symbol table is called "context sensitive". A grammar that does not require checking against the symbol table is called "context free". The third function of the parser is to generate code.

In this chapter the functions are segmented and discussed separately. However, in an actual parser the syntax analysis, the semantic analysis, and the code generation functions are integrated throughout the parsing process.

3.2 Syntax Analysis. Syntax analysis is the function of checking the input to insure it conforms with the grammar of the language. To accomplish this the parser must have a definition of the grammar for the input language in a machine readable form. It then reads the tokens created in the scanning phase and compares the tokens with this machine readable form of the grammar.

In general, there are two types of parsers, top-down parsers and bottom-up parsers. Top-down parsers that can parse a recursive language are further categorized as a recursive parser or an explicit stack parser.

These two categories are further classified by the number of tokens the parser must look ahead before it can recognize a production. For example, if the parser can determine the production by looking at one token, then the parser is classified as ll1 (look ahead, left-to-right parse, one token) or an lr1 (look ahead, right-to-left parse, one token). If the parser must look ahead more than one token then it is classified as an llk or lrk parser where k is the number of tokens it must look ahead. Generally, top-down parsers are left-to-right parsers. Another distinguishing feature of a parser is the number of passes through the tokens the parser must make to generate the code. Some parsers accomplish this in one pass, others require two or more passes.

Bottom-up parsers are categorized into three groups; simple precedence parsers, operator precedence parsers, and other lrk parsers. In a simple precedence parser every token is parsed. In an operator precedence parser only the operators are important in the parse, and therefore the operands become "invisible" in the parse. This concept of invisibility will be further discussed in the examples. Other bottom-up parsers use combinations of top-down and bottom-up parsing.

In the CS-700 Interpreter all the code is generated from a single pass, and all productions except expressions are parsed using a top-down parser. The top-down parser uses an explicit stack and for the most part is an ll1 parser. (Two productions require a look ahead of two tokens.) The bottom-up parser is an operator parser that parses expressions from right-to-left. Therefore, most of the examples will concentrate on these two types of parsers.

3.3 Top-Down Parsing.

A recursive decent parser must use a recursive language to perform syntax analysis. Therefore, languages like COBOL or FORTRAN cannot be used to implement a recursive parser. The recursive parser is the easier to implement because only one procedure needs to be written for each non-terminal in the grammar. An example of an algorithm for a recursive decent parser for an l11 grammar is illustrated in Figure 11.

Recursive Decent Parser

Grammar (A)

l11

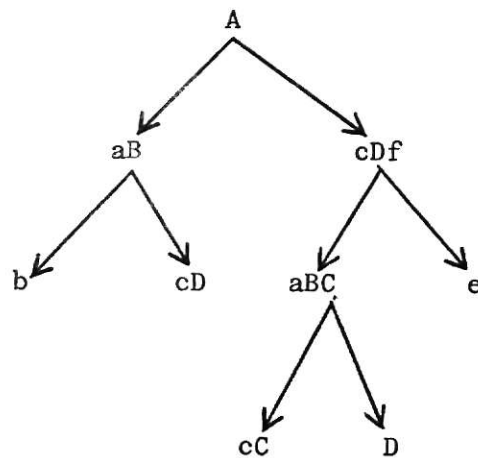


Figure 11

```

PROCEDURE A(RETURN-CODE)
CASE TOKEN = "a"
    BEGIN
        MATCH "a"
        CALL B(RETURN-CODE)
        RETURN
    END-CASE

CASE TOKEN = "c"
    BEGIN
        MATCH "c"
        CALL D(RETURN-CODE)
        IF RETURN-CODE = "BAD" THEN RETURN
        IF TOKEN = "f" THEN MATCH "f"
        ELSE RETURN-CODE = "BAD"
    END
ELSE
    BEGIN
        PRINT ERROR
        RETURN-CODE = "BAD"
        RETURN
    END-CASE

END-PROC

```

Figure 11 (continued)

```

PROCEDURE B(RETURN-CODE)
IF RETURN-CODE = "BAD" THEN RETURN
CASE TOKEN = "b"
    BEGIN
        MATCH "b"
        RETURN
    END

CASE TOKEN = "c"
    BEGIN
        MATCH "c"
        CALL D(RETURN-CODE)
        RETURN
    END

ELSE
    BEGIN
        PRINT ERROR
        RETURN-CODE = "BAD"
        RETURN
    END

END-CASE

END-PROC

```

Figure 11 (continued)

```

PROCEDURE C(RETURN-CODE)
IF RETURN-CODE = "BAD" THEN RETURN

CASE TOKEN = "c"
  BEGIN
    MATCH "c"
    CALL C(RETURN-CODE)
    RETURN
  END

CASE TOKEN = "a"
  BEGIN
    CALL D(RETURN-CODE)
    RETURN
  END

CASE TOKEN = "e"
  BEGIN
    CALL D(RETURN-CODE)
    RETURN
  END

ELSE
  BEGIN
    PRINT ERROR
    RETURN-CODE = "BAD"
    RETURN
  END

END-CASE

END-PROC

```

Figure 11 (continued)

```

PROCEDURE D(RETURN-CODE)
IF RETURN-CODE = "BAD" THEN RETURN

CASE TOKEN = "a"
  BEGIN
    MATCH "a"
    CALL B(RETURN-CODE)
    IF RETURN-CODE = "BAD" THEN RETURN
    CALL C(RETURN-CODE)
    RETURN
  END

CASE TOKEN = "e"
  BEGIN
    MATCH "e"
    RETURN
  END

ELSE
  BEGIN
    PRINT ERROR
    RETURN-CODE = "BAD"
    RETURN
  END

END-CASE

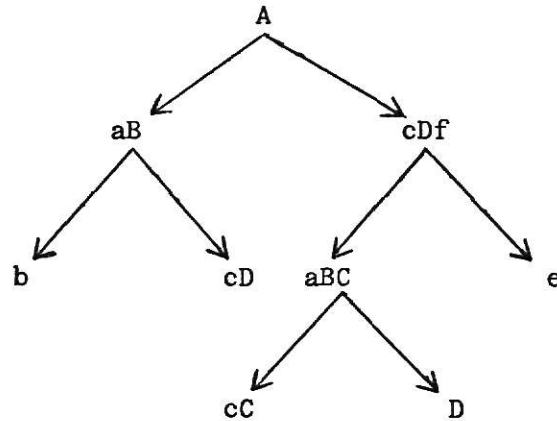
END-PROC

```

Figure 11 (continued)

The logic for this recursive decent parser can be depicted in a set of rules in a table. The rules for Grammar (A) are shown in Figure 12.

Grammar (A)



Rules for Recursive Decent Parser

Procedure	Tokens				
	a	b	c	e	f
A	Match "a" CALL B	Error	Match "c" CALL D Match "f"	Error	Error
B	Error	Match "b"	Match "c" CALL D	Error	Error
C	CALL D	Error	Match "c" CALL C	CALL D	Error
D	Match "a" CALL B CALL C	Error	Error	Match "e"	Error

Figure 12

The second method of constructing a parser uses an explicit stack. In this method the grammar in a machine readable form is pushed onto a stack as needed, and then matched to the input token stream. To accomplish this the procedures are more involved and complex, but this method can be implemented with a non-recursive language such as COBOL or FORTRAN. The CS-700 Interpreter is implemented in FORTRAN and uses an explicit stack parser. An example of an explicit stack parsing algorithm for Grammar (A) is depicted in Figure 13.

