A STRUCTURED PROGRAMMING PREPROCESSOR
FOR A VARIETY OF BASE LANGUAGES

by
Mary L. Love
B.S., Kansas State University, 1975

A MASTER'S REPORT
submitted in partial fulfillment of the requirements for the degree
MASTER OF SCIENCE

Department of Computer Science

KANSAS STATE UNIVERSITY
Manhattan, Kansas

1977

Approved by:

Virgil Walleentine
# TABLE OF CONTENTS

## I. Introduction
- Project Definition ........................................ 1
- Motivation ................................................... 2
- Organization of Remainder of Paper ....................... 3

## II. Techniques of Design and Implementation
- Design ....................................................... 5
- Implementation ............................................. 22

## III. User Information Overview
- Input ....................................................... 26
- Output ..................................................... 29
- Error Conditions ........................................... 31

## IV. Observations
- Good and Bad Aspects of Project .......................... 33
- Overall Appraisal of Direction ............................ 34
- Time, Effort, and Money Involved ......................... 35
- Elements to Be Changed .................................... 35
- Possible Further Developments ............................. 36

## Appendix
- I. Detailed Implementation Information .................. 41
- II. Detailed User Information ............................... 82
- III. Job Control Language Listings ......................... 97
- IV. Example Base Language Specifications ................ 101
- V. Preprocessor Source Code Listing ....................... 105
ILLEGIBLE DOCUMENT

THE FOLLOWING DOCUMENT(S) IS OF POOR LEGIBILITY IN THE ORIGINAL

THIS IS THE BEST COPY AVAILABLE
FIGURES

Figure II.1. Preprocessor and Language Translator Relationship . . 6
Figure II.2. Flow of Data in Preprocessor. . . . . . . . . . . . 18
Figure II.3. Modularity of the Preprocessor. . . . . . . . . . . . 20

TABLES

Table III.1. Prototype Parameters. . . . . . . . . . . . . . . . . . . . 27
Table III.2. Definitions Required for Token Recognition and
code Generation. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 28
Table A.I.1. Error Exits with Additional Information Provided. . . 5A
Table A.I.2. Scanning Routines . . . . . . . . . . . . . . . . . . . . . 14A
Table A.I.3. Production Language for Parser. . . . . . . . . . . . 31A
ACKNOWLEDGMENT

The author wishes to express her deep thanks to her major professor, Dr. Virgil Wallentine, and the other members of her committee, Drs. Linda Shapiro and William Hankley. A sincere thank you also goes to the other students in the Department of Computer Science who listened and gave suggestions and support which contributed toward the completion of this report.
SECTION I. INTRODUCTION

PROJECT DEFINITION

This report describes the design and implementation of a preprocessor which facilitates structured programming in a variety of base languages. The preprocessor is intended to allow and encourage the use of structured programming techniques. It attempts to achieve these goals by translating structured programming control structures, not currently implemented in a particular programming language, into semantically equivalent statements which are standard in the language. The preprocessor also includes a text-replacement macro facility which allows the user to provide additional control constructs.

The user provides three types of input to the preprocessor. The first type consists of information about the base language including prototypes of several simple types of statements, operators, and other syntactic information. The second type of input consists of macro definitions. The third type of input is a program written in an extended version of a programming language. The extended version of a language includes all the usual features of that language plus the structured programming control constructs made available by the preprocessor and any macros for which the user has provided definitions.

The output of the preprocessor is a program which can be translated by the usual compiler for the language which was being extended. This program is provided both in printed form and in a temporary data set which can be used as input to the language translator. The output program is logically equivalent to the input program which was written in the extended language.
MOTIVATION

The first and foremost incentive behind this project is to make the techniques and desirable features of structured programming [Dah72,Loc74] available to programmers despite the programming language they may use. While there currently exist a number of processors designed to provide structured programming control structures for only FORTRAN [One74,Hig75], the author's preprocessor may be used with a variety of programming languages. The current implementation works with the PL/I, FORTRAN, and SNOBOL programming languages.

There are several reasons for using a preprocessor to accomplish this task. First, a preprocessor can be built to accommodate multiple languages. Second, the language translator currently being used may be retained because the preprocessor outputs code equivalent to the code written in the extended language but which is standard in the base language. Since the user need not acquire a new translator, programmers do not have to learn an entirely new language, just simple modifications to the current one. Furthermore, old programs do not have to be recoded in a new language. Finally, a preprocessor such as described here is easy to use in conjunction with the existing language translator because the preprocessor's output can be input directly into the existing translator.

A new preprocessor was implemented instead of using an existing general-purpose macro processor [Bro67,Bro69] because the new preprocessor is intended to encourage the use of structured programming as well as make it available. To do this, eight control structures are implemented in the preprocessor, thus relieving the programming of the task of having to define them for use with a general-purpose macro processor. This
definition of control structures would be necessary once for each 
language being used, whereas they are already defined for all languages 
in the new preprocessor.

Since the author recognizes the fact that no one person can 
think of all the structured programming control structures which may 
possibly be desirable for use at one time or another, a text-replacement 
nacro facility [Sol74] has also been incorporated into the preprocessor. 
This facility allows the user to define macros which will aid in the 
programming function. Inclusion of the macro facility introduces the 
possibility that the preprocessor's goals of providing structured 
programming capabilities and encouraging their use may be undermined 
by a user who defines macros which conflict with structured programming 
principles. Because of this possibility a set of guidelines for 
macro definitions is provided in Appendix II. If the user follows 
these guidelines in writing macro definitions, the likelihood of 
violating structured programming principles in user-defined macros will 
be decreased.

The preprocessor is written in SNOBOL4 and run under the 
SPIRECL compiler [Pew71]. This language was chosen because 
the pattern-matching capability provided is well-suited to 
token recognition, a major part of this application.

ORGANIZATION OF REMAINDER OF PAPER

The remainder of this paper is organized into three sections.
The first section describes the techniques of design and implementation 
of the preprocessor. The second section presents an overview of 
information required to use the preprocessor. The third section is 
a summary of the project. The Appendix of the paper includes details
of both the implementation and use of the preprocessor, the Job Control Language statements required to run the preprocessor, example base language specifications, and a listing of the preprocessor source code.
SECTION II. TECHNIQUES OF DESIGN AND IMPLEMENTATION

DESIGN

Design of Processor

The project was intended to effect a mapping from an extended language to the corresponding base language. The extended version of a programming language consists of all the features normally found in the language plus the structured programming control constructs made available by the processor and macros for which the user has provided definitions. To effect this mapping for a variety of languages, a generalized tool, that is, one capable of processing different languages, is required. Since a preprocessor could be designed to accept multiple languages for processing, the first design decision was to use a preprocessor.

There were other factors supporting this decision. The preprocessor could output code which is standard in the base language. As a result, the user need not acquire a new translator and programmers need to learn only simple modifications to the language they currently use. Programs already in use would not have to be recoded in a new language. Finally, since the preprocessor's output could be input directly into the language translator, it would be easy to use with the existing translator. Figure II.1 illustrates the relationship between the preprocessor and the language translator.

The decision to implement a new preprocessor rather than use an existing general-purpose macro processor stemmed from the fact that one of the goals of the project is to encourage the use of structured programming techniques. If an existing macro processor were used, the
Figure II.1. Preprocessor and Language Translator Relationship.

user would have to define the extended constructs each time the macro processor was used. Furthermore, the extended constructs would have different definitions for each language. Selected structured programming control constructs could be implemented within a new preprocessor. Consequently, the need to redefine the constructs at each use and for each language would be eliminated.

Another design decision made to enable the project to be generalized was to use a table-driven structure parser. The parser was designed based on Production Language [Gri71]. Using a table-driven parser would enable the parser to recognize constructs in different languages because the tokens would be recognized by a scanner using the appropriate definitions.

Eight structured programming control structures were chosen for implementation in the preprocessor. Each of these eight was chosen
either because it receives support in the literature as being a structure necessary for structured programming \[Kel, Lec74, Led75, Weg75\] or because it is practical to use. The structures included are:

\begin{itemize}
  \item IF - THEN - ENDIF
  \item WHILE - ENDWHILE
  \item CASE OF - ENDCASE
  \item DOGROUP - ENDDOGROUP
  \item IF - THEN - ELSE - ENDIF
  \item REPEAT UNTIL - ENDREPEAT
  \item DO (iterative) - ENDDO
  \item QUITLOOP
\end{itemize}

The decision to include a text-replacement type macro facility was a difficult one to make. On the one hand it seemed obvious that no one person could think of all the structured programming control structures which may possibly be desirable for use at one time or another. However, it also appeared that to include such a facility would defeat the purpose of the project by allowing users to define any constructs they wished, whether or not those constructs conflicted with the principles of structured programming.

After much deliberation it was decided to include the macro facility. A set of guidelines is provided along with this facility for the user to follow in defining macros. The author feels that by following these guidelines, the user can write macro definitions which have little chance of violating structured programming principles.

A final consideration in the design of the preprocessor was keeping its development and testing time and effort to a minimum. A project of this size, written in a language which does not currently support many structured programming techniques, would certainly be unmanageable if written as one large program. Therefore, to keep the coding and testing as simple as possible, the preprocessor was designed
to be modular. The final version of the preprocessor contains fifty modules in addition to the driver routine.

Design of Execution

The preprocessor may perform one or two parses in processing the input program to produce the final output. If the user has provided any macro definitions, the preprocessor will make a second parse using the output of the first parse as its input. The second parse is required because users are allowed to use the extended constructs which are built into the preprocessor within macro definitions. This second parse will expand these occurrences of the extended constructs into standard code.

Although the main processing of the input program will be the same for all languages, there are some cases for specific languages which must be handled separately. For example, in SNOBOL a logical not operator requires post processing to make the output produced by the preprocessor legal in the standard language. In many other languages the logical not may be formed in a manner similar to an equal relation, that is, with an argument followed by the operator and a second argument, but utilizing a null first argument. In SNOBOL, however, post processing is required to eliminate the comma which appears illegally when a null first argument is used in a logical not relation. That is, the form \( \neg((\_,\text{arg2})) \) must be replaced by \( \neg((\text{arg2})) \). Special functions such as this are accomplished by doing post processing for each language which requires it. Currently, the post processing is written into the driver of the preprocessor. However, for generality
it could be provided as a call to an external routine. The appropriate routine would have to be linked with the preprocessor when it is loaded for execution.

Another function of the preprocessor is to insert declarations, if required, for any new variables it introduces. To allow declarations to be properly placed in a program, the user includes an indication of the appropriate location by inserting a special statement which will be replaced by declarations. If the declarations are not required, the special statement should be omitted and the prototype for a declaration statement should be blank.

Design of Extended Constructs

The code which is generated for each of the eight extended constructs is explained briefly in the following paragraphs. Along with each explanation is a simple example designed to aid the reader in visualizing the replacement strategy while eliminating the details of any particular language and the preprocessor's exact output.

IF - THEN - ENDIF: If the negation of the Boolean expression provided in the IF statement is true, a branch is taken to a unique label which will be placed on a null statement at the position of the ENDIF statement. The code following the THEN is inserted following the test of the negation.

```
IF bool THEN assign
  assign
ENDIF

 IF not bool then go to unilab_1
   unilab_1: null
```

IF - THEN - ELSE - ENDIF: If the negation of the Boolean expression provided in the IF statement is true, a branch is taken to a unique
label which will be placed on a null statement at the position of the ELSE. The code following the THEN is inserted following the test of the negation. An unconditional branch to a second unique label (unilab_{i+1}) is added at the end of this code. The second unique label will be placed on a null statement at the position of the ENDIF. The code following the ELSE is inserted after the unconditional branch to the second unique label.

IF bool THEN assign
    if not bool then go to unilab_{i}
    assign
    go to unilab_{i+1}
ELSE branch
    unilab_{i}; null
    branch
    unilabel_{i+1}; null
ENDIF

DOGROUP - ENDDOxffff: No replacement code is generated for this construct. The code necessary to delineate the code between the DOGROUP and ENDDOxffff statements as a separate entity is included within the replacement code for the other constructs.

DOGROUP
    assign
    call
ENDDOxffff

DO (iterative) - ENDDO: The four elements extracted from the DO statement are the index variable (INDEX), the beginning value (BEGIN), the ending value (END), and the increment value (INCREMENT). Three statements are inserted at the position of the DO statement. The first statement sets a unique variable (univar_{i}) equal to END minus INCREMENT. The second sets INDEX equal to BEGIN less INCREMENT. The third statement has a unique label (unilab_{i}) and assigns INDEX plus INCREMENT to INDEX.
A set of five statements is inserted at the position of the ENDDO statement. The first tests if the increment value is positive and branches to a unique label (unilab_{i+1}) if that is true. The next statement branches to unilab_i if INDEX is greater than or equal to univar_i. The third statement is an unconditional branch to another unique label (unilab_{i+2}). The fourth statement has unilab_{i+1} as the label and branches to unilab_i if INDEX is less than or equal to univar_i. The fifth statement has unilab_{i+2} as the label and assigns INDEX the value of INDEX plus INCREMENT. This rather complex set of statements at the ENDDO position is required because the increment value may be either positive or negative, and the condition for ending the loop differs according to the sign of the increment value.

DO index = begin TO end BY incr
  univar_i ← end - incr
  index ← begin - incr
  unilab_i; index ← index + incr
  assign
  call
  ENDDO

WHILE - ENDFOR: A unique label is placed on a statement which branches to a second unique label if the negation of the Boolean expression provided in the WHILE statement is true. An unconditional branch to the first unique label and a null statement with the second unique label are placed at the position of the ENDFOR. A WHILE - ENDFOR will not
be executed if the condition is not satisfied when the loop is encountered.

\[
\text{WHILE } \text{bool} \quad \text{unilab}_i; \quad \text{if not bool then } \quad \text{go to unilab}_{i+1} \\
\text{assign} \quad \text{assign} \\
\text{ENDWHILE} \quad \text{go to unilab}_i \\
\text{unilab}_{i+1}; \quad \text{null}
\]

**REPEAT UNTIL - ENDREPEAT:** A null statement with a unique label is placed at the position of the REPEAT UNTIL statement. A statement is inserted at the position of the ENDREPEAT which branches to that unique label if the negation of the Boolean expression specified in the REPEAT UNTIL statement is true. A REPEAT UNTIL - ENDREPEAT will always be executed at least once.

\[
\text{REPEAT UNTIL } \text{bool} \quad \text{unilab}_i; \quad \text{null} \\
\text{assign} \quad \text{assign} \\
\text{ENDREPEAT} \quad \text{if not bool then go to unilab}_i
\]

**CASE OF - ENDCASE:** A conditional branching statement is inserted at the position of the CASE OF statement which branches to a unique label if the index of the CASE statement is not equal to a test value which is initialized to one. Two statements are inserted after each statement or statement group within the CASE which is treated as a separate entity. The first is an unconditional branch to a unique label which will be placed on the second null statement at the position of the ENDCASE. The second statement has the unique label which was the destination of the last conditional branch and branches to another unique label if the index of the CASE is not equal to the test value which has been incremented by one. These two statements are repeated until all cases for this CASE structure have been processed. At the ENDCASE a null statement with the unique label
which was the destination in the last conditional branch is placed before the null statement described above.

```plaintext
CASE OF index
    case_1
        if index ne 1 then go to unilab_{i+1}
        case_1
        go to unilab_i
    unilab_{i+1}; if index ne 2 then
        go to unilab_{i+2}
    case_2
        go to unilab_i
    unilab_{i+2}; if index ne 3 then
        go to unilab_{i+3}
    case_3
    unilab_{i+3}; null
    unilab_i; null
ENDCASE

QUITLOOP: A QUITLOOP is replaced by an unconditional branch to a unique label which is placed on a null statement after the inserted code for the enclosing loop.

```plaintext
WHILE ------
    statement_1
    QUITLOOP
    statement_2
ENDWHILE

while replacement code
    statement_1
    go to unilab_i
    statement_2
    endwhile replacement code
    unilab_i; null

Design of Input

The following is a brief description of the input to the preprocessor. A more detailed description of the input may be found in Appendix II.

There are three types of input to the preprocessor. The first type consists of information about the base language. This information is provided by the user in a form derived from Backus-Naur Form [Gri71]. These definitions are used by the preprocessor in two ways. First, the preprocessor uses part of this information to recognize tokens in the
input program. Second, the preprocessor outputs statements standard in the base language by utilizing this information. Some of the information is used in only one manner. For example, the definition for end-of-statement is used only to recognize the end of a statement in the input program, and the statement prototypes are used only for producing output standard in the base language. On the other hand, some information is used in both ways. An example is the definition for a comment. The preprocessor uses this definition to recognize comments in the input program so they will not be analyzed as statements and also to produce comments in the output program which allow the user to locate where extended statements have been replaced.

The information required for each definition was determined and a definition form was then derived. In some cases, statement prototypes being an example, the definition simply reflects the proper form of the statement using parameter names where actual values would be placed. In other cases column numbers, lengths, and character strings are appropriate. It is also necessary to specify in some cases whether columns or character strings or both are utilized in the recognition of syntactic tokens. The comment definition is again a good example. It may contain column numbers or character strings or both. This example of a comment definition for PL/I contains an opening character string, an opening column number, and a closing character string.

EXAMPLE: Comment definition for PL/I

〈COMMENT〉:= '/*' + 10 '*/'

The definition of unique labels contains lengths and character strings required for generation while the definition of language labels contains
character strings and column numbers needed for recognition. The following example of a unique label definition for SNOBOL defines labels of the form 'LABxxxxxx:NEW' where xxxxxx is a sequence number. The example of FORTRAN language labels contains column numbers for positioning labels, a character set for the first character of a label, and a character set for the remaining characters of a label.

EXAMPLE: Unique label definition for SNOBOL

<UNIQUE LABEL> ::=LAB 6 NEW

EXAMPLE: Language label definition for FORTRAN

<LANGLABEL> ::=1-5 1-5 '0123456789' '0123456789'

Another piece of information relating to the base language which may be provided by the user concerns whether or not to recognize and expand extended constructs. If a base language has a construct which is recognized by keywords identical to those used by the preprocessor, the user must either specify that the extended construct keywords are to be overridden or not use the feature as provided in the base language. An override card to accomplish this task for the keywords DOGROUP and ENDDOGROUP would appear as follows:

OVERRIDE:DOGROUP,ENDDOGROUP

The second type of input consists of macro definitions. This information is optional. If the user wishes to introduce no macros, this is indicated by omission of any macro definition statements. The user provides a macro name, parameter names, and replacement text in each macro definition. The preprocessor will replace each macro call encountered with the corresponding replacement text, replacing parameter names with the actual arguments of the call.
The third type of input is the program written in the extended version of the programming language for which definitions have been provided. A special statement is included in the program to indicate the proper position for declarations of new variables.

Of some interest is the manner in which the definitions which have been provided are stored in the preprocessor. Each definition has an array associated with it. Although the specific contents of the array elements depend upon the particular definition, the concepts behind the storage of all the definitions are similar. Each array element contains a single piece of information which has been extracted from the definition. In the case of statement prototypes, the first element contains the number of statement components and each of the remaining elements contains one component, either text or a parameter name. For definitions of other syntactic elements, the definition array elements may contain information such as column numbers, character strings, or length requirements provided in the definitions.

The array for the FORTRAN language label definition above would contains the values 2,'COL',2,1,5,1,5,'0123456789', and '0123456789'. The first item indicates a series of column numbers is provided for the label starting position. The next two items indicate a series of column numbers is provided for the label ending position. The next four values specify the two series of columns. The two remaining character strings are the sets of valid characters for the first character of a label and the remaining characters of a label, respectively.
Design of Output

The output produced by the preprocessor is a program logically equivalent to the input program but which is standard in the base language. This output program is produced in two forms, a printed listing and a temporary disk file. The disk file can be input directly into the translator for the base language.

The output program contains at least one comment for each extended construct which is replaced by code produced by the preprocessor. There are also comments placed to indicate code inserted by the preprocessor which is not directly related to the use of a particular extended construct but of which the user should be aware. These comments indicating where new code has been produced are provided to aid the user in determining where errors have been made which relate to the error messages produced by the language translator and in correcting logic errors resulting from incorrect usage of the extended constructs.

The preprocessor's output does not go directly into the print file and the temporary disk file. There are intermediate files which hold these outputs. Figure II.2 illustrates the flow of data in the preprocessor. First the outputs are placed in intermediate files before it is determined whether a second parse will be required. If the second parse is not needed, the files are copied into the next set of intermediate files. If the second parse is required, the output of the second parse is placed in this set of intermediate files. The preprocessor then does any post processing necessary for the language being processed and places the outputs in the third set of intermediate
Figure II.2. Flow of Data in Preprocessor.
files. Finally, any required declarations of new variables are added and the outputs are placed in the final output files.

If an error condition occurs at any time, the preprocessor outputs an error message and prints the contents of whichever intermediate punch file contains the results of processing up to the error. This is done so the user can determine how far the preprocessor had progressed and what output had been produced thus far.

**Design of Specific Modules**

The following is a brief discussion of some of the modules of the implementation. Appendix I contains a detailed description of the final implementation.

Each definition provided by the user has an associated scanning routine which decodes the definition into its components and places them in the appropriate elements of the associated definition array. The scanning of all the definitions except those for macros is controlled by one routine which determines the scanning routine which should be called to process each definition.

The six statement prototypes are used to control the creation of output statements standard in the base language. Each of the prototypes has a statement creation routine with which it is associated which creates one type of statement and is controlled by the prototype.

There are three different insertion routines. One inserts a string with appropriate syntax to be a comment. A second routine inserts a string as a statement, skipping the columns specified in the ignored columns definition. The third insertion routine inserts a string which is already in the proper form for a statement.
Figure II.3. Modularity of the Preprocessor.
The routine which scans macro definitions is similar in concept to the other definition scanning routines in that it decodes a definition and stores its components in an array. However, the macro scanning routine creates two arrays, one for the parameters in the calling sequence and one for the components of the replacement text. These arrays are placed in elements of a SNOBOL table and are accessed via the macro name.

The structure parser is table-driven and written in a form based on Production Language. It has one syntactic stack which is represented by a character string. It was necessary to order the tests made against this stack such that the terminating symbol of a construct, for example ENDDO, at the stack top would not be mistakenly recognized as the opening symbol, in this case DO. The parser calls a scanning routine to place new tokens at the top of the syntactic stack and seventeen semantic routines to perform code generation and replacement.

The scanner uses much of the information provided by the user about the base language. Literals and comments are recognized as such and, as a result, their contents are not scanned for extended constructs. Another token is placed on the top of the syntactic stack each time the scanner is called. The following example represents the contents of the syntactic stack following the recognition of a DOGROUP statement, an assignment statement, and a REPEAT UNTIL statement. The tokens have been examined and reduced by the parser. The DOGROUP statement is represented by the token DOGROUPSTMT. The assignment statement has been reduced to a structure list (STRLIST) token. The REPEAT UNTIL statement is represented by the REPEATSTMT token. Note that the tokens are in a character string and separated by single blanks.
EXAMPLE: Contents of syntactic stack

DOGROUPSTM T STRLIST REPEATSTM T

There are seventeen semantic routines in the preprocessor. The basic strategy of the semantic routines is to make the extended statement into a comment and insert appropriate code which is standard in the base language. One semantic routine copies statements containing no extended constructs or macro calls directly into the output or replaces macro calls with the appropriate replacement text. Several other semantic routines, for example the one which processes the token DOGROUP, simply make extended statements into comments and produce no new code.

The semantic routines use a semantic stack represented by an array to store items to be transmitted between semantic routines. The stack may contain such items as labels, values, variables, and indications of the occurrence of the QUITLOOP construct. The following example of the semantic stack shows two labels and one indicator of the occurrence of a QUITLOOP structure.

EXAMPLE: Contents of the semantic stack.

<table>
<thead>
<tr>
<th>LAB03</th>
<th>LAB04</th>
<th>QUIT</th>
</tr>
</thead>
<tbody>
<tr>
<td>element 1</td>
<td>element 2</td>
<td>element 3</td>
</tr>
</tbody>
</table>

IMPLEMENTATION

Functional Classification of Modules

There are five functional categories of modules in the preprocessor. The categories are:
1) definition scanners
2) statement, label, and variable creators
3) semantic routines
4) service routines
5) special purpose modules

There are three types of definition scanners. The first type of definition scanning routine consists of the six routines which decode statement prototypes. The only routine of the second type is the routine which scans macro definitions. The third type consists of the routines which decode the remainder of the definitions.

