

AD-A101 628

AIR FORCE INST OF TECH WRIGHT-PATTERSON AFB OH
THE DESIGN AND IMPLEMENTATION OF A TRANSLATOR FOR ARITHMETIC AN--ETC(U)
1980 M L BISHOP

F/G 9/2

UNCLASSIFIED

AFIT-CI-80-61R

NL

1 OF 1
40 A
10 628

END
DATE
FILMED
8-81
DTIC

UNCLASS

①

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

AFIT - CI - REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

AD A101628

| | | |
|--|-----------------------------------|--|
| 1 REPORT NUMBER 80-61R | 2 GOVT ACCESSION NO AD-A101628 | 3 RECIPIENT'S CATALOG NUMBER 608 |
| 4 TITLE (and Subtitle) The Design and Implementation of a Translator for Arithmetic and Boolean Expressions | | 5 TYPE OF REPORT & PERIOD COVERED THESES/DISSEMINATION REPORT |
| 7 AUTHOR(S) Marin L. Bishop | | 8 CONTRACT OR GRANT NUMBER(S) 71111111 |
| 9 PERFORMING ORGANIZATION NAME AND ADDRESS AFIT STUDENT AT: Texas A&M University | | 10 PROGRAM ELEMENT PROJECT, TASK AREA & WORK UNIT NUMBERS |
| 11 CONTROLLING OFFICE NAME AND ADDRESS AFIT/NR WPAFB OH 45433 | | 12 REPORT DATE 1980 |
| 14 MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office) [Signature] | | 13 NUMBER OF PAGES 75 |
| 16 DISTRIBUTION STATEMENT (of this Report) APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED | | 15 SECURITY CLASS (of this report) UNCLASS |
| 18 SUPPLEMENTARY NOTES APPROVED FOR PUBLIC RELEASE: IAW AFR 190-17 23 JUN 1981 | | 15a DECLASSIFICATION DOWNGRADING SCHEDULE |
| 19 KEY WORDS (Continue on reverse side if necessary and identify by block number) | | |
| 20 ABSTRACT (Continue on reverse side if necessary and identify by block number) ATTACHED | | |

LEVEL II

DTIC ELECTED
S JUL 21 1981 D

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

DTIC FILE COPY.

81 7 16 008

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):

| | |
|--------------------|-------------------------------------|
| Approved For | |
| TRD | <input checked="" type="checkbox"/> |
| TRD TRD | <input type="checkbox"/> |
| TRD TRD | <input type="checkbox"/> |
| Availability Codes | |
| Dist | Avail and/or Special |
| A | |

FOLD DOWN ON OUTSIDE - SEAL WITH TAPE

AFIT/NR
WRIGHT-PATTERSON AFB OH 45433
OFFICIAL BUSINESS
PENALTY FOR PRIVATE USE. \$300



NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES



BUSINESS REPLY MAIL
FIRST CLASS PERMIT NO. 73236 WASHINGTON D.C.

POSTAGE WILL BE PAID BY ADDRESSEE

AFIT/ DAA
Wright-Patterson AFB OH 45433

FOLD IN

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

ABSTRACT

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 Manning 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

| Chapter | | Page |
|---------|-------------------------------------|------|
| I | INTRODUCTION | 1 |
| II | THE INTERPRETER | 7 |
| III | DESIGN CONSIDERATIONS | 14 |
| IV | DESCRIPTION OF ALGORITHMS | 19 |
| | Full Assignment Clause | 19 |
| | Boolean Expression | 30 |
| V | CONCLUSION | 36 |
| | Future Efforts | 37 |

APPENDICES

| Appendix | | |
|----------|----------------------------|----|
| A | SYNTAX GRAPHS | 41 |
| B | USER'S GUIDE | 45 |
| C | PROGRAM LISTINGS | 49 |
| | REFERENCES | 75 |

FIGURES

| Figure | Page |
|------------------------------------|------|
| 1. HEIRARCHY CHART | 17 |
| 2. FLOW CHART FOR CTrips | 21 |
| 3. FLOW CHART FOR RELATE | 23 |
| 4. FLOW CHART FOR ECSTFX | 26 |
| 5. FLOW CHART FOR BLDIPP | 28 |
| 6. FLOW CHART FOR TRIPLS | 34 |

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 duties. 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 an 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 3. 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, A, B, and C on task 3, and A and C on task 4. With this grouping 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. Their 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 often gathered, 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 quantified form. It should be obtained from a large sample of respondents to minimize the effect of perturbations. It is in the processing and analysis of the 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, modifiability, 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 5 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 = 3" portion is a relational expression which is tested for a true or false indication to determine which assignment clause is executed. Only one-

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

```
IN G1 .AND. ((T1 <= 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 <= 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 <= 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 <= 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 parenthesis, 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 add 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

$$(\langle \text{Operator} \rangle, \langle \text{Operand 1} \rangle, \langle \text{Operand 2} \rangle)$$

where $\langle \text{Operand 1} \rangle$ and $\langle \text{Operand 2} \rangle$ specify the arguments for the operator. As an example, $A + B$ might be represented by

$$+, A, B$$

and $A * B + C * D$ could be represented by the sequence

1. $*, A, B$
2. $*, C, D$
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 3 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 range 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 COBOL programs. They will be used by various agencies on numerous 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 in to 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 PL/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

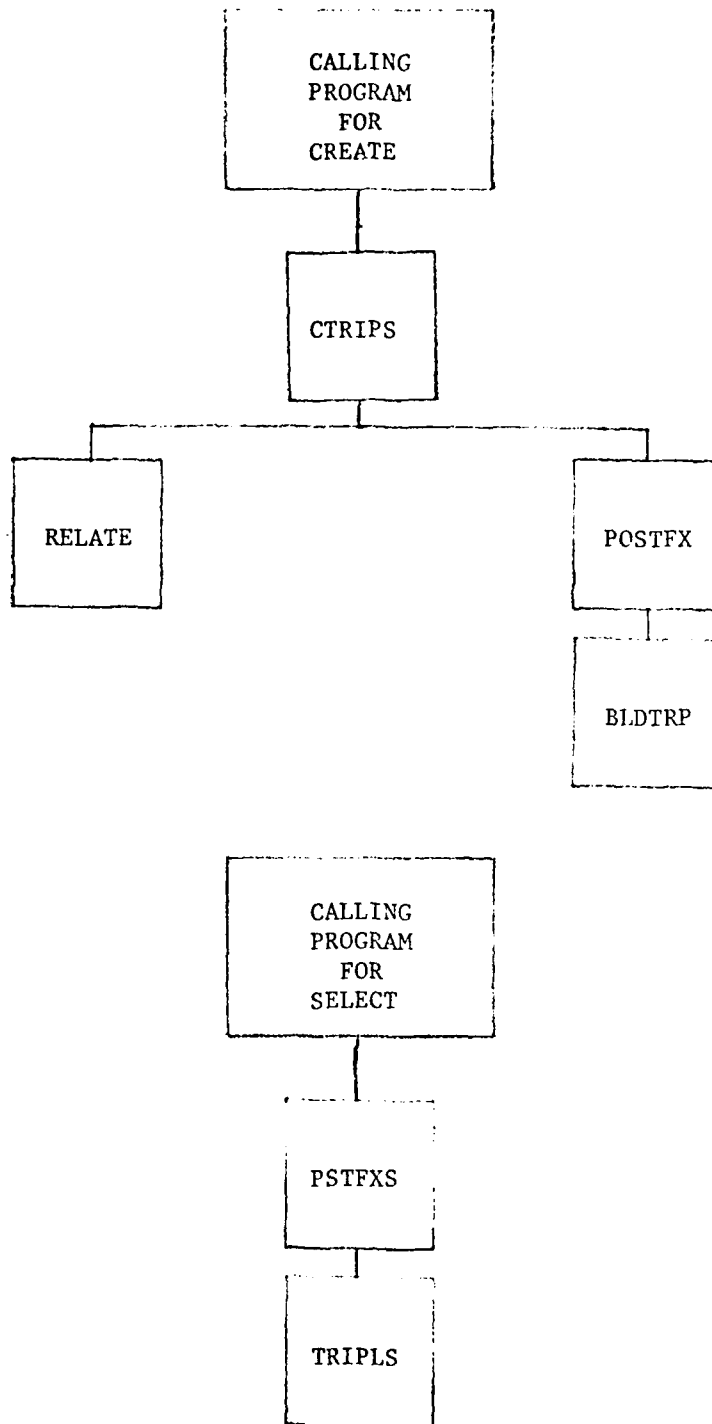


Figure 1. Hierarchy Chart

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 GO TOs, they are the only means of transferring control for a conditional construct in the FORTRAN language. Usually the GO TO 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 GO TOs 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 triples.

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 needed to be transformed into the internal form of ordered triples for efficient execution. Although the two outputs were to be of the same format, separate processes were designed 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 Tom := 5
IF Bill = 4 THEN Tom := 6
      ELSE Tom := 1 'comment'.

```

When transformed to ordered triples this expression becomes

| Row | Operator | Operand 1 | Operand 2 |
|-----|----------|-----------|-----------|
| 1. | SUE | Bill | 3 |
| 2. | BNZ | POP | (5) |
| 3. | := | Tom | 5 |
| 4. | B | | (10) |
| 5. | SUP | Bill | 4 |
| 6. | BNZ | POP | (9) |
| 7. | := | Tom | 6 |
| 8. | B | | (10) |
| 9. | := | Tom | 1 |

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 POP 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

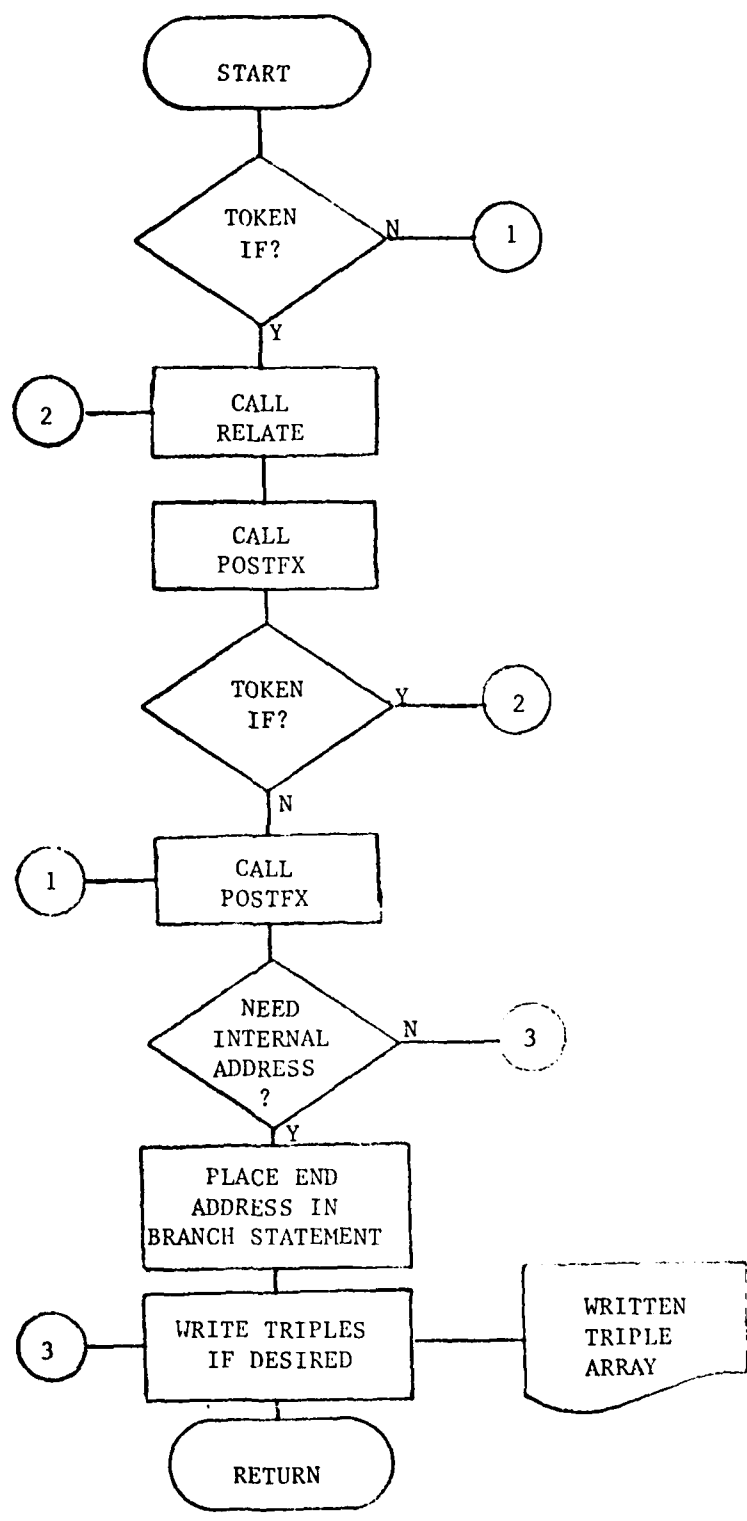


Figure 2. Flow Chart For CTRIPS

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-THENS. 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.