Explicit Stack Parsing Algorithm

PROCEDURE PARSE-USING-EXPLICIT-STACK

PUSH "A" ON STACK

TOKEN-POINTER = 1

DO UNTIL DONE

BEGIN

IF BOT-OF-STACK AND TOKEN = END-OF-LINE THEN DONE

IF NOT BOT-OF-STACK AND TOKEN = END-OF-LINE THEN ERROR

IF BOT-OF-STACK AND TOKEN NOT = END-OF-LINE THEN ERROR

IF STACK.TOP = TOKEN THEN BEGIN

POP STACK

TOKEN-POINTER = TOKEN-POINTER + 1

ELSE

CASE STACK.TOP = "A"

BEGIN

CASE TOKEN = "a"

POP STACK

PUSH "aB" ON STACK

Figure 13

```

    CASE TOKEN = "c"
        POP STACK
        PUSH "cDf" ON STACK
    ELSE ERROR
END-CASE

CASE STACK.TOP = "B"
    BEGIN
        CASE TOKEN = "b"
            POP STACK
            PUSH "b" ON STACK
        CASE TOKEN = "c"
            POP STACK
            PUSH "cD" ON STACK
        ELSE ERROR
    END-CASE

CASE STACK.TOP = "C"
    BEGIN
        CASE TOKEN = "a"
            POP STACK
            PUSH "aBC" ON STACK
        CASE TOKEN = "c"
            POP STACK
            PUSH "cC" ON STACK
        CASE TOKEN = "e"
            POP STACK
            PUSH "D" ON STACK
        ELSE ERROR
    END-CASE

```

Figure 13 (continued)

```

CASE STACK.TOP = "D"
    BEGIN
        CASE TOKEN = "a"
            POP STACK
            PUSH "aBC" ON STACK
        CASE TOKEN = "e"
            POP STACK
            PUSH "e" ON STACK
        ELSE ERROR
    END-CASE

END-CASE

END-DO

END-PROC

```

Figure 13 (continued)

The CS-700 Interpreter uses two algorithms to parse statements. The top-down algorithm begins the parse, but once an expression is detected it calls the bottom-up parser. The bottom-up parser parses the expression and returns to the top-down parser. Top-down parsing will be discussed first, and bottom-up parsing will be discussed in the next section. The following are general rules used for top-down parsing.

1. Start: Push the start symbol on the explicit stack and set pointer to first token.
2. Loop:
 - a. If bottom-of-stack and end-of-line, then done.
 - b. If not bottom-of-stack and end-of-line, then error.
 - c. If bottom-of-stack and not end-of-line, then error.
 - d. If term on stack matches token, then pop stack, move pointer, and continue to loop.
 - e. If non-terminal on stack, then use parse table to choose which production to use, pop the non-terminal, and push the selected production on the stack with the left edge up.

example-

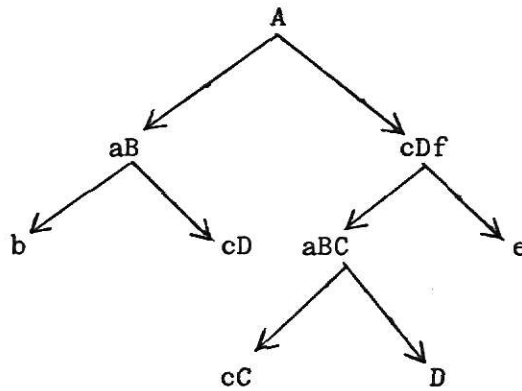
selected production- cDf

stack-

c
D
F

The logic for this explicit stack parser can be depicted in a set of rules in a table. The rules for Grammar (A) are shown in Figure 14.

Grammar (A)
111



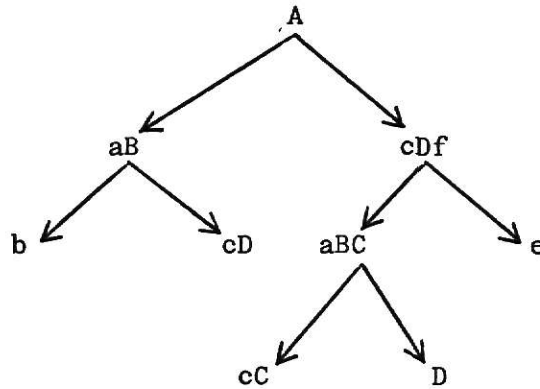
Rules for Explicit Stack Parser

Procedure	Tokens				
	a	b	c	e	f
A	Pop "A" Push "aB"	Error	Pop "A" Push "cDf"	Error	Error
B	Error	Pop "B" Push "b"	Pop "B" Push "cD"	Error	Error
C	Pop "C" Push "D"	Error	Pop "C" Push "cC"	Pop "C" Push "D"	Error Error
D	Pop "D" Push "aBC"	Error	Error	Pop "D" Push "e"	Error

Figure 14

The example in Figure 15 illustrates the parsing of an input string from Grammar (A) by an explicit stack parser.

Grammar (A)
111



Input string

a	c	a	b	e
---	---	---	---	---

<u>Step</u>	<u>Token</u>	<u>Execution</u>	<u>Explicit Stack</u>				
1		PUSH "A" TOKEN-POINTER = 1	<table><tr><td>A</td></tr></table>	A			
A							
2	a	(STACK.TOP = A & TOKEN = a) POP STACK PUSH "aB"	<table><tr><td>a</td></tr><tr><td>B</td></tr><tr><td>A</td></tr></table>	a	B	A	
a							
B							
A							
3	a	(STACK.TOP = TOKEN) POP STACK MOVE TOKEN-POINTER	<table><tr><td>a</td></tr><tr><td>B</td></tr></table>	a	B		
a							
B							
4	c	(STACK.TOP = B & TOKEN = c) POP STACK PUSH "cD"	<table><tr><td>c</td></tr><tr><td>D</td></tr><tr><td>B</td></tr></table>	c	D	B	
c							
D							
B							
5	c	(STACK.TOP = TOKEN) POP STACK MOVE TOKEN-POINTER	<table><tr><td>c</td></tr><tr><td>D</td></tr></table>	c	D		
c							
D							
6	a	(STACK.TOP = D & TOKEN = a) POP STACK PUSH "aBC"	<table><tr><td>a</td></tr><tr><td>B</td></tr><tr><td>C</td></tr><tr><td>D</td></tr></table>	a	B	C	D
a							
B							
C							
D							

