Math

9

UIUCDCS-R-74-658

A BASIC LANGUAGE INTERPRETER FOR THE INTEL 8008 MICROPROCESSOR

by

Alfred C. Weaver
Michael H. Tindall
Ronald L. Danielson

June, 1974

DEPARTMENT OF COMPUTER SCIENCE
UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN · URBANA, ILLINOIS

UIUCDCS-R-74-658

A BASIC LANGUAGE INTERPRETER FOR THE INTEL 8008 MICROPROCESSOR

by

Alfred C. Weaver
Michael H. Tindall
Ronald L. Danielson

June, 1974

Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, Illinois

ACKNOWLEDGEMENT

The authors wish to express their appreciation to Professor James Vander Mey, who originally suggested and subsequently supervised the entire project; to Professor Thomas Wilcox, who consulted on the design of the software techniques; to Gordon Peterson, who wrote an assembler for a version of the Intel assembly language; and to Guy Riddle, who wrote an Intel 8008 simulator for the IBM 360/75.

TABLE OF CONTENTS

1. INTRODUCTION

This report concerns the design and implementation of a BASIC language interpreter which is restricted to the environment of the Intel 8008 microprocessor. The value of the project was in overcoming the severe restrictions imposed by an 8-bit machine with only four general purpose registers, one accumulator, and two registers used simultaneously to form one core address. Furthermore, the machine's numerical capability is characterized by its two arithmetic instructions: 8-bit add and subtract. All addresses (14 bits) must be broken into high and low order bytes and stored in the H and L registers, respectively, for any memory reference. With a cycle time of 20 microseconds average, the machine is no number-crunching giant, but is more than adequate for the dedicated processor we envisioned.

This project is the direct result of a seminar course on mini-computers taught at the University of Illinois by Professor James Vander Mey. While the project itself was executed entirely on an Intel simulator, we feel that the techniques developed are directly applicable to the hardware described.

The hardware environment envisioned for this project included: a microprocessor with a 16K x 8-bit word memory (Intel MCS-8), an ASCII keyboard, a black-and-white television for the display, and a custom-designed hardware interface which continuously displays the contents of a 2K buffer in Intel memory. Design and implementation of the interface was wholly controlled by another group.

The language which has been implemented is a version of BASIC which incorporates all of the most common features of the language and a few versatile extensions. A conscious effort was made to "design with forethought" so that future modifications, either by the authors or others, would not require extensive modification of the existing routines. The BASIC language statements currently implemented include: LET, DIM, DEF, REM, GOTO, IF, FOR, TO, NEXT, GOSUB, RETURN, INPUT, PRINT, STEP, THEN, STOP, and END. The operators provided are addition, subtraction, multiplication, division, exponentiation, and the Boolean operators AND, OR, NOT, LESS-THAN, LESS-THAN-OR-EQUAL, GREATER-THAN, GREATER-THAN-OR-EQUAL, EQUAL, and NOT-EQUAL. The extensions to the language include: up to 26 3-dimensional arrays, up to 26 user-defined functions, user functions with nested (but not recursive) definitions 5 levels deep, six built-in functions (sine, cosine, arctangent, exponential, logarithm, and square root), full floating point arithmetic performed with 5 byte operands (4 byte mantissa, 1 byte exponent) with results rounded to 4 bytes (3 byte mantissa and 1 byte exponent), character strings in the PRINT statement, and FOR-loops nested up to 5 levels deep.

The BNF for the BASIC language implemented is supplied in Appendix A. The use of specially encoded keyboard keys for all multi-character keywords (LET, DIM, ..., END, relational operators, and user functions FNx) allows our grammer to be LL(1), thus simplifying the parsing technique.

The project broke neatly into two phases - design and development - and the development phase itself was split into three basic areas:

(1) the operating system, including master control of the CPU, mode switching, context switching, polling routine for the keyboard,

complete text editing system, screen display management, user source text memory management, and user "output page" memory management;

(2) the syntactic/semantic analyzer, including BASIC syntax recognizer, error detection/location routines, lexical analyzer, and symbol table manager;

(3) the execution routines, including specific utility routines needed by the syntax analyzer (stack control and lexical pointer movement), a floating point arithmetic package (add, subtract, multiply, divide, fix, float, compare, load, store, and error branching), a transcendental function evaluator (sine, cosine, log, exponential, arctangent, and square root), and a two-way floating-point-to-character-string and character-string-to-floating-point conversion package.

The following sections and appendices present an explicit report of the design philosophy and characteristics of the three major areas.

2. OPERATING SYSTEM

The operating system is charged with the responsibility of managing the CPU. As a dedicated processor, the CPU exists only to serve the user, but it does so in one of six possible environments.

2.1 Text Editing

In pure text editing mode the keyboard is used to build text on a display screen using the elementary operations of insert character, delete character, replace character, delete line, scroll up 1 or 8 lines, and scroll down 1 or 8 lines. No syntax checking of any kind is performed in this mode. To aid the user, a moveable cursor (@) is positioned immediately to the left of the character or line to be inserted, replaced, or deleted. Thus the programmer has constant and precise control over the location of any of his changes. The cursor is moved by depressing one of its special control keys:

(right →arrow)	move right one character; AB@CD becomes ABC@D
(left ←arrow)	move left one character; AB@CD becomes A@BCD
(up ↑arrow)	move cursor to the beginning of the current line or, if it is already at the beginning of a line, to the beginning of the previous line;

```
ABCD    becomes    ABCD
EF@G               @EFG
```

and

```
ABCD    becomes    @ABCD
@EFG               EFG
```

↓ (down arrow) move cursor to the beginning of the next line;

```
AB@CD    becomes    ABCD
EFGH                @EFGH
```

The user's memory space, at addresses 14K through 16K-1, is bounded by non-insertable markers at the beginning of the user's source text (static), end of source text (dynamic), and end of user's data page (dynamic). The display is initialized as follows:

```
[@!!!!!!!!!!!!!!!![!!!!!!!!!!!!!!!]
```

where [represents the beginning-of-text marker (source page and output page),

@ represents the cursor,

! represents the end-of-line marker,

and] represents the end-of-memory marker.

The text editor functions are accomplished by manipulating a line of text or the entire program as necessary. A replace character requires no text movement, but an insert or delete character requires that all of the text from the cursor to the end-of-memory marker be shifted one byte in core.

Insertion is typical of the operations performed, and some operation counts show that if an insertion is to occur n characters

to the left of the end-of-memory marker then the elapsed time t from picking up the character in the polling loop until the CPU returns to the polling loop is:

20 microseconds/instruction * (50 + 20n) instructions

For example:

n	t
10	0.5 milliseconds
100	41 milliseconds
1000	0.4 seconds

Assuming an average keyboard rate of 3 characters/second, a burst rate of 10 characters/second, and an average text movement of less than 100 characters, we found that the machine will adequately keep up with the user by using only a single one-byte hardware buffer.

Additional command codes were reserved (but not implemented) for such handy functions as FILE-SCREEN-TO-TAPE, MOVE-TAPE-TO-SCREEN, SORT-SCREEN-BY-SEQUENCE-NUMBERS, etc.

Appendix D documents the operating system/text editor. Each screen display shows the state of the screen just prior to the receipt of the next character from the keyboard.

2.2 Syntax Checking

When operating in the "BASIC" mode, the operating system provides line-by-line syntax checking of the text being inserted on the screen, in addition to providing all of the text editing features described in section 2.1 above. Every time the cursor moves right over an end-of-line marker (in response to a right arrow, down arrow, or insertion of an end-of-line character), the syntax analyzer is called to parse the line just created. Whenever the cursor moves left over an end-of-line,

the syntax analyzer (SYNA) is called to "unparse" the line just entered. This feature allows the user to interactively construct a syntactically correct BASIC program. Special command keys are provided to allow the user to change from "BASIC" mode to "TEXT" mode, and vice-versa.

2.3 Run-time Monitoring

When a RUN command is detected by the operating system (OS) command decoder, OS enters the third of its major roles, that of an execution monitor. In response to the RUN command, OS initializes all of the interpreter variables (primarily the symbol table and the parsing stacks), sets the lexical (LEXI) pointer to the first executable statement, removes the cursor from the program text, switches the screen display base address to point to the data page, and begins calling SYNA on a line-by-line basis. After each return from SYNA the keyboard strobe (an "in use" bit) is checked and any character on the data bus is read. The only character recognized while executing is a STOP IMMEDIATE command, which causes OS to jump to the immediately executable statement routines described below. Other characters are ignored until execution is terminated, either normally (by execution of a STOP statement) or as the result of an error. Normal termination leaves OS in the "BASIC" (syntax checking) mode. Commands are provided for jumping from data page to program page and back again. Detection of an error causes SYNA to jump to an OS routine for error handling.

2.4 Error Handling

When the error routine is entered the screen base pointer is shifted back to the program page, a message describing the error is inserted on a line following the one in which the error was detected,

and the cursor is inserted in the program at the spot nearest the error. This enables the user to quickly locate his error. The operating system then jumps to the text editing/syntax checking routines in anticipation of error correction attempts by the user.

2.5 Immediately Executable Statements

Immediately executable statements are handled as described under "run-time monitoring". The user may stop a running program by pressing the STOP IMMEDIATE key, thus causing the program to halt at the end of the currently executing line. The operating system then allows code to be entered line-by-line in the "immediately executable" mode. Each line must be followed by the EXECUTE IMMEDIATE command key, which causes the line just constructed to be parsed and immediately executed. The user may jump back and forth between the program and data pages, but may not modify the program unless he leaves the immediately executable mode. A RESTART command is provided which causes the user program to begin execution at the point where it was interrupted (unless one of the immediately executed statements caused a branch, in which case execution begins at the branch address). The user may operate the system as a desk calculator by initially jumping to the data page and inserting unnumbered BASIC statements, each followed by the EXECUTE IMMEDIATE command.