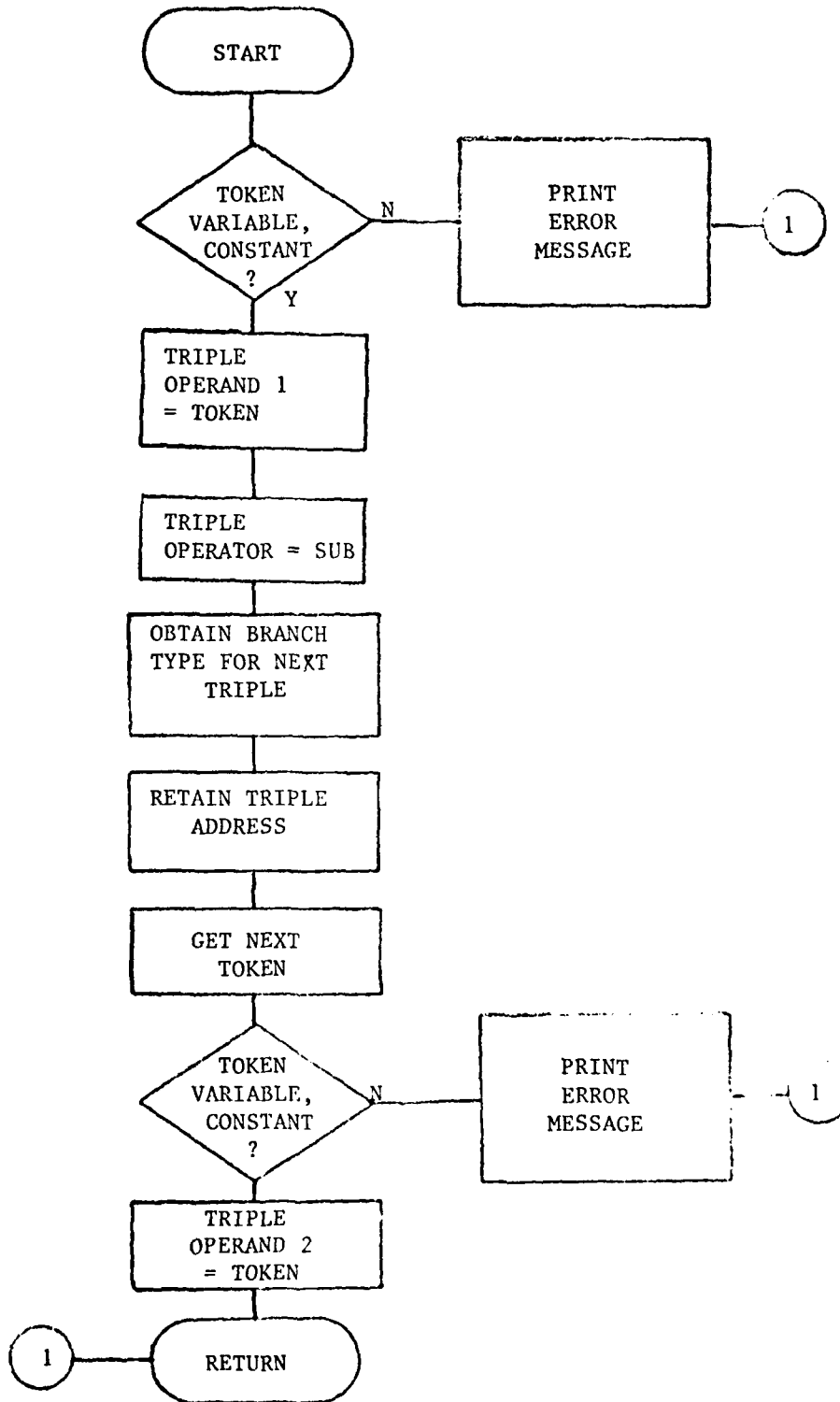


Figure 3. Flow Chart For RELATE

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

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

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

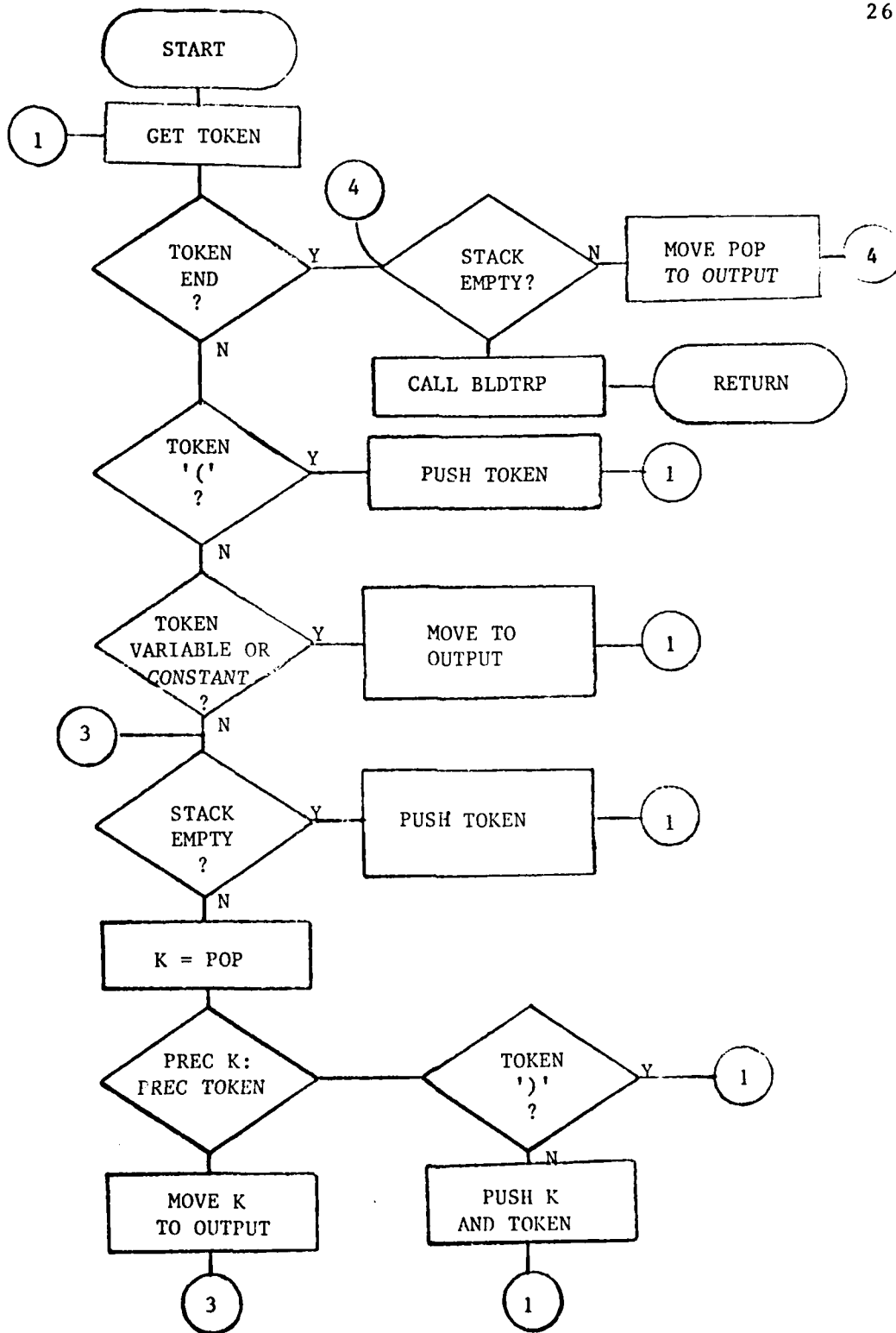


Figure 4. Flow Chart For POSTFX

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 comparison 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 delimiters 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 delimiters are encountered, the stack is moved to output and an internal end delimiter is placed in the output stream. POSTFX then calls BLDTRP to convert the postfix assignment clause into ordered triples.

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

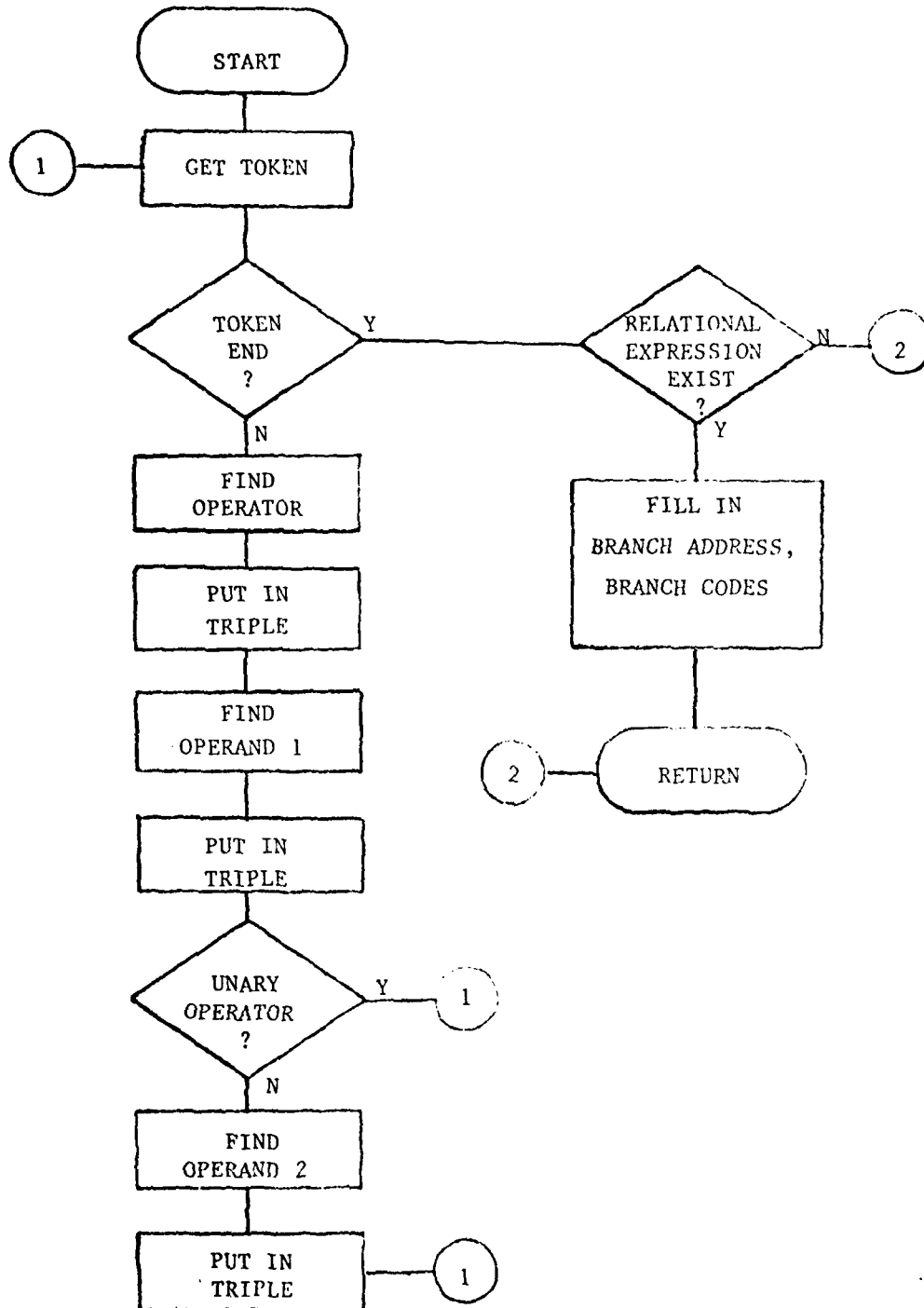


Figure 5. Flow Chart For BLDTRP

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 data base or selected from columns not within the specified 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:

| <u>Row</u> | <u>Operator</u> | <u>Operand 1</u> | <u>Operand 2</u> |
|------------|-----------------|------------------|------------------|
| 1 | IN | | G1 |
| 2 | SUF | T1 | 3 |
| 3 | BNZ | POP | (6) |
| 4 | ADD | 0 | 1 |
| 5 | E | | (7) |
| 6 | ADD | 0 | 0 |
| 7 | AND | | |
| 8 | SUF | T2 | 2 |
| 9 | EP | POP | (12) |
| 10 | ADD | 0 | 1 |
| 11 | E | | (13) |
| 12 | ADD | 0 | 0 |
| 13 | SUF | T2 | 5 |
| 14 | BMZ | POP | (17) |
| 15 | ADD | 0 | 1 |
| 16 | E | | (18) |
| 17 | ADD | 0 | 0 |
| 18 | CR | | |
| 19 | AND | | |
| 20 | SETMSK | POP | SELECT |

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, OR 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 baseline 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 PSTFYS, a six letter name for "postfix select". The flow chart for this subroutine is the same as that of POSTFY (Figure 4) with minor exceptions of delimiters, operators, and operands. The tokens NOSAVE and REMARK are the only delimiters for the Boolean expression. The precedence of possible operators from highest to lowest is:

.AND.