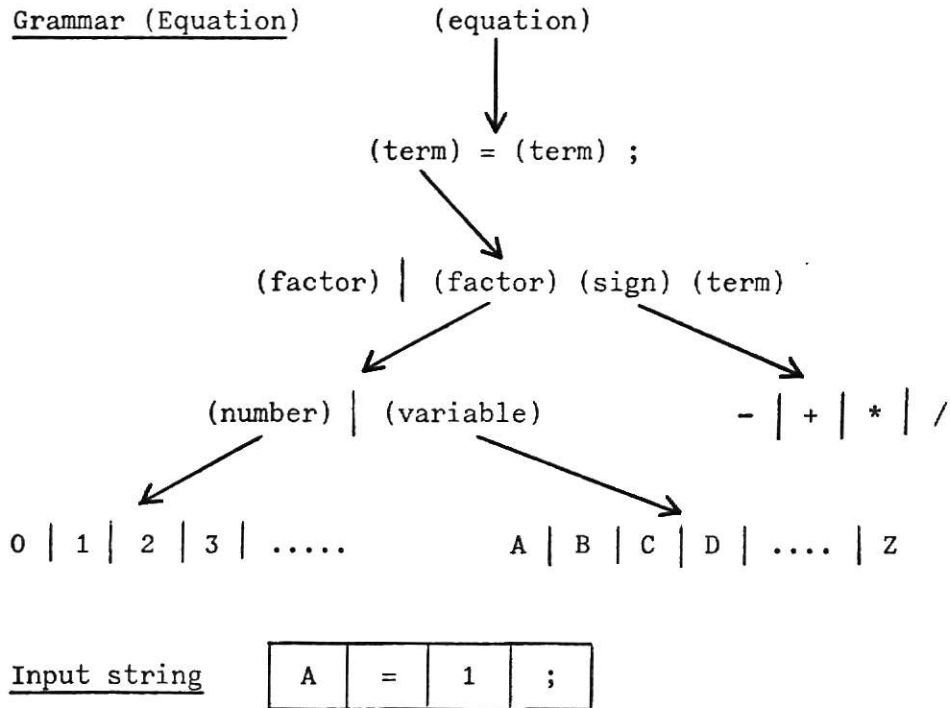
Figure 15

<u>Step</u>	<u>Token</u>	<u>Execution</u>	<u>Explicit Stack</u>
7	a	(STACK.TOP = TOKEN) POP STACK MOVE TOKEN-POINTER	<div style="border: 1px solid black; padding: 5px; display: inline-block;">A B C</div>
8	b	(STACK.TOP = B & TOKEN = b) POP STACK PUSH "b"	<div style="border: 1px solid black; padding: 5px; display: inline-block;">b B C</div>
9	b	(STACK.TOP = TOKEN) POP STACK MOVE TOKEN-POINTER	<div style="border: 1px solid black; padding: 5px; display: inline-block;">B C</div>
10	e	(STACK.TOP = C & TOKEN = e) POP STACK PUSH "D"	<div style="border: 1px solid black; padding: 5px; display: inline-block;">D C</div>
11	e	(STACK.TOP = D & TOKEN = e) POP STACK PUSH "e"	<div style="border: 1px solid black; padding: 5px; display: inline-block;">e D</div>
12	e	(STACK.TOP = TOKEN) POP STACK MOVE TOKEN-POINTER	<div style="border: 1px solid black; padding: 5px; display: inline-block;">D</div>
13		(BOT-OF-STACK & END-OF-LINE) DONE	<div style="border: 1px solid black; padding: 5px; display: inline-block;"></div>

Figure 15 (continued)

The previous examples used an abstract grammar for tutorial purposes. The example in Figure 16 uses a more functional grammar and a parse of an input string using an explicit stack parser.

Example of Explicit Stack Parsing



Note: Non-terminals are enclosed in parenthesis
 This grammar is almost ll1. The production where term reduces to factor or factor,sign,term is an ll2 production.

Figure 16

<u>Step</u>	<u>Token</u>	<u>Execution</u>	<u>Explicit Stack</u>
1		(START) PUSH (equation) TOKEN-POINTER = 1	<div>(equation)</div>
2	A	(STACK.TOP = (equation) & TOKEN = A) POP STACK PUSH (term) = (term) ;	<div>(term) = (term) ; (equation)</div>
3	A	(STACK.TOP = (term) & TOKEN = A) POP STACK PUSH (factor)	<div>(factor) (term) = (term) ;</div>
4	A	(STACK.TOP = (factor) & TOKEN = A) POP STACK PUSH (variable)	<div>(variable) (factor) = (term) ;</div>
5	A	(STACK.TOP = (variable) & TOKEN = A) POP STACK PUSH A	<div>A (variable) = (term) ;</div>
6	A	(STACK.TOP = TOKEN) POP STACK MOVE TOKEN-POINTER	<div>A = (term) ;</div>
7	=	(STACK.TOP = TOKEN) POP STACK MOVE TOKEN-POINTER	<div>= (term) ;</div>
8	1	(STACK.TOP = (term) & TOKEN = 1) POP STACK PUSH (factor)	<div>(factor) (term) ;</div>
9	1	(STACK.TOP = (factor) & TOKEN = 1) POP STACK PUSH (number)	<div>(number) (factor) ;</div>

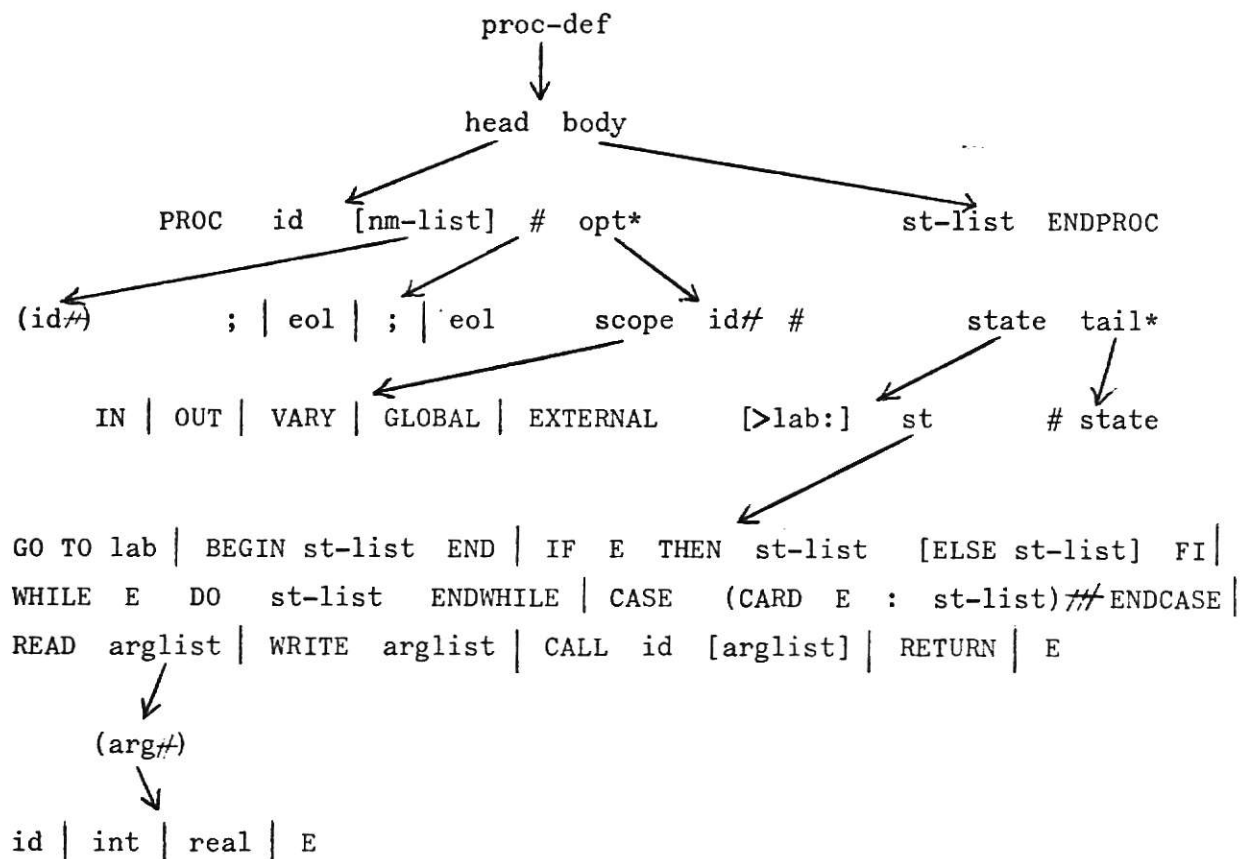
Figure 16 (continued)

<u>Step</u>	<u>Token</u>	<u>Execution</u>	<u>Explicit Stack</u>
10	1	(STACK.TOP = (number) & TOKEN = 1) POP STACK PUSH 1	<div style="border: 1px solid black; padding: 5px; width: fit-content; margin: 0 auto;"> 1 (number) ; </div>
11	1	(STACK.TOP = TOKEN) POP STACK MOVE TOKEN-POINTER	<div style="border: 1px solid black; padding: 5px; width: fit-content; margin: 0 auto;"> 1 ; </div>
12	;	(STACK.TOP = TOKEN) POP STACK MOVE TOKEN-POINTER	<div style="border: 1px solid black; padding: 5px; width: fit-content; margin: 0 auto;"> 1 </div>
13		(BOT-OF-STACK & END-OF-LINE) DONE	<div style="border: 1px solid black; height: 20px; width: fit-content; margin: 0 auto;"></div>

Figure 16 (continued)

The grammar for the CS-700 Interpreter is depicted in Figure 17.