There are six statement creation routines, one label creation routine, and one variable creation routine in the creator category. Each of the statement creator routines creates one type of output statement. The label and variable creators provide unique labels and identifiers, respectively. The semantic routine category contains the seventeen semantic routines. These are the routines which generate output code sequences.

There are six service routines which perform recurring tasks. Among these are two semantic stack manipulation routines, pop.sem and push.sem, and three string insertion routines. The other routine in this category is the one which retrieves single characters from the input text.

The special purpose modules include the preprocessor driver, the structure parser, the lexical scanner, and the routine which controls definition scanning.
Examples of Output

The following three examples illustrate the preprocessor's output for specific constructs. The first example illustrates the use of a CASE construct in PL/I. The second example is a REPEAT UNTIL structure in SNOBOL. The last example uses FORTRAN to illustrate the IF-THEN-ENDIF construct.

In each of these examples, the assumption is made that the preprocessor has not created any unique labels prior to the processing of the extended construct in the example. Also assumptions are made that the specification examples given for each language in Appendix IV were used by the preprocessor.

EXAMPLE -- PL/I LANGUAGE:

Partial Input Program:

```
CASE OF I;
    B = A + 6;
    B = A * 6;
    B = A - 6;
ENDCASE;
```

Corresponding Output Program:

```
/* CASE OF I; */
L0003: IF(I != 1) THEN GO TO L0002 ;
    B = A + 6;

/* NEXT TWO STATEMENTS INSERTED FOR CASE PROCESSING */
L0005: GO TO L0001 ;
L0002: IF(I != 2) THEN GO TO L0004 ;
    B = A * 6;

/* NEXT TWO STATEMENTS INSERTED FOR CASE PROCESSING */
L0007: GO TO L0001 ;
L0004: IF(I != 3) THEN GO TO L0006 ;
    B = A - 6;

/* NEXT TWO STATEMENTS INSERTED FOR CASE PROCESSING */
L0009: GO TO L0001 ;
L0006: IF(I != 4) THEN GO TO L0008 ;

/* ENDCASE; */
L0008: ;
L0001: ;
EXAMPLE -- SNOBOL LANGUAGE:

Partial Input Program:

    REPEAT UNTIL (LT(I,3))
       I = I / 6
       J = J + I
    ENDREPEAT

Corresponding Output Program:

    *  REPEAT UNTIL (LT(I,3))
L000001NEW
       I = I / 6
       J = J + I
    *  ENDREPEAT
L000002NEW  ~(LT(I,3))  )  :S(L000001NEW  )

EXAMPLE -- FORTRAN LANGUAGE:

Partial Input Program:

    IF (I,I.E.6) THEN       X = X + I
    ENDIF

Corresponding Output Program:

    C    IF (I,I.E.6) THEN
10024  IF( .NOT. (I,I.E.6) ) GO TO 10014
    X = X + I
    C    ENDIF
10014  CONTINUE
SECTION III. USER INFORMATION OVERVIEW

This overview is not intended to be used as a user's guide for the preprocessor. Appendix II contains the detailed information required for its use.

INPUT

There are three types of input to the preprocessor. The first type of input consists of information about the base language in a form derived from Backus-Naur Form. This portion of the input is described in the following paragraphs.

There are six statement prototypes provided by the user. These are patterns for the preprocessor to follow in creating output statements. The prototypes are as follows:

- unconditional branch
  - prototype for an unconditional branching statement

- conditional branch
  - prototype for statement which tests a condition and branches if the condition is true

- declaration
  - prototype for a statement which identifies a variable to the compiler; if no declarations are required, this prototype is blank

- null
  - prototype for statement which serves no function but to mark a location with a label

- assignment with plus
  - prototype for a statement which does a single addition operation and places the result in a variable

- assignment with minus
  - prototype for a statement which does a single subtraction operation and places the result in a variable
<table>
<thead>
<tr>
<th>Prototype</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>unconditional branch</td>
<td>label, destination</td>
</tr>
<tr>
<td>conditional branch</td>
<td>label, value 1, relation, value 2, destination</td>
</tr>
<tr>
<td>declaration</td>
<td>variable name</td>
</tr>
<tr>
<td>null</td>
<td>label</td>
</tr>
<tr>
<td>assignment with plus</td>
<td>label, variable, value 1, value 2</td>
</tr>
<tr>
<td>assignment with minus</td>
<td>label, variable, value 1, value 2</td>
</tr>
</tbody>
</table>

Table III.1 Prototype Parameters.

Each prototype has specific parameter names in it. The parameters for the prototypes are shown in Table III.1.

Each of the parameter names will be replaced by an appropriate value during generation of output code by the preprocessor. All the parameter names for each prototype must be present except in the declaration statement prototype. Since this prototype may be entirely blank under certain conditions, a test is not made for the absence of the parameter name.

The definitions required by the preprocessor for token recognition and output code generation are given in Table III.2. The definition for each of these items is represented in a special form based on the types of information and how much information is required. For example, there are four different ways in which a continuation card may be indicated. The first method is by a special continuation symbol. Another method is to start in a certain column. The third way is to
<table>
<thead>
<tr>
<th>Definition</th>
<th>Information Provided</th>
</tr>
</thead>
<tbody>
<tr>
<td>end of statement</td>
<td>recognition criteria for end of statement</td>
</tr>
<tr>
<td>continuation</td>
<td>recognition criteria for continuation card</td>
</tr>
<tr>
<td>language label</td>
<td>recognition criteria for a label in the input program</td>
</tr>
<tr>
<td>literal</td>
<td>recognition criteria for a literal string</td>
</tr>
<tr>
<td>comment</td>
<td>recognition and generation criteria for a comment</td>
</tr>
<tr>
<td>ignored columns</td>
<td>specification of columns whose contents are ignored in recognition and generation of statements</td>
</tr>
<tr>
<td>unique label</td>
<td>generation criteria for unique labels</td>
</tr>
<tr>
<td>unique variable</td>
<td>generation criteria for unique identifiers</td>
</tr>
<tr>
<td>greater than, greater than or equal, less than or equal, not equal, logical not</td>
<td>symbol strings for these Boolean operators</td>
</tr>
</tbody>
</table>

Table III.2. Definitions Required for Token Recognition and Code Generation.

use a special continuation symbol in a particular column. The final way is to have no indication at all, as in PL/I. Consequently, the continuation definition must convey which method is utilized and the information relevant to the method.

The user has the option of overriding the recognition of an extended construct keyword if that keyword will conflict with a
feature already present in the language. For example, if the user
wishes to code the IF-THEN and IF-THEN-ELSE constructs already
available in PL/I instead of using the extended constructs, it would
be necessary to include the keywords IF, THEN, ELSE, and ENDIF in
the list on the OVERRIDE card. This would allow the user to include
the desired constructs and prevent the preprocessor from mistaking
them for extended constructs.

The second type of input is macro definitions. The use of macros
is optional. Therefore, this part of the input may be omitted. Each
macro definition has a name, parameter names, and replacement text for
the macro. When the replacement text is inserted at a macro call,
the parameter names in the replacement text are replaced by the actual
arguments of the macro call.

There is a set of guidelines for writing macro replacement texts
given in Appendix II. The user is encouraged to follow these guidelines
to reduce the possibility of writing macro definitions which conflict
with the principles of structured programming.

OUTPUT

Output is produced by the preprocessor in two forms, as a printed
listing and a temporary disk file. The disk file may be input directly
into the translator for the base language.

Two messages will be produced in the printed output which do not
appear in the disk file. These messages indicate the success or failure
of the parses. If a second parse is not required, a message to that
effect is printed.
Each extended statement appears as a comment immediately before the code inserted by the preprocessor at the position of the extended statement. A comment appears every place a statement is inserted by the preprocessor, even if an extended statement has not been replaced by the inserted code. An example of this situation is the insertion of statements between the cases of a CASE structure even though no extended statement appears between the cases.

Each statement label appearing in the input program is detached from the associated statement and placed on a null statement. This null statement immediately precedes the original statement. This is necessary because a label and a statement cannot be reassociated after the statement has been recognized as an extended structure or macro call and replaced by inserted code. To maintain the proper locations of labels, they are placed on the null statements and positioned as described above.

There will be redundant comments and null statements when two parses have been made in processing the program. This is a result of label processing. The preprocessor will place all labels on null statements during the first parse. On the second parse, the label on each null statement created for a label during the first parse will be detached and placed on another null statement, and a comment will be generated which notes this insertion. Consequently, each label in the input program will result in two comments and two null statements when two parses are made during processing.

Post processing may alter this situation. For instance, the post processing for the PL/I language takes the label which has been
removed from the PROCEDURE OPTIONS statements and replaces it on that statement while eliminating the null statement(s) created for the label. For FORTRAN, the statement labels are returned to FORMAT statements in post processing.

**ERROR CONDITIONS**

The preprocessor does not detect errors in the base language portion of the input program. This task is left to the base language translator.

The preprocessor is a rather unforgiving tool. When an error is detected, the preprocessor outputs information pertinent to the cause of the error and terminates. Depending on what type of error occurred, the contents of the syntactic stack and certain variables may or may not be output. For any type of error, an error message and the output program produced so far are printed.

The common types of errors fall into three categories. The first type of error consists of all errors in specifying information about the base language and macro definitions. Among the errors falling in this category are incorrect definition syntax, missing definitions, invalid definition contents, and a missing ADDED DECLARATIONS statement when one is required. The user must be especially careful to provide a full eighty bytes in each of the six statement prototypes. This area is the only one in which the preprocessor is at all forgiving. If the preprocessor encounters an input record not containing a definition when one is expected, the preprocessor will skip that record. Normally this condition occurs when a statement prototype is not provided as a
full eighty bytes and the preprocessor uses the beginning of the next
definition as the end of the previous one. Consequently, the preprocessor
will later detect a missing definition and terminate processing.

The second type of error consists of omitting a keyword in a
construct -- for example, leaving out the ENDIF following an IF - THEN.
The preprocessor does not recognize this error until it attempts to
process the ending keyword of an enclosing construct or at the end
of the input program.

The third type of error includes syntax errors in writing extended
constructs. This type of error generally results in abnormal termination
because some expected keywords are not recognized. A good example of this
type of error is the mistake of not leaving column one of a SNOBOL
statement blank and placing a keyword for an extended construct in
that position. The keyword will be recognized as a label instead of
the appropriate keyword. This type of error may be considered a subset
of the second type since the preprocessor does not recognize this error
as a syntax error, but as an omission error.

When the preprocessor determines that an expected keyword is
missing, either by omission or syntax error, the current contents of
the syntactic stack are printed. This will help the user in determining
what keyword was not recognized. The output program produced so far
is printed to enable the user to determine at what point in analyzing
the input the error was encountered.
SECTION IV. OBSERVATIONS

GOOD AND BAD ASPECTS OF PROJECT

Naturally, there are both good and poor aspects of this project. The good points include the fact that SNOBOL has been implemented on a variety of computers produced by various manufacturers [Gri72], so the preprocessor could be ported from machine to machine without a great deal of revision. Only statements incompatible between SNOBOL and SPITBOL would have to be modified.

Another good aspect of the preprocessor is that it is generalized for use with a variety of languages. Even though the macro feature is somewhat unsophisticated, this feature contributes to the preprocessor's generality. One could also note that the preprocessor is easy to use with regard to the syntax of the input program since the original syntax of the programming language is changed only minimally.

The modular design and implementation of the preprocessor is a good point about the project. This became quite apparent during total system testing. The total system tests were completed in only eight days.

Among the poor aspects of the project is the fact that both the storage utilization and execution of the preprocessor are inefficient. A more experienced SPITBOL programmer could probably have produced a more efficient product. However, efficiency was not a major goal of this project.

The overhead involved in the use of the preprocessor is enormous at this point. To require the user to provide and the preprocessor
to decode all the base language definitions for each execution is undesirable and costly. This matter is addressed later in this section.

The fact that the user must provide the base language information for each execution is admittedly cumbersome and costly, but not overly difficult for the user. Perhaps the first development of the language specifications is somewhat difficult, but this is an initial investment only. However, the user may find it possible to use the language specifications provided as examples in Appendix IV, in which case this poor aspect is minimized.

The author would have liked to use PL/I rather than SPITBOL to code the preprocessor because SPITBOL lacks many structured programming features. However, the pattern matching feature of SPITBOL was of considerable value in writing the preprocessor.

OVERALL APPRAISAL OF DIRECTION

The author feels that the concept behind the project, that is, to create a generalized tool to provide structured programming capability, is a valid one and extremely desirable. However, it is also recognized that the current implementation will likely receive no utilization outside the academic environment. This limited use will be a result of the lack of efficiency of the final product, the cumbersome requirements for specifying base language information, and the high cost of execution. Consequently, the author feels the project was a success as far as demonstrating that the concept is viable, but the final product will receive a minimum amount of practical use.
TIME, EFFORT, AND MONEY INVOLVED

The time which the project required is difficult to provide accurately because the project was stretched out over an extended period of time. From the initial design efforts to the completion of documentation was a time span of two years. Therefore, the following figures are based on person days and not calendar days.

Design --- 2 months
Coding --- 1 month
Unit Tests --- 1 month
Integrated Tests --- 8 days
Documentation --- 2 months

Costs for computer time utilized in the testing phases amounted to $642.35. This figure was based on the following rates:

<table>
<thead>
<tr>
<th>Description</th>
<th>Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU time</td>
<td>1 min. = 9 units</td>
</tr>
<tr>
<td>K-byte time</td>
<td>1 min. = .037 unit</td>
</tr>
<tr>
<td>cards read</td>
<td>1 card = .0006 unit</td>
</tr>
<tr>
<td>lines printed</td>
<td>1 line = .0005 unit</td>
</tr>
<tr>
<td>pages printed</td>
<td>1 page = .0007 unit</td>
</tr>
<tr>
<td>disk EXCP's</td>
<td>1 EXCP = .0002 unit</td>
</tr>
</tbody>
</table>

1 unit = $1

ELEMENTS TO BE CHANGED

The author hopes that one of the first changes to be implemented in the preprocessor will be to allow specifications for a language to be provided once and only once. After being provided once, they would be stored on tape or disk and retrieved when execution begins. A user would also be given the option of providing a new set of specifications to temporarily override those specifications stored in the system. This option was provided in the original design and subsequently eliminated during implementation.
Another desirable change would be to increase the efficiency of
the preprocessor, both in storage utilization and execution time. A
programmer knowledgeable in the intricacies of SPITBOL could accomplish
this task.

A change to the keyword overriding policy would be useful. To
allow the user to specify an alternate form of a keyword instead of
simply eliminating it completely would alleviate a problem in using
the preprocessor with the PL/I language. Currently, to use the macro
facility with PL/I requires that there be no THEN keyword in the output
program produced by the first parse unless it is part of an extended
IF - THEN construct. This means no THEN keyword could be produced
during the first parse except in the expansion of a macro call, but
many of the extended constructs produce that particular keyword. It
seems, therefore, that the user would be able to write very little
useful code outside the macros.

POSSIBLE FURTHER DEVELOPMENTS

There are obviously possibilities for further development. One
of these is to implement a more sophisticated macro facility. The
macro feature should not be made elegant, because the preprocessor
was not intended as a general purpose preprocessor. However, the
techniques for manipulation of formal parameters and actual arguments
might be upgraded to allow similar names for parameters and semantic
stack manipulation.

One other possibility is to allow the user to use continuation
cards in extended construct statements when continuation is indicated
by a symbol. This would involve deleting the continuation symbol when it is encountered and replacing it in all occurrences of continuation cards.
REFERENCES


Kel Keller, Roy F. A Modern Beginning Programming Course. Computer Science Department and Ames Laboratory, ERDA, Ames, Iowa.


Shn74  Shneiderman, Ben. The Chemistry of Control Structures, SIGPLAN Notices, Vol. 9, No. 12, December 1974, pp. 29-34.


APPENDIX I. DETAILED IMPLEMENTATION INFORMATION

This Appendix contains detailed information about the implementation of the preprocessor. It is intended as a guide to anyone wishing to make modifications to the source code. Consequently, comprehension will be severely limited unless the reader refers frequently to the source listing in Appendix V. The remainder of this Appendix describes the implementation by categories of modules as follows: driver, definition scanning routines, macro definition scanner, creation routines, service routines, semantic routines, insertion routines, parser, scanner, and character retrieval routine.

Driver

The driver of the preprocessor defines output associations, function calling sequences, and arrays. Patterns and variables are also initialized.

The driver reads the language name and checks it against those which the preprocessor can analyze. Each input card is read and appropriate processing is initiated. An OVERRIDE card is processed directly by the driver. The first definition card encountered causes the preprocessor to call SCANDEFS. The preprocessor calls the MACRSCAN routine when the first card of a macro definition is read. The preprocessor will begin processing the input program when the first card not recognized as a definition or an override card is read. The parse number indicator is set to one, and the parser is called. When control is returned to the driver, a test is made to determine if the parse was successful.