2.6 Input/Output

The operating system provides input-output facilities which may be used by SYNA when in the execute or execute immediate mode. All input requests and output text are displayed on the data page. The conversion routines in the floating point arithmetic package are used to provide the ASCII-to-binary and binary-to-ASCII conversions. Input

requests may be made to provide data for a scalar variable or an array element. Array subscripts are taken from SYNA's expression stack. A typical such input request might be:

```
INPUT X(2,5)> @
```

The input data is inserted to the right of the greater-than sign, and is converted to internal floating point by the appropriate conversion routine.

There are three separate output routines: one which causes the data page display to skip to a new line, one which outputs numbers, and one which outputs character strings. The character string routine requires the H and L registers to point to the beginning of the string and leaves the LEXI pointer to the right of the terminal quote. The numeric routine displays a number pointed to by the top of SYNA's address stack. Commands are ignored during this I/O activity. The output routine for numbers will print them in its choice of FORTRAN I, F, or E format, depending upon the magnitude and precision of the particular number being output. The input routine accepts the input string in any format.

All of the above routines are written as pure procedures, and in a modular fashion. There is a reasonable amount of optimization possible in the OS routines, particularly in the n-way branches. Total size of the text editor/operating system is about 3000 bytes.

3. SYNTACTIC AND SEMANTIC ANALYSIS

3.1 Syntactic Analysis

SYNA is the heart of the BASIC language capabilities of our system. It performs both syntactic correctness checking and interpretive BASIC language program execution. SYNA is called on a statement-by-statement basis from the operating system; when a statement has been correctly parsed or executed, a normal return is made to OS; if an error is detected, control passes to an error recovery procedure in OS where appropriate recovery measures are taken (these are documented elsewhere).

The syntax analyzer is essentially a stack oriented pushdown automaton. As a statement is parsed, the analyzer makes a sequence of state transitions that depend on the current input symbol (token) in the statement. Entering certain states causes the syntax analyzer to accept the statement, and likewise, other combinations of the analyzer's state and input token cause the analyzer to reject the statement and signal an error.

In an effort to conserve space, the state transitions are encoded in table form, making the syntax analyzer table-driven. A fairly small table driver routine causes new tokens to be input when required by a lexical analysis routine, moves the analyzer through the state table, manipulates the analyzer's stack, and invokes a set of execution routines that perform the actual operations defined in the BASIC language. For a more detailed explanation of the state transitions and a view of the

syntax table, see appendices B and C.

In addition to the analyzer's parser stack, a number of other stacks are maintained by SYNA. These include an address stack, an expression value stack and corresponding operator stack used for expression evaluation, a FOR-loop stack and a user function evaluation stack. The parser stack is used strictly for state transitions by the pushdown automaton. The address stack (2 bytes wide) is used for executing GOTO and GOSUB statements, for array reference resolution, and for any general bookkeeping operation requiring two bytes of stack storage. The expression stack (4 bytes wide) is used to store temporary arithmetic components and results involved in expression evaluation. The operator stack (2 bytes wide) uses one byte to identify the arithmetic operator involved and the other byte to hold the operator's F precedence (see D. Gries, Compiler Construction for Digital Computers, chapter 5, "Simple Precedence Grammers"). The stack is used in conjunction with the expression stack in evaluating expressions. The FOR-loop stack is used to keep track of FOR-loop nesting in the program. Each time a new FOR-loop is entered, an entry is pushed onto the FOR-loop stack. When the loop is exited, the stack is popped one level. The last stack is a user function stack (3 bytes wide). One byte points to the symbol table location of the function, and the other two bytes hold the return address for use after the function has been evaluated. The function stack is needed since one function may be defined in terms of another function.

Under the current implementation (easily modifiable) the function stack is 5 deep, the FOR-loop stack is 4 deep, the expression and operator stacks are each 10 deep, and the parser stack is 15 deep.

The syntax analyzer operates in one of three modes, determined

by the operating system. When the programmer pushes the RUN key, all variables (both compiler and program) are initialized to zero and then SYNA is called statement-by-statement throughout the program; SYNA again performs a syntax check on each statement and then executes the statement. This is the EXECUTE mode, in which all stacks are fully operational and utilized.

As explained in the operating system chapter, the programmer can type in his program in an interactive BASIC environment, giving him a syntax check on his statement as it is entered, but not actually executing the statement. This is the normal PARSE mode. The FOR-loop stack, parser stack, and address stack are operational in this mode. The expression stack, operator stack, and function evaluation stacks are not used since the statement is not actually executed.

One primary goal for the project was to provide good syntax checking and error detection in this normal PARSE mode. To do this, FOR-loop nesting should be checked even though the loop is not actually being executed (indeed, at the time the analyzer is parsing the FOR statement, the loop contents and the NEXT statement probably do not even exist!). Thus, SYNA builds the FOR-loop stack at parse time, enabling the check on loop nesting. Also, we require array and function definition before use so that we can insure that all uses of the same variable are consistent. To handle this, SYNA sets a flag bit in the symbol table when an array is defined by a DIM statement, and likewise when a function is defined by a DEF statement. Now all uses of the array or function name can be checked for syntactic correctness and previous definition.

The only complication which arises in the above scheme of syntax checking occurs when the programmer backs up the cursor and changes,

inserts, or deletes statements which have already been parsed. To adequately handle these occurances, the syntax parser actually needs to back up with the cursor and undo any FOR-loop nest checking or array/function definitions. For this purpose, the operating system will call SYNA in the UNPARSE mode whenever the cursor is backed up over an end-of-line. For example, if a NEXT statement is backed over, the FOR-loop stack will be pushed up one level to indicate that we are back inside a FOR-loop again. If a DIM statement is backed over, all the arrays dimensioned within that statement will have their "defined" bit turned off (i.e., "not defined") in the symbol table.

Eventually, the programmer will move the cursor forward again using the right arrow or down arrow keys; when this happens SYNA is called again in the regular PARSE mode, the statement is parsed again, and the normal parsing actions are taken.

3.2 Semantic Analysis

It is apparent that, depending upon the mode when SYNA is called, different operations are performed when a statement is parsed. As far as the formal parser is concerned, it means that any given state can have different semantic routines associated with it, depending upon the mode. Thus, each state entry in the table has three bits, one for each of the EXECUTE, PARSE, and UNPARSE modes, as well as an execution/semantic routine number. When a state is used in the table the execution/semantic routine is invoked only if the bit corresponding to the current mode is set. For example, the expression evaluation routine is an execution-only routine, having its EXECUTE bit set and its PARSE and UNPARSE bits not set in the SYNA table; thus this routine is called only if the mode is EXECUTE.

This concludes the discussion of the syntatic/semantic analyzer. The lexical analyzer and symbol table manager are discussed separately.

4. LEXICAL ANALYZER AND SYMBOL TABLE

4.1 Lexical Analyzer

The lexical analyzer is a duty routine for the syntactic recognizer. The lexical pointer LPTR contains in two contiguous bytes the page number (high order 6 bits specifing which 256-byte page) and byte number (low order 8 bits) of the 14 bit addresses within the user area, 14K through 16K-1. On entry to LEXI, LPTR points one byte beyond the most recently scanned token (initially SYNA points it to 14K+1, the first byte of user source text). LEXI isolates the next non-blank (possibly the current) character, sets SEM (semantic class variable) to the ASCII code of the character being examined if it is not an identifier or sets SEM to the variable's displacement in the symbol table if it is an identifier, and performs a pseudo binary search to identify the currently scanned symbol as one of 15 token classes. The token class is recorded in SYMB. If the token is an operator, LEXI records its F and G precedence (see Gries) in the variables F and G.

The token classes are:

TOKEN	ASCII	SYMB	SEM	F	G
LET	129	15	129		
DIM	130	15	130		
DEF	131	15	131		
REM	132	15	132		
GOTO	133	15	133		
IF	134	15	134		
FOR	135	15	135		

TOKEN	ASCII	SYMB	SEM	F	G
TO	136	15	136		
NEXT	138	15	138		
GOSUB	139	15	139		
RETURN	140	15	140		
INPUT	141	15	141		
PRINT	142	15	142		
STOP	143	15	143		
END	144	15	144		
STEP	145	15	145		
THEN	146	15	146		
NOT	147	3	147	12	13
AND	148	4	148	5	4
OR	149	4	149	3	2
FN	154	7	stp	7	6
SIN	155	6	155	7	6
ATN	156	6	156	7	6
COS	157	6	157	7	6
EXP	158	6	158	7	6
LOG	159	6	159	7	6
SQR	160	6	160	7	6
<	161	5	161	12	13
<=	162	5	162	12	13
>	163	5	163	12	13
>=	164	5	164	12	13
=	165	5	165	12	13
#	166	5	166	12	13
+	43	1	43	9	8
-	45	1	45	9	8
*	42	2	42	11	10
/	47	2	47	11	10
↑	94	2	94	11	10

TOKEN	ASCII	SYMB	SEM	F	G
(	40	8	40	0	14
)	41	9	41	0	0
"	34	10	34		
,	44	10	44		
end-of-line	92	10	92		
A	65	11	stp		
B	66	11	stp		
...					
Z	90	11	stp		
A(	65 40	12	stp		
B(	66 40	12	stp		
...					
Z(	90 40	12	stp		
0	48	14			
1	49	14			
...					
9	57	14			
others (unknown)		16			

stp = symbol table pointer

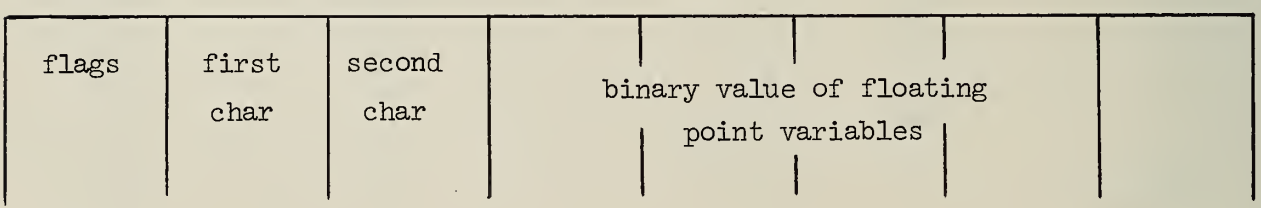
4.2 Symbol Table Manager

If a token is an identifier, a simple one character lookahead resolves the question of whether the identifier is an array or a simple scalar; likewise, after receiving the special FN key, the next character determines which function (FNA, FNB, ..., FNZ) is being referenced. If an identifier is referenced, LEXI calls the symbol table manager (LOOKUP) which, in turn, finds or inserts that identifier in the symbol table and returns the identifier's location as a displacement from the beginning of the symbol table; the displacement is returned in the variable SEM.

