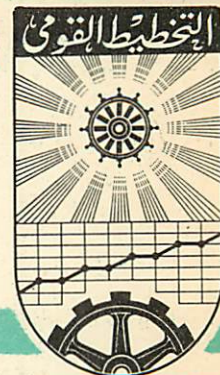


UNITED ARAB REPUBLIC

THE INSTITUTE OF NATIONAL PLANNING



Memo. No. 567

Advanced Compiler Techniques
(Lecture Notes)

Lionello A. Lombardi

May 1965

Operations Research Center

Table of Contents

	<u>Page</u>
Cmpiler Techniques	2
An interactive, Algebraic Compiler	5
Input-Output Considerations	24
Organization of an executive Routine	48
One Pass Translation of Do-loops	63
Notes on FORMAC	85

Compiler Techniques

(Extensions from FORTRAN:

- a) Mixed mode expressions
- b) Variables with any number of dimensions)

1. Polish Notation (prefixes)

The operators (e.g., +, -, etc.) precede the operands.

Examples:

- 1) + 3. A instead of 3. + A
- 2) + 4. * SQRTF A 2. instead of 4. + SQRTF (A) * 2.

2. Order of operators number of operands.

Examples:

- 1) SQRTF has order 1.
- 2) + has order 2.

3. Parentheses in polish notation are necessary only if the order of operators varies.

4. A push down list (pd l) consists of :

- a) A top pointer
- b) an indication of its bottom
- c) Same memory space

It can be organized as association List for the purpose of saving space when several pdl coexist.

(Clue : NEVER tie up index registers as top pointers of a pdl)

5. Translation FORTRAN into Polish needs one push down (pdl) list for infix operators, named infix pdl, which also contains closed parontheses.

6. Translation from Polish into machine code needs one pdl for operands, named operand pdl. It only contains addressess of operands and intermediate variables, along with indication of whether they are declared variables or constants or intermediate variables. The operands pdl can be augmented by associating to each entry the following information:

mode (e.g. floating, Boolean, etc.)

7. One pass formula translation

Both infix pdl and operand pdl coexist at compilation time. The Polish stage is skipped.

8. Treatment of intermediate variables (iv)

IV are the results of applying operators to operands within a formula.

Example:

In $4 \times (3 + N)$, the number $3 + N$ is an intermediate variable.

They are actually only computed at execution time, but space for them is allocated at compilation time in the ghost pdl. The ghost pdl at compilation time simply consists of a top pointer (without space) and a register to remember its maximum length. It has space but no top pointer at execution time.

9. Organization of the compiled program (COMMON excluded)

(This is a very simple one, more efficient ones are desirable)

- 1) Transfer vector (list of names of subroutines and functions)
- 2) Executable code followed by constants
- 3) Areas for declared variables and constants.
- 4) Space for ghost pdl.

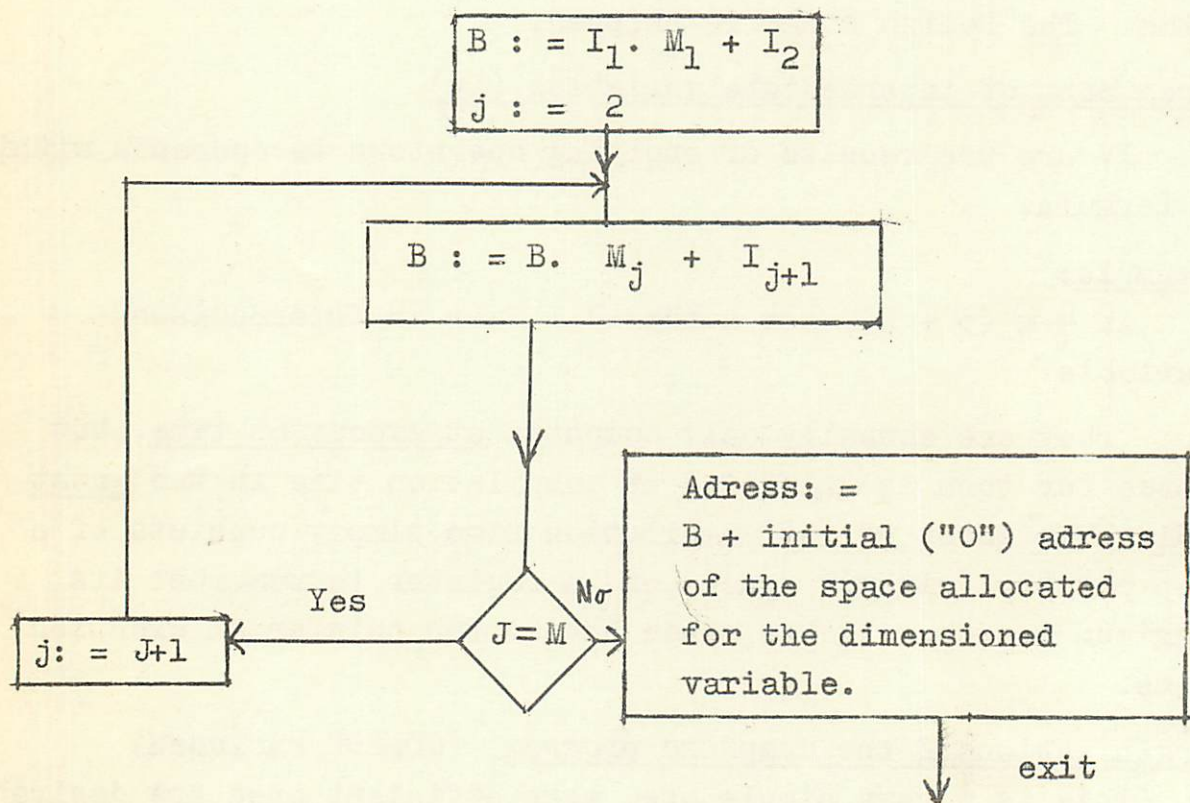
During compilation it is necessary to build the transfer vector and to keep track of the length of items 1,2,3 and 4 above. At the end of compilation or at leading time item 2 above need to be relocated.

10. Formula for dimensioned variables

The address of $A(I_1, I_2, \dots, I_n)$,
where I_i is an integer formula, preceded by DIMENSION

$A(M_1, M_2, \dots, M_n)$

where M_j is an integer constant,
is computed at execution time by the following routine:



An Interactive, Algebraic Compiler

Introduction

A computer time-sharing system, such as that under development at MIT's Computation Center and at Project MAC provides one with the unusual ability, in this age of ultra high-speed and ultra high cost computer equipment, of enjoying "hands-on" operation of the machine. The programmer need not wait long periods of time to locate his program bugs; he may instead debug his program on'line, with direct supervision of the machine's operation.

Seated at a remote console, usually consisting of a typewriter and printer, the user may "interact" with the computer. In the MIT Compatible Time Sharing System (CTSS), for example, the user enters his system commands, such as INPUT, LOAD, START, EDIT, and compilation commands in response to observed action by the computer, as reflected by the printer portion of the console. Thus, a normal sequence of commands might be

INPUT	Enables the user to enter his source statements
FILE	Causes list of source statements to be retain on the disc file.
FAP	Causes the source program to be assembled in the FAP language
LOAD	Loads the assembled object code into the computer.
START	Causes execution of the previously loaded object code.

In each instance, the user enters a command after he receives a confirmation that his last command has been successfully executed. For example, if the FAP assembly, in the above illustration, was not successeful, the user will be able to learn this and take steps to correct his source program, rather than to load the hoped-for object code. The important point here is that the user does, in fact, interact with the computer. His actions are generally based upon information received from the computer, and the computer's actions are generally based upon instructions received

from the programmer. Both the user and the machine make decisions, each upon some action taken by the other.

If the user makes a mistake, for example, trying to load the object code of the FAP program that did not assemble, the computer will inform him of his error immediately. Similarly, if the user enters a command that is misspelled or that just does not exist, the computer will inform him of this immediately. The user is thus given an opportunity to take appropriate measures.

Let us go into a bit more detail about the means of entering a source program. In particular, let us suppose that a hypothetical user wishes to enter a program written in FORTRAN, the most widely used algebraic programming system. Under the present CTSS development, the computer user follows the following procedures. He enters an INPUT command, which causes the computer to type out a line number, starting with 00010. After the line number appears, the programmer enters his first source program statement. He enters a carriage return to denote the end of the line, and a new line number is given automatically by the computer. The line number progress in jumps of 10, thus, the second line number is 00020, the third is 00030, etc. The programmer continues to enter his source statements until his program has been completely entered. If he makes an error in any statement, he may use the regular CTSS error correcting procedures ("to delete the previous character, to delete the entire line) or he may reenter the entire statement by entering the line number followed by the corrected text of the statement. A new statement may be inserted by first entering a line number between the two statements that border the inserted one, such as 15 between lines 10 and 20, and then following this number the text of the new statement. A statement may be deleted by using the DELETE command. Finally, the corrected list of statements is given a symbolic name and is file. It is here that the computer is informed of the particular language in which the file has been written. Thus, the statement to file an input list named, say, TEST, would be

FILE TEST FORTRAN

This means "Store the previously created input file in a disc file named TEST in the FORTRAN language." To compile this program, the user enters the command

FORTRAN TEST

which tells the computer to compile the source program file called TEST in the FORTRAN mode using the FORTRAN compiler associated with the CTSS monitor. After the program has been compiled, the user must load it, and then start it, by means of the LOAD and START sequence, or the newer LOADGO command. If an error occurred during compilation, due to either a typographical error in the source program, or possibly some more substantive mistake, such as branching to a nonexistent statement number, the user must correct the source program, by using the EDIT command to reinstate the program in the INPUT mode, make his correction (s), file the program, and recompile it. Basically, he does not learn about errors until he commences the compilation process, or until he executes the compiled program. And he cannot correct these errors except by reinstating his source program in the input mode and then recompiling it from scratch.

Although the time sharing concept provides for the relatively short delay time in learning about and correcting program errors, much of the potential of the interactive features of the time sharing system are not present in the existing approach to program writing. It is for these reasons that work is being done towards the development of an algebraic programming system that will, in fact, be based upon its inclusion in the repertoire of a time sharing operation, such as CTSS. This system will be based, in general upon the existing FORTRAN language, since it has been found to be the most widely used algebraic language in existence today. Hopefully, the new system will be fully compatible with the existing FORTRAN systems, such that any program or subprogram written in languages such as FORTRAN II or FORTRAN IV may be included in systems using the new interactive language. A questionnaire was recently circulated to computer users in the MIT community for the purpose of determining their preferences and requirements for such

an interactive algebraic programming system. Detailed results of this study are given in a later section of this paper.

The interactive language will provide its users with the many features that will best enable them to take full advantage of the time sharing hardware at their disposal. For example, the entire structure of the input-output aspects of the FORTRAN system are being redesigned so as to provide for the requirements of the time sharing console. A description of this aspect of the work is also included in a later section of this paper.

But most important, the new system will give the programmer significant advantages in his ability to effectively take advantage of the time sharing system in the process of debugging. First of all, he will be able to make changes in the source program during execution and without total recompilation. He will be able to keep track of the progress of program's execution by means of some powerful tracing features which may be selectively controlled at the remote console. The programmer will be provided with some powerful interactive features, enabling him to enter a part of his program, in the form of regular FORTRAN source statements, during execution of the program. Finally, the programmer will be able to dump and reload his completed object code in a manner that will provide the most efficient operation of the program during its regular execution.

The programming system under discussion here is not of the interpretive form. Rather, it will generate actual machine code in response to each FORTRAN source statement, without the use of an intermediate language. (An interpretive system, incorporating some of the features described here, has been developed by the IBM Applied Programming group of the Data Systems Division in New York. However, because it is an interpretive system, it produces an object code that is rather slow, and hence cannot be used to compile programs intended for production runs.)

The purpose of this paper is to describe the various component parts of this system, with a view toward specifying the means by which it should be constructed. Thus, the paper includes sections

on the language specifications, the input-output operations, the construction of the compiler and the compiled executive routine, and finally the results of the aforementioned questionnaire.

Specifications of the Language

In general, the interactive system will include all of the language features now present in FORTRAN II, and probably will include those present in FORTRAN IV. The latter system includes the ability or using recursive subscripts [e.g., $A(I(J(K)))$], Boolean statements, [e.g., $IF (A .LE. B .OR. (C+2*D) .GE. 50) GO TO 25$], and the ability to use an expression anywhere in the program. In addition, FORTRAN IV provides for the legality of mixed mode expressions, and also permits the programmer to define labels as being either fixed or floating, so that he need not adhere to the I ... N rules in FORTRAN II. There will be some changes in the input-output statements, and there will be a more complete discussion of these in the next section.

In addition to the regular FORTRAN statements, the interactive system will provide a number of new features that aid in writing and debugging a program. We may call these pseudo instructions, since they direct the operation of the compiler, but do not have any direct affect upon the generated object program.

The Alteration Statements. These statements permit a programmer to change the nature of his program either before or after compilation.

DELETE $s_1 + s_2$ LINES

where s_1 is a statement number, and s_2 is the number of additional lines that define the statement to be deleted. If s_2 is 0, the "+ s_2 LINES" may be omitted.

Causes deletion of the defined statement from the source program.

DELETE $s_1 + s_2$ LINES
THROUGH $s_3 + s_4$ LINES

Causes deletion of a string of statements inclusive of the beginning and end statement as defined by s_1, s_2, s_3 and s_4 .

INSERT $s_1 + s_2$ LINES

Causes the compiler to accept and sequentially compile into the over-all object code a string of statements starting with the first to be located in the position defined by s_1 and s_2 .

INSERT $s_1 + s_2$ IN $s_3 + s_4$

Causes the statement located at $s_1 + s_2$ lines to be removed from that spot and be inserted at the point defined by s_3 and s_4 . The statement previously at $s_3 + s_4$ lines is pushed up one line.

ALTER $s_1 + s_2$ LINES AT c

where c is the column number of the card image where the alteration is to commence.

Causes the compiler to accept changes in the statement number defined by s_1 and s_2 starting at position given by c.

ENDCHANGE

Causes the end of the insert mode.

TRACE $s_1 + s_2$ LINES THROUGH
 $s_3 + s_4$ LINES

Results in a logical trace on all statements of the source program that lie between the two statements defined by the parameters of the TRACE statement. The tracing includes the statement and line number, and the value of the statement if an arithmetic expression, the value of the index of a DO if a DO statement, or the type of statement if some other kind is being executed.

TRACE ON $v_1 \dots v_n$

where v_i is the symbolic name of a variable to be traced.

Causes tracing of all statements where the variables v_i are affected or appear. In addition to the tracing information described above, the symbolic name of the variable is printed out for each statement traced.

Definition of an operation at object time. Present FORTRAN systems, which provide for an alphanumeric FORMAT specification, also provide for a so-called "variable FORMAT statement," which is read into the program at object time. This is made possible in the present FORTRAN systems because the execution of input-output statements with FORMAT specifications is accomplished by means of an interpretive process. Thus, the FORMAT statement is never compiled; rather, it is stored in memory in basically the same form as it was written in the source program. Thus, a FORMAT specification may be quite readily read in at object time and used in the same manner as any other type of FORMAT statement. With a remote access time sharing system, it is possible for the programmer to thus enter a FORMAT statement at the console, followed by, say, input data, which will be read in according to this specification.

It was suggested, by Professor L. A. Lombardi and S. S. Alexander, that an analogous feature be extended to arithmetic expressions that may occur in virtually any source program statement. This feature would work as follows: The programmer will insert a special symbol in any particular position where he wishes to enter an expression at object time. This symbol will consist of a dollar sign followed by an alphanumeric label of from one to five characters in length. Thus, such a label might be \$PRICE, or \$X. During execution, when a statement containing such a label is encountered, the program comes to a pause, and the location (in the source program) and the name of the insertion variable are printed out at the console. The programmer then enters an expression, in the source program language (FORTRAN) that is to be substituted for the insertion symbol. The expression is compiled, and execution continues.

When the expression is entered, the programmer may attach, at the beginning, a special character, such as another dollar sign, to indicate that this inserted expression should be used throughout the current execution of the program. Without this special

symbol, the system will call for the insertion of a new expression each time the particular insertion symbol is encountered during execution.

The programmer may, if he wishes, include informational printouts of specific instructions governing the particular insertion. This would be done in the conventional manner, as will be more fully discussed in the section on input-output statements.

The insertion feature provides for great flexibility in the use of the remote console. The same program may thus be used to handle similar problems with varying parameters, decision rules, etc.

Finally, it should be noted that the insertion feature need not be limited to the insertion of expressions; it would be quite feasible to permit the insertion of one or more statements in basically the same manner. However, such inserted statements will, collectively, take the form of a single-valued FUNCTION subprogram.

Tracing. The interactive algebraic programming system will incorporate a number of powerful program tracing features to facilitate the debugging process. The trace is selective, i.e., it is defined and requested by the programmer, who also starts it and stops it as he pleases.

An example of some of the trace features to be incorporated in this new system may be found in the FORGO system developed at the University of Wisconsin for use with the IBM 1620. FORGO is a load-and-go version of FORTRAN II, with some language restrictions and, in the case of the 1620 FORTRAN II system, a few extensions. FORGO is an interpretive language, i.e., it translates the source language into an intermediate language, and translates this into machine code for the purposes of execution. FORGO was recently converted such that it may be used interactively on the 1620, making it an excellent demonstration of the type of system under discussion here.

Since FORGO, like the interactive system under development, maintains an image of each source program statement, logical tracing, by source statement, is made possible. FORGO includes a number of trace options. They include a complete trace on all statements between two given limits, a complete trace on all branches in the program, and other features that permit the programmer to limit the length of the program, if it appears to be excessively long, and to dump the object program.

The new system will incorporate these same features, but will, in addition, provide even greater tracing ability. Most important will be the ability to trace selectively on a single variable, or on a group of several given variables. The trace will contain the source program name of the variable, the location in the source program of the statement being traced, and the net result of the statement on the variable in question. This one feature is, in particular, a very significant improvement in the repertoire of debugging aids.

In addition to the trace features for logical tracing at execution time, the interactive algebraic system will include some of this same ability for use by the programmer during compilation. In particular, the computer will inform the user, either voluntarily or on request, the means by which his program may get to a numbered statement as it is entered. Thus, when the programmer enters, say, statement 47, the computer will inform him how he might get to this statement from the various other parts of the program. This feature would be voluntary, i.e., it would be requested by the programmer, on the initial input of the source program; however, it would be mandatory, i.e., generated automatically, whenever the programmer inserts, alters, or deletes a numbered statement.

Diagnostics

When an error is detected, the column number in which the error was made, and the type of error made (in some code) is printed out directly below a print-out of the statement to which it refers.

We classify programmer errors into the following categories:

- 1) Dimension errors
- 2) Statement numbers and constants
- 3) Format and input-output errors
- 4) Arithmetic operations
- 5) DO statement errors
- 6) GO TO, computed GO-TO and assigned GO TO errors
- 7) Errors in IF statements
- 8) Errors in EQUIVALENCE, COMMON and FREQUENCY statements.
- 9) Errors in calling and using subroutines and functions.
- 10) Subscripting errors.

Since the program is being compiled one statement at a time, certain errors will not be detected immediately, and the diagnostics will not always be correct. However, the programmer needs some information to guide him in his corrections. Either the compiler "guesses" the error, and the diagnostic is printed out, or the compiler has more than one diagnostic for the error. All of the diagnostics for the error are printed out.

A system where the programmer has the option of asking for more diagnostics if he does not understand the original one seems to be the most satisfactory idea. In this way he will be able to "interact" with the machine.

This paper will tabulate programmer errors according to the scheme given above. Beside each error there will be a suggested (possible) diagnostic. Since we cannot expect the diagnostics to be exact, some of them will not even indicate what the real error was.

DIMENSION ERRORS

Error

Suggested Diagnostic

- | | |
|---|--|
| 1) Dimension statement missing commas, between variables. | Missing comma in dimension, |
| 2) Missing $\left(\begin{smallmatrix} \text{left} \\ \text{right} \end{smallmatrix}\right)$ parenthesis in DIMENSION, | Missing $\left(\begin{smallmatrix} \text{left} \\ \text{right} \end{smallmatrix}\right)$ parenthesis |
| 3) Missing variable name in DIMENSION, e.g. DIMENSION ALPH (3,5), (2,5) | Illegal variable name. This occurs since it is reading it as a new variable name after the comma. |
| 4) Function name in Dimension variable table. | |
| 5) More dimensions than specified in language (e.g. ALPH (2,3,4,5)). | Too many dimensions |
| 6) Missing subscript of variable in Dimension statement, (e.g. DIMENSION ALPHA (6,), B(101)). | Illegal character in DIMENSIONed variable. |

FORMAT AND I/O ERRORS

- | | |
|--|---|
| 1) Format has no statement number. | This will be diagnosed as an illegal variable name, since the compiler will have no way of identifying whether the statement is a FORMAT. |
| 2) Missing $\left(\begin{smallmatrix} \text{left} \\ \text{right} \end{smallmatrix}\right)$ parenthesis in FORMAT. | Missing $\left(\begin{smallmatrix} \text{left} \\ \text{right} \end{smallmatrix}\right)$ parenthesis in FORMAT. |
| 3) Unequal number of parentheses in FORMAT. | ditto. |
| 4) Illegal I/O unit specification. | |

- | | |
|---|---|
| 5) Comma following format statement | Illegal character in format |
| 6) Too many Formats. (Identifiable format has not been referred to by READ statement) | Too many formats. |
| 7) Variable overflow in format statement table | |
| 8) Commas in READ statement. | Illegal character in read statement. |
| 9) No number in read statement (e.g. READ A | Variable name too long. This happens because it is read as an expression. |
| 10) Too many format continuation cards | Too many continuation cards. |
| 15) Undefined variable in output specification. | Variable specified in output is not defined. |
| 16) Undefined format | Illegal control statements in format. |

If the diagnostic is not correct, it is because the compiler detected a different kind of error from what actually happened. This is because it has no way of knowing what the programmer really meant.

Also error in I/O can cause errors to occur in statements that strictly have no error. For example, if the programmer leaves the statement number off the format statement number, then two diagnostics can be made. One diagnostic will be made on the incorrect format statement, and the other will be on the preceding READ statement. The error on the read statement will be that it is referencing a FORMAT statement that does not exist. Thus there will be a diagnostic on the READ statement as well as the FORMAT statement.

STATEMENT NUMBERS AND CONSTANTS

- | | |
|--|--|
| 1) Statement label has an invalid character. | Invalid character in statement name. |
| 2) Statement label has too many digits. | Statement number too long. |
| 3) Misuse of column six. | Illegal character in column 6. |
| 4) Illegal character in constant (e.g. 2 3 4 I J) | Missing operator between operands. |
| 5) Missing digit after E+ or E- in floating point constant | Missing digit in constant. |
| 6) Floating point constant greater than b^N . | Floating point constant greater than b^N . |
| 7) Invalid character in expression. | Invalid character in expression. |
| 8) Two statements have the same label | Statement number has occurred before in line ... |
| 9) Tape number must be fixed point number. | Error tape number must be fixed point. |

The diagnostics for arithmetic expressions will usually be quite accurate since the errors are syntactical and these are easiest to detect.

ARITHMETIC OPERATIONS

- | | |
|---|--|
| 1) Expression on left side of equals sign (e.g. A B = (C+D)/E) | Variable on left hand side cannot be referenced. |
| 2) Two operations occurring in a row. (e.g. A++B-+D) | Arithmetic operations are not successive. |
| 3) Variable name begins with a number (eg AJAX = 8 JAR + SBU) | No operator between operands. (In this case the error would be interpreted as a missing arithmetic operation between the number and the letter.) |

- | | |
|---|---|
| 4) Unequal no. of parentheses in arithmetic expression. | Unequal number of parentheses in expressions. |
| 5) Illegal characters in expression (e.g. A=B +, C) | Illegal character in expression. |
| 6) Fixed point variable begins with A-H or O-Z (on right side of = sign.) | Mixed Fixed and floating mode expression. Right side of = sign. |
| 7) Too many characters in variable name. | Missing operator between operands. Left side of = sign. Variable name too long. |
| 8) Variable name same as function name. (e.g. ABLE = BAKEF + CHARL). | Arguments missing in function. |
| 9) Variable has too many subscripts. (eg ABLE (3,4,5,6) | Illegal operator between operands. |
| 10) Program too large for memory | Program too large for memory |

The diagnostics in Arithmetic statements can be quite ambiguous. This is because of a number of complicating factors of which the major one is the side of the equals sign that the error occurs. Again the compiler has to guess the kind of error that was made and then print-out a diagnostic.