If the parse was unsuccessful, a branch is taken to the error exit for an unsuccessful first parse. If the parse was successful, a message indicating such is printed, the output files are rewound, and new input and output associations are defined. If no macro definitions were provided, a branch is taken around the second parse and the temporary print and punch files are copied to the next intermediate files. Otherwise, variables are reset to appropriate initial values, and the parse number indicator is set to two. The parser is called again, and a test for success is made when control is returned to the driver. If an error has occurred, a branch is taken to an error exit. Whether one or two parses are made, the intermediate files are rewound and new input and output associations are defined.

If the declaration statement prototype is non-blank, declarations are added for all new variables introduced by the preprocessor. The statement containing the string 'ADDED DECLARATIONS' is found in the output files and a declaration is added at that point for each variable name in the array VARLIST. Comments are also inserted to indicate that these declarations were added by the preprocessor. The remainder of the files are then copied. If no ADDED DECLARATIONS statement is found, an error exit is taken.

The next step is to execute post-processing if required for the language being processed. The intermediate files are rewound and new input and output associations are defined. The language type is checked.
If the language is SNOBOL, each statement is scanned for the `(` symbols followed by blanks and `,`. At each occurrence, the blanks and comma are deleted.

If the language is FORTRAN, all sequences of 'comment-null statement-FORMAT statement' and 'comment-comment-null statement-null statement-FORMAT statement' are found. In each case, the statement number from the first null statement is placed on the FORMAT statement. The comments and the null statements are deleted.

If the language is PL/I, the first records of the output files are scanned for either one or two comments followed by the same number of null statements and a procedure statement. The label from the first null statement is placed on the procedure statement. The comments and the null statements are deleted, and the remainders of the files are copied.

If any other language is being processed (possible only if the language type pattern, LANCPAT, is enlarged to include other languages), the files are simply copied to the final output files.

The preprocessor prints a successful completion message and terminates.

The section to print the appropriate intermediate files on an error condition tests the parse number and rewinds the appropriate intermediate punch file. An input association is defined for the rewound file, and an output association with a real printer file is defined. The contents of the temporary punch file to be displayed are read and printed. An unsuccessful termination message is also printed and execution halts.
All error exits are provided in the driver. An explanatory message is printed for each error. Other information is also provided for certain errors as shown in Table A.I.I. If the preprocessor has begun analyzing the input program, a branch is taken to the section to print the intermediate file contents.

Definition Scanning Routines

One routine, SCANDEFS, controls all definition scanning except that for macro definitions. When the first definition is encountered, the first card of that definition is passed to this controlling routine. The definition is processed as described below, and another definition is read. After all nineteen definitions have been processed, or an attempt has been made to decode nineteen definitions, control is returned to the calling routine.

To process a definition, the SCANDEFS routine determines which one of fifteen individual scanning routines to call, based on the definition name. The definition name, its enclosing angle brackets, and the := symbol following it are stripped away. The remainder of the definition is then passed to the routine being called. If the definition is for a statement type, the routine reads another card and passes the entire prototype to the routine. If the definition is that for a language label, the routine determines whether or not the entire definition is contained on the single card. If not, the routine reads another input card and passes the entire definition to the routine for the LANGLABEL definition. If an attempt is made to read a definition, but the card contains an invalid definition name or no definition name in angle
<table>
<thead>
<tr>
<th>ERROR EXIT</th>
<th>ERROR CONDITION</th>
<th>ADDITIONAL INFORMATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>LANGERR</td>
<td>invalid language type</td>
<td>language type provided in input</td>
</tr>
<tr>
<td>ENDPARSEERR</td>
<td>first parse terminated abnormally</td>
<td>contents of syntactic stack</td>
</tr>
<tr>
<td>END2ERR</td>
<td>second parse terminated abnormally</td>
<td>contents of syntactic stack</td>
</tr>
<tr>
<td>GTHANDEFERR</td>
<td>error in definition of greater-than operation</td>
<td>definition provided in input</td>
</tr>
<tr>
<td>LEQUDFERR</td>
<td>error in definition of less-than-or-equal oper.</td>
<td>definition provided in input</td>
</tr>
<tr>
<td>GEQUDFERR</td>
<td>error in definition of greater-than-or-equal oper.</td>
<td>definition provided in input</td>
</tr>
<tr>
<td>NOTDEFERR</td>
<td>error in definition of not operator</td>
<td>definition provided in input</td>
</tr>
<tr>
<td>NEQUDFERR</td>
<td>error in definition of not-equal operator</td>
<td>definition provided in input</td>
</tr>
<tr>
<td>MACCALLERR</td>
<td>error in macro calling sequence definition</td>
<td>macro name</td>
</tr>
<tr>
<td>PSCANERR</td>
<td>no value returned from scanner</td>
<td>input statement being processed, contents of syntactic stack</td>
</tr>
<tr>
<td>PUNIDERR</td>
<td>parser unable to identify contents of syntax stack</td>
<td>input statement being processed, contents of syntactic stack</td>
</tr>
<tr>
<td>PSTRERR</td>
<td>failure to parse STRUC to STRLIST reduction</td>
<td>input statement being processed, contents of syntactic stack</td>
</tr>
<tr>
<td>PEDOGRPFERR</td>
<td>failure to parse ENDDOGRGROUP</td>
<td>input statement being processed, contents of syntactic stack</td>
</tr>
<tr>
<td>PEDCERR</td>
<td>failure to parse ENDDO</td>
<td>input statement being processed, contents of syntactic stack</td>
</tr>
</tbody>
</table>

Table A.I.1. Error Exits with Additional Information Provided.
<table>
<thead>
<tr>
<th>ERROR EXIT</th>
<th>ERROR CONDITION</th>
<th>ADDITIONAL INFORMATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>PIFTHERR</td>
<td>failure to parse IF - THEN clause</td>
<td>input statement being processed, contents of syntactic stack</td>
</tr>
<tr>
<td>PEWHERR</td>
<td>failure to parse ENDWHILE</td>
<td>input statement being processed, contents of syntactic stack</td>
</tr>
<tr>
<td>PECASEERR</td>
<td>failure to parse ENDCASE</td>
<td>input statement being processed, contents of syntactic stack</td>
</tr>
<tr>
<td>PQUERR</td>
<td>failure to parse QUITLOOP</td>
<td>input statement being processed, contents of syntactic stack</td>
</tr>
<tr>
<td>PWHERR</td>
<td>failure to parse WHILE</td>
<td>input statement being processed, contents of syntactic stack</td>
</tr>
<tr>
<td>PREPUERR</td>
<td>failure to parse REPUNTIL</td>
<td>input statement being processed, contents of syntactic stack</td>
</tr>
<tr>
<td>PREPERR</td>
<td>failure to parse REPEAT</td>
<td>input statement being processed, contents of syntactic stack</td>
</tr>
<tr>
<td>PIFERR</td>
<td>failure to parse IF</td>
<td>input statement being processed, contents of syntactic stack</td>
</tr>
<tr>
<td>PDCRPERR</td>
<td>failure to parse DOGROUP</td>
<td>input statement being processed, contents of syntactic stack</td>
</tr>
<tr>
<td>PDCERR</td>
<td>failure to parse DO</td>
<td>input statement being processed, contents of syntactic stack</td>
</tr>
<tr>
<td>PCASEERR</td>
<td>failure to parse CASE</td>
<td>input statement being processed, contents of syntactic stack</td>
</tr>
<tr>
<td>PEREPERR</td>
<td>failure to parse ENDREREPEAT</td>
<td>input statement being processed, contents of syntactic stack</td>
</tr>
</tbody>
</table>

Table A.I.1. Error Exits with Additional Information Provided. (cont'd).
<table>
<thead>
<tr>
<th>ERROR EXIT</th>
<th>ERROR CONDITION</th>
<th>ADDITIONAL INFORMATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>ELSEERROR</td>
<td>error in semantic</td>
<td>portion of current input statement not processed by semantic routine</td>
</tr>
<tr>
<td></td>
<td>processing of ELSE</td>
<td></td>
</tr>
<tr>
<td>CASEERROR</td>
<td>error in semantic</td>
<td>portion of current input statement not processed by semantic routine</td>
</tr>
<tr>
<td></td>
<td>processing of CASE</td>
<td></td>
</tr>
<tr>
<td>DOERROR</td>
<td>error in semantic</td>
<td>portion of current input statement not processed by semantic routine</td>
</tr>
<tr>
<td></td>
<td>processing of DO</td>
<td></td>
</tr>
<tr>
<td>IFERROR</td>
<td>error in semantic</td>
<td>portion of current input statement not processed by semantic routine</td>
</tr>
<tr>
<td></td>
<td>processing of IF</td>
<td></td>
</tr>
<tr>
<td>REERROR</td>
<td>error in semantic</td>
<td>portion of current input statement not processed by semantic routine</td>
</tr>
<tr>
<td></td>
<td>processing of REPEAT</td>
<td></td>
</tr>
<tr>
<td>WHERROR</td>
<td>error in semantic</td>
<td>portion of current input statement not processed by semantic routine</td>
</tr>
<tr>
<td></td>
<td>processing of WHERE</td>
<td></td>
</tr>
<tr>
<td>QUITERROR</td>
<td>error in semantic</td>
<td>portion of current input statement not processed by semantic routine</td>
</tr>
<tr>
<td></td>
<td>processing of QUIT</td>
<td></td>
</tr>
<tr>
<td>ECASEERROR</td>
<td>error in semantic</td>
<td>portion of current input statement not processed by semantic routine</td>
</tr>
<tr>
<td></td>
<td>processing of ENDCASE</td>
<td></td>
</tr>
<tr>
<td>EIFERROR</td>
<td>error in semantic</td>
<td>portion of current input statement not processed by semantic routine</td>
</tr>
<tr>
<td></td>
<td>processing of ENDIF</td>
<td></td>
</tr>
<tr>
<td>EWHILEERROR</td>
<td>error in semantic</td>
<td>portion of current input statement not processed by semantic routine</td>
</tr>
<tr>
<td></td>
<td>processing of ENDWHILE</td>
<td></td>
</tr>
<tr>
<td>EREERROR</td>
<td>error in semantic</td>
<td>portion of current input statement not processed by semantic routine</td>
</tr>
<tr>
<td></td>
<td>processing of ENDREPEAT</td>
<td></td>
</tr>
<tr>
<td>EDOERROR</td>
<td>error in semantic</td>
<td>portion of current input statement not processed by semantic routine</td>
</tr>
<tr>
<td></td>
<td>processing of ENDDO</td>
<td></td>
</tr>
</tbody>
</table>

Table A.I.1. Error Exits with Additional Information Provided (cont'd).
<table>
<thead>
<tr>
<th>ERROR EXIT</th>
<th>ERROR CONDITION</th>
<th>ADDITIONAL INFORMATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>EDOGRPERROR</td>
<td>error in semantic processing of ENDDOGROUP</td>
<td>portion of current input statement not processed by semantic routine</td>
</tr>
<tr>
<td>DCLBLKERR</td>
<td>too few blanks in DECLARATION STATEMENT prototype to accommodate actual argument</td>
<td>actual argument</td>
</tr>
<tr>
<td>GOTOBLKERR</td>
<td>too few blanks in GO TO STATEMENT prototype to accommodate actual argument</td>
<td>input statement being processed</td>
</tr>
<tr>
<td>PLUSBLKERR</td>
<td>too few blanks in ASSIGN PLUS STATEMENT prototype to accommodate actual argument</td>
<td>input statement being processed</td>
</tr>
<tr>
<td>MINBLKERR</td>
<td>too few blanks in ASSIGN MINUS STATEMENT prototype to accommodate actual argument</td>
<td>input statement being processed</td>
</tr>
<tr>
<td>NULLBLKERR</td>
<td>too few blanks in NULL STATEMENT to accommodate actual argument</td>
<td>input statement being processed</td>
</tr>
<tr>
<td>IFBLKERR</td>
<td>too few blanks in IF THEN GO TO STATEMENT prototype to accommodate actual argument</td>
<td>input statement being processed</td>
</tr>
</tbody>
</table>

Table A.I.1. Error Exits with Additional Information Provided (cont'd).
brackets, input cards are skipped until another definition is encountered. If the error occurs when the routine is attempting to read the nineteenth definition, control returns to the calling routine without reading any additional input cards.

The individual scanning routines called by SCANDEFs are summarized in Table A.I.2 and fall into four categories. The first category contains those routines which scan statement prototypes. Each of these routines has a single parameter which contains the prototype. Each prototype is broken down by the individual routines into components. Each parameter name and the blanks immediately following it comprise a component. All other parts before and between these components also constitute components. For example, the prototype for a GO TO STATEMENT contains two parameters -- a label (LAB) and a destination (DES). If the prototype for a GO TO STATEMENT were 'LAB BRANCH DES', the contents of the first component would be null. The second component would contain 'LAB '. The third component would contain 'BRANCH ', and 'DES' would be in the fourth component. Additional columns are considered to contain blanks through column eighty. The maximum number of components possible for each definition is based on the number of parameter names in the prototype. The numbers of components are as follows:
<table>
<thead>
<tr>
<th>DEFINITION NAME</th>
<th>MAX. NUMBER OF COMPONENTS</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF THEN GO TO STATEMENT</td>
<td>11</td>
</tr>
<tr>
<td>GO TO STATEMENT</td>
<td>5</td>
</tr>
<tr>
<td>NULL STATEMENT</td>
<td>3</td>
</tr>
<tr>
<td>ASSIGN PLUS STATEMENT</td>
<td>9</td>
</tr>
<tr>
<td>ASSIGN MINUS STATEMENT</td>
<td>9</td>
</tr>
<tr>
<td>DECLARATION STATEMENT</td>
<td>3</td>
</tr>
</tbody>
</table>

The components for each prototype are placed in an array. The zeroeth element of each array is used to record the number of components. For the example given above, the zeroeth element would contain 4. The arrays associated with the definitions are as follows:

<table>
<thead>
<tr>
<th>DEFINITION NAME</th>
<th>ARRAY AND DIMENSIONS</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF THEN GO TO STATEMENT</td>
<td>IFPAT(0:11)</td>
</tr>
<tr>
<td>GO TO STATEMENT</td>
<td>GOTOpat(0:5)</td>
</tr>
<tr>
<td>NULL STATEMENT</td>
<td>NULLPAT(0:3)</td>
</tr>
<tr>
<td>ASSIGN PLUS STATEMENT</td>
<td>PLUSPAT(0:9)</td>
</tr>
<tr>
<td>ASSIGN MINUS STATEMENT</td>
<td>MINUSPAT(0:9)</td>
</tr>
<tr>
<td>DECLARATION STATEMENT</td>
<td>DCLPAT(0:3)</td>
</tr>
</tbody>
</table>

Each of the scanning routines which decodes a statement prototype functions in basically the same way. The parameter names for that statement type are placed in a SNOBOL pattern. The definition is searched for pairs of components, the second of which is a parameter name. The first component of the pair may be null, and the second component will include any blanks following the parameter name. If
the entire prototype has been scanned after entering the component pair into the appropriate array, the routine immediately records the number of components. If the last portion of the definition is not a parameter name, a component pair will not be found when that is all that remains to be scanned. Therefore, the remainder of the prototype is placed in the array and the number of components is then recorded. In either case, once the number of components has been placed in the array, the routine tests to see if all the necessary parameters have been specified. The exception to this procedure is in the routine which scans the DECLARATION STATEMENT prototype. Since a blank prototype may be specified for this statement type, no check is made for the appropriate number of parameter names.

Another category of definition scanning routines contains those routines which scan the definitions of unique identifiers, unique labels, literals, and continuations. These scanning routines are similar because the numbers and types of parameters are fairly stable. Both the identifier and label definitions must have a string followed by an integer. Another string may optionally follow the integer. The literal definition contains two strings, and a third string is optional. The continuation definition contains either the single word NONE or a string followed by a column number. Each of these components is placed in an element of the array associated with the definition as indicated by the following:
<table>
<thead>
<tr>
<th>DEFINITION NAME</th>
<th>ARRAY AND DIMENSIONS</th>
</tr>
</thead>
<tbody>
<tr>
<td>UNIQUE IDENTIFIER</td>
<td>VARPAT(1:3)</td>
</tr>
<tr>
<td>UNIQUE LABEL</td>
<td>LABELPAT(1:3)</td>
</tr>
<tr>
<td>LITERAL</td>
<td>LITPAT(1:3)</td>
</tr>
<tr>
<td>CONTINUATION</td>
<td>CONTPAT(1:2)</td>
</tr>
</tbody>
</table>

The third category of scanning routines contains those routines which scan the definitions for end-of-statement, ignored columns, comment, and language label. Each of these definitions has alternative forms of information, the types of which must be recorded in the array associated with the definition. The arrays and the definitions with which they are associated are:

<table>
<thead>
<tr>
<th>DEFINITION NAME</th>
<th>ARRAY AND DIMENSIONS</th>
</tr>
</thead>
<tbody>
<tr>
<td>END OF STATEMENT</td>
<td>ECSPAT(0:1)</td>
</tr>
<tr>
<td>IGNORE COLUMNS</td>
<td>IGNPAT(0:2)</td>
</tr>
<tr>
<td>COMMENT</td>
<td>COMPAT(1:6)</td>
</tr>
<tr>
<td>LANGLABEL</td>
<td>LANLABPAT(1:9)</td>
</tr>
</tbody>
</table>

The END OF STATEMENT definition may specify either a symbol string or a column number. Based on the contents of the definition, either 'STR' or 'COL' is placed in the zeroth element of ECSPAT. The string or column number specified is placed in the next element. The IGNORE COLUMNS definition may specify either a single column number or a sequence of column numbers separated by a hyphen. If only one column number is specified, 'ONE' is placed in the zeroth element, and the column number is placed in element one. If a sequence is specified.
'SEQ' goes into the zeroeth element, and elements one and two will contain the two column numbers given as the bounds of the sequence.

The COMMENT definition contains two sets of information. Each set may be a symbol string, a column number, or both with a '+' between them. Element one of the associated array contains an indication of what is present for the comment opener. Element four contains similar information for the comment closer. The information is specified as a string with the following meanings:

<table>
<thead>
<tr>
<th>STRING</th>
<th>MEANING</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>symbol string only</td>
</tr>
<tr>
<td>01</td>
<td>column number only</td>
</tr>
<tr>
<td>11</td>
<td>symbol string and column number</td>
</tr>
</tbody>
</table>

Elements two and three contain the string and column number, if present, for the opener. Elements five and six contain the string and column number, if present, for the closer.

The language label definition contains four pieces of information. The first may be either a string, column number, or a sequence of columns represented by two column numbers separated by a hyphen. The first element of the language label array contains either 1 or 2, for one column number or a sequence of them, respectively. The fourth element contains the single column number or the first of the two column numbers specified as the sequence bounds. The fifth element contains the second column number of the sequence specification. The second piece of information may be of the same form as the first or may be a symbol string. If the column or column sequence form is specified, the
<table>
<thead>
<tr>
<th>ROUTINE NAME</th>
<th>'PATTERN' CREATED</th>
<th>PARAMETERS</th>
</tr>
</thead>
<tbody>
<tr>
<td>SCANNULLDEF</td>
<td>NULLPAT(0:3)</td>
<td>NULLDEF</td>
</tr>
<tr>
<td>SCANGOTODEF</td>
<td>GOTOPAT(0:5)</td>
<td>GOTODEF</td>
</tr>
<tr>
<td>SCANIFDEF</td>
<td>IFPAT(0:11)</td>
<td>IFDEF</td>
</tr>
<tr>
<td>SCANPLUSDEF</td>
<td>PLUSPAT(0:9)</td>
<td>PLUSDEF</td>
</tr>
<tr>
<td>SCANMINUSDEF</td>
<td>MINUSPAT(0:9)</td>
<td>MINUSDEF</td>
</tr>
<tr>
<td>SCANDECLDEF</td>
<td>DCLPAT(0:3)</td>
<td>DCLDEF</td>
</tr>
<tr>
<td>SCANVARDEF</td>
<td>VARPAT(1:3)</td>
<td>VARDEF</td>
</tr>
<tr>
<td>SCANLABELDEF</td>
<td>LABELPAT(1:3)</td>
<td>LABELDEF</td>
</tr>
<tr>
<td>SCANLITDEF</td>
<td>LITPAT(1:3)</td>
<td>LITDEF</td>
</tr>
<tr>
<td>SCANCONTDEF</td>
<td>CONTPAT(1:6)</td>
<td>CONTDEF</td>
</tr>
<tr>
<td>SCANCOMDEF</td>
<td>COMPAT(1:6)</td>
<td>COMDEF</td>
</tr>
<tr>
<td>SCANECSDEF</td>
<td>EOSPAT(0:1)</td>
<td>ECSDEF</td>
</tr>
<tr>
<td>SCANIGNCOLDEF</td>
<td>IGNPAT(0:2)</td>
<td>IGNCOLDEF</td>
</tr>
<tr>
<td>SCANLANLABELDEF</td>
<td>LANLABELPAT(1:9)</td>
<td>LANLABELDEF</td>
</tr>
<tr>
<td>SCANOPERDEF</td>
<td>GTHAN</td>
<td>TYPEDEF, OPERDEF</td>
</tr>
<tr>
<td></td>
<td>GEQU</td>
<td></td>
</tr>
<tr>
<td></td>
<td>LEQU</td>
<td></td>
</tr>
<tr>
<td></td>
<td>NEQU</td>
<td></td>
</tr>
<tr>
<td></td>
<td>NOT</td>
<td></td>
</tr>
</tbody>
</table>

SCANDEFS

CARD

Table A.I.2. Scanning Routines,
information is stored as indicated above, but in elements three, six and seven. If a symbol string is given, 'STR' is placed in element two, and the string is placed in element six. The third and fourth pieces of information in this definition are character strings. They are placed in elements eight and nine, respectively.