.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 end delimiter is placed in the string and subroutine TRIPLS is called.

Subroutine TRIPLS (Figure 6) accepts the Pcolean expression in postfix notation and arranges the tokens into ordered triples. Again there are similarities, yet differences, when comparing subroutines IRIPLS and BLETEP.

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 identifier are placed in the triples. No branch conditions are required since the execution 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 subtract code 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 Pcolean expression is evaluated. To do this the stack is popped. A one indicates a true, a zero a false expression. A flag (SETMSK) is set to reflect this condition. Now that the triples are completed, TRIPLS returns to PSTFXS which returns to its calling module.

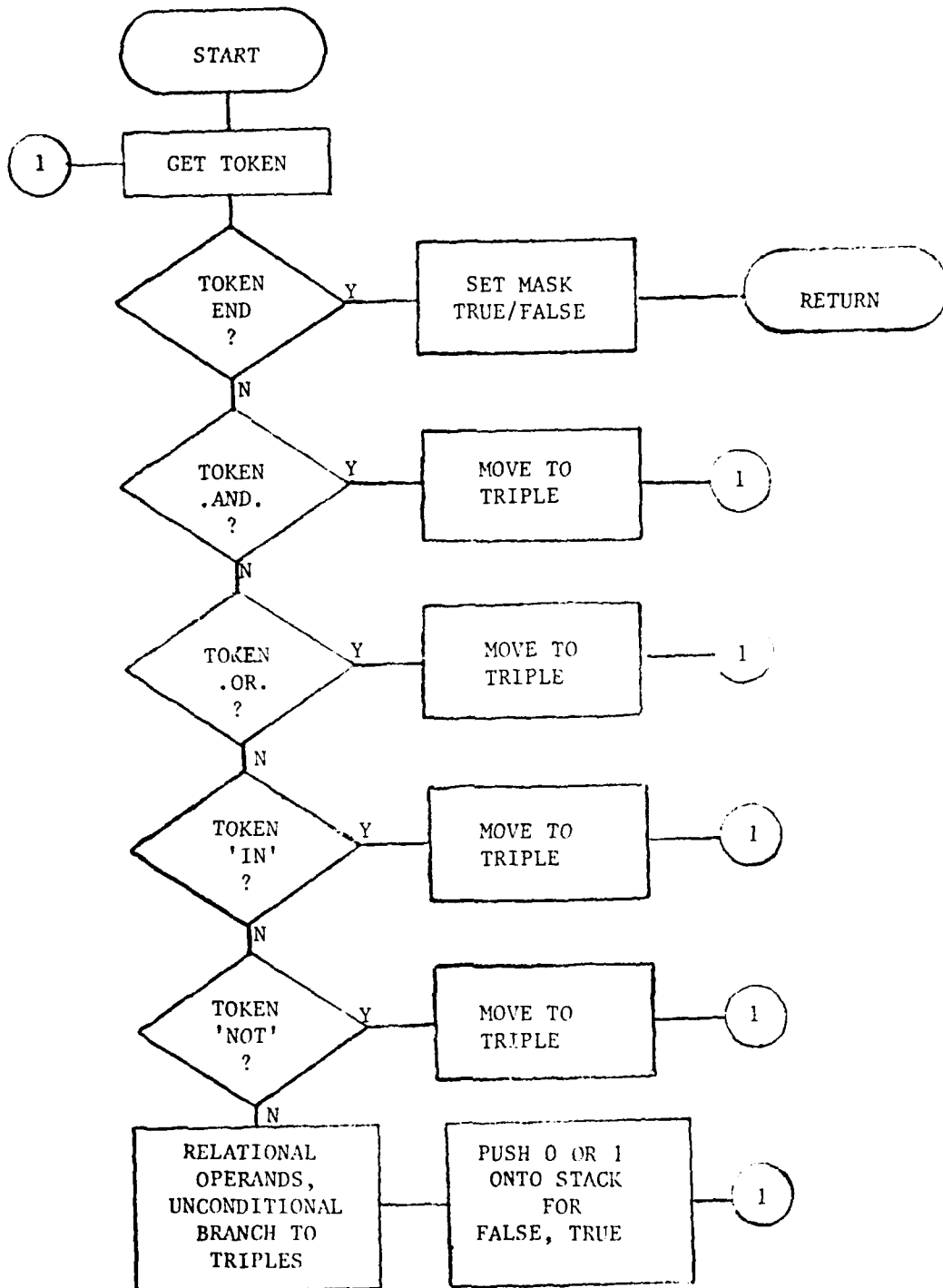


Figure 6. Flow Chart For TRIPLS

At this point, the routines to transform the full assignment clause and Eolean 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 GO TOs 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 CODAP project, thus initially did not have the broad perspective of the CODAP 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 chosen 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.

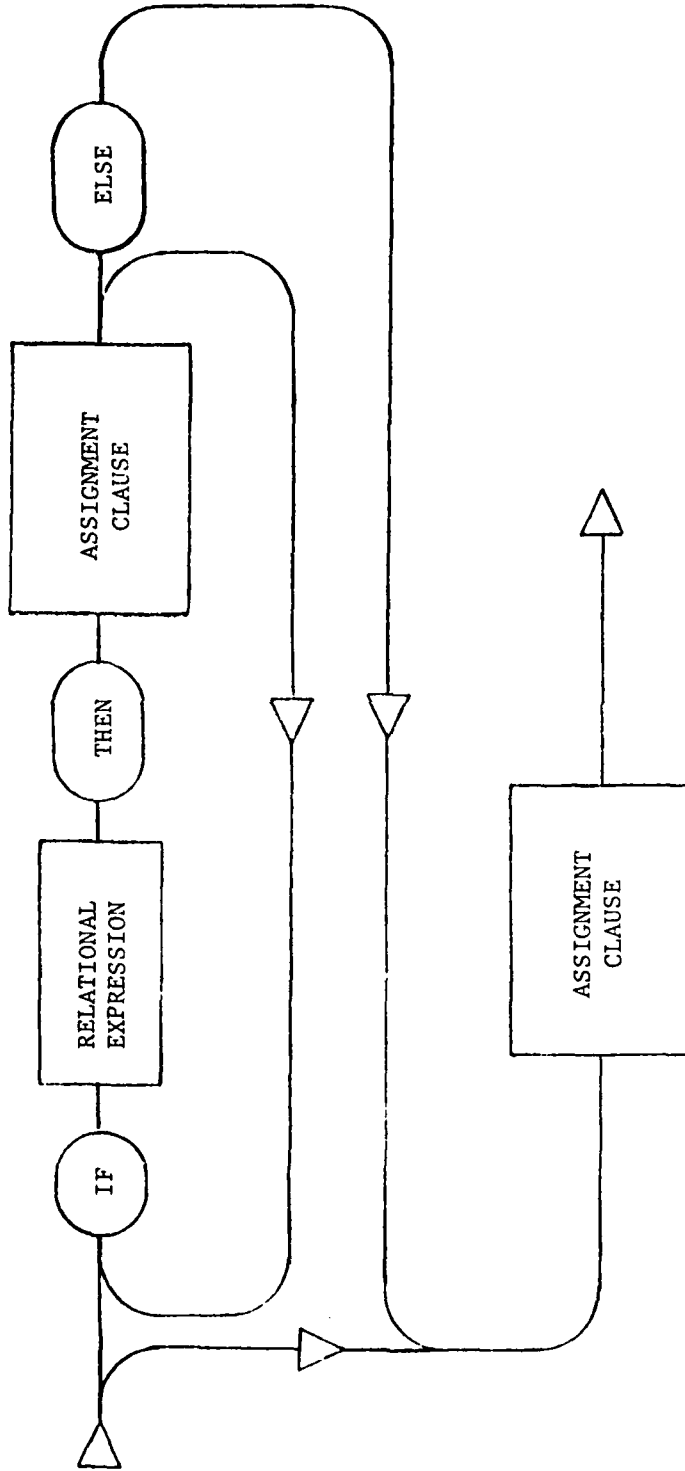
A second area for future study concerns the ordered triples constructed from the Boolean expression. The Boolean expression is composed of a series of logical operators (.AND., .OR.) and the associated operands. As developed, the entire set of triples must be executed to determine the result of the expression. 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.