DO STATEMENT

- | | |
|---|--|
| 1) DO statement refers to non executable statement. eg DO 15 I = 1,5
15 FORMAT (. . . . | Do <u>references</u> non-executable statement. |
| 2) DO refers to statement number that does not <u>exist</u> | Reference of DO does not exist |
| 3) DO refers so statement number that occurs more than once. | This would not occur since there would be a diagnostic as soon as duplicate statement labels are punched in. |
| 4) Incorrect nesting of DO loops.
(eg DO IO I = 1, 20
DO 11 3 = 2, 20
10 - - -
11 - - - | Imptoper nesting of DO loops. |

- | | |
|---|--|
| 5) Illegal comma in DO statement,
eg DO 10, I = 1, 20) | Illegal character in DO
statement. |
| 6) Object of DO is a transfer
statement eg DO 12 I = 1, 20
12 GO TO 87 | DO statement references a
transfer Statement. |
| 7) DO loop starts at 0.
(eg DO 17 I = 0, 10) | Improper indexing in DO
statement. |
| 8) Floating point increment in DO
loop (eg DO 17 I = 10, 0.2) | Improper indexing in DO
statement |
| 9) Illegal transfer into DO loop.
GO TO 15
DO 15 I = 1, 10
15 A(J) = B (I) +1. | Illegal transfer into loop. |
| 10) Illegal transfer out of DO loop
DO 15 I = 1, 10
15 IF (ALP (I) -10.) 16, 17, 18
16 - - -
17 - - - | Illegal transfer out of DO
loop. |
| 11) Illegal indexing on DO loop.
(DO 15 I = 1, 10)
(DO 16 J = 1, 11) | Illegal indexing in DO state-
ment. |

GO TO; COMPUTED GO TO AND ASSIGNED GO TO ERRORS

- | | |
|--|--|
| 1) More than 11 statement numbers
in computed go to. | Illegally formed GO TO. Too
many statement references. |
| 2) GO-TO transfers illegally into
a DO loop. | Illegal transfer into DO
loop |
| 3) Transfer (GO TO) to non-existent
statement numbers | Transfer to non-existent state-
ment no. (This diagnostic will
occur only at the end of the
program when it old statement
numbers. |
| 4) Transfer to a non-executable
(e.g. transfer to a FORMAT
statement.) | Transfer to a non-executable
statement |

- | | |
|---|--|
| 5) Floating point number as index in computed GO TO. | Floating point index in computed GO TO. |
| 6) Missing parenthesis in computed GO TO. | Missing ^(left) (right) parenthesis in computed GO TO. |
| 7) Missing commas in computed GO TO. (eg GO TO 56, 57, 58, 59) I | Transfer to non-existent statement. This occurs since, if a comma is missing, two numbers will be read as one.) |
| 8) Too many if GO TO assign statements | |
| 9) Transfer goes to itself, e.g.
15 GO TO 15 | Illegal transfer. Transfer goes to itself. |
| 10) Missing index in computed GO TO | Illegally formed GO TO.
(Because it will seem to be a GO TO with illegal characters in such as commas and parantheses.) |
| 11) A predecessor cannot be found to this statement, e. g. GO TO 20 | A predecessor to this statement cannot be found. |
| <div style="text-align: center;">A = B + 3.
14 D = A + 5.</div> | |
| 12) Missing, in Assigned GO TO | Missing, in assigned GO TO |
| 13) Transfer to a transfer statement, e.g. GO TO 15
15 GO TO 85 | Illegal transfer. |

ERRORS IN IF STATEMENTS

- | | |
|---|--|
| 1) Transfer to non-executable statement. | Transfer to non-executable statement. |
| 2) Too many commas in IF statement. | Illegal characters in IF statement. |
| 3) Control to more than 3 statements, i.e., IF (A-B) 3,4,5,6. | Too many statement references in IF statement. |
| 4) = sign in IF statement. | Illegal character in IF statement. |
| 5) IF statement transfers to itself. (e.g. 25 IF(A-B) 26, 26, 25) | Illegal transfer. |
| 6) Illegal sense light number. | Illegal sense light number. |
| 7) Illegal sense switch. | Illegal sense switch. |

ERRORS IN EQUIVALENCE AND COMMON STATEMENTS (also FREQUENCY Statement)

- | | |
|---|---|
| 1. EQUIVALENCE statement as 1st statement in DO loop. | } Illegal statement in range of DO loop. |
| 2. FREQUENCY statement in range of DO loop. | |
| 3. Wrong variables put in EQUIVALENCE statement. | There is no way of knowing that this error has been made. |
| 4. Missing parentheses in EQUIVALENCE. | Missing (left) parenthesis in EQUIVALENCE. |
| 5. Unwanted symbols in EQUIVALENCE i.e. (= , +, extra commas) | } All these errors should be immediately detectable. |
| 6. COMMON - too many commas | |
| 7. COMMON - missing parentheses | |
| 8. Only one variable given in EQUIVALENCE. | |
| 9. Missing C in FREQUENCY. | |
| 10. Missing comma in FREQUENCY. | |

ERRORS IN SUBROUTINES AND FUNCTIONS

- | | |
|---|--|
| 1. Subroutine statement not first. | Subroutine statement not first. |
| 2. Function statemnt must be first. | Function statement not first. |
| 3. Unpaired parentheses in call statement. | Missing ^(right) _(left) parentheses in call. |
| 4. Improper function name. (eg SIN (A,B) Not ending in F. | Floating point subscripts are illegal. |
| 5. Improper function argument (eg SINF (J,K) | Fixed point arguments are illegal. |
| 6. Multiply defined SUBROUTINE | Subroutine has been defined previously. |
| 7. Subroutine not defined. | |
| 8. Missing)in function statement | Illegal character in variable name. |
| 9. Non-alphanumeric character in function name. | |
| 10. Missing parentheses in subroutine statement. | |
| 11. Function name is in Dimensioned variable table. | This error would have been diagnosed if the DIMENSION statement has been entered previously. |
| 12. Return statement not in SUBROUTINE or function program. | No entry back into main program. |
| 13. End not preceded by If, GO TO, stop, or return. | |
| 14. SUBROUTINE program name has too many letters. | Illegal subroutine name. |
| 15. Too many characters in function name. | Illegal function name. |
| 16. Function definition not first. | |

SUBSCRIPTING ERRORS

- | | |
|--|--|
| 1. Undefined variable in subscript. | |
| 2. Missing DIMENSION for subscripted variable | Undimensioned subscripted variable. |
| 3. Too many dimensions for subscripted variable. | Too many dimensions. |
| 4. Floating point subscript, e.g.,
TABLE (5, 6) | Floating point subscripts are illegal. |
| 5. Program too large for memory. | |
| 6. Symbol table too large for memory. | |
| 7. Subscript is an expression (floating point). | Floating point subscripts are illegal. |

MISCELLANEOUS

- | | |
|-------------------------------------|-------------------------------------|
| 1. Illegal Digit in end statement | Illegal character in END statement. |
| 2. Missing, of tape number | |
| 3. Too many continuation cards | |
| 4. Statement number in column 1 | Illegal character in column 1. |
| 5. Continue statement has no number | Continue statement has no number. |

INPUT-OUTPUT CONSIDERATIONS IN AN INTERACTIVE ALGEBRAIC COMPILER

INTRODUCTION

Input-Output in relation to a higher level language has conventionally had to deal with files of data which were wholly prepared in advance. The programmer had to be concerned to some degree with the type of input-output device that was used and the command structure had to specify which units were used and what the precise form of the data looked like. Since most output was destined to be read by another program the format of the data had to be explicit and unambiguous.

With the advent of time-sharing systems, a different approach to input, output and formats must take place. Characteristically, the user is interacting with the computer on a remote basis and is generally not concerned with what physical units are being used to input data to his program and to accept the data generated by his program. The Compatible Time-Sharing System in use at M.I.T. uses random-access devices (disk and drum) for active secondary storage.

There are now two major aspects of input-output to be considered. On the one hand we have the storage of files of data and programs on some large secondary storage device. The user is not particularly concerned with where and how it is stored but merely with the fact that files can be stored and can be referenced easily.

The second major consideration is the input and output of data to the user via some remote device such as a typewriter keyboard and printer or a scope. In this case there are two important things to be noted: the information must be readable to the human and yet the human has a great facility for being able to

interpret the information. This allows a high degree of flexibility in the format of the information as it is presented and also there must be very flexible ways of accepting input. We no longer have experienced keypunch operators putting data into exactly the right set of character positions in the input record. It is now becoming increasingly important to have data identified by such things as labels rather than specific positioning in a record.

One of the objectives of a time-sharing system is to get the users thinking-bound. Thus, the user would like to delegate as much as possible to the compiler and generated object code. With his initial declaration of variable name, mode and dimension the user should not have to be concerned with such things as mixed modes and output formats. However, we still must allow the user the flexibility of specifying his own formats as he desires. We recognize that users may have different personal desires and, hence, we must provide both a simple and a flexible system.

The following treatment of formats, lists and other features, in order to be general, will be kept relatively independent of both the interval representation of information and the configuration of any system with its many possible combinations of components. We would like to speak in terms of simple or complex systems using general or special purpose compilers.

The approach will be to present alternative methods of evaluating formats, setting up the structure of commands and manipulating the lists of variables. With this background, a compiler writer will hopefully be in a better position to cater to the needs of the user and his problems and to take advantage of the particular machine he has to work with--its particular storage facilities and input/output devices.

FORMATS AND LISTS

In carrying out input and output operations in a higher-level language a LIST is used to specify what is to be transmitted and to signal the end of the operation. The actual operation is under the control of a set of FØRMAT specifications whether explicit or assumed. It is convenient to think of the set of FØRMAT specifications as being a program which has as its data an explicit or implied list of variables and constants.

The FØRMAT language can consist of "commands" which operate at the micro level by manipulating characters in a buffer or at the macro level by dealing with fields of characters. At run time the FØRMAT PRØGRAM can be executed interpretively or it can be a relatively frozen block of subject code generated by the compiler. The LIST of variables may consist of ordered or unordered data; it may be dealt with in free-form or under the explicit control of the programmer. In the following sections we will deal with these three areas and then evaluate them in the light of an interactive compiler and time-sharing.

The design of a compiler must be done within the framework and devinition of what is given--the macro language, and what is wanted in the form of object code and tables subject to the machine characteristics and components. Once these two end points are clearly defined then the job of design is easier and can be accomplished at a more concrete level. It is essentially a job of transforming the information given at the macro level into a procedure to be used at the machine level. The transform operator (the compiler) cannot be designed until the two sets at each end are clearly specified.

INPUT LISTS

Conventionally, input to a computer program has been in a fixed form. This was not unreasonable in the light of batch processing, which was often under the control of a monitor system,

and to retain simplicity. However, with today's applications and the increased use of on-line computation, freer and more flexible forms of input expression are desirable and necessary.

FORMAT-FREE input consists of data contained in arbitrary fields and to an arbitrary number of figures.

Data which can be ORDER INDEPENDENT goes even one step further. In this case the data must carry with it information which will identify it to both the user and the program. This would consist of a name or label which would appear in the input record along with the data. This method of input is especially desirable if the data does not possess any natural ordering as would be the case with elements of a matrix for example. It should not be necessary to require the programmer to use an artificial arrangement. Sometimes each cycle of a program requires a large number of input items but only a small subset would be altered from cycle to cycle. It is even more important if this subset is dependent upon the output of the previous cycle as in on-line simulations or optimization processes. This identification by names also makes the input record more intelligible to the user. He would no longer be faced with endless strings of anonymous numbers.

In the following paragraphs three forms of a read statement are given which could be used to give the programmer simplicity or flexibility, whichever better fits the requirements of the application. These are independent of the input device being referenced but, of course, that would have to be specified. In the case of a time-sharing system they would reference the device used by the programmer--the typewriter keyboard and pointer in the time-sharing system at M.I.T., for example.

READ n, 1

This is the most flexible form of the statement. The n refers to a set of format specifications which define the fields or character positions within the input record. The order of

the data would be that shown in the list of variables to receive values, and would correspond to the order as specified in the format statement.

READ, *1*

In this form the data is still ordered according to the list but now its specific form must be made inherent in the data. The fields in the input record would be identified as to mode and would be separated by appropriate delimiters. An appropriate delimiter specification would be: one comma and/or one or more spaces. For example (where s is used to represent one or more spaces):

/ , / s / ,s / s, / s,s/

would all be acceptable. To specify the mode of the data appropriate identifiers could be used. A period would serve the usual decimal point function; octal or binary information could be subscripted with identifiers such as 0, OCT, BIN, (2), or (8); alphanumeric data could be bracketed by a specified character such as \$ALPHA\$; and floating point data could be identified in the usual way of using E such as -123.79E+7. At object time a check would be made for consistency between the declared mode of the variable (as stored in a symbol table) and the data in the input record. In an on-line system any detected inconsistency would not result in a halting of execution but rather a request for varification and possible re-input of the data.

READ

In its simplest form this statement would allow the user to specify at run time what variables would receive values--the order and form of the input data being completely arbitrary. Once again appropriate delimiters would be used and also each value would have to be identified by the name or label of the variable which would be known both to the user and the program via the symbol table. If a label did not appear in the symbol table, then in an on-line system

the user would be given the opportunity of saying this label is incorrect and should have been such and such or that it is valid and a new entry should be made to the symbol table.

An example of some entries which could appear in the input record would be:

```
X = -7.4, CODE = $ALPHA$,  
ARRAY(5,1) . . . ARRAY(5,3) = 8,1.3,-.091
```

This is analogous to the \$A feature covered in an earlier section of this report with one major difference. The function of an input statement is to assign a value to a variable whereas the \$A would interrupt execution and allow the user to input anything from a single value to a whole sequence of statements. With this command the end of the list must be signalled by the user. This could be accomplished simply by having him hit the carriage return key a second time.

OUTPUT LISTS

When data is on an input record its form is already determined. With the output of data format becomes a greater concern since it is under the control of the user or the user's program. There are times when he would like to receive data in an appealing tabular form and then there are times when he is merely interested in the values of some variables and just wants to receive that information without explicit regard to its format. The latter becomes increasingly desirable with on-line computation where the user is right there to read the information and initialize further action on the basis of this information. There may be many forms of intermediate data of interest only to him, such as would be desired for debugging purposes. Here, the user would like to insert a temporary print command quickly and easily without the corresponding format specification.

Similar to the input commands we propose three forms of a PRINT command which could be incorporated into a higher-level

language. In the case of the time-sharing system at M.I.T. these would refer to the printer on the user's typewriter console.

PRINT n, L

This again is the most flexible form of one command allowing the user to fully specify the format of the output record which is referenced by n. The values of the variables in the list, L, would be "fitted" in sequence into the format specifications.

PRINT, L

In this form the user is freed from having to specify the format of the data to be output. At run time the program (the code for such a program would be inserted by the compiler at time of compilation if this format-free PRINT command appeared anywhere in the user's program) would look at the symbol table and the current value of the variable in order to establish the format under which the data will be printed. The symbol table would provide information as to mode and dimension and the value would be used to determine the size of the data.

Analogous to the READ command the data would be printed out along with the symbolic variable names to identify it. An example of such output might be:

```
X = -7.4  
CODE = ALPHA  
ARRAY(5,1) = 8.  
N(22) = -.14230000E + 12  
BØØL = OE
```

Not that if a variable happens to be declared in the alphanumeric mode then it would also be output as shown above. If the value of a decimal variable is within a certain range (say, $10^{-n} < |D| < 10^n$ where n is the maximum number of allowable character print positions), then it would be printed as a fixed point decimal number; otherwise it would be printed as a floating point number with the exponent. It would also be simple to accept implied DO loops in the list and then print out all the values.

PRINT

In this form the statement could have only one meaning--to print out the current values of all the variables listed in the symbol table. This could be helpful in debugging small programs and subroutines.

THE FØRTRAN LANGUAGE

The format language gives the programmer a set of specifications or "commands" which enable him to specify the form in which data is to be output (or received as input). One way, which is a characteristic of FØRTRAN, is to have a set of possible specifications which refer to whole fields to be inserted in an output string. This leads to simplicity but by referring to data at this aggregated level a considerable degree of flexibility is lost. In view of the fact that we can provide a great deal of simplicity to the programmer by having free-form input and output it would now be desirable to increase the flexibility of the language by allowing a more complete and extensive set of commands with which to specify formats.

For complete flexibility the commands can be structured so as to allow the programmer to build up an output record on a character-by-character basis (see ALGOL report, Comm. ACM, May 1964). Within the framework of existing macro languages, such as FØRTRAN, some users have taken great pains to provide more powerful formatting capabilities (see ATLAS paper Number 32, by I. C. Pyle).

In setting up such a language we must think in terms of getting things in and out, character by character, of a buffer string. Each format specification can be thought of as an instruction or operator on this buffer and a sequence of these can be thought of as a Format Program. The other essential item to be associated with the buffer is a pointer and each operation may affect the position of this pointer. This pointer would point to some character position in the buffer string.

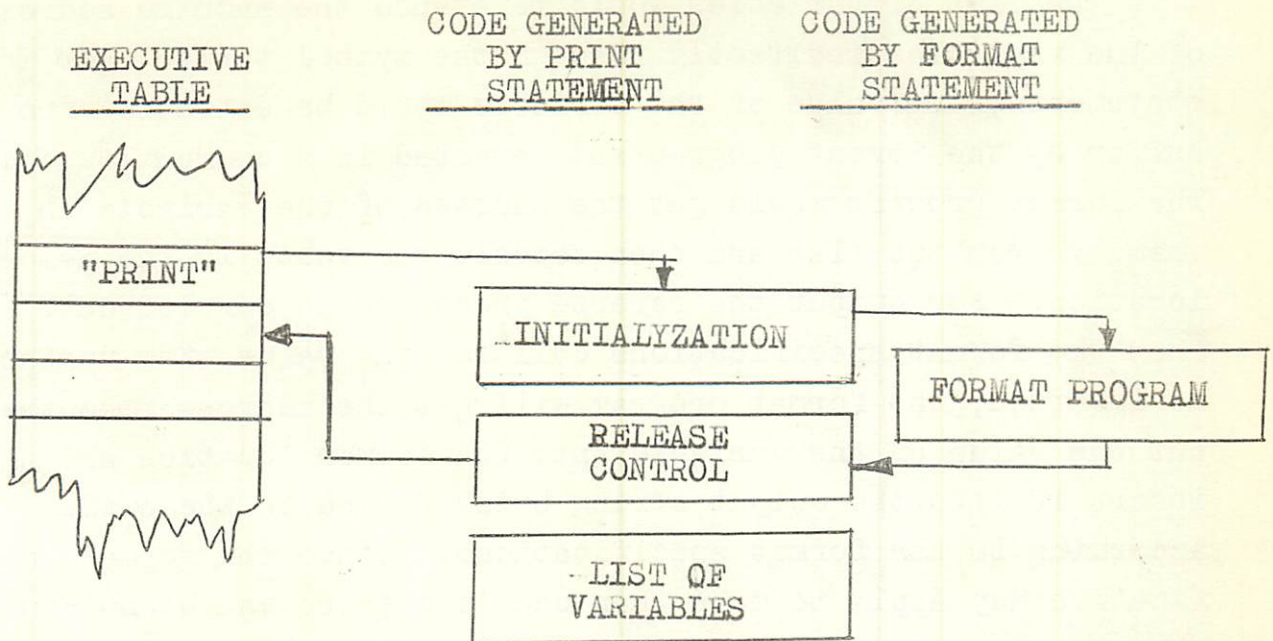
The format program would operate on the string of character positions represented by the buffer on a one-line-at-a-time basis. Initially this buffer would be set to blanks at run time. Then some primitive operations would be set up to operate on the character positions or the pointer. Such commands as reset or update the pointer, insert the digits in the value of the variable, sign control, insertion of such characters as a dollar sign and other editing functions.

The purpose of this paper is not to attempt to completely specify such a format language, but it is important to emphasize what should be considered and possibly how one would go about setting down the specifications and within what framework.

THE GENERATED OBJECT CODE

We have discussed what sort of language could be used at the macro level. Before we can design a compiler we must specify what we are trying to produce in the way of machine coded procedures. The code which is to be generated for the READ and PRINT statements must be rather modular in form--in other words, we want to generate a quantum of code for each statement. In the case of the input-output statements the primary block of code generated would be that represented by the FORMAT specification, either explicit or implied. At run time this block of code would reference another block of code generated in place of the list, which in turn would reference the symbol table.

The following diagram, (Fig. 1) is schematic of the generated blocks of code. The blocks shown in a vertical sequence would be somewhat contiguous in the memory of the computer.



(Fig. 1)

In the executive table a branch to the initialization block would transfer control to begin the execution of the input-output statement. At the point of initialization a number of things would be done.

1. Attach a buffer to format statement to be used in setting up the output string of characters or to accept the input string.
2. A link would be set up to the buffer associated with the specific input-output unit. This buffer would be used to accept completed data records for input or output (see later section on buffers). Within the time-sharing system at M.I.T. this buffer is controlled within the system and hence the generated object code for this user would not have to be concerned with this buffer.
3. A pointer would be set to point to the first element in the list of variables.
4. Then a branch to the referenced format statement would be made. This reference could be made through a table of macro-language statement numbers or, the reference could be inserted at compilation time. If the format is implied rather than explicitly specified by the user, then reference to a standard block of code would be made. This standard block of code would be inserted at compilation time.

The list of variables would reference the machine addresses of the variables indirectly through the symbol table. For input statements, the value of the variable would be extracted from the buffer by the format program and inserted in a common location. The format program would get the address of the variable in memory from the list and then deposit the value in the memory location. For output the reverse procedure is carried out. When the format specifications call for the value of a variable to be inserted, the format program will get the address from the list, put the value of the variable into the common location and then insert it into the output string being formed in the buffer according to the format specifications. Since the format specification may apply to more than one list there can be no explicit correspondence between them. This is accomplished by means of the common location to hold the value of the variables.

Once the initialization procedure is carried out the format controls the entire operation. It prepares the output string or examines an input string according to the set of format specifications and references the list of variables when a value is called for or when a value is ready to be deposited in memory. When the last item in the list of variables has been dealt with the format program releases control back to the executive routine to begin execution of the next instruction in the higher level language.

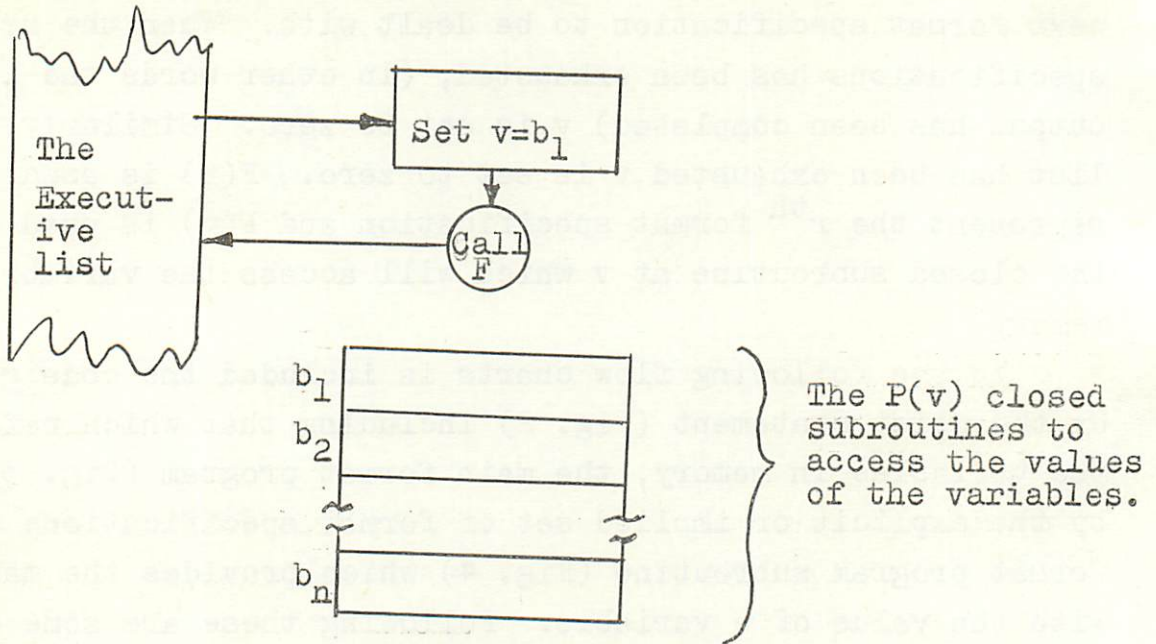
In order to clarify the functioning of these blocks of code we will present a possible scheme for the execution of a print command using relatively fixed object code (see A. J. Perlis, "A Format Language").

In a fixed location (v) of the Format program is stored the address b_1 which is the initial address of the sequence of instructions in the list of variables block. Then the format program calls on a subroutine which uses the contents of (v), in other words b_1 , to access the current value of the next variable to be output. At compilation time the list of variables is constructed and consists of a succession of closed subroutines—one

for each variable. In the format program v is the address of the next format specification to be dealt with. When the set of specifications has been exhausted, (in other words the line of output has been completed) v is set to zero. Similarly, when the list has been exhausted v is set to zero. $F(r)$ is used to represent the r^{th} format specification and $P(v)$ is used to represent the closed subroutine at v which will access the variable in memory.