Grammar for CS-700 Interpreter



- Notes:
1. Terminals are in upper-case type.
 2. E represents an expression.
 3. [] represents an entry that may or may not be present.
 4. * represents an entry that may be present more than one time.
 5. // represents more than one entry separated by commas.
 6. /// represents more than one entry separated by semicolons.
 7. eol represents an end-of-line.
 8. This grammar is almost an lli, however, the productions from "#" and "arg" cause it to look ahead two tokens.

Figure 17

The rules for the CS-700 Interpreter grammar are depicted in Figure 18.

Rules for CS-700 Production (Head)

Production	Terminals											
	PROC	id	(,)	;	eol	IN	OUT	VARY	GLOB	EXT
proc-def	push prod	Note 3										
head	push prod											
PROC	pop both											
id		pop both										
nm-list			push prod			pop prod	pop prod					
(pop both									
id#		pop both			pop prod							
,				Note 1	pop prod							
)					pop both							
#												
;						Note 2	Note 2					
eol							pop both					
opt*								push prod	push prod	push prod	push prod	push prod
scope								push prod	push prod	push prod	push prod	push prod
IN								pop both				
OUT									pop both			
VARY										pop both		
GLOB											pop both	
EXT												pop both

Figure 18

Rules for CS-700 Production (Body)

Production	>	GO TO	BEGIN	IF	WHILE	READ	WRITE	CALL	RETURN	E	:	eol	END-PROC	(id int real	other terminals
opt*	pop prod	pop prod	pop prod	pop prod	pop prod	pop prod	pop prod	pop prod		pop prod						
body	push prod	push prod	push prod	push prod	push prod	push prod	push prod	push prod		push prod						
st-list	push prod	push prod	push prod	push prod	push prod	push prod	push prod	push prod		push prod						
state	push prod	push prod	push prod	push prod	push prod	push prod	push prod	push prod	push prod	push prod			END			
tail*											push prod	push prod				
>	pop prod	pop prod	pop prod	pop prod	pop prod	pop prod	pop prod	pop prod	pop prod	pop prod						
st	push prod	push prod	push prod	push prod	push prod	push prod	push prod	push prod	push prod	push prod						
arglist														push prod		
arg //										Note 5					Note 5	
E										pop both						
other terminals											Note 2					Note 4

Figure 18 (continued)

Notes:

1. Since there may be a series of id's separated by commas, both are popped and another "id," production is pushed.
2. This is one place where the grammar is $ll2$. Therefore the parser must look at the next terminal before taking action.
3. Blanks are either errors or impossible conditions.
4. Pop both if match, otherwise it is an error.
5. Since there may be a series of arg's separated by commas, both are popped and another "arg," production is pushed.

Figure 18 (continued)

The example in Figure 19 is a parse of a procedure using an explicit stack and the grammar for the CS-700 Interpreter. This example will omit the parsing action for expressions (E), and instead, will call the Bottom-Up Parser when an E is encountered. Bottom-up parsing will be explained in the next section. Normally, the source code would be in the form of 3-tuple tokens (see Chapter 2). However, for ease in reading, the source code is not converted to tokens in this example.

Example of Parsing in CS-700 Interpreter

Grammar (ref. Figure 17)

<u>Source code</u>	PROC A (NUM):
	VARY NUM;
	NUM ← NUM * 3.142; (see Note)
	ENDPROC

Note: For this example the expression is replaced by "E".

<u>Step</u>	<u>Token</u>	<u>Execution</u>	<u>Explicit Stack</u>
1		(Start) PUSH "proc-def" TOKEN-POINTER = 1	proc-def
2	PROC	POP STACK PUSH "head body"	head body -proc-def-
3	PROC	POP STACK PUSH "PROC id [nm-list] # opt*"	PROC id [nm-list] # opt* -head- body

Figure 19

<u>Step</u>	<u>Token</u>	<u>Execution</u>	<u>Explicit Stack</u>
4	PROC	(Match) POP STACK MOVE TOKEN-POINTER	<div> PROC id [nm-list] # opt* body </div>
5	A	(Match) POP STACK MOVE TOKEN-POINTER	<div> id [nm-list] # opt* body </div>
6	(POP STACK PUSH "(id#)"	<div> (id#) [nm-list] # opt* body </div>
7	((Match) POP STACK MOVE TOKEN-POINTER	<div> (id#) # opt* body </div>
8	NUM	(Match) POP STACK MOVE TOKEN-POINTER	<div> id#) # opt* body </div>
9)	(Match) POP STACK MOVE TOKEN-POINTER	<div>) # opt* body </div>
10	;	POP STACK PUSH ";;"	<div> ; # opt* body </div>
11	;	(Match) POP STACK MOVE TOKEN-POINTER	<div> ; opt* body </div>

Figure 19 (continued)

<u>Step</u>	<u>Token</u>	<u>Execution</u>	<u>Explicit Stack</u>
12	VARY	POP STACK PUSH "scope id// #"	<div> scope id// # opt* body </div>
13	VARY	POP STACK PUSH "VARY"	<div> VARY -seepe- id// # opt* body </div>
14	VARY	(Match) POP STACK MOVE TOKEN-POINTER	<div> VARY id// # opt* body </div>
15	NUM	(Match) POP STACK MOVE TOKEN-POINTER	<div> -id// # opt* body </div>
16	;	POP STACK PUSH ";;"	<div> ; -#- opt* body </div>
17	;	(Match) POP STACK MOVE TOKEN-POINTER	<div> -#- opt* body </div>
18	E	(No more head) POP STACK	<div> -opt*- body </div>
19	E	POP STACK PUSH "st-list ENDPROC"	<div> st-list ENDPROC -body- </div>
20	E	POP STACK PUSH "state tail"	<div> state tail* -st-list- ENDPROC </div>

Figure 19 (continued)

<u>Step</u>	<u>Token</u>	<u>Execution</u>	<u>Explicit Stack</u>
21	E	POP STACK PUSH ">lab:] st"	<div> [>lab:] st -state- tail* ENDPROC </div>
22	E	(No optional label) POP STACK	<div> -[>lab+] - st tail* ENDPROC </div>
23	E	POP STACK PUSH "E"	<div> E -st- tail* ENDPROC </div>
24	E	POP STACK CALL BOTTOM-UP PARSER MOVE TOKEN-POINTER	<div> -E- tail* ENDPROC </div>
25	;	POP STACK PUSH "# state"	<div> # state -tail*- ENDPROC </div>
26	;	POP STACK PUSH ";;"	<div> ; -#- state ENDPROC </div>
27	;	(Match) POP STACK MOVE TOKEN-POINTER	<div> -#- state ENDPROC </div>
28	ENDPROC	(No more statements) POP STACK	<div> -state- ENDPROC </div>
29	ENDPROC	(Match) POP STACK MOVE TOKEN-POINTER	<div> -ENDPROC- </div>
30		(Bot-of-Stack and End-of-Line) DONE	<div> </div>

Figure 19 (continued)

3.4 Bottom-Up Parsing. In the previous example whenever an expression was detected the algorithm called for the Bottom-Up Parser. Expressions are composed of identifiers, strings, numbers, operators and delimiters. The function of a bottom-up parser is to check the syntax of these elements with the operator grammar, the rules for expressions. The operator grammar sets the rules for the relative positioning of the elements of an expression and the precedence of the operators. Since this is an operator precedence parser, the operands are invisible.