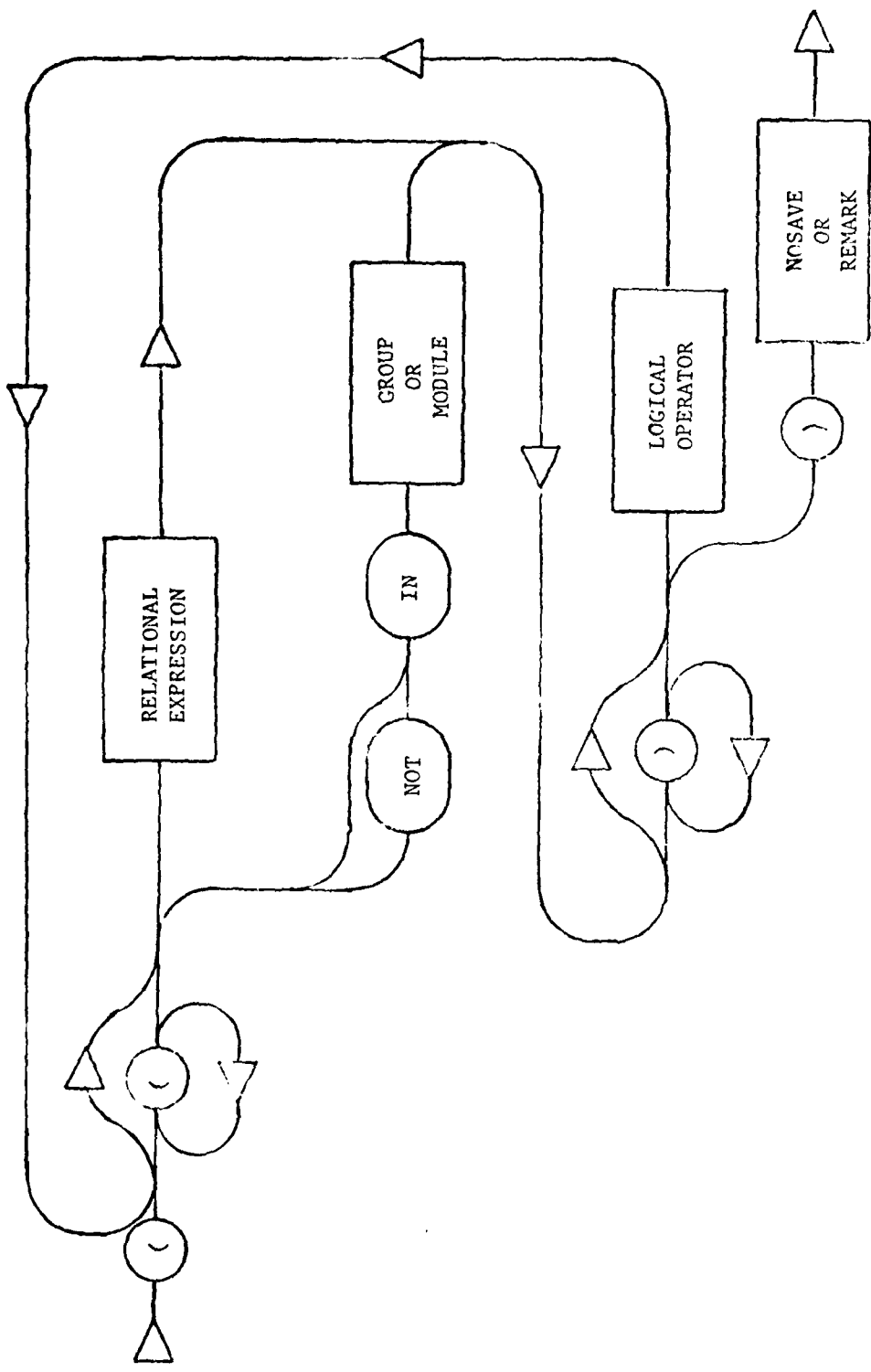
APPENDICES

APPENDIX A

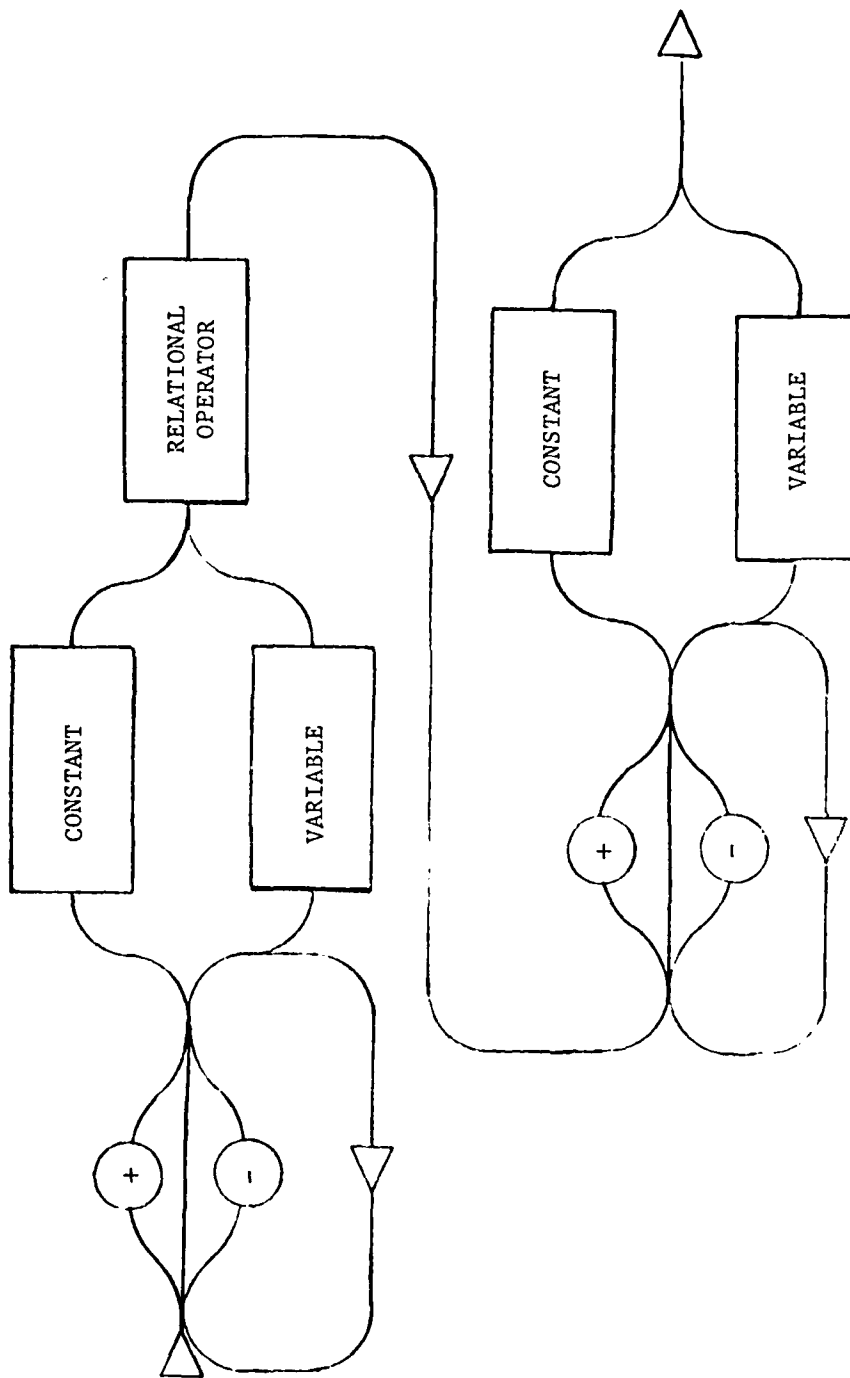
SYNTAX GRAPHS



Full Assignment Clause Syntax Graph



Boolean Expression Syntax Graph



Relational Expression Syntax Graph

APPENDIX B

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, TRIPLES, may need to be enlarged. Its present size is 80 by 3.

A stack is used in the subroutines POSTFX and PSTFXS 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.

PRTLVL 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 terminator to denote the end of the string. Anticipated delimiters are NOSAVE or REMARK.

The tokens applicable to the programs designed 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

| | |
|-------|-----------------------|
| 50003 | left parentheses '(' |
| 50004 | right parentheses ')' |
| 50005 | plus '+' |
| 50006 | minus '-' |
| 50007 | divide '/' |
| 50008 | multiply '**' |
| 50009 | exponentiation '***' |
| 50010 | assign ':=' |

Functions

| | |
|-------|-------------|
| 60002 | logarithm |
| 60009 | square root |

Variables

70000-109999 and 150000 or larger

Specific Identifiers

| | |
|--------|--------|
| 140805 | IF |
| 140806 | IN |
| 141607 | NOT |
| 142408 | THEN |
| 142409 | ELSE |
| 144002 | NOSAVE |
| 144008 | REMARK |

Generated Internal Code

| | | |
|-------|--------|-------------------------|
| 50011 | BNZ | branch on not zero |
| 50012 | BZ | branch on zero |
| 50013 | BMZ | branch on minus or zero |
| 50014 | BPZ | branch on plus or zero |
| 50015 | BM | branch on minus |
| 50016 | BP | branch on plus |
| 50017 | B | unconditional branch |
| 50018 | ADD | push onto stack |
| 50019 | POP | pop from stack |
| 50020 | SETMSK | set the mask |
| 50021 | .AND. | logical and |
| 50022 | .OR. | logical or |
| 50023 | IN | in group or module |
| 50024 | NOT IN | not in group or module |
| 50025 | CREATE | create new data |
| 50026 | SELECT | select from data base |

APPENDIX C

PROGRAM LISTINGS

```

1.          SUBROUTINE CTRIPS (INPUT, PRTLVL, TRIPLE)
2.          C
3.          C   XXXXX XXXXX XXXXX XXX XXXXX XXXXX
4.          C   X       X   X   X   X   X   X   X
5.          C   X       X   XXXXX X   XXXXX XXXXX
6.          C   X       X   X   X   X   X       X
7.          C   XXXXX   X   X   X   XXX X       XXXXX
8.          C
9.          C * * * * * FUNCTION OF MODULE * * * * *
10.         C
11.         C   CTRIPS IS THE CONTROL MODULE WHICH ACCEPTS A STRING OF TOKENS
12.         C   WHICH FORM THE FULL ASSIGNMENT CLAUSE AND CALLS THE MODULES
13.         C   WHICH CONVERT THE STRING INTO ORDERED TRIPLES FOR EXECUTION.
14.         C
15.         C * * * * * PROCEDURE FOLLOWED * * * * *
16.         C
17.         C   THE FIRST TOKEN IS CHECKED TO DETERMINE WHAT IT IS. FOR AN 'IF'
18.         C   TOKEN, THE RELATE SUBROUTINE IS CALLED TO PLACE THE TOKENS OF THE
19.         C   RELATIONAL EXPRESSION INTO THE ORDERED TRIPLE ARRAY. FOLLOWING
20.         C   THE RETURN FROM RELATE (OR IMMEDIATELY IF THERE IS NO RELATIONAL
21.         C   EXPRESSION) POSTFX SUBROUTINE IS CALLED TO REORDER THE INFIX
22.         C   ASSIGNMENT CLAUSE INTO A POSTFIX ASSIGNMENT CLAUSE.
23.         C   POSTFX CALLS BLDTRP WHICH PLACES THE POSTFIX ASSIGNMENT CLAUSE
24.         C   INTO THE ORDERED TRIPLE ARRAY BEFORE RETURNING TO THIS MODULE.
25.         C   REPEATED CALLS OF THE RELATE AND POSTFX SUBROUTINES MAY BE MADE
26.         C   TO HANDLE NUMEROUS RELATIONAL AND ASSIGNMENT CLAUSES.
27.         C
28.         C * * * * * SUBROUTINES CALLED * * * * *
29.         C
30.         C   RELATE-MOVES TOKENS OF RELATIONAL EXPRESSIONS TO THE TRIPLE ARRAY
31.         C   POSTFX-CHANGES TOKENS OF ASSIGNMENT CLAUSES INTO POSTFIX
32.         C   ORDER; THEN CALLS BLDTRP WHICH PLACES THOSE TOKENS INTO
33.         C   THE TRIPLE ARRAY
34.         C
35.         C * * * * * VARIABLE DECLARATIONS * * * * *
36.         C
37.         C   PARAMETERS
38.         C
39.         C   INPUT -INTEGER ARRAY, LENGTH 100, FOR PASSING TOKENS
40.         C   PRTLVL-INTEGER, VARIABLE TO ALLOW PRINTING OF EXTRA INFORMATION
41.         C   WHICH SHOULD BE HELPFUL IN MODIFYING OR DEBUGGING THE
42.         C   SYSTEM
43.         C   TRIPLE-INTEGER ARRAY, SIZE 80 BY 3, FOR RETURNING ORDERED TRIPLES
44.         C
45.         C   INTEGER INPUT(100), PRTLVL, TRIPLE(80,3)
46.         C

```

```

47. C LOCAL VARIABLES
48. C
49. C BRADDR-INTEGER ARRAY, LENGTH 20, HOLDS INTERNAL ADDRESS OF TRIPLE
50. C ROWS WHICH WILL GET UNCONDITIONAL BRANCH TO END OF TRIPLES
51. C INPNT -INTEGER, POINTS TO TOKENS IN THE INPUT VECTOR
52. C NUMBX -INTEGER, COUNTS THE NUMBER OF PLACES FOR UNCONDITIONAL
53. C BRANCH TO END OF TRIPLES, INCREMENTS THE BRADDR ARRAY
54. C OUTPNT-INTEGER, POINTS TO LAST TOKEN IN OUTPUT VECTOR
55. C OUTPUT-INTEGER ARRAY, LENGTH 100, FOR PASSING POSTFIX
56. C ASSIGNMENT CLAUSE
57. C REMBER-INTEGER, SAVES INTERNAL ADDRESS OF TRIPLE ARRAY TO INSERT
58. C ADDRESS FROM BLDTRP SUBROUTINE
59. C TRPROW-INTEGER, POINTER TO ROWS OF THE ARRAY OF ORDERED TRIPLES
60. C
61. C INTEGER BRADDR(20), INPNT, NUMBX, OUTPNT, OUTPUT(100), REMBER,
62. C & TRPROW
63. C
64. C * * * * * EXECUTABLE CODE * * * * *
65. C
65.1 C INPNT = 1
65.2 C NUMBX = 0
65.3 C REMBER = 0
65.4 C TRPROW = 0
66. C IF THE FIRST TOKEN IS 'IF' THEN CALL SUBROUTINE RELATE
67. C IF (INPUT(INPNT).NE.140805) GO TO 80
68. 70 INPNT = INPNT + 1
69. C CALL RELATE (INPNT, INPUT, REMBER, TRIPLE, TRPROW)
70. C END OF IF
71. C
72. C CALL POSTFX (BRADDR, NUMBX, INPNT, INPUT, OUTPUT, PRTLVL,
73. C & REMBER, TRIPLE, TRPROW)
74. C
75. C IF THE NEXT TOKEN IS 'IF' (NESTED IF-THEN CLAUSE)
76. C THEN CALL SUBROUTINE RELATE
77. C IF (INPUT(INPNT).EQ.140805) GO TO 70
78. C END OF IF
79. 80 CALL POSTFX (BRADDR, NUMBX, INPNT, INPUT, OUTPUT, PRTLVL,
80. C & REMBER, TRIPLE, TRPROW)
81. C IF INTERNAL TRIPLE ADDRESSES ARE INCOMPLETE
82. C IF (NUMBX.EQ.0) GO TO 100
83. C REPEAT
84. C PLACE THE SAVED TRIPLE ROW ADDRESS IN THE PROPER PLACES AS THE
85. C OBJECTS OF THE UNCONDITIONAL BRANCHES.
86. C DO 90 K = 1, NUMBX
87. 90 TRIPLE(BRADDR(K),3) = TRPROW + 1
88. C END OF REPEAT
89. C END OF IF
90. C
91. C IF PRTLVL = 0 WRITE OUT THE ORDERED TRIPLES
92. 100 IF (PRTLVL.NE.0) GO TO 600
93. C N = TRPROW