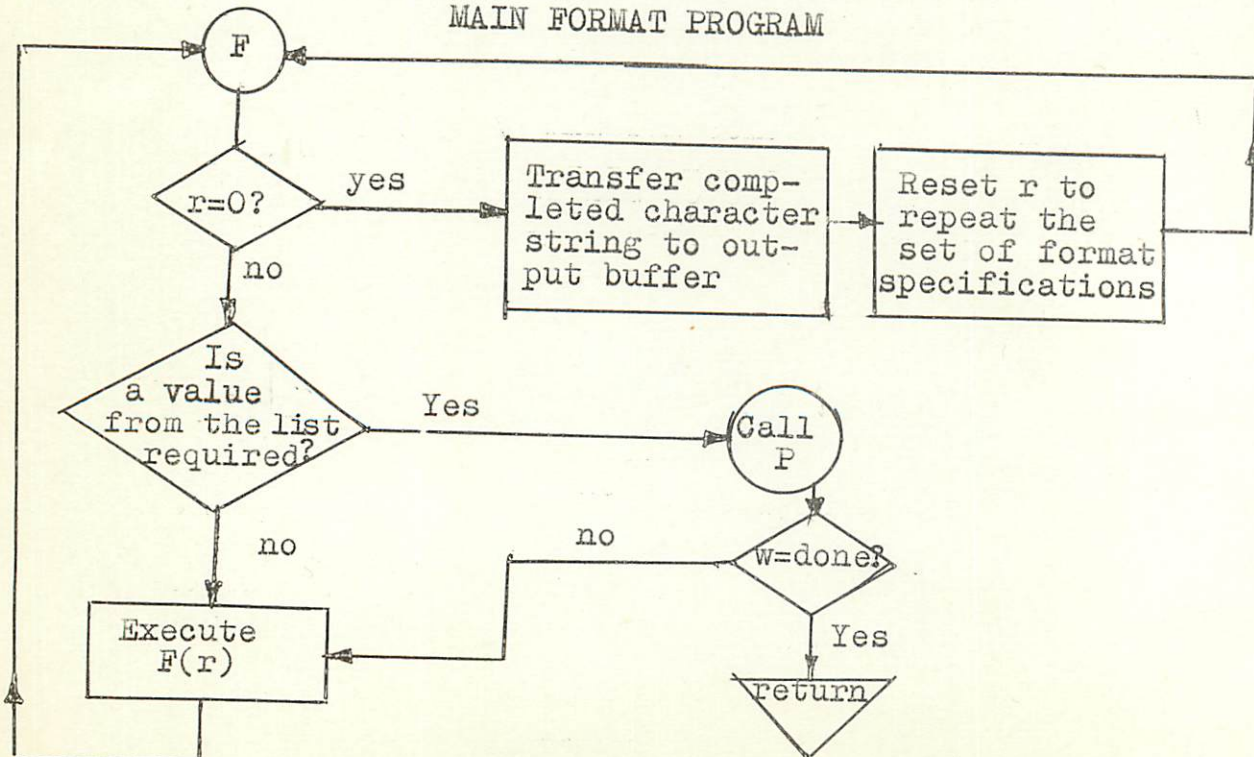
In the following flow charts is included the code generated by the print statement (Fig. 2) including that which references the variables in memory, the main format program (Fig. 3) generated by the explicit or implied set of format specifications and the format program subroutine (Fig. 4) which provides the main program with the value of a variable. Following these are some examples of $P(v)$ code that would be included in the piled list of closed subroutines representing the list of variables. (Figures 5, 6 and 7).

INITIALIZATION AND LIST OF VARIABLES



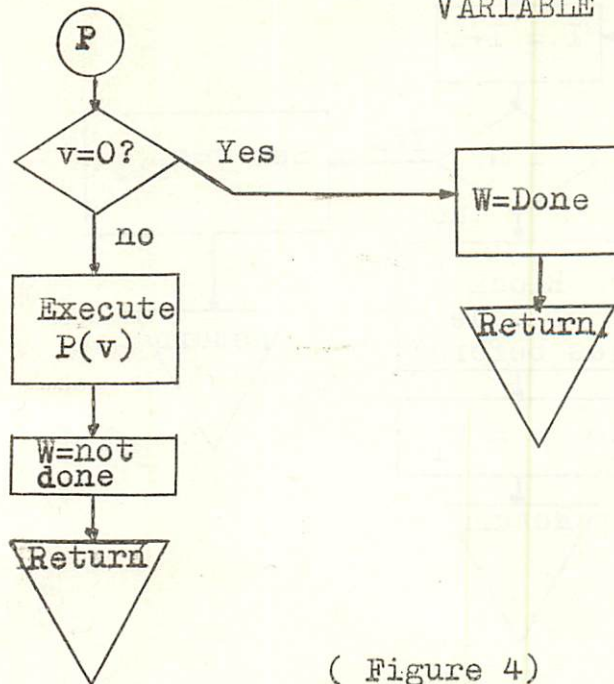
(Figure 2)

MAIN FORMAT PROGRAM



(Figure 3)

FORMAT PROGRAM SUBROUTIN
TO ACCESS VALUE OF THE
VARIABLE



(Figure 4)

P(v) SUBROUTINES TO ACCESS THE
VALUE OF THE VARIABLE

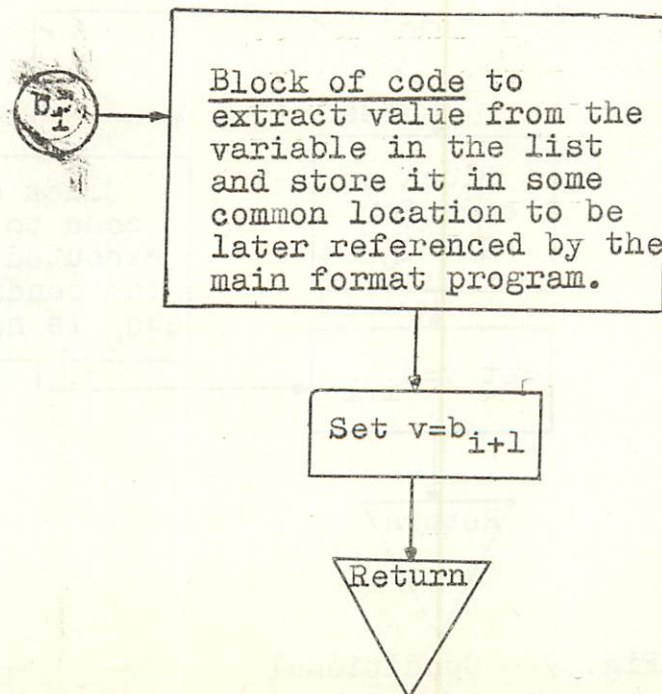
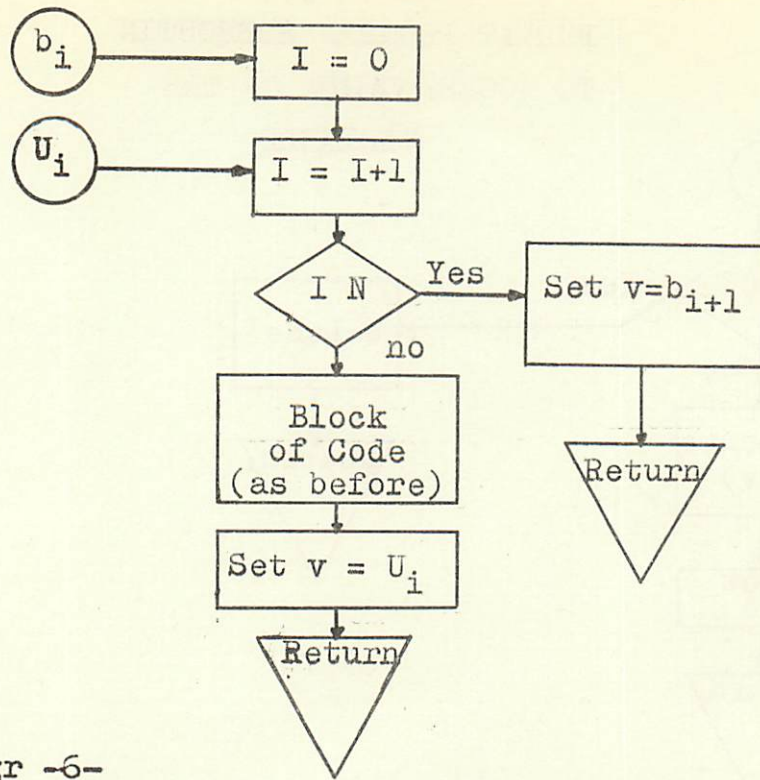


Fig. 5 - Simple Case



Figr -6-

Repetition of Dimensioned variable in the list

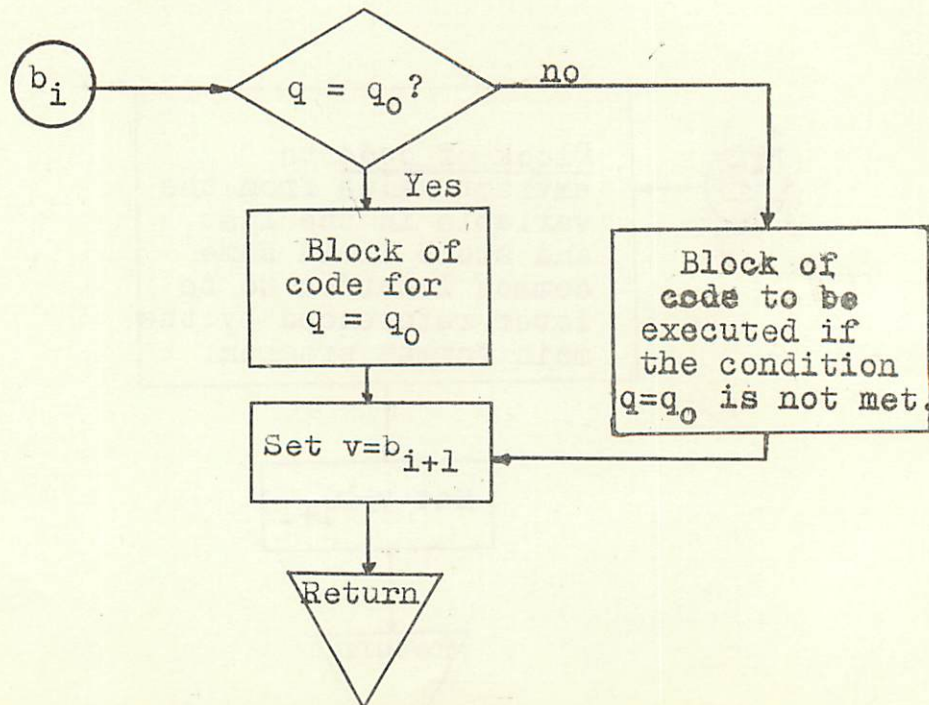


Fig. 7 - Conditional

The scheme which has just been covered would be entirely satisfactory if no changes in the statements or lists were anticipated. The code which is generated is relatively frozen at run time and hence modification is difficult. Within an interactive scheme the basic framework is valid and useful but some modifications are needed. The first thing that should be done is to make use of the symbol table and the information it contains. This would eliminate the need for the set of closed subroutines $P(v)$ and also make additions or deletions to the list much easier at run time.

The next problem which faces us at the interactive level is that of modifications to the set of format specifications at run time. There are two modes of representing the format program in memory (the $F(v)$ routines as in our preceeding diagrams). One way is to generate object code at compilation time. This may be rather efficient at run time but it doesn't provide us with enough flexibility to modify the format specifications at run time. In order to facilitate this requirement it would be better to execute the format statement interpretively. The format specifications themselves would be retained in memory to be interpreted and executed at run time. This only applies to explicitly specified formats. In the case of the implicit format statements one would have a choice between inserting fixed object code or merely inserting the appropriate specifications into the set to be interpreted. A decision as to which approach would be the most efficient will not be made here. It would be desirable to try them both on the implied formats and actually see on a computer which is best from the standpoint of speed and space.

So far we have talked mainly concerning output and this is the most important use of the format statement. With an input statement the format specifications are used to provide a "picture" of the input record. The chart in Fig. 1 can still be used to show how the code would be set up to execute a read command. Here we must add to the initialization block the physical reading

of the input record into the buffer. Then the format program would appear as in Fig. 8.(See p. 23).

We now have the facility of changing a format to a print statement and it would be desirable to have an analogous operation for the read statement. This could be accomplished by providing a command which will re-examine an input record. A set of such commands would be as follows:

REREAD n, L

REREAD , L

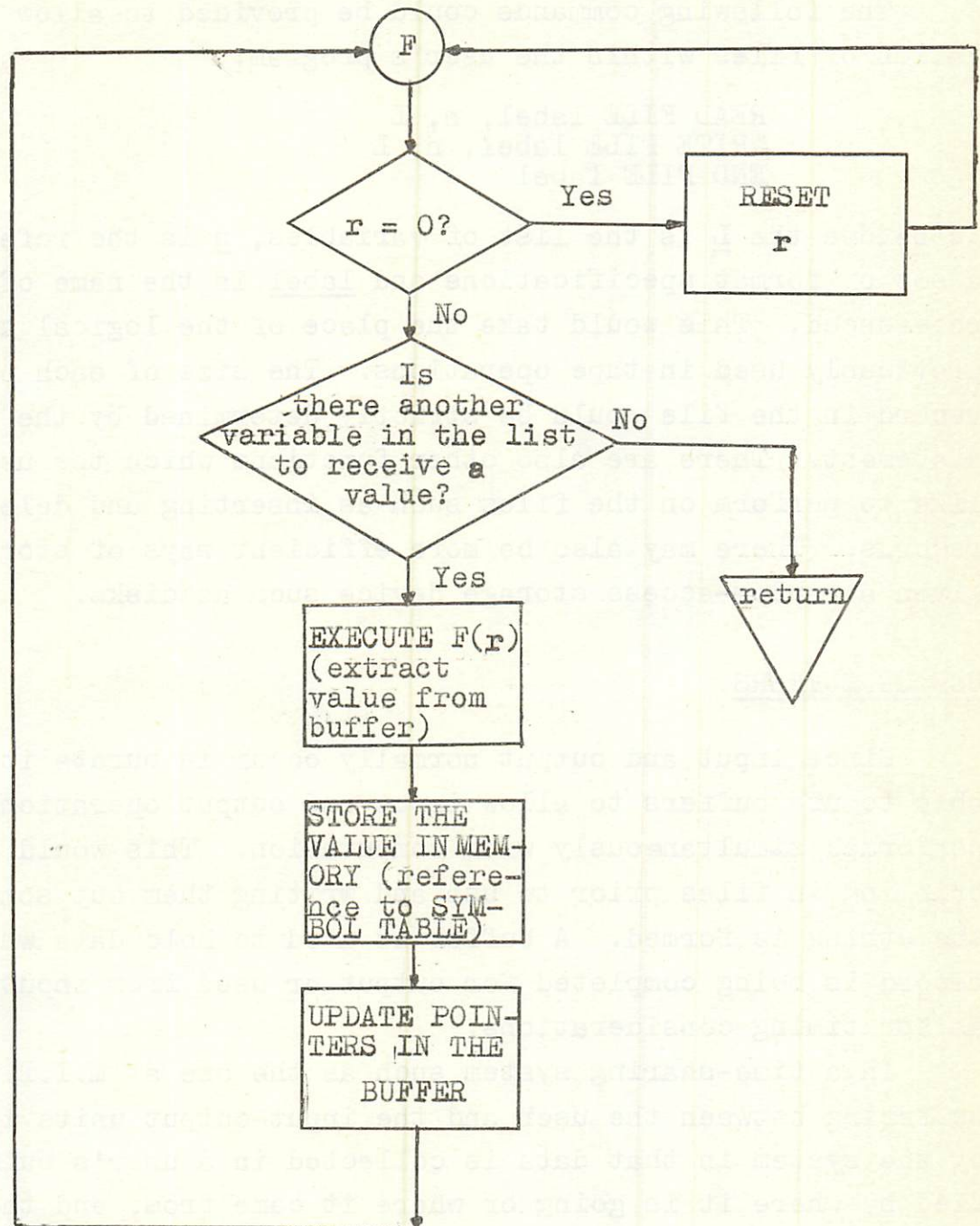
These commands would function exactly as the corresponding READ commands except that no physical read takes place. This would allow the programmer to reopen an input record and extract information from it. If a mixture of records were being input, each with a code identifying it, the user could read once, identify the type of input record, and then reread under the appropriate format.

FILE COMMANDS

So far we have concentrated on the higher-level language commands to accomplish input and output with reference to the user's device--namely, a typewriter keyboard and printer. In the specifications, we have allowed both flexibility and simplicity. The second major consideration in the input-output facility of the language must be the reading and writing of files.

As was mentioned earlier the user need not concern himself with the physical units being used for input and output and for secondary storage; he is more concerned with the storing and accessing of files. Since, in general, these files will be written and subsequently read under control of a program it is reasonable to require that the programmer explicitly provide a set of format specifications for each record transmitted in the file.

FORMAT PROGRAM - INPUT



(Figure 8)

The following commands could be provided to allow the manipulation of files within the user's program.

```
READ FILE label, n, L
WRITE FILE label, n, L
END FILE label
```

As before the L is the list of variables, n is the reference to a set of format specifications and label is the name of the file referenced. This would take the place of the logical tape number previously used in tape operations. The size of each physical record in the file would be strictly determined by the format statement. There are also other functions which the user would like to perform on the files such as inserting and deleting records. There may also be more efficient ways of storing files given a random-access storage device such as disks.

USE OF BUFFERS

Since input and output normally occur in burata it is desirable to use buffers to allow input and output operations to be performed simultaneously with computation. This would facilitate bringing in files prior to use and writing them out sometime after the string is formed. A buffer is used to hold data while a record is being completed for output or used from input, as well as for timing considerations.

In a time-sharing system such as the one at M.I.T. this buffering between the user and the input-output units is handled by the system in that data is collected in a user's buffer identified by where it is going or where it came from, and then, when the physical record is formed, it is transmitted by the system. However, when the compiler is being designed to work within a less structured environment it is necessary to understand the use of buffers and how they can be set up.

This requires 1) a somewhat standardized set of record formats within each file, 2) a set of internal tables describing the

current status of internal buffers and the buffers themselves, and 3) a set of routines which operate on these records and internal tables and the particular input-output unit. Each of these routines should be independent--in other words, refer to each other only through the internal tables, the buffers and the data records. This leaves the system open-ended for possible expansion to include other input-output units.

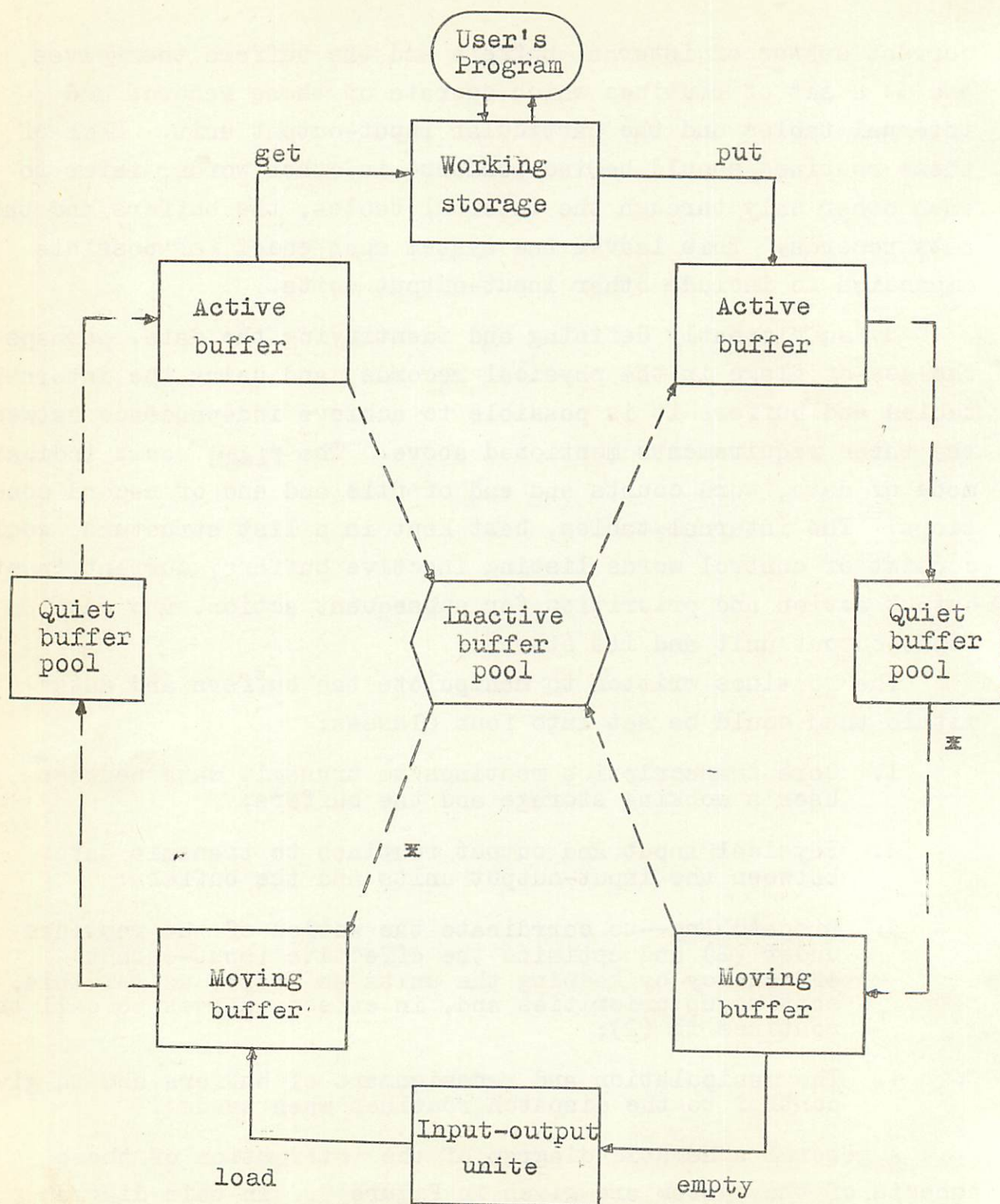
By sufficiently defining and identifying the data, perhaps by the use of flags in the physical records, and using the internal tables and buffers it is possible to achieve independence between the three requirements mentioned above. The **flags** would indicate mode of data, word counts and end of file and end of record conditions. The internal tables, best kept in a list structure, would consist of control words listing inactive buffers, current input-output action and priorities for subsequent action, and each input-output unit and its status.

The routines written to manipulate the buffers and data within them could be set into four classes:

1. Core transcription routines to transmit data between user's working storage and the buffers;
2. Physical input and output routines to transmit data between the input-output units and the buffers;
3. Dispatching--to coordinate the action of the routines under (2) and optimize the effective input-output efficiency by keeping the units as active as possible, setting up priorities and, in effect, serves to call the routines in (2);
4. The manipulation and reassignment of buffers and to give control to the dispatch routines when needed.

A general schematic diagram of the interaction of these aspects of the system are given in Figure 9. In this diagram working storage is used to hold program, data, intermediate and final results. A buffer is active when the user's program is in the process of transmitting data. A buffer is classed as a moving

-44-
THE BUFFER SYSTEM



Data flow —————→
 Buffer flow with data ———→
 Buffer flow without data - - - ->

buffer when it is currently being operated on by the hardware that controls the action of the particular input-output unit. A buffer enters the quiet buffer pool when it contains data which is currently awaiting use.

This explanation of buffers and how they can be incorporated into the system of machine procedures generated by the compiler is in no way exhaustive and is not completely defined. It is given to provide the compiler designer **on basic** understanding of some of the problems which must be considered and a possible approach to a solution.

Remarks on FORTRAN INPUT-OUTPUT

Most people who have used FORTRAN are keenly aware of some of its shortcomings, which become even more evident to programmers who have used it as a higher-level programming language within a time-sharing system such as the one in use at M.I.T. In relation to input-output we will indicate the more pressing problems and possible solutions that could be implemented in the short-run.

The first is the need for explicit statement of format on input and output records. It would be desirable to provide statements which will output data in free-form, and accept data in free form. The format language provides a low degree of flexibility since it deals with fields and not characters. This is somewhat of a compromise between no free-form and no character manipulation. Within the format language the Hollerith specification is the greatest source of error and frustration in the entire FORTRAN language. Here it is necessary to insert a count of the numbers of characters in the Hollerith field before the field has even been typed on the console. This is clearly redundant information and one solution would be to immediately follow the H specification with a defining character. The second appearance of this character in the string after the H would signal the end of the Hollerith field. For example

HOTHIS IS A HOLLERITH FIELD@

where 0 represents any character chosen by the programmer at the time of writing this particular hollerith field.

CONCLUSION

It is hoped that this paper can serve as a guide to programmers who are designing an Interactive Algebraic Compiler and perhaps the associated higher-level language. In relation to input-output, once the language is defined and the general set-up of machine language procedures has been established the programmer can take into account the existing system with which he has to work--its internal structure, external components and the programming system (time-sharing)--and begin to design an interactive compiler to link these segments together into a useful system.

REFERENCES

- "FORMAT-FREE Input In FORTRAN", Bailey, Baruett, Futrelle of M.I.T. Cooperative Computing Laboratory, Communications AOM, Vol. 6, No. 10, (October, 1963), p. 605.
- "A Format Language", A. J. Perlis, Communications ACM, Vol. 7 No. 2, (February, 1964), p. 89.
- "Simple I/O MAD Statements", B. A. Galler, Letter to the editor, Communications ACM, Vol. 7, No. 1 (January, 1964), p. 3.
- "A Proposal for Input-Output Conventions in ALGOL 60", A report of the subcommittee on ALGOL of the ACM Programming Languages Committee, D. E. Knuth, Chairman, Communications ACM, Vol. 7, No. 5, (May, 1964), p. 273-283.
- "Input-Output in Atlas Fortran", I. C. Pyle, Atlas Paper Number 32.
- "The Share 709 System: Programmed Input-Output Buffering", O. Mock and C. J. Swift, Journal ACM, 1959, No. 2, p. 145.
- "Input-Output Buffering in Fortran", D. E. Ferguson, Journal ACM, January, 1960, p. 1.
- "Fortran General Information Manual", International Business Machines, C28-6100-2.
- "7070 Input-Output Control System, Programming Systems Analysis Guide", International Business Machines, C28-6119.
- "The Compatible Time-Sharing System: A Programmer's Guide" F. J. Corbato et al at the M.I.T. Computation Center. The M.I.T. Press, Cambridge, Mass., 1963.

Organization of an Executive
Routine for the object
Program

This section will cover those features of the object program which will result from compilation with the interactive algebraic system under discussion here. The first part will discuss the construction of the executive table and the symbol table, the second section will cover the procedures to be used by the compiler for handling alterations at object time, and, finally, the third section, written by D. Thornhill, discusses the compilation of iterative DO loops using the one-pass, incremental compilation approach.

I. The Executive and Symbol Tables.

The object program will be in the form of small quanta, each of which will represent one source program statement. To implement the various on-line features of the proposed system, each quantum will contain test and branch instructions to certain interactive routines, such as those involved in source program tracing, on-line entry of formulae (the \$ prefix), the entry of variable format information, and the entry of any other on-line information required during execution.