The fourth and final category of scanning routines contains a single routine which scans the definitions for the five relations -- greater than, less than or equal, greater than or equal, not equal, and not. Each of these definitions has a single variable associated with it in which the symbol string for the appropriate operator is placed. The definitions and their associated variables are as follows:

<table>
<thead>
<tr>
<th>RELATION</th>
<th>VARIABLE</th>
</tr>
</thead>
<tbody>
<tr>
<td>GREATER THAN</td>
<td>GTHAN</td>
</tr>
<tr>
<td>LESS THAN OR EQUAL</td>
<td>LEQU</td>
</tr>
<tr>
<td>GREATER THAN OR EQUAL</td>
<td>GEQU</td>
</tr>
<tr>
<td>NOT EQUAL</td>
<td>NEQU</td>
</tr>
<tr>
<td>NOT</td>
<td>NOT</td>
</tr>
</tbody>
</table>

Macro Definition Scanner

The macro definition scanner (MACROSCAN routine) is not included with the other scanning routines because it has some basic structural and technique differences from those routines. It is called directly from the driver. It is called once for each macro definition and is therefore likely to be called more than once per execution of the preprocessor. The information extracted is stored in a SNOBOL table as opposed to simply an array.
The macro scanning routine is called when the first card of a macro definition is encountered by the driver. This card contains the string '{MACRO CALL}:=*' followed by the name of the macro. The macro name is immediately followed by its formal parameters separated by commas and enclosed within a single set of parentheses.

EXAMPLE:

{MACRO CALL}:=WHEN(X,Y)

MACROSCAN deletes the angle brackets and their contents before extracting the macro name and the parameters. A second card is read if the parameter list extends past the end of the first card. No blanks, imbedded or at the end of the first of two cards, are allowed. The zeroeth element of the MPARM array is set equal to the number of formal parameters provided. The first parameter is placed in element one of MPARM and succeeding parameters are placed in consecutive elements of MPARM.

The next card is read and should contain only the string '{MACRO DEFINITION}:=*'. The cards following are read and their contents placed in the MACREPSTR variable until the end of the definition is found. The end is signified by a card containing only the string 'END MACRO DEFINITION' beginning in column one. This string is not added to MACREPSTR.

The parameter list pattern is started with a series of 81 blanks. The formal parameters are then added to this pattern from the MPARM array. The macro scanner next breaks the MACREPSTR contents into components and stores those components in the MPAT array. The zeroeth element of MPAT is assigned the number of components in the replacement text. A component is defined as for the statement scanning routines.
The contents of the MPAT array are copied into the macro table MACROTAB indexed by the macro name in the form of a string. The parameter list for the macro, currently in the MPARM array, is copied into the element of MACROTAB indexed by the macro name in the form of a string suffixed with the character 'P'. The routine then outputs a message that this macro definition has been processed and returns to the driver.

Creation Routines

There are eight routines which create items for the preprocessor. Six of them create statements. The other two create unique identifiers and labels. Each of the statement-creating routines has a number of parameters equal to the number of parameter names which appear in the corresponding statement prototype. Each of these routines utilizes the array containing the appropriate statement components. By replacing the components containing parameter names with the corresponding arguments supplied in the call to the creating routine and including enough blanks to retain the format and length of the prototype (to eighty characters) in the output, the routines create statements which are standard in the base language.

The routines which create statements, their outputs and parameters are as follows:
<table>
<thead>
<tr>
<th>ROUTINE NAME</th>
<th>OUTPUT STATEMENT</th>
<th>PARAMETERS</th>
</tr>
</thead>
<tbody>
<tr>
<td>CRNULL</td>
<td>NULL STATEMENT</td>
<td>LAB</td>
</tr>
<tr>
<td>CRIF</td>
<td>IF THEN GO TO STATEMENT</td>
<td>LAB, VAL1, RL,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>VAL2, DES</td>
</tr>
<tr>
<td>CRGOTO</td>
<td>GO TO STATEMENT</td>
<td>LAB, DES</td>
</tr>
<tr>
<td>CRMINUS</td>
<td>ASSIGN MINUS STATEMENT</td>
<td>LAB, VAR, VAL1,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>VAL2</td>
</tr>
<tr>
<td>CRPLUS</td>
<td>ASSIGN PLUS STATEMENT</td>
<td>LAB, VAR, VAL1,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>VAL2</td>
</tr>
<tr>
<td>CRDCL</td>
<td>DECLARATION STATEMENT</td>
<td>VARNAME</td>
</tr>
</tbody>
</table>

The routines which create unique identifiers and labels also utilize the arrays associated with the definitions provided. VARNUM and LABELNUM are the variables used to make the identifiers and labels unique. Each routine uses the appropriate variable and then increments it by one. Enough leading zeroes are used with these variables, which are initialized to one, to create items of the lengths specified in the definitions. The routine which creates unique identifiers adds each one to the array VARLIST which is indexed using the variable NEXTVAR. This array is later used to add declarations to the program if required. Neither UNILABEL nor UNIVAR has any parameters. The routines and their outputs are as follows:

<table>
<thead>
<tr>
<th>ROUTINE NAME</th>
<th>OUTPUT</th>
</tr>
</thead>
<tbody>
<tr>
<td>UNIVAR</td>
<td>unique identifier</td>
</tr>
<tr>
<td>UNILABEL</td>
<td>unique label</td>
</tr>
</tbody>
</table>

Service Routines

Two service routines are present in the preprocessor which manipulate the semantic stack. The routines are POP.SEM and PUSH.SEM which pop items off and push items onto the semantic stack, respectively.
The semantic stack is represented by the array SEMSTACK which contains ten elements. Element 1 represents the element at the bottom of the stack. The variable TOP contains the subscript of the top element in the stack.

Each of these routines may have from two to seven parameters. The first parameter is always the number of items to be acted upon, that is, the number of items to pop from or push onto the stack. The second through seventh parameters are passed differently in the two routines. Variables are passed to the POP,SEM routine in which the popped items will be returned. These variables’ names are passed as character strings, and values are assigned to the corresponding arguments using two levels of indirection. Variables passed to the PUSH,SEM routine as parameters two through seven contain values to be pushed onto the stack. The values are assigned from the corresponding arguments using only one level of indirection.

POP,SEM extracts a value from the top of the stack and decrements the stack pointer, TOP. PUSH,SEM increments TOP, the stack pointer, and assigns a value to the stack’s new top element. Each routine loops until the number of values indicated by the first argument have been processed.

Semantic Routines

Most of the semantic routines call the statement creation routines. These routines are always called with a label value. This is necessary because if a symbol is normally used for terminating a label, an extraneous symbol will appear in the output if a null value is passed as the label.
Semantic Routine 0 - This semantic routine is called when the parser recognizes a label on a statement. The routine inserts a null statement with the label just recognized. If a language label is terminated by a symbol, that symbol is removed from the label before the null statement is created. This step is necessary because the terminating symbol is already provided in the null statement prototype.

Semantic Routine 01 - This routine processes items recognized as unidentified-to-end-of-statement (U-EOS) and macro calls. Any sequence of one or more items terminating in the end-of-statement condition and not containing any token recognized as part of an extended construct is reduced to the U-EOS token. This routine checks the string recognized as U-EOS to determine if it is a macro call.

If the U-EOS is a macro call, the actual arguments of the macro call are broken out and placed in an array. The macro's formal parameters and the number of parameters, as well as the macro's replacement text components are retrieved from the macro table. The parameters are placed in a SNOBOL pattern. The components of the replacement text are copied into a string, replacing the formal parameter names with actual arguments. The macro call is made into a comment and inserted into the output, followed by the expanded macro text.

If the sequence recognized as an U-EOS does not contain a macro call, the statement is copied directly from the input to the output.

Semantic Routine 1 - This routine is called by the parser when the extended construct keyword ELSE has been recognized. The routine
pops a label name from the semantic stack and creates another unique label. An unconditional branch to the new unique label and a null statement with the label popped from the semantic stack are inserted into the output, following a comment indicating that the ELSE keyword was processed at that point. The unique label created in this routine is pushed onto the semantic stack.

Semantic Routine 2 - This routine processes the statement which begins a CASE extended construct. The index of the CASE is extracted and two unique labels are created. A counter indicating the nesting level of the structure is incremented by one. Another counter which records the number of CASE structures currently started but not finished and a counter indicating which case within the current CASE structure is being processed are incremented. A single conditional branch is created which branches to the second unique label created in this routine if the value of the index of the CASE is not equal to one (the value of the case counter for this CASE structure). This statement is inserted into the output after the CASE statement in the form of a comment. The index of the CASE, the label which was the destination of the conditional branch, and the first unique label created in this routine are pushed onto the semantic stack.

Semantic Routine 3 - DO statements are processed by this semantic routine. The items of the DO -- the index, beginning value, ending value, and increment value -- are broken out of the statement. If no increment value is specified, a default value of 1 is used. A unique label and a unique identifier are created, and the counter
indicating the nesting level of the structure is incremented. Two assignments with subtraction and one assignment with addition are created and inserted in the output. The first subtraction assigns the unique identifier the value of the ending value less the increment value. The second subtraction assigns the index the beginning value less the increment value. The addition has the unique label and assigns the index the value of the index plus the increment. The DO statement is made into a comment and inserted into the output, followed by the three created statements. The increment value, the index of the DO, the unique identifier, and the unique label are pushed onto the semantic stack.

Semantic Routine 4 - A DOGROUP statement is processed by this routine. The DOGROUP statement is made into a comment and inserted in the output. The counter indicating the structure nesting level is incremented.

Semantic Routine 5 - When the parser recognizes an IF - THEN statement, this routine is called to process it. The Boolean condition is extracted from the statement and a unique label is created. A conditional branch which tests the negation of the Boolean condition and branches to the unique label on truth of the negation is inserted in the output following the IF - THEN statement in the form of a comment. The unique label is pushed onto the semantic stack.

Semantic Routine 6 - This routine processes a REPEAT UNTIL statement. The Boolean condition is extracted, a unique label is created, and the nesting level counter is incremented. A null statement with the unique label is created and inserted after a comment containing the REPEAT UNTIL. The unique label and the Boolean condition are pushed onto the semantic stack.
Semantic Routine 7 - A WHILE statement is processed by this semantic routine. The Boolean condition of the WHILE is extracted and two unique labels are created. The nesting level counter is incremented. A conditional branch with the first unique label is created which branches to the second unique label if the negation of the Boolean condition is true. The WHILE statement is converted to a comment and inserted in the output, followed by the created statement. The second and first unique labels are pushed onto the semantic stack, in that order.

Semantic Routine 8 - This semantic routine processes the QUITLOOP extended construct. A unique label is created and an unconditional branch to that label is created. The QUITLOOP statement is converted to a comment and inserted in the output, followed by the created branching statement. The unique label and the string 'QUIT' are pushed onto the semantic stack.

Semantic Routine 9 - The parser calls this routine when an ENDCASE has been recognized. Three items are popped off the semantic stack. Two null statements are created with the second and first items from the stack as labels. The number of cases for this CASE structure is reset to zero. The number of CASE structures and the nesting level are both decremented by one. The created statements are inserted following the ENDCASE in the form of a comment.

Semantic Routine 10 - This semantic routine is called by the parser to process an ENDIF. One item is popped from the semantic stack. There are two possible types of values for this item, the string 'QUIT' or a label. If the item is the character string 'QUIT', two more items are popped off the stack and the first of
those two items and the string 'QUIT' are pushed back onto the stack. If the item is not the string 'QUIT', it should be a label.

A null statement using the item from the stack is created and inserted into the output following the ENDIF in the form of a comment.

Semantic Routine 11 - This semantic routine processes the ENDWHILE statement. Two items are popped off the semantic stack and the nesting level counter is decremented by one.

If the first item popped off the semantic stack is the character string 'QUIT', the second item is the label which was the destination of the branch created for a QUITLOOP structure occurring within the WHILE - ENDWHILE construct. The variable indicating the presence of an enclosed QUITLOOP structure is set to one. Two more items are popped off the semantic stack. These items are now considered to be the first and second items popped off the stack.

Whether or not a QUITLOOP structure has occurred, an unconditional branching statement is created with the first item popped off the stack as the destination. A null statement with the second item popped off the stack as the label follows the unconditional branch. These two statements are inserted in the output following the ENDWHILE statement in the form of a comment.

If the QUITLOOP structure indicator was set, another null statement is created. This statement has the label which was the destination of the unconditional branch created for the QUITLOOP occurrence and is inserted in the output.
Semantic Routine 12 - The ENDREPEAT statement is processed by this semantic routine. Two items are popped off the semantic stack and the nesting level counter is decremented by one.

If the first item popped off the semantic stack is the character string 'QUIT', the second item is the label which was the destination of the branch created for a QUITLOOP structure occurring within the REPEAT UNTIL - ENDREPEAT construct. The QUITLOOP occurrence indicator is set to 1 and two more items are popped off the semantic stack. These items are now considered to be the first and second items popped off the stack.

Whether or not a QUITLOOP structure has occurred, the second item popped off the stack is used as the destination of a conditional branching statement which tests the negation of the first item popped off the stack. The ENDREPEAT is converted to a comment and inserted in the output, followed by the created conditional statement.

If the QUITLOOP occurrence flag was set, a null statement is created with the label used as the destination of the branching statement which replaced the QUITLOOP statement. This null statement is inserted in the output.

Semantic Routine 13 - The parser calls this semantic routine when an ENDDO has been recognized. Four items are popped from the semantic stack and the nesting level counter is decremented by one.

If the first item popped off the semantic stack is the character string 'QUIT', the second item is the label created as the branching destination during QUITLOOP processing. Two more items are popped from the semantic stack. These are added to the third and fourth
items popped previously, and the four items are considered to be
the first four items popped from the stack. The QUITLOOP occurrence
flag is set to one.

Whether or not a QUITLOOP structure has occurred, two unique
labels are created. Five statements are created to be inserted in
the output following the ENDDO statement in the form of a comment.

The first statement is a conditional branch which tests if the
increment value (the fourth item popped from the stack) is positive.
The destination of this statement is the first of the two new unique
labels. Another conditional branching statement is the second
created statement. It has the first item popped off the stack as its destination and tests if the index value (third item from the stack) is greater than or equal to the second item popped from the stack. An unconditional branch with the second unique label as its destination is created next and is followed by another conditional branch. This statement has the first unique label as its label, tests if the index value is less than or equal to the second item popped from the stack, and branches to the first item popped off the stack if the condition is true. The fifth created statement is an assignment statement with an addition operation. The statement's label is the second new unique label and it increments the index value by the increment value.

If the QUITFLAG occurrence flag was set, a null statement is created and inserted in the output. The label on this statement is the destination label of the unconditional branch created during QUITLOOP processing.
Semantic Routine 14 - This semantic routine processes the ENDDOGROUP statement. The statement is converted to a comment and inserted in the output. The nesting level counter is decremented by one.

Semantic Routine 15 - This semantic routine is called whenever the parser reduces an item or series of items to the STRLIST (structure list) token.

The routine returns immediately if there are no unfinished CASE structures or if the preprocessor is processing statements within another structure contained in a CASE structure. Otherwise, the number of cases for this CASE structure is incremented by one, and a unique label is created. Three items are popped from the semantic stack.

An unconditional branch with the first item popped from the semantic stack as the destination is created. A conditional branch with the second item from the semantic stack as the label is created next. This statement tests if the third item from the semantic stack (index of the CASE) is not equal to the counter incremented above and branches to the new unique label if the condition is true.

A comment noting that two statements are being inserted is created and inserted in the output. The two branching statements are inserted immediately following the comment.

The index of the CASE, the new unique label, and the first item popped from the stack are now pushed onto the stack, in that order.

Insertion Routines

There are three routines which insert strings into the output, each in its own way.
Insertcom - The single parameter COMSTR contains a string to be converted to a comment and inserted in the output. The routine will attempt to start the comment in column one. If this column is to be ignored, the routine determines where to start the comment. If a sequence of columns has been specified for the ignored columns in the IGNORE COLUMNS definition, the preprocessor discovers the first column which is not an ignored column. If no sequence was specified, the preprocessor uses the first column following the ignored column.

If a comment begins in a specific column, the columns preceding that column are filled with blanks. If a symbol string starts the comment in that column, the string is added, followed by the parameter. If no symbol string is needed, the parameter is added to the comment being built. The comment is truncated, if necessary, to eighty characters.

