The Design and Implementation of a Translator for Arithmetic and Boolean Expressions.

Marin L. Bishop

AFIT/STUDENT AT: Texas A&M University

AFIT/NR
WPAFB OH 45433

MONITORING AGENCY NAME AND ADDRESS

14 MONITORING AGENCY NAME AND ADDRESS (different from Controlling Office)

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

DTIC ELECTED
D JUL 21 1981

SUPPLEMENTARY NOTES

APPROVED FOR PUBLIC RELEASE: IAW AFR 190-17

23 JUN 1981

FREDRIC C. LYNCH
Major, USAF
Director of Public Affairs
Air Force Institute of Technology (AFIT)
Wright-Patterson AFB, OH 45433

KEYWORDS (Continue on reverse side if necessary and identify by block number)

ABSTRACT (Continue on reverse side if necessary and identify by block number)

ATTACHED

81716008
AFIT RESEARCH ASSESSMENT

The purpose of this questionnaire is to ascertain the value and/or contribution of research accomplished by students or faculty of the Air Force Institute of Technology (AFIT). It would be greatly appreciated if you would complete the following questionnaire and return it to:

AFIT/NR
Wright-Patterson AFB OH 45433

RESEARCH TITLE: The Design and Implementation of a Translator for Arithmetic and Boolean Expressions

AUTHOR: Marin L. Bishop

RESEARCH ASSESSMENT QUESTIONS:

1. Did this research contribute to a current Air Force project?
   ( ) a. YES ( ) b. NO

2. Do you believe this research topic is significant enough that it would have been researched (or contracted) by your organization or another agency if AFIT had not?
   ( ) a. YES ( ) b. NO

3. The benefits of AFIT research can often be expressed by the equivalent value that your agency achieved/received by virtue of AFIT performing the research. Can you estimate what this research would have cost if it had been accomplished under contract or if it had been done in-house in terms of manpower and/or dollars?
   ( ) a. MAN-YEARS ( ) b. $_______

4. Often it is not possible to attach equivalent dollar values to research, although the results of the research may, in fact, be important. Whether or not you were able to establish an equivalent value for this research (3. above), what is your estimate of its significance?
   ( ) a. HIGHLY SIGNIFICANT ( ) b. SIGNIFICANT ( ) c. SLIGHTLY SIGNIFICANT ( ) d. OF NO SIGNIFICANCE

5. AFIT welcomes any further comments you may have on the above questions, or any additional details concerning the current application, future potential, or other value of this research. Please use the bottom part of this questionnaire for your statement(s).

NAME ____________________ GRADE ____________________ POSITION ____________________
ORGANIZATION ____________________ LOCATION ____________________

STATEMENT(s):

[Blank for statement(s)]
THE DESIGN AND IMPLEMENTATION OF A TRANSLATOR
FOR ARITHMETIC AND BOOLEAN EXPRESSIONS

RESEARCH REPORT

Presented in Partial Fulfillment of the Requirements
For the Degree Master of Computing Science, Industrial
Engineering Department of Texas A&M University

By

Marvin L. Bishop

Approved by

Jack A. Barnes
Chairman

Sallie Sheppard

Winston T. Shearon

Texas A&M University
1980
This paper describes an algorithm for scanning commands of a specific query language for a data management system. The commands include relational, arithmetic assignment, and Boolean expressions. The algorithm accepts the expressions in conventional infix notation, transforms them into postfix notation, then into an efficient set of computing steps known as ordered triples. Structured programming is used in that extensive, indented comments form the structure and FORTRAN code carries out the instructions of the comments.
ACKNOWLEDGEMENTS

I would like to thank Pocky Wanting for providing the specifications and requirements for this project. He was patient and willing to explain the system interface and uses. Jay Soper was helpful in developing structured methods for structuring and documenting the programming code. Thanks go to Dick Dickinson for allowing my participation in the project.

I am also grateful to Dr. Jack A. Barnes, the chairman of my committee, for accepting the topic for research, for his advice, and for his patience during the preparation of this paper. The remainder of my advisory committee, Dr. Sallie Sheppard and Dr. Winston Shearon, should also be thanked for their guidance throughout my graduate program.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>I    INTRODUCTION</td>
<td>1</td>
</tr>
<tr>
<td>II   THE INTERPRETER</td>
<td>7</td>
</tr>
<tr>
<td>III  DESIGN CONSIDERATIONS</td>
<td>14</td>
</tr>
<tr>
<td>IV   DESCRIPTION OF ALGORITHMS</td>
<td>19</td>
</tr>
<tr>
<td>Full Assignment Clause</td>
<td>19</td>
</tr>
<tr>
<td>Boolean Expression</td>
<td>30</td>
</tr>
<tr>
<td>V    CONCLUSION</td>
<td>36</td>
</tr>
<tr>
<td>Future Efforts</td>
<td>37</td>
</tr>
</tbody>
</table>

# APPENDICIES

<table>
<thead>
<tr>
<th>Appendix</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>A    SYNTAX GRAPHS</td>
<td>41</td>
</tr>
<tr>
<td>B    USER'S GUIDE</td>
<td>45</td>
</tr>
<tr>
<td>C    PROGRAM LISTINGS</td>
<td>49</td>
</tr>
</tbody>
</table>

REFERENCES ...................................................... 75
## FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. HEIRARCHY CHART</td>
<td>17</td>
</tr>
<tr>
<td>2. FLOW CHART FOR CTIFS</td>
<td>21</td>
</tr>
<tr>
<td>3. FLOW CHART FOR RELATE</td>
<td>23</td>
</tr>
<tr>
<td>4. FLOW CHART FOR FCSTFX</td>
<td>26</td>
</tr>
<tr>
<td>5. FLOW CHART FOR BLDIPP</td>
<td>28</td>
</tr>
<tr>
<td>6. FLOW CHART FOR TRIPLES</td>
<td>34</td>
</tr>
</tbody>
</table>
Chapter I

Introduction

Training personnel to perform their assigned duties is a never-ending responsibility of managers. Personnel and duties are both continually changing. Even if the personnel and the basic work to be accomplished could be considered to be constant, the approach to accomplish the work and the detailed tactics used will vary as individuals seek easier and better methods to make their work more pleasing. Managers and supervisors are also responsible for change as they blend their personality into the organization and search for more efficient, more productive, and less expensive means to accomplish the goal.

One significant aspect of the continual change in personnel and work is the appropriate training they need to properly perform the duty. Certainly they must be sufficiently trained to achieve a certain level of proficiency to make them productive in the job. However, over training or training in tasks which they do not perform is an unnecessary expense and thus should be avoided if at all possible. To optimize the benefit-training ratio, a manager should expend resources to train personnel only for
those tasks which they will actually perform on the job.

Optimized training appears ideal when considering one
person and one job which can be specifically defined or if
numerous individuals are performing exactly the same tasks.
This ideal situation seldom occurs in the real world. A
more plausible situation within a given organization is for
a number of the employees to perform identical or similar
tasks for a certain portion of their work-week, but only a
few employees would be doing tasks which are identical or
sufficiently similar such that the same training curriculum
would be suitable.

The next logical step in analyzing training
requirements is in effort to identify, categorize, and group
the tasks performed. As an example, suppose employee A
spends 50% of his time performing task 1, 30% performing task
2, and 20% performing task 4. Employee B's time is
distributed as 30% on task 1, 60% on task 3, and 10% on task
4 which employee A did not perform. For employee C, assume
a distribution of 40% on task 2, 20% on task 3, and 40% on
task 4. The overlap of duties is quite evident and
considering economy of scale, all employees should be
trained together instead of a training program being
customized for each of them. On the other hand, training
employee A on task 4 is a wasted effort. To obtain maximum
return for training resources expended, the employees should
be grouped for the various training sessions. Employees A and B would be trained on task 1, A and C on task 2, B, C, and D on task 3, and A and C on task 4. With this procedure the employees are trained on only those tasks which they will be performing (no training is wasted) and the training classes are as large as possible (or feasible) for economic considerations.

Expanding the example to a large organization of several hundred employees performing various tasks can provide considerable savings if the job analysis is properly conducted. The same information used to conduct the job analysis may also be quite beneficial for preparing job descriptions, reassigning tasks to employees, reducing time consuming tasks, identifying problem areas, and in general improving the overall effectiveness of the organization.

This is where occupational analysis can be effectively used. Occupational analysis is one means of collecting data about the tasks performed within an organization. It includes collecting information about the job from the people, their supervisors, and others. Occupational analysis is well accepted throughout the world and is used extensively by military services, civilian governments, universities, and industry (2).

In gathering the information, respondents are asked to indicate on a relative scale how much time they spent
performing the different tasks listed on a questionnaire. Those responses are processed to provide a percentage of
time devoted to the various tasks. In addition to the task-related information, history and secondary data is also
netheral, again through responses to written questions.
History data will usually include age, sex, race, skills,
experience in the position, and similar information.
Secondary data is normally related to how important each
task is or how much training is required for each task. The
data collected is in a quantified form or can be readily
converted to a quantified form. It should be obtained from
large samples of respondents to minimize the effect of
variations. It is in the processing and analysis of these
responses that the computer plays a significant role.
The Air Force has been one of the pioneers in
occupational analysis. In the early sixties, they began
developing several computer programs to do the analysis, a
system which has become known as the Comprehensive
Occupational Data Analysis Program (CODAP). Since the
beginning of CODAP and as more experience in occupational
analysis was gained, several programs have been added to the
system to increase its capability. Unfortunately, no
overall system design existed and the documentation left
much to be desired. As might be expected and too often
experienced, this evolutionary process resulted in a system
which is difficult to modify and not totally responsive to the needs of the users. Realizing the limitations of the original system, the Occupational Research Program of Texas A&M University is under contract with the U.S. Navy to rewrite the programs. Following the initial analysis of the system and through consultation with the Navy, other military services, and other CODAP users, it was agreed that an entirely new design of portions of the system would be preferable to just rewriting existing programs. The new design should emphasize documentation, understandability, reliability, and transportability while providing increased data manipulation capability at a decrease in cost and time involved.

The data gathering, editing, sorting, and creating of the initial data files were included in the rewrite portion of the system. The portion which is being redesigned includes a consolidation of programs which will manipulate the data in the files. The data manipulation programs will be based on an interpreter which processes statements of the newly designed CODAP language.

With this introduction to job analysis and the CODAP rewrite project, the remainder of this paper will focus on the design of a small segment of the CODAP programs. An interpreter to manipulate the CODAP data base is discussed in chapter 2. Chapter 3 delineates the overall design
objectives of achieving good, useful software and the application to the programs being developed. A detailed description of the program logic and interface is in chapter 4. Chapter 6 summarizes the paper and suggests areas which may be of interest for additional work.
CHAPTER II

THE INTERPRETER

The decision to redesign the portion of the system which manipulates the data was based in part on a desire to provide a more natural, easier to use system. The original CODAP system was quite tedious since users were required to indicate their desired data manipulation action by placing numbers in specified card columns. This procedure was slow and quite prone to inducing errors. To overcome this problem, it was decided to use free format, easily understandable English-like commands.

The Statistical Analysis System (SAS) language nearly satisfied the above requirements but was not totally satisfactory (7). Some of the actions in occupational analysis are quite unique. Consequently a new language, the CODAP language, was developed to enhance the usefulness of the system. The new CODAP language is designed to permit convenient manipulation of the job analysis data. The data, which is logically stored in table format, can be manipulated by either rows or columns with nearly equal ease.

Since the purpose of this paper is to present the design and implementation of only a portion of the CODAP interpreter,
the language is not covered in detail. Only those portions of the language applicable to this paper are explained. A full description of the CODAP language may be found in the CODAP User's Manual (5).

This paper is limited to the design and implementation of modules which will handle the "full assignment clause" and the "Boolean expression". The syntax graphs which illustrate these two commands are in appendix 1. The lines indicate the possible paths which may be followed. A divergence of lines means either path is possible. Circles and ovals denote the enclosed expression is to be included in the command string exactly as specified.

As an example, consider the full assignment clause.

It may consist of a simple assignment clause or the IF-THEN-ELSE expression which may have one or more nested IF-THENs, but only one ELSE-clause. A possible full assignment clause, as expressed in the CODAP language, might be

IF Bill = 3 THEN Tom := 5
IF Bill = 4 THEN Tom := 6
ELSE Tom := 1 'comment'.

The full assignment clause is the major portion of the CREATE command which may be used to add new columns or rows to the data base. The "Bill = 1" portion is a relational expression which is tested for a true or false indication to determine which assignment clause is executed. Only on-
assignment clause of the full assignment clause will be
executed.