For each quantum of code there will be an entry in an executive program, which will consist, basically, of a list of all of the quanta with their respective absolute (or subprogram relocatable) addresses. Each entry in the executive table contains the following elements:

- Line number of the source statement
- Source statement number, if any
- Address of the corresponding quantum.

The first word in each quantum contains

- Statement type code
- Size of quantum
- Address of variable on left of = ,
if any.

The statement type code defines the various categories of source statement varieties, and will be useful in tracing as well as in execution and alteration of the program. A set of such categories is suggested below:

Non-executable statements: FORMAT, DIMENSION, EQUIVALENCE,
COMMON, END

CONTINUE, PAUSE, STOP

Arithmetic Statements : $A = D + C$. . .

Conditional and Unconditional branches: IF's and GO TO's
DO statements

All executable input-output statements.

CALL, SUBROUTINE, FUNCTION, RETURN

When a source program is first compiled by the system, the executive table has a one-to-one correspondence with the source program. That is, for each statement in the source program, there will be an entry in the executive table in the corresponding position. All exits, whether there are one or more than one, are referenced to an entry in the executive table. And, since the address of the new quantum is contained in the address field of the word in the table, the transfer can be accomplished quite efficiently by a flagged transfer instruction from the old quantum, on the next entry in the executive table, to the first executable six instruction of the new quantum. Thus, the compiler need not maintain a record of the addresses it assigns to each individual machine instruction; it must be concerned only with the addresses of the executive table entries and of the beginning of each executable quantum of code.

Thus, the program may be compiled in one pass, without reference to any instruction addresses other than those corresponding to source statement images in the executive table. Thus, a GO TO 30 quantum would have as its link address the address of the table entry for statement 30, and would not have to be concerned as to the exact whereabouts of statement 30 itself.

Some more specific discussion of the executive table itself is in order here. First, note that the quantum concept is used in this system for the purpose of providing both the interactive and alteration features, as well as providing an efficient means of obtaining a one-pass compilation. Some of the advantages of this approach for one-pass compilation will be discussed in the next section, on the compilation of DO loops. This section will provide some illustration of this concept in the handling of the alteration features of the system.

When a programmer first enters his source statements, each will be assigned an internal line number, which will start at 00010 and will be incremented in units of 10. The programmer will not have to be concerned with this number. He need be concerned only with the statement number plus n lines in order to define any statement in his program. Incidentally, note that the statement number he will use will refer only to executable statements, and such statements as FORMATS will not necessarily be included in his count. An executive table will be set up, as well as a symbol table, as the source statements are entered. The executive table may be considered to be a symbol table for the source statement labels, since the entries equate the statement label, if any, with the appropriate memory location denoting the first executable instruction in a quantum of code. Similarly, the symbol table, which will contain necessary information on the variables and constants in the program, will be built up as the program is entered. The symbol table should contain the following elements:

- BCD representation of the variable or constant name, e.g., PRICE, J, 3.2.
- The Mode of the variable, e.g., integer, floating, Boolean, complex, octal, binary.
- Dimension (i,j,k) of the variable; these will be 1,1,1 if the variable in question is not a dimensioned array.
- Storage address of the variable; this will be the first location of an array in the case of dimensioned variable.

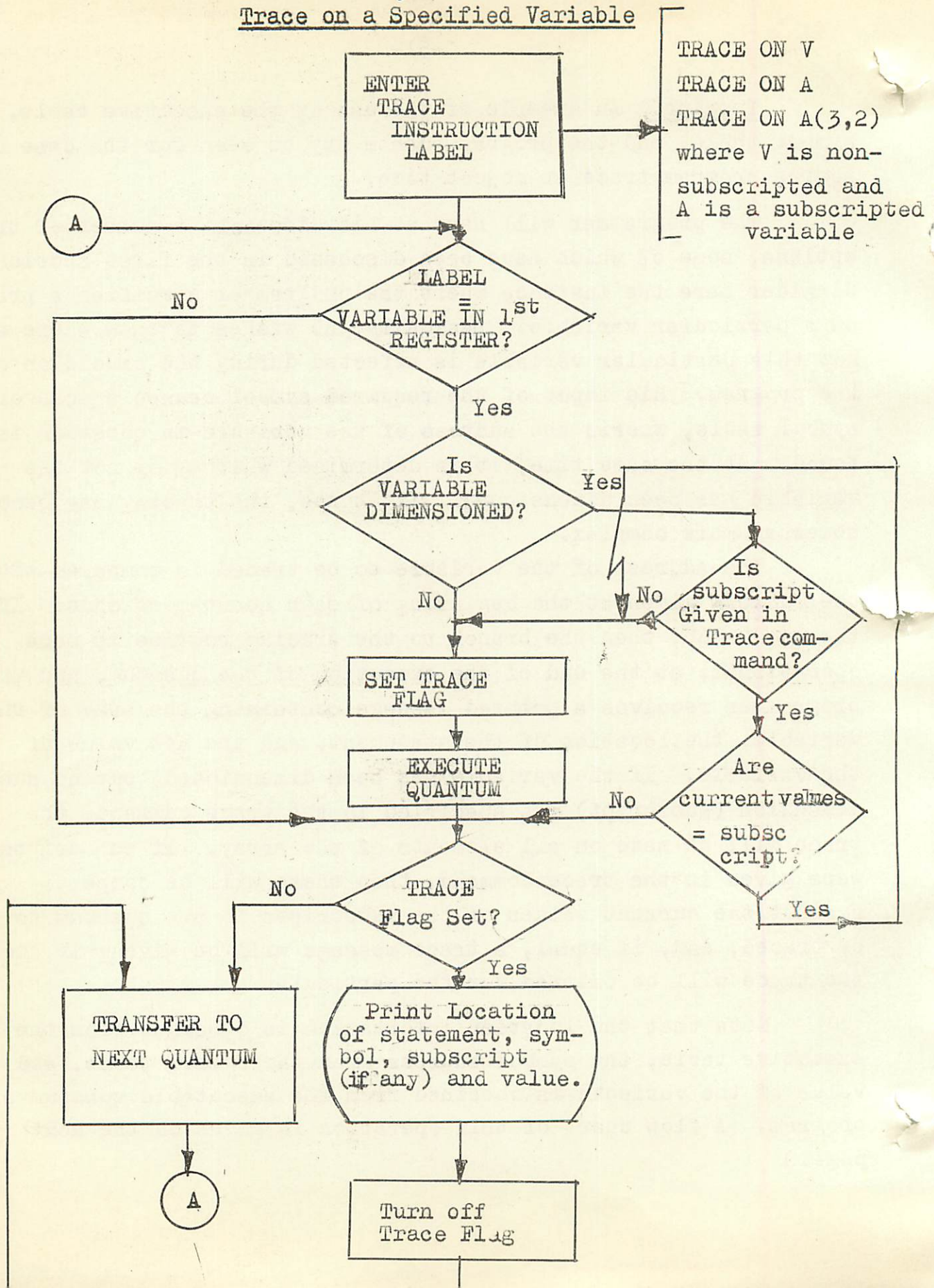
Tracing. An example of the use of the executive table, the symbol table, and the program quantum may be seen for the case of a source program trace at object time.

The programmer will have at his disposal a number of tracing options, some of which have been discussed in the first section. Consider here the instance where the programmer specifies a trace on a particular variable. Basically, he wishes to know where and how this particular variable is affected during the execution of the program. His input of the required symbol causes a scan of the symbol table, wherein the address of the variable in question is found. At the same time, it is determined whether or not the variable has been dimensioned. If it has, the tracing may become somewhat more complex.

The address of the variable to be traced is compared with the address given at the beginning of each quantum of code. If the two match, then the branch to the tracing routine is made operational, at the end of the execution of the quantum, and the programmer receives a printed message containing the name of the variable, the location of the statement, and the new value of the variable. If the variable has been dimensioned, but no such dimension (subscript) was specified in the trace command, the trace will be made on all elements of the array. If subscripts were given in the trace command, then these will be compared against the current values of the subscripts in the quantum to be traced, and, if equal, a trace message will be given; if unequal, the trace will be omitted for the particular quantum.

Note that the location information is obtained from the executive table, the symbol address from the symbol table, and the value of the variable is obtained from the executable quantum of program. (A flow chart of this operation is given on the next page.)

Trace on a Specified Variable



Source program Alterations. Consider next the various types of program alterations that may occur after the initial compilation. The simplest case is the deletion of a previously compiled source program statement, or of a string of such statements. The programmer enters an appropriate DELETE command. The compiler will alter the executive table entries corresponding to the deleted statements to provide a branch to a system NO OP quantum, and will place the locations and sizes of the deleted quanta on an available storage list maintained by the system.

The second type of alteration consists of a change in one existing source program statement. If the alteration does not involve the exit address, all changes occur wholly within the quantum of code, and no change is made to the executive table. If the change involves some change in one or more of the exits from the quantum, then again the changes are made directly to the quantum, replacing the old executive table addresses with the new ones. If the size of the new quantum is smaller than the old one, then the freed storage is placed on the available storage list. If more memory is required by the revised quantum, the compiler will establish whatever links are necessary within the block in the event that it cannot be stored in a continuous sequence in core. Such internal links do not concern and hence are not contained in the executive table. (To some extent, this storage assignment may be made by the loader instead of the compiler. Specifically, the dump and reload package, to be discussed shortly, will reassign storage such that, after reloading an object program, each of its quanta will run in a continuous sequence in memory.)

Finally, consider the last general category of alterations the insertion of a statement or a string of statements in the source program. The programmer enters an INSERT command, defining the particular location in the source program at which he wishes to start his insertion. He then enters one or more new source

program statements, and terminates the insertion by entering an ENDCHANGE command. For each inserted statement, the compiler will follow these procedures:

1. Insert the address of the first free word in the executive table in the exit address of the previous statement quantum ($n_1 + n_2 - 1$) if this previous statement was not a branch or an object of a DO.
2. If the inserted statement is numbered, a new exit address is placed in any existing branch statements that refer to the newly inserted statement number.
3. Create new quanta and executive table entries for the inserted statements, according to the availability of storage as given in the available storage list. This list will be updated accordingly.
4. When the final statement has been entered for a given INSERT instruction, as detected by the insertion of an ENDCHANGE command, the compiler will place the executive table address of the next sequential quantum as the exit address of the last statement quantum entered by the programmer.
5. Multiple address exits are handled in the same manner as for original statements and alterations, as given above.
6. Assign to each inserted statement a line number that lies between the two numbers corresponding to the statements on either side of the inserted statement. Since the programmer is not concerned with these number, some automatic renumbering may be used if necessary.

General Description of the Dump and Loader Programs

When source program statements are first entered by the programmer and compiled by the computer, the executive table and quanta in core are in the same sequence as the original source program. And, if no changes are made in this program after this initial compilation, the coding will remain in this same order. However, if alterations are made, then the ordering of the object code may be affected.

Deletions will cause "holes" in both the executive table and in the object code of the quanta. Alterations of specific statements may result in no change in sequence, but will, in most instances, result in either holes in the object code or in disjoint quanta. Insertions will cause breaks in the sequence of both the executive table and the string of program blocks.

From the point of view of object time efficiency, it will be necessary to reorder the entire program every time it is dumped and reloaded into the computer. After these operations, the object code should appear as if the current version of the Program had just been entered.

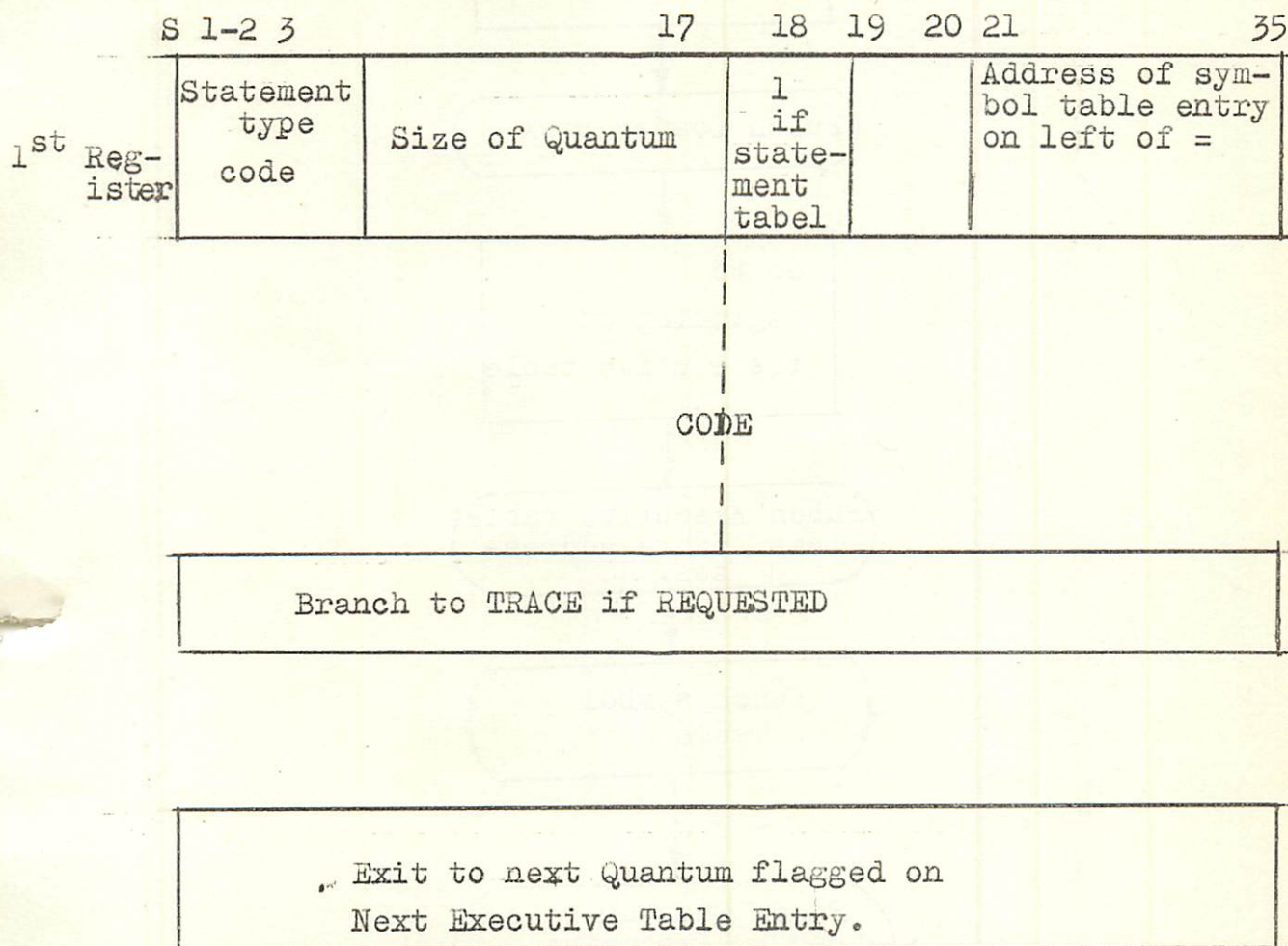
The reordering is accomplished by the dump routine, which may also be used to obtain a permanent copy of the object code. First, the executive table is sorted according to the line numbers contained in each entry. Then, the dump routine proceeds sequentially through the table, picking up the corresponding quanta in their correct order. Where a particular quantum is disjoint, the dump routine picks up a flag indicating this fact, and, by picking up the size of the total quantum and of that part up to the first break point, it is able to write out the entire quantum in a single unit form. When a disjoint quantum is being dumped, the chaining register, which connects the two disjoint sections, is not written out, and all addresses in the disjoint portion(s) are altered to correspond to local relocatable form,

referenced on the first register of the first section of the disjoint quantum. The cumped object program appears as follows:

1. Loader
2. Executive table, symbol table, and other reference information
3. Ordored object code quanta
4. Interactive and trace routines
5. System subroutines and other utility programs

The Loader places the executive table in memory, and then proceeds to load the quanta consecutively. The addresses of the starting points of each of the quants are placed in the appropriate position on the executive table. Finally, the interactive and trace routines, and then the system subroutines, are placed in core, and the program is ready for execution.

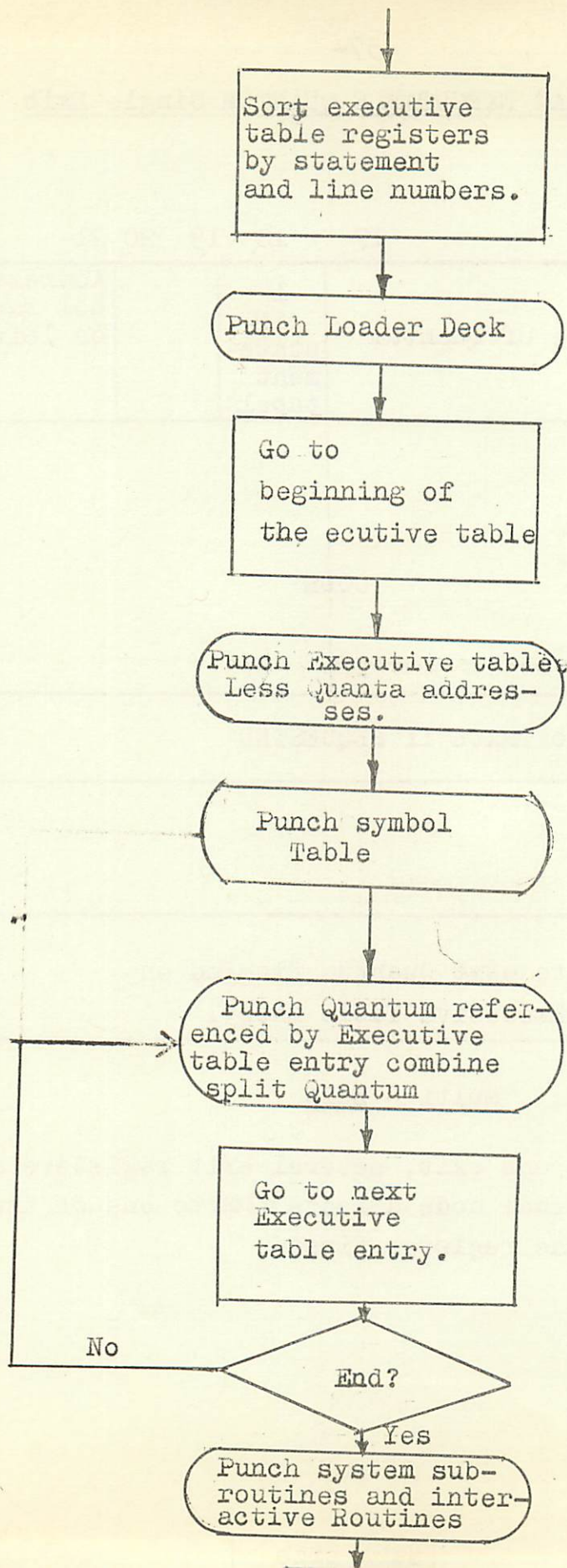
FORMAT OF AN EXECUTABLE QUANTUM Single Exit



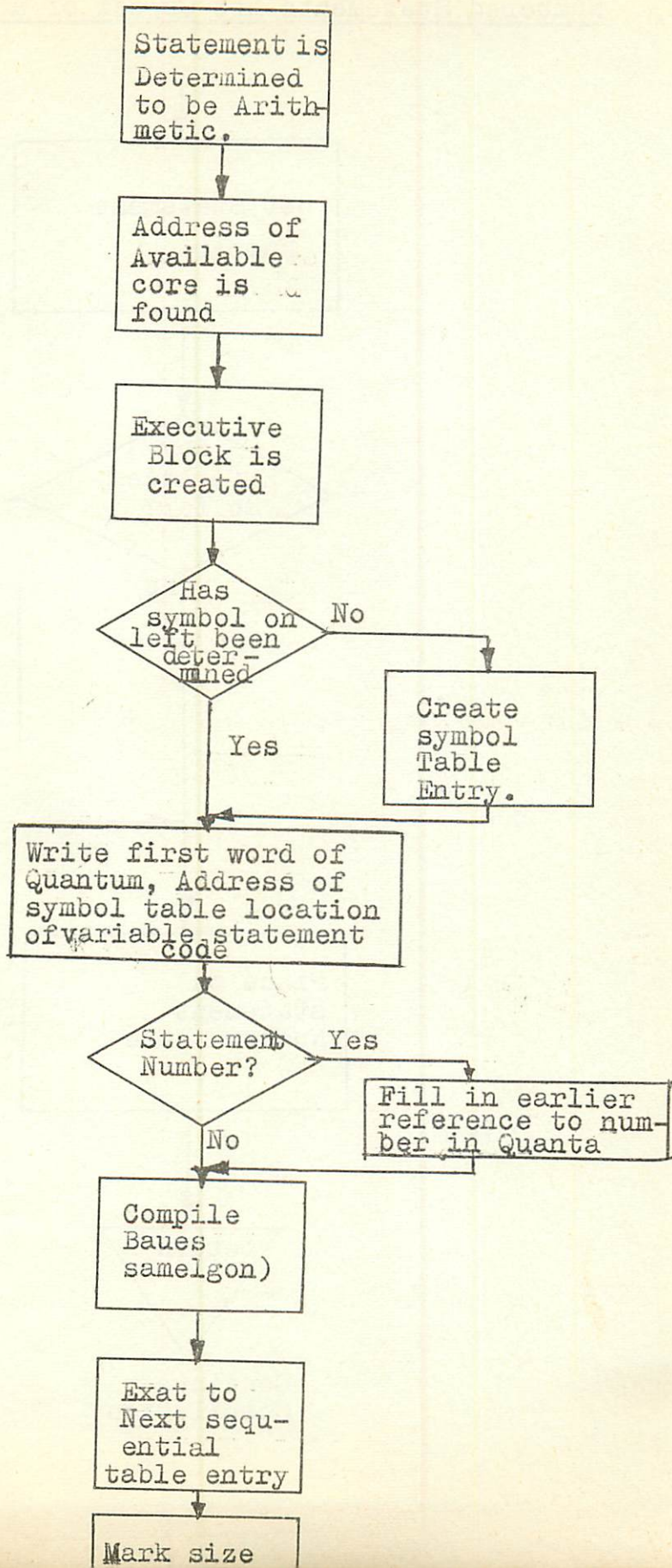
Multi - Exit

If more than one exit, several exit registers appear at end, internal code directs clowto one of them, but checks the register first.