If a comment ends in a particular column, the string is truncated or padded with blanks, as necessary. If a comment ends with a certain string, the current comment string is truncated as necessary to maintain only eighty characters. If the comment will end in an ignored column, enough characters are deleted within the parameter string portion of the comment to ensure the comment ender will not appear in an ignored column. The comment closing string is added after the appropriate position has been determined. Enough blanks are added, if necessary, to make the final string eighty characters long. Finally, the comment is inserted in the output.

Insertstmt - The single parameter STR.INSRT contains the string to be inserted in the output as a statement. A counter of the positions in the output string (I) and a counter of the positions
in the parameter string (CHARKTR) are set to zero. The variable TYPE is set depending on whether a single column or a sequence of columns was specified in the IGNORE COLUMNS definition.

CHARKTR and I are both incremented by one. The final portion of the statement has been constructed if CHARKTR is equal to the number of characters in the parameter string. An entire card has been constructed if I is equal to 80. Branches are made if either of these conditions occurs.

A branch is made to check the current position in the output (value of I) against ignored columns. If only one column is ignored and the column currently indicated by I is that column, a blank is inserted. If a sequence of columns is specified and the column indicated by I is one of the ignored columns, a blank is inserted. A branch is made to increment I and continue processing this statement. In either case, if the column to which I points is a usable column, the character in the position of the parameter string indicated by CHARKTR is added to the output string being constructed and a branch is made to increment both CHARKTR and I.

When an entire card image has been constructed, it is inserted in the output. I is reset to 1 and the card image area is nullified. The preprocessor then branches to check for a column to ignore.

When the last portion of a statement has been completed, this portion is padded with blanks and inserted in the output. Insert - The single parameter INSERT.STR contains the statement to be inserted. This string contains blanks in ignored columns. The statement is broken into sections of eighty characters and
inserted in the output one at a time. If the last section of the statement is less than eighty characters long, this portion is padded with blanks before inserting it in the output.

Parser

The structure parser has no parameters. It is called once by the driver for each parse desired. Control is returned to the point of invocation when the entire input program has been analyzed. Error conditions take special error exits out of the parser.

When the parser is called, it immediately requests a token be placed on the syntactic stack by the scanner. The parser then tests the contents of the syntactic stack, reduces those contents to appropriate tokens, and calls semantic routines and the scanner according to the strategy of the modified Production Language in Table A.1.3.

Scanner

The scanning routine retrieves single characters from the input program and determines what syntactic tokens are to be presented to the parser. A character string is built of the input characters, and when an item is recognized, a token is placed on the syntactic stack and return is made to the parser. The scanner makes tests in the following order: comment, literal, end of statement, keyword appearing at end of statement, end of a language label, blank character, extended construct keyword followed by a blank. If the scanner is in the middle of a statement, the test for end of a language label is skipped. If the end of statement condition is not recognized, the test for a
<table>
<thead>
<tr>
<th>LINE NO</th>
<th>STACK TOP</th>
<th>REDUCE TO</th>
<th>CALL SEMANTIC ROUTINE</th>
<th>SCAN</th>
<th>ON SUCCESS, BRANCH TO TEST ON LINE #</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>LABEL</td>
<td>LABEL</td>
<td>ROUT0</td>
<td>yes</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>ENDCASE</td>
<td>ENDCASE</td>
<td>--</td>
<td>yes</td>
<td>26</td>
</tr>
<tr>
<td>3</td>
<td>ENDIF</td>
<td>ENDIF</td>
<td>--</td>
<td>yes</td>
<td>27</td>
</tr>
<tr>
<td>4</td>
<td>ENDWHILE</td>
<td>ENDWHILE</td>
<td>--</td>
<td>yes</td>
<td>29</td>
</tr>
<tr>
<td>5</td>
<td>ENDBEPEAT</td>
<td>ENDBEPEAT</td>
<td>--</td>
<td>yes</td>
<td>30</td>
</tr>
<tr>
<td>6</td>
<td>ENDDO</td>
<td>ENDDO</td>
<td>--</td>
<td>yes</td>
<td>31</td>
</tr>
<tr>
<td>7</td>
<td>ENDDGROUP</td>
<td>ENDDGROUP</td>
<td>--</td>
<td>yes</td>
<td>32</td>
</tr>
<tr>
<td>8</td>
<td>ELSE</td>
<td>ELSE</td>
<td>ROUT1</td>
<td>yes</td>
<td>1</td>
</tr>
<tr>
<td>9</td>
<td>QUITLOOP</td>
<td>QUITLOOP</td>
<td>--</td>
<td>yes</td>
<td>25</td>
</tr>
<tr>
<td>10</td>
<td>CASE</td>
<td>CASE</td>
<td>--</td>
<td>yes</td>
<td>18</td>
</tr>
<tr>
<td>11</td>
<td>DO</td>
<td>DO</td>
<td>--</td>
<td>yes</td>
<td>19</td>
</tr>
<tr>
<td>12</td>
<td>DOGROUP</td>
<td>DOGROUP</td>
<td>--</td>
<td>yes</td>
<td>20</td>
</tr>
<tr>
<td>13</td>
<td>IF</td>
<td>IF</td>
<td>--</td>
<td>yes</td>
<td>21</td>
</tr>
<tr>
<td>14</td>
<td>REPEAT</td>
<td>REPEAT</td>
<td>--</td>
<td>yes</td>
<td>22</td>
</tr>
<tr>
<td>15</td>
<td>WHILE</td>
<td>WHILE</td>
<td>--</td>
<td>yes</td>
<td>24</td>
</tr>
<tr>
<td>16</td>
<td>STRLIST EOF</td>
<td>PROG</td>
<td>--</td>
<td>no</td>
<td>return</td>
</tr>
<tr>
<td>17</td>
<td>UEOCS</td>
<td>STRUC</td>
<td>ROUT01</td>
<td>no</td>
<td>33</td>
</tr>
<tr>
<td>18</td>
<td>CASE UEOCS</td>
<td>CASESTMT</td>
<td>ROUT2</td>
<td>yes</td>
<td>1</td>
</tr>
<tr>
<td>19</td>
<td>DO UEOCS</td>
<td>DOSSTMT</td>
<td>ROUT3</td>
<td>yes</td>
<td>1</td>
</tr>
</tbody>
</table>

Table A.I.3. Production Language for Parser.
<table>
<thead>
<tr>
<th>LINE NO</th>
<th>STACK TOP</th>
<th>REDUCE TO</th>
<th>CALL SEMANTIC ROUTINE</th>
<th>SCAN</th>
<th>ON SUCCESS, BRANCH TO TEST ON LINE #</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>DOGROUP UEOS</td>
<td>DOGROUPSTMT</td>
<td>ROUT4</td>
<td>yes</td>
<td>1</td>
</tr>
<tr>
<td>21</td>
<td>IF UTHEN</td>
<td>IFTHEN</td>
<td>ROUT5</td>
<td>yes</td>
<td>1</td>
</tr>
<tr>
<td>22</td>
<td>REPEAT UNTIL</td>
<td>REPUNUTIL</td>
<td>--</td>
<td>yes</td>
<td>23</td>
</tr>
<tr>
<td>23</td>
<td>REPUNUTIL UEOS</td>
<td>REPEATSTMT</td>
<td>ROUT6</td>
<td>yes</td>
<td>1</td>
</tr>
<tr>
<td>24</td>
<td>WHILE UEOS</td>
<td>WHILESTMT</td>
<td>ROUT7</td>
<td>yes</td>
<td>1</td>
</tr>
<tr>
<td>25</td>
<td>QUITLOOP UEOS</td>
<td>STRUC</td>
<td>ROUT8</td>
<td>no</td>
<td>33</td>
</tr>
<tr>
<td>26</td>
<td>CASESTMT STRLIST ENDOCASE UEOS</td>
<td>STRUC</td>
<td>ROUT9</td>
<td>no</td>
<td>33</td>
</tr>
<tr>
<td>27</td>
<td>IFTHEIL ENDIF UEOS</td>
<td>STRUC</td>
<td>ROUT10</td>
<td>no</td>
<td>33</td>
</tr>
<tr>
<td>28</td>
<td>IFTHEISE ENDIF UEOS</td>
<td>STRUC</td>
<td>ROUT10</td>
<td>no</td>
<td>33</td>
</tr>
<tr>
<td>29</td>
<td>WHILE STMT STRLIST ENDOWHILE UEOS</td>
<td>STRUC</td>
<td>ROUT11</td>
<td>no</td>
<td>33</td>
</tr>
<tr>
<td>30</td>
<td>REPEATSTMT STRLIST ENDOREPEAT UEOS</td>
<td>STRUC</td>
<td>ROUT12</td>
<td>no</td>
<td>33</td>
</tr>
<tr>
<td>31</td>
<td>DOSTMT STRLIST ENDDO UEOS</td>
<td>STRUC</td>
<td>ROUT13</td>
<td>no</td>
<td>33</td>
</tr>
<tr>
<td>32</td>
<td>DOGROUPSTMT STRLIST ENDDOGRP UEOS</td>
<td>STRUC</td>
<td>ROUT14</td>
<td>no</td>
<td>33</td>
</tr>
<tr>
<td>33</td>
<td>LABEL STRUC</td>
<td>STRUC</td>
<td>--</td>
<td>no</td>
<td>33</td>
</tr>
<tr>
<td>34</td>
<td>ELSE STRUC</td>
<td>ELSE STRUC</td>
<td>--</td>
<td>no</td>
<td>33</td>
</tr>
</tbody>
</table>

Table A.I.3, Production Language for Parser (cont’d).
keyword appearing at the end of a statement is skipped. If the blank character test is not satisfied, the test for an extended construct keyword followed by a blank is skipped. A minimum amount of backup is used in the scanner. A more detailed description is contained in the remainder of this section.

The scanner begins by recording whether or not an end-of-statement condition (e-o-s) was encountered in the last card scanned. If the variable EOSNEXT is equal to one, that is, the e-o-s condition should be signalled immediately, a branch is taken to return an e-o-s condition. If EOSNEXT is not one, the GETCHAR function is called to return a character and its column number. If this character is the first character of a token or the first non-blank character of a token, the starting column of the token is recorded. If this column is the first non-ignored column of a card, it is determined whether or not all blanks left over from a previous card should be discarded. If only a comment was found on the previous card, these blanks will be retained. Another condition under which the blanks will be retained is when a comment was not the only thing found on the previous card and an e-o-s condition was found in the previous call to the scanner.

Whether or not the blanks are retained, the next step is to add the new character to the current token. The label start position variable and the comment only indicator are set to zero.

The scanner then examines the token string for the beginning of a comment. There are three different forms of comment openers to look for, and the scanner uses whichever one corresponds to the definition provided by the user. When the column opening string is two characters
long, it is necessary for the preprocessor to inspect the character following an occurrence of the first character of the string. If the second character is not the second of the specified string, a backup procedure is invoked by restoring the first character thought to open the comment and its column number. If a comment opener is not recognized, the scanner branches to test for a literal. If the beginning of a comment is found, the e-o-s flag is reset to the value it contained before the call to the scanner, the comment-only indicator is set to one if nothing else has been encountered in this statement, and a search is made for the end of the comment.

The search for the end of the comment also has three possible methods and is controlled by the definition of a comment for this particular language. If the comment closer is a string of two characters, the backup procedure will again be utilized if necessary. When the end of the comment is found, the token, which consists of the text of the comment and any opening and/or closing strings, is added to the current statement string, and the token variable is set to null. A branch is then taken to the point near the beginning of the scanner where another character is retrieved, thus avoiding the manipulation and test of the e-o-s indicator. The comment is not returned to the parser.

If the test for a comment is unsuccessful, the scanner analyzes the input characters for a literal. The literal beginning may be defined as one or two characters. If a two-character opening is used, the backup procedure is used if necessary. If the beginning of a
literal is not recognized, the scanner branches to test for an

e-o-s condition.

If the literal opening is found, a search is made for the end of

the literal. In searching for the end, special processing is utilized

for any special sequence specified in the literal definition. If the

first character of the ending string, which is the first character of

any special sequence, is encountered, a test is made for the special

sequence if one was provided. If a special sequence is found, the

scanner continues to search for the literal end. Otherwise, a test is

made for the end of the literal at that point. If a literal ender is

not found, the scanner continues to search for it. If the end is

recognized, the literal is added to the token string and a branch is

taken to the portion of the scanner which processes items not identi-

fied as extended constructs or comments.

The next section of the scanner tests for the e-o-s condition.

The scanner utilizes the information provided by the user in the

end of statement definition to analyze the input in this section.

If the end of statement is signified by one or two symbols, the scanner

tests for these characters, using the backup procedure for a two-character

string if necessary. If the end of statement is signalled by a

column, the scanner tests if the column number of the most recently

retrieved character is that specified in the end-of-statement definition.

If the end-of-statement condition is not recognized, a branch is taken

to test for a language label. If what appears to be an end-of-statement

condition is recognized and that condition is indicated by a column
number, the next input card is read and tested to determine if it is a continuation. If so, a branch is taken to the section of the scanner which tests for a blank character.

If the end-of-statement condition is recognized, the token string is added to the current statement. It is necessary at this point to determine if an extended construct keyword ends at the end of the statement but has not been recognized. This will be the case if e-o-s is indicated by a symbol and a keyword ends immediately preceding that symbol or if e-o-s is indicated by column number and a keyword ends immediately preceding a blank in the ending column or in the ending column itself. If a keyword is recognized in this manner, it is necessary to set the EOSNEXT variable to one so the scanner will signal e-o-s immediately upon its next invocation. The keyword found is added to the syntactic stack and control is returned to the parser. If a keyword is not found at the end of the statement, the e-o-s flag is set for this statement, the EOSNEXT flag is reset to zero, the 'unidentified to end of statement' (UEOS) token is added to the syntactic stack, the token string is set to null and control is returned to the parser.

The next portion of the scanner tests for the end of a language label if this is at the beginning of a statement. If there is anything in the current statement string followed by a blank or only blanks occur in the current statement string, the test for the end of a language label is not made, and a branch is made to the section of the scanner which tests for a blank character.

The test for the end of a language label is based on the definition provided for language labels. If the definition specifies a column
number or a series of column numbers as the determination of the end of a language label, the first test will be satisfied if the current column is the single column number or within the specified sequence. Otherwise, a branch is taken to the section of the scanner which tests for a blank character.

If the definition specifies a symbol to terminate a language label, the current character is tested against the specified symbol. If a two-character language label separator is specified, the next character is tested and backup is used if necessary to restore the previous character as the current character. If the symbol string does not match, a branch is taken to the section of the scanner which tests for a blank character.

Once the scanner has determined that this character/column could be terminating a language label, the label is extracted from the token string. It is either the entire token or the last portion of it. The column number the label actually started in is computed. The column number computed is compared with the starting column value(s) specified in the language label definition. If this column number cannot start a language label, any necessary backup is performed and a branch is taken to the section which tests for a blank character. At this point the scanner knows a label has been found if a string is the label terminator. If the label is terminated by column numbers, the first character of the label is checked to see if it is a valid character for the start of a label. Any remaining characters are checked against the characters allowable for the remainder of the label. If either of these tests fails, any necessary backup is performed
and a branch is taken to the section of the scanner which tests for a blank character. If these two tests are passed, the next input character is tested for a label delimiter, that is, a blank or a special character which is allowed directly following a label. If this test fails, backup is performed and a branch is taken to the section which tests for a blank character.

If a label is recognized by the scanner, the label is placed in a comment indicating that it was removed from the statement and placed on a null statement inserted immediately preceding the current statement. This insertion function will be performed by the semantic routine which will be invoked when the scanner returns control to the parser. The label's character positions in the current statement are replaced by blanks, the LABEL token is placed on the syntactic stack, and return is made to the parser.

The next section of the scanner tests for a blank character. A blank encountered at this point in the scanner's execution causes a branch to the section to search for a keyword. If the character is any non-blank character, a branch is made to the BEGSCAN1 label in the scanner. The effect of this branch is to allow the scanner to read the next input character and begin its analysis once again.

The next section tests for a keyword. The first of two tests compares the token string with extended construct keywords which begin or end constructs, such as DO and ENDDO. The second test is against the extended construct keywords which appear in the middle of extended constructs, that is, the keywords THEN and UNTIL. If the token string does not satisfy either of these tests, a branch is taken to the section of the scanner which handles items not yet
recognized. If the keyword for THEN has been recognized, the
unidentified-to-then (UTHEN) token is added to the syntactic stack.
Otherwise, the keyword itself is added to the syntactic stack. The
token string is added to the current statement, the token string is
set to null, and control is returned to the parser.

The section which processes unidentified items simply branches
to the BEGSCAN1 label in the scanner to read the next input character
and begin its analysis again.

A section at the end of the scanner tests for an error condition
when the scanner attempts to retrieve another input character at the
beginning of the scanner but fails. If the token string is currently
all blanks or is of zero length, a test is made to determine whether
or not an e-o-s condition was found before this call to the scanner.
If so, the failure to return a character indicates the end-of-file
condition. The ECF token is added to the syntactic stack and control
is returned to the parser. If an e-o-s condition had not been discovered,
an unended statement occurs at the end of the input and an error exit
is taken. If the token string contains at least one non-blank
character when the failure to return a character is encountered, an
error exit is taken.

Character Retrieval Routine

The GETCHAR routine retrieves a single character from the input
program and returns the character and its column number to the scanner.
The two parameters are CHAR and COL, which will contain the character
and its column number, respectively, on a successful return. These
arguments are passed as variable names in the form of character strings and one level of indirection is used to assign the appropriate values to them.

The routine first increments the column number counter (global variable COLNUM) modulo 80. If the resultant column number is 1, the routine begins retrieving characters from the next card. This card may have already been read by the scanner in testing for a continuation card. If the card is not available, the routine tests if the end of file condition has been encountered previously, either by this routine or by the scanner in testing for a continuation card. If not, the next card is read into the program. Otherwise, the card which was read for the continuation test is copied into the variable from which the routine retrieves characters. If the end of file condition is encountered when reading a new card in this routine, the GETEND variable is set to 1 and a return with failure is made.

To retrieve a character, the routine tests if the current column is an ignored column as specified in the user's IGNORE COLUMNS definition. If this column is to be ignored, a branch is made to the beginning of the routine so the next character can be analyzed. If this column is a usable one, the character in that column and the column number are assigned to the appropriate arguments and a successful return is made to the scanner.
APPENDIX II. DETAILED USER INFORMATION

The three types of input to the preprocessor are information about the base language, macro definitions (optional), and a program written in the extended version of a programming language. The first type consists of a keyword override card and definitions given in a form similar to Backus-Naur Form. The definitions must be grouped together, and the override card may either precede or follow the definitions.

The override card contains the string 'OVERRIDE:' starting in column one. The keywords to be overridden follow the colon and are separated by commas. The first blank indicates the end of the keywords.

EXAMPLES:

OVERRIDE:IF,THEN,ELSE,ENDIF
  overrides the keywords IF, THEN, ELSE, AND ENDIF

OVERRIDE:WHILE
  overrides the keyword WHILE

All the definitions about the base language begin with the definition name enclosed in angle brackets and followed by ::=.
