AN EXPERIMENT IN THE DESIGN OF A BASIC INTERPRETER

Hal Pitts
Graduate Student
Department of Information Systems
Georgia State University
Atlanta, GA 30303

Abstract:

This paper describes the author's experience on the design and implementation of a BASIC interpretive compiler written in Algol W programming language. The experiment was initiated to gain full control of an executing BASIC program for run time debugging and automating the grading of students programs in beginning BASIC programming classes. The interpreter is implemented in Algol W programming language. The block structure and the control structures of Algol W provided a convenient environment for designing a truely structured program. The powerful data structures facilities available in Algol W gave the author better control of the program. The input/output facilities of the Algol W version at Georgia State University simplified the task of internal/external numerical conversions. In the opinion of this author, Algol W is an excellent language for debugging and writing interpretive compilers. The BASIC interpreter operates in two loosely connected phases: a parsing phase and an interpretive phase. The parsing phase analyzes the complete source program and translates the input program into pseudo machine-like code. Major tasks of this phase are: Allocation of memory locations to variables and constants, conversion of expressions into Reverse Polish form, and the generation of pseudo machine code. The interpretive phase executes the pseudo code simulating a BASIC machine. The pseudo machine code was designed to minimize the time needed to decode and execute it.

1. Introduction

In a University teaching environment there is a need to alleviate the routine work associated with teaching introductory programming classes. With an enrollment of approximately 300 students each quarter in the introductory programming courses in BASIC at Georgia State University (GSU), a method of grading students programs was sought to relieve the instructors of this task. It was also felt that some form of run-time debugging would be helpful to students in debugging their programming assignments. The first component to meet these goals was the author's experiment in the design of a BASIC interpretive compiler written in the Algol W programming language. With this interpretive compiler it was felt that we could gain full control of an executing BASIC program for run-time debugging and automating the grading of students' programs at a later time.

Inspite of renewed interest in interpretive compilers, triggered by microprocessor technology, this author discovered that very little has been reported in the literature concerning the construction of interpretive compilers. The author used the University of Washington BASIC Interpretive Compiler [1] and Gauthier and Ponto's [2] description of interpreters as a guide to his design. The BASIC language implemented is similar to Lee's [3] formal description of BASIC.

2. Salient features of Algol W programming language:

The Algol W programming language was chosen for implementing our BASIC interpretive compiler because of the ease of program formulation. The Algol type block and control structures made the task almost elementary in that the logical divisions could be separated without the use of numerous program branches. This created a convenient environment for desinging a truely structured program. Recursion was helpful in that some of the interpreter's modules were easily implemented as recursive routines. Algol W's numerous data types also gave flexibility to the resulting BASIC interpretive compiler since data types could be transformed into a different representation for variable translation and the indexing into arrays. The input/output facilities of the Algol W version at Georgia State made the messy task of numerical conversion almost trivial. These features make Algol W an excellent choice as a designing language for writing interpretive compilers.

3. The Interpretive Compiler

The BASIC interpreter operates in two loosely connected phases: a parsing phase and an interpretive phase. The parsing phase

analyzes the complete source program and translates the input program into pseudo machine-like code. Major functions of this phase are: 1) Allocation of memory locations to variables and constants, 2) Conversion of expressions into Reverse Polish form, and 3) The generation of pseudo machine code. The interpretive phase executes the pseudocode simulating a BASIC machine. The pseudo machine code was designed to minimize the time to decode and execute it.

An essential part of this interpretive compiler are the tables which are used in the parsing phase. Some of these are also used during interpretation of the pseudo machine code. The most important table is a 8192-member array (Figure 1) which contains variables, constants, pseudo machine code and a run-time stack. Locations of simple variables are statically located from location 0 through 285. The variables A-Z are located in 0-25 and A0-Z9 are located in locations 26 through 285. This makes it convenient to access the locations of simple variables during parsing. Of course memory locations are wasted if all the simple variables are not used. The constant number one is kept at location 286 and is used at the assumed step size in a FOR-NEXT statement if not given. A run-time stack takes up the next 100 locations. After the stack constants, array allocation takes place which are inter-mixed and are allocated upon their occurence in the source program. Every constant is allocated a separate location so no time will be lost in a search for the desired constant. Starting at location 8195 the pseudo machine code is placed in descending order. Memory is full when the allocation of variables and code start overlapping. This means that programs with large arrays can not be run with this interpreter but this is not expected to be a real hindrance in beginning programming classes. The allocation scheme makes effective use of memory since more memory than required for pseudocode and variable allocations need not be used, as would be the case if separate arrays were used for code and variables.