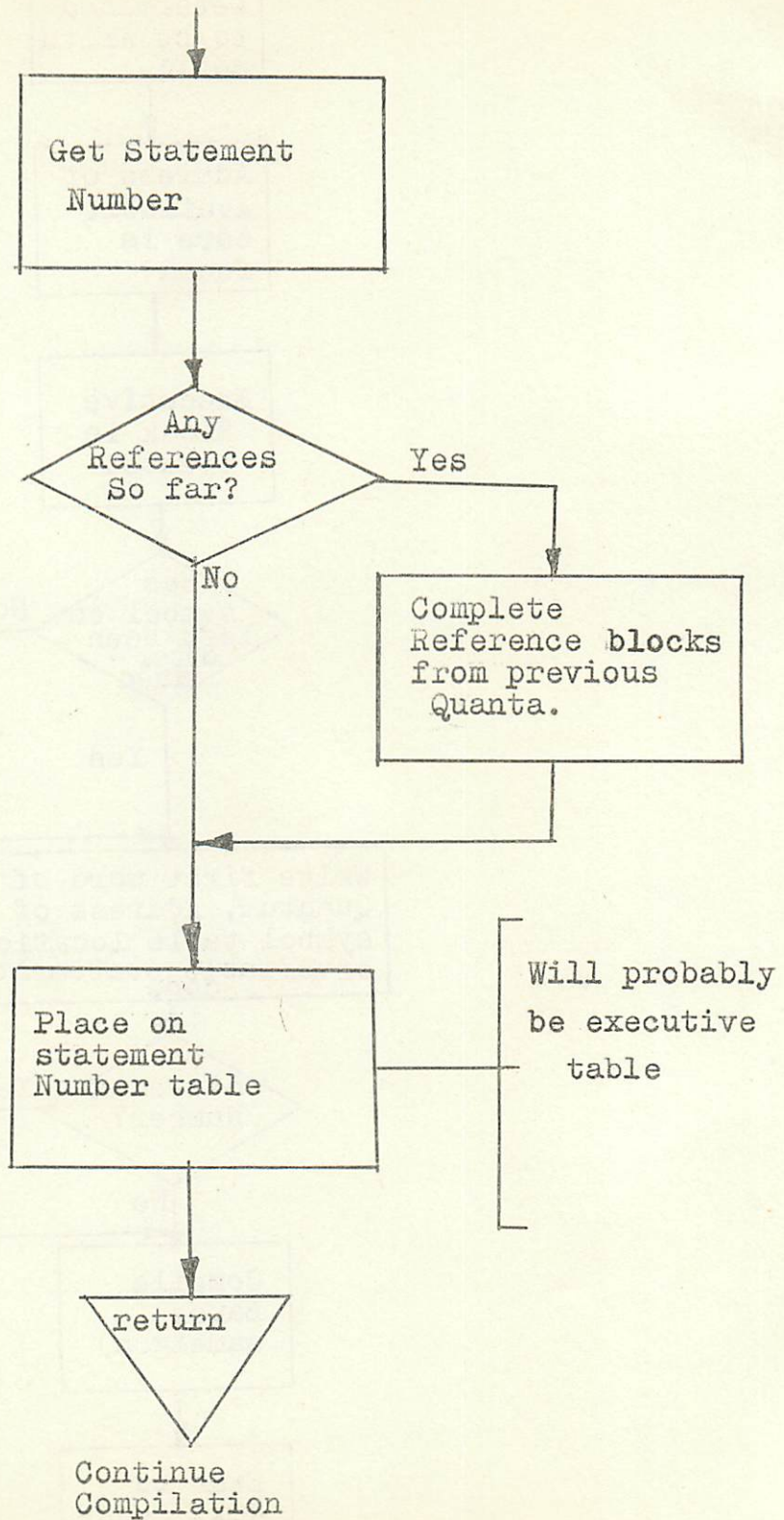
Object Program Dump



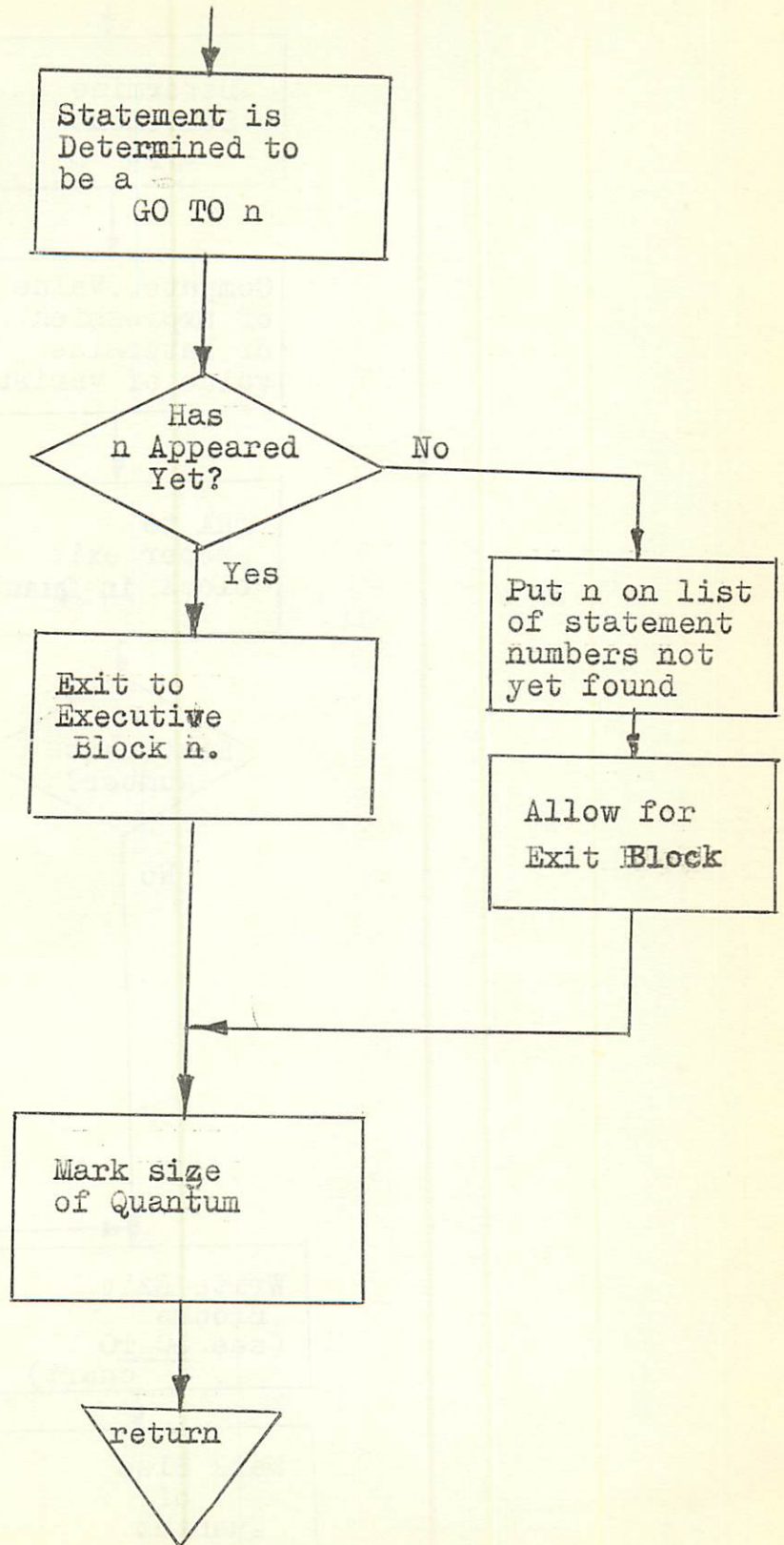
Compilation of Arithmetic Quanta



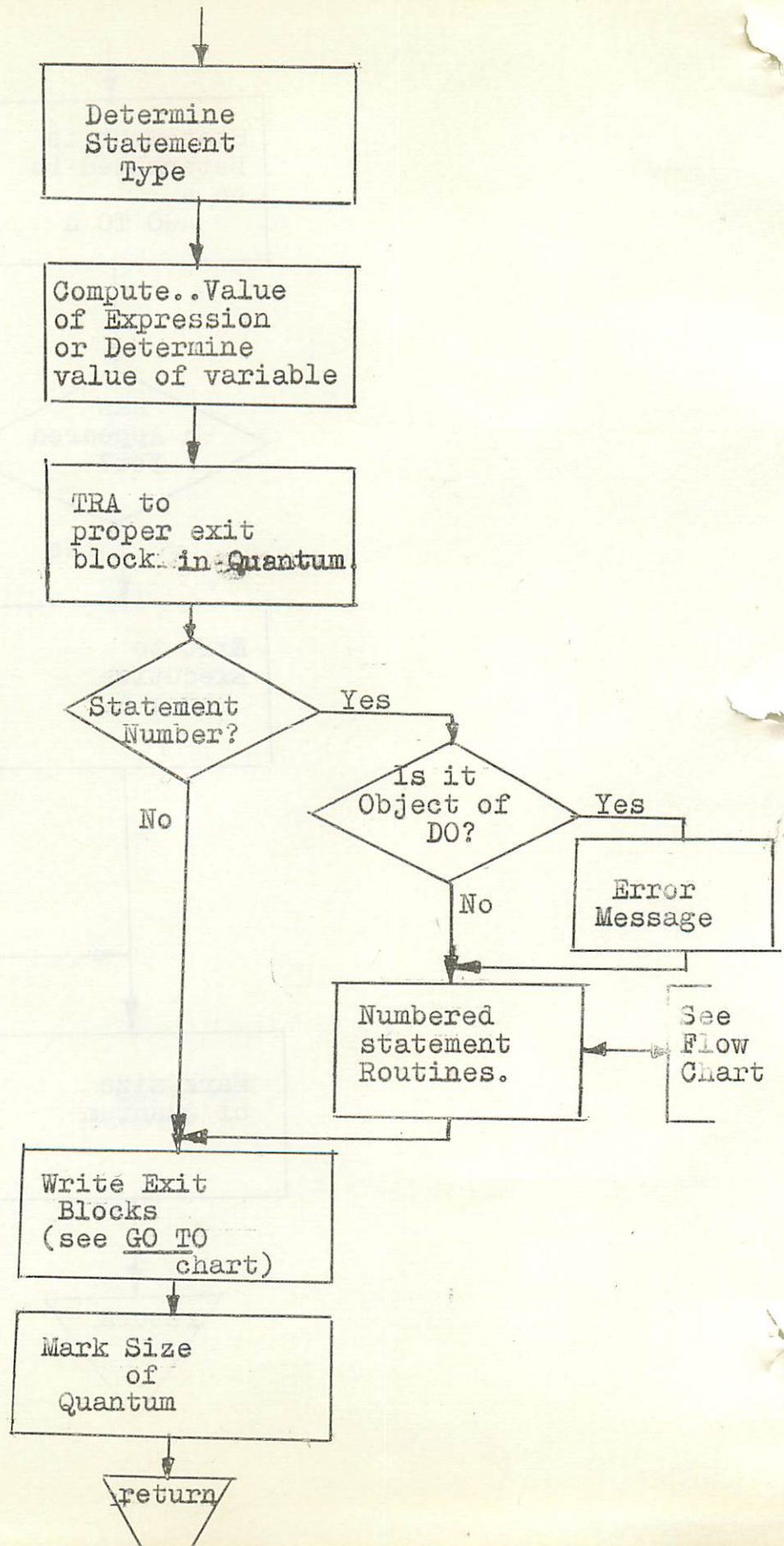
Numbered Statements Not Object of a DO or FORMAT



Unconditional GO TO



IF's and GO TO (n,n, ..., n)i



ONE PASS TRANSLATION
OF FORTRAN DO-LOOPS

ABSTRACT

Fortran Do-loops can be fitted into a scheme for one pass sequential translation by employing a push down list storage. The same techniques can be employed in an incrementally compiled version of Fortran which provides for on-line insertions and deletions of, or changes to, statements. For optimization, the addresses of subscripted variables used within Do-loops should be obtained by recursive address calculation but this is not feasible in a true one pass system.

ONE PASS TRANSLATION OF FORTRAN
DO-LOOPS

I. Introduction

In the process of translating a computer program from programming language into machine language, it is often necessary to have information contained in a subsequent statement in order to translate a given source statement completely. For this reason, most translators use a multipass system, on each pass scanning the program all the way through accumulating information to be used the next time through. The actual machine code is generated on the last pass.

However, for an algebraic language such as Fortran or Algol, a scheme has been devised for compilation using only one pass through the source statement cards. A clear formulation of the elements of this method, called sequential formula translation, has been presented by K. Samuelson and F.L. Bauer.⁽¹⁾ In their approach, not only is the whole program read only once, but there is only one sequential reading of each card, avoiding the scanning back and forth that others have used. To accomplish this, they use three push-down lists for storing symbols, numbers, and addresses. In each case, only the item at the top of the list will be needed next in translation.

One particularly difficult statement to handle in one pass compilation is the iteration statement, which in Fortran is the Do statement.

A Do-loop is a means of executing repeatedly a group of statements in a program until a certain condition for termination is met. There is a counter, called the index, which is given an initial value when the loop is entered and is incremented by a constant each time through the loop. The termination condition may depend on the value of the index. The DO-loop consists of

the DO statement itself, followed by the group of statements which are to be executed repeatedly, called the scope. The last statement in the scope, the target statement, is numbered and its number i is specified by the DO statement in Fortran.

Since the location of the target statement is not known to the compiler when a DO statement is read, a one pass compiler would have to save for use when the target statement is reached any information necessary for terminating the loop. The following sections present a scheme for the handling of Fortran DO-loops in a one pass compiler, with additional information on its application to an incremental compilation scheme and on recursive address calculation in DO-loops.

II. Syntax of the DO statement

The Fortran DO statement takes on the following general form:

DO s FOR i = j, l, k

where

- s: the statement label of the last step in the loop
- i: the variable used as the index for the loop
- j: the initial value for i
- k: the increment for i each time through the loop
- l: the upper limit for i; terminate loop when exceeded

With the present Fortran compilers, i must be a non-subscripted integer variable and j, k, and l must be integer constants greater than zero or non-subscripted integer variable names.

There is one major deficiency in the syntax of the DO statement; namely, that the termination condition must depend on the index of the loop. The presently used Fortran compilers have several shortcomings. The worst has been that the test for termination is made only after the loop has been executed once. The requirement of only positive increments and of only non-subscripted integer variables are also unnecessary restrictions.

Therefore, the following more general form of the DO statement will be used for this development of the one pass compiler:

DO s FOR i = j, k, b

where

s: the statement label for the last step in the loop
i: the index variable for the loop
j: the initial value for i
k: the increment for i
b: any boolean expression; terminate loop when true

The compiler will translate the loop in such a way that the boolean expression is tested before any instructions in the range of the loop are executed. i may now be any variable, including subscripted ones, and j and k are any arithmetic expressions. b is any boolean expression.

It will be evident in the further development that this greater flexibility places almost no extra burden on the translator.

III. Translation of the DO-loop in an ordinary compiler

The first case is that in which the compiler loads the program into the machine during translation. In addition to Bauer and Samuelson's symbols, numbers, and address push-down lists, there will be a DO pushdown list. (It is definitely possible to handle the DO statements without creating a new list simply by putting all the information into the old lists. The separate list is a bit simpler conceptually.) This list will store the target statement label s, and two machine addresses.

The flow chart of Figure 1 will be used to represent the translation of all statements other than DO statements.

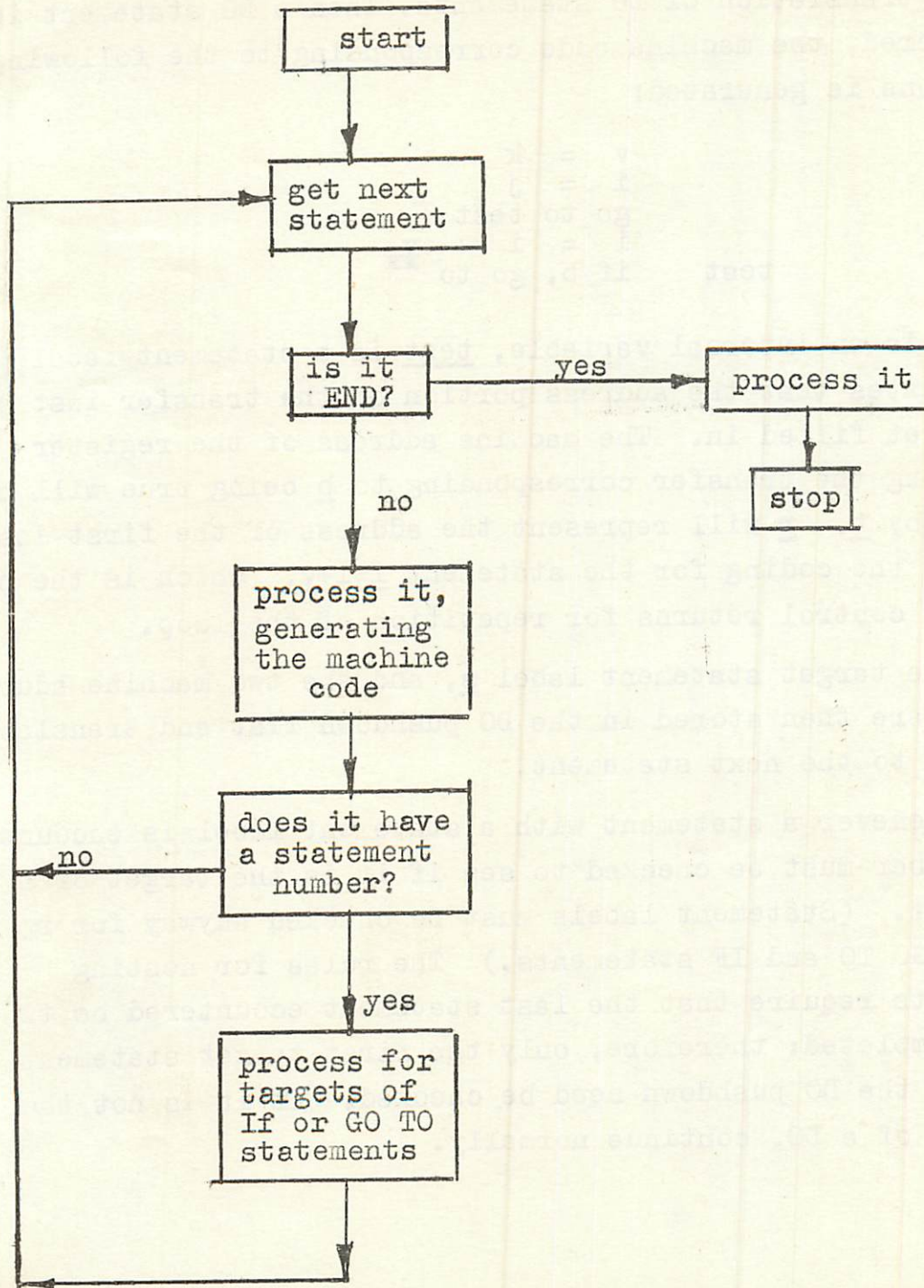


Figure 1.

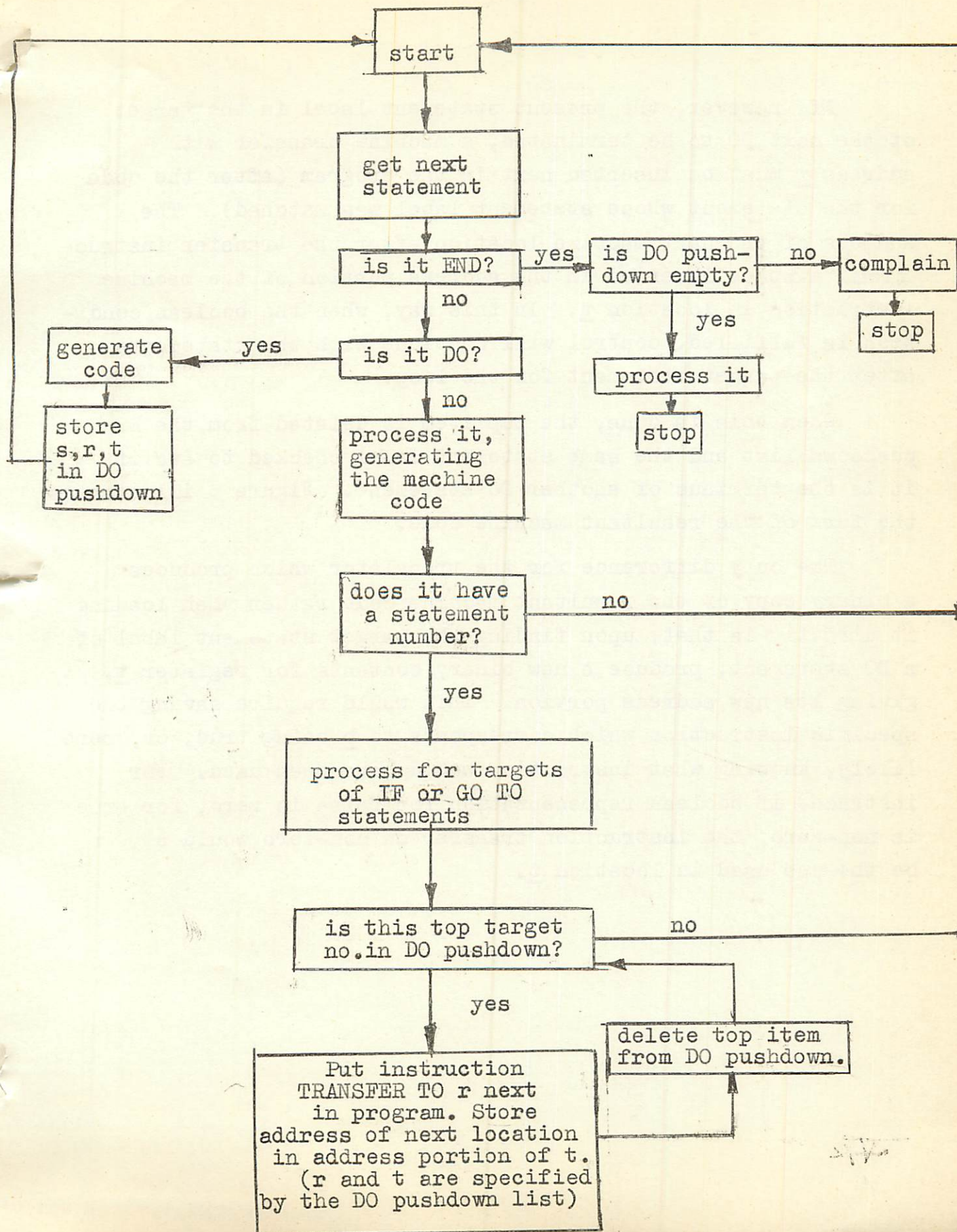
Figure 2 shows the modifications necessary for providing for the translation of DO statements. When a DO statement is encountered, the machine code corresponding to the following operations is generated:

```
      v = k  
      i = j  
      go to test  
      i = i + v  
test  if b, go to xx
```

where v is an internal variable, test is a statement label, and ~~xx~~ indicates that the address portion of the transfer instruction is not yet filled in. The machine address of the register containing the transfer corresponding to b being true will be denoted by t. r will represent the address of the first instruction in the coding for the statement i=i+v, which is the place to which control returns for repetition of the loop.

The target statement label s, and the two machine addresses r and t are then stored in the DO pushdown list and translation proceeds to the next statement.

Whenever a statement with a statement label is encountered, this number must be checked to see if it is the target of a DO statement. (Statement labels must be checked anyway for references by GO TO and IF statements.) The rules for nesting of DO statements require that the last statement encountered be the first completed: therefore, only the first target statement label in the DO pushdown need be checked. If it is not the terminus of a DO, continue normally.



If, however, the present statement label is the target of the next DO to be terminated, a machine transfer with address r must be inserted next in the program (after the code for the statement whose statement label was matched). The address of the next machine location after the transfer instruction must be inserted in the address portion of the machine instruction in location t. In this way, when the boolean condition is fulfilled, control will continue with the statements after the target statement for the loop.

When this is done, the top item is deleted from the DO pushdown list and the same statement label checked to see if it is the terminus of another DO statement. Figure 3 illustrates the form of the resultant machine code.

The only difference for the translator which produces a binary copy of the resultant machine code rather than loading it directly is that, upon finding the target statement label of a DO statement, produce a new binary contents for register t, giving its new address portion. This would require saving the specific instruction which corresponds to b being true, or, more likely, knowing what instruction would have been used. For instance, if boolean representation for false is zero, for true is non-zero, the instruction transfer on non-zero would always be the one used in location t.

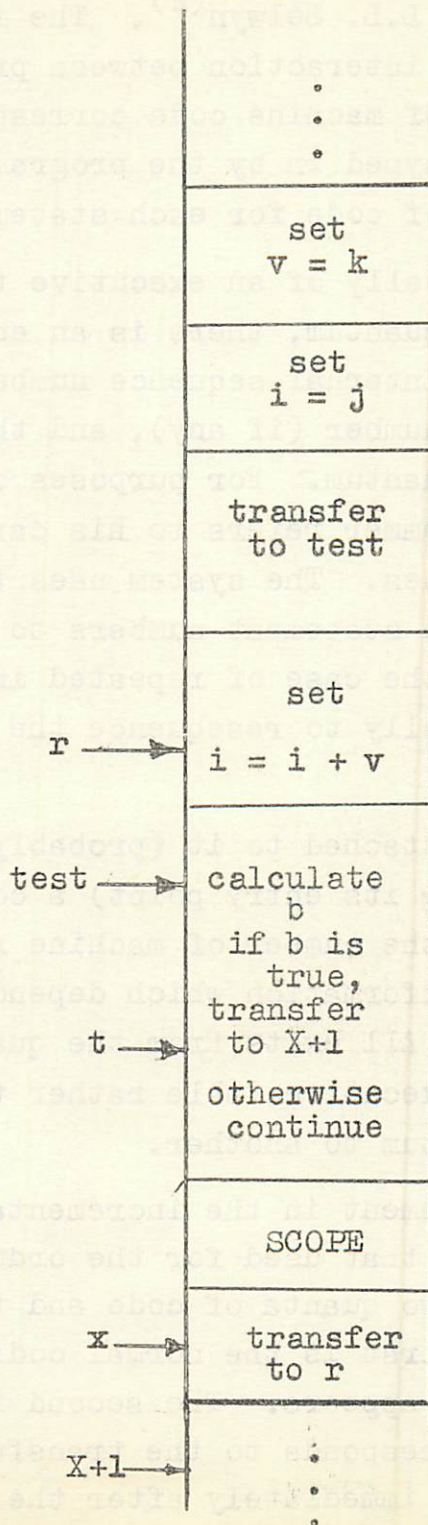


Figure 3.

IV. Incremental compilation of DO statements

The framework for incremental compilation of Fortran programs has been presented by L.L. Selwyn⁽²⁾. The incremental compiler, designed for on-line interaction between programmer and program, generates quanta of machine code corresponding to the source program statements typed in by the programmer. In general, there is one quantum of code for each-statement.

The system consists basically of an executive table and the quanta of code. For each quantum, there is an entry in the executive table containing an internal sequence number, the programmer's source statement number (if any), and the entry address of the corresponding quantum. For purposes of insertion, deletion, or change, the programmer refers to his cards as statement number n_1 plus n_2 lines. The system uses the internal sequence numbers along with the statement numbers to find the exact entry in the table. In the case of repeated insertions, it would be necessary occasionally to resequence the internal numbers.

Each quantum will have attached to it (probably in the registers immediately preceding its entry point) a code number giving the type of statement, the number of machine registers it occupies, plus additional information which depends on the particular type of statement. All exits from the quanta are referenced to entries in the executive table rather than being direct transfers from one quantum to another.