The symbol table search is based on a linear hash generated by ANDing the symbol's ASCII code with 31 ($=00011111_2$) and multiplying the result by 8 (three left shifts). The 26 alphabetic characters now hash uniquely to locations 8, 16, 24, ..., 208. Then a linear search is begun at the hash address HASH = 8 * mod(ASCII,32), checking each entry for one of three possibilities: (1) an empty entry (flag byte pointed to by HASH is zero), indicating that the desired entry is presently not in the table and causing insertion; (2) the variable name in the symbol table matches the recognized name in the B,C registers (SEM then returns the offset from STABH, the base address of the symbol table); or (3) the search fails 32 times and then signals an error - "symbol table full".

4.3 Symbol Table Structure

The structure of the symbol table is:



The flag byte is subdivided into its bit fields as:

$$b_7 b_6 b_5 b_4 b_3 b_2 b_1 b_0$$

where

b_0 = 0 entry empty
= 1 entry in use

b_7 = 0 not an array
= 1 variable is an array

b_6 = 0 not a user function
= 1 variable is a user function

SYNA uses b_5 to signal "defined" or undefined" during the parse. The remaining bits could be used individually or in combination to signal other possible data types (e.g., character strings) in a future improved version.

Single character scalar names are stored in the symbol table as [first char] [second char] = [letter] [blank], double character scalars as [letter] [digit], arrays as [letter] [(], and user functions as [FN] [letter]. Thus the type of any entry may be determined from either its flag bits or its name.

For arrays and user functions whose value is not stored in the symbol table, the 5 byte value field is used for other information. In particular, the entries look like:

	5 byte value				
scalar entry					
array entry	X	X	up_1	up_2	up_3
user function	high	low	disp	X	X

where: up_i is the i^{th} upper bound of the array, with all arrays internally having three dimensions whose default upper and lower bounds are zero;

(high,low) is an address pointer which marks the return address when the function has been evaluated;

disp is the symbol table displacement of the function's dummy parameter.

Note that the design of the symbol table allows the simultaneous use of the scalar A, the scalars A0, A1, ..., A9, the array A(, and the user function FNA with no ambiguity and no extra overhead in either LEXI or LOOKUP.

5. EXECUTION PACKAGE

No computing system intended for scientific use would be complete without true number-crunching ability. Since our BASIC system is designed for a whole range of applications, it seemed imperative that extensive floating point arithmetic capabilities be included. In an effort not to "reinvent the wheel," we decided to purchase the floating point arithmetic package utilized by the Datapoint 2200, a popular minicomputer whose CPU is similar to an Intel chip. The Datapoint software provides the means for floating point addition, subtraction, multiplication, division, and comparison, as well as both ASCII-character-string-to-internal-floating-point and internal-floating-point-to-ASCII-character-string conversion.

6. SUMMARY

The whole project, which extended for a period of approximately five months, successfully illustrated the feasibility of a BASIC interpreter running in a microprocessor environment. The basic design of the system is good, and serves well as a model for similar interactive systems.

It is regretable that time did not permit us to try our design on real-world hardware. We believe that the actual implementation of such a system might well be a project for future research.

APPENDIX A

BASIC LANGUAGE BNF

<basic program> ::= <program statement> | <basic program> <program statement>

<program statement> ::= <sequence number> <basic statement> (CR)

<sequence number> ::= <digit> $\{\text{<digit>}\}_0^2$

<basic statement> ::= <let statement> | <dim statement> | <def statement> | <rem statement> | <goto statement> | <if statement> | <for statement> | <next statement> | <gosub statement> | <return statement> | <input statement> | <print statement> | <stop statement> | <end statement>

<let statement> ::= (LET) <variable> (=) <formula>

<formula> ::= <conjunction> | <formula> (OR) <conjunction>

<conjunction> ::= <Boolean> | <conjunction> (AND) <Boolean>

<Boolean> ::= <expression> | <Boolean> <relative> <expression>

<expression> ::= <term> | <expression> (+ | -) <term>

<term> ::= <factor> | <term> (* | /) <factor>

<factor> ::= <primary> | <unary> <primary>

<primary> ::= <operand> | <primary> (↑) <operand>

<relative> ::= (>) | (<) | (>=) | (<=) | (=) | (#)

<operand> ::= <variable> | <number> | <function>

<variable> ::= <scalar variable> | <array variable>

<scalar variable> ::= <letter> [<digit>]

<array variable> ::= <letter> <subscript>

<subscript> ::= (<expression> $\{\text{, <expression>}\}_0^2$)

<letter> ::= A | B | C | ... | Z

<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

<unary> ::= + | - | NOT

<function> ::= <function name> (<expression>)

<function name> ::= FN <letter> | SIN | COS | LOG | SQR | EXP | ATN

<number> ::= [+ | -] {<digit>$^{+}$ [.] | <digit>* .<digit>$^{+}$}
[E[+ | -] <digit> [<digit>]]

<dim statement> ::= DIM <array list>

<array list> ::= <letter> $\underline{(}$ <upper bound> {, <upper bound>}$_{0}^{2}$ $\underline{)}$
{,<letter> $\underline{(}$ <upper bound> {,<upper bound>}$_{0}^{2}$ }$^{+}$

<upper bound> ::= <number> such that 0 $\leq$ number $\leq$ 255

<statement> ::= DEF FN <letter> $\underline{(}$ <letter> $\underline{)}$ (=) <formula>

<rem statement> ::= REM <character string> CR

<goto statement> ::= GOTO <sequence number>

<sequence number> ::= <number> such that 0 $\leq$ number $\leq$ 999

<if statement> ::= IF <formula> THEN <sequence number>

<for statement> ::= FOR <scalar variable> (=) <formula> TO <formula>
[STEP <formula>]

<next statement> ::= NEXT <scalar variable>

<gosub statement> ::= GOSUB <sequence number>

<return statement> ::= RETURN

<input statement> ::= INPUT <variable> {, <variable> }*

<print statement> ::= PRINT <print list> {, <print list>}*

<print list> ::= "<character string>" | <formula>

<character string> ::= any TTY character except "

<stop statement> ::= STOP

<end statement> ::= END

APPENDIX B

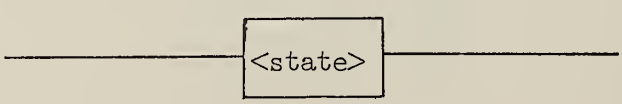
DESCRIPTION OF THE STATE TRANSITIONS FOR THE SYNTACTIC RECOGNIZER

The parser that is used in our BASIC system is essentially a one stack push down automata; the parsing technique is top-down goal oriented with the stack used for state sub-calls and recursion book-keeping. In the following discussion the abbreviations used are:

SYMB: a token's symbol class, i.e., identifier, reserved word, etc.

SEM: a token's concrete representation, i.e., "+", "IF", etc.

In order to describe the parser's state transitions a few state transition diagram conventions or notations have been developed. A transition from one state to another will occur at least once for every new token from the input string. Each state in the diagram will have at least one branch coming out of it. These branches are usually labeled with the conditions necessary for that branch to be taken. An unlabeled branch is taken as a last resort. If none of the branches is satifsfied, it indicates an error condition. If a branch has more than one condition on it, the conditions are each placed on a separate branch and the branches are connected together; to reach the end of a branch, all intermediate branches must be satisfied. The end of a branch sequence contains the state transition information. Usually, just the "next state to move into" is indicated; sometimes a box such as



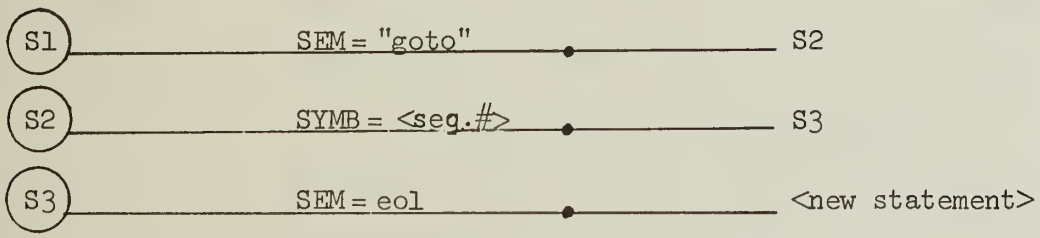
may appear at the end of a branch; this indicates an intermediary "state call" must take place before the transition to the "next state" occurs (in other words, save the "next state" on the parser stack and make the state transition to the state in the box). Likewise, sometimes a symbol such as

may occur in a branch; this indicates the state transition is to the state at the top of the stack.