Other tables which are used are: a table made up of line numbers and pointers to the pseudocode for the line number, an array of data values from DATA statements, a table of parameter counts and pointers to user defined function definitions, a table of index variable names and return points for FOR-NEXT statements, a table of character literals for use in PRINT statements, and an array table with the number of subscripts, second dimension, and pointers to the arrays location in memory.

Variables and constants are mapped into memory as follows (see figure 2):

Single letter variables are translated into a numerical equivalent with Ø corresponding to A and 25 to Z with the other letters falling inbetween. This number corresponds to the variable's location in memory.

Single letter and digit variables are translated in the following way:
(1) The letter is translated as before. (ℓ)
(2) The digit is converted into internal form. (d)

location = 26 + (ℓ * 10) + d
Example: location B6 = 26 + (1 * 10) + 6 = 42

If the simple variable (A, Z8) is a parameter in a user defined function, it can be located in the stack relative to a stack pointer.

Array variables are located in memory after the stack and are intermixed with constants allocation. The location of a newly allocated array would be in the next available locations in memory. The format of the array when in memory is as follows:
(1) The content of the first allocated location is the number of elements in the second dimension of the array.
(2) The content of the second allocated location is a pointer to the last element in the array.
(3) The remaining locations contain the array elements

To find an array element in memory the following formula is used:

Let P = pointer to array location in memory from array table;
location of X(I,J) = (I*MEMORY(P)) + J + P + 3
location of X(I) = I + P + 3

Constants are located in memory locations following the stack and are intermixed with array allocation. The constant is converted into internal floating point and placed into the next available location in memory. This location will be used in the pseudo machine code as the constants location. Every constant encountered is given a separate location to minimize the time needed to locate the constant.

3.1 Parsing Phase:

Consideration of the overall efficiency of the interpretive compiler dictates that the BASIC source statements should be parsed to a form which is suitable for efficient interprelation. This parsed form or pseudo machine code will in all probability be interpreted many times while the parser only parses each statement once. This implies that as much preprocessing of the source statements should be accomplished during the parsing phase as possible and thus increase the overall system efficiency.

The parsing module is always in one of two states, viz. the program state or the function state. These states are differentiated by the first parameter of the parser module call which when TRUE is in the function state and a function definition body is to be parsed or when FALSE all other statements are to be parsed. The format of the parser module call is:

PARSER (FUNCTION STATE(Boolean); Beginning of the parameter list, End of the parameter list, Function name (Integer));

An example of each call:
Program: PARSER(FALSE,-,-,-);

Function: PARSER(TRUE,0,5,0) - This would be the call if function A's parameter list starts at paramater table (0) and there are six parameters to the function.

The parsing phase is divided into three major modules: POLISH, VARIABLE, and STATEMENT. POLISH accepts an infix expression string and outputs pseudo machine code in Polish form. VARIABLE allocates and locates memory space for simple and array variables. Also function and parameter identifiers are resolved in this module. STATEMENT meanwhile locates the correct Algol W code which contains the statement's syntax equation and controls the parsing of the source statement.

The first step in the parsing phase is to read in a source statement as a character string and remove all blanks. A delimiter is placed at the end of the source statement and parsing of the statement ends when the delimiter is encountered. The statements line number is placed in the line number table along with the location of the starting address of that statements code. Next the statement module is invoked and its syntax equation is followed to produce pseudo machine code for the source statement. If the statement does not fit it's syntax equation exactly an error message is produced. Parsing continues until the statement delimiter is reached. This process continues until an END statement is encountered or an end of file occurs in which case the code for the END statement is produced.

Basic Modules:

The Polish module transforms an expression string into pseudo machine code in reverse Polish form. The order in which expression elements are looked for is as follows:

(1) A unary operator - code is only produced for the negate operator.
(2) A parenthesies expression in which case Polish is invoked again recursively.
(3) Variables, constants, or function calls
(4) Binary operators (**, *, /, +, -)
(5) Go back to (1)

This process is stopped when a statement element is found which does not correspond to one of the above expression elements.

The VARIABLE module is invoked when a letter is found in the statement string which is not a part of a key word (TO, THEN, GO, etc). The variable type is then located and memory space is allocated as stated before. If a variable does not correspond exactly to its variable type format an error message is produced. This location and other identifing codes are output to the statements code stream. Control them passes to the calling module.

When a label is encountered in one of the branch statements (GO TO, GO SUB, IF, etc) one of the two things take place. The label is converted into internal floating point and compared with the last line in the line number table. If the label is less than that value a binary search is made of the line number table and the location of that statements code is used as part of the pseudocode for a direct branch. If it was greater an indirect branch code will be produced and the search will take place when the instruction is interpreted.