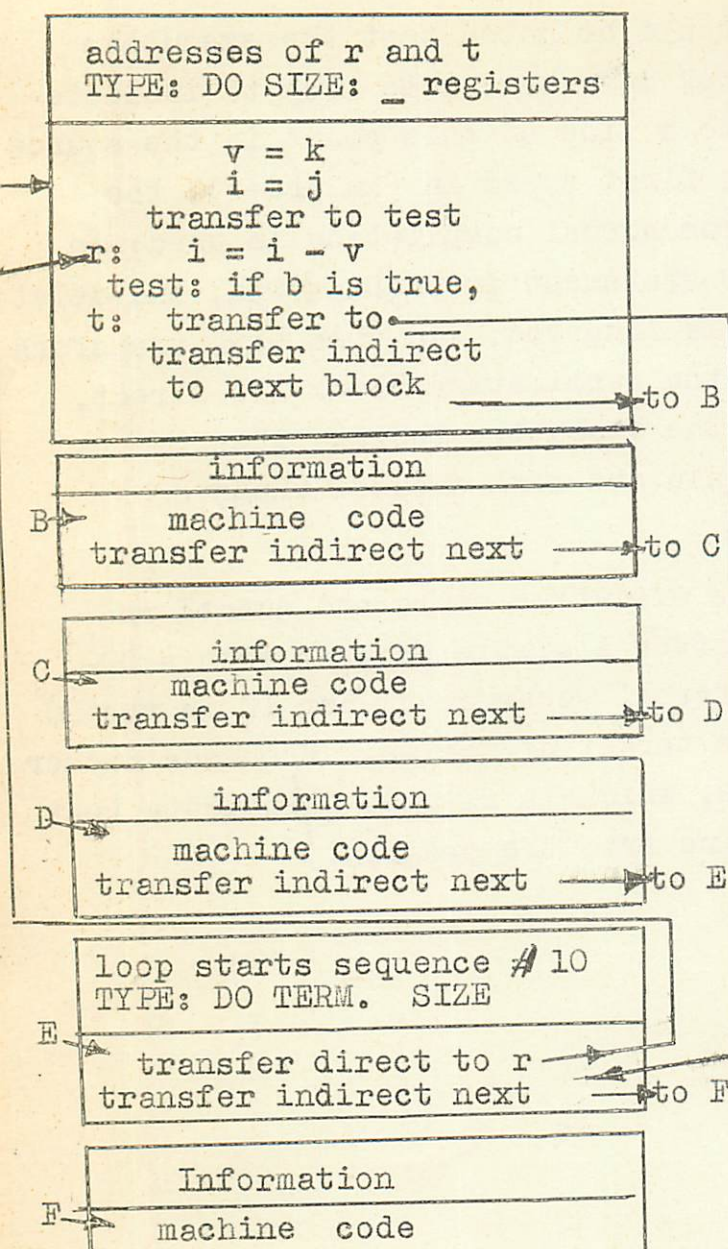
The coding for a DO statement in the incremental compilation system is very similar to that used for the ordinary compilation. However, it requires two quanta of code and two entries in the executive table. The first is the normal coding which appears where the DO statement appears. The second is the DO termination quantum, which corresponds to the transfer to 3 in the normal coding. This comes immediately after the target

statement of the loop. It should be noted that the executive entry for the termination block must have some flag to indicate that it does not correspond to a line at this point in the source program. When the program is first typed in (in order), the standard DO pushdown list from normal compilation is used; an entry being made when the DO statement is encountered, and delete when the termination quantum is inserted. In this way, transfers between the DO statement and the termination block are direct, rather than indirect through the executive table. This causes no problems since the two quanta are both checked whenever a change is made to either.

Figure 4 illustrates the executive table and quanta and the connections between them after a simple program with a DO-loop has been translated. Internal sequence number 10 is the DO statement itself and 40 is the target of the DO. Sequence number 50 is the DO termination block, which is flagged to indicate that it does not correspond to a line from the program.

QUANTA OF CODE

EXECUTIVE TABLE



Internal sequence number	F l a g	Statement number (if any)	Location of quantum of coding
10		W	A
20			B
30		X	C
40		S	D
50	X		E
60			F

...
 Number Statement
 W DO s FOR i = j, k, b
 (statement) (B)
 X (statement) (C)
 S (statement) (D)
 (statement) (F)

Figure 4.

The information section of the DO quantum gives the machine addresses of r and t, the return point and exit transfer locations (the same items that are stored in the DO pushdown list). The termination block information section gives the internal sequence number of the DO statement which it terminates.

Given the program of figure 4, suppose the command

DELETE LINE w

were given. The result is shown in Figure 5. The blocks labelled a and e, corresponding to the quanta for the DO statement and the termination blocks have been replaced with system no-operation blocks, and their executive table entries have been flagged as not corresponding to program statements. The termination block was found by using the address t, and its executive table entry by using s in the information section of the DO quantum.

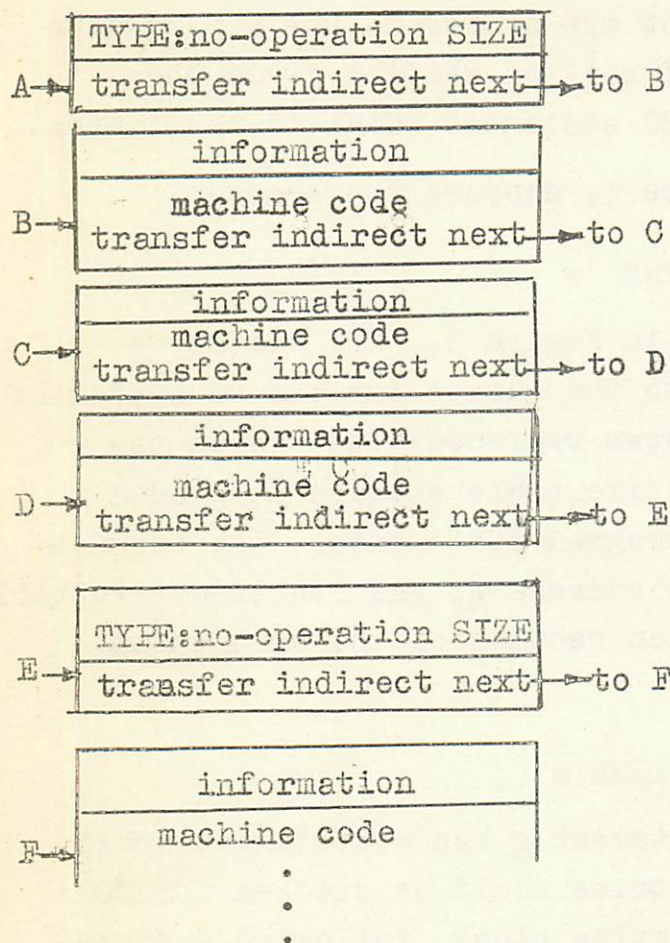
If the command

DELETE LINE s

had been given instead, since statement s has a statement label, the succeeding executive table entries would be checked for DO termination entries. A DO termination block following a statement which is to be deleted would cause the termination block (here labelled e) as well as the block named in the delete command to be replaced with no-operation blocks. From the information section of the termination block, the sequence number of its corresponding DO statement is found. This sequence number leads to the information section of the DO statement quantum from which s, r, and t are obtained and placed on an unsatisfied-DO list. It should be noted that the unsatisfied-DO list is no longer a DO-pushdown list since modifications may be made in any order. Of course, the programmer must provide target statement for all the items on the unsatisfied-DO list before the program is executed. The lack of order in this new list means that there is no longer any check to prevent improper nesting of loops. It would be possible, but most likely unnecessary, to include such a feature.

QUANTA OF CODE

EXECUTIVE TABLE



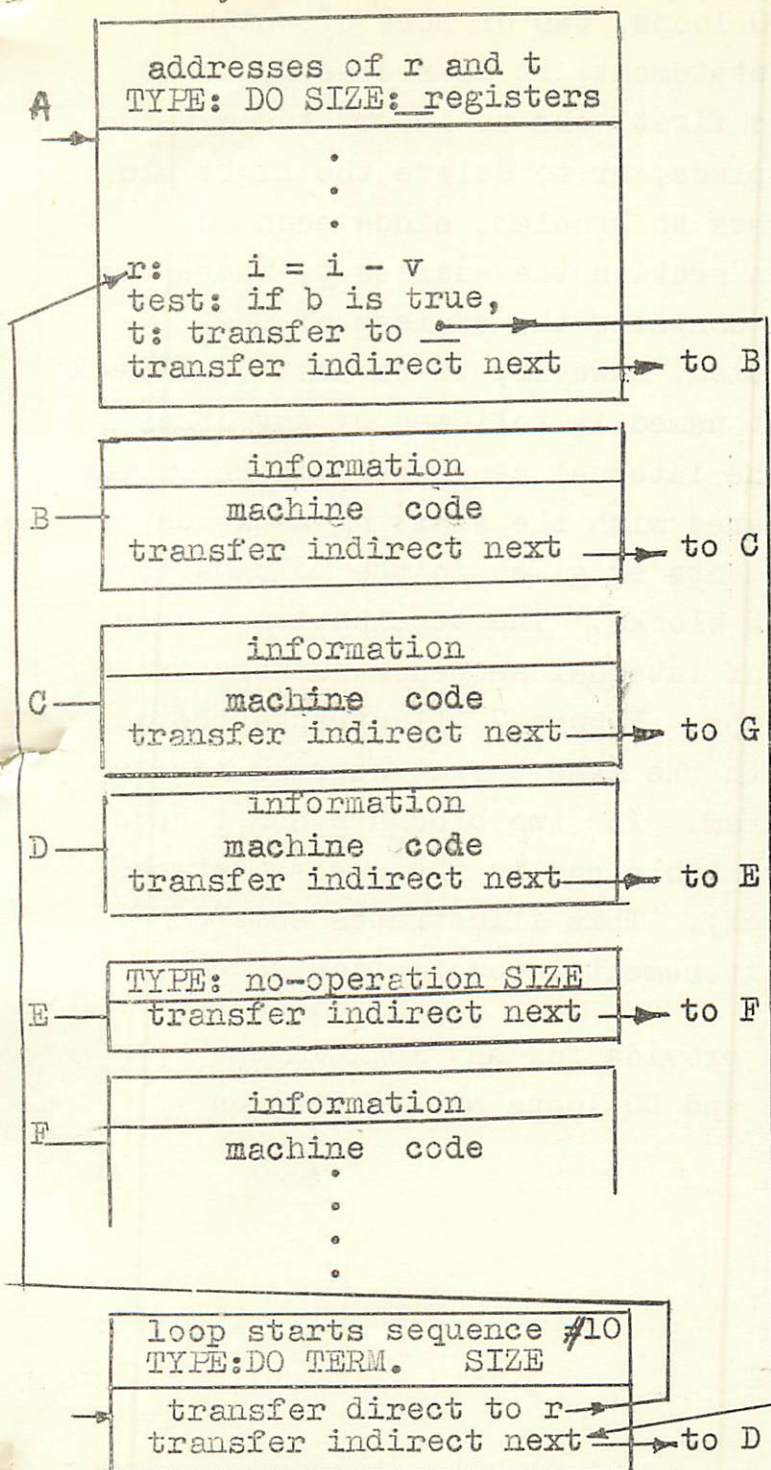
Internal sequence number	Flag	Statement number (if any)	Location of quantum of coding
10	x		A
20			B
30		x	C
40		s	D
50	x		E
60			F

Figure (5)

Returning to the program of figure 4, consider the command to modify the DO statement so that its target is changed from s to x. Figure 6 shows the result. The address t from the information section of the DO quantum was used to find the termination block, which was then replaced with a no-operation. The executive table is then searched for the statement number x. If x is not found, the information x, r, and t is placed on the unsatisfied-DO list. Since x is found, a DO found, a DO termination block is inserted after it in the executive table, and, using r and t appropriate connections are made with its quantum.

To insert a new DO statement, the process is almost the same as that for changing the target address of an old one.

QUANTA OF CODE



EXECUTIVE TABLE

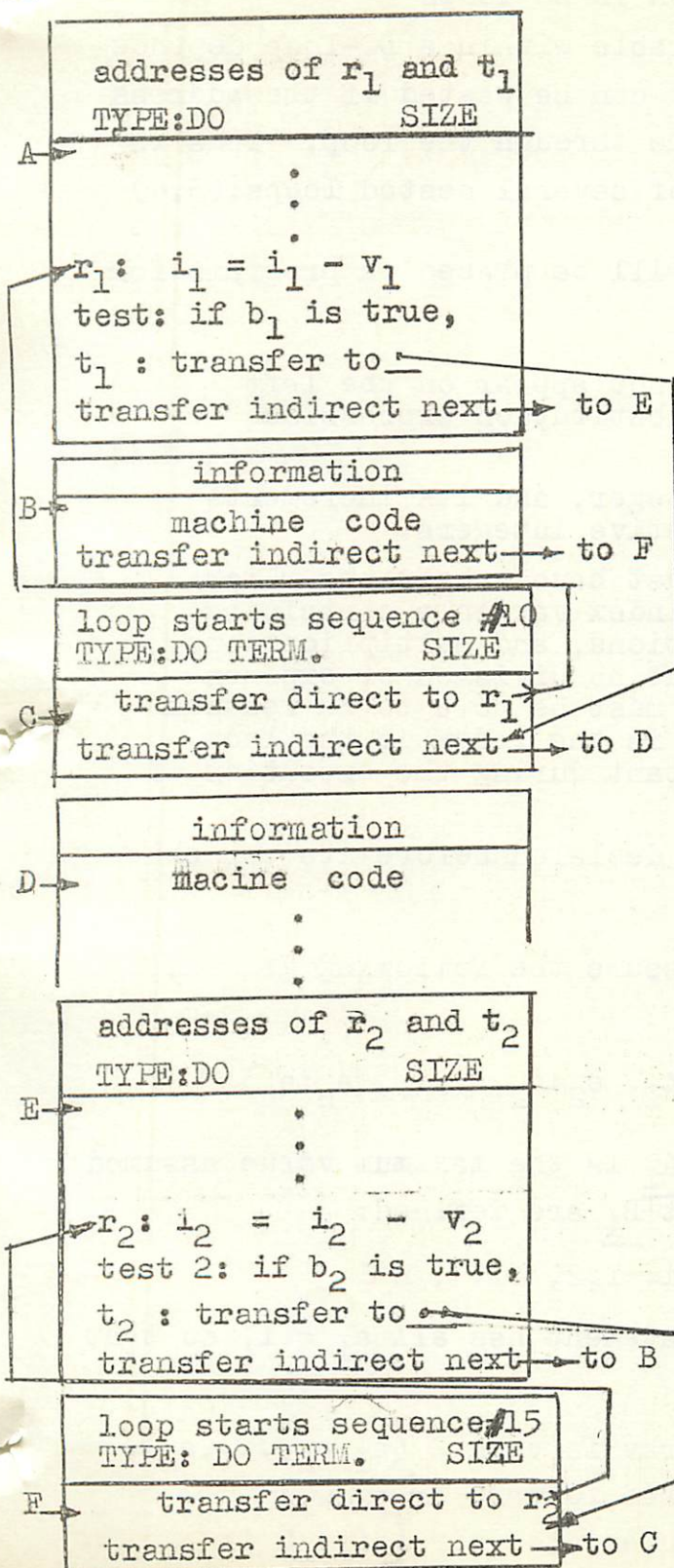
Internal sequence number	Flag	Statement number (if any)	Location of quantum of coding
10		W	A
20			B
30		X	C
40		S	D
50	X		E
60			F
45	X		G

In the case of nested DO loops, two or more of which terminate on the same target statement, it is necessary to be careful as to which terminates first, and to insert a new termination block into the right place, or to delete the right old block. Deletion actually causes no problem, since each DO quantum has in its information section the address t, where the address portion of register t contains the address of the termination block. For insertion, however, there must be a check to see if the target statement named is followed by any other DO termination blocks. If so, the internal sequence numbers of the new DO statement must be compared with the starting sequence numbers of the other DO statements as given in the information sections of the DO termination blocks. The termination blocks must end up in reverse order of internal sequence numbers of the starting points of the loops. Figure 7 illustrates the addition of a DO statement with the same target address as a statement already in the program. The two blocks e and f have been inserted in the executive table and in the quanta between a and b and b and c respectively. This illustrates some of the great flexibility of the incremental compilation scheme.

These facilities should provide for any legitimate manipulation of DO statements and DO loops within Fortran programs.

QUANTA OF CODE

EXECUTIVE TABLE



Internal sequence number	F l a g	Statement number (if any)	Location of quantum of coding
10			A
20		s	B
30	*		C
40			D
15			E
25	*		F

ORIGINAL FORTRAN PROGRAM

DO s FOR i₁ = j₁,k₁,b₁

s (statement) (B)
(statement) (D)

MODIFIED FORTRAN PROGRAM

DO s FOR i₁ = j₁,k₁,b₁
DO s FOR i₂ = j₂,k₂,b₂

s (statement) (B)
(statement) (D)

Figure 7.

V. Recursive address calculation in DO-loops

When the subscript of a variable within a DO-loop depends on the index of the loop, much time can be wasted if the address is calculated from scratch each time through the loop. This is especially for the innermost loop of several nested loops.(3,4)

The following restrictions will be placed on programs for this discussion:

1. The index variable must not appear on the left side of an arithmetic substitution expression within a DO-loop.
2. The index must be an integer, and its increments must be positive or negative integers.
3. Subscripted variables must have subscripts which are expressions in the index variable containing only additions, subtractions, and multiplications, (parentheses allowed) but no division or exponentiation. Each subscript must be able to be reduced to $d_k - i \times c_k$, where i is the index of the loop, and c_k and d_k are constant during the execution of the loop.
4. An array must have been declared before its variable is used.

Dimension statements will assume the following general form:

DIMENSION Z ($a_1:A_1, a_2:A_2, \dots, a_n:A_n$)

where a_i is the minimum value and A_i is the maximum value assumed by the i th subscript. The constant B_i are defined:

$$B_i = A_i - a_i + 1 \quad \text{for } i = 1, 2, \dots, n-1$$

(The standard Fortran DIMENSION statement has all $a_i = 1$, so that $B_i = A_i$.)

The first element in the array is then $Z(a_1, a_2, \dots, a_n)$, abbreviated $Z(a)$. For arrays stored forward in memory, the address of $Z(y_1, y_2, \dots, y_n)$, abbreviated $\text{adr}[Z(y)]$ is

$$\text{adr}[Z(y)] = \text{adr}[Z(a)] - f(a_1, a_2, \dots, a_n) + f(y_1, y_2, \dots, y_n)$$

where

$$f(b_1, b_2, \dots, b_n) = (\dots((b_n x_{B_{n-1}} + b_{n-1} + \dots) x_{B_{n-2}} + b_{n-2}) x \dots) B_1 + b_1$$

which is abbreviated $f(b)$. Note here that

$$f(x+y) = f(x) + f(y)$$

$$f(ax) = af(x), \quad a \text{ constant}$$

Then

$$\begin{aligned} \text{adr}[Z(y)] &= \text{adr}[Z(a)] - f(a) + f(y) \\ &= \text{adr}[Z(0)] + f(y) \end{aligned}$$

The constant, $\text{adr}[Z(0)]$, as well as the B_i , need be calculated only once during compilation.

Each subscript position of Z within a loop has been restricted to the form

$$y_k = d_k + i x c_k$$

where i is the index variable of the loop and c_k and d_k are constant during any particular execution of the loop. Then

$$\text{adr}[Z(y)] = \text{adr}[Z(0)] + f(d) + i x f(c)$$

If i has the initial value r and is incremented each time through by s , then the initial value of the address will be

$$\text{adr}[Z(y)] = \text{adr}[Z(0)] + f(d) + r x f(c)$$

and the address is incremented each time through the loop by

$$s x f(c)$$

Now, since $f(c)$, $f(d)$, and $s x f(c)$ are constant during each execution of the DO-loop, they can be calculated during execution before entering the loop. Then, in the execution of the loop, only one addition is required each time around to modify the address of the subscripted variable, rather than the $n-1$ multiplications and n additions required to compute the address completely each time. When the same subscripted variable, or variables with the same subscript, is used in several places within a loop, the saving is even greater.

An example will best illustrate the procedure.

DIMENSION Z (1:5, 3:8)

$$B_1 = 5 - 1 + 1 = 5$$

The first element of array Z, namely Z (1,3) is assigned to the location named q.

$$\text{adr } Z(0) = q - f(1,3) = q - (3 \times 5 + 1) = q - 16$$

$$\text{adr } Z(y_1, y_2) = q - 16 + y_2 \times 5 + y_1$$

Then, for example

$$\text{adr } Z(3,3) = q - 16 + 3 \times 5 + 3 = q + 2$$

$$\text{adr } Z(4,5) = q - 16 + 5 \times 5 + 4 = q + 13$$

$$\text{adr } Z(5,7) = q - 16 + 7 \times 5 + 5 = q + 24$$

Suppose within the DO-loop

DO s FOR i = j, k, b

there is an expression involving a form expressible as

$$Z(0 + ix_1, -3 + ix_2)$$

for the particular execution of the loop in which

$$j = 3 \text{ and } k = 1$$

The following calculations are made before entering the loop:

$$d_1 = 0 \quad d_2 = -3 \quad c_1 = 1 \quad c_2 = 2$$

$$f(d) = -3 \times 5 + 0 = -15 \quad f(c) = 2 \times 5 + 1 = 11$$

$$s = 1 \quad \text{ssf}(c) = 11$$

The initial address is

$$\begin{aligned} \text{adr } Z(3,3) &= q - 16 - 15 + 3 \times 11 \\ &= q + 2 \end{aligned}$$

The increment was found to be 11, so the next two times we should get

$$\text{adr } Z(4,5) = q + 2 + 11 = q + 13$$

$$\text{adr } Z(5,7) = q + 13 + 11 = q + 24$$

which check with the values for $Z(3,3)$, $Z(4,5)$ and $Z(5,7)$ calculated previously.

For nested DO-loops, c, d, r, s, may depend on outer loop indexes. The calculation of values of $f(c)$ and $f(d)$ must therefore be made immediately before entering the innermost loop in which the subscripted variable is used.

Now, to fit this scheme into one pass translation poses significant problems. Before recursive address calculation can be set up, it is necessary to know first whether or not a loop contains any subscripted variables depending on its index, and whether or not they satisfy the conditions given at the beginning of this section. This at least requires look-ahead whenever a DO statement is encountered (or the programmer would have to give the information explicitly). In any case it does not fit in with the last-in first-out scheme of translation scheme. The most practical scheme for recursive address calculation would employ a first pass to determine the structural layout of loops and to collect information on the form of subscripted variables within them. The main pass could then effectively translate the program.

VI. Conclusions

A one pass Fortran scheme can readily include processing of DO-loops, with extended features available. Though an extra pushdown list is used during compilation, it is not left around during execution. These DO-loops fit well into the incremental compiler system, allowing easily for insertions, deletions, changes of target address, etc

If recursive address calculation is used for subscripted variables within DO-loops, either very stringent restrictions are placed on the language and extra requirements on the programmer, or a look-ahead is required in the one pass system, destroying the sequential nature of translation. The use of two passes could solve

BIBLIOGRAPHY

1. K. Samuelson and F. L. Bauer: Sequential Formula Translation. Communications of the ACM , Vol. 3, No. 2. (February, 1960) pp. 76-83.
2. L.L. Selwyn : The Incremental Fortran Compiler. Internal memo, Project MAC, MIT.
3. Bauer and Samuelson, op. cit., Section 7, p. 80-82.
4. U. Hill, H. Langmaack, H. R. Schwarz, G. Seegmuller; Efficient Handling of Subscripted Variables in Algol 60 Compilers. Symbolic Languages in Data Processing. (Proceedings of the Symposium). Gordon and Breach, New York, 1962. pp. 331-340.

Notes on FORMAC

OUTLINE

I. BASIC CONCEPTS

1. General Description
2. Difference Between FORMAC and FORTRAN
3. Expression Formation
4. Types of Simplification

II. LANGUAGE DESCRIPTION

1. List of Executable Statements
2. List of Declarative Statements
3. Metalanguage for Syntax
4. Syntax of Executable Statements
5. Syntax of Declarative Statements

III. IMPLEMENTATION

1. Compilation Process
2. Summary of Implementation Characteristics
 - 2.1 Purpose and Operation of Preprocessor
 - 2.2. Object Time Subroutines
 - 2.3 Symbol Table
 - 2.4 Expression Table
 - 2.5 Storage Allocation at Object Time
3. Object Time Execution

IV. EXAMPLES

1. Mathematical Induction
2. Series Generation and Error Analysis
3. Matrix Multiplication (Computer Printouts)

I. BASIC CONCEPTS

1. General Description

FORMAC (FORMac MANipulation Compiler) is an experimental system designed to provide a practical tool for doing non-numerical mathematics (i.e. symbol manipulation). FORMAC is an extension of FORTRAN IV, and as such the user has numeric and loop capabilities which can be intermingled with the capability of doing formal mathematical manipulation.

It should be emphasized that FORMAC is only an experimental program and at present there are no plans to release it to customers. It has been developed by IBM's Boston Advanced programming Department.

2. Difference Between FORMAC ~~x~~ and FORTRAN

Assume the equation $Y = X^2 + 2.5 \frac{Z}{X}$

A FORTRAN program can evaluate Y for numerical values of X and Z. For example, the FORTRAN program

```
X = 5.  
Z = 4.  
Y = X**2 + Z**2.5/X
```

generates code to calculate the value of Y, which is 27.

A FORMAC program can create a symbolic value (i.e. an expression for) numeric or symbolic values of X and Z. For example, the FORMAC program

```
LET X = A + B  
Z = 4.  
LET Y = X**2 + Z**2.5/X
```

generates an internal representation of the expression $(A+B)^2 + \frac{10}{A+B}$ which is named Y.

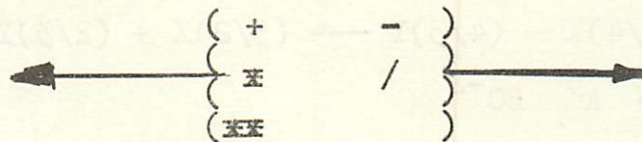
*Throughout the rest of these notes, FORMAC will be used to refer only to those elements which have been added to FORTRAN; strictly speaking, FORTRAN is a subset of FORMAC.

3. Expression Formation

Expressions in FORMAC are written in essentially the same way as in FORTRAN, except that some additional operations are added. FORMAC expressions can contain both FORMAC (i.e. symbolic) and FORTRAN (i.e. numeric) variables.

FORTRAN operators

(for numbers)



ALOG
SIN
COS
EXP
ATAN
TANH

FORMAC operators

(for expressions)

FMCLG
FMCSIN
FMCCOS
FMCEXP
FMCATN
FMCHTN

FMCDIF (differentiation)
FMCOMB (combinatorial)
FMCFAC (factorial)
FMCDFC (double factorial)

As an illustration of the differentiation operator, the FORMAC statement

LET Y = FMCDIF (~~X**2~~ + 4~~X**3~~, X, 1)

causes the name Y to be assigned to the expression $2X + 12X^2$.