The definition name and the remainder of the contents of each definition are described below.

UNIQUE LABEL -- The information needed to create unique labels is as follows:

  1) Uniform beginning -- a sequence of one or more characters
     which conforms to the rules for writing labels in the
standard language but which is shorter than the maximum allowed length. The preprocessor will use this uniform beginning to begin any labels it creates.

2) Variable digits -- an integer specifying how many digits should be used to make the labels unique. The preprocessor will begin counting at 1 using these digits (filling with leading zeros) and will imbed them in any labels it creates.

3) Remainder -- a sequence of zero or more characters which conform to the rules for ending labels. This sequence should not include any character(s) used to separate a label from the statement with which it is associated.

The uniform beginning must immediately follow the = and at least one blank must separate the components of the definition.

EXAMPLE:

(for FORTRAN) \langle UNIQUE LABEL \rangle ::= 24 3

This example will create labels such as 24001 and 24002.

UNIQUE IDENTIFIER -- The information needed to create unique variable names is as follows:

1) Uniform beginning -- a sequence of one or more characters which conforms to the rules for naming identifiers in the standard language but which is shorter than the maximum allowed length. The preprocessor will use this uniform beginning to begin the names of any new identifiers it creates.

2) Variable digits -- an integer specifying how many digits should be used to make the identifiers unique. The preprocessor will begin counting at 1 using these digits
(filling with leading zeros) and will imbed the digits in any identifier names it creates.

3) Remainder -- a sequence of zero or more characters which conforms to the rules for ending identifier names.

The uniform beginning must immediately follow the = and at least one blank must separate the components of the definition.

EXAMPLE:

(for PL/I) \texttt{<UNIQUE IDENTIFIER> ::=LAB 3 XY}

This example will create identifiers such as LAB001XY and LAB002XY.

LITERAL -- The information the user must provide about literals is as follows:

1) Literal opener -- a one- or two-character symbol string which begins a literal.

2) Literal closer -- a one- or two-character symbol string which ends a literal.

3) Special sequence (optional) -- a two-character symbol string which does not terminate a literal but would cause an error in determining where a literal ends if no special handling were provided because the first character of this special sequence is the same as the single character which normally terminates a literal. An example of this is the string "" which may appear within a literal in some languages in which a single quote normally terminates a literal.

The literal opener must directly follow the = and at least one blank must separate the definition components.
EXAMPLE:

If ' is both the literal opener and literal closer, and '' is a sequence requiring special handling (as in PL/I):

\[ \text{LITERAL} ::= \text{' ' ' ' } \]

COMMENT -- The information the user must provide about comments is as follows:

1) Comment opener -- a one- or two-character symbol string in single quotes which begins a comment; or a column number in which a comment begins; or a one- or two-character symbol string in single quotes which begins a comment followed by a + and the column number in which the symbol string must start to indicate a comment.

2) Comment closer -- a one- or two-character symbol string in single quotes which ends a comment; or a column number in which a comment ends; or a one- or two-character symbol string in single quotes which ends a comment followed by a + and the column number in which the symbol string must start to terminate a comment.

The comment opener must directly follow the = and at least one blank must separate the definition components.

EXAMPLE:

If a C must appear in column 1 to indicate a comment, and a comment ends in column 72 (as in FORTRAN):

\[ \text{COMMENT} ::= \text{C' + 1 72 } \]

LANGLABEL -- The information the user must provide about labels on statements is as follows:
1) Label opening delimiter -- the column number of the statement in which the label must begin; or a series of column numbers within which the label must begin represented by two column numbers separated by a hyphen.

2) Label closing delimiter -- a one- or two-character symbol string enclosed in single quotes which separates a label from the statement with which it is associated; or a column number in which a label must end; or a series of column numbers within which a label must end represented by two column numbers separated by a hyphen.

3) Starting character set -- a set of characters enclosed in single quotes which may start a label.

4) Remaining character set -- a set of characters enclosed in single quotes which may be used in all other character positions of a label.

The opening delimiter must begin directly following the = and at least one blank must separate the components of the definition.

**EXAMPLES:**

(for FORTRAN)

\[ \langle \text{LANGLABEL} \rangle ::= 1-5 \ 1-5 \ '0123456789' \ '0123456789' \]

(for PL/I)

\[ \langle \text{LANGLABEL} \rangle ::= 2-72 \ ': ' \ 'ABCDEFGHIJKLMNOPQRSTUVWXYZ\$' \ 'abcdefghijklmnopqrstuvwxyz\$' \ 'ABCDEFGHIJKLMNOPQRSTUVWXYZ\$\_' \]

(for SYMBOL)

\[ \langle \text{LANGLABEL} \rangle ::= 1 \ 1-72 \ 'ABCDEFGHIJKLMNOPQRSTUVWXYZ\$' \ 'abcdefghijklmnopqrstuvwxyz\$' \]

'ABCDEFGHIJKLMNOPQRSTUVWXYZ\$\_'
GREATER THAN, GREATER THAN OR EQUAL,
LESS THAN OR EQUAL, NOT EQUAL, NOT -- The operator for each logical operation should be enclosed in single quotes and directly follow the =.

EXAMPLES:
(for PL/I) "NOT EQUAL" ::= '='
(for SNOBOL) "LESS THAN" ::= 'LT'
(for FORTRAN) "NOT" ::= ', NOT.'

IGNORE COLUMNS -- The information the user must provide about columns to ignore in creating and scanning statements is as follows:

Column numbers -- the number of a column to be completely ignored; or a series of column numbers for columns to be ignored represented by two column numbers separated by a hyphen.

The definition item must immediately follow the =.

EXAMPLE:
(for SNOBOL, FORTRAN, and PL/I) "IGNORE COLUMNS" ::= 73-80

END OF STATEMENT -- The information the user must provide about the end of statement condition is as follows:

End of statement indicator -- a one- or two-character symbol string in single quotes which indicates the end of a statement; or a column number which terminates a statement (either a regular statement or on a continuation card).

The indicator must immediately follow the =.

EXAMPLES:
(for SNOBOL) "END OF STATEMENT" ::= 72
(for PL/I) "END OF STATEMENT" ::= ';'
CONTINUATION -- The information the user must provide about continuation cards is as follows:

Continuation symbol -- a single character followed by the column number in which that symbol must appear to indicate a continuation card; or the string NONE to indicate that statements may cross card boundaries without specifically indicating continuation.

The continuation symbol or the word NONE must immediately follow the =. If a symbol is indicated, at least one blank must separate it from the column specification.

EXAMPLES:

(for SNOBOL) \langle CONTINUATION \rangle ::= 1
(for PL/I) \langle CONTINUATION \rangle ::= NONE

GO TO STATEMENT, IF THEN GO TO STATEMENT,
ASSIGN MINUS STATEMENT, ASSIGN PLUS STATEMENT, DECLARATION STATEMENT --

In every statement definition, there are certain conventions which must be followed. At least one blank must appear between a parameter name and the items on either side of it. A parameter name may, however, begin in column 1 of the prototype, i.e., directly following the =. The parameter names must be spelled exactly as specified in the description of each definition.

The definition names of the statements, their function, and their parameters are as follows:

NULL STATEMENT -- a null statement.

Parameter names:

LAB -- label on the null statement
GO TO STATEMENT -- a branching statement.

Parameter names:

LAB -- label on the branching statement
DES -- label of the statement which is the destination of the branch

IF THEN GO TO STATEMENT -- a conditional statement in which the result of a true value is a branch.

Parameter names:

LAB -- label on the conditional statement
VAL1 -- first value in the condition
RL -- relational operator in the condition
VAL2 -- second value in the condition
DES -- label of the statement which is the destination of the branch

ASSIGN MINUS STATEMENT -- an assignment statement in which the expression being evaluated is a subtraction.

Parameter names:

LAB -- label on the assignment statement
VAR -- identifier to be assigned a value
VAL1 -- minuend
VAL2 -- subtrahend

ASSIGN PLUS STATEMENT -- an assignment statement in which the expression being evaluated is an addition.

Parameter names:

LAB -- label on the assignment statement
VAR -- identifier to be assigned a value
VAL1 - first addend

VAL2 - second addend

DECLARATION STATEMENT -- a declaration statement for one variable which is global in scope and is of type integer.

Parameter names:

VARNAME -- identifier to be declared.

Note: If declarations of identifiers are not required, a blank prototype must be provided.

The first column following the equal sign, column 1 of the prototype, is treated as column 1 of a card image. Therefore, if a label must appear in a particular position and the remainder of the statement must appear in a particular position, the definition must appear in the same form with enough blanks included to position items correctly. Since the user also defines the length of identifier names and labels created by the preprocessor, the user must be sure to allow enough space for these to fit if the parameter name is shorter than the item which will be created to replace it. In some cases, a user's variable will be used in the place of a parameter name. Consequently, if blank spaces are critical, it may be wise to use identifiers with the same number of characters in the names. Since the definition name takes up some space and the preprocessor requires an 80-column prototype, two cards will be needed for each definition. Even if the prototype provided by the user could fit on one card with the definition name, it is necessary to provide two cards, leaving the second one blank.

All variables introduced by the preprocessor are integers and have global scope. Consequently, declarations for these new variables should be placed wherever declarations for global variables are normally
placed in a program written in the base language. To allow the preprocessor to position these declarations correctly a special input statement must be included in the extended program containing only the string ADDED DECLARATIONS. When the preprocessor is ready to add declarations, it will replace this statement with the appropriate declarations. If declarations are not required and such has been specified in the declaration definition, the 'ADDED DECLARATIONS' statement should be omitted.

EXAMPLES:

(for FORTRAN) '('NULL STATEMENT') ::=LAB CONTINUE

(for SNOBOL) '('GO TO STATEMENT') ::=LAB : (DES)

(for PL/I) '('IF THEN GO TO STATEMENT') ::=LAB :

IF VAL1 RL VAL2 THEN GO TO DES ;

(for PL/I) '('ASSIGN MINUS STATEMENT') ::=LAB :

VAR = VAL1 - VAL2 ;

(for SNOBOL) '('ASSIGN PLUS STATEMENT') ::=LAB :

VAR = VAL1 + VAL2

(for SNOBOL) '('ASSIGN PLUS STATEMENT') ::=LAB :

VAR = VAL1 + VAL2

(for FORTRAN) '('DECLARATION STATEMENT') ::=LAB INTEGER VARNAME

Macro definitions have three parts, the macro name and parameters, the replacement text, and an indication of the end of the definition. The first card of a macro definition begins in column 1 with the string '('MACRO CALL') ::=' followed immediately by the macro name and a list of parameter names enclosed in parentheses. The parameter names are separated by commas and no inbedded blanks are allowed. The next card starts with the string '('MACRO DEFINITION') ::=' starting in column 1. The remainder of the card is left blank. The next card is the beginning
of the replacement text for the macro. The last card of the replacement text is followed by a card containing only the string 'END MACRO DEFINITION' starting in column 1.

The parameter names appearing in the replacement text will be replaced by the actual arguments of a macro call during code expansion by the preprocessor. Because of storage limitations, there can be no more than twenty components in a macro replacement text, where a component is defined to be a parameter name and following blanks or a string of characters appearing between parameter names.

EXAMPLES:

\[
\langle \text{MACRO CALL} \rangle ::= \text{SWAP}(X, Y) \\
\langle \text{MACRO DEFINITION} \rangle ::= \\
\quad T = X \\
\quad X = Y \\
\quad Y = T \\
\]

END MACRO DEFINITION

In the above example, SWAP is the macro name. X and Y are the formal parameter names. When the preprocessor expands the call SWAP(A,B), the following code will be inserted:

\[
\quad T = A \\
\quad A = B \\
\quad B = T \\
\]

This macro definition contains the following seven components:
If this macro were to be used in the extended version of SNOBOL, it could not be called with arguments longer than one character because the space before the equal sign would have to be retained. In FORTRAN the arguments could be one or two characters long.

EXAMPLE:

The call SWAP(A1,A2) would result in the following inserted code:

\begin{verbatim}
T = A1
A1 = A2
A2 = T
\end{verbatim}

No argument may be of longer length than the corresponding parameter name and the blanks immediately following it. A call of this macro with either argument longer than two characters would cause an error during the creation of the code to be inserted.

The same macro could be defined as follows:

\begin{verbatim}
\langle MACRO CALL \rangle ::= SWAP(X,Y)
T = X
X = Y
Y = T
\end{verbatim}

END MACRO DEFINITION

Using this definition, the SWAP macro may be called with arguments of longer length than two characters.

Since the objectives of the preprocessor include encouraging the use of structured programming techniques, the following guidelines are provided for the user to use in writing macro definitions. It is felt that by following these guidelines, a user will be more
likely to write macro definitions of constructs in line with the aims of structured programming.

1) Sequentially execute statements within the definition wherever possible. In other words, avoid using branches among statements within the definition.

2) Avoid branching out of the definition wherever possible. That is, do not specify branches to labels not within the macro definition.

3) Use extended language statements defined in the preprocessor if possible.

Parameter names must be unique and no parameter name may be identical to the beginning of another parameter name of the same macro. This is due to the scanning technique used by the preprocessor in decoding the macro replacement text into its components. For example, the names ABCD and ABCDE may not be used as parameter names in the same macro definition. The preprocessor will fail to recognize ABCDE as a parameter name in the replacement text because it will always assume the first four characters constitutes an occurrence of the parameter name ABCD.

The syntax of the input program is nearly identical to the syntax of the base language involved. Many of the input program statements will be copied directly into the output. Extended statements should follow the same syntax as normal statements, even though the constructs are unknown in the base language. For example, when processing an extended FORTRAN program, the preprocessor will expect statement labels to be entirely numeric and appear in columns 1-5, continuation
to be indicated by a special symbol in column 6, and Boolean expressions to be expressed using the FORTRAN relational operators.

The syntax of the extended constructs statements will be similar to the syntax of other statements in the base language. There are specific requirements, however, which must be followed.

1) A blank must follow each extended construct keyword or the keyword must appear at the end of a statement.

2) The conditions specified in extended construct statements must be specified in forms appropriate for the base language because these conditions will be copied directly into the expanded text.

3) The conditions in the WHILE and REPEAT UNTIL statements must be enclosed in parentheses.

4) The following pairs of extended construct keywords must appear in the same statement: REPEAT and UNTIL, IF and THEN, CASE and OF. All other extended construct keywords may not appear in a statement with a second keyword. The ending keywords must appear in statements with no other text. The DOGROUP and QUITLOOP keywords are the only construct-opening keywords which must appear alone.

The special statement to position new declarations should be placed wherever the user wishes to have the declarations added. All variables introduced by the preprocessor are integers with global scope. The user should keep this in mind when inserting the special statement. The statement should contain only the string ADDED DECLARATIONS and any end-of-statement specification normally required by the base language translator.
EXAMPLE: (PL/I)

TEST: PROC OPTIONS (MAIN);

ADDED DECLARATIONS:

Y = 5;
DO I = 1 TO 3;
    Y = Y - 1;
ENDDO;
END TEST;

Restrictions

The user must keep the following restrictions in mind when using the preprocessor:

1) Only one QUITLOOP structure may be used within another loop.

2) The text following the THEN keyword and its succeeding blank must appear as a normal statement on a separate card. That is, if a specific number of columns must be blank at the beginning of a statement, that same number of columns must be blank in this situation.

3) Array elements may not be used as actual arguments in macro calls.

4) No continuation cards may be used in statements containing extended constructs.
APPENDIX III. JOB CONTROL LANGUAGE LISTINGS
EXEC SPIRTOL

* JOB STEP TO EXECUTE PREPROCESSOR FROM SOURCE DECK
* 
TEMPCH DD DSN=&TEMPCH,DCB=(BLKSIZE=800,LRECL=80,RECFM=FB), 
   SPACE=(TRK,(5,1)),UNIT=3330,DISP=(NEW,DELETE)
TEMPCH2 DD DSN=&TEMPCH2,DCB=(BLKSIZE=800,LRECL=80,RECFM=FB), 
   SPACE=(TRK,(5,1)),UNIT=3330,DISP=(NEW,DELETE)
TEMPOUT DD DSN=&TEMPOUT,DCB=(LRECL=132,BLKSIZE=132,RECFM=F), 
   UNIT=3330,SPACE=(TRK,(5,1)),DISP=(NEW,DELETE)
TEMPOUT2 DD DSN=&TEMPOUT2,DCB=(LRECL=132,BLKSIZE=132,RECFM=F), 
   UNIT=3330,SPACE=(TRK,(5,1)),DISP=(NEW,DELETE)
FINPUNCH DD SYSOUT=A,DCB=(BLKSIZE=800,LRECL=80,RECFM=FB), 
   DISP=(NEW,PASS),UNIT=3330,SPACE=(TRK,(5,1))
SYSIN DD *

* PREPROCESSOR SOURCE AND INPUT DATA

JCL to execute preprocessor from source deck.
JCL to execute preprocessor from object deck.
EXEC PGM=PREFPROC,
PARM="L=16K,H=1000K,R=20K,C=100000,P=2000,T=120,D=10,N=58"
//
/* JOB STEP TO EXECUTE PREPROCESSOR FROM LOAD MODULE
//**
//STEPLIB DD DSN=DSP42.PREPROC.LOADMOD,DISP=SHR
//SYSPRINT DD SYSPUT=A
//SYSJCL DD SYSOUT=B,DCB=(BLKSIZE=800,LRECL=80,RECFM=FB,BUFNO=1)
//TEMPCH DD DSN=ETEMPCH,DCB=(BLKSIZE=800,LRECL=80,RECFM=FB),
//SPACE=(TRK,(5,1)),UNIT=3330,DISP=(NEW,DELETE)
//TEMPCH2 DD DSN=ETEMPCH2,DCB=(BLKSIZE=800,LRECL=80,RECFM=FB),
//SPACE=(TRK,(5,1)),UNIT=3330,DISP=(NEW,DELETE)
//TEMOUT DD DSN=ETEMOUT,DCB=(LRECL=132,BLKSIZE=132,RECFM=F),
//UNIT=3330,SPACE=(TRK,(5,1)),DISP=(NEW,DELETE)
//TEMOUT2 DD DSN=ETEMOUT2,DCB=(LRECL=132,BLKSIZE=132,RECFM=F),
//UNIT=3330,SPACE=(TRK,(5,1)),DISP=(NEW,DELETE)
//FINOUT DD SYSOUT=A,DCB=(BLKSIZE=133,LRECL=133,RECFM=FA)
//FINPUNCH DD DSN=ETINPCH,DCB=(BLKSIZE=800,LRECL=80,RECFM=FB),
//DISP=(NEW,PASS),UNIT=3330,SPACE=(TRK,(5,1))
//SYSIN DD *
	**** PREPROCESSOR INPUT DATA

JCL to execute preprocessor from load module.
APPENDIX IV. EXAMPLE BASE LANGUAGE SPECIFICATIONS
Sample language specifications for PL/I.
Sample language specifications for SNOBOL.
FORTRAN

<IF THEN GO TO STATEMENT> ::= LAB IF ( VAL1 RL VAL2 ) GO TO DES

<DECLARATION STATEMENT> ::= INTEGER VARNAME

<NUL L STATEMENT> ::= LAB CONTINUE

<GO TO STATEMENT> ::= LAB GO TO DES

<ASSIGN MINUS STATEMENT> ::= LAB VAR = VAL1 - VAL2

<ASSIGN PLUS STATEMENT> ::= LAB VAR = VAL1 + VAL2

<GREATER THAN> ::= '.' GT'

<NOT> ::= '.' NOT'

<NOT EQUAL> ::= '.' NE'

<GREATER THAN OR EQUAL> ::= '.' GE'

<LESS THAN OR EQUAL> ::= '.' LE'

<LITERAL> ::= '

<COMMENT> ::= 'C' + 1 72

<UNIQUE LABEL> ::= 10 2 4

<UNIQUE IDENTIFIER> ::= 1JK 2

<CONTINUATION> ::= 6

<END OF STATEMENT> ::= 72

<IGNORE COLUMNS> ::= 73-80

<LANG LABEL> ::= 1-5 1-5 '0123456789' '0123456789'

Sample language specifications for FORTRAN.
APPENDIX V. PREPROCESSOR SOURCE CODE LISTING
\* ESTLIMIT = 150000
\*
\* ASSOCIATE OUTPUT VARIABLES WITH A PRINTER FILE, A TEMPORARY
\* 'PRINT' FILE, AND A TEMPORARY 'PUNCH' FILE
\*
\* OUTPUT('FINDOUT','FINDOUT')
\* OUTPUT('OUTPUT','TEMOUT')
\* OUTPUT('PUNCH','TEMPCH')
\*
\* INITIALIZE KEYWORDS
\*
\&ANCHOR = 0
\&DUMP = 1
\&TRIM = 0
\*
\* DEFINE FUNCTIONS
\*
DEFINE('SCANCMDEF(COMDEF)I')
DEFINE('SCANIGNCOLDEF(IGNCOLDEF)')
DEFINE('SCANLANLABDEF(LANLABDEF)SPEC')
DEFINE('SCANLITDEF(LITDEF)')
DEFINE('SCANOESDEF(EOSDEF)')
DEFINE('SCANCONDEF(CONDEF)')
DEFINE('SCANHCHAR(CHAR,COL)')
DEFINE('SCAN()','BEGSCAN')