FOR-NEXT Statement

The processing of the FOR-NEXT loop is interesting in the way that loop control is set up. Processing of the FOR statement preceeds as follows: (See figure 3)

(1) Op-code for FOR is outputted to code stream
(2) The index variable is transformed (A-0, Z9-285) and placed in the loop stack. Also it is placed in the code stream.
(3) The "=" is passed by and POLISH is called for the initial value expression.
(4) The "TO" is passed by and Polish is called for the final value expression.

(5) If the statement delimiter is the next character then the constant number one located in location 286 is output as the step size; otherwise "STEP" is passed by and POLISH is called for the step size expression.
(6) Two locations are set aside for use in interpreting the FOR-NEXT loop.
(7) Code for a FOR loop compare is produced and an extra location is saved to contain the location of the next statement after the loop's range.
(8) The location of the FOR compare code is kept in the for-compare stack for branching back after the NEXT statement is encountered.
(9) When the NEXT is encountered the index of the NEXT is checked to see if it corresponds to the last FOR index encountered. If not an error exists.
(10) Code is produced to increment the index variable and branching code to the FOR compare whose location is contained in the FOR compare stack.
(11) The next location is placed into the FOR-compare code.

Function Definitions

This BASIC interpreter supports two types of function definitions - multiple and single line definition. Single line definitions are straight forward and parsed as follows:

(1) Locate the function name and place it in temporary storage.
(2) Check for prior definition of the function and if defined before, produce an error message
(3) Produce code to jump around the function definition - this will be completed when the end of the definition is encountered.
(4) Store the parameters in the parameter table in translated form (A-0, Z9-286) while keeping the beginning and ending locations for use in the PARSER Call.
(5) Store the beginning location of the function expression or body definition in the function pointer table.
(6) Check the number of parameters against the number in the parameter count table if the function was previously encountered, and if unequal an error exists.
(7) Check for "=" and if found the function is a single line definition.
(8) Save the parsing call parameters
(9) Load parsing parameters with new values
(10) Call Polish
(11) Replace the saved parameters

Multiple line definitions are parsed as follows:

(1) Steps 1-6 are the same as above
(2) Check for statement delimiter and if not found error occurs
(3) Call the parsing module with the following parameters
    (a) TRUE - function state
    (b) beginning of parameter list
    (c) end of parameter list
    (d) The name of the function (FNA=0, FNZ=25)
(4) The parsing module is exited when a FNEND statement is encountered. Code for this statement is output to the code stream and control returns to the point of the call

Illegal branches out of the function body are not checked until the function is executed during interpretation.

3.2 Interpretive Phase

The interpretive phase is divided into three major modules which executes the pseudo machine code produced during the parsing phase. These modules are: the expression module, array variable module, and the instruction decode module. The expression module executes all expression instructions and leaves the expression value on top of the stack. The array variable module located an array element in memory and checks for an out of bounds condition. The instruction decode module locates that section of Algol W code which will interpret the statements pseudo instructions specified by its op-code.

The interpretive phase operates in one of two states - the run state or the function state. These states are differentiated by the first parameter of the interpretation module call which when has the value TRUE is in the function state otherwise it will be in the run state. The form of the call is as follows:

Interpret (Function state (boolean), function code location, function name, stack pointer (Integer), End of function, Function value returned (Boolean), Function return value (Real))

An example of each!

Run: Interpret (FALSE,-,-,-,-,-,-)

Function: Interpret (TRUE,6217, 1, TRUE,-, FALSE, 15)

This would be the call for the function FNB whose code begins at location 6217, and whose parameter list begins at stack location 15).

Basic Modules

The instruction decode module locates the Algol W code which interprets a given statements pseudo machine code. The section of code is located by using an Algol W CASE statement which branches on the op-code of the BASIC statement's instruction code. The interpretation of each statement is confined to a section of Algol W code, with calls to the array variable module and expression. This process of interpreting one statement as a complete variable length instruction continues until the code for a STOP or END statement is encountered, or an error occurs in the interpretation process.

The expression module is called when an expression needs to be evaluated. The expression will be evaluated until an expression stop code is executed which means expressions could be of any length. This module works as follows:

(1) load the stack with the contents of the memory locations prescribed in the expression code until an operator code is located.

- (a) load function values
- (b) load array element value

(c) load simple variable value
(d) load function parameter value

(2) execute the required operation on the top of the stack for unary operator code or on the top and top-1 of the stack for binary operation code

(3) repeat (1) and (2) until the stop expression code is the expressions value

The array variable module locates the specified array element in memory and checks for the out of bounds condition. The location of an array element is interpreted in the following way.