4. Types of Simplification

(1) Automatic - Performed after each executable command and sometimes during the execution of the commands. The user can control the type of evaluation being done, (see AUTSIM) but has no control over when the automatic simplification is performed.

(2) Removal of parentheses - EXPAND.

(3) Factoring with respect to powers of single variable - COEFF.

Examples of Automatic Simplification

$$A + 0 \longrightarrow A$$

$$A^1 \longrightarrow A$$

$$A^0 \longrightarrow 1$$

$$A^1 \longrightarrow A$$

$$1^A \longrightarrow 1$$

$$3A + 4C \longrightarrow A \longrightarrow 2A + 4C$$

$$3.2A + (4/3)B - (1/3)B + 6.3A - (1/3)B \longrightarrow 9.5A + .666666667B$$

$$(1/4)X - (2/3)Y + (5/4)X + (4/3)Y \longrightarrow (3/2)X + (2/3)Y$$

$$C^2 A^2 B A^3 C^{-4} \longrightarrow A^5 B C^{-2}$$

II. LANGUNCE DESCRIPTION

1. List of Executable Statements

In addition to all FORTRAN statements, there are 15 executable FORMAC statements which are listed below in 3 categories. Later pages show the exact syntax and an example of each command.

Statements Yielding FORMAC Variables

- LET - Construct specified expressions.
- SUBST - Replace variables with expressions.
- EXPAND - Remove parentheses.
- COEFF - Obtain coefficient of variable or its powers.
- PART - Separate expressions into terms, factors, exponents.

Statements Yielding FORTRAN Variables

- EVAL - Evaluate expression.
- MATCH - Compare two expressions for equivalence or identity.
- FIND - Determine dependence relations.
- CENSUS - Count words, terms, or factors.

Miscellaneous

BCDCON - Convert to BCD form from internal form.
ALGCON - Convert to internal form from BCD form.
AUTSIM - Control arithmetic done during automatic simplification.
ERASE - Eliminate expressions no longer needed.
ORDER - Specify sequence of variables in output expressions.
FMCDMP - Provide dump of all or some symbolic expressions during execution.

2. List of Declarative Statements

In addition to the FORTRAN declarations, there are 4 declarative FORMAC statements listed below. Later pages show the exact syntax and an example of each statement.

ATOMIC - Declare basic variables.
DEPEND - Declare implicit dependence relations.
PARAM - Declare parametric pairs for SUBST and EVAL.
SYMARG - Declare subroutine arguments as FORMAC variables; flag program beginning.

3. Metalanguage for Syntax

The formal syntax of each statement is shown in a metalanguage which is defined as follows:

(1) Metalanguage Used in Statement Definition

UPPER CASE	Constants, or literals which stand for themselves.
	The vertical stroke denotes "or".
{a}	The curly braces denote grouping.
{b}	
....	Repetition of preceding pattern.
[a]	The square brackets mean 'optional'.
max n {var}	The pattern "var" can occur no more than n times.

(2) Metavariables Used in Statement Definition

The meaning of the metavariables can be derived from the abbreviations used. That is, "cons" is short for constant, "fit" is short for floating "fxd" is short for fixed, "fmc" is short for FORMAC, "ftn" is short for FORTRAN, "exp" is short for expression and "var" is short for variable.

In general all variables can be subscripted and may be of any mode.

i.e. ftn-var stands for FORTRAN variable which may or may not be subscripted.

4. Syntax of Executable Statements

LET

LET let-var = fmc-exp

Purpose - Basic assignment command. It generates, at object time, a FORMAC expression and assigns the FORMAC let variable as the name of that expression.

Example

ATOMIC Z

M = 4.

LET C = M + FMCDIF (Z**2, Z, 1) - Z
results in C → 4. + Z

SUBST

LET let-var = SUBST fmc-exp, $\left\{ \begin{array}{l} \text{param-label max 9} \\ \text{param-list} \end{array} \right\}, \text{param-label} \}$

param-list = (seek-var, fmc-exp₁) $\left\{ , (seek-var, fmc-exp_1) \right\} \dots$

- 1) fmc-exp is the expression into which the substitutions will be made.
- 2) param-list specifies the variables to be replaced and the expressions which are to replace them.
- 3) param-label (s) reference param-lists which are specified elsewhere in the program.

Purpose - Replace variables in an expression by constants, other variables or other expressions according to the specifications of a referenced or accompanying parameter list.

Example

ATOMIC X, Y, Z, A, B, C

LET R = SUBST (X + X ~~XX~~ 2 + FMCSIN(Y)),
(X, A+B, (Y, A-B))

EXPAND

results in $A = A+B + (A+B) \times 2 + FMCSIN (A-B)$

LET let-var = EXPAND fmc-exp

[, CODEM]

Purpose - To remove parentheses in the FORMAC expression by applying the distributive law and/or multinomial theorem.

Example

ATOMIC A,B,C,D

LET R = EXPAND (A+C)~~XX~~2 + A~~X~~(C-D)

results in $R \rightarrow A \times 2 + 3 \times A \times C +$
 $C \times 2 - A \times D$

COEFF

LET let-var = COEFF fmc-exp, seek-var, fit-var-1,
ftn-flt-var-2

- 1) fmc-exp is the expression containing the variable whose coefficient is desired.
- 2) seek-var is the variable whose coefficient is desired.
- 3) ftn-flt-var-1 is a FORTRAN variable which is assigned the next highest power of the seek-var which exists in the expression fmc-exp.
- 4) ftn-flt-var-2 is a FORTRAN variable which is assigned the next lowest power of the seek-var which exists in the expression, fmc-exp.

Purpose - Obtain as a new expression the coefficient of a variable (which may be raised to a power) within a given expression.

Example

ATOMIC X, Y, Z

LET R = COEFF (X + A~~X~~~~X~~~~X~~2 + ~~X~~~~X~~~~X~~2 + Y~~X~~~~X~~~~X~~3),

~~X~~~~X~~~~X~~2, B, C

results in $R \rightarrow 1. + A$

B = 3.

C = 1.

PART

LET let-var₁ = PART let-var₂, ftn-~~fxd~~-var

1) let-var₂ is the name of the expression to be partitioned.

2) ftn-~~fxd~~-var is the FORTRAN variable which receives a value which describes the partitioned expression.

Purpose - Separate out the first well formed sub-expression from the expression named.

Example

ATOMIC X, Y, A

LET R = X~~X~~Y+A

LET S = PART R,I

results in $S \rightarrow A$

$R \rightarrow X~~X~~Y$

I = 4 which indicates a sum of terms was partitioned.

now

LET Q = PART R,I

results in $Q \rightarrow X$

$R \rightarrow Y$

I = 5 which indicates a product of factor was partitioned.

EVAL

LET ftn-num-var = EVAL fmc-exp , $\left\{ \begin{array}{l} \text{param label} \\ \text{param list} \end{array} \right\}$

Purpose - Evaluate a FORMAC expression and produce a FORTRAN numeric value.

ATOMIC X,Y,Z

Example

LET AM = EVAL (X+X)~~≠~~Z, (X,3),(Y,FMCFAC(2)),(Z,.5)
results in AM = 2.5

MATCH

LET ftn-log-var = MATCH $\left\{ \begin{array}{l} \text{ID} \\ \text{EQ} \end{array} \right\}$, ftn-flt-num $\left\{ \begin{array}{l} \text{fmc-exp-1,} \\ \text{fmc-exp-2} \end{array} \right\}$

- 1) fmc-exp-1 and fmc-exp-2 are the expressions to be compared.
- 2) ftn-flt-num is the tolerance value if the equivalence option is specified.
- 3) ftn-log-var is the FORTRAN logical variable which receives the resultant value of true or false depending on the success or failure of the match.

Purpose - Compare two FORMAC expressions for either identity or mathematical equivalence.

Example

ATOMIC A,B
LOGICAL Q
LET X = (A+B)~~≠~~2
LET Y = A~~≠~~2 + 2~~≠~~A*B + B~~≠~~2
LET Q = MATCH ID,X,Y
results in Q = .FALSE.
LET Q = MATCH EQ, .001, X,Y
results in Q = .TRUE.

FIND

LET ftn-log-var = FIND fmc-exp, $\left\{ \begin{array}{l} \text{APP} \\ \text{DEP} \end{array} \right\}$, $\left\{ \begin{array}{l} \text{ALL} \\ \text{ONE} \end{array} \right\}$,
(seek-var $\left\{ \begin{array}{l} \text{seek-var} \end{array} \right\}$...)

- 1) fmc-exp is the expression to be examined.
- 2) seek-var are the variables being investigated.
- 3) $\left\{ \begin{array}{l} \text{APP} \\ \text{SEP} \end{array} \right\}$ specifies whether the seek-var must appear explicitly in the expression or whether a dependence relationship is sufficient.
- 4) $\left\{ \begin{array}{l} \text{ALL} \\ \text{ONE} \end{array} \right\}$ specifies whether ALL or ONE of the seek-var must meet the APP or DEP condition.

Purpose - To determine if one or more specified variables exist explicitly in an expression or if they are implicitly dependent (see DEPEND declaration) on any variables which do exist in the expression.

Example

```
ATOMIC B,C,X,Y,Z,D
LET A = B + C + D - F
LET Q = FIND A, APP, ALL(B,C,D,F,G)
      results in Q = .FALSE.
whereas
LET Q = FIND A, APP, ONE, (B,C,D,F,G)
      results in Q = .TRUE.
```

CENSUS

LET ftn-fxd-var = CENSUS let-var,

$\left\{ \begin{array}{l} \text{WORD} \\ \text{TERM} \\ \text{FACTOR} \end{array} \right\}$

- 1) let-var is the name of the expression in which the number of words, terms, or factors is to be counted.
- 2) ftn-fxd-var is the FORTRAN variable which receives the result of the count.

Purpose - Count the number of terms, factors or computer words which appear in an expression.

Example

```
ATOMIC A,B,C,D
LET Z = A + B + C + D
LET M = CENSUS Z, TERM
      results in M = 4
LET M = CENSUS Z, FACTOR
      results in M = 1
LET M = CENSUS Z, WORD
      results in M = 6
```

BCDCON

LET ftn-num-var = BCDCON fmc-exp, ftn-fxd-var, fxd-num

- 1) `fmc-exp` is the FORMAC expression which is to be converted.
- 2) `ftn-fxd-var` is the FORTRAN array which acts as a buffer and receives the resultant BCD string of characters.
- 3) `fxd-num` is a FORTRAN value which specifies how many words of the buffer are to be used by the `BCDCON` command.
- 4) `ftn-num-var` is a FORTRAN variable which receives a value indicating whether or not the translation for an expression is complete.

Purpose - When used in conjunction with the FORTRAN output statements this command permits expressions to be put out during execution of a program.

Example `ATOMIC AB,BC,X`
 `DIMENSION LIST (4)`
 `LET R = AB + BC + FMCSIN(X) FMCCOS(X) + 4.3`
 `Q = 0.`
 `LET Q = BCDCON R, LIST,4`

would result in LIST(1)

(1)	binary 18
(2)	<code>ABBC+</code>
(3)	<code>FMCSIN</code>
(4)	<code>(X) FMCCOS</code>

indicates
18 significant
characters in
the buffer.

and $Q \neq 0$. indicates there is more translation to perform. Upon re-entry to `BCDCON` with $Q \neq 0$ the translation would continue.

ALGCON

`LET let-var = ALGCON ftn-num-var, ftn-fxd-var`

- 1) `ftn-fxd-var` is the name of the FORTRAN array which contains the BCD representation of the expression to be converted.
- 2) `ftn-num-var` is the FORTRAN variable which indicates whether translation is to begin at the top of the buffer or from where the last translation ended.

Purpose - When used in conjunction with FORTRAN input statements this command permits expressions to be read in at object time. With a FORTRAN read statement under control of an "A" (alpha) conversion in a FORMAT statement, the user may read into an array, an expression which is in BCD form.

Example

If ARRAY (1)

(2)

(3)

(4)

A+B X I
FMCDIF
(X,X,1
)\$ØØØØ

ATOMIC A,B

DIMENSION ARRAY (4)

j = 0

LET I = EVAL (A+B), (A,3), (B,1)

LET Y = ALGCON ARRAY (1), J

results in Y \Rightarrow A+B~~X~~4.-1.

AUTSIM

AUTSIM

QINT
QNUM
QNINT
ON

- 1) QINT causes evaluation of FMCFAC, FMCDIF and FMCOMB operators.
- 2) QNUM causes evaluations of FMCSIN, FMCCOS, FMCLG, FMCATN, FMCHTN, FMCEXP and ~~XX~~ operators.
- 3) QNINT combines QINT and QNUM options
- 4) ON specifies that none of the above operators are to be evaluated.

Purpose - Provides the user with control over which FORMAC operators will be evaluated during Automatic Simplification.

Example

ATOMIC X,Y,Z

AUTSIM QINT

I = 3

LET A = 4 + ~~I~~I + FMCFAC(I)~~FMCSIN(I)~~

results in A = 4 + 3~~3~~ + 6 FMCSIN(3)

ERASE

ERASE let-var {, let-var } ...

Purpose - Delete the expression (s) named and make available the storage which the expression (s) used.

Example

ATOMIC X,Y

LET R = X + Y

ERASE R

causes the expression R to be destroyed and more space to be available to the FORMAC dynamic storage allocation system.

ORDER

LET let-var = ORDER fmc-exp, $\left\{ \begin{array}{c} \text{INC} \\ \text{DEC} \end{array} \right\}$, $\left\{ \begin{array}{c} \text{FUL} \\ \text{PRT} \end{array} \right\}$

$\left[\left\{ \begin{array}{l} ,(\text{seek-var}_i \{, \text{seek-var}_i\} \dots), (\text{seek-var}_j, \text{seek-var}_j \dots) \\ ,(\text{seek-var}_i \{, \text{seek-var}_i\} \dots) \\ ,(\text{seek-var}_j \{, \text{seek-var}_j\} \dots) \end{array} \right\} \right]$

- 1) fmc-exp is the expression to be ordered.
- 2) The INC, DEC option specifies that powers of a variable are to be put in increasing or decreasing order.
- 3) The FUL or PRT option permits full or partial ordering.
- 4) The lists of seek-var(s) specify the order in which variables are to appear.

Purpose - To specify the sequence in which factors in products, and terms in sums, are to appear in expressions. Normally to be used just prior to a BCDCON or FMCDMP statement.

Example

ATOMIC X,Y,Z

LET R = ORDER $X^3 + YX^2 + X^2Y^2$, INC, FUL, (X)

results in $Z \rightarrow 3X + 2X^2Y + X^3Y$

FMCDMP

Syntax omitted because the options are too numerous to list here.

Purpose - To assist the user in debugging a FORMAC program by providing snapshots of symbolic expressions and dumps during execution of a program.

Example

FMCDMP NOW, CURRENT, ALL, INFIX

means print all variables from the routine currently being executed in regular (i.e.infix) form.

5. Syntax of Declarative Statement

ATOMIC

ATOMIC $\left\{ \begin{array}{l} \text{name} \\ \text{name} \end{array} \right. (\text{dim-size}) \left\{ \begin{array}{l} \text{name} \\ \text{name} \end{array} \right. (\text{dim-size}) \left. \right\}$

Purpose - To declare those variable names listed as "FORMAC" atomic variables.

Example

ATOMIC X,Y,Z, (50)

DEPEND

DEPEND $(a_{11} \{, a_{12} \} \dots / b_{11} \{, b_{12} \} \dots) \{, (a_{21} \{, a_{22} \} \dots / b_{21} \{, b_{22} \} \dots) \} \dots$

- 1) Each a and b are atomic variables.
- 2) Each a_{ij} is dependent on all b_{ir} .

Purpose - To declare an implicit dependence relationship between atomic variables. It permits the inclusion of derivations where functional relationships between two variables are not explicitly given.

Example

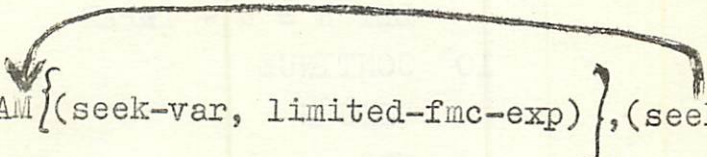
```

ATOMIC X,Y,Z,A,B,C
DEPEND (X/A,B,C),(Y,Z/B)
LET R = FMCDIF(X + A**2,A,1)
      results in R = FMCDIF(X,A,1)+2.*A

```

PARAM

param-label ∇ PARAM { (seek-var, limited-fmc-exp) }, (seek-var limited-fmc-exp) ...



- 1) seek-var are the FORMAC variables and operators which are to be replaced.
- 2) limited-fmc-exp are the FORMAC expressions which are to replace the seek-var.

Purpose - To be used with the FORMAC commands SUBST and EVAL to simplify the listing of pairs of parameters.

Example

```

ATOMIC X,Y,Z,A
LBL PARAM (X,3),(Y,FMCFAC(4)+A),(Z,A-FMCSIN(A)),
      (A,5.)
LET R = SUBST (X+Y) , LBL
      results in R = 3. + FMCFAC(4) + A
LET BR = EVAL R, LBL
      results in BR = 32.

```


SYMARG

SYMARG [name { , name } ...]

Purpose - A necessary flag for the implementation and also declares which variables in FUNCTION and SUBROUTINE arguments lists are FORMAC variables.

Example

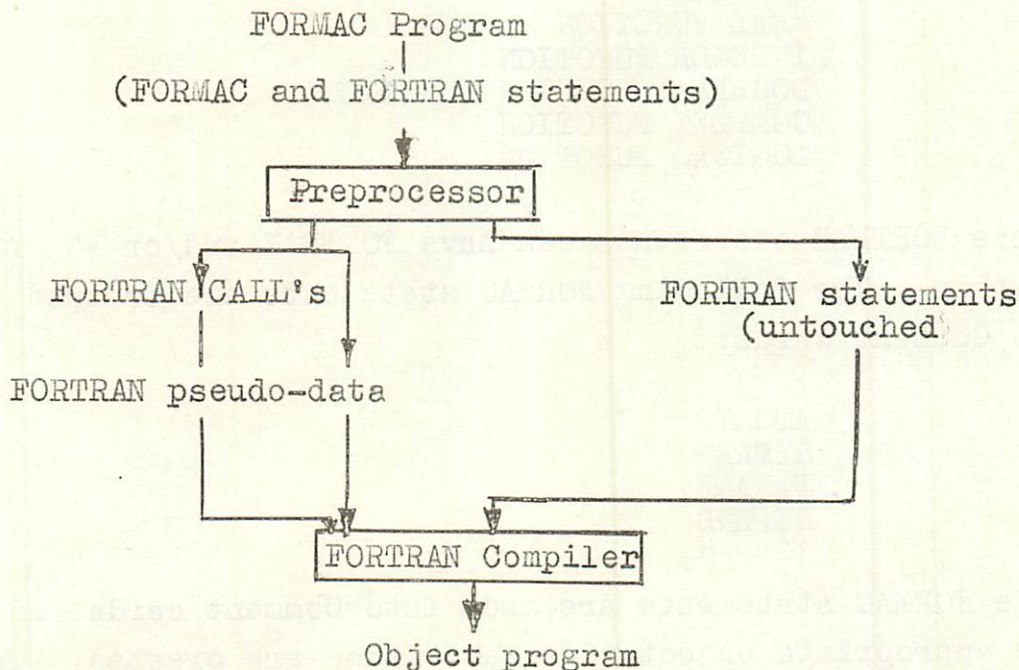
```
SUBROUTINE X,Y,Z (A, PAR, I)
SYMARG PAR
RETR = 0.
DO 10 M = I,15
LET R = R + I*PAR
10 CONTINUE
RETURN
END
```

PAR is a FORMAC dummy variable and hence appear in the SYMARG statement.

III. IMPLEMENTATION

1. Compilation Process

The compilation process for FORMAC had been based on the use of a preprocessor which translates the FORMAC program into a correct FORTRAN program, and then automatically compiles the FORTRAN program. The entire process operates under IBSYS, and uses FORTRAN IV. The following diagram shows this schematically.



2. Summary of Implementation Characteristics

The most significant characteristics of the implementation shown below. They are best understood and illustrated by the use of the computer printouts of the Matrix Multiplication Example shown later, and by Section III.3.

2.1. Purpose and Operation of Preprocessor

Build Symbol Table

Build Expression Table

Transform FORMAC statements to FORTRAN CALL's

Generate additional statements as input to FORTRAN compiler

The preprocessor scans the entire FORMAC program. It looks at each FORTRAN statement only enough to determine whether any information is needed from it. The following FORTRAN statements are scanned and information extracted from them, but they are not modified in any way:

INTEGER
REAL
DIMENSION
SUBROUTINE
FUNCTION
REAL FUNCTION
INTEGER FUNCTION
DOUBLE PRECISION FUNCTION
COMPLEX FUNCTION
LOGICAL FUNCTION

Note that these FORTRAN statements can have FORTRAN and/or FORMAC variables in them. The following FORMAC statements are scanned and made into Comment cards:

ATOMIC
SYMARG
PARAM
DEPEND

All executable FORMAC statements are made into Comment cards and CALL's to the appropriate object time subroutine are created. The necessary statements to "trick" the FORTRAN compiler are generated.

2.2 Object Time Subroutines

Operate interpretively, using algorithms for each executable command, and service routines to assist in executing the algorithms.

Use Information from Symbol and Expression Tables
Create Generated Expression Table

2.3 Symbol Table

The symbol table contains data on both FORTRAN and variables, i.e.

BCD name

Characteristics(e.g.FTN,FMC;atomic,let;dimension,etc.)

Pointer to Expression Table (FORMAC variables only)

Value (FORTRAN values only)

Each entry takes at least 2 words (more if it is a dimensioned variable). The first word always contains the BCD name of the variable. The second word contains the bits defining the status and characteristics of a FORMAC variable, or the current value of a FORTRAN variable.

2.4 Expression Table

The expression table contains all expressions which appear in FORMAC statements; it also contains DEPENDENCE and PARAM lists.

Internal Representation is Delimiter Polish

Variables are defined by references to the Symbol Table

Symbols are packed and there is no chaining

2.5. Storage Allocation at Object Time

Expressions being created are stored in Delimiter Polish with 1 symbol per word.

Symbols in expressions are chained; no sublists are used.

Storage used by expressions named on left hand side of a LET or in ERASE is returned to the free list.

Expressions not needed for the specific command being executed are automatically put out on tape if space is needed.

Expressions on tape are automatically returned to core when needed.

3. Object Time Execution

There are object time routines called command level routines to correspond to each of the FORMAC executable statements. There are also a number of service routines which are used by the command level routines. These involve operations on lists, getting symbols, conversion of arithmetic modes, etc. Each command level routine calls Automatic Simplification when it is finished, and sometimes calls it during the execution of the algorithm.

The simplest command is the basic LET, which is always executed as part of all the other commands. The main service routine is UNPACK (which in turn calls on other more basic service routines.) A simplified version of the flow chart for LET is as follows:

LET

Initialize

UNPACK

Automatic Simplification

Get Mode Information

Does mode
of RHS
equal mode
of LHS?

No

Convert

YES

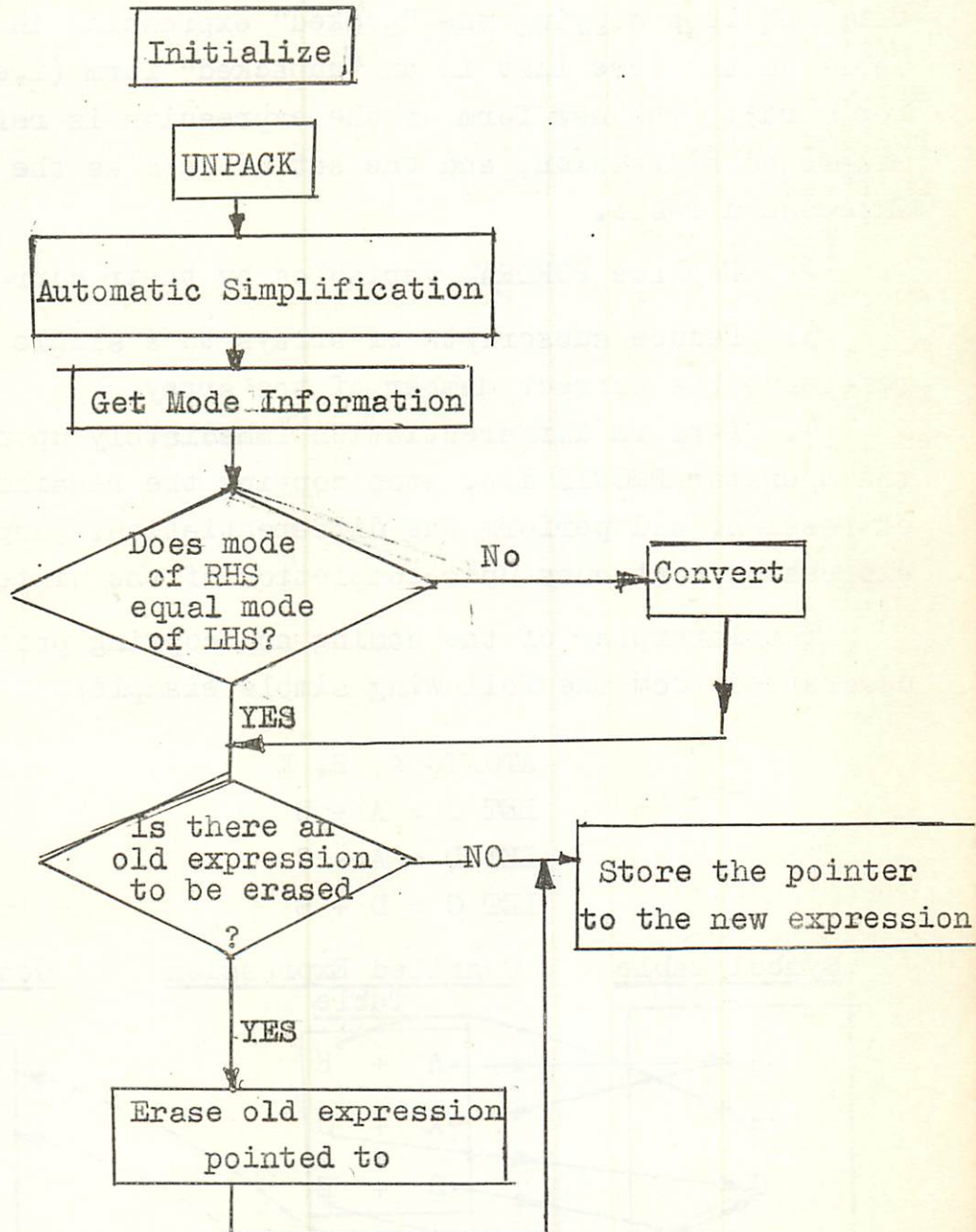
Is there an
old expression
to be erased

NO

Store the pointer
to the new expression

YES

Erase old expression
pointed to



The service routine UNPACK performs the following operations:

1. Replace let variables which appear in the Expression Table by the expressions which they name. (This is sometimes referred to as "unravelling" or "reducing to atomic level".) This requires copying the "packed" expression in the Expression Table on the free list in an "unpacked" form (i.e. one symbol per word). The new form of the expression is referred to as a Generated Expression, and the set of them as the Generated Expression Table.