```

```
94.          DO 110 TRPROW = 1,N
95.      110      WRITE (6,510) (TRIPLE(TRPROW,J), J = 1, 3)
96.      C      END OF IF
97.      510      FORMAT (' ', T10, I7, T25, I7, T40, I7)
98.      600      RETURN
99.          END
100.     C$EJECT
```

```

1.          SUBROUTINE RELATE (INPNT, INPUT, REMBER, TRIPLE, TRPROW)
2.          C
3.          C   XXXXX XXXXX X           X   XXXXX XXXXX
4.          C   X  X X   X           X X   X  X
5.          C   XXXXX YXXX X           X  X  X  XXXX
6.          C   X  X X   X           XXXXX  X  X
7.          C   X  X XXXXX XXXXX X   X  X  XXXXX
8.          C
9.          C * * * * * FUNCTION OF MODULE * * * * *
10.         C
11.         C   RELATE SCANS THE INPUT STACK OF TOKENS AND CHANGES THOSE IN A
12.         C   RELATIONAL EXPRESSION INTO ORDERED TRIPLES WHICH ARE OUTPUT IN
13.         C   AN ARRAY.
14.         C
15.         C * * * * * PROCEDURE FOLLOWED * * * * *
16.         C
17.         C   INPUT RELATIONAL EXPRESSIONS ARE RESTRICTED TO THE FOLLOWING
18.         C   FORMAT:  1.  OPTIONAL STRING OF UNARY PLUSES AND/OR MINUSES;
19.         C                 2.  CONSTANT OR VARIABLE IDENTIFIER;
20.         C                 3.  RELATIONAL OPERATER;
21.         C                 4.  OPTIONAL STRING OF UNARY PLUSES AND/OR MINUSES;
22.         C                 5.  CONSTANT OR VARIABLE IDENTIFIER.
23.         C   UNARY PLUSES ARE DISREGARDED.  UNARY MINUSES ARE ACCUMULATED
24.         C   AND, IF THERE ARE AN ODD NUMBER OF THEM, THE CONSTANT OR VARIABLE
25.         C   TOKEN IS CHANGED TO A NEGATIVE VALUE.  THE FIRST CONSTANT OR
26.         C   VARIABLE IS PLACED IN THE SECOND COLUMN OF THE TRIPLE ARRAY,
27.         C   THE RELATIONAL OPERATOR IN THE FIRST COLUMN, AND THE SECOND
28.         C   CONSTANT OR VARIALBE IN THE THIRD COLUMN.
29.         C
30.         C * * * * * VARIABLE DECLARATIONS * * * * *
31.         C
32.         C  PARAMETERS
33.         C
34.         C   INPNT -INTEGER, POINTS TO TOKENS IN THE INPUT VECTOR
35.         C   INPUT -INTEGER ARRAY, LENGTH 100, FOR PASSING TOKENS TO THIS
36.         C           MODULE
37.         C   REMBER-INTEGER, SAVES INTERNAL ADDRESS OF TRIPLE ARRAY TO INSERT
38.         C           ADDRESS FROM BLDTRP SUBROUTINE
39.         C   TRIPLE-INTEGER ARRAY, SIZE 80 BY 3, FOR RETURNING ORDERED TRIPLES
40.         C   TRPROW-INTEGER, POINTER TO ROWS OF THE ARRAY OF ORDERED TRIPLES
41.         C
42.         C   INTEGER INPNT, INPUT(100), REMBER, TRIPLE(80,3), TRPROW
43.         C
44.         C  LOCAL VARIABLES
45.         C
46.         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. BNZ 50011
56. C 30002 .NE. BZ 50012
57. C 30003 .GT. BMZ 50013
58. C 30004 .LT. BPZ 50014
59. C 30005 .GE. BK 50015
60. C 30006 .LE. BP 50016
61. C RELPNT-INTEGER, POINTER 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
67. C INTEGER MINUS, PLUS, POP, RELOPS(2,6), RELPNT, SUB, TEMP, THEN,
68. C & UMINUS
69. C DATA MINUS/50006/, PLUS/50005/, POP/50019/, RELOPS/30001, 50011,
70. C & 30002, 50012, 30003, 50013, 30004, 50014, 30005, 50015,
71. C & 30006, 50016/, SUB/50006/, THEN/142408/
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. C IF (INPUT(INPNT).EQ.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. C IF (INPUT(INPNT).NE.MINUS) GO TO 40
88. C 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. C & (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.  C      TRIPLE(TRPROW,2) = INPUT(INPNT) * UMINUS
101.  C      END OF IF
102.  C
103.  C      RESET UNARY MINUS FACTOR TO POSITIVE
104.  C      UMINUS = +1
105.  C      GET THE NEXT TOKEN
106.  C      INPNT = INPNT + 1
107.  C      PLACE SUBTRACT CODE IN COLUMN ONE OF TRIPLE ARRAY
108.  C      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.  C      DO 50 RELPNT = 1,6
114.  C      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.  C      TRIPLE(TRPROW + 1, 1) = RELOPS(2,RELPNT)
122.  C      PLACE 'POP' OPERATOR IN COLUMN TWO OF THE TRIPLE ARRAY
123.  C      TRIPLE(TRPROW + 1, 2) = POP
124.  C      RETAIN THE TRIPLE ROW NUMBER FOR LATER INSERTION OF BRANCH ADDRESS
125.  C      REHBER = TRPROW + 1
126.  70   CONTINUE
127.  C      INPNT = INPNT + 1
128.  C      IF THE TOKEN IS UNARY PLUS DISREGARD AND GET THE NEXT TOKEN
129.  C      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.  C      IF (INPUT(INPNT).NE.MINUS) GO TO 90
134.  C      UMINUS = - UMINUS
135.  C      AND GET THE NEXT TOKEN
136.  C      GO TO 70
137.  C      END OF IF
138.  C
139.  90   CONTINUE
140.  C      IF TOKEN IS A CONSTANT OR VARIABLE
141.  C      TEMP = INPUT(INPNT)/10000
142.  C      IF ((TEMP.NE.2).AND.((TEMP.LT.7).OR.(TEMP.GT.9)).AND.
143.  C      & (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.  C      TRIPLE(TRPROW,3) = INPUT(INPNT) * UMINUS
147.  C      INPNT = INPNT + 1
148.  C      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
```

```

1.      SUBROUTINE POSTFX (BRADDR, NUMBX, INPNT, INPUT, OUTPUT,
2.      &                   PRTLVL, RENBER, TRIPLE, TRPROW)
3.      C
4.      C XXXXX XXXXX XXXXX XXXXX XXXXX X  X
5.      C X  X X  X X      X  X      X X
6.      C XXXXX X  X XXXXX  X  XXXX  X
7.      C X    X  X    X  X  X    X X
8.      C X      XXXXX XXXXX  X  X    X  X
9.      C
10.     C * * * * * FUNCTION OF MODULE * * * * *
11.     C
12.     C     POSTFX TAKES AN INFIX ASSIGNMENT CLAUSE AS INPUT AND CONVERTS
13.     C     IT INTO AN EXPRESSION OF POSTFIX NOTATION WHICH IS OUTPUT.
14.     C
15.     C * * * * * PROCEDURE FOLLOWED * * * * *
16.     C
17.     C     IF THE TOKEN IS AN END DELIMITER OR A REMARK, THE TOKENS IN THE
18.     C     STACK ARE OUTPUT UNTIL THE STACK IS EMPTY. IF THE TOKEN IS '(',
19.     C     IT IS PUSHED ONTO THE STACK. IF THE TOKEN IS A MINUS SIGN A
20.     C     CHECK IS MADE TO DETERMINE IF IT IS A UNARY MINUS. IF IT IS
21.     C     A UNARY MINUS, THE CONSTANT OR VARIABLE TOKEN ON WHICH IT
22.     C     OPERATES IS CHANGED TO A NEGATIVE VALUE.
23.     C     IF THE TOKEN IS A CONSTANT OR A VARIABLE, IT IS MOVED DIRECTLY
24.     C     TO THE OUTPUT. IF THE TOKEN IS AN OPERATOR AND THE STACK IS
25.     C     EMPTY, IT IS PUSHED ONTO THE STACK. IF THE STACK IS NOT EMPTY,
26.     C     THE TOKEN IS COMPARED WITH THE TOKEN AT THE TOP OF THE STACK
27.     C     AND IF THE STACK TOKEN IS OF EQUAL OR HIGHER PRECEDENCE IT IS
28.     C     MOVED TO OUTPUT. OTHERWISE IF THE TOKEN IS ')', BOTH IT AND
29.     C     THE '(' FROM THE STACK ARE DISREGARDED. FINALLY IF NONE OF THE
30.     C     ABOVE CONDITIONS ARE TRUE, THE TOKEN IS PUSHED ONTO THE STACK
31.     C     AND THE NEXT TOKEN IS FETCHED. TOKEN ORDER-OF-PRECEDENCE FROM
32.     C     LOWEST TO HIGHEST IS OPENING PARENTHESIS '(', CLOSING PARENTHESIS
33.     C     ')', PLUS OR MINUS SIGN '+ OR -', MULTIPLY OR DIVIDE '* OR /',
34.     C     EXPONENTIATION '**', UNARY MINUS, AND FUNCTIONS.
35.     C
36.     C * * * * * SUBROUTINES CALLED * * * * *
37.     C
38.     C     BLDTRP-MOVES POSTFIX-ORDERED TOKENS INTO ARRAY OF ORDERED TRIPLES
39.     C
40.     C * * * * * VARIABLE DECLARATIONS * * * * *
41.     C
42.     C PARAMETERS
43.     C
44.     C     BRADDR-INTEGER ARRAY, LENGTH 20, HOLDS INTERNAL ADDRESS OF TRIPLE
45.     C     ROWS WHICH WILL GET UNCONDITIONAL BRANCH TO END OF TRIPLES
46.     C     NUMBX -INTEGER, COUNTS THE NUMBER OF PLACES FOR UNCONDITIONAL
47.     C     BRANCH TO END OF TRIPLES, INCREMENTS THE BRADDR ARRAY