A typical Boolean expression for selecting a subset
of the CODAP data base might be

\[ \text{IN G1 .AND. } ((T1 \leq 3) \ .OR. (T4 = 5)) \] .

The "IN G1" indicates only the group of columns included in
the label G1 are to be considered. "NOT IN" is an alternate
indicator of which groups are to be excluded. "T1 \leq 3"
indicates only those columns where the T1 variable is less
than or equal to three will be accepted. However, in the
above example, if the T4 variable is five, the value of T1
may be greater than three and the column is still considered.
The relational expressions "T1 \leq 3" and "T4 = 5" are the
operands for the ".OR." logical operator. The operands for
the ".AND." operator are "IN G1" and the interim result of
\((T1 \leq 3) \ .OR. (T4 = 5))\).

It is possible to structure the source language for
direct execution. However, such a procedure does restrict
the flexibility in formatting the source language and reduces
the efficiency of program execution. For complex source-
languages which have a somewhat complicated goal it is common
practice to first translate the source language into an
internal form which is easier to handle mechanically. In
most internal forms, the operators normally appear in the
order in which they are to be executed (3). Long command
strings or sentences are broken down into short phrases of a single operator and the necessary operand(s). The program which does the translation places the phrases in the proper order for execution. Thus the execution program does not need to be concerned with precedence of operators, but has the simplified task of executing in order of sequence, one at a time.

Arithmetic and Boolean expressions, as commonly written, can be quite complex for a computer to process. The normal expressions used are known as "infix notation" since the binary operators are placed between the two operands on which they are to operate. Unary operators are placed immediately in front of their respective operand. To establish a single, correct sequence of execution of the operators, the operators are generally assigned a precedence—those of a higher precedence being executed before those of a lower precedence. The precedence execution order can be modified by the use of parentheses executing the expression enclosed by parentheses before the operations outside the parentheses.

Human beings find the infix notation quite understandable since it is the most commonly used system and they have generally been exposed to it since their earliest arithmetic classes. The ability to scan and comprehend more than one symbol at a time coupled with the free use of parentheses, sometimes added merely to improve clarity, aids in the
comprehension of this notation. However, a mechanical device, or electronic in the case of a computer, does not possess the same capability as humans and thus does not respond as well to infix notation. A computer is essentially restricted to considering a single symbol at a time and comparing that symbol with another. Further, parentheses which humans aid for clarity are an unnecessary added symbol to the computer, requiring additional processing time. Consequently, to aid the computer, infix expressions are often translated into another form such as suffix or postfix notation.

Postfix notation will be used in the CODAP interpreter. Parentheses are not required in postfix notation and operators are placed in exactly the order in which they are to be executed. This eliminates the possible confusion of operator precedence and reduces the number of symbols which the computer must process. As examples, A + B would be written as AB+ in postfix notation; and A + B * C as ABC*. It should also be noted that the variables and constants appear in exactly the same order in both infix and postfix notation, and in postfix the operands appear immediately to the left of the operators.

Once an expression has been converted to postfix notation, a second conversion is easily accomplished to achieve a convenient form for a single binary operator—"a triple."
The triple may be expressed as

\[(\text{<operator>}, \text{<operand 1>}, \text{<operand 2>})\]

where \text{<operand 1>} and \text{<operand 2>} specify the arguments for the operator. As an example, \(A + B\) might be represented by 

\[+, A, B\]

and \(A \times 3 + C \times D\) could be represented by the sequence

1. \[\times, A, B\]
2. \[\times, C, D\]
3. \[+, (1), (2)\].

The numbers in parentheses indicate that operand is the interim result obtained from the triple row illustrated (see reference 3). Interim results are often pushed onto a stack, then popped from the stack whenever required in a succeeding triple. For unary operators, one of the operand positions is left blank. The significance of ordered triples is that the triples appear in the sequence in which they are to be executed. Once an expression is in ordered triple form, it can be efficiently executed numerous times by a computer.

The objective of the module designs discussed in this paper is to accept the numerical tokens of normal infix notation which are the English-like commands as input. These commands are converted to postfix notation where necessary for easier processing. The tokens are then placed in an ordered triple array which is the output from the modules. Chapter 1 discusses the considerations of
good design which will be applied in the design of these modules.
CHAPTER III

DESIGN CONSIDERATIONS

Several factors should be given careful consideration when beginning the design of computer software. The software should be understandable, modifiable, reliable, useful, transportable, and efficient. Understandable code can be produced by good documentation techniques. The documentation should be adequate to meet the needs of the user and those responsible for maintaining the software. It should be standardized and uniform. External documentation should provide an overall design of the program modules and the relationship with calling and called modules. In line comments should be sufficient to allow reasonably knowledgeable programmers to easily follow the logic and understand the code. The comments in the programs produced for this report set the structure of the main program logic and the code carries out the logic of the comments.

Software which is understandable has a head start on being modifiable. If it can be understood, it can usually be modified. Additionally, tricky code should be avoided. It should be modularized, each module performing a single function. Thus any necessary changes will affect the
fewest number of lines of code possible. Subroutines should be completely contained on one page to allow an easy overall view. Subroutines developed for this project have been limited to less than 100 lines of code which should be comprehensible to any programmer familiar with the programming language.

To be reliable, the code should respond appropriately with any and all of the possible ranges of input data which it might encounter. The command string which is to be provided to the modules under consideration will have been previously checked for syntax errors. Consequently few routines have been included for handling erroneous input. This also conforms with the specifications provided.

Disallowing syntax errors, reliability can be checked by testing the range of possibilities of acceptable input. Although not all possible combinations can reasonably be tested, sufficient variations of input should be encountered to cause each part of the code to be exercised. Such a procedure should provide a high degree of reliability.

To be efficient, the code should execute in minimal computer time. Programming shortcuts and machine dependent techniques are methods used to improve efficiency. Both are contrary to the goals of understandable, modifiable, and transportable. Consequently some loss in efficiency will be accepted to enhance other goals.
Transportability is of major concern for the CPAI programs. They will be used by various agencies on various makes of computers and should perform well on all occasions. To enhance transportability standard ANSI FORTRAN has been selected as the language for all modules. It is common enough that any facility of reasonable size would likely already have, or can reasonably acquire, the necessary FORTRAN compiler.

The last and probably the most important goal discussed is usefulness. To be useful, the code must perform the desired process for which it was produced. This is verified by a rigorous test effort, inputting test data and carefully checking the output for the intended results. The system will be useful to more facilities by using a common programming language, FORTRAN, as discussed with transportability.

Structured programming has been used extensively throughout. A top-down approach was taken to break the problem into manageable portions and to enhance the goals of understandable, reliable, and modifiable code. A hierarchy chart of the system designed can be found in Figure 1. The program structure could have been improved using a language with structured constructs, such as FL/I or ALGOL. However, FORTRAN was the specified language. Therefore, to provide as much structure as possible, comments are plentiful and structured in nature. The "if-then-else" construct is used
Figure 1. Hierarchy Chart
frequently to begin the comments associated with a conditional statement. Indentation is employed to provide a visual indication of block structure and the range of the if-then-else construct. The comment "end of if" signifies the end of an if-then-else block of code. Although such an approach requires an abundant use of GC T0s, they are the only means of transferring control for a conditional construct in the FORTRAN language. Usually the GC TC construct should be avoided as much as possible to make code easy to follow. However, the author believes the structure of the comments and indentation along with the controlled use of GC T0s has resulted in a well-structured and quite understandable program considering the limitations of the FORTRAN language.

Based on the overview of the design philosophy, chapter 4 will describe in detail the algorithms used. It will describe logic used to transform command strings into ordered triplets.
CHAPTER IV

DESCRIPTION OF ALGORITHMS

The purpose of this project was to design and implement modules to perform two separate functions. Both the full assignment clause and Boolean expressions had to be transformed into the internal form of ordinal triples for efficient execution. Although the two outputs were to be of the same format, separate processes were desired since the inputs are not sufficiently similar. This is also in agreement with the good software engineering techniques of one function per module which enhances program understandability. The programs to process the full assignment clause were developed first. These were used as a starting point for developing the programs which process the Boolean expressions.

Full Assignment Clause

The full assignment clause, as discussed in chapter 2, consists of an IF-THEN-ELSE clause. The IF portion is optional and if present is composed of a simple relational expression. The objects of THEN and ELSE will be an arithmetic expression whose value is computed and assigned to a variable. The syntax graph appears in appendix 1.
The full assignment clause example presented in chapter 2 is repeated here for clarity.

IF Bill = 3 THEN TCM := 5
IF Bill = 4 THEN TCM := 6
ELSE TCM := 1 "comment".

When transformed to ordered triples this expression becomes:

<table>
<thead>
<tr>
<th>Row</th>
<th>Operator</th>
<th>Operands</th>
<th>Operands</th>
<th>Operands</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>SUB</td>
<td>Bill</td>
<td></td>
<td>3</td>
</tr>
<tr>
<td>2.</td>
<td>PNZ</td>
<td>POP</td>
<td></td>
<td>(5)</td>
</tr>
<tr>
<td>3.</td>
<td>:=</td>
<td>TCM</td>
<td></td>
<td>5</td>
</tr>
<tr>
<td>4.</td>
<td>B</td>
<td></td>
<td>Bill</td>
<td>(10)</td>
</tr>
<tr>
<td>5.</td>
<td>SUB</td>
<td>Bill</td>
<td></td>
<td>4</td>
</tr>
<tr>
<td>6.</td>
<td>PNZ</td>
<td>POP</td>
<td></td>
<td>(9)</td>
</tr>
<tr>
<td>7.</td>
<td>:=</td>
<td>TCM</td>
<td></td>
<td>6</td>
</tr>
<tr>
<td>8.</td>
<td>B</td>
<td></td>
<td>TCM</td>
<td>(16)</td>
</tr>
<tr>
<td>9.</td>
<td>:=</td>
<td>TCM</td>
<td></td>
<td>1</td>
</tr>
</tbody>
</table>

To form the above triples, all relational expressions are subtracted. The execution phase will push the interim result onto a stack which may be subsequently retrieved with the FOP command. Immediately following the relational triple is a conditional branch to the next assignment clause to be considered should the relational expression be false. After each assignment clause is an unconditional branch to the end of the full assignment clause, since only one of the assignments is to be executed.

When a full assignment clause is to be processed, the appropriate tokens will be passed by an array parameter to subroutine CTRIPS, the control module for creating triples. The flow chart for CTRIPS appears in Figure 2. CTRIPS checks
Figure 2. Flow Chart For CTRIFS
for an IF-token and if found calls subroutines RELATE and POSTFX. On return from POSTFX, another check is made for an IF-token which exists for nested IF-THEN-ELSE. If found, the program loops back until the ELSE-token is encountered which will generate a call of POSTFX to process the ELSE assignment clause. Upon the final return from POSTFX a check is made to verify the requirement for internal addresses within the triples. This requirement exists whenever the full assignment clause contains the IF-THEN-ELSE tokens, that is anytime it consists of more than a single, simple assignment clause. The needed address is the address of the end of the triples and will become the operand for the unconditional branch at the end of each set of triples representing a simple assignment clause. The triples are then written out if desired for maintenance or debugging purposes and the subroutine returns to the calling module.

Subroutine RELATE processes the relational expression following the IF-token and places it in the ordered triples. The flow chart is illustrated in Figure 3. Relational expressions are limited to the following components:

1. Optional string of unary pluses and/or minuses;
2. Constant or variable identifier;
3. Relational operator;
4. Optional string of unary pluses and/or minuses;
5. Constant or variable identifier.
Figure 3. Flow Chart For RELATE
Unary pluses are disregarded since they have no effect. Unary minuses will negate the constant or variable token on which they operate. (The processing of unary pluses and minuses is a minor detail which does not appear in any of the flow charts for sake of simplicity.) The operands of the relational operator (variables and/or constants) are placed in the second and third columns of the triple array. An internal code for subtract is placed in the first column of the triples as the operator.

A branch condition based on the specific relational operator is placed in the next triple. The address of this triple is saved for inserting the object of the branch at a later time when it becomes known. This occurs in subroutine BLDTRP. The purpose here is to generate a branch to the end of the THEN assignment clause should the relational expression be false. The address of the end of the THEN-clause is not known until BLDTRP completes building the triples for the THEN assignment clause.

Subroutine POSTFX receives the string of tokens representing a simple assignment clause such as

\[ A := B \times (C + D ** E) - \text{Sqrt}(F). \]

With this input, POSTFX transforms the token string into postfix notation which would be

\[ A, B, C, D, E, **, +, *, F, \text{Sqrt}, -. \]

BLDTRP is then called to build the ordered triples from the
simple assignment clause. The flow chart in Figure 4 illustrates the basic algorithm used.