(1) Find the location of the array in memory from the array table. Call this location L.

(2) If Memory (1)>0 the a two dimensional array. Call EXPRESSION TWICE otherwise for one dimensional array called EXPRESSION once.

(3) The subscript values are now on top of the stack

(4) The elements location is given by the following formula:

```
two dimensional = Memory (L+1) * STACK (TOP-1)
                + STACK (TOP) + 2 + L
one dimensional = STACK (TOP) + 2 + L
```

(5) This location is checked for subscript error by comparing with Memory (L+1) which is the last array element's location

FOR-NEXT loop Interpretation

The FOR loop is another interesting feature of this BASIC interpretive compiler in how it is encountered. When the FOR statement op-code is encountered control passes to the Algol W code which executes the FOR statement. Expression is then called three times to evaluate the initial, final, and step values. The initial value is loaded into the index variables location. The final value and step size are loaded into the next two locations in the code stream. Control passes now to the instructor decode module for the next statement in the FOR loops range. The next statement will be a FOR-loop compare instruction which compares the index value with the final loop value and branches past the loops range if less than, for a negative increment or greater than for a positive increment. Statements in the FOR-loop range are executed until the index value is out of range. When a NEXT statement is encountered the index variable is incremented and a branch to the FOR-compare code is initiated.

User defined function interpretation

The evaluation of user defined functions is interesting in the way the interpretation phase handles them. When the op-code for a user defined function is encountered, the EXPRESSION module is called until all the parameter expressions have been evaluation. If the function is a single line type the function expression is evaluated by a call on the interpretation module in the function state. For a multiple line definition the interpretive module is called in the function state and the function body statements are interpreted until the code for a FNEND statement is encountered. At which time control passes back to the point where the interpretive module was invoked. If the function was not given a value ot an illegal exit from the interpretive module was taken as error condition exists and processing is stopped. The stacking of function parameters on the stack give this BASIC the ability to have recursively defined function definitions.

Conclusion

In practice this BASIC interpretive compiler works very well. The execution speed is somewhat slower than a compiler would be, but I feel the added flexibility and control which an interpretive compiler gives more than makes up for the loss of execution speed, especially in an educational environment. With the added features of run time debugging and the automation of BASIC programming assignment grading the interpreter will become a useful and much used learning tool.

REFERENCES

[1] W. F. Sharpe, "University of Washington BASIC Interpretive Compiler", UWBIC (1967).

[2] R. L. Gauthier and S. D. Ponto, "Designing Systems Programs", (1970).

[3] J. A. N. Lee, "The Formal Definition of the BASIC Language", The Computer Journal, Vol. 15, No. 1, February 1972.