```

```

48. C   INPNT -INTEGER, POINTS TO TOKENS IN THE INPUT VECTOR
49. C   INPUT -INTEGER ARRAY, LENGTH 100, FOR PASSING TOKENS TO THIS
50. C       MODULE
51. C   OUTPUT-INTEGER ARRAY, LENGTH 100, FOR RETURNING POSTFIX
52. C       ARITHMETIC EXPRESSION
53. C   PRTLVL-INTEGER, VARIABLE TO ALLOW PRINTING OF EXTRA INFORMATION
54. C       WHICH SHOULD BE HELPFUL IN MODIFYING OR DEBUGGING THE
55. C       SYSTEM
56. C   REMBER-INTEGER, SAVES INTERNAL ADDRESS OF TRIPLE ARRAY TO INSERT
57. C       ADDRESS FROM BLDTRP SUBROUTINE
58. C   TRIPLE-INTEGER ARRAY, SIZE 80 BY 3, FOR RETURNING ORDERED TRIPLES
59. C   TRPROW-INTEGER, POINTER TO ROWS OF THE ARRAY OF ORDERED TRIPLES
60. C
61. C   INTEGER BRADDR(20), INPNT, INPUT(100), NUMBX, OUTPUT(100),
62. C   &      PRTLVL, REMBER, TRIPLE(80,3), TRPROW
63. C
64. C LOCAL VARIABLES
65. C
66. C   ARRPT-INTEGER, POINTS TO ENTRIES IN THE PRECEDENCE ARRAY
67. C   CPAREN-INTEGER, POSSIBLE INPUT TOKEN OF CLOSING PARENTHESIS
68. C   ELSE -INTEGER, POSSIBLE INPUT TOKEN 142409
69. C   MINUS -INTEGER, POSSIBLE INPUT TOKEN 50006
70. C   NOSAVE-INTEGER, POSSIBLE INPUT TOKEN 144002
71. C   OPAREN-INTEGER, POSSIBLE INPUT TOKEN OF OPENING PARENTHESIS
72. C   OUTPNT-INTEGER, POINTS TO TOKENS IN OUTPUT VECTOR
73. C   PLUS -INTEGER, POSSIBLE INPUT TOKEN 50005
74. C   PREC -INTEGER ARRAY, SIZE 2 BY 18, HOLDS PRECEDENCE OF
75. C       ARITHMETIC OPERATORS
76. C   PRECS -INTEGER, PRECEDENCE OF TOKEN AT TOP OF THE STACK
77. C   PRECT -INTEGER, PRECEDENCE OF TOKEN CURRENTLY UNDER CONSIDERATION
78. C   STACK -INTEGER ARRAY, LENGTH 30, STACK FOR PROCESSING OPERATORS
79. C   STKPNT-INTEGER, POINTER TO NEXT EMPTY POSITION ON THE STACK
80. C   TEMP -INTEGER, USED FOR DETERMINING IF TOKEN IS A VARIABLE OR
81. C       CONSTANT
82. C   THEN -INTEGER, POSSIBLE INPUT TOKEN 142408
83. C   UMINUS-INTEGER, UNARY MINUS MULTIPLYING FACTOR
84. C
85. C   INTEGER ARRPT, CPAREN, ELSE, END, IF, MINUS, OPAREN, OUTPNT,
86. C   &      PLUS, PREC(2,18), PRECS, PRECT, STACK(30), STKPNT, TEMP,
87. C   &      THEN, UMINUS
88. C   DATA CPAREN/50004/, ELSE/142409/, END/50030/, IF/140805/,
89. C   &      MINUS/50006/, NOSAVE/144002/, OPAREN/50003/, PLUS/50005/,
90. C   &      THEN/142408/
91. C   DATA PREC/50003, 0, 50004, 1, 50005, 3, 50006, 3, 50007, 4,
92. C   &      50008, 4, 50009, 5, 50010, 2, 60001, 7, 60002, 7, 60003, 7,
93. C   &      60004, 7, 60005, 7, 60006, 7, 60007, 7, 60008, 7, 60009, 7,
94. C   &      60010, 7/
95. C
96. C * * * * * EXECUTABLE CODE * * * * *
97. C
98. 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. C      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 500
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.         IF (INPUT(INPNT).NE.OPAREN) GO TO 50
133. C      THEN PUSH TOKEN ONTO STACK
134.         STACK(STKPNT) = INPUT(INPNT)
135.         STKPNT = STKPNT + 1
136.         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.         & .AND.(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      RESET UNARY MINUS FACTOR TO POSITIVE
147.         UMINUS = +1
148.         OUTPNT = OUTPNT + 1
149.         GO TO 20

```

```

150. C   END OF IF.
151. C
152. 60  CONTINUE
153. C   IF TOKEN IS '+' OR '-'
154.     IF((INPUT(INPNT).NE.PLUS).AND.(INPUT(INPNT).NE.MINUS)) GO TO 80
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 ARRPT = 1,18
191. C     100   IF (STACK(STKPNT - 1).EQ.PREC(1,ARRPT)) GO TO 110
192. C   END OF REPEAT.
193. C   110   PRECS = PREC(2,ARRPT)
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 ARRPT = 1,18
200. C     120   IF (INPUT(INPNT).EQ.PREC(1,ARRPT)) GO TO 130

```

```

201. C END OF REPEAT.
202. 130 PRECT = PREC(2,ARRPNT)
203. C
204. 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.
209. STKPNT = STKPNT - 1
210. OUTPUT(OUTPNT) = STACK(STKPNT)
211. OUTPNT = OUTPNT + 1
212. GO TO 80
213. 140 CONTINUE
214. C ELSE IF TOKEN = ')'
215. IF (INPUT(INPNT).NE.CPAREN) GO TO 150
216. C DISREGARD BOTH THE TOKEN AND '(', AND GET NEXT INPUT TOKEN
217. STKPNT = STKPNT - 1
218. GO TO 20
219. C END OF IF.
220. 150 CONTINUE
221. C ELSE PUSH TOKEN ONTO STACK
222. STACK(STKPNT) = INPUT(INPNT)
223. STKPNT = STKPNT + 1
224. GO TO 20
225. C END OF IF.
226. C
227. 500 CONTINUE
228. C PLACE AN IDENTIFIALBE DELIMITER ONTO THE STACK
229. OUTPUT(OUTPNT) = END
230. C
231. C IF PRTLVL = 0 WRITE OUT THE POSTFIX ASSIGNMENT CLAUSE
232. IF (PRTLVL.EQ.0) WRITE (6,505) (OUTPUT(I), I = 1, OUTPNT)
233. C END OF IF
234. C
235. CALL BLDTRP (BRADDR, NUMBx, OUTPNT, OUTPUT, REMBER, TRIPLE,
236. & TRPROW)
237. 505 FORMAT (' ', 16(I7, 1X))
238. RETURN
239. END
240. C#EJECT

```

```

1.          SUBROUTINE BLDTRP (BRADDR, NUMBX, OUTPNT, OUTPUT, RENBER, TRIPLE,
2.          & TRPROW)
3.          C
4.          C XXXX X      XXXX XXXXX XXXXX XXXXX
5.          C X  X X      X  X  X  X  X X X  X
6.          C XXXX X      X  X  X  XXXXX XXXXX
7.          C X  X X      X  X  X  X  X X  X
8.          C XXXX XXXXX XXXX  X  X  X X
9.          C
10.         C * * * * * FUNCTION OF MODULE * * * * *
11.         C
12.         C   BLDTRP ACCEPTS THE VECTOR OF ASSIGNMENT CLAUSE TOKENS WHICH ARE
13.         C   IN POSTFIX ORDER AS INPUT AND PROCESSES THEM INTO AN ARRAY OF
14.         C   ORDERED TRIPLES WHICH ARE OUTPUT.
15.         C
16.         C * * * * * PROCEDURE FOLLOWED * * * * *
17.         C
18.         C   THE VECTOR OF ASSIGNMENT CLAUSE TOKENS IS PARSED UNTIL AN
19.         C   OPERATOR IS FOUND WHICH IS PLACED IN THE FIRST COLUMN OF THE
20.         C   ARRAY OF ORDERED TRIPLES. THEN THE MODULE BACKSPACES IN THE
21.         C   ASSIGNMENT CLAUSE VECTOR TO FIND THE OPERAND(S) FOR THAT
22.         C   OPERATOR. ONE OPERAND IS REQUIRED FOR FUNCTION OPERATORS AND
23.         C   TWO OPERANDS FOR THE COMMON BINARY OPERATORS. THE OPERAND(S)
24.         C   ARE PLACED IN THE SECOND AND THIRD COLUMNS OF THE ORDERED
25.         C   TRIPLE ARRAY. THE INPUT VECTOR POSITION OF THE OPERAND OF
26.         C   UNARY OPERATORS AND THE SECOND OPERAND OF BINARY OPERATORS ARE
27.         C   REPLACED WITH THE 'POP' OPERATOR (113) WHICH WILL POP THE TOP
28.         C   INTERIM RESULT FROM A STACK DURING EXECUTION. THE INPUT VECTOR
29.         C   LOCATION OF OPERATORS AND FIRST OPERAND OF THE BINARY OPERATORS
30.         C   ARE ASSIGNED A VALUE OF ZERO TO INDICATE THE OPERATOR OR OPERAND
31.         C   HAS BEEN MOVED TO THE TRIPLE ARRAY. THE ARRAY OF ORDERED
32.         C   TRIPLES IS RETURNED TO THE CALLING MODULE.
33.         C
34.         C * * * * * VARIABLE DECLARATIONS * * * * *
35.         C
36.         C PARAMETERS
37.         C
38.         C   BRADDR-INTEGER ARRAY, LENGTH 20, HOLDS INTERNAL ADDRESS OF TRIPLE
39.         C   ROWS WHICH WILL GET UNCONDITIONAL BRANCH TO END OF TRIPLES
40.         C   NUMBX -INTEGER, COUNTS THE NUMBER OF PLACES FOR UNCONDITIONAL
41.         C   BRANCH TO END OF TRIPLES, INCREMENTS THE BRADDR ARRAY
42.         C   OUTPNT-INTEGER, BRINGS IN POINTER TO LAST TOKEN IN OUTPUT VECTOR;
43.         C   ALSO USED AS POINTER IN OUTPUT VECTOR
44.         C   OUTPUT-INTEGER ARRAY, LENGTH 100, FOR PASSING POSTFIX
45.         C   ASSIGNMENT CLAUSE
46.         C   RENBER-INTEGER, SAVES INTERNAL ADDRESS OF TRIPLE ARRAY TO INSERT

```

```

47. C ADDRESS FROM BLDTRP SUBROUTINE
48. C TRIPLE-INTEGERS ARRAY, SIZE 80 BY 3, FOR RETURNING ORDERED TRIPLES
49. C TRPROW-INTEGERS, POINTER TO ROWS OF THE ARRAY OF ORDERED TRIPLES
50. C
51. C INTEGER BRADDR(20), NUMBX, OUTPNT, OUTPUT(100), REMBER,
52. C & TRIPLE(80,3), TRPROW
53. C
54. C LOCAL VARIABLES
55. C
56. C BEGIN -INTEGER, KEEPS POSITION IN OUTPUT VECTOR WHERE LAST
57. C OPERAND WAS FOUND
58. C END -INTEGER, DELIMITER FOR POSTFIX STACK
59. C POP -INTEGER, OPERATOR TO POP THE TOP INTERIM ORDERED TRIPLE
60. C RESULT FROM A STACK
61. C STAKLN-INTEGERS, STORES LENGTH OF THE POSTFIX STACK
62. C TEMP -INTEGER, USED FOR DETERMINING IF TOKEN IS A VARIABLE OR
63. C CONSTANT
64. C
65. C INTEGER BEGIN, BRANCH, END, POP, STAKLN, TEMP
66. C DATA BRANCH/50017/, END/50030/, POP/50019/
67. C
68. C * * * * * EXECUTABLE CODE * * * * *
69. C
70. C STAKLN = OUTPNT
71. C BEGIN = 1
72. C OUTPNT = 1
73. C
74. 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
81. 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
93. C 40 BEGIN = OUTPNT + 1
94. C TRPROW = TRPROW + 1
95. C PLACE OPERATOR FOUND IN FIRST COLUMN OF THE TRIPLE ARRAY
96. C TRIPLE(TRPROW,1) = OUTPUT(OUTPNT)
97. C BLANK OUT THE POSITION OF THE OPERATOR TOKEN IN THE POSTFIX ARRAY