For example, examine the BASIC GOTO statement:

<goto> ::= GOTO <seq.#> EOL

The state transition diagram for this statement is:



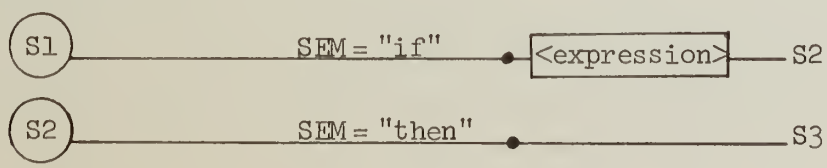
This is interpreted as:

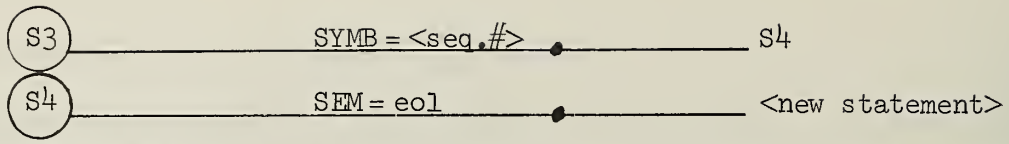
> If you are in state S1 and the current token is a "GOTO", then move into state S2. If you are in S2 and the current token class is <seq.#>, move into state S3. If you are in state S3, and the current token is "EOL" (end of line), move to the state for the beginning of a new statement. Note that in all cases here, if the given condition is not met, there is an error.

As another example, consider the IF statement:

<if> ::= IF <expression> THEN <seq.#> EOL

The transition diagram for this is:





This is interpreted as:

> If you are in state S1 and the current token is an "if", then push S2 onto the parser stack and make the transition to state <expression> (although not shown, state <expression> will parse an expression as defined by BASIC and, when finished, will make the transition to the state at the top of the stack, namely S2, and then pop the stack). The rest of the diagram is interpreted similarily.

This concludes the discussion of the state transitions for our system.

The table system

Given the state transition diagrams for a language, the remaining problem is to encode the information and actions of the diagrams into a table format. To succeed, the table must contain fields for:

1) Checking the branch conditions.
   (a) Checking the SYMB or SEM values of the current token
   (b) Ability to check efficiently for all the branches of a state
   (c) Ability to detect an error condition
   (d) Ability to check multiple conditions, i.e., more than one branch condition to reach the end of the branch

2) Stack operations.
   (a) Push a state onto the parser stack
   (b) Pop a state off the parser stack and make the transition to that state

3) Normal "next state" transitions.

4) Since taking a branch normally means accepting the current token and moving on to the next token in the input string, there must be someway of making an 'epsilon move', i.e., changing the state while not moving the input pointer.

5) In this system we wanted to be able to have the table drive the actual execution of the statement as well as just syntactically parse it. Also, since the system will keep up with the programmer (FOR-loop nesting checks, defined functions and arrays) even during simple parsing mode, the table needs to indicate what routines to perform and when they should be performed.

With these requirements in mind, we describe the actual table system. The system is divided into two parts, the table itself and a routine to interpret the table. Consider the table to be actually a large vector, where each element of the vector is one "entry" in the table. There are four types of entries in the table: HEADER, VALUE, CONTROL, and ERROR.

HEADER

The HEADER entry contains information about the next few entries in the table. It tells what tests to make, how many of them to make, and what to do if they all fail. In particular, there are three fields in the entry:

TYPE (2 bits): The TYPE field contains a number (0 - 3) that identifies the type of branches for that state.

QUAN (4 bits): The QUAN field contains the number of branches for the state.

FAIL (2 bits): The FAIL field contains a number indicating what to do if all the branch tests fail.

These fields are discussed in more detail on page 29.

VALUE

The VALUE entry contains only one field. This field holds a number which is to be compared with either the current SYMB or SEM token value, depending on the previous HEADER that was processed. If the field's number matches the current token properly, then the correct branch has been

located for this state and the next CONTROL entry should be interpreted. On the other hand, if the field's number did not match the token, then this branch has failed; in this case, the previous HEADER information will determine what table entry to look at next.

CONTROL

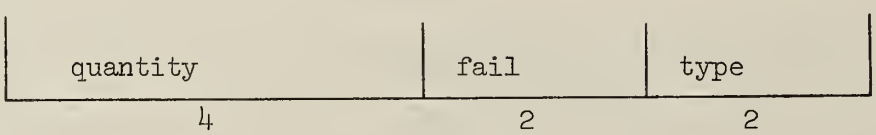
The CONTROL entry is interpreted only when a correct branch has been located for a state. The CONTROL entry corresponds to "the end" of a branch in the state transition diagram. It contains fields for Push state, POP, Nextstate, whether more conditions must be checked, whether this is an epsilon move, and also an execution/parse routine number, along with three bits for the the mode checking - one for each mode: unparse, parse, execute. If the mode bit in the table that corresponds to the actual mode of the system is set, the routine is called. In all cases, if a field value is 0, then that option is ignored when the CONTROL entry is interpreted.

ERROR

The ERROR entry is interpreted when all the branches have failed for a state and the HEADER's FAIL field equals 1, indicating an error exists. The ERROR entry contains an error number, which can be extracted and passed on to the error display routine.

These are the entries in the table. The detailed bit implementation follows:

HEADER



TYPE

The TYPE field determines exactly what branch conditions must be checked for and what the order of the next entries in the table is.

type = 0 Unconditional TRUE branch. The next table entry is a CONTROL entry that is immediately interpreted.

type = 1 Branch condition: SYMB value check.
The following two table entries consist of a VALUE entry followed by a CONTROL entry; the current token's SYMB value is compared to the VALUE table entry; if they are equal, then the CONTROL entry is interpreted; if unequal, then the CONTROL entry is jumped over in the table and different tests can be made (see fail).

type = 2 Branch condition: SEM value check.
Same form as type = 1.

type = 3 Special case test. QUAN contains routine number. CONTROL follows.

QUAN

The QUAN field determines how many similar branches a state has. It gives the number of VALUE/CONTROL pair entries following the HEADER. For type = 3, the QUAN value is a special case routine number.

FAIL

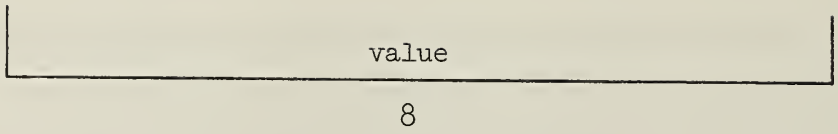
The FAIL field is used only when all the other checks have failed.

fail = 0 The next entry is a new HEADER with a new set of condition checks. It is "flowed" into.

fail = 1 The next entry is an ERROR entry.

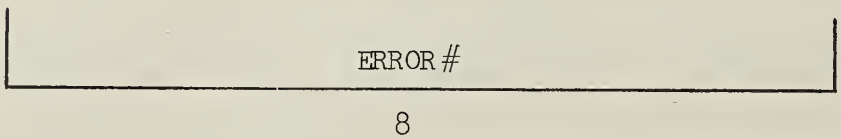
fail = 2 The next entry is an alternate CONTROL entry that should be immediately interpreted.

VALUE



Contains one number to be used according to the previous HEADER information.

ERROR



Contains an error number that is passed to the error message writer when an error is detected.