MEMORY ALLOCATION

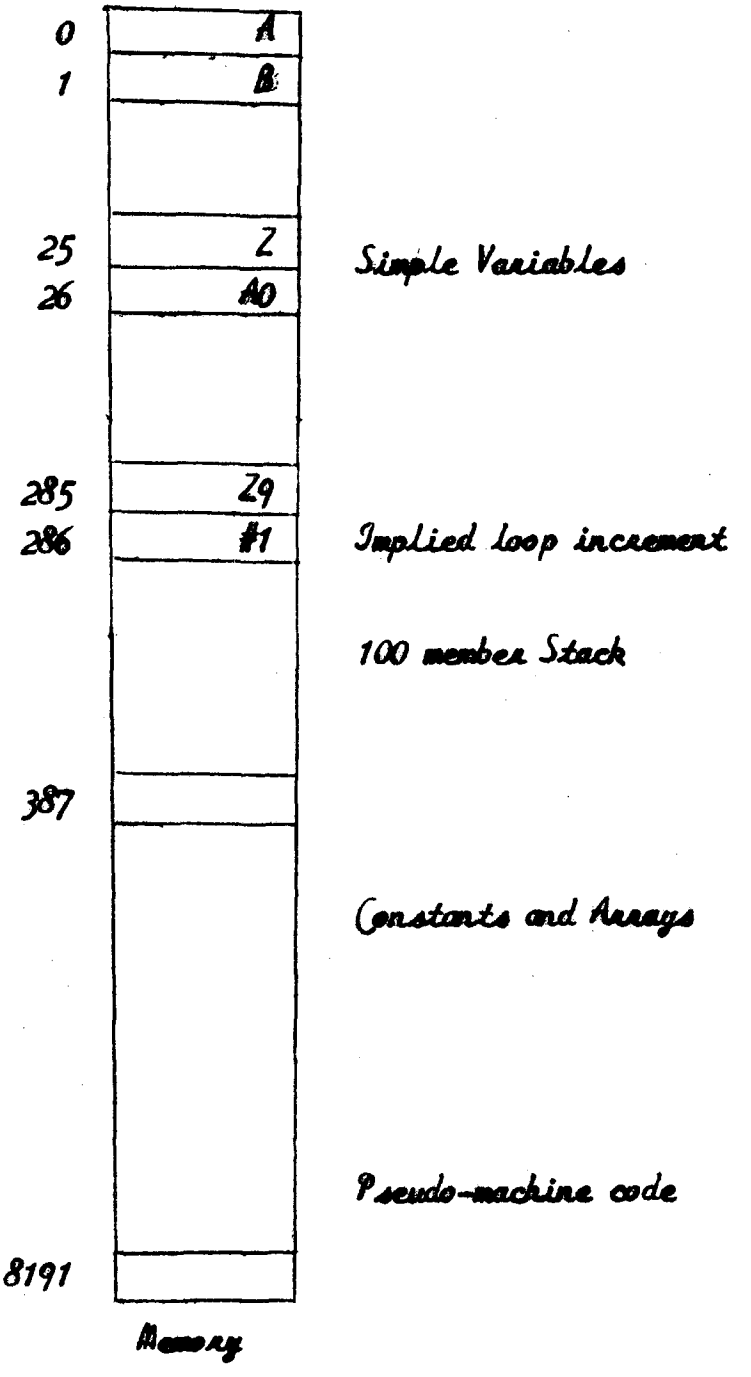
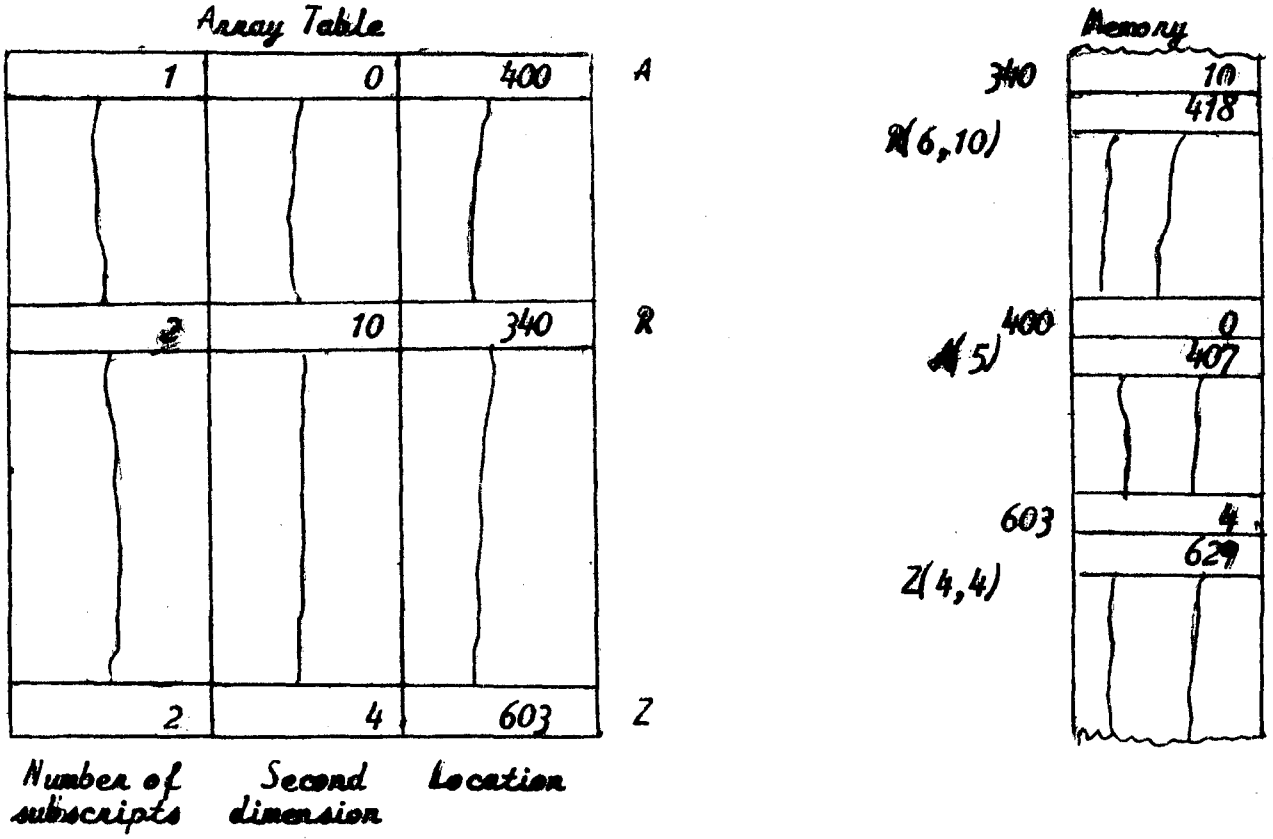