A bottom-up parser also can be implemented using a stack. Some bottom-up parsers use two stacks, one for operators and delimiters, and the other for identifiers and numbers. Other bottom-up parsers use one stack for all. In the examples in this section only one stack will be used.

Instead of the operators Push and Pop, the stack operators in the bottom-up parsing examples in this section will use Push and Reduce. The Push operator performs in the same manner as in the top-down parser. However, the Reduce operator performs multiple pop operations, and replaces the popped elements with a temporary element (E). For example, assume the elements "A + 5" are in a stack and the reduce operator is performed. The stack would now contain "E", the temporary element.

The bottom-up parser is faced with three problems; when to push, when to replace, and how much is to be replaced. In the following examples and, in general, in the CS-700 Interpreter the question of how much is to be replaced is limited to the following:

1. operand, operator, operand
2. parenthesis, operand, parenthesis
3. bracket, operand, bracket

An operand can be an identifier, number, or temporary (E).

The following are some examples:

	<u>Before Reduction</u>	<u>After Reduction</u>
Example #1- A + 5	<div style="border: 1px solid black; padding: 5px; display: inline-block;"> 5 + A </div>	<div style="border: 1px solid black; padding: 5px; display: inline-block;"> E </div>
Example #2- 500 + E	<div style="border: 1px solid black; padding: 5px; display: inline-block;"> E * 500 </div>	<div style="border: 1px solid black; padding: 5px; display: inline-block;"> E </div>
Example #3- E + A	<div style="border: 1px solid black; padding: 5px; display: inline-block;"> A + E </div>	<div style="border: 1px solid black; padding: 5px; display: inline-block;"> E </div>
Example #4- E + E	<div style="border: 1px solid black; padding: 5px; display: inline-block;"> E + E </div>	<div style="border: 1px solid black; padding: 5px; display: inline-block;"> E </div>
Example #5- (A + 5)	<div style="border: 1px solid black; padding: 5px; display: inline-block;"> 5 + A (</div>	<div style="border: 1px solid black; padding: 5px; display: inline-block;"> E </div>
	<div style="border: 1px solid black; padding: 5px; display: inline-block;">) E (</div>	<div style="border: 1px solid black; padding: 5px; display: inline-block;"> E </div>

The questions of when to push and when to replace are determined by rules formed from the precedence of the operators allowed in a grammar. For example, normally the expression $3 + 4 * 5$ would give an answer of 23 because the multiplication operation is assumed to have a higher precedence than the addition operation. But, $(3 + 4) * 5$ would be

35 because the parentheses (delimiters) have a higher precedence than the multiplication operation.

A bottom-up parser requires a grammar similar to a top-down parser. A simple grammar for a bottom-up parser is depicted in Figure 20. In this example the operations allowed are multiplication (*) and addition (+). The delimiters allowed are the left and right parentheses.

Simple Grammar for a Bottom-Up Parser

Grammar (E)
operator precedence

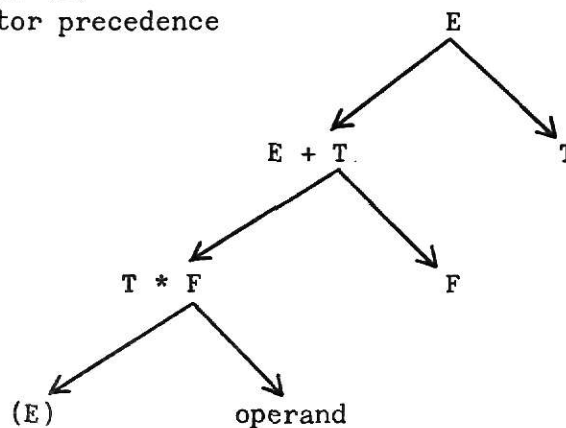


Figure 20

Next, the rules for the simple grammar must be defined. The first rule is that there cannot be two adjacent operands. The second rule is that the low to high precedence of operators is (, +, *, and) respectively. These rules are depicted in the table for Rules for Parsing Grammar (E) in Figure 21.

Rules for Parsing Grammar (E)

Operator on Stack.Top	Next Token in Input					
	+	*	()	operand	end-of- input
+	Reduce	Push *	Push (Reduce	Push operand	Reduce
*	Reduce	Reduce	Push *	Reduce	Push operand	Reduce
(Push +	Push *	Push (Push)	Push operand	Reduce
)	Reduce	Reduce	Reduce	Reduce	Error	Reduce
bot-of- stack	Push +	Push *	Push (Error	Push operand	Done

Figure 21

Notice that in the above rules there is no row for operands. This is because once an operand has been pushed onto the stack, it becomes invisible when making the next comparison. For example, if "A + B" are in the stack and the next input token is "*", then the "+" would be compared to the "*" because the operand is invisible.

In summary, how-much-to-reduce is determined by a set of rules developed from formations of expressions. The when-to-push and when-to-reduce questions are determined by rules for parsing the grammar which in turn are developed from the relative positioning of elements of the expressions and the precedence of the operators in the expressions.

Figure 22 depicts the parsing of an input string (expression) using Grammar (E) and the Rules for Parsing Grammar (E).

Bottom-Up Parse Using Grammar (E)

Input string-

(A	+	100)	+	5	*	B
---	---	---	-----	---	---	---	---	---

<u>Step</u>	<u>Token on Stack.Top</u>	<u>Token in String</u>	<u>Execution</u>	<u>Stack</u>
1	bot-of-stack	(Push ((
2	(A	Push operand	A (
3	(+	Push +	+ A (
4	+	100	Push operand	100 + A (
5	+)	Reduce	100 + A } E (
6	()	Push)) E (
7)	+	Reduce) E } E (
8	bot-of-stack	+	Push +	+ E
9	+	5	Push operand	5 + E
10	+	*	Push *	* 5 + E

Figure 22

<u>Step</u>	<u>Token on Stack.Top</u>	<u>Token in String</u>	<u>Execution</u>	<u>Stack</u>
11	*	B	Push operand	<div> <div>B</div> <div>*</div> <div>5</div> <div>+</div> <div>E</div> </div>
12	*	end-of-input	Reduce	<div> <div> <div>B</div> <div>*</div> <div>5</div> <div>+</div> <div>E</div> </div> <div>} E</div> </div>
13	+	end-of-input	Reduce	<div> <div> <div>E</div> <div>+</div> <div>E</div> </div> <div>} E</div> </div>
14	bot-of-stack	end-of-input	Done	<div> <div>E</div> </div>

Figure 22 (continued)

In the CS-700 Interpreter the precedence of the operators is a right-to-left precedence. For example, for the expression "3 * 4 + 5" the answer would be 27 — not 60 — because the addition operator is to the right of the multiplication operator, and therefore has a higher precedence.