Input tokens are parsed one at a time and, depending on what they represent, either moved directly to the output stream or pushed onto a last-in-first-out (LIFO) stack for an interim period. When the "end" token is encountered, any tokens remaining on the stack are moved to the output stream. Since the order of variables and constants as read from left to right is the same for both infix and postfix notation, they are always moved directly to the output stream. The order of operators is changed as necessary to obtain the proper sequence of execution. The standard precedence of operators from highest to lowest is:

- Functions
- Unary plus or minus (+, -)
- Exponentiation (**)
- Multiplication, division (*, /)
- Addition or subtraction (+, -)
- Assignment (:=)
- Right parenthesis
- Left parenthesis

A series of relative comparisons result in the proper sequence of operators. A left parenthesis is always pushed onto the stack. When the stack is empty the operator is pushed onto the stack. Plus and minus signs must be checked
Figure 4. Flow Chart For POSTFX
to determine if they are unary or binary operators. Unary
pluses are disregarded while each unary minus negates its
operand token. Remaining operator tokens are compared with
the token at the top of the stack. If the stack token is of
equal or higher precedence, it is output and another compar-
sion is made with the new top of the stack token. When the
input token is of higher precedence and a right parenthesis,
then the stack token must be a left parenthesis, so both are
discarded. In other cases when the input token is of higher
precedence it is pushed onto the stack and the next token is
fetched.

The simple assignment clause has four possible delimit-
ers to mark its end. A second IF-token indicates a nested
IF-THEN situation and a relational expression will follow.
An ELSE-token marks the end of the THEN assignment clause
which will be followed by an ELSE assignment clause. The
tokens NOSAVE and COMMENT are special tokens to denote the
end of the full assignment clause. When any of these delim-
itarors are encountered, the stack is moved to output and an
internal end delimiter is placed in the output stream.
POSTFX then calls BLDTP to convert the postfix assign-
ment clause into ordered triples.

Subroutine BLDTP builds triples from the postfix ex-
pression. Figure 5 is the flow chart for BLDTP. Using the
while-do construct, the underlying logic is "while the token

Figure 5. Flow Chart For BLDRP
is not the internal end delimiter, to place the operators and operands into the proper position in the array of ordered triples".

The postfix token string is parsed until an operator is found. That operator and the preceding two operands (one operand for function operators which are unary operators) are placed into the triple array. After being inserted in the triple, a zero replaces operator tokens in the postfix string. This step will prevent any attempt to use the same operator twice. It should also be noted that the interim result of a triple (an operator and its respective operands) may be the operand for a subsequent triple. Thus the right operand in the postfix string is replaced with a POP command to denote the interim result of a preceding triple is to be popped from a stack. (During execution, interim triple results will be automatically pushed onto a stack for later use.) After the second operand is placed in the triple a zero is placed in its position in the postfix string to preclude that operand from being used a second time.

The subroutine then finalizes some addresses within the ordered triples. The conditional branch for a false relational expression (false IF condition) is given the address of the present triple plus two (the beginning of the ELSE-clause). The next triple is given an unconditional branch to the end of the full assignment clause. As only one assign-
ment clause is to be executed, each clause is followed with
a branch to the end of the last assignment clause. Because
that address is not known yet, the address of the end of each
simple assignment clause is retained and the final "end"
address is inserted in each in a wrap-up procedure in CTRIPS.

Boolean Expression

The Boolean expression, as discussed in chapter 2,
consists of a series of relational expressions which are the
operands of logical operators. They have the same possible
syntax as the full assignment clause relational expressions
presented earlier. The tokens IN and NOT IN and their group
identifier are also possible operands, indicating data is to
be either selected from the specified group of columns of
the database or selected from columns not within the speci-
fied group. Parentheses may also be used to modify the
precedence of operators or to improve understandability of
the expression. An example of a proper Boolean expression
might be

```
IN G1, .AND., T1 = 3, .AND., (T2 <= 2, .OR., T2 > 5).
```

After transforming this expression into postfix notation
it would appear as

```
IN G1, T1 = 3, .AND., T2 <= 2, T2 > 5, .OR., .AND..
```

The string of tokens in postfix order must now be converted
into ordered triples for efficient execution. The desired
ordered triples for the preceding expression are as follows:

<table>
<thead>
<tr>
<th>Row</th>
<th>Operator</th>
<th>Operand 1</th>
<th>Operand 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>IN</td>
<td>71</td>
<td>71</td>
</tr>
<tr>
<td>2</td>
<td>SUB</td>
<td>71</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>BNZ</td>
<td>POP</td>
<td>(6)</td>
</tr>
<tr>
<td>4</td>
<td>ADC</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>E</td>
<td>0</td>
<td>(7)</td>
</tr>
<tr>
<td>6</td>
<td>ADC</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>AND</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>SUB</td>
<td>T2</td>
<td>2</td>
</tr>
<tr>
<td>9</td>
<td>EP</td>
<td>POP</td>
<td>(12)</td>
</tr>
<tr>
<td>10</td>
<td>ADD</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>11</td>
<td>E</td>
<td>0</td>
<td>(14)</td>
</tr>
<tr>
<td>12</td>
<td>ADD</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>13</td>
<td>SUB</td>
<td>T2</td>
<td>5</td>
</tr>
<tr>
<td>14</td>
<td>BMZ</td>
<td>POP</td>
<td>(17)</td>
</tr>
<tr>
<td>15</td>
<td>ADD</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>16</td>
<td>E</td>
<td>0</td>
<td>(19)</td>
</tr>
<tr>
<td>17</td>
<td>ADD</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>18</td>
<td>CR</td>
<td></td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>AND</td>
<td></td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>SETMSK</td>
<td>POP</td>
<td>SELECT</td>
</tr>
</tbody>
</table>

During execution of the above triples the relational evaluations and the conditional and unconditional branches function identical to those explained previously with the full assignment clause. The IN group operand will cause a one to be pushed onto a stack if the group exists, otherwise a zero will be pushed. The "ADD 0 1" triple is executed if the relational expression is true and will cause a one to be pushed onto the stack. For a false relational expression, the "ADD 0 0" triple is executed, causing a zero to be pushed onto the stack. The AND triple will pop the stack twice and if there are two ones, a one is pushed, otherwise a zero is pushed. Similarly, CR pops the stack twice and pushes a one if at least one of the pops is a one. Row 20
sets a mask or flag to one if the Boolean expression is true, otherwise a zero is set.

Obviously there are many similarities between building ordered triples for Boolean and arithmetic expressions. However sufficient differences exist to warrant separate routines. To preclude excessive repetition, the description of the full assignment clause modules will be used as a base line and only the significant differences for Boolean expressions will be explained in this section.

When a Boolean expression is to be translated, the main program will call subroutine POSTFX, a six letter name for "postfix select". The flow chart for this subroutine is the same as that of POSTFX (Figure 4) with minor exceptions of delimiters, operators, and operands. The tokens NOTAVE and REMARK are the only delimiters for the Boolean expression. The precedence of possible operators from highest to lowest is:

\[ \text{AND} \]
\[ \text{OR} \]
Right parenthesis
Left parenthesis.

The operands for the Boolean expression are more than one token in length. Consequently the routine recognizes each portion of an operand (unary plus or minus, constant, variable, relational operator, IN, NOT, and group) and moves them directly to the output string. Once the expression is
converted to postfix notation, a recognizable and limited
is placed in the string and subroutine TRIPS is called.

Subroutine TRIPS (Figure 6) accepts the Boolean expres-
sion in postfix notation and arranges the tokens into condi-
tional triples. Again there are similarities, yet differences, when
comparing subroutines TRIPS and BLEEPS.

The postfix string is parsed and the tokens identified.
Logical operators (.AND., .OR.) are moved directly to the
triple and the next token is fetched. The operands IN and
NOT IN along with the group identifiers are placed in the
triples. No branch conditions are required since the execu-
tion phase will determine whether these operands are true or
false, pushing a one or zero as appropriate onto the stack.
The only other token type is a logical operand of three
tokens which form a relational expression. The subtrac-
tion will be placed in the triple followed by the two
relational operands. The next triple will get a conditional
branch based on a false relational expression. A one or
zero, depending on a true or false relational expression, is
pushed onto the stack. Finally the triples must provide for
evaluating the entire Boolean expression is evaluated. To
do this the stack is popped. A one indicates a true, a zero
a false expression. A flag (STEXK) is set to reflect this
condition. Now that the triples are completed, TRIPS
returns to PSTFXS which returns to its calling module.
Figure 6. Flow Chart for TRIPLS
At this point, the routines to transform the full assignment clause and Boolean expressions into ordered triples have been fully discussed. The next chapter, chapter 5, will conclude this paper.
CHAPTER V

CONCLUSION

This project has been successful in developing the desired programs. Several varying sets of data have been processed and the desired results have been obtained. It should be emphasized, however, that the programs are quite sensitive to receiving syntactically correct input. It was assumed in the specifications that the tokens would be edited and checked for correct syntactic order before being processed into ordered triples. Consequently, these programs have a very low tolerance to erroneous input.

FORTRAN was specified as the programming language in order to enhance transportability. ANSI FORTRAN is generally considered to be an unstructured language since its only means of changing the order of execution of statements from sequential is with GOTOs and the DO-loop. However, it was possible to provide a perception of structure to the programs. This was accomplished by a carefully structured use of comments and precise indentation of comments and executable statements. The resulting code is more understandable than most FORTRAN programs.

The most difficult aspect of the project was defining
and understanding the specifications and requirements of the project. Nearly all of the communication was oral which may be subject to different interpretation and may often be forgotten or take on different meanings after a period of time. Further, the author was not involved in the overall COCAB project, thus initially did not have the broad perspective of the COCAB system. With attention focused on only the problem at hand, it was often more difficult to understand the need and reason for certain specifications. Finally, the specifications were not precisely available at the start of the project. Specifications are normally developed or modified as the project progresses, as was true in this situation. As experienced in probably all software development projects, the specifications evolve and change as time passes, the users think of other requirements and peculiarities, and the designers perceive new and expanding capabilities. Real world experiences encountered in this project will certainly be of significant value to the author.

Future Efforts

Although the subroutines developed function according to the specifications, additional improvements could still be added. Separate subroutines were developed for transforming the input tokens into postfix notation, then
into ordered triples. The separate subroutines were caused
to break the overall problem down into smaller, more
manageable problems and to make them small enough to be
easily understood. This does, however, increase the
computer overhead devoted to linking the various subroutines
together. Algorithms exist (1,6) to convert an infix token
string directly into ordered triples. A comparison between
the length, complexity, and execution efficiency of the two
methods may be enlightening.

A second area for future study concerns the ordered
triples constructed from the Boolean expression. The
Boolean expression is composed of a series of logical
operators (.AND., .OR.) and the associated operands. As
developed, the entire set of triples must be executed to
determine the result of the expression. However, for
certain sequences of operators, it can be determined that
the expression is false before the entire expression is
evaluated. This would be desirable since there is no need
to evaluate the remainder of the expression once the final
results have already been determined. Being able to stop
the expression evaluation once its results have been
determined would save execution time. But developing the
algorithm to insert the proper branches into the triples is
not an easy problem, considering the various possible
sequences of operators and how parentheses are used to
change the the normal precedence of execution. Such an optimizing step would likely require the tokens to be scanned several times while developing the possible paths through the Boolean expression. Due to the complexity, such an optimizing effort could likely result in errors as are sometimes encountered with commercial optimizing compilers.
APPENDICES
APPENDIX A

SYNTAX GRAPHS
APPENDIX 2

USEP'S GUIDE
USER'S GUIDE

The modules are designed to accept an input array of 100 tokens. If it is determined that longer commands will be used, the arrays INPUT and OUTPUT will need to be dimensioned to a larger size in all subroutines. Additionally, the matrix which returns the ordered triples, TRIPLE, may need to be enlarged. Its present size is 80 by 2.

A stack is used in the subroutines POSTF and POSTF0 to temporarily hold operators while they are being reordered from infix to postfix notation. The stack size is 30. It is conceivable, although unlikely, for very long strings of input tokens and with certain arrangements of operators, operands, and parentheses, for the stack to overflow. If this should occur, the stack size should be increased.

PRIVLVL is an integer parameter which permits printing out interim data for maintenance or debugging purposes. For a value of zero, the postfix string of tokens and the completed ordered triples are printed.