DEFINE('INSERTCM(MICMSTR)I')
DEFINE('PARSE()','STARTP')
DEFINE('SCANDefs(IA,DS,SEL,T,SMTTTypeDef,UNITYPE',I,REST')
DEFINE('SCANOPERDEF(TYPERDEF,OPERDEF)I')
DEFINE('SCANULLDEF(NULLDEF)I,J')
DEFINE('SCANGOTODEF(GOTODEF)I,J')
DEFINE('SCANIFDEF(IFDEF)I,J')
DEFINE('SCANCLDEF(CLUDF)I,J')
DEFINE('SCANMINUSDEF(MINUSDEF)I,J')
DEFINE('SCANPLUSDEF(PLUSDEF)I,J')
DEFINE('SCANLABELDEF(LABELDEF)I')
DEFINE('SCANVARDEF(VARDEF)I')
DEFINE('MACRSKAN(MACCALL),MACREPSTRT,NAME,MACDEF,PLIST')
DEFINE('CRMINUS(LAB,VAR,VAL,VAL2)STMTLEN,I')
DEFINE('CRPLUS(LAB,VAR,VAL,VAL2)STMTLEN,I')
DEFINE('CRNULL(LAB)STMTLEN,I')
DEFINE('CRGOTO(LAB,DES)STMTLEN,I')
DEFINE('CIFIF(LAB,VAL1,RL,VAL2,DESF)STMTLEN,I')
DEFINE('CRDCL(VAR,NAME)STMTLEN,I')
DEFINE('UNILABEL()')
DEFINE('UNIVAR()')
DEFINE('POP_SEM(NOARGS,A1,A2,A3,A4,A5,A6)I')
DEFINE('PUSH_SEM(NOARGS,A1,A2,A3,A4,A5,A6)I')
DEFINE('INSERT_STMT(STR,INSRT,TYP,OUTSTR,CHARKTR,I)')
DEFINE('INSERT_INSRT(STR)')
DEFINE('ROUTE0()')
DEFINE('ROUTE1()SEL,MAC,STM mẫu,I,PARMLIST,PARM,MACSTR')
DEFINE('ROUTE2(VAR1,VAR2)')
DEFINE('ROUTE3(INC,IND,INC,VAR1,VAR2)')
DEFINE('ROUTE4()')
DEFINE('ROUTE5 BOOL,VAR1')
DEFINE('ROUTE6 BOOL,VAR1')
DEFINE('ROUTE7 BOOL,VAR1,VAR2')
DEFINE('ROUTE8 VAR1')
DEFINE('ROUTE9 VAR0,VAR1,IND')
DEFINE('ROUTE10 VAR1')
DEFINE('ROUTE11 VAR0,VAR1,VAR2,QUITLAB,QUITFLAG')
DEFINE('ROUTE12 BOOL,VAR1,QUITLAB,QUITFLAG')
DEFINE('ROUTE13(VAR1,VAR2,IND,INC,QUITLAB,QUITFLAG,VAR3,VAR4)')
DEFINE('ROUTE14()')
DEFINE('ROUTE15(VAR0,VAR1,VAR3,IND')

* DECLAR ARRAYS

CONTPT = ARRAY('1:2')
EOSPT = ARRAY('0:1')
LITPT = ARRAY('1:3')
LANGLABPT = ARRAY('1:9')
IGNPT = ARRAY('0:2')
COMPT = ARRAY('1:6')
NULLPT = ARRAY('0:3')
GOTOPT = ARRAY('0:5')
IFPT = ARRAY('0:11')
MINUSPT = ARRAY('0:9')
PLUSPT = ARRAY('0:9')
DCLPT = ARRAY('0:3')
LABLPT = ARRAY('1:3')
VARPT = ARRAY('1:3')
MACPTAB = TABLE()
SFMSSTACK = ARRAY('1:10')
VARLIST = ARRAY('1:20')
MPAT = ARRAY('0:20')
MPARM = ARRAY('0:6')
CPARM = ARRAY('1:6')

* INITIALIZE STRINGS AND PATTERNS

MACNAMEPT = DUPL('81')
LABELS = '
KEYCASE = 'CASE'
KEYDO = 'DO'
KEYDGROUP = 'DOGROUP'
KEYIF = 'IF'
KEYTHEN = 'THEN'
KEYUNTIL = 'UNTIL'
KEYWORDS = 'CASE'  |  'KEYDO'  |  'KEYDGROUP'  |  'IF'  |  'REPEAT'  |  'WHILE'  |  'ELSE'  |  'QUITLOOP'  |  'ENDCASE'  |  'ENDIF'
KEYENDWHILE = 'ENDWHILE'
KEYENDREPEAT = 'ENDREPEAT'
KEY ENDDO = 'ENDDO'
KEYENDDOGR = 'ENDDOGR'
NULL =
CURSTMT =
WORKING =
TCKEN =
NEXTCARD =
LANGPAT = 'FORTAN' | 'PL/I' | 'SNOBOL'

**INITIALIZE SCALARS AND STRINGS TO BE USED AS SCALARS**

CALLNUM = 80
VAPNUM = '1'
NEXTVAR = 1
LABELNUM = '1'
TOP = 0
ENDFILE = 0
STARTCOL = 0
STIPSEP = 0
GETEND = 0
COMMENTONLY = 0
STATE = 1
EGSNEXT = 0
MACROSPRS = 0
INSTRUCT = 0
NU. OF CASES = 0
KTRO = 0
DETERMINE LANGUAGE BEING PROCESSED --
FORTAN, PL/I, OR SNOBOL

READ THE LANGUAGE TYPE; BRANCH IF NOT VALID
LANGUAGE = TRIM(INPUT)
LANGUAGE LANGPAT :F(LANGERR)

DECODE DEFINITIONS

READ A CARD, DETERMINE IF IT IS AN OVERRIDE CARD, THE FIRST
DEFINITION, A MACRO CALLING FORM, OR THE FIRST CARD OF THE
INPUT PROGRAM

INLOOP CARD = INPUT

DECODE AN OVERRIDE CARD; BRANCH TO READ THE NEXT CARD
CARD POS(0) 'OVERRIDE:=' :F(NOTOVERRIDE)

NEXTOVERKEY CARD BREAK(' ') ; OVERKEY '=' :F(LASTOVERRKEY)

I('KEY' OVERKEY) = DUPL('X',81) :F(NEXTOVERKEY)

LASTOVERKEY CARD BREAK(' ') ; OVERKEY =
CAPID REM . OVERKEY =

KEYSDONE $('KEY' OVERKEY) = DUPL('X',81)

FINDOUT = OVERKEY ' KEYWORD OVERRIDDEN.'
FINDOUT = 1
FINDOUT = '

NOTOVERRIDE CARD POS(0) '<MACRO CALL>:' :F(NOTMACRO)

SET THE MACROS PRESENT FLAG TO 1; CALL THE MACRO SCANNING ROUTINE;
BRANCH TO READ THE NEXT CARD
MACROSPRES = 1
MACROSCAN(TRIM(CARD)) :F(INLOOP)

NOTMACRO CARD POS(0) '<' :F(NOTDEFNS)

CALL THE DEFINITION SCANNING ROUTINE; SET THE SEMANTIC
DELIMITER STRING; BRANCH TO READ ANOTHER CARD
SCANDefs(CARD)
SEMDEL = '  
EOSPAT<0> 'COL' :S(INLDP) 
SEMDEL = '  
SUBSTR(EOSPAT<1>,1,1) :S(INLDP) 
NOTDEFS 
NEXTCARD = CARD 
*
**PARSE THE INPUT PROGRAM ON PARSE NUMBER 1**
*
**------------------------------------------------------------------------------------------------**
*
P,NUMBER = 1
PARSE()
WORKING = POS(0) 'PROG' RPOS(0) :E(ENDPARSER)
FINOUT = 'FIRST PARSE OF PROGRAM WAS SUCCESSFUL.'
FINOUT = '  
FINOUT = '  
*
**REWIND THE TEMPORARY PUNCH FILE; ASSOCIATE AN INPUT VARIABLE**
**WITH THAT FILE AND OUTPUT VARIABLES WITH A SECOND TEMPORARY**
**PRINT FILE AND A SECOND TEMPORARY PUNCH FILE**
*
REWIND('TEMPC1')
INPUT('INPUT','TEMPC1')
OUTPUT('OUTPUT','TEMOUT2')
OUTPUT('PUNCH','TEMPC1')
*
**BRANCH IF NO SECOND PARSE IS REQUIRED (NO MACROS WERE PROVIDED)**
*
 EQ(MACROSREP,0) :S(NODCECPARSE)
*
**REINITIALIZE STRINGS AND SCALARS FOR SECOND PARSE**
*
CURSMT =
WORKING =
TOKEN =
CLCNUM = 80
TOP = 0
CURNEXT = 0
ENDFILE = 0
GETEND = 0
COMMENTONLY = 0
STATFOS = 1
STARTCOL = 0
STRINGSEP = 0
INSTRUCT = 0
NOFLAGCASES = 0
KTR = 0
PARSE THE RESULTS OF THE FIRST PARSE ON PARSE NUMBER 2

P_NUMBER = 2
NEXTCARD = INPUT
PARSE()
WORKING_POS = 1
PROG_RPS = 1
F_SENSOR = 'SECOND PARSE OF PROGRAM WAS SUCCESSFUL.'
FOUT = ';
FOUT = ';
FOUT = ';

NOSFCPARSE FOUT = 'SECOND PARSE OF PROGRAM NOT REQUIRED.'
FOUT = ';
FOUT = ';
FOUT = ';
REWIND THE TEMPORARY PRINT FILE; ASSOCIATE AN INPUT VARIABLE WITH THAT FILE
REWIND('TEMOUT')
INPUT('INTEMP','TEMOUT')

COPY THE PREVIOUS TEMPORARY 'PRINT' FILE TO A SECOND TEMPORARY 'PRINT' FILE
COPYTEMP1 CARD = INTEMP
OUTPUT = CARD :F(COPYTEMP2)
COPYTEMP2 CARD = INTEMP
PUNCH = CARD :F(ADDCLS)
REWIND TEMPORARY FILES AND ASSOCIATE INPUT AND OUTPUT VARIABLES WITH TEMPORARY FILES
REWIND('TEMOUT2')
REWIND('TEMOUT')
REWIND('TEMOUT')
INPUT('INCARD', 'TEMOUT2')
INPUT('INLINE', 'TEMOUT2')
OUTPUT('PUNCH', 'TEMOUT')
OUTPUT('OUTPUT', 'TEMOUT')
ADD DECLARATIONS FOR CREATED VARIABLES IF A NON-BLANK DECLARATION
STATEMENT PROTOTYPE WAS PROVIDED.

READ A CARD AND ITS CORRESPONDING OUTPUT LINE

COPY3  
  CARD = INCARD  
  LINE = INLINE  

IF THE CARD IS NOT THE ADD DECLARATIONS INDICATOR, COPY THIS
CARD AND OUTPUT LINE AND BRANCH TO READ MORE;
IF NO ADD DECLARATIONS INDICATOR IF FOUND, TAKE THE
ERROR EXIT

  CARD 'ADDED DECLARATIONS'  
  PUNCH = CARD  
  OUTPUT = LINE

DECREMENT NEXTVAR, BRANCH TO COPY THE REMAINDER OF THE
FILES IF THERE ARE NO VARIABLES FOR WHICH TO ADD
DECLARATIONS

ADDDECS  
  NEXTVAR = NEXTVAR - 1  
  EQ(NEXTVAR,0)

CREATE A STRING OF A DECLARATION STATEMENT; INSERT A COMMENT
ABOUT ADDING A DECLARATION; INSERT THE DECLARATION;
IF THERE ARE MORE NEW VARIABLES, DECREMENT NEXTVAR AND
BRANCH TO ADD THE NEXT DECLARATION

ADDNEXTDEC  
  STRING = CRDCL VARLIST(NEXTVAR)  
  INSERTCOM( 'THE FOLLOWING IS AN ADDED DECLARATION' )  
  INSERT STRING  
  NEXTVAR = GT(NEXTVAR,1) NEXTVAR + 1

COPY THE REMAINDER OF THE FILES .

COPYRESTFILES PUNCH = INCARD  
  OUTPUT = INLINE

REWIND THE TEMPORARY FILES AND ASSOCIATE INPUT VARIABLES
WITH TEMPORARY FILES AND OUTPUT VARIABLES WITH
REAL OUTPUT FILES
CKSNOBOL
  REWIND('TEMPC');
  REWIND('TEMOUT');
  INPUT('INCARD', 'TEMPC');
  INPUT('INLINE', 'TEMOUT');
  OUTPUT('PUNCH', 'FINPUNCH');
  OUTPUT('OUTPUT', 'FINOUT');

* BRANCH IF THE LANGUAGE BEING PROCESSED IS NOT SNOBOL
* LANGUAGE 'SNOBOL'
:IF(NOT SNOBOL)

* POST PROCESSING FOR SNOBOL:
* SCAN THE FILES FOR CONDITIONALS USING \{\ FOLLOwed BY
  BLANKS AND \} -- THESE WERE ADDED BY THE PREPROCESSOR;
  DELETE THE INTERVENTING BLANKS AND THE COMMA TO MAKE THE
  STATEMENT A LEGAL SNOBOL STATEMENT

* SNOBOLTEST CARD = INCARD
  LINE = INLINE
  CARD \ = \ (\ SPAN(\ : \ )); \ = \ \{\ \}
  LINE \ = \ (\ SPAN(\ : \ )); \ = \ \{\ \}
  PUNCH = CARD
  OUTPUT = LINE
\{\ (SNOBOLTEST)

* BRANCH IF THE LANGUAGE BEING PROCESSED IS NOT FORTRAN
* NOTSNOBOL LANGUAGE 'FORTRAN'
:IF(NOT FORTRAN)

* POST PROCESSING FOR FORTRAN:
* SCAN THE FILES FOR SEQUENCES OF A COMMENT, A CONTINUE, AND A
  FORMAT STATEMENT; RETURN THE STATEMENT NUMBER FROM THE
  CONTINUE (INDICATION IS IN THE COMMENT THAT THE STATEMENT
  NUMBER WAS TAKEN FROM THE FORMAT AND PLACED ON THE CONTINUE)
  TO THE FORMAT STATEMENT AND DELETE THE CONTINUE STATEMENT

* FORTEST CARD1 = INCARD
  LINE1 = INLINE
\{\ (COMPLETE)
FORFTST2 CARD1 POS(0) 'C' BREAK(''') . STMT.NUMB :F(NDFORTCOM)
CARD2 = INCARD
LINE2 = INLNE
CARD3 = INCARD
LINE3 = INLNE
CARD2 POS(6) 'CONTINUE' :F(NDFORTCONT)
CARD4 = INCARD
LINE3 = INLNE
TESTCONTINUE CARD3 POS(0) ' CONTINUE' :F(THISFORM)
CARD3 = INCARD
LINE3 = INLNE
THISFORM CARD3 POS(0) ' ARB - REP ' FORMAT' =
+ DUPL(' ',5 - SIZE(STMT.NUMB)) STMT.NUMB :F(NUFORTFORM)
+ LINE3 POS(0) ' AI = DUPL(' ',5 - SIZE(STMT.NUMB))
+ STMT.NUMB :F(NDFORTFORM)
PUNCH = CARD3
OUTPUT = LINE3
NODFORTCOM PUNCH = CARD1
OUTPUT = LINF1
NDFORTCOM PUNCH = CARD1
OUTPUT = LINF2
LINF1 = LINF2
NDFORTCOM PUNCH = CARD1
OUTPUT = LINF1
LINF1 = LINF1
NDFORTCOM PUNCH = CARD1
OUTPUT = LINF2
LINF1 = LINF3
END OF DUMP

BRANCH IF THE LANGUAGE BEING PROCESSED IS NOT PL/I

NODFORTTRAN LANGUAGE 'PL/I' :F(NOTPLI)

POST PROCESSING FOR PL/I:

SCAN THE BEGINNING OF THE FILES FOR EITHER ONE OR TWO
COMMENTS FOLLOWED BY THE SAME NUMBER OF NULL STATEMENTS AND
A PROCEDURE STATEMENT; RETURN THE LABEL WHICH WAS ORIGINALY
TAKEN OR THE PROCEDURE STATEMENT TO THE PROCEDURE STATEMENT
AND DELETE THE NULL STATEMENTS; COPY THE REMAINDER
OF THE FILES

CARD1 = INCARD
LINE1 = INLNE

PRE03480
PRE03490
PRE03500
PRE03510
PRE03520
PRE03530
PRE03540
PRE03550
PRE03560
PRE03570
PRE03580
PRE03590
PRE03600
PRE03610
PRE03620
PRE03630
PRE03640
PRE03650
PRE03660
PRE03670
PRE03680
PRE03690
PRE03700
PRE03710
PRE03720
PRE03730
PRE03740
PRE03750
PRE03760
PRE03770
PRE03780
PRE03790
PRE03800
PRE03810
PRE03820
PRE03830
PRE03840
PRE03850
PRE03860
PRE03870
PRE03880
PRE03890
CARD1  POS(0) SPAN(' ')  /* BREAK(' ') */  STMT.NUMB
CARD2 = INCARD
LINE2 = INLINE
CARD2  POS(0) SPAN(' ')  /* F(THISPROC) */
CARD2 = INCARD
LINE2 = INLINE
CARD2 = INCARD
LINE2 = INLINE
CARD2 = INCARD
LINE2 = INLINE
CARD2 = INCARD
LINE2 = INLINE
CARD2  POS(0) SPAN(' ')  /* REP 'PROC' = */
   DUPL(' ', SIZE(REP) - SIZE(STMT.NUMB))
   STMT.NUMB 'PROC'
   LINE2  POS(0) SPAN(' ')  REP 'PROC' = DUPL(' ', SIZE(REP) -
   SIZE(STMT.NUMB)) STMT.NUMB 'PROC'
   PUNCH = CARD1
   PUNCH = CARD2
   OUTPUT = LINE1
   OUTPUT = LINF2
   COPYRESTPLI PUNCH = INCARD
   (OUTPUT = INLINE)  /* F(COMPLETE) */
   (COPYRESTPLI)  /* F(COMPLETE) */
   /* THF LANGUAGE WAS NOT FORTRAN OR SNOBOL OR PL/I, SO SIMPLY
   COPY THE FILES */
   /* NOTPLI PUNCH = INCARD */
   OUTPUT = INLINE  /* F(COMPLETE) */
   /* NOTFORTRAN */
   OUTPUT A SUCCESSFUL COMPLETION MESSAGE AND TERMINATE
   COMPLETE  FINSUT = 'PREPROCESSING COMPLETED SUCCESSFULLY.'
   /* END */
   SECTION TO PRODUCE OUTPUT AFTER ERROR MESSAGES HAVE BEEN
   PRINTED FOR ERRORS DURING PROCESSING OF THE INPUT PROGRAM
   COPY OUTPUT PRODUCED UP TO THE ERROR WHICH HAS BEEN ENCOUNTERED;
   THE FILE OUTPUT DEPENDS ON THE PARSE NUMBER DURING WHICH
   THE ERROR OCCURRED
COPYTEMPOUT \texttt{EQ} (P\textunderscore NUMBER, 2) \
*: (SECONDPARSE) \texttt{PRE}\texttt{04300}
\texttt{PRE}\texttt{04310}
\texttt{PRE}\texttt{04320}
\texttt{PRE}\texttt{04330}
\texttt{PRE}\texttt{04340}
\texttt{PRE}\texttt{04350}
\texttt{PRE}\texttt{04360}
\texttt{PRE}\texttt{04370}
\texttt{PRE}\texttt{04380}
\texttt{PRE}\texttt{04390}
\texttt{PRE}\texttt{04400}
\texttt{PRE}\texttt{04410}
\texttt{PRE}\texttt{04420}
\texttt{PRE}\texttt{04430}
\texttt{PRE}\texttt{04440}
\texttt{PRE}\texttt{04450}
\texttt{PRE}\texttt{04460}
\texttt{PRE}\texttt{04470}
\texttt{PRE}\texttt{04480}
\texttt{PRE}\texttt{04490}
\texttt{PRE}\texttt{04500}
\texttt{PRE}\texttt{04510}
\texttt{PRE}\texttt{04520}
\texttt{PRE}\texttt{04530}
\texttt{PRE}\texttt{04540}
\texttt{PRE}\texttt{04550}
\texttt{PRE}\texttt{04560}
\texttt{PRE}\texttt{04570}
\texttt{PRE}\texttt{04580}
\texttt{PRE}\texttt{04590}
\texttt{PRE}\texttt{04600}
\texttt{PRE}\texttt{04610}
\texttt{PRE}\texttt{04620}
\texttt{PRE}\texttt{04630}
\texttt{PRE}\texttt{04640}
\texttt{PRE}\texttt{04650}
\texttt{PRE}\texttt{04660}
\texttt{PRE}\texttt{04670}
\texttt{PRE}\texttt{04680}
\texttt{PRE}\texttt{04690}
\texttt{PRE}\texttt{04700}
\texttt{PRE}\texttt{04710}
\texttt{PRE}\texttt{04720}
\texttt{PRE}\texttt{04730}

* REWIND THE TEMPORARY PUNCH FILE; ASSOCIATE AN INPUT VARIABLE
WITH THAT FILE

REWIND( 'TEMPCH1' )
INPUT( 'INTERM', 'TEMPCH1' )
*: (OUTPUTASSOC)

REWIND( 'TEMPCH2' )
INPUT( 'INTERM', 'TEMPCH2' )
*: (SECONDPARSE)

ASSOCIATE AN OUTPUT VARIABLE WITH THE REAL PRINTER FILE

OUTPUTASSOC OUTPUT( 'OUTPUT', 'FINDOUT', ' ' )
OUTPUT = ' '
OUTPUT = ' BECAUSE AN ERROR HAS BEEN ENCOUNTERED.'
" THE CONTENTS OF THE TEMPORARY FILE HOLDING THE "
" PUNCHED OUTPUT PRODUCED SO FAR FOLLOWS:"
OUTPUT = ' '
OUTPUT = ' '

PRINT THE OUTPUT PRODUCED SO FAR AND BRANCH TO END

TEMCOPY CARD = 'INTERM'
OUTPUT = ' ' : (END) TEMCOPY

ENDTEMCOPY OUTPUT = ' ' : (END)
OUTPUT = ' PREPROCESSING COMPLETED UNSUCCESSFULLY.' : (END)

EXITS FCP ERRORS WHICH OCCURRED IN THE DRIVER

LANGERR FINOUT = ' AN INVALID LANGUAGE WAS SPECIFIED'
FINOUT = ' LANGUAGE = ' LANGUAGE
FINOUT = ' ABNORMAL TERMINATION' : (END)

ENDPARSERR FINOUT = ' THE FIRST PARSE OF THE INPUT PROGRAM TERMINATED ' ABNORMAL
FINOUT = ' WORKING = ' WORKING
FINOUT = ' THE SECOND PARSE OF THE INPUT PROGRAM TERMINATED ' ABNORMAL
FINOUT = ' WORKING = ' WORKING

EXITs FOR ERRORS WHICH OCCURRED IN THE PARSER
FINOUT = 'SCAN RETURNED TO THE PARSER WITH FAILURE' : (END) PRE05530
+ WHEN A RETURNED VALUE WAS REQUIRED' PRE05540
FINOUT = 'CURSTM = CURSTM' PRE05550
FINOUT = 'WORKING = WORKING' :(COPYTEMPOUT) PRE05560
FINOUT = 'PASER IS UNABLE TO RECOGNIZE THE SEQUENCE' PRE05570
+ OF TOKENS IN THE WORKING STRING' PRE05580
FINOUT = 'CURSTM = CURSTM' PRE05590
FINOUT = 'WORKING = WORKING' :(COPYTEMPOUT) PRE05600
FINOUT = 'FAILURE TO PASE STRUC TO STRLIST REDUCTION' PRE05610
EXIT FOR ERRORS WHICH OCCURRED IN THE SCANNER
** Exits for Errors Which Occurred in the Semantic Routines **

- ELSEERROR FOUT = 'ERROR IN SEMANTIC PROCESSING OF ELSE'
- CASEERROR FOUT = 'ERROR IN SEMANTIC PROCESSING OF CASE'
- DUERROR FOUT = 'ERROR IN SEMANTIC PROCESSING OF DO'
- DOGROUPERROR FOUT = 'ERROR IN SEMANTIC PROCESSING OF DOGROUP'
- IFERROR FOUT = 'ERROR IN SEMANTIC PROCESSING OF IF'
- REPERROR FOUT = 'ERROR IN SEMANTIC PROCESSING OF REPEAT'
- WHERROR FOUT = 'ERROR IN SEMANTIC PROCESSING OF WHILE'
- QUITEROR FOUT = 'ERROR IN SEMANTIC PROCESSING OF QUITGROUP'
- ECASEERROR FOUT = 'ERROR IN SEMANTIC PROCESSING OF ENDCASE'
- EIPERROR FOUT = 'ERROR IN SEMANTIC PROCESSING OF ENDEIF'
- EWHILEERROR FOUT = 'ERROR IN SEMANTIC PROCESSING OF ENDOF'
- EREPERERROR FOUT = 'ERROR IN SEMANTIC PROCESSING OF ENDRPEPEAT'
ENDERROR
FINOUT = 'ERROR IN SEMANTIC PROCESSING OF ENDDO'
FINOUT = 'INSTR = INSTR (COPYTEMPOUT)'  
PRE06500

ERRGROUPERROR FINOUT = 'ERROR IN SEMANTIC PROCESSING OF ENDDO'
FINOUT = 'INSTR = INSTR (COPYTEMPOUT)'  
PRE06500

MACCALLERROR FINOUT = 'MORE THAN SIX ARGUMENTS WERE SUPPLIED IN A CALL'
+ 'TO THE MACRO SELMAC (COPYTEMPOUT)'  
PRE06540

ROUTEERROR FINOUT = 'THE STATEMENT VARIABLE EQUALS 0 IN THE MACRO'
+ 'DEFINITION FOR SELMAC (COPYTEMPOUT)'  
PRE06560

MACARGERROR FINOUT = 'A PARAMETER FOUND IN THE MACRO DEFINITION ARRAY'
+ 'FOR SELMAC IS NOT IN THE PARAMETER ARRAY FOR'
+ 'THAT MACRO (COPYTEMPOUT)'  
PRE06590

MACBLKERROR FINOUT = 'NOT ENOUGH BLANKS LEFT AFTER A PARAMETER'
+ 'NAME IN THE DEFINITION FOR THE MACRO SELMAC'
+ 'NAME (COPYTEMPOUT)'  
PRE06600

TOFFULLSTK FINOUT = 'ATTEMPT TO PUSH ONTO A FULL SEMANTIC STACK'
+ 'COPYTEMPOUT'  
PRE06640

EMPTYSTK FINOUT = 'ATTEMPT TO POP FROM AN EMPTY SEMANTIC STACK'
+ 'COPYTEMPOUT'  
PRE06660

/* EXITS FOR ERRORS WHICH OCCURRED IN THE CREATION ROUTINES */

CRMINERROR FINOUT = 'AN ERROR WAS ENCOUNTERED DURING AN ATTEMPT TO'
+ 'CREATE AN ASSIGN MINUS STATEMENT (COPYTEMPOUT)'  
PRE06700

CRDCLERROR FINOUT = 'AN ERROR WAS ENCOUNTERED DURING AN ATTEMPT TO'
+ 'CREATE A DECLARATION STATEMENT (COPYTEMPOUT)'  
PRE06740

CRIFERROR FINOUT = 'AN ERROR WAS ENCOUNTERED DURING AN ATTEMPT TO'
+ 'CREATE AN IF THEN GO TO STATEMENT (COPYTEMPOUT)'  
PRE06750

CRNULEXERROR FINOUT = 'AN ERROR WAS ENCOUNTERED DURING AN ATTEMPT TO'
+ 'CREATE A NULL STATEMENT (COPYTEMPOUT)'  
PRE06790

CRPLERROR FINOUT = 'AN ERROR WAS ENCOUNTERED DURING AN ATTEMPT TO'
+ 'CREATE A NULL STATEMENT (COPYTEMPOUT)'  
PRE06800

CRGOTOERROR FINOUT = 'AN ERROR WAS ENCOUNTERED DURING AN ATTEMPT TO'
+ 'CREATE A GO TO STATEMENT (COPYTEMPOUT)'  
PRE06820

DCLBLKERROR FINOUT = 'NOT ENOUGH BLANKS LEFT AFTER A PARAMETER NAME'
+ 'IN THE DECLARATION PROTOTYPE (COPYTEMPOUT)'  
PRE06830

GOTUBLKERROR FINOUT = 'VARNAME = VARNAME (COPYTEMPOUT)'  
PRE06850

PLUSBLKERROR FINOUT = 'NOT ENOUGH BLANKS LEFT AFTER A PARAMETER NAME'
+ 'IN THE ASSIGN PLUS PROTOTYPE (COPYTEMPOUT)'  
PRE06890

MINUBLKERROR FINOUT = 'NOT ENOUGH BLANKS LEFT AFTER A PARAMETER NAME'
PRE06930
'IN THE ASSIGN MINUS PROTOTYPE'
FINOUT = 'CURSTM = ' CURSTM : (COPYTEMPOUT)
PRE06940
NULLBKERR FINOUT = 'NOT ENOUGH BLANKS LEFT AFTER A PARAMETER NAME'
'IN THE NULL STATEMENT PROTOTYPE'
PRE06950
IFBLKERR FINOUT = 'NOT ENOUGH BLANKS LEFT AFTER A PARAMETER NAME'
'IN THE IF THEN GO TO STATEMENT PROTOTYPE'
PRE06960
FINOUT = 'CURSTM = ' CURSTM : (COPYTEMPOUT)
PRE06970
TOOMANYLABELS FINOUT = '(ATTEMPT TO CREATE MORE UNIQUE LABELS THAN THE DEFINITION ALLOWS)
: (COPYTEMPOUT)
PRE06980
TOOMANYVARS FINOUT = 'ATTEMPT TO CREATE MORE UNIQUE IDENTIFIERS THAN THE DEFINITION ALLOWS'
: (COPYTEMPOUT)
PRE06990
BEGSCAN OLDESTMTEOS = STMTEOS
PRE07000
***************************************************************************
ROUTINE TO SCAN INPUT FOR TOKENS RECOGNIZABLE BY THE PARSER
***************************************************************************
NO PARAMETERS
***************************************************************************
A CARD IMAGE MAY BE PRESENT IN NEXTCARD OR NOT WHEN THIS ROUTINE IS INITIALLY CALLED
***************************************************************************
IF AN EOS SHOULD BE SIGNALLED IMMEDIATELY, BRANCH TO THAT LABEL
***************************************************************************
SAVE FLAG INDICATING WHETHER OR NOT AN EOS WAS FOUND IN THE LAST STATEMENT; SET EOS FLAG FOR THIS STATEMENT TO 0
***************************************************************************
STMTEOS = 0
EQ (ECSNEXT, 1) : S (FOSIMMED)
***************************************************************************
GET THE NEXT USEABLE CHARACTER AND ITS COLUMN NUMBER; IF GETCHAR RETURNS WITH FAILURE, BRANCH
***************************************************************************
BEGSCAN1 GETCHAR ('INCHAR', 'CHARCOL') : FIENDOFINPUT
***************************************************************************
IF THIS IS THE FIRST CHARACTER OF A TOKEN OR THE FIRST NON-BLANK CHARACTER OF A TOKEN, SET THE STARTCOL VARIABLE
***************************************************************************
STARTCOL = EQ (SIZE (TOKEN), 0) CHARCOL
***************************************************************************
IF THIS IS THE FIRST USEABLE COLUMN ON A CARD, CHECK TO SIE IF THE TOKEN IS CURRENTLY ALL BLANKS LEFT OVER FROM A PREVIOUS CARD WHICH SHOULD BE DISCARDED
***************************************************************************
OLDSTART NE (IGNPAT<1>, 1) : S (CKATCOLONF)
PRE07071
PRE07072
PRE07080
PRE07081
PRE07090
PRE07091
PRE07100
PRE07101
PRE07110
PRE07111
PRE07120
PRE07121
PRE07130
PRE07131
PRE07140
PRE07141
PRE07150
PRE07151
PRE07160
PRE07161
PRE07170
PRE07171
PRE07180
PRE07181
PRE07190
PRE07191
PRE07200
PRE07201
PRE07210
PRE07211
PRE07220
PRE07221
PRE07230
PRE07231
PRE07240
PRE07241
PRE07250
PRE07251
PRE07260
PRE07261
PRE07270
PRE07271
PRE07280
PRE07281
PRE07290
PRE07291
PRE07300
PRE07301
PRE07310
PRE07311
PRE07320
PRE07321
PRE07330
PRE07331
PRE07340
PRE07341
PRE07350
PRE07351
PRE07360
PRE07361
PRE07370
PRE07371
PRE07380
PRE07381
IGNPAT<>'ONE'
EQ(CHARCOL2):F(LOOKATSEQEND) PREO7390
(LOOKATSEQEND EQ(CHARCOL1, IGNPAT<2> + 1)) :F(LEAVETOKEN) PREO7410
CKATCOLONE EQ(CHARCOL1) :F(LEAVETOKEN) PREO7420
CKBLANKTOKEN TOKEN POS(0) SPAN(' ') RPOS(0) :S(LEAVETOKEN) PREO7430
** IF THE TOKEN IS OF 0 LENGTH AND A COMMENT WAS NOT ALL THAT
* APPEARED ON THE LAST CARD, LEAVE THE TOKEN AND CURSTM INTACT
* FO(COMMENTONLY,1)
* INSERTCOM(CURSTM)
** THE CURSTM AND TOKEN ARE NULLIFIED IF AN EOS WAS FOUND BEFORE
* THIS CALL OF SCAN
** TESTOLDEOS EQ(DISTMTEOS,1)
* CURSTM =
** TOKEN =
* ADD THE CHARACTER TO THE TOKEN STRING
** LEAVETOKEN TOKEN = TOKEN INCHAR
* LABSTART = 0
* COMMENTONLY = 0
** LOOK FOR BEGINNING OF A COMMENT
** BRANCH TO APPROPRIATE SECTION DEPENDING ON OPENING
* FOR COMMENT SPECIFIED IN DEFINITION
* COMPAT<1> '10' :S(COMSYM) PREO7646
* COLUMN NUMBER PART OR ALL OF COMMENT OPENING
* FO(CHARCOL,COMPAT<4>) :F(NOTCOM) PREO7670
* COMMENT FOUND IF COLUMN ONLY STARTS COMMENT
* COMPAT<1> '11' :F(COMFOUND) S(COMCOLSYM) PREO7700
* SYMBOL ONLY STARTS COMMENT
* COMCOLSYM COMPAT<3> POS(0) INCHAR :F(NOTCOM) PREO7730
* COMMENT FOUND IF ONLY ONE SYMBOL REQUIRED
TOKEN = TOKEN INCHAR : (COMFOUND)

BACKUP BY RESTORING THE OLD CHARACTER AND OLD COLUMN NUMBER;
BRANCH TO SEARCH FOR SOMETHING ELSE

NOTCOM2CHARS INCHAR = OLDCHAR
CHARCOL = OLDCHAR
COLUMN = OLDCHAR : (NOTCOM)

LOOK FOR THE END OF THE COMMENT

COMMENT HAS BEEN FOUND; SET THE STATEMENT FLAG TO WHATEVER IT WAS
BEFORE THIS CALL OF SCAN; SET THE COMMENTONLY FLAG IF NOTHING ELSE HAS BEEN ENCOUNTERED IN THIS STATEMENT; SEARCH FOR THE END OF THE COMMENT

COMFOUND
STATEMENTS = OLDSTATEMENTS
NEW (SIZE(CURSTM),0)
COMMENTONLY = 1

GET THE NEXT USEABLE CHARACTER AND ITS COLUMN NUMBER; ADD THE CHARACTER TO THE TOKEN STRING

SKIPNOTSTMT GETCHAR('INCHAR', 'CHARCOL')
TOKEN = TOKEN INCHAR

BRANCH TO APPROPRIATE SECTION DEPENDING ON CLOSING FOR COMMENT SPECIFIED IN DEFINITION

COMFOUND INCHAR COMPAT<2> '10'