The grammar for the CS-700 Interpreter Bottom-Up Parser is depicted in Figure 23.

Grammar for CS-700 Bottom-Up Parser

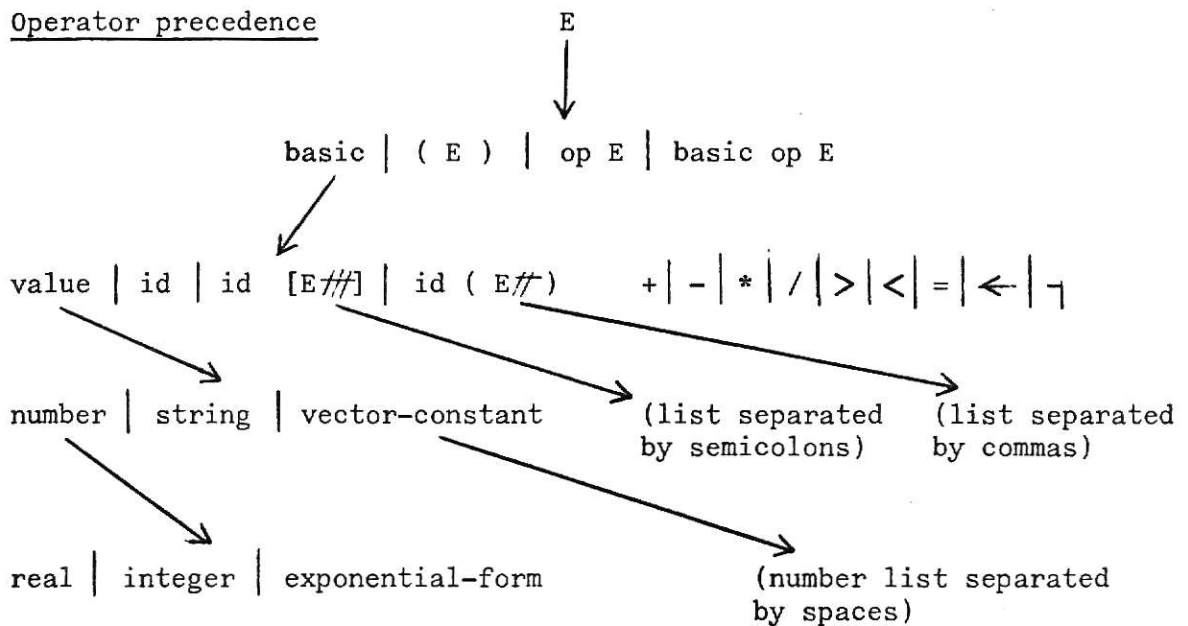


Figure 23

The rules for CS-700 Interpreter parsing are depicted in Figure 24.

Rules for CS-700 Interpreter Bottom-Up Parser

Stack Top	Next Token in Input String								
	[]	;	,	()	opr	opnd	end input
[push	push	push	push	push	error	push	push	error
]	reduce	reduce	reduce	reduce	reduce	reduce	reduce	reduce	reduce
;	push	push	push	error	push	error	push	push	error
,	push	error	error	push	push	push	push	push	error
(push	error	error	push	push	push	push	push	error
)	error	reduce	reduce	reduce	error	reduce	reduce	reduce	reduce
opr	push	reduce	reduce	reduce	push	reduce	push	push	reduce
bot-of- stack	push	error	error	error	push	error	push	push	done

NOTE: operands once pushed onto the stack are invisible.
when an operator is on Stack.Top and another operator is the
next token, the push action results in right-to-left
precedence of operators.

Figure 24

An example of a bottom-up parse using the CS-700 Interpreter grammar and rules for parsing is depicted in Figure 25.

Bottom-Up Parse in CS-700 Interpreter

Input string-		A	(B	,	C)	*	2
---------------	--	---	---	---	---	---	---	---	---

Step	Token on Stack.Top	Token in String	Execution	Stack
1	bot-of-stack	A	Push operand	A
2	bot-of-stack	(Push ((A
3	(B	Push operand	B (A
4	(,	Push ,	, B (A
5	,	C	Push operand	C , B (A
6	,)	Push)) C , B (A
7)	*	Reduce) C , B (A

Figure 25

<u>Step</u>	<u>Token on Stack.Top</u>	<u>Token in String</u>	<u>Execution</u>	<u>Stack</u>
8	bot-of- stack	*	Push *	<div style="border: 1px solid black; padding: 2px; display: inline-block; text-align: center;">* E</div>
9	*	2	Push operand	<div style="border: 1px solid black; padding: 2px; display: inline-block; text-align: center;">2 * E</div>
10	*	end-of- line	Reduce	<div style="border: 1px solid black; padding: 2px; display: inline-block; text-align: center;">2 * E } E</div>
11	bot-of stack	end-of- line	Done	<div style="border: 1px solid black; padding: 2px; display: inline-block; text-align: center;">E</div>

NOTE: Operands on stack are invisible

Figure 25 (continued)

An algorithm for a bottom-up parser using a decision table of rules for parsing can be a rather simple procedure. Figure 26 is an example of such an algorithm for the CS-700 Interpreter.

Algorithm for Bottom-Up Parser
in CS-700 Interpreter

```

PROCEDURE BOTTOM-UP PARSER (RETURN-CODE)
COMMENT: ROW IS THE ROW NUMBER (FIGURE 24), COL IS THE COLUMN NUMBER
            (FIGURE 24)
COMMENT: SET ROW NUMBER TO BOTTOM-OF-STACK
ROW = 8
DO UNTIL DONE
    BEGIN
        READ TOKEN
        IF END-TOKEN THEN COL = 9
        IF TOKEN.INDEX = "LBRAK" THEN COL = 1
        IF TOKEN.INDEX = "RBRAK" THEN COL = 2
        IF TOKEN.INDEX = "SCOLON" THEN COL = 3
        IF TOKEN.INDEX = "COMMA" THEN COL = 4
        IF TOKEN.INDEX = "LPAREN" THEN COL = 5
        IF TOKEN.INDEX = "RPAREN" THEN COL = 6
        IF TOKEN.TAG = "ID" OR "INT" OR "REAL" THEN COL = 8
        COMMENT: CHECK TABLE FOR ACTION REQUIRED
        IF TABLE(ROW,COL) = "PUSH" THEN CALL PUSH-TOKEN
        IF TABLE(ROW,COL) = "REDUCE" THEN CALL REDUCE-STACK
        IF TABLE(ROW,COL) = "DONE" THEN RETURN
        IF RETURN-CODE = "BAD" THEN RETURN
    END-DO
END-PROC

```

Figure 26

PROCEDURE PUSH-TOKEN

COMMENT: IF TOKEN IS AN OPERATOR OR DELIMITER IT BECOMES THE ROW VALUE
PUSH TOKEN ON STACK

IF TOKEN.TAG = "OPR" OR "DEL" THEN ROW = COL

RETURN

END-PROC

PROCEDURE REDUCE-STACK

COMMENT: L-H-E IS A FUNCTION THAT FINDS THE LEFT-HAND-END OF THE TOKENS
ON THE STACK, AND POPS THESE TOKENS.