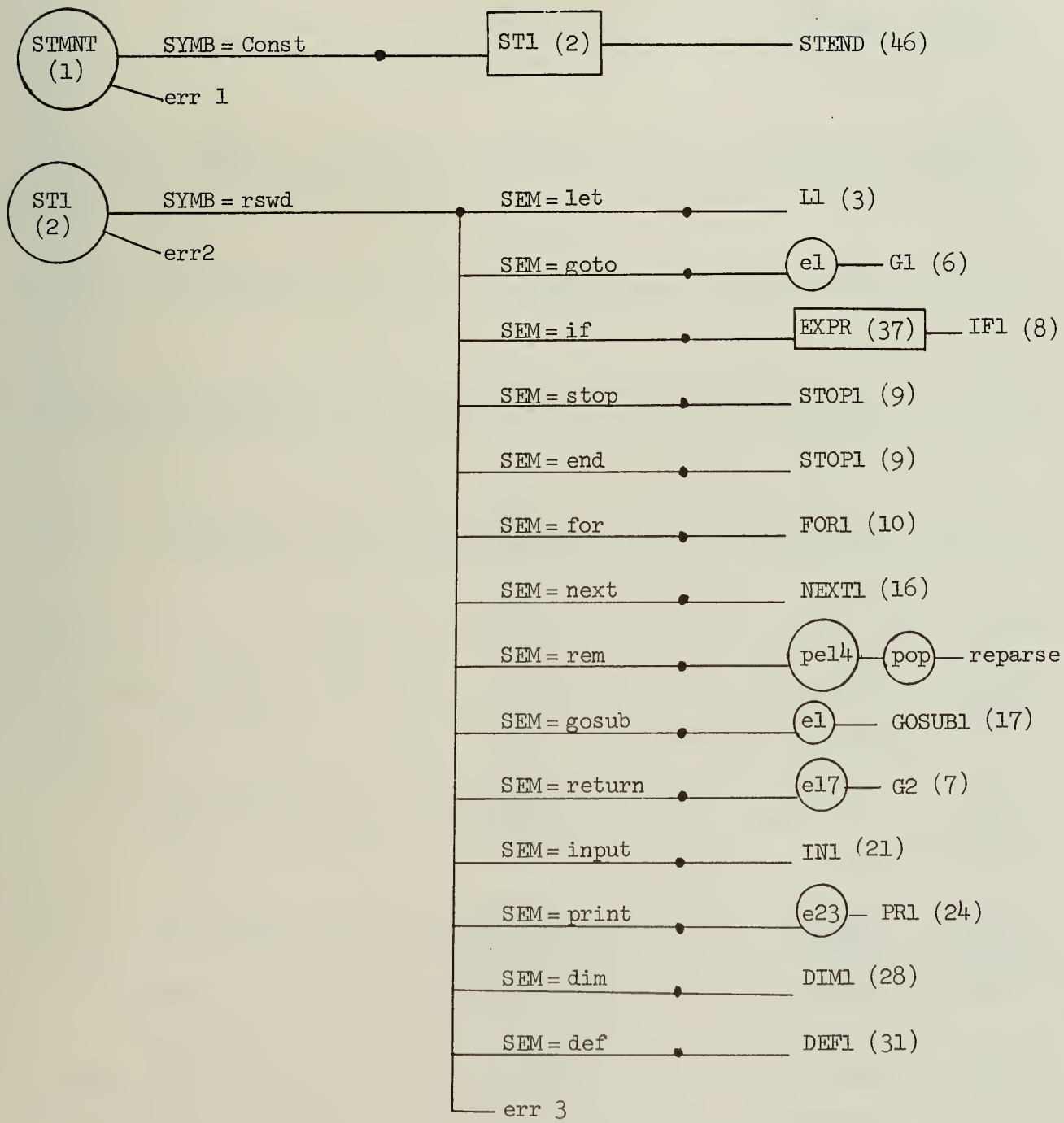
CONTROL entry: 18 bits

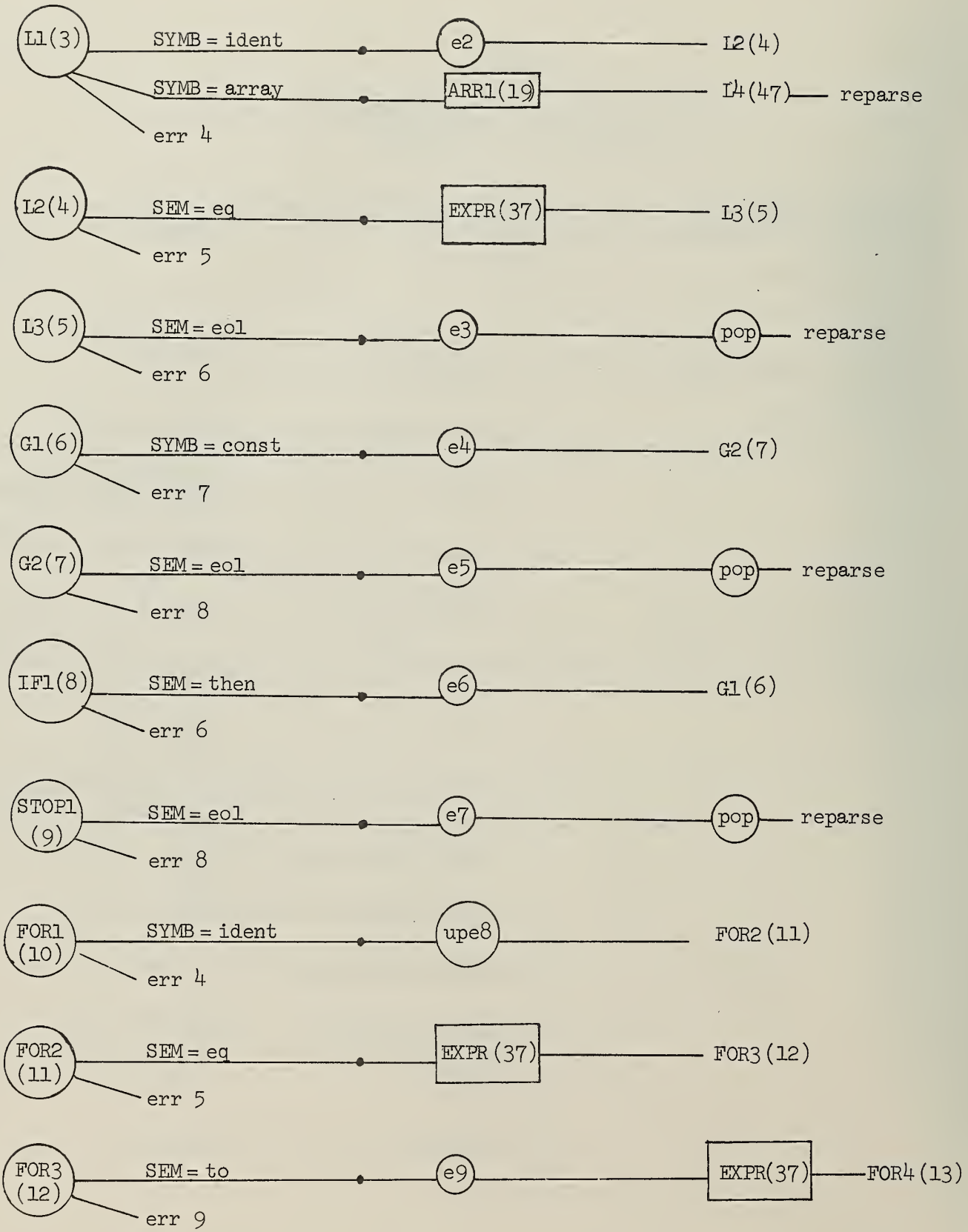
Unparse mode bit	POP	Push state	Execution Parse Routine#	Mode bits		End of branch bit	Epsilon move reparse	Next state
				Parse	Execute			
1	1	6	6	1	1	1	1	6

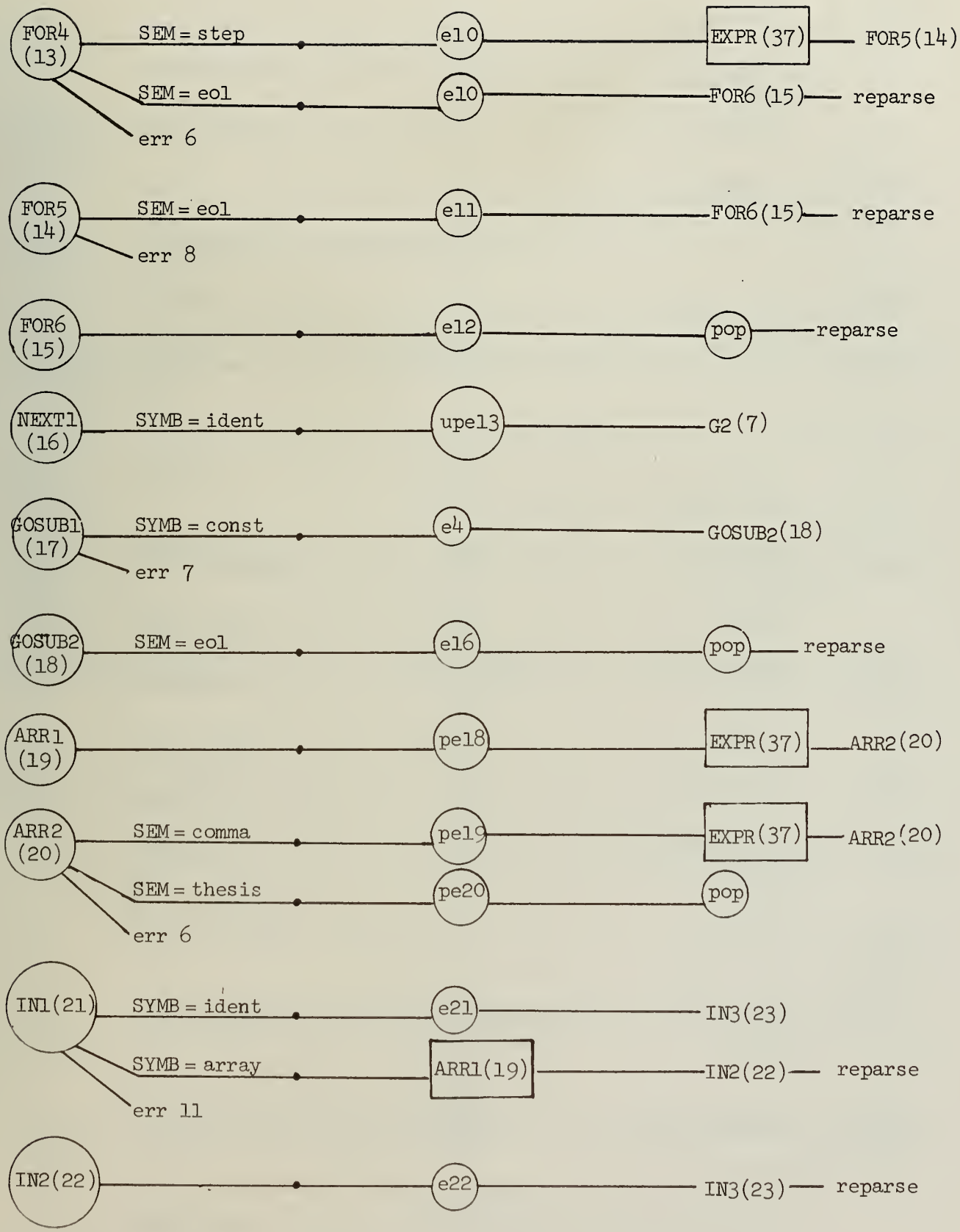
The CONTROL entry is divided into three bytes. With the exception of the Unparse bit, all the fields are interpreted in order as they come. For the Execution/Parse/Unparse routine, the Unparse bit is ORed into the third bit position of a byte, and the other mode bits are ORed into the first two bit positions. Then this byte is compared with the current mode switch (which has values: Unparse = 4, Parse = 2, Execute = 1). If the corresponding bit is set the execution routine is called.