The tokens must be in proper syntactic order when passed to these subroutines. The specifications were for other modules to do the editing and syntax checking. The designed subroutines have little fault tolerance for improper tokens or incorrect token order.
Each token string must have an acceptable delimiter to denote the end of the string. Appropriate delimiters are NOSAVE or REMARK.

The tokens applicable to the programs described are:

10000-19999 comment
20000-29999 constants

Relational Operators
30001 .EQ.
30002 .NE.
30003 .GT.
30004 .LT.
30005 .GE.
30006 .LE.

Logical Operators
40001 .AND.
40002 .OR.

Mathematical Operators
50001 left parentheses '('
50002 right parentheses ')
50003 plus '+'
50004 minus '-'
50005 divide '/'
50006 multiply '*'
50007 exponentiation '**'
50010 assign ':='

Functions
60002 log
60009 square root

Variables
70000-19999 and 150000 or larger

Specific Identifiers
142805 IF
142806 IN
142807 NOT
142808 THEN
142809 ELSE
144002 NOSAVE
144008 REMARK
<table>
<thead>
<tr>
<th>Code</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>50011</td>
<td>BNZ branch on not zero</td>
</tr>
<tr>
<td>50012</td>
<td>BZ branch on zero</td>
</tr>
<tr>
<td>50013</td>
<td>BMZ branch on minus or zero</td>
</tr>
<tr>
<td>50014</td>
<td>BPZ branch on plus or zero</td>
</tr>
<tr>
<td>50015</td>
<td>BM branch on minus</td>
</tr>
<tr>
<td>50016</td>
<td>BP branch on plus</td>
</tr>
<tr>
<td>50017</td>
<td>B unconditional branch</td>
</tr>
<tr>
<td>50018</td>
<td>ADD push onto stack</td>
</tr>
<tr>
<td>50019</td>
<td>POP pop from stack</td>
</tr>
<tr>
<td>50020</td>
<td>SETMSK set the mask</td>
</tr>
<tr>
<td>50021</td>
<td>.AND. logical and</td>
</tr>
<tr>
<td>50022</td>
<td>.OR. logical or</td>
</tr>
<tr>
<td>50023</td>
<td>IN in group or modulo</td>
</tr>
<tr>
<td>50024</td>
<td>NOT IN not in group or modulo</td>
</tr>
<tr>
<td>50025</td>
<td>CREATE create new data</td>
</tr>
<tr>
<td>50026</td>
<td>SELECT select from data base</td>
</tr>
</tbody>
</table>
APPENDIX C

PROGRAM LISTINGS
SUBROUTINE CTRIPS (INPUT, PRTLVL, TRIPLE)

C XXXX XXXX XXXX XXX XXXX XXXX
C X   X   X   X   X   X   X
C X   X   XXXX X   XXXX XXXX
C X   X   X   X   X   X   X
C XXXX X   X   XXXX X   XXXX
C

C * * * * * * FUNCTION OF MODULE * * * * * *

CTRIPS IS THE CONTROL MODULE WHICH ACCEPTS A STRING OF TOKENS
WHICH FORM THE FULL ASSIGNMENT CLAUSE AND CALLS THE MODULES
WHICH CONVERT THE STRING INTO ORDERED TRIPLES FOR EXECUTION.

C * * * * * * * PROTOCOL FOLLOWED * * * * * *

THE FIRST TOKEN IS CHECKED TO DETERMINE WHAT IT IS. FOR AN 'IF'
TOKEN, THE RELATE SUBROUTINE IS CALLED TO PLACE THE TOKENS OF THE
RELATIONAL EXPRESSION INTO THE ORDERED TRIPLE ARRAY. FOLLOWING
THE RETURN FROM RELATE (OR IMMEDIATELY IF THERE IS NO RELATIONAL
EXPRESSION) POSTFX SUBROUTINE IS CALLED TO REORDER THE INFINX
ASSIGNMENT CLAUSE INTO A POSTFIX ASSIGNMENT CLAUSE.
POSTFX CALLS BLDTRP WHICH PLACES THE POSTFIX ASSIGNMENT CLAUSE
INTO THE ORDERED TRIPLE ARRAY BEFORE RETURNING TO THIS MODULE.
REPEATED CALLS OF THE RELATE AND POSTFX SUBROUTINES MAY BE MADE
TO HANDLE NUMEROUS RELATIONAL AND ASSIGNMENT CLAUSES.

C * * * * * * SUBROUTINES CALLED * * * * * *

RELATE-MOVES TOKENS OF RELATIONAL EXPRESSIONS TO THE TRIPLE ARRAY
POSTFX-CHANGES TOKENS OF ASSIGNMENT CLAUSES INTO POSTFIX
ORDER; THEN CALLS BLDTRP WHICH PLACES THOSE TOKENS INTO
THE TRIPLE ARRAY

C * * * * * * VARIABLE DECLARATIONS * * * * * *

PARAMETERS

INPUT - INTEGER ARRAY, LENGTH 100, FOR PASSING TOKENS
PRTLVL-INTEGER, VARIABLE TO ALLOW PRINTING OF EXTRA INFORMATION
WHICH SHOULD BE HELPFUL IN MODIFYING OR DEBUGGING THE
SYSTEM
TRIPLE-INTEGER ARRAY, SIZE 80 BY 3, FOR RETURNING ORDERED TRIPLES

INTEGER INPUT(100), PRTLVL, TRIPLE(80,3)
51

C LOCAL VARIABLES

C BRADDR-INTEGER ARRAY, LENGTH 20, HOLDS INTERNAL ADDRESS OF TRIPLE ROWS WHICH WILL GET UNCONDITIONAL BRANCH TO END OF TRIPLES

C INPNT -INTEGER, POINTS TO TOKENS IN THE INPUT VECTOR

C NUMBX -INTEGER, COUNTS THE NUMBER OF PLACES FOR UNCONDITIONAL BRANCH TO END OF TRIPLES, INCREMENTS THE BRADDR ARRAY

C OUTPNT-INTEGER, POINTS TO LAST TOKEN IN OUTPUT VECTOR

C OUTPUT-INTEGER ARRAY, LENGTH 100, FOR PASSING POSTFIX ASSIGNMENT CLAUSE

C REMBER-INTEGER, SAVES INTERNAL ADDRESS OF TRIPLE ARRAY TO INSERT ADDRESS FROM BLDRP SUBROUTINE

C TRPROU-INTEGER, POINTER TO ROWS OF THE ARRAY OF ORDERED TRIPLES

C INPNT, NUMBX, OUTPNT, OUTPUT(100), REMBER, TRPROU

C * * * * * * EXECUTABLE CODE * * * * * * *

C

C INPNT = 1

C NUMBX = 0

C REMBER = 0

C TRPROW = 0

C IF THE FIRST TOKEN IS 'IF' THEN CALL SUBROUTINE RELATE

C IF (INPUT(INPNT).NE.140805) GO TO 80

C CALL RELATE (INPNT, INPUT, REMBER, TRIPLE, TRPROW)

C END OF IF

C CALL POSTFX (BRADDR, NUMBX, INPNT, INPUT, OUTPUT, PRTLVL, & REMBER, TRIPLE, TRPROU)

C IF THE NEXT TOKEN IS 'IF' (NESTED IF-THEN CLAUSE)

C THEN CALL SUBROUTINE RELATE

C IF (INPUT(INPNT).EQ.140805) GO TO 70

C END OF IF

C CALL POSTFX (BRADDR, NUMBX, INPNT, INPUT, OUTPUT, PRTLVL, & REMBER, TRIPLE, TRPROU)

C IF INTERNAL TRIPLE ADDRESSES ARE INCOMPLETE

C IF (NUMBX.EQ.0) GO TO 100

C REPEAT

C PLACE THE SAVED TRIPLE ROW ADDRESS IN THE PROPER PLACES AS THE OBJECTS OF THE UNCONDITIONAL BRANCHES.

C DO 90 K = 1, NUMBX

C TRIPLE(BRADDR(K),3) = TRPROW + 1

C END OF REPEAT

C END OF IF

C IF PRTLVL = 0 WRITE OUT THE ORDERED TRIPLES

C IF (PRTLVL.NE.0) GO TO 600

C N = TRPROW
DO 110 TRPROW = 1, N
110     WRITE (6,510) (TRIPLE(TRPROW,J), J = 1, 3)
C     END OF IF
510     FORMAT (' ', T10, I7, T25, I7, T40, I7)
600     RETURN
END
C$EJECT
SUBROUTINE RELATE (INPNT, INPUT, REMBER, TRIPLE, TRPROW)

C XXXX XXXXX X X XXXX XXXX
C X X X X X X X X
C XXXX XXXX X X X X XXXX
C X X X X X XXXX X X
C X X XXXX XXXXX X X X XXXX

C

C * * * * * * FUNCTION OF MODULE * * * * * * *

C RELATE SCANS THE INPUT STACK OF TOKENS AND CHANGES THOSE IN A
C RELATIONAL EXPRESSION INTO ORDERED TRIPLES WHICH ARE OUTPUT IN
C AN ARRAY.

C

C * * * * * * PROCEDURE FOLLOWED * * * * * * *

C

C INPUT RELATIONAL EXPRESSIONS ARE RESTRICTED TO THE FOLLOWING
C FORMAT: 1. OPTIONAL STRING OF UNARY PLUSES AND/OR MINUSES;
C 2. CONSTANT OR VARIABLE IDENTIFIER;
C 3. RELATIONAL OPERATOR;
C 4. OPTIONAL STRING OF UNARY PLUSES AND/OR MINUSES;
C 5. CONSTANT OR VARIABLE IDENTIFIER.
C UNARY PLUSES ARE DISREGARDED. UNARY MINUSES ARE ACCUMULATED
C AND, IF THERE ARE AN ODD NUMBER OF THEM, THE CONSTANT OR VARIABLE
C TOKEN IS CHANGED TO A NEGATIVE VALUE. THE FIRST CONSTANT OR
C VARIABLE IS PLACED IN THE SECOND COLUMN OF THE TRIPLE ARRAY,
C THE RELATIONAL OPERATOR IN THE FIRST COLUMN, AND THE SECOND
C CONSTANT OR VARIABLE IN THE THIRD COLUMN.

C

C * * * * * * VARIABLE DECLARATIONS * * * * * * *

C

C PARAMETERS

C

C INPNT -INTEGER, POINTS TO TOKENS IN THE INPUT VECTOR
C INPUT -INTEGER ARRAY, LENGTH 100, FOR PASSING TOKENS TO THIS
C MODULE
C REMBER-INTEGER, SAVES INTERNAL ADDRESS OF TRIPLE ARRAY TO INSERT
C ADDRESS FROM BLDTRP SUBROUTINE
C TRIPLE-INTEGER ARRAY, SIZE 80 BY 3, FOR RETURNING ORDERED TRIPLES
C TRPROW-INTEGER, POINT TO ROWS OF THE ARRAY OF ORDERED TRIPLES
C
C INTEGER INPNT, INPUT(100), REMBER, TRIPLE(80,3), TRPROW