Table 1

Allocation

ARRAYS



FUNCTION PARAMETERS

FNA(2,45,5)

The parameter locations:

2 is 0 + STACKPTR
45 is 1 + STACKPTR
5 is 2 + STACKPTR

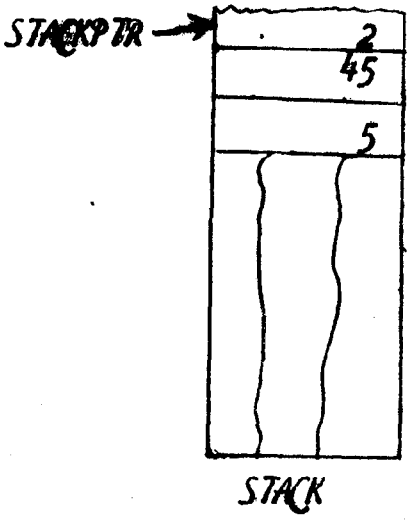


Figure 2

FOR-NEXT STATEMENT

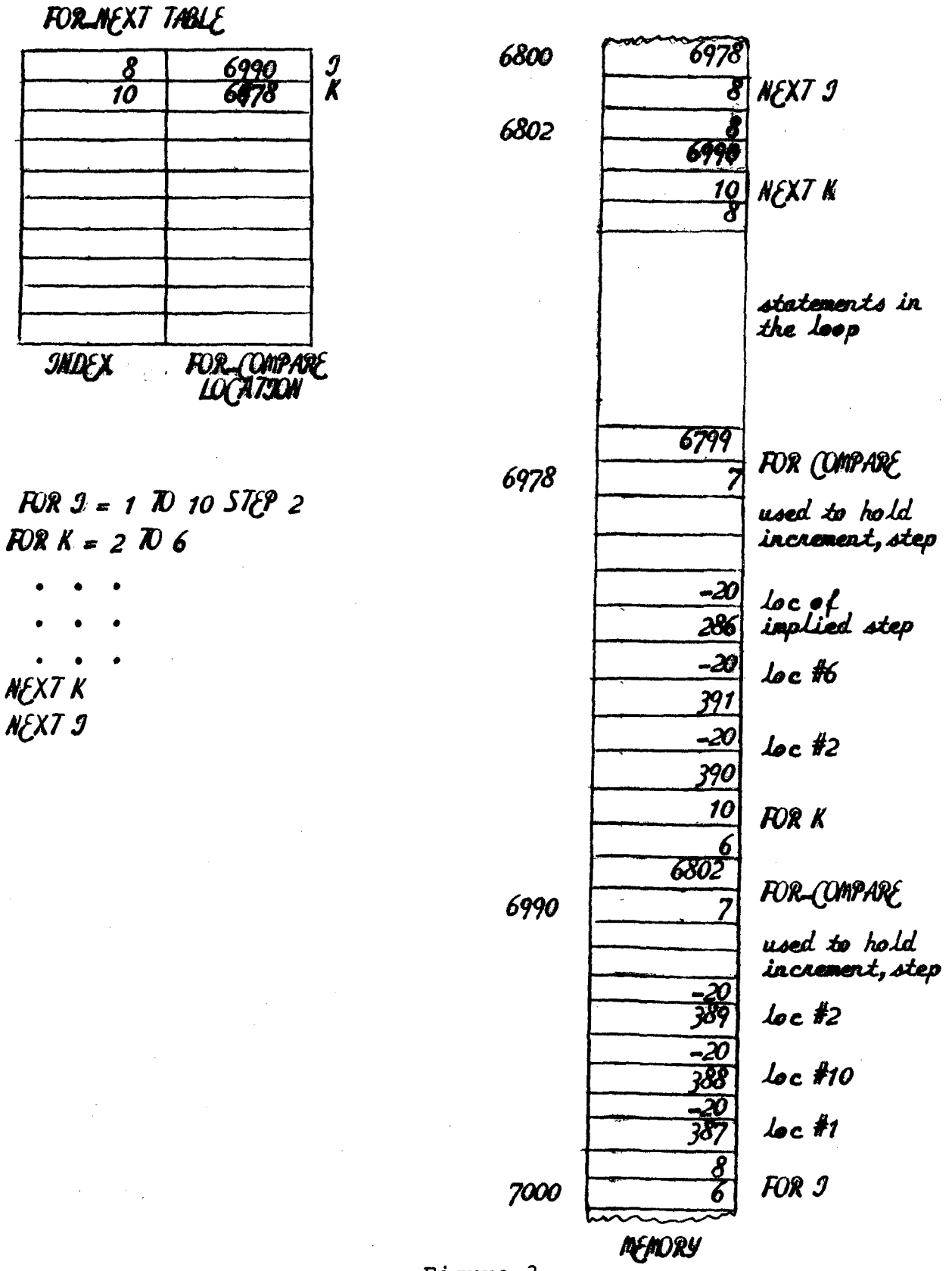


Figure 3

USER DEFINED FUNCTIONS

PARSING PHASE

FUNCTION TABLE

	LOCATION	NUMBER OF PARAMETERS
0		
1		
2		
3	7642	3

```
DEF FND(I,J,K)
  . . .
  . . .
LET FND = I + J - K
FNEND
```

```
CALL PARSER(TRUE,0,2,3)
```

0	8	I
1	9	J
2	10	K
3		

PARAMETER LIST

7595	—	
	3	FNEND 0
	-5	
	-20	stop expression
	-23	-
	3	K
	-12	
	-24	+
	1	J
	-12	
	0	I
	-12	
	3	
	-13	LET FND =
	1	