2. Replace FORTRAN variables by their current values.

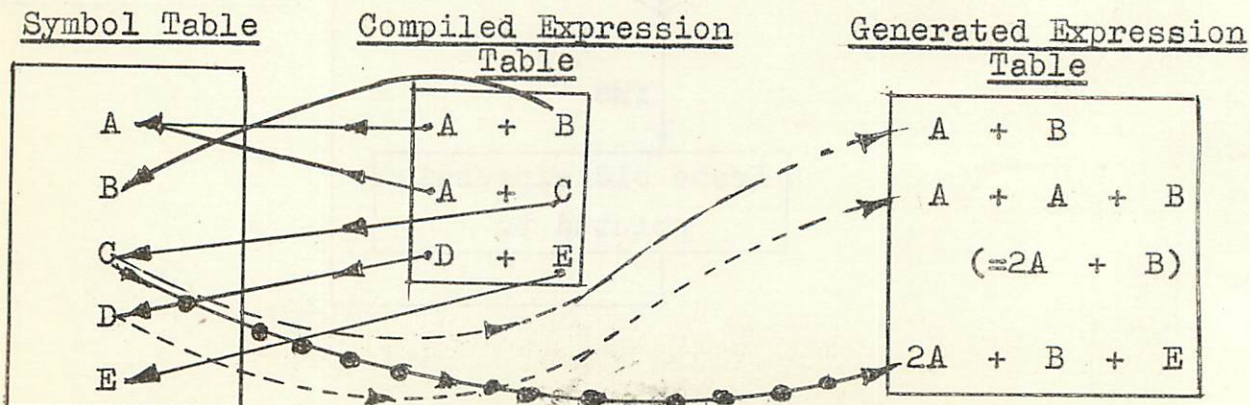
3. Reduce subscripts of arrays to a single value so as to reference the correct member of the array.

4. Perform differentiation immediately upon encountering the operator FMCDIF i.e. stop copying the remainder of the expression, and perform the differentiation. Copying of the expression continues upon completion of the differentiation.

The interplay of the naming and copying process is more easily understood from the following simple example:

```

ATOMIC A, B, E
LET C = A + B
LET D = A + C
LET C = D + E
  
```



All the lines represent pointers to addresses. The solid lines are established at compilation time. The dotted line from C to A + B and the establishment of A + B in the Generated Expression Table is done as part of the execution of the statement "LET C = A + B". The dotted line from D and the creation of A + A + B in the Generated Expression Table is done as part of the statement "LET D = A + C". The Automatic Simplification routine transforms the A + A + B into 2A + B, which is the final form in the Generated Expression Table. The expression 2A + B + E is established as part of executing the statement "LET C = D + E"; execution of this statement also erases the dotted line from C and establishes the dotdash-line from C. The expression A + B in the Generated Expression Table is then erased i.e. the space used by this expression is returned to the free list.

IV. EXAMPLES

1. Mathematical Induction

This FORMAC program proves by mathematical induction that:

$$\sum_{i=1}^n i = n \frac{(n+1)}{2}$$

SYMARG

ATOMIC N

LOGICAL Q

LET SUMN = N*(N + 1.)/2.

results in SUMN → N*(N + 1.)/2.

LET R = EVAL SUMN, (N,1.)

first results in 1.* (1. + 1.)/2.

then R = 1.

IF (R . NE. 1.) GO TO 10


```
C  AT THIS POINT THE ORIGINAL EQUATION HAS BEEN
C  PROVEN TRUE FOR N = 1 IT IS NOW NECESSARY TO
C  PROVE IT FOR THE (N + 1) TH CASE
C  PEPLACE N BY N @ 1 IN ONE EXPRESSION
  LET SUMNPI = SUBST SUMN, (N,N+1.)
      results in SUMPI  $\rightarrow (N + 1.) \times (N + 2.) / 2.$ 
C  ADD THE (N + 1) TH TERM TO THE OTHER EXPRESSION
  LET PROOF = SUM + N + 1.
      results in PROOF  $\rightarrow N \times (N + 1.) / 2 + N + 1.$ 
C  COMPARE THE TWO EXPRESSIONS FOR EQUIVALENCE
  LET Q = MATCH EQ, 0., PROOF, SUMNPI
      results in Q = .TRUE.
  IF (Q. EQ. .TRUE.) GO TO 20

10  STOP
20  CONTINUE
    STOP
    END
```

2. Series Generation and Error Analysis

This example illustrates first the generation of the terms of a series from an expression, and then the creation of series resulting from expanding with respect to error terms. The resultant series has eliminated all terms which are considered to small to be significant in future work.

For the purposes of this example, arbitrary choice were for the function to be expanded, the values for which it would be evaluated and the size of the terms which are to be eliminated. The program as coded will definitely cause an overflow of core for a single expression, and therefore could not run, but was written this way to illustrate the use of the commands.

Given

$$f = \sum_{i=1}^{50} a^i t^{i+1} x^{i-1}$$

$$= at^2 + a^2 t^3 x + a^3 t^4 x^2 + \dots + a^{50} t^{51} x^{49}$$

create the series, then expand each term with respect to errors in two of the variables, and remove terms which are too small.

SYMARG

ATOMIC A, T, X, DT, DA

INTEGER A, T, X, DT, DA, SERF, ERSERF, FNSERF

LET SERF = 0

DO 10 I = 1, 50

LET SERF = SERF + (A~~xx~~1) * T~~xx~~(I + 1) * X~~xx~~(I-1)

10 CONTINUE

C THIS DO LOOP RESULTS IN THE CREATION OF THE SERIES SHOWN ABOVE.

LET ERSERF = SUBST SERF, (T, T + DT), (A, A + DA)

results in ERSERF →

$$(a + \Delta a)(t + \Delta t)^2 + (a + \Delta a)^2(t + \Delta t)^3 x + (a + \Delta a)^3(t + \Delta t)^4 x^2 + \dots$$

LET ERSERF = EXPAND ERSERF

results in ERSERF →

$$at^2 + 2at(\Delta t) + a(\Delta t)^2 + t^2(\Delta a) + 2t\Delta a\Delta t + \Delta a(\Delta t)^2 + \dots$$

C THE RESULT OF THE SUBST AND EXPAND STATEMENTS IS A SERIES IN THE

C FORM SHOWN ABOVE. IF ACTUALLY PROGRAMMED THIS WAY, THE

C VARIABLE ERSERF WILL OVERFLOW CORE WHEN EXPANDED. THE PROPER WAY TO DO THIS IS TO DO THE SUBSTITUTION AND EXPANSION ON ONE TERM AT A TIME.

C THE NEXT PART OF THE PROGRAM SEPARATES OUT EACH TERM AND EVALUATES.

C IT FOR THE DESIRED CONSTANTS AND THEN TESTS THE RESULT.
TERMS WHICH

C HAVE A VALUE CONSIDERED TOO SMALL ARE DROPPED.
LET FNSERT = 0

5 LET TEST = ERSERF

LET TERM = PART ERSERF, M

results the first time through in TERM $\rightarrow at^2$

M = 4

and the second time through in TERM $\rightarrow 2at (\Delta t)$

M = 4

C THE NEXT TWO STATEMENTS ARE NECESSARY TO

C DETERMINE WHEN THE LAST TERM OF THE SERIES IS

C REACHED AND TAKE CARE OF IT.

IF (M .EQ. 4) GO TO 7

SET TERM = TEST

7 LET VALTRM = EVAL TERM, (T,8.), (DT,.02),(A,75),(DA,.5),(X,15)

results the first time through in VALTRM = 24

IF(VALTRM .LE. 1.) GO TO 9

LET FNSERF = FNSERF + TERM

9 IF (M .EQ. 4) GO TO 5

C THIS IS NECESSARY TO DETERMING WHEN THE END OF THE SERIES
HAS

C BEEN REACHED.

STOP

END

INPUT TO FORMAC PREPROCESSOR

SIBFMC MATMUL NODECK

SYMARG
ATOMIC X,Y,Z

FMCDMP LATER

```

LET A(1,1)=X-Y
LET A(2,1)= FMCFAC(3)
LET A(1,2)=Y+FMCDIF(X=3,X,2)
LET A(2,2)=FMCDIF(FMCSIN(Z)+1,Z,1)

```



```

C
C THE ELEMENTS OF THE SECOND MATRIX ARE READ IN
  READ(5,18)(BUF(I),I=1,14)
18 FORMAT(14A6)

```

```

C
C THE DATA READIN AND CONVERTED IS AS FOLLOWS
C
X+Y$ Y-6X$ 3$ Y$3$
LET B(1,1)=ALGCON BUF,R
LET B(2,1)=ALGCON BUF,R
LET B(1,2)=ALGCON BUF,R
LET B(2,2)=ALGCON BUF,R

```

```

C
C
INPUT TO FORMAC PREPROCESSOR

```

```

DO 7 I=1,N
DO 7 J=1,M
88 LET C(I,J)=0
DO 7 K=1,N
99 LET C(I,J)=C(I,J)+A(I,K)*B(K,J)
7 CONTINUE

```

```

C
C THE SUBROUTINE OUTPUT MERELY CONVERTS EXPRESSIONS AND OUTPUTS THEM
CALL OUTPUT(A,6H A)
CALL OUTPUT(B,6H B)
CALL OUTPUT(C,6H C)

```

```

C
C THIS LOOP EXPANDS EACH ELEMENT OF THE MATRIX C
DO 9 I=1,N
DO 9 J=1,M
LET EXPNDC(I,J)=EXPAND C(I,J)
9 CONTINUE
CALL OUTPUT(EXPNDC,6HEXPNDC)

```

```

C
C THE NEXT STATEMENT GET THE DETERMINENT OF THE MATRIX C
LET KDETR=EVAL C(1,1)*C(2,2)-C(1,2)*C(2,1),(Z,0),(X,3/2),(Y,FMCFAC
1(M/N))

```

```

C
WRITE(6,25)KDETR
25 FORMAT( ///15X33H THE VALUE OF THE DETERMINENT IS ,I5)

```

```

C
FMCDMP
STOP
END

```


MATMUL

EXTERNAL FORMULA NUMBER - SOURCE STATEMENT - INTERNAL FORMULA NU

FORTRAN PROGRAM PRODUCED BY PREPROCESSOR
THIS PROGRAM ILLUSTRATES THE USE OF FORMAC TO MULTIPLY
TWO MATRICES WHICH HAVE NON-NUMERIC ELEMENTS.
NOTE THAT THE PROGRAM IS VERY SIMILAR TO THE
FORTRAN PROGRAM NEEDED TO MULTIPLY MATRICES WITH
NUMERIC ELEMENTS. THE MAIN DIFFERENCE IS THE
USE OF THE FORMAC KEY WORD LET IN STATEMENTS
88 AND 99 AND INITIAL VALUE ASSIGNMENTS, AND THE ADDITION OF
THE FORMAC DECLARATIONS AND CONVERT COMMAND. NOTE THAT THE
ELEMENTS OF THE MATRICES BEING MULTIPLIED COULD BE BROUGHT IN FROM
TAPE OR CARDS USING READ AND ALGCON. ALTHOUGH ONLY 2 X 2 MATRICES
WERE USED, OBVIOUSLY THE PROGRAM WILL WORK FOR OTHER SIZES JUST BY
CHANGING THE VALUES OF N AND M.
THE RESULTS ARE PRINTED IN TWO FORMS. ONE HAS
THE ELEMENTS OF THE RESULT IN UNEXPANDED FORM
AND THE OTHER SHOWS THEM EXPANDED AND SIMPLIFIED. THE
DOLLAR SIGN IS SIMPLY AN END OF EXPRESSION MARKER.

SYMARG

THE FOLLOWING STATEMENTS WERE ADDED BY FORMAC

DIMENSION FMCSTB(1), FMCETB(1)

CALL FMCUSE(FMCSTB(1), FMCETB(1))

ATOMIC X,Y,Z

DIMENSION C(2,2), ANS(21), EXPANS(21),

1 A(2,2), B(2,2), EXPNDC(2,2)

DIMENSION BUF(14)

INTEGER A,B,C,X,Y,ANS,EXPANS,EXPNDC,Z

FMCDMP LATER

CALL FMCDMP(002040000000,0,0,FMCSTB)

N=2

M=2

R=0.

INTERNAL FORMULA NOS.

(pul in by hand because
of width of printout

1

2

3

4

C	LET A(1,1)=X-Y	5
C	CALL LET(A(1,1),FMCETB(2))	
C	LET A(2,1)= FMCETB(3)	6
C	CALL LET(A(2,1),FMCETB(4))	
C	LET A(1,2)=Y+FMCDIF(X 3 3,X,2)	7
C	CALL LET(A(1,2),FMCETB(6))	
C	LET A(2,2)=FMCDIF(FMCSIN(Z)+1,Z,1)	8
C	CALL LET(A(2,2),FMCETB(11))	
C	THE ELEMENTS OF THE SECOND MATRIX ARE READ IN	9
C	READ(5,18)(BUF(I),I=1,14)	10,11,12,13,14
18	FORMAT(14A6)	
C	THE DATA READING AND CONVERTED IS AS FOLLOWS	
C	X+Y 3 Y-6 X 3 3 3 Y 3 3	
C	LET B(1,1)=ALGCON BUF,R	
C	CALL ALGCON(B(1,1), BUF,R ,FMCSTB)	
C	LET B(2,1)=ALGCON BUF,R	15
C	CALL ALGCON(B(2,1), BUF,R ,FMCSTB)	
C	LET B(1,2)=ALGCON BUF,R	16
C	CALL ALGCON(B(1,2), BUF ,R ,FMCSTB)	
C	LET B(2,2)=ALGCON BUF,R	17
C	CALL ALGCON(B(2,2), BUF,R ,FMCSTB)	
C		18
C		19
C	DO 7 I=1,N	
C	DO 7 J=1,M	
C 88	LET C(I,J)=0	20
C 88	CALL LET(C(I,J),FMCETB(16))	21
C	DO 7 K=1,N	
C 99	LET C(I,J)= C(I,J)+A(I,K) *B (K,J)	22
C 99	CALL LET(C(I,J),FMCETB(18))	23
7	CONTINUE	

THE SUBROUTINE OUTPUT MERELY CONVERTS EXPRESSIONS AND OUTPUTS THEM 24,25,26,27
 CALL OUTPUT (A,6H A) 28
 CALL OUTPUT (B,6H B) 29
 CALL OUTPUT (C,6H C)

THIS LOOP EXPANDS EACH ELEMENT OF THE MATRIX C 30
 DO 9 I=1,N 31
 DO 9 J=1,M
 LET EXPNDC(I,J)=EXPAND C(I,J) 32
 CALL EXPAND(EXPNDC(I,J),FMCETB(25),0) 33
 CONTINUE 34,35,36
 CALL OUTPUT(EXPNDC,6HEXPNDC)

THE NEXT STATEMENT GET THE DETERMINENT OF THE MATRIX C
 LET KDETR=EVAL C(1,1)*C(2,2)-C(1,2)*C(2,1),(Z,0),(X,3/2),(Y,FMCFAC);- 37
 1(M/N))
 CALL EVAL(KDETR,0,FMCETB(27),FMCETB(40))

WRITE(6,25)KDETR 38
 25 FORMAT(///15X33H THE VALUE OF THE DETERMINENT IS , I5) 39,40,41

FMCDMP
 CALL FMCDMP(000040000000,0,0,FMCSTB) 42
 STOP 43
 EQUIVALENCE (FMCFST(1),FMCSTB(1)),(FMCFET(1),FMCETB(1))
 DIMENSION FMCFST(48),FMCFET(50)
 DATA(FMCFST(IMCZ),IMCZ = 1, 48)/0000017000020,1HI,0,1HJ,0,1H
 1K,0,5HKDETR,0,1HM,0,1HN,0,1HR,0,0000000000001,00000000000037,1HA,02
 100002200004,4*6H000+00,1HB,0200002200004,4*6H000+00,1HC,0200002200
 1004,4*6H000+00,6HEXPNDC,0200002200004,4*6H000+00,1HX,0000000100000
 1,1HY,0000000100000,1HZ,0000000100000,0004000000000/
 DATA(FMCFET(IMCZ),IMCZ = 1, 50)/00000000000062,0500001252400,
 10026214000000,07110000000000,00016000000000,0500001317256,0000124400
 1000,00000000600012,04400000000000,0050614000000,0752462000027,010000
 10000000,0061000056200,00000000000143,00000000000000,02000000000000,00
 1400000000000,0511000017440,0000420000443,0322000023100,000104000150
 16,0200003110000,0304000110614,03000000000000,0440000762000,00210000

122140,0513220000372,00000000000001,02000000000000,0143100001750,0000
 10000000011,00000000000001,0214326644000,0076400000000,0000240000000,
 10000050620000,0372000000000,0002200000000,0000143061400,0000056200
 1000,0000000046000,0125310000000,0000016720000,0000000022400,000000
 10001431,0400026345304,0000253440003,0050000000000,0030600000000,06
 1000000000000/

EQUIVALENCE (FMCFST(3),I),(FMCFST(5),J),(FMCFST(7),K),(FMCFST(9),K
 1DETR),(FMCFST(11),M),(FMCFST(13),N),(FMCFST(15),R),(FMCFST(20),A),
 1(FMCFST(26),B),(FMCFST(32),C),(FMCFST(38),EXPND), (FMCFST(43),X),(
 1FMCFST(45),Y),(FMCFST(47),Z)

END

44

INPUT TO FORMAC PREPROCESSOR

\$IBFMC PRNT NODECK

SUBROUTINE OUTPUT(EXP,NME)

C E.R.BOND - BAP DEPT.

C THIS ROUTINE CONVERTS AND PRINTS OUT EXPRESSIONS

DIMENSION BUF(21),NME(2),EXP(2,2)

SYMARG EXP

DO 10 I=1,2

DO 10 J=1,2

TELL=0.

20 LET TELL = BCDCON EXP(I,J),BUF,21

C

WRITE (6,22) NME(1),I,J,(BUF(M),M=2,13)

22 FORMAT(/15XA6,1H(,11,1H,11,2H)=,2X12A6)

10 CONTINUE

RETURN

END

PRNT

EXTERNAL FORMULA NUMBER - SOURCE STATEMENT - INTERNAL FORMULA. N -

SUBROUTINE OUTPUT(EXP,NME)

E.R.BOND - BAP DEPT.

THIS ROUTINE CONVERTS AND PRINTS OUT EXPRESSIONS

DIMENSION BUF(21),NME(2),EXP(2,2)

SYMARG EXP

THE FOLLOWING STATEMENTS WERE ADDED BY FORMAC

DIMENSION FMCSTB(1),FMCETB(1)

CALL FMCUSE(FMCSTB(1),FMCETB(1))

CALL FMCUPD (FMCETB(2), EXP)

DO 10 I=1,2

DO 10 J=1,2

TELL=0.

20 LET TELL = BCDCON EXP(I,J),BUF,21

20 CALL BCDCON(TELL ,FMCETB(3),BUF,21)

WRITE (6,22) NME(1),I,J,(BUF(M),M=2,13)

22 FORMAT(/15XA6,1H(,11,1H,11,2H)=,2X12A6)

10 CONTINUE

RETURN

EQUIVALENCE (FMCFST(1),FMCSTB(1)),(FMCFET(1),FMCETB(1))

DIMENSION FMCFST(12),FMCFET(5)

DATA(FMCFST(IMCZ),IMCZ = 1, 12)/0000007000010,1HI,0,1HJ,0,4H

1TELL,0,00000000000010000000000003,3HEXP,0,01000000000000/

DATA(FMCFET(IMCZ),IMCZ = 1, 5)/00000000000005,0000012400000,

10440000242000,0021000022140,06000000000000/

EQUIVALENCE (FMCFST(3),I),(FMCFST(5),J),(FMCFST(7),TELL)

END

1

2

3

4

5

6

7,8,9,10,11,12

13,14,15

16

17

FMCDMP CALLED FROM STATEMENT NUMBER 2 IN DECK "MATMUL"

$A(1,1) = X - Y$
 $A(1,2) = X + 6 + Y$
 $A(2,1) = 6$
 $A(2,2) = \text{FMCCOS}(Z)$
 $B(1,1) = X + Y$
 $B(1,2) = 3$
 $B(2,1) = X(-6) + Y$
 $B(2,2) = Y + 3$
 $C(1,1) = (X + Y)(X - Y) + (X(-6) + Y)(X + 6 + Y)$
 $C(1,2) = (X - Y)3 + (X + 6 + Y)Y$
 $C(2,1) = (X + Y)6 + (X(-6) + Y)\text{FMCCOS}(Z)$
 $C(2,2) = Y\text{FMCCOS}(Z) + 3 + 18$
 $\text{EXPND}(1,1) = X^2(-35)$
 $\text{EXPND}(1,2) = XY + 18 + X^3 + Y(-3) + Y^2 + 3$
 $\text{EXPND}(2,1) = X\text{FMCCOS}(Z)(-6) + X + 6 + Y\text{FMCCOS}(Z) + Y + 6$
 $\text{EXPND}(2,2) = Y\text{FMCCOS}(Z) + 3 + 18$

THE VALUE OF THE DETERMINENT IS -882

THIS IS A FORMAC DUMP OF ROUTINE "PRNT"

FORTRAN VALUES

ADDRESS	SYMBOL	SUBSCRIPTS	TYPE	VALUE
13134	I		INTG-NUMB	2
13136	J		INTG-NUMB	2
13140	TELL		REAL-NUMB	0.

FORMAC VARIABLES

ADDRESS	SYMBOL	SUBSCRIPTS	TYPE	VALUE	EXPRESSION
13144	EXP				THIS SYMBOL IS A FORMAC DUMMY - IT REPRESENTS THE SYMBOL BELOW FROM DECK "MATM"
12604	EXPNDG	1 1	INTG-SET	000000262320	XXX2X(-35)8

MCDMP CALLED FROM STATEMENT NUMBER 42 IN DECK "MATMUL"

THIS IS A FORMAC DUMP OF ROUTINE "MATMUL"

FORTTRAN VALUES

ADDRESS	SYMBOL	SUBSCRIPTS	TYPE	VALUE
12541	I		INTG-NUMB	2
12543	J		INTG-NUMB	2
12545	K		INTG-NUMB	2
12547	KDETR		INTG-NUMB	-882
12551	M		INTG-NUMB	2
12553	N		INTG-NUMB	2
12555	R		REAL-NUMB	0.1723144E-39

FORMAC VARIABLES

ADDRESS	SYMBOL	SUBSCRIPTS	TYPE	VALUE	EXPRESSION
12562	A	1 1	INTG-SET	000000262067	X-Y \S
12563	A	2 1	INTG-SET	000000262076	6 \S
12564	A	1 2	INTG-SET	000000262071	X \S 6+Y \S
12565	A	2 2	INTG-SET	000000262131	FMCCOS(Z) \S
12570	B	1 1	INTG-SET	000000262135	X+Y \S
12571	B	2 1	INTG-SET	000000262156	X \S (-6)+Y \S
12572	B	1 2	INTG-SET	000000262155	3 \S
12573	B	2 2	INTG-SET	000000262147	Y \S 3 \S
12576	C	1 1	INTG-SET	000000262153	(X+Y) \S (X-Y)+(X \S (-6)+Y) \S (X \S 6+Y) \S
12577	C	2 1	INTG-SET	000000262142	(X+Y) \S 6+(X \S (-6)+Y) \S FMCCOS(Z) \S
12600	C	1 2	INTG-SET	000000262146	(X-Y) \S 3+(X \S 6+Y) \S Y \S 3
12601	C	2 2	INTG-SET	000000262247	Y \S FMCCOS(Z) \S 3+18 \S
12604	EXPND C	1 1	INTG-SET	000000262320	X \S 2 \S (-35) \S
12605	EXPND C	2 1	INTG-SET	000000262306	X \S FMCCOS(Z) \S (-6)+X \S 6+Y \S FMCCOS(Z) \S
12606	EXPND C	1 2	INTG-SET	000000262375	X \S Y \S 18+X \S 3+Y \S (-3)+Y \S 2 \S 3 \S
12607	EXPND C	2 2	INTG-SET	000000262272	X \S FMCCOS(Z) \S 3+18 \S
12611	X		INTG-ATOM	000000100000	
12613	Y		INTG-ATOM	000000100000	
12615	Z		INTG-ATOM	000000100000	