C

C LOCAL VARIABLES

C

C MINUS -INTEGER, UNARY MINUS OPERATOR TOKEN 50006
47. C PLUS -INTEGER, UNARY PLUS OPERATOR TOKEN 50005
48. C POP -INTEGER, OPERATOR WHICH MAY BE INSERTED INTO THE ORDERED
49. C TRIPLE ARRAY FOR POPPING RESULT FROM A STACK
50. C RELOPS-INTEGER ARRAY, SIZE 2 BY 6, HOLDS RELATIONAL OPERATOR AND
51. C BRANCH CODES. THE OPERATOR CODES INSERTED INTO THE ORDERED
52. C TRIPLE ARRAY ARE:
53. C INPUT RELATIONAL BRANCH BRANCH
54. C TOKEN OPERATOR CONDITION CODE
55. C 30001 .EQ. B NZ 50011
56. C 30002 .NE. BZ 50012
57. C 30003 .GT. BNZ 50013
58. C 30004 .LT. BPZ 50014
59. C 30005 .GE. BM 50015
60. C 30006 .LE. BP 50016
61. C RELPNT-INTEGER, POINT TO ENTRIES IN RELOPS ARRAY
62. C TEMP -INTEGER, USED FOR DETERMINING IF TOKEN IS A VARIABLE OR
63. C CONSTANT
64. C THEN -INTEGER, POSSIBLE INPUT TOKEN 142408
65. C UMINUS-INTEGER, UNARY MINUS MULTIPLYING FACTOR
66. C INTEGER MINUS, PLUS, POP, RELOPS(2,6), RELPNT, SUB, TEMP, THEN,
67. & UMINUS
68. C DATA MINUS/50006/, PLUS/50005/, POP/50019/, RELOPS/30001, 50011,
69. & 30002, 50012, 30003, 50013, 30004, 50014, 30005, 50015,
70. & 30006, 50016/, SUB/50006/, THEN/142408/
71. C
72. C
73. C ******** EXECUTABLE CODE ********
74. C
75. C UMINUS = +1
76. C INPNT = INPNT - 1
77. C TRPROW = TRPROW + 1
78. C
79. C 10 CONTINUE
80. C INCREMENT THE INPUT ARRAY POINTER
81. C INPNT = INPNT + 1
82. C IF THE TOKEN IS UNARY PLUS DISREGARD AND GET THE NEXT TOKEN
83. & IF (INPUT(INPNT).EO.PLUS) GO TO 10
84. C END OF IF
85. C
86. C IF TOKEN IS UNARY MINUS CHANGE THE SIGN OF THE UNARY MINUS FACTOR
87. & IF (INPUT(INPNT).NE.MINUS) GO TO 40
88. & UMINUS = - UMINUS
89. C AND GET THE NEXT TOKEN
90. C GO TO 10
91. C END OF IF
92. C
93. C 40 CONTINUE
94. C IF TOKEN IS A CONSTANT OR VARIABLE
95. C TEMP = INPUT(INPNT)/10000
96. C IF (TEMP.NE.2).AND.((TEMP.LT.7).OR.(TEMP.GT.9)).AND.
97. & (TEMP.LT.13)) GO TO 400
98. C THEN MULTIPLY BY UNARY MINUS FACTOR AND MOVE THE TOKEN TO
99. C COLUMN TWO OF THE TRIPLE ARRAY
100. TRIPLE(TRPROW,2) = INPUT(INPNT) * UMINUS
101. C END OF IF
102. C
103. C RESET UNARY MINUS FACTOR TO POSITIVE
104. UMINUS = +1
105. C GET THE NEXT TOKEN
106. INPNT = INPNT + 1
107. C PLACE SUBTRACT CODE IN COLUMN ONE OF TRIPLE ARRAY
108. TRIPLE(TRPROW,1) = SUB
109. C
110. C OBTAIN THE PROPER CODE FOR THE RELATIONAL OPERATOR
111. C REPEAT
112. C COMPARE INPUT TOKEN WITH RELATIONAL OPERATOR ARRAY
113. DO 50 RELPNT = 1,6
114. IF (INPUT(INPNT).EQ.RELOPS(1,RELPNT)) GO TO 60
115. 50 CONTINUE
116. C END OF REPEAT
117. C
118. 60 CONTINUE
119. C PLACE THE PROPER BRANCH CODE, BASED ON THE RELATIONAL OPERATOR,
120. C IN COLUMN ONE OF THE NEXT TRIPLE ROW
121. TRIPLE(TRPROW + 1, 1) = RELOPS(2,RELPNT)
122. C PLACE 'POP' OPERATOR IN COLUMN TWO OF THE TRIPLE ARRAY
123. TRIPLE(TRPROW + 1, 2) = POP
124. C RETAIN THE TRIPLE ROW NUMBER FOR LATER INSERTION OF BRANCH ADDRESS
125. RENDER = TRPROU + 1
126. 70 CONTINUE
127. INPNT = INPNT + 1
128. C IF THE TOKEN IS UNARY PLUS DISREGARD AND GET THE NEXT TOKEN
129. IF (INPUT(INPNT).EQ.PLUS) GO TO 70
130. C END OF IF
131. C
132. C IF TOKEN IS UNARY MINUS CHANGE THE SIGN OF THE UNARY MINUS FACTOR
133. IF (INPUT(INPNT).NE.MINUS) GO TO 90
134. UMINUS = - UMINUS
135. C AND GET THE NEXT TOKEN
136. GO TO 70
137. C END OF IF
138. C
139. 90 CONTINUE
140. C IF TOKEN IS A CONSTANT OR VARIABLE
141. TEMP = INPUT(INPNT)/10000
142. IF ((TEMP.NE.2).AND.((TEMP.LT.7).OR.(TEMP.GT.9)).AND.
143. .AND. (TEMP.LT.15)) GO TO 400
144. C THEN MULTIPLY BY UNARY MINUS FACTOR AND MOVE THE TOKEN TO
145. C COLUMN THREE OF THE TRIPLE ARRAY
146. TRIPLE(TRPROW,3) = INPUT(INPNT) * UMINUS
147. INPNT = INPNT + 1
148. GO TO 500
149. C END OF IF
150. C
151. 400 CONTINUE
152. C PRINT ERROR MESSAGE
153. PRINT, 'ERROR--TOKEN NOT A CONSTANT OR VARIABLE. TOKEN = ',
154. & INPUT(INPNT)
155. TRPROW = TRPROW + 1
156. INPNT = INPNT + 1
157. 500 CONTINUE
158. TRPROW = TRPROW + 1
159. RETURN
160. END
161. C\EJECT
SUBROUTINE POSTFX (BRADDR, NUMBX, INPNT, INPUT, OUTPUT, PRTLVL, REMBER, TRIPLE, TRPROW)

C XXXXX XXXXX XXXXX XXXXX X X
C X X X X X X X X X X
C XXXX X X XXXX X XXXX X
C X XXXX XXXXX X X X X
C
C ***** FUNCTION OF MODULE *****
C
POSTFX TAKES AN INFIX ASSIGNMENT CLAUSE AS INPUT AND CONVERTS IT INTO AN EXPRESSION OF POSTFIX NOTATION WHICH IS OUTPUT.
C
PROCEDURE FOLLOWED 

IF THE TOKEN IS AN END DELIMITER OR A REMARK, THE TOKENS IN THE STACK ARE OUTPUT UNTIL THE STACK IS EMPTY. IF THE TOKEN IS '(', IT IS PUSHED ONTO THE STACK. IF THE TOKEN IS A MINUS SIGN A CHECK IS MADE TO DETERMINE IF IT IS A UNARY MINUS. IF IT IS A UNARY MINUS, THE CONSTANT OR VARIABLE TOKEN ON WHICH IT OPERATES IS CHANGED TO A NEGATIVE VALUE.
C
C
* * * SUBROUTINES CALLED * * * * * * * * 
C
BLDTRP-MOVES POSTFIX-ORDERED TOKENS INTO ARRAY OF ORDERED TRIPLES 
C
* * * * * * * * * * * VARIABLE DECLARATIONS * * * * * * * * 
C
PARAMETERS 
C
BRADDR-INTEGER ARRAY, LENGTH 20, HOLDS INTERNAL ADDRESS OF TRIPLE ROWS WHICH WILL GET UNCONDITIONAL BRANCH TO END OF TRIPLES 
C
NUMBX -INTEGER, COUNTS THE NUMBER OF PLACES FOR UNCONDITIONAL BRANCH TO END OF TRIPLES, INCREMENTS THE BRADDR ARRAY
C INPNT -INTEGER, POINTS TO TOKENS IN THE INPUT VECTOR