7642		Function body
	7595	Jump around function body
	11	

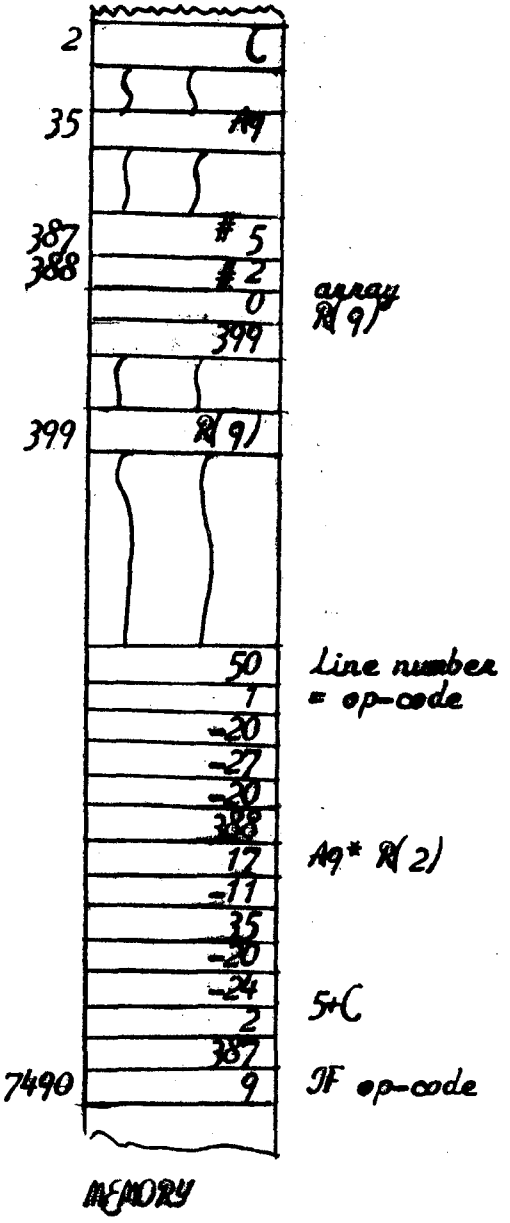
Figure 4 MEMORY

Example: Parsing

Statement: 25 IF 5+C = A9 * A(2) THEN 50

Syntax Equation:

IF <expression> <relational operator> <expression> THEN <line number>



(1) Produce op-code for IF
(2) Call POLISH
(3) Save code for "="
(4) Call POLISH
(5) Store code for "="
(6) Convert "50" into internal form
(7) Store 50

Figure 5

PSEUDO-MACHINE CODE

Statement op-codes

1 LET
2 PRINT
3 STOP
4 READ
5 FNEND
6 FOR
7 FOR COMPARE
8 NEXT
9 IF
10 INDIRECT JUMP
11 DIRECT JUMP
12 COMPUTED GO TO
13 REPLACE
14 RETURN
15 GO SUB
16 USER FUNCTION- one line