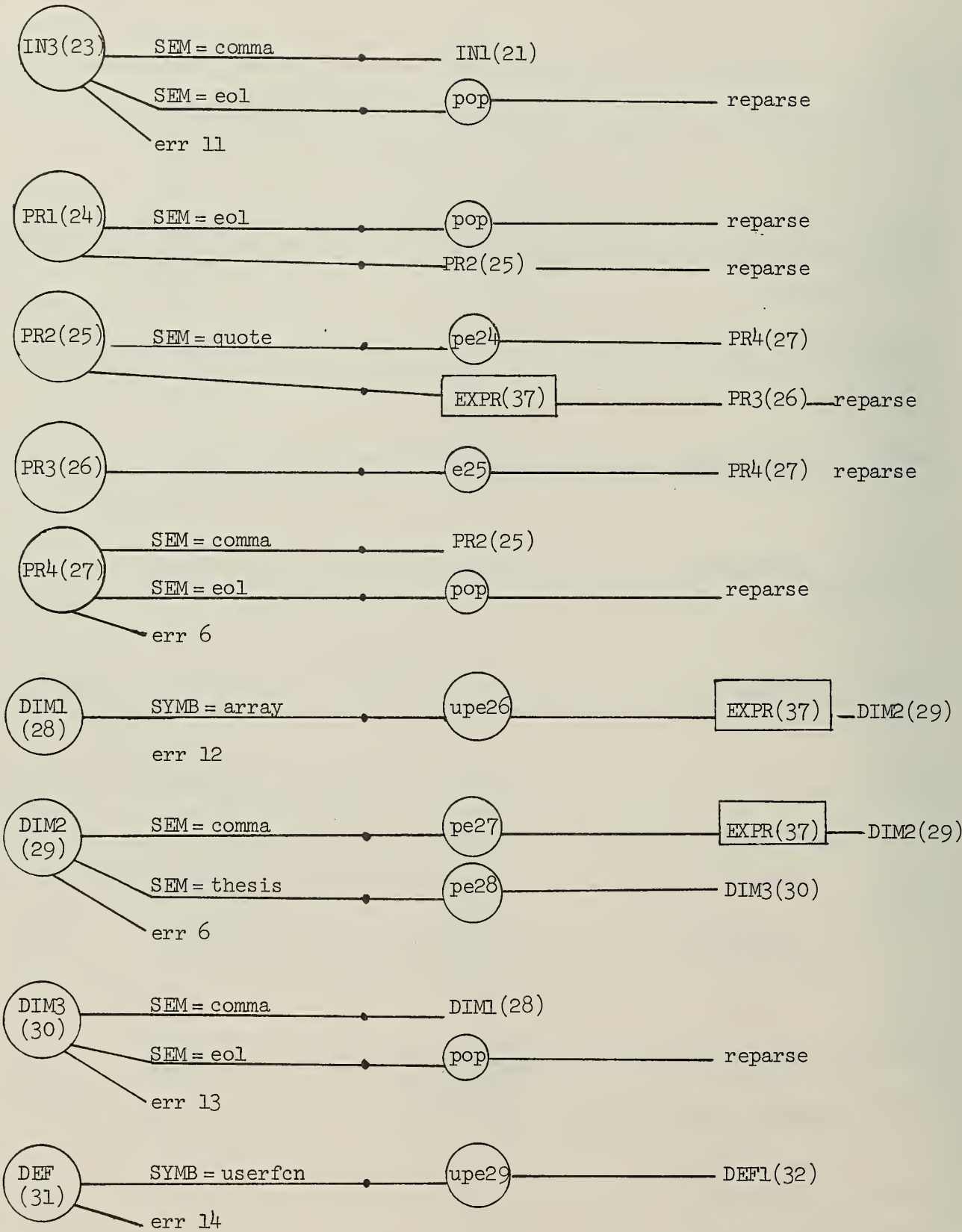
APPENDIX C

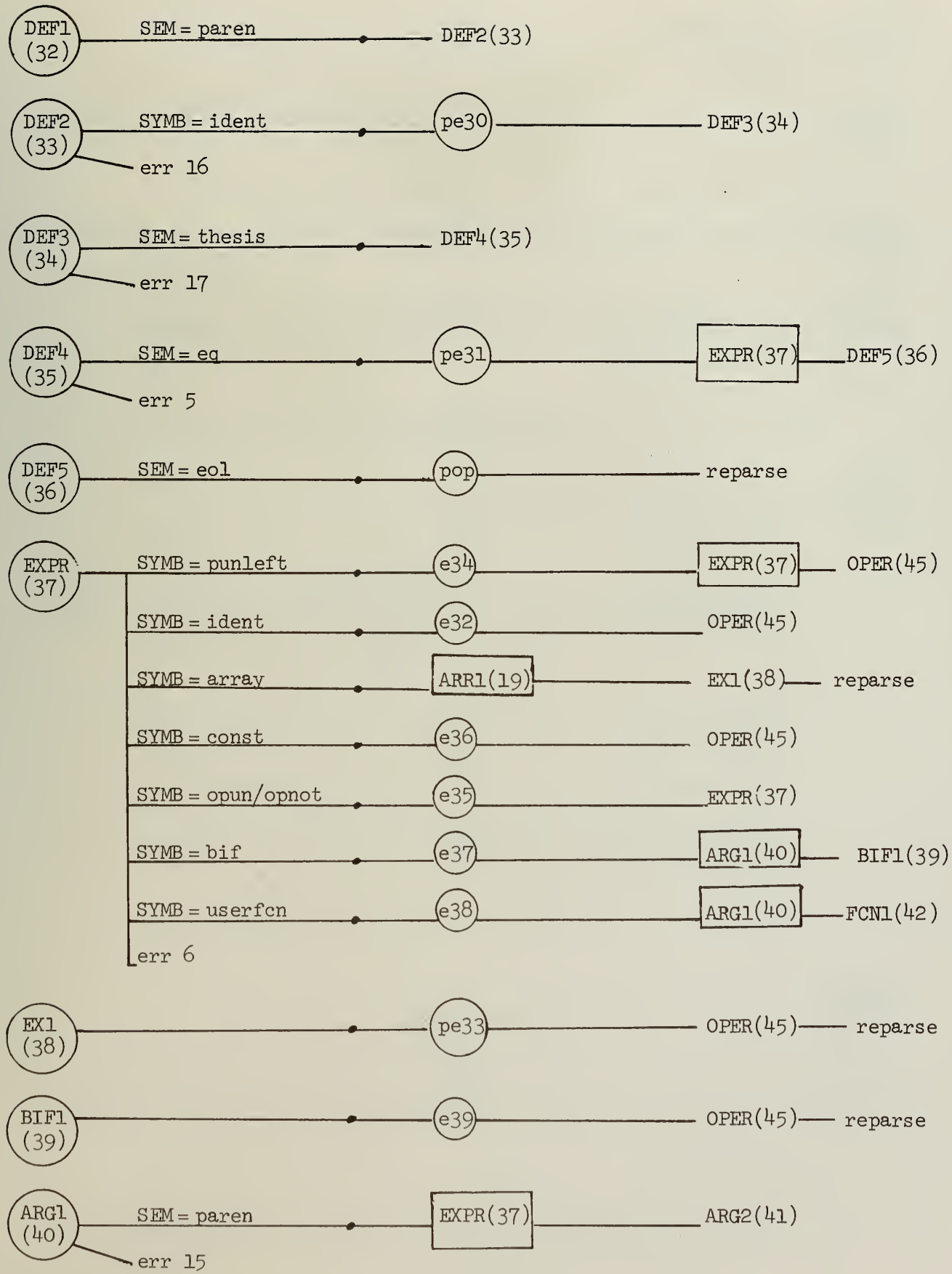
STATE DIAGRAMS FOR BASIC LANGUAGE TABLE-DRIVEN INTERPRETER

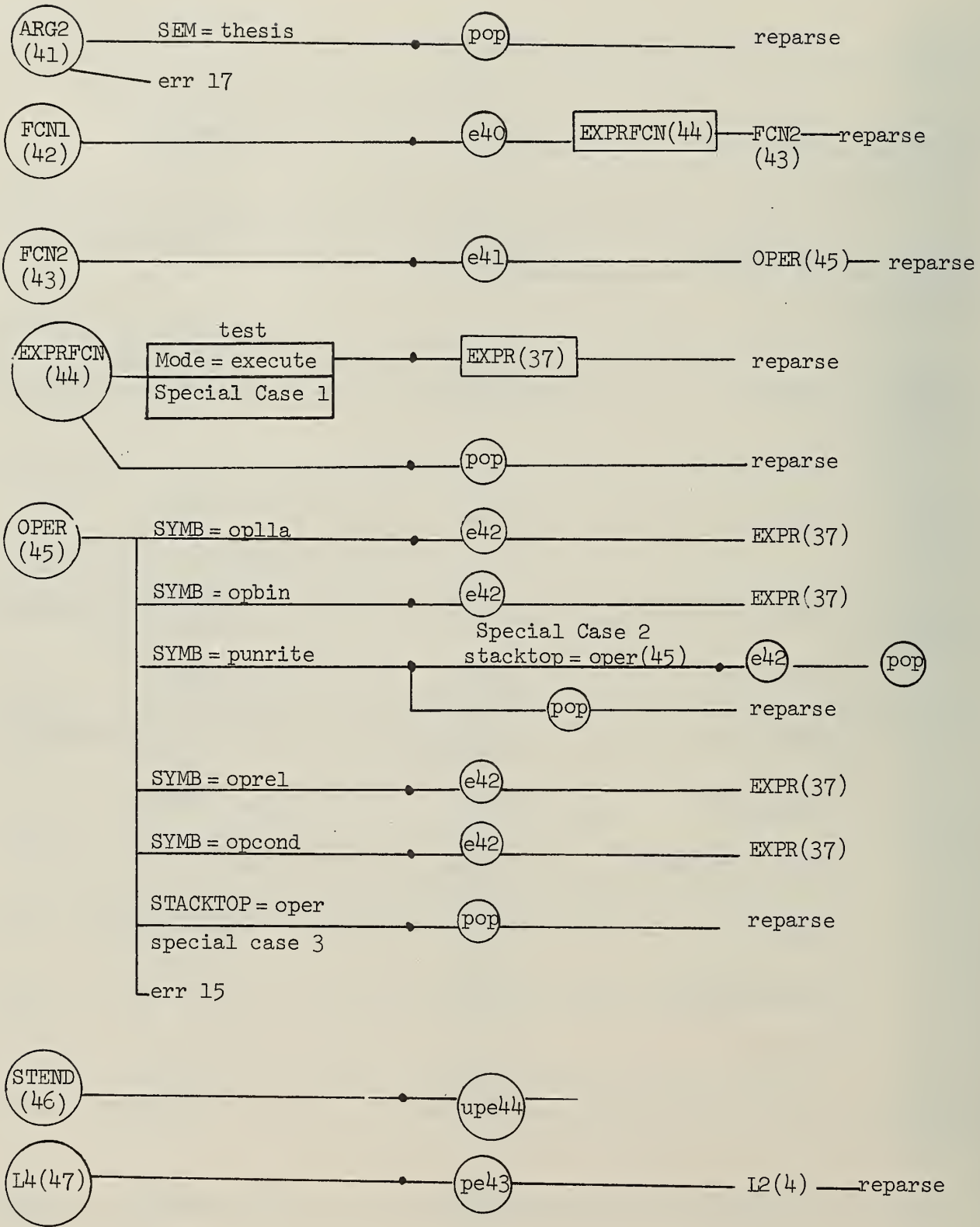



L1(3)
SYMB = ident
e2
I2(4)
SYMB = array
ARR1(19)
L4(47)
reparse
err 4
L2(4)
SEM = eq
EXPR(37)
L3(5)
err 5
L3(5)
SEM = eol
e3
pop
reparse
err 6
G1(6)
SYMB = const
e4
G2(7)
err 7
G2(7)
SEM = eol
e5
pop
reparse
err 8
IF1(8)
SEM = then
e6
G1(6)
err 6
STOP1 (9)
SEM = eol
e7
pop
reparse
err 8
FOR1 (10)
SYMB = ident
upe8
FOR2(11)
err 4
FOR2 (11)
SEM = eq
EXPR(37)
FOR3(12)
err 5
FOR3 (12)
SEM = to
e9
EXPR(37)
FOR4(13)
err 9


FOR4 (13)
SEM = step
e10
EXPR (37)
FOR5(14)
SEM = eol
e10
FOR6 (15)
reparse
err 6
FOR5 (14)
SEM = eol
e11
FOR6(15)
reparse
err 8
FOR6 (15)
e12
pop
reparse
NEXT1 (16)
SYMB = ident
upe13
G2(7)
GOSUB1 (17)
SYMB = const
e4
GOSUB2(18)
err 7
GOSUB2 (18)
SEM = eol
e16
pop
reparse
ARR1 (19)
pe18
EXPR(37)
ARR2(20)
ARR2 (20)
SEM = comma
pe19
EXPR(37)
ARR2(20)
SEM = thesis
pe20
pop
err 6
IN1(21)
SYMB = ident
e21
IN3(23)
SYMB = array
ARR1(19)
IN2(22)
reparse
err 11
IN2(22)
e22
IN3(23)
reparse


IN3(23)
SEM = comma
IN1(21)
SEM = eol
pop
reparse
err 11
PR1(24)
SEM = eol
pop
reparse
PR2(25)
reparse
PR2(25)
SEM = quote
pe24
PR4(27)
EXPR(37)
PR3(26)
reparse
PR3(26)
e25
PR4(27)
reparse
PR4(27)
SEM = comma
PR2(25)
SEM = eol
pop
reparse
err 6
DIM1 (28)
SYMB = array
upe26
EXPR(37)
DIM2(29)
err 12
DIM2 (29)
SEM = comma
pe27
EXPR(37)
DIM2(29)
SEM = thesis
pe28
DIM3(30)
err 6
DIM3 (30)
SEM = comma
DIM1(28)
SEM = eol
pop