C INPUT -INTEGER ARRAY, LENGTH 100, FOR PASSING TOKENS TO THIS

C MODULE

C OUTPUT-INTEGER ARRAY, LENGTH 100, FOR RETURNING POSTFIX

C ARITHMETIC EXPRESSION

C PRTLVL-INTEGER, VARIABLE TO ALLOW PRINTING OF EXTRA INFORMATION

C WHICH SHOULD BE HELPFUL IN MODIFYING OR DEBUGGING THE

C SYSTEM

C REMBR-INTEGER, SAVES INTERNAL ADDRESS OF TRIPLE ARRAY TO INSERT

C ADDRESS FROM BLDTRP SUBROUTINE

C TRIPLE-INTEGER ARRAY, SIZE 80 BY 3, FOR RETURNING ORDERED TRIPLES

C TRPROU-INTEGER, POINTER TO ROWS OF THE ARRAY OF ORDERED TRIPLES

C

C INTEGER BRADDR(20), INPNT, INPUT(100), NUMBR, OUTPUT(100),

C &

C PRTLVL, REMBR, TRIPLE(80,3), TRPROU

C

C LOCAL VARIABLES

C

C ARRPNT-INTEGER, POINTS TO ENTRIES IN THE PRECEDENCE ARRAY

C CPAREN-INTEGER, POSSIBLE INPUT TOKEN OF CLOSING PARENTHESIS

C ELSE -INTEGER, POSSIBLE INPUT TOKEN 142409

C MINUS -INTEGER, POSSIBLE INPUT TOKEN 50006

C NOSAVE-INTEGER, POSSIBLE INPUT TOKEN 144002

C OPAREN-INTEGER, POSSIBLE INPUT TOKEN OF OPENING PARENTHESIS

C OUTPNT-INTEGER, POINTS TO TOKENS IN OUTPUT VECTOR

C PLUS -INTEGER, POSSIBLE INPUT TOKEN 50005

C PREC -INTEGER ARRAY, SIZE 2 BY 18, HOLDS PRECEDENCE OF

C ARITHMETIC OPERATORS

C PRECS -INTEGER ARRAY, PRECEDENCE OF TOKEN AT TOP OF THE STACK

C PRECT -INTEGER, PRECEDENCE OF TOKEN CURRENTLY UNDER CONSIDERATION

C STACK-INTEGER ARRAY, LENGTH 30, STACK FOR PROCESSING OPERATORS

C STKPNT-INTEGER, POINTER TO NEXT EMPTY POSITION ON THE STACK

C TEMP -INTEGER, USED FOR DETERMINING IF TOKEN IS A VARIABLE OR

C CONSTANT

C THEN -INTEGER, POSSIBLE INPUT TOKEN 142408

C UMINUS-INTEGER, UNARY MINUS MULTIPLYING FACTOR

C

C INTEGER ARRPNT, CPAREN, ELSE, END, IF, MINUS, OPAREN, OUTPNT,

C &

C PLUS, PREC(2,18), PRECS, PRECT, STACK(30), STKPNT, TEMP,

C &

C THEN, UMINUS

C DATA CPAREN/50004/, ELSE/142409/, END/50030/, IF/140805/,

C &

C MINUS/50006/, NOSAVE/144002/, OPAREN/50003/, PLUS/50005/,

C &

C THEN/142408/

C DATA PREC/50003/, 3, 50004, 1, 50005, 3, 50006, 3, 50007, 4,

C &

C 50008, 4, 50009, 5, 50010, 2, 60001, 7, 60002, 7, 60003, 7,

C &

C 60004, 7, 60005, 7, 60006, 7, 60007, 7, 60008, 7, 60009, 7,

C &

C 60010, 7/

C

C EXECUTABLE CODE

C

C OUTPNT = 1
99.  STKPNT = 1
100.  UMINUS = +/-1
101.  C IF THE TOKEN IS "THEN" OR "ELSE" MOVE IT TO OUTPUT
102.     IF (((INPUT(INPNT),NE,THEN),AND,(INPUT(INPNT),NE,ELSE)) GO TO 10
103.     OUTPUT(OUTPNT) = INPUT(INPNT)
104.     OUTPNT = OUTPNT + 1
105.     GO TO 20
106.  C END OF IF
107.  C
108.  10 CONTINUE
109.     INPNT = INPNT - 1
110.  20 CONTINUE
111.  C INCREMENT THE INPUT ARRAY POINTER
112.     INPNT = INPNT + 1
113.  C
114.  C IF THE TOKEN DELIMITS THE ASSIGNMENT CLAUSE (NOSAVE, IF, ELSE,
115.     OR REMARK)
116.     IF (((INPUT(INPNT),NE,NOSAVE),AND,(INPUT(INPNT),NE,140805),AND,
117.       & (INPUT(INPNT),NE,ELSE),AND,((INPUT(INPNT)/10000),NE,1)) GO TO 40
118.     C THEN
119.     C WHILE THE STACK IS NOT EMPTY
120.     30 CONTINUE,
121.     IF (STKPNT.EQ.1) GO TO 50
122.     STKPNT = STKPNT - 1
123.     C MOVE STACK TO OUTPUT
124.     OUTPUT(OUTPNT) = STACK(STKPNT)
125.     OUTPNT = OUTPNT + 1
126.     GO TO 30
127.     C END OF WHILE
128.  C END OF IF.
129.  C
130.  40 CONTINUE
131.     C IF THE TOKEN = ’(‘,
132.     C IF ((INPUT(INPNT),NE,OPAREN) GO TO 50
133.     C THEN PUSH TOKEN ONTO STACK
134.     C STACK(STKPNT) = INPUT(INPNT)
135.     C STKPNT = STKPNT + 1
136.     C GO TO 20
137.     C END OF IF.
138.  C
139.  50 CONTINUE
140.     C IF TOKEN IS A CONSTANT OR VARIABLE
141.             TEMP = INPUT(INPNT)/10000
142.             IF (((TEMP,NE,2),AND,(TEMP,LT,7),OR,(TEMP,GT,9))
143.                  & (TEMP,LT,15)) GO TO 60
144.     C THEN MULTIPLY BY UNARY MINUS FACTOR AND MOVE TOKEN TO OUTPUT
145.             OUTPUT(OUTPNT) = INPUT(INPNT)*UMINUS
146.     C END OF IF.
147.  C
148.  C
149.  C
60
150. C END OF IF.
151. C
152. 60 CONTINUE
153. C IF TOKEN IS '+' OR '-'
155. C THEN IF THIS IS THE FIRST TOKEN OF THIS ASSIGNMENT CLAUSE
156. C THEN TOKEN IS A UNARY MINUS OR UNARY PLUS
157. C IF (INPNT.EQ.1) GO TO 70
158. C END OF IF.
159. C ELSE IF PREVIOUS TOKEN IS A FUNCTION OR OPERATOR BUT NOT ')
160. C THEN TOKEN IS A UNARY MINUS OR UNARY PLUS
161. C TEMP = INPUT(INPNT - 1)/10000
162. C IF (((TEMP.NE.5).AND.(TEMP.NE.6)).OR.
163. C (INPUT(INPNT-1).EQ.CPAREN)) GO TO 80
164. C END OF IF.
165. C END OF IF.
166. 70 CONTINUE
167. C
168. C IF TOKEN IS UNARY PLUS DISREGARD AND GET NEXT TOKEN
169. C IF (INPUT(INPNT).EQ.PLUS) GO TO 20
170. C END OF IF
171. C IF TOKEN IS UNARY MINUS CHANGE SIGN OF UNARY MINUS FACTOR
172. C UMINUS = -UMINUS
173. C RETURN FOR NEXT TOKEN
174. C GO TO 20
175. C END OF IF.
176. C
177. 80 CONTINUE
178. C IF STACK IS EMPTY
179. C IF (STKPNT.NE.1) GO TO 90
180. C THEN PUSH TOKEN ONTO STACK
181. C STACK(STKPNT) = INPUT(INPNT)
182. C STKPNT = STKPNT + 1
183. C GO TO 20
184. C END OF IF.
185. C
186. 90 CONTINUE
187. C ASSIGN PRECEDENCE OF OPERATOR AT TOP OF STACK TO 'PRECS'
188. C REPEAT
189. C COMPARE TOKEN AT TOP OF STACK WITH PRECEDENCE ARRAY
190. C DO 100 ARRPNT = 1,18
191. C IF (STACK(STKPNT - 1).EQ.PREC(1,ARRPNT)) GO TO 110
192. C END OF REPEAT.
193. 110 PRECS = PREC(2,ARRPNT)
194. C
195. C ASSIGN PRECEDENCE OF OPERATOR CURRENTLY UNDER CONSIDERATION TO
196. C 'PRECT'
197. C REPEAT
198. C COMPARE INPUT TOKEN WITH PRECEDENCE ARRAY
199. C DO 120 ARRPNT = 1,18
200. 120 IF (INPUT(INPNT).EQ.PREC(1,ARRPNT)) GO TO 130
C  END OF REPEAT.
130  PRECT = PRFC(2, ARRPNT)
C
C  COMPARE PRECEDENCE OF TOKEN AT TOP OF STACK WITH INPUT TOKEN
205.  C  IF PRECS > OR = PRECT
206.  IF (PRECS.LT.PRECT) GO TO 140
207.  C  THEN MOVE TOP OF STACK OPERATOR TO OUTPUT AND COMPARE TOKEN
208.  C  OPERATOR TO NEXT OPERATOR IN STACK.

STKPNT = STKPNT - 1
OUTPUT(OUTPNT) = STACK(STKPNT)
OUTPNT = OUTPNT + 1
GO TO 80

140  CONTINUE

C  ELSE IF TOKEN = '
C  IF (INPUT(INPNT).NE.CPAREN) GO TO 150
C  DISREGARD BOTH THE TOKEN AND '\', AND GET NEXT INPUT TOKEN
C

STKPNT = STKPNT - 1
GO TO 20

C  END OF IF.

150  CONTINUE

C  ELSE PUSH TOKEN ONTO STACK

STACK(STKPNT) = INPUT(INPNT)
STKPNT = STKPNT + 1
GO TO 20

C  END OF IF.

C

500  CONTINUE

C  PLACE AN IDENTIFIABLE DELIMITER ONTO THE STACK

OUTPUT(OUTPNT) = END
C

C  IF PRTLVL = 0 WRITE OUT THE POSTFIX ASSIGNMENT CLAUSE
232.  IF (PRTLVL.EQ.0) WRITE (6,505) (OUTPUT(I), I = 1, OUTPNT)
C  END OF IF

C

C  CALL BLDTRP (BRADDR, NUMBX, OUTPNT, OUTPUT, REMBER, TRIPLE, TRPROW)
37.  505  FORMAT (' ', 16(I7, 1X))
238.  RETURN
239.  END

C#EJECT
SUBROUTINE BLDTRP (BRADDR, NUMBX, OUTPNT, OUTPUT, REMBER, TRIPLE, TRIPROW)

C XXXX X XXXX XXXX XXXX XXXX
C X X X X X X X X X X X
C XXXX X X X XXXX XXXX
C X X X X X X X X X
C XXXX XXXX XXXX X X X
C XXXX X X X X X
C
C * * * * * * * FUNCTION OF MODULE * * * * * * *
C
C BLDTRP ACCEPTS THE VECTOR OF ASSIGNMENT CLAUSE TOKENS WHICH ARE
C IN POSTFIX ORDER AS INPUT AND PROCESSES THEM INTO AN ARRAY OF
C ORDERED TRIPLES WHICH ARE OUTPUT.
C
C * * * * * * * PROCEDURE FOLLOWED * * * * * * *
C
C THE VECTOR OF ASSIGNMENT CLAUSE TOKENS IS PARSED UNTIL AN
C OPERATOR IS FOUND WHICH IS PLACED IN THE FIRST COLUMN OF THE
C ARRAY OF ORDERED TRIPLES. THEN THE MODULE BACKSPACES IN THE
C ASSIGNMENT CLAUSE VECTOR TO FIND THE OPERAND(S) FOR THAT
C OPERATOR. ONE OPERAND IS REQUIRED FOR FUNCTION OPERATORS AND
C TWO OPERANDS FOR THE COMMON BINARY OPERATORS. THE OPERAND(S)
C ARE PLACED IN THE SECOND AND THIRD COLUMNS OF THE ORDERED
C TRIPLE ARRAY. THE INPUT VECTOR POSITION OF THE OPERAND OF
C UNARY OPERATORS AND THE SECOND OPERAND OF BINARY OPERATORS ARE
C REPLACED WITH THE 'POP' OPERATOR (113) WHICH WILL POP THE TOP
C INTERIM RESULT FROM A STACK DURING EXECUTION. THE INPUT VECTOR
C LOCATION OF OPERATORS AND FIRST OPERAND OF THE BINARY OPERATORS
C ARE ASSIGNED A VALUE OF ZERO TO INDICATE THE OPERATOR OR OPERAND
C HAS BEEN MOVED TO THE TRIPLE ARRAY. THE ARRAY OF ORDERED
C TRIPLES IS RETURNED TO THE CALLING MODULE.

C * * * * * * * VARIABLE DECLARATIONS * * * * * * *
C
C PARAMETERS
C
C BRADDR-INTEGER ARRAY, LENGTH 20, HOLDS INTERNAL ADDRESS OF TRIPLE
C ROWS WHICH WILL GET UNCONDITIONAL BRANCH TO END OF TRIPLES
C NUMBX-INTEGER, COUNTS THE NUMBER OF PLACES FOR UNCONDITIONAL
C BRANCH TO END OF TRIPLES, INCREMENTS THE BRADDR ARRAY
C OUTPNT-INTEGER, BRINGS IN POINTER TO LAST TOKEN IN OUTPUT VECTOR;
C ALSO USED AS POINTER IN OUTPUT VECTOR
C OUTPUT-INTEGER ARRAY, LENGTH 100, FOR PASSING POSTFIX
C ASSIGNMENT CLAUSE
C REMBER-INTEGER, SAVES INTERNAL ADDRESS OF TRIPLE ARRAY TO INSERT
C ADDRESS FROM BLDTIP SUBROUTINE
48. C TRIPLE-INTEGER ARRAY, SIZE 80 BY 3, FOR RETURNING ORDERED TRIPLES
49. C TRIPRO-INTEGER, POINTER TO ROWS OF THE ARRAY OF ORDERED TRIPLES
50. C INTEGER BRADDR(20), NUMBX, OUTPNT, OUTPUT(100), REMBER,
8 & TRIPLE(80,3), TRIPRO
53. C LOCAL VARIABLES
56. C BEGIN -INTEGER, KEEPS POSITION IN OUTPUT VECTOR WHERE LAST
57. C OPERAND WAS FOUND
59. C POP -INTEGER, OPERATOR TO POP THE TOP INTERIM ORDERED TRIPLE
61. C STAKLN-INTEGER, STORES LENGTH OF THE POSTFIX STACK
62. C TEMP -INTEGER, USED FOR DETERMINING IF TOKEN IS A VARIABLE OR
64. C CONSTANT
65. C INTEGER BEGIN, BRANCH, END, POP, STAKLN, TEMP
66. C DATA BRANCH/50017/, END/50030/, POP/50019/
70. C * * * * * * EXECUTABLE CODE * * * * * *
70. C STAKLN = OUTPNT
71. C BEGIN = 1
72. C OUTPNT = 1
73. C 10 CONTINUE
75. C REPEAT
76. C WHILE THE TOKEN IS NOT THE 'END' DELIMITER
77. C IF (OUTPUT(OUTPNT).EQ.END) GO TO 500
78. C DO PARSE THE TOKENS, MOVING OPERATORS AND OPERANDS INTO THE
79. C PROPER PLACE IN THE ARRAY OF ORDERED TRIPLES
80. C REPEAT
82. C PARSE THE POSTFIX VECTOR UNTIL AN OPERATOR IS FOUND
83. C STARTING WHERE LAST OPERAND WAS FOUND
84. C DO 30 OUTPNT = BEGIN, STAKLN
85. C IF POSTFIX STACK DELIMITER IS ENCOUNTERED, STOP SUBROUTINE
86. C IF (OUTPUT(OUTPNT).EQ.END) GO TO 500
87. C END OF IF
88. C TEMP = OUTPUT(OUTPNT)/10000
89. C IF ((TEMP.EQ.5).OR.(TEMP.EQ.6)) GO TO 40
90. C 30 CONTINUE
91. C END OF REPEAT
92. C 40 BEGIN = OUTPNT + 1
94. C TRIPRO = TRIPRO + 1
95. C PLACE OPERATOR FOUND IN FIRST COLUMN OF THE TRIPLE ARRAY
96. C TRIPLE(TRIPRO,1) = OUTPUT(OUTPNT)
97. C BLANK OUT THE POSITION OF THE OPERATOR TOKEN IN THE POSTFIX ARRAY
OUTPUT(OUTPNT) = 0

C REPEAT
C BACKSPACE IN OUTPUT VECTOR UNTIL OPERAND OR PREVIOUS TRIPLE
C INTERIM RESULT IS FOUND
DO 50 I = 1, STAKLN
  OUTPUT(OUTPNT) = OUTPUT(OUTPNT) - 1
  TEMP = IABS(OUTPUT(OUTPNT)/10000)
  IF (((TEMP.EQ.2).OR.((TEMP.GE.7).AND.(TEMP.LE.9)).OR.
       (TEMP.GE.15)).OR.(OUTPUT(OUTPNT).EQ.POP)) GO TO 70