Relational operators

1 =
2 >=
3 >
4 <>
5 <=
6 <

Other

-20 stop code

Functions

-1 SGN
-2 SQR
-3 COS
-4 ABS
-5 ATN
-6 LOG
-7 EXP
-8 INT
-9 TAN
-10 USER FUNCTION

Arithmetic Operators

-23 -
-24 +
-26 /
-27 *
-29 **
-30 - (unary)

Variable

0-285 SIMPLE
-11 ARRAY
-12 STACK
-13 FNx

Figure 6

INTERPRETIVE PHASE

LET X = FND(2+J,J,6*A1)

CALL INTERPRET(TRUE,7642,3,0,FALSE,4)

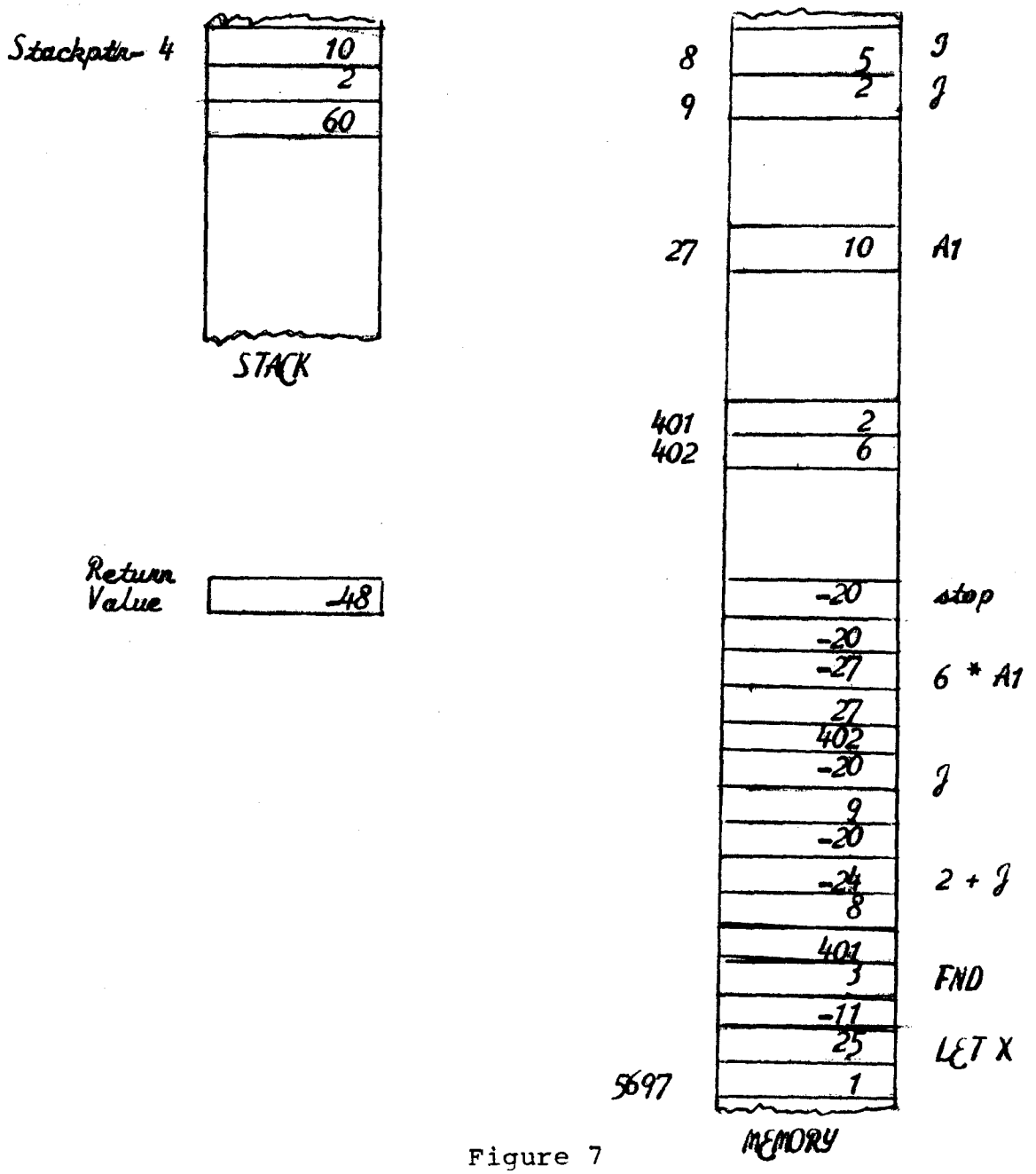


Figure 7