```



```

98.          OUTPUT(OUTPNT) = 0
99.          C      REPEAT
100.         C      BACKSPACE IN OUTPUT VECTOR UNTIL OPERAND OR PREVIOUS TRIPLE
101.         C      INTERIM RESULT IS FOUND
102.         C      DO 50 I = 1, STAKLN
103.         C      OUTPUT = OUTPUT - 1
104.         C      TEMP = IABS(OUTPUT(OUTPNT)/10000)
105.         C      IF ((TEMP.EQ. 2).OR.((TEMP.GE.7).AND.(TEMP.LE.9)).OR.
106.         &      (TEMP.GE.15).OR.(OUTPUT(OUTPNT).EQ.POP)) GO TO 70
107.         50      CONTINUE
108.         C      END OF REPEAT
109.         C
110.         70      CONTINUE
111.         C      PLACE FIRST OPERAND IN THIRD COLUMN OF TRIPLE ARRAY
112.         C      TRIPLE(TRPROW,3) = OUTPUT(OUTPNT)
113.         C      STORE 'POP' OPERATOR IN THE POSTFIX STRING
114.         C      OUTPUT(OUTPNT) = POP
115.         C      IF OPERATOR WAS A UNARY OPERATOR (FUNCTION), THEN GET NEXT TOKEN
116.         C      IF ((TRIPLE(TRPROW,1))/10000.EQ.6) GO TO 10
117.         C      ELSE GET THE SECOND OPERAND
118.         C      REPEAT
119.         C      BACKSPACE IN OUTPUT VECTOR UNTIL OPERAND OR PREVIOUS TRIPLE
120.         C      INTERIM RESULT IS FOUND
121.         C      DO 90 I = 1, STAKLN
122.         C      OUTPUT = OUTPUT - 1
123.         C      TEMP = IABS(OUTPUT(OUTPNT)/10000)
124.         C      IF ((TEMP.EQ. 2).OR.((TEMP.GE.7).AND.(TEMP.LE.9)).OR.
125.         &      (TEMP.GE.15).OR.(OUTPUT(OUTPNT).EQ.POP)) GO TO 100
126.         90      CONTINUE
127.         C      END OF REPEAT
128.         C      END OF IF
129.         C
130.         100     CONTINUE
131.         C      PLACE SECOND OPERAND IN SECOND COLUMN OF TRIPLE ARRAY
132.         C      TRIPLE(TRPROW,2) = OUTPUT(OUTPNT)
133.         C      BLANK OUT THE POSITION OF THE OPERAND TOKEN IN POSTFIX STRING
134.         C      OUTPUT(OUTPNT) = 0
135.         C      END OF IF
136.         C      GET THE NEXT TOKEN
137.         C      GO TO 10
138.         C      END OF REPEAT WHILE
139.         C
140.         500     CONTINUE
141.         C      IF A RELATIONAL EXPRESSION EXISTS IN THE ORDERED TRIPLES
142.         C      IF (REMBER.EQ.0) GO TO 600
143.         C      THEN INSERT THE ADDRESS OF THE NEXT TRIPLE ROW AS THE ADDRESS OF
144.         C      THE CONDITIONAL BRANCH ASSOCIATED WITH THE RELATIONAL
145.         C      EXPRESSION.
146.         C      TRIPLE(REMBER,3) = TRPROW + 2
147.         C      ZERO OUT PARAMETER
148.         C      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.          NUMBX = NUMBX + 1
156. C        SAVE THE INTERNAL TRIPLE ADDRESS FOR USE IN CTRIPS SUBROUTINE
157.          BRADDR(NUMBX) = TRPROW
158. C        END OF IF
159. C
160. 600      RETURN
161.          END
162. C$EJECT
```

```

1.          SUBROUTINE PSTFXS (INPUT, PRTLVL, TRIPLE, TRPROW)
2.          C
3.          C   XXXXX XXXXX XXXXX XXXXX X   X XXXXX
4.          C   X  X X      X  X      X X  X
5.          C   XXXXX XXXXX  X  XXXX  X  XXXXX
6.          C   X      X  X  X      X X      X
7.          C   X      XXXXX  X  X      X  X XXXXX
8.          C
9.          C * * * * * FUNCTION OF MODULE * * * * *
10.         C
11.         C   PSTFXS TAKES A BOOLEAN EXPRESSION COMPOSED OF LOGICAL OPERATORS
12.         C   WITH RELATIONAL EXPRESSION OPERANDS AS INPUT AND CONVERTS THEM
13.         C   INTO POSTFIX NOTATION WHICH IS OUTPUT.
14.         C
15.         C * * * * * PROCEDURE FOLLOWED * * * * *
16.         C
17.         C   IF THE TOKEN IS AN END DELIMITER OR A REMARK, THE TOKENS IN THE
18.         C   STACK ARE OUTPUT UNTIL THE STACK IS EMPTY. IF THE TOKEN IS '(',
19.         C   IT IS PUSHED ONTO THE STACK. IF THE TOKEN IS A MINUS SIGN A
20.         C   CHECK IS MADE TO DETERMINE IF IT IS A UNARY MINUS. IF IT IS
21.         C   A UNARY MINUS, THE CONSTANT OR VARIABLE TOKEN ON WHICH IT
22.         C   OPERATES IS CHANGED TO A NEGATIVE VALUE. IF THE TOKEN IS AN
23.         C   OPERAND (OR PORTION OF AN OPERAND) OF A LOGICAL OPERATOR (.AND.,
24.         C   .OR.) TO THE OUTPUT. IF THE TOKEN IS AN OPERATOR AND THE STACK IS
25.         C   EMPTY, IT IS PUSHED ONTO THE STACK. IF THE STACK IS NOT EMPTY,
26.         C   THE TOKEN IS COMPARED WITH THE TOKEN AT THE TOP OF THE STACK
27.         C   AND IF THE STACK TOKEN IS OF EQUAL OR HIGHER PRECEDENCE IT IS
28.         C   MOVED TO OUTPUT. OTHERWISE IF THE TOKEN IS ')', BOTH IT AND
29.         C   THE '(' FROM THE STACK ARE DISREGARDED. FINALLY IF NONE OF THE
30.         C   ABOVE CONDITIONS ARE TRUE, THE TOKEN IS PUSHED ONTO THE STACK
31.         C   AND THE NEXT OPERATOR FETCHED. OPERATOR ORDER-OF-PRECEDENCE FROM
32.         C   LOWEST TO HIGHEST IS OPENING PARENTHESIS '(', CLOSING PARENTHESIS
33.         C   ')', LOGICAL OR '.OR.', AND LOGICAL AND '.AND.'.
34.         C
35.         C * * * * * SUBROUTINES CALLED * * * * *
36.         C
37.         C   TRIPLS-MOVES POSTFIX-ORDERED TOKENS INTO ARRAY OF ORDERED TRIPLES
38.         C
39.         C * * * * * VARIABLE DECLARATIONS * * * * *
40.         C
41.         C   PARAMETERS
42.         C
43.         C   INPUT -INTEGER ARRAY, LENGTH 100, FOR PASSING TOKENS TO THIS
44.         C           MODULE
45.         C   PRTLVL-INTEGER, VARIABLE TO ALLOW PRINTING OF EXTRA INFORMATION
46.         C           WHICH SHOULD BE HELPFUL IN MODIFYING OR DEBUGGING THE

```

```

47. C          SYSTEM
48. C TRIPLE-INTEGERS ARRAY, SIZE 80 BY 3, FOR RETURNING ORDERED TRIPLES
49. C TRPROW-INTEGERS, POINTER TO ROWS OF THE ARRAY OF ORDERED TRIPLES
50. C
51. C          INTEGER INPUT(100), PRTLVL, TRIPLE(80,3), TRPROW
52. C
53. C LOCAL VARIABLES
54. C
55. C ARRPNL-INTEGERS, POINTS TO ENTRIES IN THE PRECEDENCE ARRAY
56. C CPAREN-INTEGERS, POSSIBLE INPUT TOKEN OF CLOSING PARENTHESIS
57. C IN      -INTEGERS, POSSIBLE INPUT TOKEN 140806
58. C INPNT -INTEGERS, POINTS TO TOKENS IN THE INPUT VECTOR
59. C MINUS -INTEGERS, POSSIBLE INPUT TOKEN 50006
60. C NOSAVE-INTEGERS, POSSIBLE INPUT TOKEN 144002
61. C NOT    -INTEGERS, POSSIBLE INPUT TOKEN 141607
62. C OPAREN-INTEGERS, POSSIBLE INPUT TOKEN OF OPENING PARENTHESIS
63. C OUTPNT-INTEGERS, POINTS TO TOKENS IN OUTPUT VECTOR
64. C OUTPUT-INTEGERS ARRAY, LENGTH 100, FOR PASSING POSTFIX
65. C      BOOLEAN EXPRESSION
66. C PLUS  -INTEGERS, POSSIBLE INPUT TOKEN 50005
67. C PREC  -INTEGERS ARRAY, SIZE 2 BY 4, HOLDS PRECEDENCE OF
68. C      ARITHMETIC OPERATORS
69. C PRECS -INTEGERS, PRECEDENCE OF TOKEN AT TOP OF THE STACK
70. C PRECT -INTEGERS, PRECEDENCE OF TOKEN CURRENTLY UNDER CONSIDERATION
71. C STACK -INTEGERS ARRAY, LENGTH 30, STACK FOR PROCESSING OPERATORS
72. C STKPNL-INTEGERS, POINTER TO NEXT EMPTY POSITION ON THE STACK
73. C TEMP  -INTEGERS, USED FOR DETERMINING IF TOKEN IS A VARIABLE OR
74. C      CONSTANT
75. C UMINUS-INTEGERS, UNARY MINUS MULTIPLYING FACTOR
76. C
77. C          INTEGER ARRPNL, CPAREN, END, IN, INPNT, MINUS, NOSAVE, NOT,
78. C      &      OPAREN, OUTPNT, OUTPUT(100), PLUS, PREC(2,4), PRECS, PRECT,
79. C      &      STACK(30), STKPNL, TEMP, UMINUS
80. C
81. C CONSTANTS
82. C
83. C DATA CPAREN/50004/, END/50030/, IN/140806/, MINUS/50006/,
84. C      &      NOSAVE/144002/, NOT/141607/, OPAREN/50003/, PLUS/50005/,
85. C      &      PREC/50003, 0, 50004, 1, 40001, 9, 40002, 8/
86. C
87. C * * * * * EXECUTABLE CODE * * * * *
88. C
89. C          INPNT = 0
90. C          OUTPNT = 1
91. C          STKPNL = 1
92. C          UMINUS = +1
93. C
94. C      20  CONTINUE
95. C INCREMENT THE INPUT ARRAY POINTER
96. C          INPNT = INPNT + 1
97. C

```

68

```

98. C IF THE TOKEN DELIMITS THE BOOLEAN EXPRESSION (NOSAVE, REMARK)
99. C IF ((INPUT(INPNT).NE.NOSAVE).AND.
100. & ((INPUT(INPNT)/10000).NE.1)) GO TO 40
101. C THEN
102. C WHILE THE STACK IS NOT EMPTY
103. C DO 30 I = 1,30
104. C IF (STKPNT.EQ.1) GO TO 500
105. C STKPNT = STKPNT - 1
106. C MOVE STACK TO OUTPUT
107. C 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. C IF (INPUT(INPNT).NE.OPAREN) GO TO 50
115. C THEN PUSH TOKEN ONTO STACK
116. C STACK(STKPNT) = INPUT(INPNT)
117. C STKPNT = STKPNT + 1
118. C 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. C TEMP = INPUT(INPNT)/10000
125. C 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. C OUTPUT(OUTPNT) = INPUT(INPNT) * UMINUS
130. C END OF IF
131. C RESET UNARY MINUS FACTOR TO POSITIVE
132. C UMINUS = +1
133. C OUTPNT = OUTPNT + 1
134. C GO TO 20
135. C
136. C IF TOKEN IS '+' OR '-'
137. 60 CONTINUE
138. C 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. C 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. C TEMP = INPUT(INPNT - 1)/10000
146. C IF (((TEMP.NE.5).AND.(TEMP.NE.6)).OR.
147. & (INPUT(INPNT-1).EQ.CPAREN)) GO TO 80
148. C END OF IF.