reparse
err 13
DEF (31)
SYMB = userfcn
upe29
DEF1(32)
err 14


DEF1 (32)
SEM = paren
DEF2(33)
DEF2 (33)
SYMB = ident
pe30
DEF3(34)
err 16
DEF3 (34)
SEM = thesis
DEF4(35)
err 17
DEF4 (35)
SEM = eq
pe31
EXPR(37)
DEF5(36)
err 5
DEF5 (36)
SEM = eol
pop
reparse
EXPR (37)
SYMB = punleft
e34
EXPR(37)
OPER(45)
SYMB = ident
e32
OPER(45)
SYMB = array
ARR1(19)
EX1(38)
reparse
SYMB = const
e36
OPER(45)
SYMB = opun/opnot
e35
EXPR(37)
SYMB = bif
e37
ARG1(40)
BIF1(39)
SYMB = userfcn
e38
ARG1(40)
FCN1(42)
err 6
EX1 (38)
pe33
OPER(45)
reparse
BIF1 (39)
e39
OPER(45)
reparse
ARG1 (40)
SEM = paren
EXPR(37)
ARG2(41)
err 15


ARG2 (41)
SEM = thesis
pop
reparse
err 17
FCN1 (42)
e40
EXPRFCN(44)
FCN2 (43)
reparse
FCN2 (43)
e41
OPER(45)
reparse
EXPRFCN (44)
test
Mode = execute
Special Case 1
EXPR(37)
reparse
pop
reparse
OPER (45)
SYMB = oplla
e42
EXPR(37)
SYMB = opbin
e42
EXPR(37)
SYMB = punrite
Special Case 2
stacktop = oper(45)
e42
pop
pop
reparse
SYMB = oprel
e42
EXPR(37)
SYMB = opcond
e42
EXPR(37)
STACKTOP = oper
special case 3
pop
reparse
err 15
STEND (46)
upe44
L4(47)
pe43
L2(4)
reparse

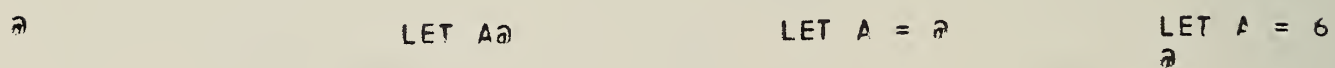
APPENDIX D

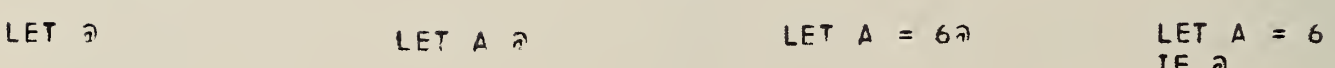
PROGRAM EXAMPLE

The following pages show the successive (simulated) states of the TV screen in response to various keyboard commands issued in the text-editing mode.

The simulated TV screens below show the successive states of the visual image as it responds to insertion, deletion, replacement, and cursor movement under control of the BASIC text editor.