50 CONTINUE
C END OF REPEAT
C
70 CONTINUE
C PLACE FIRST OPERAND IN THIRD COLUMN OF TRIPLE ARRAY
TRIPLE(TRPROU,3) = OUTPUT(OUTPNT)
C STORE 'POP' OPERATOR IN THE POSTFIX STRING
OUTPUT(OUTPNT) = POP
C IF OPERATOR WAS A UNARY OPERATOR (FUNCTION), THEN GET NEXT TOKEN
IF (((TRIPLE(TRPROU,1))/10000.EQ.6) GO TO 10
C ELSE GET THE SECOND OPERAND
C REPEAT
C BACKSPACE IN OUTPUT VECTOR UNTIL OPERAND OR PREVIOUS TRIPLE
C INTERIM RESULT IS FOUND
DO 90 I = 1, STAKLN
  OUTPUT(OUTPNT) = OUTPUT(OUTPNT) - 1
  TEMP = IABS(OUTPUT(OUTPNT)/10000)
  IF (((TEMP.EQ.2).OR.((TEMP.GE.7).AND.(TEMP.LE.9)).OR.
       (TEMP.GE.15)).OR.(OUTPUT(OUTPNT).EQ.POP)) GO TO 100
90 CONTINUE
C END OF REPEAT
C END OF IF
C
100 CONTINUE
C PLACE SECOND OPERAND IN SECOND COLUMN OF TRIPLE ARRAY
TRIPLE(TRPROU,2) = OUTPUT(OUTPNT)
C BLANK OUT THE POSITION OF THE OPERAND TOKEN IN POSTFIX STRING
OUTPUT(OUTPNT) = 0
C END OF IF
C GET THE NEXT TOKEN
GO TO 10
C END OF REPEAT WHILE
C
500 CONTINUE
C IF A RELATIONAL EXPRESSION EXISTS IN THE ORDERED TRIPLES
IF (REMBER.EQ.0) GO TO 600
C THEN INSERT THE ADDRESS OF THE NEXT TRIPLE ROW AS THE ADDRESS OF
C THE CONDITIONAL BRANCH ASSOCIATED WITH THE RELATIONAL
C EXPRESSION.
TRIPLE(REMBER,3) = TRPROU + 2
C ZERO OUT PARAMETER
REMBER = 0
149. TRPROW = TRPROW + 1
150. C INSERT UNCONDITIONAL BRANCH OPERATOR INTO ORDERED TRIPLES
151. C (THE ADDRESS OF THIS BRANCH IS INSERTED IN SUBROUTINE CTRIPS)
152. TRIPLE(TRPROW, 1) = BRANCH
153. C INCREMENT THE NUMBER OF TIMES AN UNCONDITIONAL BRANCH ADDRESS
154. C HAS BEEN SAVED
155. NUMB = NUMB + 1
156. C SAVE THE INTERNAL TRIPLE ADDRESS FOR USE IN CTRIPS SUBROUTINE
157. BRADDR(NUMB) = TRPROW
158. C END OF IF
159. C
160. 600 RETURN
161.  END
162. C$EJECT
SUBROUTINE PSTFXS (INPUT, PRTLVL, TRIPLE, TRPROW)

C
C XXXXX XXXXX XXXXX XXXX X X XXXX
C X X X X X X X X
C XXXXX XXXXX X XXXX X XXXX
C X X X X X X X X
C X XXXXX X X X X XXXX
C
C
C ******** FUNCTION OF MODULE ********
C
PSTFXS TAKES A BOOLEAN EXPRESSION COMPOSED OF LOGICAL OPERATORS
WITH RELATIONAL EXPRESSION OPERANDS AS INPUT AND CONVERTS THEM
INTO POSTFIX NOTATION WHICH IS OUTPUT.

********** PROCEDURE FOLLOWED **********

IF THE TOKEN IS AN END DELIMITER OR A REMARK, THE TOKENS IN THE
STACK ARE OUTPUT UNTIL THE STACK IS EMPTY. IF THE TOKEN IS '(',
IT IS PUSHED ONTO THE STACK. IF THE TOKEN IS A MINUS SIGN A
CHECK IS MADE TO DETERMINE IF IT IS A UNARY MINUS. IF IT IS
A UNARY MINUS, THE CONSTANT OR VARIABLE TOKEN ON WHICH IT
OPERATES IS CHANGED TO A NEGATIVE VALUE. IF THE TOKEN IS AN
OPERAND (OR PORTION OF AN OPERAND) OF A LOGICAL OPERATOR (.AND.,
.OR.) TO THE OUTPUT. IF THE TOKEN IS AN OPERATOR AND THE STACK IS
EMPTY, IT IS PUSHED ONTO THE STACK. IF THE STACK IS NOT EMPTY,
THE TOKEN IS COMPARED WITH THE TOKEN AT THE TOP OF THE STACK
AND IF THE STACK TOKEN IS OF EQUAL OR HIGHER PRECEDENCE IT IS
MOVED TO OUTPUT. OTHERWISE IF THE TOKEN IS ')', BOTH IT AND
THE '(' FROM THE STACK ARE DISREGARDED. FINALLY IF NONE OF THE
ABOVE CONDITIONS ARE TRUE, THE TOKEN IS PUSHED ONTO THE STACK
AND THE NEXT OPERATOR FETCHED. OPERATOR ORDER-OF-PRECEDENCE FROM
LOWEST TO HIGHEST IS OPENING PARENTHESIS '(' , CLOSING PARENTHESIS
')', LOGICAL OR '.OR.', AND LOGICAL AND '.AND.'.

********** SUBROUTINES CALLED **********

TRIPLS-MOVES POSTFIX-ORDERED TOKENS INTO ARRAY OF ORDERED TRIPLES

********** VARIABLE DECLARATIONS **********

PARAMETERS

INPUT -INTEGER ARRAY, LENGTH 100, FOR PASSING TOKENS TO THIS
MODULE
PRTLVL-INTEGER, VARIABLE TO ALLOW PRINTING OF EXTRA INFORMATION
WHICH SHOULD BE HELPFUL IN MODIFYING OR DEBUGGING THE
47. C SYSTEM
48. C TRIPLE-INTEGER ARRAY, SIZE 80 BY 3, FOR RETURNING ORDERED TRIPLES
49. C TRIPROW-INTEGER, POINTER TO ROWS OF THE ARRAY OF ORDERED TRIPLES
50. C INTEGER INPUT(100), PRTLVL, TRIPLE(80,3), TRIPROW
51. C
52. C LOCAL VARIABLES
53. C
54. C ARRPNT-INTEGER, POINTS TO ENTRIES IN THE PRECEDENCE ARRAY
55. C CPAREN-INTEGER, POSSIBLE INPUT TOKEN OF CLOSING PARENTHESIS
56. C IN -INTEGER, POSSIBLE INPUT TOKEN 140806
57. C INPNT -INTEGER, POINTS TO TOKENS IN THE INPUT VECTOR
58. C MINUS -INTEGER, POSSIBLE INPUT TOKEN 50006
59. C NOSAVE-INTEGER, POSSIBLE INPUT TOKEN 144002
60. C NOT -INTEGER, POSSIBLE INPUT TOKEN 141607
61. C OPAREN-INTEGER, POSSIBLE INPUT TOKEN OF OPENING PARENTHESIS
62. C OUTPNT-INTEGER, POINTS TO TOKENS IN OUTPUT VECTOR
63. C OUTPUT-INTEGER ARRAY, LENGTH 100, FOR PASSING POSTFIX
64. C BOOLEAN EXPRESSION
65. C PLUS -INTEGER, POSSIBLE INPUT TOKEN 50005
66. C PREC -INTEGER ARRAY, SIZE 2 BY 4, HOLDS PRECEDENCE OF
67. C ARITHMETIC OPERATORS
68. C PRECS -INTEGER, PRECEDENCE OF TOKEN AT TOP OF THE STACK
69. C PRECT -INTEGER, PRECEDENCE OF TOKEN CURRENTLY UNDER CONSIDERATION
70. C STACK -INTEGER ARRAY, LENGTH 30, STACK FOR PROCESSING OPERATORS
71. C STKPNT-INTEGER, POINTER TO NEXT EMPTY POSITION ON THE STACK
72. C TEMP -INTEGER, USED FOR DETERMINING IF TOKEN IS A VARIABLE OR
73. C CONSTANT
74. C UMINUS-INTEGER, UNARY MINUS MULTIPLYING FACTOR
75. C INTEGER ARRPNT, CPAREN, END, IN, INPNT, MINUS, NOSAVE, NOT,
76. & OPAREN, OUTPNT, OUTPUT(100), PLUS, PREC(2,4), PRECS, PRECT,
77. & STACK(30), STKPNT, TEMP, UMINUS
78. C
79. C CONSTANTS
80. C
81. C DATA CPAREN/50004/, END/50030/, IN/140806/, MINUS/50006/,
82. & NOSAVE/144002/, NOT/141607/, OPAREN/50003/, PLUS/50005/,
83. & PREC/50003, 0, 50004, 1, 40001, 9, 40002, 8/
84. C
85. C EXECUTABLE CODE 
86. C
87. C INPNT = 0
88. C OUTPNT = 1
89. C STKPNT = 1
90. C UMINUS = +1
91. C CONTINUE
92. C INCREMENT THE INPUT ARRAY POINTER
93. C INPNT = INPNT + 1
94. C
C IF THE TOKEN DELIMITS THE BOOLEAN EXPRESSION (NOSAVE, REMARK)
68
98.  IF ((INPUT(INPNT).NE.NOSAVE).AND.
99.    & ((INPUT(INPNT)/10000).NE.1)) GO TO 40
100. C THEN
101. C
102. C WHILE THE STACK IS NOT EMPTY
103. DO 30 I = 1,30
104. IF (STKPNT.EQ.1) GO TO 500
105. STKPNT = STKPNT - 1
106. C MOVE STACK TO OUTPUT
107. OUTPUT(OUTPNT) = STACK(STKPNT)
108. 30 OUTPNT = OUTPNT + 1
109. C END OF WHILE
110. C END OF IF.
111. C
112. 40 CONTINUE
113. C IF THE TOKEN = (‘),
114. IF (INPUT(INPNT).NE.OPAREN) GO TO 50
115. C THEN PUSH TOKEN ONTO STACK
116. STACK(STKPNT) = INPUT(INPNT)
117. STKPNT = STKPNT + 1
118. GO TO 20
119. C END OF IF.
120. C
121. 50 CONTINUE
122. C IF THE TOKEN IS AN OPERAND (OR PORTION OF AN OPERAND) OF A LOGICAL
123. C OPERATOR (CONSTANT, VARIABLE, IN, NOT, OR A RELATIONAL OPERATOR)
124. TEMP = INPUT(INPNT)/10000
125. IF (((INPUT(INPNT)/10000).EQ.4).OR.(INPUT(INPNT).EQ.CPAREN)
126.     & .OR.(INPUT(INPNT).EQ.MINUS).OR.(INPUT(INPNT).EQ.PLUS)
127.     & .OR.(INPUT(INPNT).EQ.OPAREN)) GO TO 60
128. C THEN MOVE TOKEN TO OUTPUT
129. OUTPUT(OUTPNT) = INPUT(INPNT) * UMINUS
130. C END OF IF
131. C RESET UNARY MINUS FACTOR TO POSITIVE
132. UMINUS = +1
133. OUTPNT = OUTPNT + 1
134. GO TO 20
135. C
136. C IF TOKEN IS ‘+’ OR ‘-’
137. 60 CONTINUE
138. IF((INPUT(INPNT).NE.PLUS).AND.(INPUT(INPNT).NE.MINUS)) GO TO 80
139. C THEN IF THIS IS THE FIRST TOKEN OF THIS RELATIONAL EXPRESSION
140. C THEN TOKEN IS A UNARY MINUS OR UNARY PLUS
141. IF (INPNT.EQ.1) GO TO 70
142. C END OF IF.
143. C ELSE IF PREVIOUS TOKEN IS A FUNCTION OR OPERATOR BUT NOT ‘)’
144. C THEN TOKEN IS A UNARY MINUS OR UNARY PLUS
145. TEMP = INPUT(INPNT - 1)/10000
146. IF (((TEMP.NE.5).AND.(TEMP.NE.6)).OR.
147.     & (INPUT(INPNT-1).EQ.CPAREN)) GO TO 80
148. C END OF IF.
C END OF IF.

70 CONTINUE
C IF TOKEN IS UNARY PLUS DISREGARD AND GET NEXT TOKEN
C IF (INPUT(INPNT).EQ.PLUS) GO TO 20
C END OF IF
C IF TOKEN IS UNARY MINUS CHANGE SIGN OF UNARY MINUS FACTOR
UMINUS = -UMINUS
C RETURN FOR NEXT TOKEN
GO TO 20
C END OF IF.

80 CONTINUE
C IF STACK IS EMPTY
C IF (STKPNT.NE.1) GO TO 90
C THEN PUSH TOKEN ONTO STACK
STACK(STKPNT) = INPUT(INPNT)
STKPNT = STKPNT + 1
GO TO 20
C END OF IF.