```

```

149. C END OF IF.
150. C
151. 70 CONTINUE
152. C IF TOKEN IS UNARY PLUS DISREGARD AND GET NEXT TOKEN
153. IF (INPUT(INPNT).EQ.PLUS) GO TO 20
154. C END OF IF
155. C IF TOKEN IS UNARY MINUS CHANGE SIGN OF UNARY MINUS FACTOR
156. UMINUS = -UMINUS
157. C RETURN FOR NEXT TOKEN
158. GO TO 20
159. C END OF IF.
160. C
161. 80 CONTINUE
162. C IF STACK IS EMPTY
163. IF (STKPNT.NE.1) GO TO 90
164. C THEN PUSH TOKEN ONTO STACK
165. STACK(STKPNT) = INPUT(INPNT)
166. STKPNT = STKPNT + 1
167. GO TO 20
168. C END OF IF.
169. C
170. 90 CONTINUE
171. C ASSIGN PRECEDENCE OF OPERATOR AT TOP OF STACK TO 'PRECS'
172. C REPEAT
173. C COMPARE TOKEN AT TOP OF STACK WITH PRECEDENCE ARRAY
174. DO 100 ARRPT = 1,4
175. 100 IF (STACK(STKPNT - 1).EQ.PREC(1,ARRPT)) GO TO 110
176. C END OF REPEAT.
177. 110 PRECS = PREC(2,ARRPT)
178. C
179. C ASSIGN PRECEDENCE OF OPERATOR CURRENTLY UNDER CONSIDERATION TO
180. C 'PRECT'
181. C REPEAT
182. C COMPARE INPUT TOKEN WITH PRECEDENCE ARRAY
183. DO 120 ARRPT = 1,4
184. IF (INPUT(INPNT).EQ.PREC(1,ARRPT)) GO TO 130
185. 120 CONTINUE
186. C END OF REPEAT.
187. 130 PRECT = PREC(2,ARRPT)
188. C
189. C COMPARE PRECEDENCE OF TOKEN AT TOP OF STACK WITH INPUT TOKEN
190. C IF PRECS >= PRECT
191. IF (PRECS.LT.PRECT) GO TO 140
192. C THEN MOVE TOP OF STACK OPERATOR TO OUTPUT AND COMPARE TOKEN
193. C OPERATOR TO NEXT OPERATOR IN STACK.
194. STKPNT = STKPNT - 1
195. OUTPUT(OUTPNT) = STACK(STKPNT)
196. OUTPNT = OUTPNT + 1
197. GO TO 80
198. 140 CONTINUE
199. 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 IDENTIFIALBE 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
```

```

1.          SUBROUTINE TRIPLS (OUTPNT, OUTPUT, PRTLVL, TRIPLE, TRPROW)
2.          C
3.          C   XXXXX XXXXX XXX XXXXX X   XXXXX
4.          C   X  X  X  X  X  X  X   X
5.          C   X  XXXXX X  XXXXX X   XXXXX
6.          C   X  X  X  X  X  X   X
7.          C   X  X  X  XXX X   XXXXX XXXXX
8.          C
9.          C * * * * * FUNCTION OF MODULE * * * * *
10.         C
11.         C   TRIPLS ACCEPTS THE VECTOR OF BOOLEAN EXPRESSION TOKENS WHICH ARE
12.         C   IN POSTFIX ORDER AS INPUT AND PROCESSES THEM INTO AN ARRAY OF
13.         C   ORDERED TRIPLES WHICH ARE OUTPUT.
14.         C
15.         C * * * * * PROCEDURE FOLLOWED * * * * *
16.         C
17.         C   THE VECTOR OF BOOLEAN EXPRESSION TOKENS IS PARSED AND THE
18.         C   TOKENS IDENTIFIED. LOGICAL OPERATORS (.AND., .OR.) ARE MOVED
19.         C   TO COLUMN ONE OF THE TRIPLE ARRAY. THE OPERANDS 'IN GROUP/
20.         C   MODULE' OR 'NOT IN GFOUP/MODULE' ARE MOVED DIRECTLY TO A TRIPLE.
21.         C   THE OPERANDS OF RELATIONAL EXPRESSIONS ARE PLACED IN COLUMNS TWO
22.         C   AND THREE OF THE TRIPLE WITH THE 'SUB' COMMAND IN COLUMN ONE.
23.         C   A CONDITIONAL AND UNCONDITIONAL BRANCH ARE PLACED IN THE TRIPLES
24.         C   TO CAUSE A '1' TO BE PUSHED ONTO THE STACK IF THE RELATIONAL
25.         C   EXPRESSION IS TRUE OR A '0' IF IT IS FALSE. THE FINAL TRIPLE
26.         C   IS GIVEN THE COMMAND 'SETMSK' WHICH WILL POP THE FINAL RESULT
27.         C   OF THE BOOLEAN EXPRESSION FROM THE STACK AND SET A MASK TO '1'
28.         C   IF TRUE OR '0' IF FALSE. THE ARRAY OF ORDERED TRIPLES IS
29.         C   RETURNED TO THE CALLING MODULE.
30.         C
31.         C * * * * * VARIABLE DECLARATIONS * * * * *
32.         C
33.         C   PARAMETERS
34.         C
35.         C   OUTPNT-INTEGER, BRINGS IN POINTER TO LAST TOKEN IN OUTPUT VECTOR;
36.         C   ALSO USED AS POINTER IN OUTPUT VECTOR
37.         C   OUTPUT-INTEGER ARRAY, LENGTH 100, FOR PASSING POSTFIX
38.         C   BOOLEAN EXPRESSION
39.         C   PRTLVL-INTEGER, VARIALBE TO ALLOW PRINTING OF EXTRA INFORMATION
40.         C   WHICH SHOULD BE HELPFUL IN MODIFYING OR DEBUGGING THE
41.         C   SYSTEM
42.         C   TRIPLE-INTEGER ARRAY, SIZE 80 BY 3, FOR RETURNING ORDERED TRIPLES
43.         C   TRPROW-INTEGER, POINTER TO ROWS OF THE ARRAY OF ORDERED TRIPLES
44.         C
45.         C   INTEGER OUTPNT, OUTPUT(100), PRTLVL, TRIPLE(80,3), TRPROW
46.         C

```



```

47. C LOCAL VARIABLES
48. C
49. C   BEGIN -INTEGER, KEEPS POSITION IN OUTPUT VECTOR WHERE LAST
50. C       OPERAND WAS FOUND
51. C   END   -INTEGER, DELIMITER FOR POSTFIX STACK
52. C   NOT   -INTEGER, POSSIBLE INPUT TOKEN 141607
53. C   POP   -INTEGER, OPERATOR TO POP THE TOP INTERIM ORDERED TRIPLE
54. C       RESULT FROM A STACK
55. C   RELOPS-INTEGER ARRAY, SIZE 2 BY 6, HOLDS RELATIONAL OPERATOR AND
56. C       BRANCH CODES. THE OPERATOR CODES INSERTED INTO THE ORDERED
57. C       TRIPLE ARRAY ARE:
58. C           INPUT      RELATIONAL      BRANCH      BRANCH
59. C           TOKEN      OPERATOR      CONDITION   CODE
60. C           30001      .EQ.      BNZ         50011
61. C           30002      .NE.      BZ          50012
62. C           30003      .GT.      BMZ         50013
63. C           30004      .LT.      BPZ         50014
64. C           30005      .GE.      BP          50015
65. C           30006      .LE.      BM          50016
66. C   SAVE  -INTEGER, SAVE THE ADDRESS OF OPERAND OF LOGICAL OPERATOR
67. C   TEMP  -INTEGER, USED FOR DETERMINING IF TOKEN IS A VARIABLE OR
68. C       CONSTANT
69. C
70. C   INTEGER ADD, AND, BEGIN/1/, BNZ, BRANCH, BOOLOP, END, OR, POP,
71. C   & RELOPS(2,6), SAVE, SELECT, SETMSK, SUB, TEMP
72. C   DATA ADD/50018/, AND/40001/, BNZ/50011/, BRANCH/50017/,
73. C   & END/50030/, IN/140806/, NOT/141607/, OR/40002/, POP/50019/,
74. C   & RELOPS/30001, 50011, 30002, 50012, 30003, 50013, 30004, 50014,
75. C   & 30005, 50015, 30006, 50016/, SELECT/50026/, SETMSK/50020/,
76. C   & SUB/50006/
77. C
78. C * * * * * EXECUTABLE CODE * * * * *
79. C
80. C   OUTPNT = 0
81. C   10   CONTINUE
82. C       OUTPNT = OUTPNT + 1
83. C       TRPROW = TRPROW + 1
84. C   C IF THE TOKEN IS END DELIMITER THEN FINALIZE THE TRIPLE ARRAY
85. C       IF (OUTPUT(OUTPNT).EQ.END) GO TO 400
86. C   C END OF IF
87. C
88. C   C IF THE TOKEN IS '.OR.' PLACE IT IN THE TRIPLE ARRAY
89. C       IF (OUTPUT(OUTPNT).NE.OR) GO TO 30
90. C       TRIPLE(TRPROW,1) = 50022
91. C       GO TO 10
92. C   C END OF IF
93. C
94. C   30   CONTINUE
95. C   C IF THE TOKEN IS '.AND.' PLACE IT IN THE TRIPLE ARRAY
96. C       IF (OUTPUT(OUTPNT).NE.AND) GO TO 50
97. C       TRIPLE(TRPROW,1) = 50021

```

```

98.          GO TO 10
99.    C     END OF IF
100.   C
101.   50    CONTINUE
102.   C     IF THE TOKEN IS 'IN' PLACE 'IN' AND THE GROUP IDENTIFIER IN THE
103.   C     TRIPLE ARRAY
104.         IF (OUTPUT(OUTPNT).NE.IN) GO TO 70
105.         TRIPLE(TRPROW,1) = 50023
106.         TRIPLE(TRPROW,3) = OUTPUT(OUTPNT + 1)
107.         OUTPNT = OUTPNT + 1
108.         GO TO 10
109.   C     END OF IF
110.   C
111.   70    CONTINUE
112.   C     IF THE TOKEN IS 'NOT' PLACE 'NOT IN' AND THE GROUP IDENTIFIER IN
113.   C     THE TRIPLE ARRAY
114.         IF (OUTPUT(OUTPNT).NE.NOT) GO TO 100
115.         TRIPLE(TRPROW,1) = 50024
116.         TRIPLE(TRPROW,3) = OUTPUT(OUTPNT + 2)
117.         OUTPNT = OUTPNT + 2
118.         GO TO 10
119.   C     END OF IF
120.   C
121.   C     ANY TOKENS WHICH FILTER DOWN TO THIS POINT ARE RELATIONAL EXPRESSION
122.   C     OPERANDS OF THE LOGICAL OPERATORS. PLACE THE SUBTRACT OPERATOR
123.   C     IN COLUMN ONE AND THE TWO VARIABLES OR CONSTANTS IN COLUMNS
124.   C     TWO AND THREE OF THE TRIPLE ARRAY
125.   100   TRIPLE(TRPROW,1) = SUB
126.         TRIPLE(TRPROW,2) = OUTPUT(OUTPNT)
127.         TRIPLE(TRPROW,3) = OUTPUT(OUTPNT + 2)
128.         TRPROW = TRPROW + 1
129.         OUTPNT = OUTPNT + 1
130.   C
131.   C     OBTAIN THE PROPER CODE FOR THE RELATIONAL OPERATOR
132.   C     REPEAT
133.   C     COMPARE INPUT TOKEN WITH RELATIONAL OPERATORS UNTIL MATCH
134.         DO 140 I = 1,6
135.   140   IF (OUTPUT(OUTPNT).EQ.RELOPS(1,I)) GO TO 150
136.   C     END OF REPEAT
137.   C
138.   150   CONTINUE
139.   C     PLACE THE PROPER BRANCH CODE, BASED ON THE RELATIONAL OPERATOR,
140.   C     IN COLUMN ONE OF THE NEXT TRIPLE ROW
141.         TRIPLE(TRPROW,1) = RELOPS(2,I)
142.   C     PLACE 'POP' OPERATOR IN COLUMN TWO OF THE TRIPLE ARRAY
143.         TRIPLE(TRPROW,2) = POP
144.         TRIPLE(TRPROW,3) = TRPROW + 3
145.   180   CONTINUE
146.         TRPROW = TRPROW + 1
147.   C     PLACE 'ADD 0 1' IN TRIPLE ARRAY FOR TRUE CONDITION
148.         TRIPLE(TRPROW,1) = ADD

```

```
149.          TRIPLE(TRPROW,2) = 0
150.          TRIPLE(TRPROW,3) = 1
151.          TRPROW = TRPROW + 1
152.    C    PLACE 'UNCONDITIONAL BRANCH' AND TRIPLE ADDRESS IN TRIPLE ARRAY
153.          TRIPLE(TRPROW,1) = BRANCH
154.          TRIPLE(TRPROW,3) = TRPROW + 2
155.          TRPROW = TRPROW + 1
156.    C    PLACE 'ADD 0 0' IN TRIPLE ARRAY
157.          TRIPLE(TRPROW,1) = ADD
158.          TRIPLE(TRPROW,2) = TRIPLE(TRPROW,3) = 0
159.          OUTPNT = OUTPNT + 1
160.          GO TO 10
161.    C
162.    C    FINALIZE THE TRIPLE ARRAY FOR BOOLEAN EXPRESSIONS. SETMSK WILL
163.    C    CAUSE THE EXECUTION PHASE TO SET A FLAG ACCORDING TO THE
164.    C    RESULTS OF A TRUE OR FALSE BOOLEAN EXPRESSION.
165.    400    CONTINUE
166.          TRIPLE(TRPROW,1) = SETMSK
167.          TRIPLE(TRPROW,2) = POP
168.          TRIPLE(TRPROW,3) = SELECT
169.          TRPROW = TRPROW + 1
170.    C
171.    C    IF PRTLVL = 0 WRITE OUT THE ORDERED TRIPLES
172.          IF (PRTLVL.NE.0) GO TO 700
173.    600    N = TRPROW
174.          DO 550 TRPROW = 1,N
175.    550    WRITE (6,510) (TRIPLE(TRPROW,J), J = 1, 3)
176.    510    FORMAT (' ', T10, I7, T25, I7, T40, I7)
177.    700    RETURN
178.          END
```

REFERENCES

1. Arden, Bruce W., Gallier, Bernard A., and Graham, Robert M., "An Algorithm for Translating Boolean Expressions," Journal of the Association for Computing Machinery, 222-239, Volume 9, Number 2, April 1962.
2. Department of Defense, Task Analysis COPAF Executive's Overview Guide.
3. Gries, David, Compiler Construction for Digital Computers, John Wiley & Sons, Inc., New York, 1971.
4. Manning, Rocky, "The Design and Implementation of an Interpreter for a Data Management System," OSRI Research Report, Department of Industrial Engineering, Texas A&M University, 1970.
5. Occupational Research Program, COPAF User's Guide, (will not be completed until approximately March 1980).
6. Rosen, Saul, Programming Systems and Languages, McGraw-Hill, New York, 1967.
7. SAS User's Guide, SAS Institute Inc, Raleigh, North Carolina, 1979.

DATE
FILMED
-18