The screens are presented in columns, and should be read top to bottom and left to right.





```
LET A = 6
IF A @
```

```
LET A = 6
IF A = 7 THEN @
```

```
LET A = 6
IF A = 7 THEN P = Q@
```

```
LET A = 6
IF A = @
```

```
LET A = 6
IF A = 7 THEN P@
```

```
LET A = 6
IF A = 7 THEN P = Q
@
```

```
LET A = 6
IF A = 7@
```

```
LET A = 6
IF A = 7 THEN P @
```

```
LET A = 6
IF A = 7 THEN P = 0
D@
```

```
LET A = 6
IF A = 7 @
```

```
LET A = 6
IF A = 7 THEN P = @
```

```
LET A = 6
IF A = 7 THEN P = 0
DE@
```

```
LET A = 6
IF A = 7 THEN P = 0
DEF@
```

```
LET A = 6
IF A = 7 THEN P = 0
DEFINE A(@
```

```
LET A = 6
IF A = 7 THEN P = 0
DEFINE A(1,2)@
```

```
LET A = 6
IF A = 7 THEN P = 0
DEFI@
```

```
LET A = 6
IF A = 7 THEN P = 0
DEFINE A(1@
```

```
LET A = 6
IF A = 7 THEN P = 0
DEFINE A(1,2)
@
```

```
LET A = 6
IF A = 7 THEN P = 0
DEFIN@
```

```
LET A = 6
IF A = 7 THEN P = 0
DEFINE A(1,@
```

```
LET A = 6
IF A = 7 THEN P = 0
@DEFINE A(1,2)
```

```
LET A = 6
IF A = 7 THEN P = 0
DEFINE@
```

```
LET A = 6
IF A = 7 THEN P = 0
DEFINE A(1,2@
```

```
LET A = 6
IF A = 7 THEN P = 0
D@EFINE A(1,2)
```

```
LET A = 6
IF A = 7 THEN P = Q
DE@FINE A(1,2)
```

```
LET A = 6
IF A = 7 THEN P = Q
DEF@ A(1,2)
```

```
LET A = 6
IF A = 7 THEN P = Q
@EF A(1,2)
```

```
LET A = 6
IF A = 7 THEN P = Q
DEF@INE A(1,2)
```

```
LET A = 6
IF A = 7 THEN P = Q
DE@F A(1,2)
```

```
LET A = 6
IF A = 7 THEN P = Q
@F A(1,2)
```

```
LET A = 6
IF A = 7 THEN P = Q
DEF@NE A(1,2)
```

```
LET A = 6
IF A = 7 THEN P = Q
D@EF A(1,2)
```

```
LET A = 6
IF A = 7 THEN P = Q
@ A(1,2)
```

```
LET A = 6
IF A = 7 THEN P = Q
DFF@E A(1,2)
```

```
LET A = 6
IF A = 7 THEN P = Q
@DEF A(1,2)
```

```
LET A = 6
IF A = 7 THEN P = Q
@ A(1,2)
```

```
LET A = 6
IF A = 7 THEN P = 0
DEF @ A(1,2)
```

```
LET A = 6
IF A = 7 THEN P = 0
DEF  A(1,2)
THI@
```

```
LET A = 6
IF A = 7 THEN P = 0
DEF  A(1,2)
THIS LI@
```

```
LET A = 6
IF A = 7 THEN P = 0
DEF  A(1,2)
@
```

```
LET A = 6
IF A = 7 THEN P = 0
DEF  A(1,2)
THIS@
```

```
LET A = 6
IF A = 7 THEN P = 0
DEF  A(1,2)
THIS LIN@
```

```
LET A = 6
IF A = 7 THEN P = 0
DEF  A(1,2)
T@
```

```
LET A = 6
IF A = 7 THEN P = 0
DEF  A(1,2)
THIS @
```

```
LET A = 6
IF A = 7 THEN P = 0
DEF  A(1,2)
THIS LINE@
```

```
LET A = 6
IF A = 7 THEN P = 0
DEF  A(1,2)
TH@
```

```
LET A = 6
IF A = 7 THEN P = 0
DEF  A(1,2)
THIS L@
```

```
LET A = 6
IF A = 7 THEN P = 0
DEF  A(1,2)
THIS LINE @
```

```
LET A = 6
IF A = 7 THEN P = Q
DEF  A(1,2)
THIS LINE W@
```

```
LET A = 6
IF A = 7 THEN P = Q
DEF  A(1,2)
THIS LINE WILL @
```

```
LET A = 6
IF A = 7 THEN P = Q
DEF  A(1,2)
THIS LINE WILL BE D@
```

```
LET A = 6
IF A = 7 THEN P = Q
DEF  A(1,2)
THIS LINE WI@
```

```
LET A = 6
IF A = 7 THEN P = Q
DEF  A(1,2)
THIS LINE WILL B@
```

```
LET A = 6
IF A = 7 THEN P = Q
DEF  A(1,2)
THIS LINE WILL BE DE@
```

```
LET A = 6
IF A = 7 THEN P = Q
DEF  A(1,2)
THIS LINE WIL@
```

```
LET A = 6
IF A = 7 THEN P = Q
DEF  A(1,2)
THIS LINE WILL BE@
```

```
LET A = 6
IF A = 7 THEN P = Q
DEF  A(1,2)
THIS LINE WILL BE DEL@
```

```
LET A = 6
IF A = 7 THEN P = Q
DEF  A(1,2)
THIS LINE WILL@
```

```
LET A = 6
IF A = 7 THEN P = Q
DEF  A(1,2)
THIS LINE WILL BE @
```

```
LET A = 6
IF A = 7 THEN P = Q
DEF  A(1,2)
THIS LINE WILL BE DELE@
```

```
LET A = 6
IF A = 7 THEN P = Q
DEF  A(1,2)
THIS LINE WILL BE DELET@
```

```
LET A = 6
IF A = 7 THEN P = Q
DEF  A(1,2)
THIS LINE WILL BE DELETED
R@
```

```
LET A = 6
IF A = 7 THEN P = Q
DEF  A(1,2)
THIS LINE WILL BE DELETED@
```

```
LET A = 6
IF A = 7 THEN P = Q
DEF  A(1,2)
THIS LINE WILL BE DELETED
RE@
```

```
LET A = 6
IF A = 7 THEN P = Q
DEF  A(1,2)
THIS LINE WILL BE DELETED@
```

```
LET A = 6
IF A = 7 THEN P = Q
DEF  A(1,2)
THIS LINE WILL BE DELETED
REP@
```

```
LET A = 6
IF A = 7 THEN P = Q
DEF  A(1,2)
THIS LINE WILL BE DELETED
@
```

```
LET A = 6
IF A = 7 THEN P = Q
DEF  A(1,2)
THIS LINE WILL BE DELETED
REPL@
```

```
LET A = 6
IF A = 7 THEN P = Q
DEF  A(1,2)
THIS LINE WILL BE DELETED
REPLA@
```

```
LET A = 6
IF A = 7 THEN P = Q
DEF  A(1,2)
THIS LINE WILL BE DELETED
REPLACE R@
```

```
LET A = 6
IF A = 7 THEN P = Q
DEF  A(1,2)
THIS LINE WILL BE DELETED
REPLAC@
```

```
LET A = 6
IF A = 7 THEN P = Q
DEF  A(1,2)
THIS LINE WILL BE DELETED
REPLACE P @
```

```
LET A = 6
IF A = 7 THEN P = Q
DEF  A(1,2)
THIS LINE WILL BE DELETED
REPLACE@
```

```
LET A = 6
IF A = 7 THEN P = Q
DEF  A(1,2)
THIS LINE WILL BE DELETED
REPLACE R W@
```

```
LET A = 6
IF A = 7 THEN P = Q
DEF  A(1,2)
THIS LINE WILL BE DELETED
REPLACE @
```

```
LET A = 6
IF A = 7 THEN P = Q
DEF  A(1,2)
THIS LINE WILL BE DELETED
REPLACE R WI@
```

```
LET A = 6
IF A = 7 THEN P = Q
DEF  A(1,2)
THIS LINE WILL BE DELETED
REPLACE R WITH @
```

```
LET A = 6
IF A = 7 THEN P = Q
DEF  A(1,2)
THIS LINE WILL BE DELETED
REPLACE R WITH  L@
```

```
LET A = 6
IF A = 7 THEN P = Q
DEF  A(1,2)
THIS LINE WILL BE DELETED
REPLACE R WITH@
```

```
LET A = 6
IF A = 7 THEN P = Q
DEF  A(1,2)
THIS LINE WILL BE DELETED
REPLACE R WITH  L @
```

```
LET A = 6
IF A = 7 THEN P = Q
DEF  A(1,2)
THIS LINE WILL BE DELETED
REPLACE R WITH @
```

```
LET A = 6
IF A = 7 THEN P = Q
DEF  A(1,2)
THIS LINE WILL BE DELETED
REPLACE R WITH  L  @
```

```
LET A = 6
IF A = 7 THEN P = Q
DEF  A(1,2)
THIS LINE WILL BE DELETED
REPLACE R WITH  @
```

```
LET A = 6
IF A = 7 THEN P = Q
DEF  A(1,2)
THIS LINE WILL BE DELETED
REPLACE R WITH  L  R@
```

```
LET A = 6
IF A = 7 THEN P = 0
DEF  A(1,2)
THIS LINE WILL BE DELETED
REPLACE R WITH  L  P
@
```

```
LET A = 6
IF A = 7 THEN P = Q
DEF  A(1,2)
THIS LINE WILL BE DELETED
REPLACE R WITH  L  L@
```

```
LET A = 6
IF A = 7 THEN P = Q
DEF  A(1,2)
THIS LINE WILL BE DELETED
REPLACE R WITH  L  R@
```

```
LET A = 6
IF A = 7 THEN P = Q
DEF  A(1,2)
THIS LINE WILL BE DELETED
@REPLACE R WITH  L  L
```

```
LET A = 6
IF A = 7 THEN P = Q
DEF  A(1,2)
THIS LINE WILL BE DELETED
REPLACE R WITH  L  @R
```

```
LET A = 6
IF A = 7 THEN P = 0
DEF  A(1,2)
@THIS LINE WILL BE DELETED
REPLACE R WITH  L  L
```

```
LET A = 6
IF A = 7 THEN P = 0
DEF  A(1,2)
THIS LINE WILL BE DELETED
REPLACE P WITH  L  @R
```

```
LET A = 6
IF A = 7 THEN P = 0
DEF  A(1,2)
@REPLACE R WITH  L  L
```

```
LET A = 6
IF A = 7 THEN P = Q
DEF  A(1,2)
REPLACE R WITH  L  L
a
```


```
LET A = 6
IF A = 7 THEN P = Q
DEF  A(1,2)
REPLACE R WITH  L  L
a
```

BIBLIOGRAPHIC DATA SHEET	1. Report No. UIUCDCS-R-74-658	2.	3. Recipient's Accession No.

4. Title and Subtitle
A BASIC Language Interpreter for the Intel 8008 Microprocessor

5. Report Date
June 1974

6.

7. Author(s)
Weaver, A.C., Tindall, M.H., & Danielson, R.L.

8. Performing Organization Rept. No. UIUCDCS-R-74-658

9. Performing Organization Name and Address
Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, Illinois 61801

10. Project/Task/Work Unit No.

11. Contract/Grant No.

12. Sponsoring Organization Name and Address
Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, Illinois 61801

13. Type of Report & Period Covered

14.

15. Supplementary Notes

16. Abstracts

A BASIC language interpreter has been designed for use in a microprocessor environment. This report discusses the development of 1) an elaborate text editor and 2) a table-driven interpreter. The entire system, including text editor, interpreter, user text buffer, and full floating point arithmetic routines fits in 16K 8-bit words.

17. Key Words and Document Analysis. 17a. Descriptors

BASIC language
interpreter
microprocessor
text editor

17b. Identifiers Open-Ended Terms

17c. COSATI Field/Group

18. Availability Statement	19. Security Class (This Report)	21. No. of Pages
release unlimited	UNCLASSIFIED	48
	20. Security Class (This Page) UNCLASSIFIED	22. Price no charge