IF ROW = 2 THEN L-H-E-LBRAK

IF ROW = 6 THEN L-H-E-RPAREN

IF ROW = 7 THEN L-H-E-OPR

COMMENT: SET ROW = LAST OPERATOR OR DELIMITER ON STACK

IF STACK.TOP = BOT-OF-STACK THEN ROW = 8

IF STACK.TOP.INDEX = "LBRAK" THEN ROW = 1

IF STACK.TOP.INDEX = "RBRAK" THEN ROW = 2

IF STACK.TOP.INDEX = "SCOLON" THEN ROW = 3

IF STACK.TOP.INDEX = "COMMA" THEN ROW = 4

IF STACK.TOP.INDEX = "LPAREN" THEN ROW = 5

IF STACK.TOP.INDEX = "RPAREN" THEN ROW = 6

IF STACK.TOP.TAG = "OPR" THEN ROW = 7

PUSH "E" ON STACK

RETURN

END-PROC

PROCEDURE ERR-RTNE

PRINT "ERROR"

PRINT (TOKEN.TAG, TOKEN.INDEX, TOKEN.LOCATION)

RETURN-CODE = "BAD"

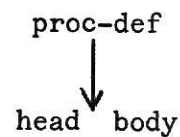
RETURN

END-PROC

Figure 26 (continued)

3.5 Code Generation. This section describes where the parser takes semantic actions, generates code and the function of the code. The master's report on Models for Translator Design presents the format of the code and execution of the code.

The Grammar for the CS-700 Interpreter, Figure 17, depicts the first production as follows:



This production shows the two basic parts of the grammar for procedures. The "head" is a section of the grammar that names the procedure, identifies the arguments, and declares the scope of the variables.

Code is not generated during the parsing of statements in the "head" section until all the "head" statements are parsed. At this time the parser reviews the identifiers in the symbol table. For each identifier that has a scope of IN or VARY, the parser generates code to link these arguments from the calling program to the corresponding value/address field in the symbol table.

Once the code for the "head" is generated, the parser begins parsing the productions in the body. The body contains a list of program instructions. As each statement is parsed, code is generated to perform the action required by the instruction. When the body is completely parsed and the "ENDPROC" is detected, the parser generates code to link return arguments back to the calling procedure. Return arguments are identified in the symbol table as those with a scope of OUT or VARY.

After the parser has generated all the code it then calls for the execution phase. Again, this phase is discussed in detail in Models for Translator Design. The next two figures depict all the statements that cause semantic actions to be taken, a description of the actions taken, and all the statements that cause code to be generated and a description of the function of the generated code.

The semantic action taken during parsing is depicted in Figure 27.

Semantic Action During Parsing

<u>Production</u>		<u>Semantic Actions</u>
1. PROC id [nm-list]		
	PROC	none
	id	Set symbol table TYPE to PROC-ID.
	(id#)	Set symbol table TYPE to IN.
2. scope id		
	IN OUT VARY GLOBAL EXTERNAL	none
	id	Set symbol table TYPE to keyword.
		Note: If TYPE is IN, then it can only be changed to VARY or OUT. If TYPE is LOCAL (the default value), then it can only be changed to GLOBAL.
3. >lab:		
	>	none
	lab	Set symbol table TYPE to LABEL.
		Note: TYPE could have been LOCAL or LABEL.
5. CALL id [arglist]		
	CALL	none
	id	Set symbol table TYPE to PROC-ID
	(arg #)	none

Figure 27

The code generation actions taken during parsing are depicted in Figure 28.

<u>Code Generation During Parsing</u>	
<u>Production</u>	<u>Code Generation During Parsing</u>
1. body	Code is generated to forward link all arguments from the calling procedure.
2. GO TO lab	GO TO none
lab	Case 1: If the label has a value of a line number, then code is generated for a branch direct instruction. Case 2: If the label does not have a value of a line number (the label has not been parsed prior to this statement), then code is generated for a branch indirect thru the symbol table instruction.
3. IF E THEN st-list [ELSE st-list] FI	
IF	none
E	Code is generated to call the bottom-up parser.
THEN	Code is generated for a branch-if-false instruction where the address of the branch is temporarily unknown. The current code address is saved so the unknown address can be completed later.

Figure 28

st-list	Code is generated as required by the statements following.
FI	Complete the unknown branch address of the instruction that jumped over the ELSE statement list.
4. WHILE E DO st-list ENDWHILE	
WHILE	Save the address of the next code to be generated. This address will be inserted into the branch address at the end of the "DO" list of statements to provide the loop back to the beginning of the "DO" list of statements.
E	Code is generated to call the bottom-up parser.
DO	Code is generated for a branch-if-false instruction. The branch address is temporarily unknown. This address when completed will be to the first instruction after the "DO" loop.
st-list	Code is generated as required by the statements following.
ENDWHILE	Code is generated to branch back to the beginning of the "DO" loop (the call to the bottom-up parser). Complete the unknown branch address of the branch-if-false instruction with the address of the next line of code to be generated.

Figure 28 (continued)

st-list Code is generated as required by the
statements following.

FI Complete the unknown branch address
with the address of the next code to be
generated.

or

IF none

E Code is generated to call the bottom-up
parser.

THEN Code is generated for a branch-if-false
instruction where the address of the
branch is temporarily unknown. This
address when completed will be to the
first statement code of the ELSE section
The current code address is saved so the
unknown branch address can be completed
later.

st-list Code is generated as required by the
statements following.

ELSE Code is generated to branch over the
statements in the ELSE statement list.
The branch address is temporarily unknown,
so the current code address is saved so
the unknown branch address can be completed
later.
Complete the unknown branch address of
the branch-if-false instruction.

Figure 28 (continued)

5. CALL id (arglist)		
CALL	none	
id	none	
(arglist)	Code is generated to call the bottom-up parser for each expression in the arglist. Code is generated to link arguments to the called procedure. Code is generated to link a count of the number of arguments to the called procedure. Code is generated to activate the called procedure.	
6. RETURN or ENDPROC		
RETURN ENDPROC	Code is generated to link arguments back to the calling procedure.	
7. start-of-line		
start-of-line token	Code is generated to set the source line counter to the appropriate line number.	
8. E		
	Code is generated to call the bottom-up parser.	

Figure 28 (continued)

The example in Figure 29 depicts a parsing of source code and indicates where semantic actions (SA) or code generation (CG) occurs. A rule number is also included to allow for ease in referencing the specific semantic action occurring or code generation action occurring. For example, an "SA1" would indicate semantic action rule number one in Figure 27. A "CG5" would indicate code generation rule number five.

Example of Parsing in CS-700 Interpreter

Grammar (ref. Figure 17)

Source code

```

PROC A (NUM);
VARY NUM;
NUM ← NUM * 3.142;      (see Note)
ENDPROC

```

Note: For this example the expression is replaced by "E".

<u>Step</u>	<u>Token</u>	<u>Execution</u>	<u>Explicit Stack</u>	<u>Semantic Action or Code Generation</u>
1		(Start) PUSH "proc-def" TOKEN-POINTER = 1	proc-def	
2	PROC	POP STACK PUSH "head body"	head body proc-def	
3	PROC	POP STACK PUSH "PROC id [nm-list] # opt*"	PROC id [nm-list] # opt* -head- body	

Figure 29.