90 CONTINUE
C ASSIGN PRECEDENCE OF OPERATOR AT TOP OF STACK TO 'PRECS'
C REPEAT
C COMPARE TOKEN AT TOP OF STACK WITH PRECEDENCE ARRAY
DO 100 ARRPNT = 1,4
100 IF (STACK(STKPNT - 1).EQ.PREC(1,ARRPNT)) GO TO 110
C END OF REPEAT.
110 PRECS = PREC(2,ARRPNT)
C REPEAT
C ASSIGN PRECEDENCE OF OPERATOR CURRENTLY UNDER CONSIDERATION TO 'PRECT'
C REPEAT
C COMPARE INPUT TOKEN WITH PRECEDENCE ARRAY
DO 120 ARRPNT = 1,4
120 IF (INPUT(INPNT).EQ.PREC(IARRPNT)) GO TO 130
130 CONTINUE
C END OF REPEAT.

C COMPARE PRECEDENCE OF TOKEN AT TOP OF STACK WITH INPUT TOKEN
C IF PRECS > PRECT
IF (PRECS.LT.PRECT) GO TO 140
C THEN MOVE TOP OF STACK OPERATOR TO OUTPUT AND COMPARE TOKEN
STKPNT = STKPNT - 1
OUTPUT(OUTPNT) = STACK(STKPNT)
OUTPNT = OUTPNT + 1
GO TO 80
140 CONTINUE
C ELSE IF Token = '}'
200. IF (INPUT(INPNT).NE.CPAREN) GO TO 150
201. C THEN DISREGARD BOTH THE TOKEN AND '(', AND GET NEXT TOKEN
202. STKPNT = STKPNT - 1
203. GO TO 20
204. 150 CONTINUE
205. C ELSE PUSH TOKEN ONTO STACK
206. STACK(STKPNT) = INPUT(INPNT)
207. STKPNT = STKPNT + 1
208. GO TO 20
209. C END OF IF.
210. C END OF IF.
211. C
212. 500 CONTINUE
213. C PLACE AN IDENTIFIABLE DELIMITER ONTO THE STACK
214. OUTPUT(OUTPNT) = END
215. C
216. C IF PRTLVL = 0 WRITE OUT THE POSTFIX BOOLEAN EXPRESSION
217. IF (PRTLVL.EQ.0) WRITE (6,505) (OUTPUT(I), I = 1, OUTPNT)
218. C END OF IF
219. C
220. 505 FORMAT (' ', 16(I7, 1X))
221. CALL TRIPLS (OUTPNT, OUTPUT, PRTLVL, TRIPLE, TRPROW)
222. RETURN
223. END
SUBROUTINE TRIPLS (OUTPNT, OUTPUT, PRTLVL, TRIPLE, TRPROU)

TRIPLS ACCEPTS THE VECTOR OF BOOLEAN EXPRESSION TOKENS WHICH ARE
IN POSTFIX ORDER AS INPUT AND PROCESSES THEM INTO AN ARRAY OF
ORDERED TRIPLES WHICH ARE OUTPUT.

THE VECTOR OF BOOLEAN EXPRESSION TOKENS IS PARSED AND THE
TOKENS IDENTIFIED. LOGICAL OPERATORS (.AND., .OR.) ARE MOVED
TO COLUMN ONE OF THE TRIPLE ARRAY. THE OPERANDS 'IN GROUP/
MODULE' OR 'NOT IN GROUP/MODULE' ARE MOVED DIRECTLY TO A TRIPLE.
THE OPERANDS OF RELATIONAL EXPRESSIONS ARE PLACED IN COLUMNS TWO
AND THREE OF THE TRIPLE WITH THE 'SUB' COMMAND IN COLUMN ONE.
A CONDITIONAL AND UNCONDITIONAL BRANCH ARE PLACED IN THE TRIPLES
TO CAUSE A '1' TO BE PUSHED ONTO THE STACK IF THE RELATIONAL
EXPRESSION IS TRUE OR A '0' IF IT IS FALSE. THE FINAL TRIPLE
IS GIVEN THE COMMAND 'SETMSK' WHICH WILL POP THE FINAL RESULT
OF THE BOOLEAN EXPRESSION FROM THE STACK AND SET A MASK TO '1'
IF TRUE OR '0' IF FALSE. THE ARRAY OF ORDERED TRIPLES IS
RETURNED TO THE CALLING MODULE.

PARAMETERS

OUTPNT-INTEGER, BRINGS IN POINTER TO LAST TOKEN IN OUTPUT VECTOR;
ALSO USED AS POINTER IN OUTPUT VECTOR
OUTPUT-INTEGER ARRAY, LENGTH 100, FOR PASSING POSTFIX BOOLEAN EXPRESSION
PRTLVL-INTEGER, VARIABLE TO ALLOW PRINTING OF EXTRA INFORMATION
WHICH SHOULD BE HELPFUL IN MODIFYING OR DEBUGGING THE
SYSTEM
TRIPLE-INTEGER ARRAY, SIZE 80 BY 3, FOR RETURNING ORDERED TRIPLES
TRPROU-INTEGER, POINTER TO ROWS OF THE ARRAY OF ORDERED TRIPLES
INTEGER OUTPNT, OUTPUT(100), PRTLVL, TRIPLE(80,3), TRPROU
C LOCAL VARIABLES

C BEGIN -INTEGER, keeps position in output vector where last operand was found
C END -INTEGER, delimiter for postfix stack
C NOT -INTEGER, possible input token 141607
C POP -INTEGER, operator to pop the top interim ordered triple
C RESULT FROM A STACK
C RELOPS-INTEGER ARRAY, SIZE 2 BY 6, HOLDS RELATIONAL OPERATOR AND BRANCH CODES. THE OPERATOR CODES INSERTED INTO THE ORDERED TRIPLE ARRAY ARE:

<table>
<thead>
<tr>
<th>INPUT</th>
<th>RELATIONAL OPERATOR</th>
<th>BRANCH CONDITION</th>
<th>BRANCH CODE</th>
</tr>
</thead>
<tbody>
<tr>
<td>30001</td>
<td>.EQ.</td>
<td>BNZ</td>
<td>50011</td>
</tr>
<tr>
<td>30002</td>
<td>.NE.</td>
<td>BI</td>
<td>50012</td>
</tr>
<tr>
<td>30003</td>
<td>.GT.</td>
<td>BNZ</td>
<td>50013</td>
</tr>
<tr>
<td>30004</td>
<td>.LT.</td>
<td>BPZ</td>
<td>50014</td>
</tr>
<tr>
<td>30005</td>
<td>.GE.</td>
<td>BP</td>
<td>50015</td>
</tr>
<tr>
<td>30006</td>
<td>.LE.</td>
<td>BM</td>
<td>50016</td>
</tr>
</tbody>
</table>

C SAVE -INTEGER, save the address of operand of logical operator
C TEMP -INTEGER, used for determining if token is a variable or constant

INTEGER ADD, AND, BEGIN/1/, BNZ, BRANCH, BOOLEAN, END, OR, POP,
& RELOPS(2,6), SAVE, SELECT, SETMSK, SUB, TEMP
DATA ADD/50018/, AND/40001/, BNZ/50011/, BRANCH/50017/,
& END/50030/, IN/140806/, OR/40002/, POP/50019/,
& RELOPS/30001, 50011, 30002, 50012, 30003, 50013, 30004, 50014,
& 30005, 50015, 30006, 50016/, SELECT/50026/, SETMSK/50020/,
& SUB/50006/

***** EXECUTABLE CODE *****

OUTPNT = 0
CONTINUE
OUTPNT = OUTPNT + 1
TRPROW = TRPROW + 1
C IF THE TOKEN IS END DELIMITER THEN FINALIZE THE TRIPLE ARRAY
IF (OUTPUT(OUTPNT).EQ. END) GO TO 400
END OF IF
C IF THE TOKEN IS `'OR.' PLACE IT IN THE TRIPLE ARRAY
IF (OUTPUT(OUTPNT).NE. OR) GO TO 30
TRIPLE(TRPROW,1) = 50022
GO TO 10
C END OF IF
C IF THE TOKEN IS `'AND.' PLACE IT IN THE TRIPLE ARRAY
IF (OUTPUT(OUTPNT).NE. AND) GO TO 50
TRIPLE(TRPROW,1) = 50021
CONTINUE
GO TO 10

C END OF IF

50 CONTINUE

C IF THE TOKEN IS 'IN' PLACE 'IN' AND THE GROUP IDENTIFIER IN THE
C TRIPLE ARRAY

IF (OUTPUT(OUTPNT),NE.IN) GO TO 70

TRIPLE(TRPROW,1) = 50023

TRIPLE(TRPROW,3) = OUTPUT(OUTPNT + 1)

OUTPNT = OUTPNT + 1

GO TO 10

C END OF IF

70 CONTINUE

C IF THE TOKEN IS 'NOT' PLACE 'NOT IN' AND THE GROUP IDENTIFIER IN
C THE TRIPLE ARRAY

IF (OUTPUT(OUTPNT),NE.NOT) GO TO 100

TRIPLE(TRPROW,1) = 50024

TRIPLE(TRPROW,3) = OUTPUT(OUTPNT + 2)

OUTPNT = OUTPNT + 2

GO TO 10

C END OF IF

ANY TOKENS WHICH FILTER DOWN TO THIS POINT ARE RELATIONAL EXPRESSION
C OPERANDS OF THE LOGICAL OPERATORS. PLACE THE SUBTRACT OPERATOR
C IN COLUMN ONE AND THE TWO VARIABLES OR CONSTANTS IN COLUMNS
C TWO AND THREE OF THE TRIPLE ARRAY

100 TRIPLE(TRPROW,1) = SUB

TRIPLE(TRPROW,2) = OUTPUT(OUTPNT)

TRIPLE(TRPROW,3) = OUTPUT(OUTPNT + 2)

TRPROW = TRPROW + 1

OUTPNT = OUTPNT + 1

C OBTAIN THE PROPER CODE FOR THE RELATIONAL OPERATOR
C REPEAT
C COMPARE INPUT TOKEN WITH RELATIONAL OPERATORS UNTIL MATCH
DO 140 I = 1,6

IF (OUTPUT(OUTPNT) .EQ. RELOPS(1,I)) GO TO 150

C END OF REPEAT

C CONTINUE

PLACE THE PROPER BRANCH CODE, BASED ON THE RELATIONAL OPERATOR,
C IN COLUMN ONE OF THE NEXT TRIPLE ROW

TRIPLE(TRPROW,1) = RELOPS(2,I)

C PLACE 'POP' OPERATOR IN COLUMN TWO OF THE TRIPLE ARRAY

TRIPLE(TRPROW,2) = POP

TRIPLE(TRPROW,3) = TRPROW + 3

CONTINUE

TRPROW = TRPROW + 1

C PLACE 'ADD 0 1' IN TRIPLE ARRAY FOR TRUE CONDITION

TRIPLE(TRPROW,1) = ADD
TRIPLE(TRPROU,2) = 0
TRIPLE(TRPROU,3) = 1
TRPROW = TRPROW + 1

C PLACE 'UNCONDITIONAL BRANCH' AND TRIPLE ADDRESS IN TRIPLE ARRAY
TRIPLE(TRPROU,1) = BRANCH
TRIPLE(TRPROU,3) = TRPROW + 2
TRPROW = TRPROW + 1

C PLACE 'ADD 0 0' IN TRIPLE ARRAY
TRIPLE(TRPROU,1) = ADD
TRIPLE(TRPROU,2) = TRIPLE(TRPROU,3) = 0
OUTPNT = OUTPNT + 1
GO TO 10

C FINALIZE THE TRIPLE ARRAY FOR BOOLEAN EXPRESSIONS. SETMSK WILL
CAUSE THE EXECUTION PHASE TO SET A FLAG ACCORDING TO THE
RESULTS OF A TRUE OR FALSE BOOLEAN EXPRESSION.
400 CONTINUE
TRIPLE(TRPROU,1) = SETMSK
TRIPLE(TRPROU,2) = POP
TRIPLE(TRPROU,3) = SELECT
TRPROW = TRPROW + 1

C IF PRTLVL = 0 WRITE OUT THE ORDERED TRIPLES
IF (PRTLVL.NE.0) GO TO 700
600 N = TRPROW
DO 550 N, TRPROW = 1,N
550 WRITE (6,510) (TRIPLE(TRPROU,J), J = 1, 3)
510 FORMAT (' ', T10, I7, T25, I7, T40, I7)
700 RETURN
END
REFERENCES


2. Department of Defense, TASK ANALYSIS USING EXECUTIVE INSTRUCTION GUIDES.


5. Occupational Research Program, UNIX USER'S GUIDE, not to be completed until approximately March 1970.