<u>Step</u>	<u>Token</u>	<u>Execution</u>	<u>Explicit Stack</u>	<u>Semantic Action or Code Generation</u>
4	PROC	(Match) POP STACK MOVE TOKEN-POINTER	<div> -PROC- id [nm-list] # opt* body </div>	
5	A	(Match) POP STACK MOVE TOKEN-POINTER	<div> -id- [nm-list] # opt* body </div>	SA1
6	(POP STACK PUSH "(id#)"	<div> (id#) [nm-list] # opt* body </div>	
7	((Match) POP STACK MOVE TOKEN-POINTER	<div> -(id#) # opt* body </div>	
8	NUM	(Match) POP STACK MOVE TOKEN-POINTER	<div> -id#-) # opt* body </div>	SA1
9)	(Match) POP STACK MOVE TOKEN-POINTER	<div> ->- # opt* body </div>	
10	;	POP STACK PUSH ";;"	<div> ; -#- opt* body </div>	

Figure 29 (continued)

<u>Step</u>	<u>Token</u>	<u>Execution</u>	<u>Explicit Stack</u>	<u>Semantic Action or Code Generation</u>
11	;	(Match) POP STACK MOVE TOKEN-POINTER	<div>-+- opt* body</div>	
12	VARY	POP STACK PUSH "scope id# #"	<div>scope id# # opt* body</div>	
13	VARY	POP STACK PUSH "VARY"	<div>VARY -scope- id# # opt* body</div>	
14	VARY	(Match) POP STACK MOVE TOKEN-POINTER	<div>VARY id# # opt* body</div>	
15	NUM	(Match) POP STACK MOVE TOKEN-POINTER	<div>-id#- # opt* body</div>	SA2
16	;	POP STACK PUSH ";;"	<div>; -#- opt* body</div>	
17	;	(Match) POP STACK MOVE TOKEN-POINTER	<div>-+- opt* body</div>	
18	E	(No more head) POP STACK	<div>-opt*- body</div>	
19	E	POP STACK PUSH "st-list ENDPROC"	<div>st-list ENDPROC -body-</div>	CS1

Figure 29 (continued)

<u>Step</u>	<u>Token</u>	<u>Execution</u>	<u>Explicit Stack</u>	<u>Semantic Action or Code Generation</u>
20	E	POP STACK PUSH "state tail"	<div> state tail* -st-list- ENDPROC </div>	
21	E	POP STACK PUSH " [>lab:] st"	<div> [>lab:] st -state- tail* ENDPROC </div>	
22	E	(No optional label) POP STACK	<div> -[>lab+]- st tail* ENDPROC </div>	
23	E	POP STACK PUSH "E"	<div> E -st- tail* ENDPROC </div>	
24	E	POP STACK CALL BOTTOM-UP PARSER MOVE TOKEN-POINTER	<div> -E- tail* ENDPROC </div>	CG8
25	;	POP STACK PUSH "# state"	<div> # state -tail*- ENDPROC </div>	
26	;	POP STACK PUSH ";;"	<div> ; -#- state ENDPROC </div>	
27	;	(Match) POP STACK MOVE TOKEN-POINTER	<div> -#- state ENDPROC </div>	
28	ENDPROC	(No more statements) POP STACK	<div> -state- ENDPROC </div>	
29	ENDPROC	(Match) POP STACK MOVE TOKEN-POINTER	<div> ENDPROC </div>	CG6

Figure 29: (continued)

<u>Step</u>	<u>Token</u>	<u>Execution</u>	<u>Explicit Stack</u>	<u>Semantic Action or Code Generation</u>
30		(Bot-of-Stack and End-of-Line) DONE		

Symbol Table After Parsing

Index No.	Number of Characters	Literal Name	Scope	Type	Value or Address
1	1	A	PROC-ID		
2	3	NUM	VARY		

Figure 29 (continued)

When the top-down parser in the CS-700 Interpreter recognizes an expression, it calls the bottom-up parser. The bottom-up parser analyzes the expression, generates code and returns control to the top-down parser. The code generated is determined by the type of expression. There are five types of expressions in the CS-700 Interpreter bottom-up grammar. One expression, indexing, is not implemented at this time and will only be referenced in the following discussion. Figure 30 depicts the types of expression and the resulting code generation.

Code Generated During Bottom-Up Parsing

<u>Type</u>	<u>Code Generated</u>
1. (E)	No code is generated when the parentheses are eliminated.
2. E op E	A code triple is generated in the form: <u>op</u> , <u>E</u> , <u>E</u>
3. op E	A code triple is generated in the form: <u>op</u> , <u>E</u> , <u>null</u>
4. id [E;E;...E]	This type is indexing and has not been implemented yet.
5. id (E,E,...E)	Code is generated in the same manner as in top-down procedures.

Note: "E" can be a temporary value, a constant, or an identifier.

Figure 30

An example of code generation in the bottom-up parser is depicted in Figure 31.

Example of Code Generation
During Bottom-Up Parsing

Grammar (ref. Figure 23)

Source code $X \leftarrow A * B + 100$

<u>Step</u>	<u>Token</u>	<u>Remarks</u>	<u>Code Generated</u>
1.	100	Note: bottom-up parsing is right-to-left	none
2.	+ 100		none
3.	B + 100	This is a Type 3 expression.	IPLUS;id,n-index,loc;int,100,loc
4.	* E	When the code in step 3 is executed, a temporary (E) will be left on the stack. Therefore, E is represented here as the token.	none
5.	A * E	This is a Type 3 expression.	IMULT;id,n-index,loc;temp
6.	$\leftarrow E$		none
7.	$X \leftarrow E$	This is a Type 3 expression.	IASG;id,n-index,loc;temp
8.	E	This is the end of bottom-up parsing of the expression. Control is returned to the top-down parser.	

Figure 31

REFERENCE NOTES

<u>Subject in Project Report</u>	<u>Subject in Compiler Construction for Digital Computers - David Gries</u>
1. Lexical Scanning.	Gries discusses the theory of scanning and problems with non-deterministic grammars. The report discusses only deterministic grammars and focuses on token and symbol table construction in the CS-700 Interpreter. The algorithms are more detailed and made more readable by the use of comments, case statements, and structured programming techniques.
2. Top-Down Parsing.	Gries discusses the theory of parsing and the various types of recognizers. The report includes a discussion of recursive decent parsing not in Gries. In addition, the parsing algorithms are in more detail and include error procedures not in Gries. Other features of the report such as the use of decision tables to portray the logic of algorithms add clarity to the discussion material. Also, the report includes parsing abstracts for the CS-700 Interpreter.
3. Bottom-Up Parsing.	Gries presents both precedence parsing and operator precedence parsing in limited detail. The report expands on operator precedence parsing and presents abstracts of algorithms for CS-700 Interpreter Bottom-Up Parsing.
4. Code Generation.	Gries' coverage on this subject is rather general. The report concentrates on code generation in the CS-700 Interpreter.

TRAINING AIDS FOR TRANSLATOR DESIGN

by

JAMES R. MEYER

B.S., Benedictine College, KS, 1971

AN ABSTRACT OF A MASTER'S REPORT

submitted in partial fulfillment of the

requirements for the degree

MASTER OF SCIENCE

Department of Computer Science

KANSAS STATE UNIVERSITY
Manhattan, Kansas

1977

TRAINING AIDS
for
TRANSLATOR DESIGN

ABSTRACT

The purpose of this project is to produce algorithms and training aids to augment the classroom instruction of Translator Design I, Course Number CS 286-700. The algorithms demonstrate both abstract tutorial cases and specific implementations of a student-developed interpreter. The training aids consist of examples of traces of execution of a lexical scanner, and a parser (with and without code generation). In addition, examples of scanning, parsing and code generation using the student-developed CS700 Interpreter are included to prepare the students for implementation projects using this model. This material can be used to supplement the text, augment the classroom presentation, or as homework problems.