COLUMN NUMBER PART OR ALL OF COMMENT CLOSING
EQCHARCOL,COMPAT<6> COMFOUND
COMMENT ENDS IF ONLY COLUMN ENDS COMMENT COMPAT<2> '11'
COLUMN NUMBER AND SYMBOL END COMMENT
COMENDSYM COMPAT<5> POS() INCHAR COMEND
COMMENT ENDS IF ONLY ONE SYMBOL REQUIRED

PRF08290
PRF08300
PRF08310
PRF08320
PRF08330
PRF08340
PRF08350
PRF08360
PRF08370
PRF08371
PRF08372
PRF08373
PRF08374
PRF08375
PRF08376
PRF08380
PRF08390
PRF08400
PRF08410
PRF08420
PRF08430
PRF08440
PRF08450
PRF08460
PRF08470
PRF08480
PRF08490
PRF08500
PRF08510
PRF08520
PRF08530
PRF08540
PRF08550
PRF08560
PRF08570
PRF08580
PRF08590
PRF08600
PRF08610
PRF08620
PRF08630
PRF08640
PRF08650
PRF08660
PRF08670
PRF08680
PRF08690
PRF08700
EQ(SIZE(COMPAT<5>),1) : S(ENDCOM)
ERROR IF COMMENT CLOSING STRING IS LONGER THAN 2 CHARS.
EQ(SIZE(COMPAT<5>),2) : F(COMENDERR)
SAVE THE CURRENT CHARACTER AND COLUMN
OLDCOL = CHARCOL
CLDCHAR = INCHAR
GET THE NEXT USEABLE CHARACTER AND ITS COLUMN NUMBER;
ADD THE CHARACTER TO THE TOKEN STRING
GETCHAR('INCHAR','CHARCOL') : F(ENDOFINPUT)
TOKEN = TOKEN INCHAR
COMMENT ENDS IF THE 2 CHARACTERS CLOSE A COMMENT;
OTHERWISE KEEP SEARCHING FOR THE END
COMPAT<5> OLDCHAR INCHAR : S(ENDCOM) F(COMFOUNDNOCHAR)
SYMBOL ONLY ENDS COMMENT, BRANCH IF THIS CHARACTER MAY
INDICATE THE END OF THE COMMENT
COMENDSTR COMPAT<5> POS(0) INCHAR : S(FIRSTCHAREND)
GET THE NEXT USEABLE CHARACTER AND ITS COLUMN NUMBER;
ADD THE CHARACTER TO THE TOKEN STRING; KEEP LOOKING
FOR THE CLOSING SYMBOL
GETCHAR('INCHAR','CHARCOL') : F(ENDOFINPUT)
TOKEN = TOKEN INCHAR : (COMENDSTR)
ONE SYMBOL HAS BEEN MATCHED; END OF COMMENT HAS BEEN FOUND
IF ONLY ONE CHARACTER IS NEEDED
FIRSTCHAREND EQ(SIZE(COMPAT<5>),1) : S(ENDCOM)
ERROR IF COMMENT CLOSING STRING LONGER THAN 2 CHARS.
EQ(SIZE(COMPAT<5>),2) : F(COMENDERR)
SAVE THE CURRENT CHARACTER AND COLUMN
OLDCOL = CHARCOL
OLDCHAR = INCHAR
GET THE NEXT USEABLE CHARACTER AND ITS COLUMN;
ADD THE CHARACTER TO THE TOKEN STRING

GETCHAR(*INCHAR*, 'CHARCOL')
TOKEN = TOKEN INCHAR

END OF COMMENT FOUND IF THE 2 CHARACTERS CLOSE A COMMENT;
OTHERWISE KEEP SEARCHING

COMPAT<5> OLDCHAR INCHAR
:F(COMENDSTR)

END OF COMMENT FOUND

ENDCOM
CURSTMT = CURSTMT TOKEN
TOKEN =
:(BEGSCAN1)

NOT A COMMENT; LOOK FOR A LITERAL OPENING

NOTCOM
LITPAT<1> POS(0) INCHAR
:F(NOTLIT)

LITERAL FOUND IF THE SINGLE SYMBOL STARTS A LITERAL
EQ(SIZE(LITPAT<1>), 1)
:S(LITFOUND)

ERROR IF LITERAL OPENING STRING LONGER THAN 2 CHARACTERS
EQ(SIZE(LITPAT<2>), 2)
:F(LITOPENERR)

SAVE THE CURRENT CHARACTER AND ITS COLUMN NUMBER
CLOSECOL = CHARCOL
CLOSECHAR = INCHAR

GET THE NEXT USEABLE CHARACTER AND ITS COLUMN NUMBER
GETCHAR(*INCHAR*, 'CHARCOL')
:F(NOTLIT2CHARS)

BRANCH IF THE 2 CHARACTERS DO NOT START A LITERAL;
OTHERWISE ADD THE CHARACTER TO THE TOKEN STRING

LITPAT<1> OLDCHAR INCHAR
TOKEN = TOKEN INCHAR
:F(NOTLIT2CHARS)
:S(LITFOUND)
BACKUP BY RESTORING OLD CHARACTER AND COLUMN NUMBER;
BRANCH TO SEARCH FOR SOMETHING ELSE

NOTLIT2CHARS INCHAR = OLDCHAR
CHARCOL = OLDCHAR
COLNUM = OLDCHAR

:(NOTLIT)

LOOK FOR THE END OF THE LITERAL

LITERAL FOUND; GET THE NEXT USEABLE CHARACTER AND ITS COLUMN NUMBER; ADD THE CHARACTER TO THE TOKEN STRING

LITFOUND
GETCHAR('INCHAR','CHARCOL') :F(ENDOFINPUT)
TOKEN = TOKEN INCHAR

IF CHARACTER DOES NOT CLOSE A LITERAL, KEEP SEARCHING

LITPAT<2> POS(0) INCHAR :F(LITFOUND)

BRANCH TO LOOK FOR SPECIAL SEQUENCE IF THE CHARACTER MATCHES THE CLOSING SINGLE CHARACTER

EQ(SIZE(LITPAT<2>),1) :S(MATCHLITEND1)

ERROR IF LITERAL CLOSING STRING IS LONGER THAN 2 CHARACTERS

EQ(SIZE(LITPAT<2>),2) :F(LITENDERR)

SAVE THE CURRENT CHARACTER AND ITS COLUMN NUMBER

GLDCLL = CHARCOL
GLDCHAR = INCHAR

GET THE NEXT USEABLE CHARACTER AND ITS COLUMN NUMBER;
ADD THE CHARACTER TO THE TOKEN STRING

GETCHAR('INCHAR','CHARCOL') :F(ENDOFINPUT)
TOKEN = TOKEN INCHAR

LITERAL FOUND IF THE 2 CHARACTERS CLOSE A LITERAL;
OTHERWISE KEEP SEARCHING FOR END

LITPAT<2> OLDCHAR INCHAR :F(LITFOUND)S(ENDLIT)
ONE CHARACTER HAS BEEN MATCHED; LITERAL END FOUND IF NO SPECIAL
SEQUENCE SPECIFIED IN DEFINITION
MATCHLITEND$EQ(SIZE(LITPAT<3>),0) :S(ENDLIT)
SAVE THE CURRENT CHARACTER AND COLUMN NUMBER
OLDCHAR = CHARCOL
OLDCOL = INCHAR
GET THE NEXT USEABLE CHARACTER AND ITS COLUMN NUMBER
GETCHAR('INCHAR', 'CHARCOL') :F(ENDLITL)
END OF LITERAL FOUND IF THIS IS NOT THE SPECIAL SEQUENCE
LITPAT<3> OLDCHAR INCHAR :F(ENDLITL)
ADD THE CHARACTER TO THE TOKEN STRING; SKIP THIS CHARACTER
AND KEEP SEARCHING FOR THE LITERAL END
TOKEN = TOKEN INCHAR :(LITFOUND)
BACKUP BY RESTORING THE OLD CHARACTER AND ITS COLUMN NUMBER
ENDLITL INCHAR = OLDCOL
CHARCOL = OLDCHAR
CCLNUM = OLDCOL
A LITERAL IS AN UNIDENTIFIED OBJECT FOR THE PARSER, SO BRANCH TO
THAT SECTION
ENDLIT :(UNIDENT)
NOT A LITERAL; SEARCH FOR AN END OF STATEMENT
NCLIT FOSPAT<0> 'COL' :S(EOSCOL)
SYMBOL SIGNIFIES END OF STATEMENT
FCSPAT<1> POS(0) INCHAR :F(NOTEOSSCAN)
END OF STATEMENT FOUND IF SINGLE SYMBOL ENDS STATEMENT
* FQ(SIZE(EOSPAT<1>),1) : S(EOSFOUND)
* ERROR IF END OF STATEMENT STRING IS LONGER THAN 2 CHARACTERS
* EQ(SIZE(EOSPAT<1>),2) : (EOSDEFNERR)
* SAVE THE CURRENT CHARACTER AND COLUMN NUMBER
* OLDCHAR = CHARCOL
* OLDCALL = INCHAR
* GET THE NEXT USEABLE CHARACTER AND ITS COLUMN NUMBER
* GETCHAR('INCHAR','CHARCOL') : F(NOTEOS2CHARS)
* BRANCH IF THE 2 CHARACTERS DO NOT MARK END OF STATEMENT;
* OTHERWISE ADD THE CHARACTER TO THE TOKEN STRING
* EOSPAT<1> OLDCHAR INCHAR
* TOKEN = TOKEN INCHAR : (EOSFOUND)
* BACKUP BY RESTORING THE OLD CHARACTER AND COLUMN NUMBER;
* BRANCH TO SEARCH FOR SOMETHING ELSE
* NOTEOS2CHARS INCHAR = OLDCHAR
* CHARCOL = OLDCALL
* COLUM = OLDCALL : (NOTEOSSCAN)
* END OF STATEMENT MARKED BY COLUMN
* END OF STATEMENT FOUND IF COLUMN NUMBER IS THAT SPECIFIED
* IN DEFINITION; OTHERWISE GO SEARCH FOR SOMETHING ELSE
* EOSCOL EQ(COLNUM,EOSPAT<1>) : S(EOSCOLFOUND)F(NOTEOSSCAN)
* END OF STATEMENT FOUND BY COLUMN NUMBER;
* READ IN ANOTHER CARD AND LOOK FOR THE CONTINUATION
* SPECIFICATION
* EOSCOLFOUND EQ(GETEND,1) : S(EOSFOUND)
* NEXTCARD = INPUT
* ENDFILE = 1 : (EOSFOUND)
* BRANCH TO SEARCH FOR SOMETHING ELSE IF IT IS FOUND THAT
* THE STATEMENT CONTINUES ON THE NEXT CARD
* READACARD SUBSTR(NEXTCARD,CONTPAT<2>,SIZE(CONTPAT<1>))
**CONTPAT<1>**

ADD THE TOKEN STRING TO THE CURRENT STMT STRING

EOSFOUND = CURSTM = CURSTM TOKEN

TEST FOR A KEYWORD ENDING DIRECTLY BEFORE THE END OF STATEMENT
SYMBOL OR A KEYWORD ENDING IN THE END OF STATEMENT COLUMN
OF IN THE COLUMN BEFORE A BLANK IN THE END OF STATEMENT COLUMN

EOSPAT<0> = COL

TESTSTR = SUBSTR(TOKEN,1,SIZE(TOKEN) - SIZE(EOSPAT<0>))

COLKEYWORD = TOKEN

TESTKEY = TESTSTR KEYWORDS . KEYFOUND RPOS(0) & KEYEOS WORKING = WORKING . KEYFOUND

TCKEN =

FOSNEXT IS SET SO AN EOS WILL BE SIGNALLED IMMEDIATELY UPON THE
NEXT CALL OF SCAN; RETURN

EOSNEXT = 1

END OF STATEMENT HAS BEEN FOUND

STATEEOS IS SET FOR THIS STATEMENT AND FOSNEXT IS SET TO 0

EOSIMMED = STATEEOS = 1

EOSNEXT = 0

ADD THE 'UNIDENTIFIED TO END OF STATEMENT' SYMBOL TO
THE WORKING STRING FOR THE PARSER AND RETURN

ADDEOS = WORKING = WORKING 'EOS'

TCKEN =

NOT AN END OF STATEMENT; LOOK FOR A LANGUAGE LABEL IF AT
BEGINNING OF STATEMENT; ANYTHING IN CURSTM FOLLOWED
BY A BLANK OR JUST BLANKS INDICATES A LABEL HAS BEEN PROCESSED
ALREADY
** **
NOTE: SCAN CURSTM BREAK(B')
CURSTM BREAK(D')
LANLABPAT<2> 'STR'

* ERROR IF LANGLABEL DEFINITION DOES NOT SPECIFY EITHER
  A STRING OR A COLUMN AS AN ENDING
  LANLABPAT<2> 'COL'
  :F(LANLABDEFNERR)

* LANLABEL ENDED BY COLUMN
* "BRANCH IF A SEQUENCE OF COLUMN NUMBERS IS NOT SPECIFIED
  AS A LANGLABEL ENDER"
  EQ(SIZE(LANLABPAT<7>),0)
  :S(ONECOLLANLABEND)

* MAY BE A LANGLABEL ENDER IF THE COLUMN NUMBER IS LESS THAN OR
  EQUAL THE UPPER LIMIT OF THE SPECIFIED SEQUENCE
  LE(COLNUM,LANLABPAT<7>)
  :F(NOTLANLABSEP)

* MAY BE A LANGLABEL ENDER IF THE COLUMN NUMBER IS GREATER THAN OR
  EQUAL THE LOWER LIMIT OF THE SPECIFIED SEQUENCE; OTHERWISE IT
  IS NOT A LANGLABEL ENDER
  GE(COLNUM,LANLABPAT<6>)
  :S(CKLLANLABSEP)F(NOTLANLABSEP)

* IF LANGLABELS END IN ONLY ONE COLUMN, AND THIS
  COLUMN NUMBER IS THE RIGHT ONE, THIS MAY BE A LANGLABEL
  ENDER
  ONECOLLANLABEND EQ(COLNUM,LANLABPAT<6>)
  :S(CKLLANLABSEP)F(NOTLANLABSEP)

* A SYMBOL STRING SEPARATES A LANGUAGE LABEL FROM A
  STATEMENT; BRANCH IF THIS IS NOT THE RIGHT CHARACTER
  TO BEGIN A LABEL SEPARATOR
  STRLABSEP LANLABPAT<6> INCHAK
  SUBSTR(TOKEN,1,SIZE(TOKEN) - 1) NOTANY(' ')
  :F(NOTLANLABSEP)

* THIS IS A LABEL SEPARATOR IF ONLY ONE SYMBOL IS REQUIRED
  STRINGSEP = 1
  EQ(SIZE(LANLABPAT<6>),1)
  :S(CKLLANLABSEP)
* ERROR IF THE LABEL SEPARATOR STRING IS LONGER THAN 2 CHARACTERS
*  
* EQ (SIZE (LANLABPAT<6>), 2) : F (LANLABDEFNERR)
*  
* SAVE THE CURRENT CHARACTER AND COLUMN NUMBER
* OLDCOL = CHARCOL
* OLDCHAR = INCHAR
*  
* GET THE NEXT USEABLE CHARACTER AND ITS COLUMN NUMBER
* GETCHAR ('INCHAR', 'CHARCOL') : F (ENDOFINPUT)
*  
* BRANCH IF THE 2 CHARACTERS ARE NOT A LANGLABEL
* SEPARATOR; OTHERWISE ADD THE CHARACTER TO THE
* TOKEN STRING AND SET THE BACKUP FLAG TO 1 TO INDICATE THAT
* BACKUP MUST BE DONE LATER IF THE ITEM PRECEDING THE SEPARATOR
* IS NOT A LABEL
*  
* LANLABPAT<6> OLDCHAR INCHAR : F (NOTLANLAB2CHAR)
* TOKEN = TOKEN INCHAR
* BACKUP = 1 : (CKLANLABSEP)
*  
* BACKUP BY RESTORING THE OLD CHARACTER AND COLUMN NUMBER
* NOTLANLAB2CHAR INCHAR = OLDCHAR
* CHARCOL = OLDCOL
* COLUMNS = OLDCOL
*  
* A LANGUAGE LABEL ENDER HAS BEEN FOUND; ADD THE 'LABEL' ITEM TO
* THE WORKING STRING FOR THE PARSER
*  
* EXTRACT THE 'LABELNAME' FROM THE TOKEN STRING
* CKLANLABSEP TOKEN SPAN (',' ) @LABSTART REM . LABELNAME : (ICKLABELCONT)
* LABELNAME = TOKEN
* LABSTART = 0
*  
* COMPUTE THE COLUMN NUMBER THE LABEL ACTUALLY STARTS IN
* CKLABELCONT LABSTART = STARTCOL + LABSTART
*  
* BRANCH IF THE LABEL MUST START WITHIN A SEQUENCE OF COLUMNS
* NE (LANLABPAT<1>, 1) : S (SEQUFCOLS)
*  
* IF THE LABEL STARTS IN THE CORRECT SINGLE COLUMN, BRANCH BECAUSE
A LABEL HAS BEEN FOUND; OTHERWISE, BRANCH BECAUSE THIS IS NOT A LABEL

EQ(LANLABPAT<4>,LABSTART) : S(FOUNDLABEL)F(NOLABEL)

ERROR IF THE LANGUAGE LABEL DEFINITION SPECIFIES OTHER THAN A SINGLE COLUMN OR A SEQUENCE OF COLUMNS (1 OR 2 IN LANLABPAT<1>) FOR A LABEL BEGINNING

SEQOFCOLS EQ(LANLABPAT<1>,2) : F(LANLABDEFNERR) PRE12410

NOT A LABEL IF THE LABEL STARTS IN A COLUMN WHICH IS GREATER THAN THE UPPER BOUND OF THE SEQUENCE OR LESS THAN THE LOWER BOUND

GT(LABSTART,LANLABPAT<5>) : S(NOLABEL)
LT(LABSTART,LANLABPAT<4>) : S(NOLABEL)

A LABEL HAS BEEN FOUND IF A STRING TERMINATED THE LABEL

LANLABPAT<2> 'STR' RPOS(O) : S(FOUNDLABEL)

CHECK THE FIRST CHARACTER OF THE LABEL TO SEE IF IT IS VALID ACCORDING TO THE LANLABEL DEFINITION; IF NOT, THIS IS NOT A LABEL

SUBSTR(LABELNAME,1,1) ANY(LANLABPAT<8>) : F(NOLABEL)

CHECK THE REMAINING CHARACTERS OF THE LABEL TO SEE IF THEY ARE VALID ACCORDING TO THE LANLABEL DEFINITION; IF NOT, THIS IS NOT A LABEL

EQ(SIZE(LABELNAME),1) : S(FOUNDLABEL)
LABELNAME LEN(1) SPAN(LANLABPAT<9>) RPOS(O) : F(NOLABEL)S(FOUNDLABEL)

THIS IS NOT A LABEL; IF NECESSARY, BACKUP SHOULD BE SET TO 0 AND A BRANCH TAKEN TO A LABEL WHERE BACKUP WILL TAKE PLACE

NOLABEL F(E(BACKUP,1), 1) : S(NOTLANLABSEP) PRE12690
BACKUP = 0 : (NOTLANLAB2CHAR) PRE12710
FOUNDLABEL F(C(StringSEP,1)) : S(LABELCOMMENT) PRE12720

IF COLUMN WAS LABEL INDICATOR AND NO CHARACTER EXISTS BEYOND THE COLUMN (GETCHAR RETURNS WITH FAILURE), IT IS NOT A LABEL

CLDCCL = CHARCOL
MLDCOL = INCHAR

PRE12210
PRE12320
PRE12330
PRE12340
PRE12350
PRE12360
PRE12370
PRE12380
PRE12390
PRE12400
PRE12410
PRE12420
PRE12430
PRE12440
PRE12450
PRE12460
PRE12470
PRE12480
PRE12490
PRE12500
PRE12510
PRE12520
PRE12530
PRE12540
PRE12550
PRE12560
PRE12570
PRE12580
PRE12590
PRE12600
PRE12610
PRE12620
PRE12630
PRE12640
PRE12650
PRE12660
PRE12670
PRE12680
PRE12690
PRE12700
PRE12710
PRE12720
PRE12730
PRE12740
PRE12750
PRE12760
PRE12770
PRE12780
GETCHAR('INCHAR', 'CHARCOL')  
       :F(BACKUPNOLAB)

* IF THE NEXT CHARACTER IS A LABEL DELIMITER (BLANK OR SPECIAL 
  (CHARACTER), THIS IS A LABEL 
  INCHAR LAUDELG 
  :S(LABELYES)

* BACKUP BY RESTORING THE OLD CHARACTER AND ITS COLUMN NUMBER;
  BRANCH BASED ON THE FINDING OF A LABEL OR NOT 

BACKUPNOLAB  INCHAR = OLDCHAR 
CHARCOL = OLDCHAR 
COLNUM = OLDCOL 

LABELYES  INCHAR = OLDCHAR 
CHARCOL = OLDCOL 
COLNUM = OLDCOL 

*:NOTLANLABSEP

*:LABELCOMMENT

* A LABEL HAS BEEN FOUND

* INSERT A COMMENT THAT THE LABEL NAME HAS BEEN PLACED ON 
  A NULL STATEMENT 

LABELCOMMENT INSERTCCM(LABELNAME, 'HAS BEEN MADE THE LABEL ON A NULL ' 
  'STATEMENT.') 

  STRINGSEP = 0

  PLACE ENOUGH BLANKS IN THE CURRENT STMT STRING TO 
  TAKE THE PLACE OF THE LABEL JUST FOUND 
  CURSTM = CURSTM DUPL(' ', SIZE(TOKEN)) 
  WORKING = WORKING ' LABEL' 
  TOKEN = 

*:RETURN

* NOT A LANGUAGE LABEL ENDER; IS IT A BLANK?

* IF NOT A BLANK CHARACTER, GO BACK TO THE BEGINNING OF THE 
  SCAN ROUTINE 

NOTLANLABSEP  INCHAR ' '  
            :S(KEYWORDSKC)F(BEGSCAN1)

PRE12770
PRE12800
PRE12810
PRE12820
PRE12830
PRE12840
PRE12850
PRE12860
PRE12870
PRE12880
PRE12890
PRE12900
PRE12910
PRE12920
PRE12930
PRE12940
PRE12950
PRE12960
PRE12970
PRE12980
PRE12990
PRE13000
PRE13010
PRE13020
PRE13030
PRE13040
PRE13050
PRE13060
PRE13070
PRE13080
PRE13090
PRE13092
PRE13100
PRE13110
PRE13111
PRE13112
PRE13120
PRE13130
PRE13140
PRE13150
PRE13160
LOOK FOR A KEYWORD APPEARING IMMEDIATELY BEFORE THIS
BLANK; BRANCH TO THE UNIDENTIFIED SECTION IF A
KEYWORD IS NOT FOUND

KEYWORDS

TOKEN KEYWORDS . KEYFOUND " RPOS(0) : (ISKEYWORD)

TCKEN KEYWORD2 . KEYFOUND " RPOS(0) : (UNIDENT)

A KEYWORD HAS BEEN FOUND

IF THE KEYWORD MATCHED IS 'THEN', ADD 'UTHEN' TO THE WORKING
STRING FOR
THE PARSER; BRANCH TO ADD THE TOKEN STRING TO THE CURRENT
STATEMENT STRING

ISKEYWORD KEYFOUND KEYTHEN : (NOTATEHEN)
WORKING = WORKING ' UTHEN' : (ADDTOCURSTM)

THE KEYWORD MATCHED IS NOT 'THEN'; ADD THE KEYWORD
TO THE WORKING STRING FOR THE PARSER

NOTATEHEN WORKING = WORKING ' KEYFOUND

ADD THE TOKEN STRING TO THE CURRENT STATEMENT STRING AND RETURN

ADDTOCURSTM CURSTM = CURSTM TOKEN
TOKEN = :

UNIDENTIFIED-TO-END-OF-STATEMENT (UEOS) HAS BEEN FOUND

THE ITEM LOCATED IS UNIDENTIFIED; GO BACK TO THE BEGINNING OF
THE SCANNING ROUTINE TO FIND MORE OF THE TOKEN SO
IT MIGHT BE IDENTIFIED

UNIDENT :

CHECK FOR ERROR AT END OF FILE
END OF FILE WAS FOUND WHEN TRYING TO GET A CHARACTER AT THE
BEGINNING OF THE SCAN ROUTINE (NOT WHEN LOOKING FOR A
SECOND CHARACTER OF A STRING); IF THE TOKEN IS ALL BLANK OR NULL,
TEST IF AN EOS HAS BEEN FOUND BEFORE THIS CALL TO SCAN

ENDOFINPUT TOKEN POS(0) SPAN(' ') RPOS(0)
EQ(SIZE(TOKEN),0)
EQL(OLDSTMENTS,1)

AN EOS HAD BEEN FOUND; ADD END-OF-FILE TO WORKING STRING FOR
PARSER AND RETURN WITH FAILURE

ADDFD WORKING = WORKING ' EOF'
GETCHAR COLNUM = COLNUM + 1

ROUTINE TO GET THE NEXT CHARACTER FROM A NON-IGNORED COLUMN;
RETURN THE CHARACTER AND ITS COLUMN NUMBER

TWO PARAMETERS -- CHAR AND COL, THE NAMES (IN STRING FORM)
OF THE VARIABLES TO HOLD THE RETURN VALUES

INCREMENT COLUMN NUMBER MODULO 80
LE(COLNUM,80)
EQL(COLNUM,1)

READ A NEW CARD IF ONE HAS NOT BEEN SCANNED FOR CONTINUATION
OTHERWISE USE THE ONE THAT WAS SCANNED

EQ(SIZE(NEXTCARD),0)
EQL(NEXTCARD,1)

IF THE SCANNER HAS REACHED END OF FILE PREVIOUSLY OR THIS
ROUTINE HAS REACHED END OF FILE PREVIOUSLY, RETURN WITH FAILURE

EQ(ENDFILE,1)
EQ(GETEND,1)

READ THE CARD AND BRANCH

CARDIMAGE = INPUT

SET THE FLAG IF JUST NOW REACHED END OF FILE AND
FFRETURN WITH FAILURE

GETEND = 1
COPYNZXTCARD CARDIMAGE = NEXTCARD
NEXTCARD =

BRANCH TO GET THE NEXT CHARACTER IF THIS IS AN IGNORED

COLUMN

NOINPUT LO(COLUMN,IGNPAT<1>)
IGNPAT<0> *SFQ*
LE (COLUMN,IGNPAT<2>)
GE (COLUMN,IGNPAT<1>)

RETURN ONE CHARACTER AND ITS COLUMN NUMBER

NOTIGNORE CARDIMAGE TAB(COLUMN-1) LEN(1) *CHAR
$COL = COLUMN
SCANEOSDEF EOSDEF """" ARB . EOSPAT<1> """" =

RETURN

SYMSTR = EOSDEF

ROUTINE TO SCAN DEFINITION OF END OF STATEMENT

ONE PARAMETER -- EOSDEF

EOSPAT<1> = EOSDEF

TFST FOR COLUMN SPECIFICATION CONTAINING ONLY DIGITS

EOSPAT<1> SPAN('0123456789') RPOS(0)

SPECIFICATION INDICATES A COLUMN FOR END OF STATEMENT

EOSPAT<0> = 'COL'
SYMSTR +C_SIZE(EOSDEF),0

SPECIFICATION INDICATES A SYMBOL STRING FOR END OF STATEMENT

EOSPAT<0> = 'STR'

ENDFOSSCAN SCANFOSDEF =
SCAI NGNCOLDEF IGNCOLDEF ARB . IGNPAT<1> ' ' ARB . IGNPAT<2> RPOS(0) =

RETURN

PRE13850
PRE13860
PRE13870
PRE13880
PRE13890
PRE13900
PRE13910
PRE13920
PRE13930
PRE13940
PRE13950
PRE13960
PRE13970
PRE13980
PRE13990
PRE14000
PRE14010
PRE14020
PRE14030
PRE14040
PRE14050
PRE14060
PRE14070
PRE14080
PRE14090
PRE14100
PRE14110
PRE14120
PRE14130
PRE14140
PRE14150
PRE14160
PRE14170
PRE14180
PRE14190
PRE14200
PRE14210
PRE14220
PRE14230
PRE14240
PRE14250
PRE14260
PRE14270
PRE14280
PRE14290
PRE14291
PRE14292
PRE14293
PRE14294
PRE14295
PRE14296
PRE14297
PRE14298
PRE14299
PRE14300
PRE14301
PRE14302
PRE14303
PRE14304
PRE14305
PRE14306
PRE14307
PRE14308
ROUTINE TO SCAN DEFINITION OF IGNORE COLUMNS

** ONE PARAMETER -- IGNCOLDEF
**
** TEST FOR COLUMN SPECIFICATIONS CONTAINING ONLY DIGITS
**
** IGNPAT<1> SPAN(\'0123456789\') RPOS(0) :F(IGNERR)
** IGNPAT<2> SPAN(\'0123456789\') RPOS(0) :F(IGNERR)
**
** SPECIFICATION INDICATES A SEQUENCE OF COLUMNS TO BE IGNORED
**
** IGNPAT<0> = \'SEQ\'
**
** (ENDIGNCOLSCAN)
**
** IGNPAT<1> = IGNCOLDEF
**
** ONEIGN
**
** TEST FOR COLUMN SPECIFICATION CONTAINING ONLY DIGITS
**
** IGNPAT<1> SPAN(\'0123456789\') RPOS(0) :F(IGNERR)
**
** SPECIFICATION INDICATES A SINGLE COLUMN TO BE IGNORED
**
** IGNPAT<0> = \'ONE\'
**
** ENDIGNCOLSCAN SCANIGNCOLDEF =
**
** (RETURN)
**
** SCANIGNCOLDEF
**
** ROUTINE TO SCAN DEFINITION OF COMMENT
**
** ONE PARAMETER -- COMDEF
**
** COMDEF \"\" BREAK(\"\") . CMPAT<3> \"\" SPAN(\'\') =
**
** (OPENCOL)
**
** SPECIFICATION INDICATES A SYMBOL FOR COMMENT OPENER
**
** I = \'1\'
**
** COMDEF \†\† SPAN(\'\') BREAK(\'\') . CMPAT<4> SPAN(\'\') =
**
** (NOOPENCOL)
**
** SPECIFICATION INDICATES A COLUMN NUMBER FOR OPENER
**
** CMPAT<4> SPAN(\'0123456789\') RPOS(0) :F(COMERROR)
**
** SPECIFICATION INDICATES A COLUMN NUMBER AND A COLUMN NUMBER FOR OPENER
**
* I = '11'
NOOPENCOL CCMPAT<4> = : (COMENDER) PRE14690
OPENCOL CCMPAT<3> = : (COMENDER) PRE14700
** SPECIFICATION INDICATES ONLY A COLUMN NUMBER FOR COMMENT OPENER
** I = '01'
COMDEF BREAK(' ') . CCMPAT<4> SPAN(' ') = :F(COMERROR) PRE14710
** TEST FOR COLUMN SPECIFICATION CONTAINING ONLY DIGITS
** CCMPAT<4> SPAN('0123456789') RPOS(0) :F(COMERROR) PRE14720
** SAVE COMMENT OPENER 'DESCRIPTOR'
COMENDER CCMPAT<1> = I
I = '10'
COMDEF """" BREAK("""") . CCMPAT<5> """" = :F(ENDCOL) PRE14730
COMDEF SPAN(' ') = :F(NOCLOSECOL) PRE14740
** SPECIFICATION INDICATES A SYMBOL FOR COMMENT ENDER
** COMDEF '+' SPAN(' ') REM . CCMPAT<6> = :F(NOCLOSECOL) PRE14750
** TEST FOR COLUMN SPECIFICATION CONTAINING ONLY DIGITS
** CCMPAT<6> SPAN('0123456789') RPCS(0) :F(COMERROR) PRE14760
** SPECIFICATION INDICATES A SYMBOL AND A COLUMN NUMBER FOR ENDER
** I = '11'
NCLOSECOL CCMPAT<6> = : (ENDCOMSCAN) PRE14770
ENDCOL CCMPAT<5> = : (ENDCOMSCAN) PRE14780
** SPECIFICATION INDICATES ONLY A COLUMN NUMBER FOR COMMENT CLOSER
** I = '01'
COMDEF REM . CCMPAT<6> = :F(COMERROR) PRE14790
** TEST FOR COLUMN SPECIFICATION CONTAINING ONLY DIGITS
** CCMPAT<6> SPAN('0123456789') RPOS(0) :F(COMERROR) PRE14800
** SAVE COMMENT ENDER 'DESCRIPTOR'
ENDCOMSCAN CCMPAT<2> = I
SCANCCMDEF = : (RETURN) PRE14810
SCANCONTDEF CONTDEF BREAK(' ') . CONTPAT<1> SPAN(' ') = :F(NOCONT)
+ * ROUTINE TO SCAN DEFINITION OF CONTINUATION
+ * ONE PARAMETER -- CONTDEF
+ * CONTPAT<2> = CONTDEF
+ * CONTPAT<2> SPAN('0123456789') RPOS(0) :F(CONTERR)
+ * CONTDEF = :F(FINDCONTSCAN)
+ * NFCONT CONTPAT<1> = CONTDEF
+ * NFCONT 'NONE' RPOS(0) :F(CONTERR)
+ * NFCONT :F(RETURN)
+ * FINDCONTSCAN SCANCONTDEF =:F(RETURN)
* SCANLITDEF LITDEF BREAK(' ') . LITPAT<1> SPAN(' ') = :F(LITERERROR)
* ROUTINE TO SCAN DEFINITION OF LITERAL
* ONE PARAMETER -- LITDEF
* LITDEF BREAK(' ') . LITPAT<2> SPAN(' ') = :F(NOSPESIAL)
+ NOSPESIAL LITDEF REM . LITPAT<3> = :F(ENDLITSCAN)
+ LITDEF REM . LITPAT<2> = :F(RETURN)
+ LITPAT<3> = :F(RETURN)
+ ENDLITSCAN SCANLITDEF = :F(RETURN)
* SCANLANLABDEFF
+ ROUTINE TO SCAN DEFINITION OF LANGUAGE LABEL
+ ONE PARAMETER -- LANLABDEFF
+ LANLABDEFF BREAK(' ') . SPEC = :F(LANLABERR)
+ SPEC BREAK(' ' ). LANLABPAT<4> ' ' REM . LANLABPAT<5> = :F(ONECOLOPEN)
+ SPECIFICATION INDICATES A SEQUENCE OF COLUMNS TO START LABEL
+ TEST FOR COLUMN SPECIFICATIONS CONTAINING ONLY DIGITS
+ LANLABPAT<4> SPAN('0123456789') RPOS(0) :F(LANLABERR)
+ LANLABPAT<5> SPAN('0123456789') RPOS(0) :F(LANLABERR)
LANLABPAT<1> = 2

* TEST FOR COLUMN SPECIFICATION CONTAINING ONLY DIGITS
ONECOLOPEN SPEC SPAN(0123456789) RPOS(0) :F(LANLABERR)
* SPECIFICATION INDICATES A COLUMN TO START LABEL
  LANLABPAT<1> = 1
  LANLABPAT<4> = SPEC
  LANLABPAT<5> =
CLOSELANLAB LANLABDEF SPAN(‘‘‘’) BREAK(‘‘‘’) . SPEC = :F(LANLABERR)
  SPEC “” BREAK(“”‘”) . LANLABPAT<6> “” = :F(COLCLOSE)
* SPECIFICATION INDICATES A SYMBOL STRING TO TERMINATE LABEL
  LANLABPAT<2> = ‘STR’
  LANLABPAT<7> = :OPENCHARS
  COLCLOSE LANLABAT<2> = ‘COL’
  SPEC BREAK(‘‘‘’) . LANLABPAT<6> ‘‘’’ REM . LANLABPAT<7> =
  :F(ONECOLCLOSE)
* SPECIFICATION INDICATES A SEQUENCE OF COLUMNS TO TERMINATE A LABEL
* TEST FOR COLUMN SPECIFICATIONS CONTAINING ONLY DIGITS
  LANLABPAT<6> SPAN(0123456789) RPOS(0) :F(LANLABERR)
  LANLABPAT<7> SPAN(0123456789) KPOS(0) :F(LANLABERR)
  LANLABPAT<3> = 2
* TEST FOR COLUMN SPECIFICATION CONTAINING ONLY DIGITS
ONECOLCLOSE SPEC SPAN(0123456789) RPOS(0) :F(LANLABERR)
* SPECIFICATION INDICATES A COLUMN TO TERMINATE LABEL
  LANLABPAT<3> = 1
  LANLABPAT<6> = SPEC
  LANLABPAT<7> =
  DETERMINE SET OF OPENING CHARACTERS
  OPENCHARS LANLABDEF SPAN(‘‘‘’”) “” BREAK(“”‘”) . LANLABPAT<8> “” =
  :F(LANLABERR)
  DETERMINE SET OF REMAINING CHARACTERS
  LANLABDEF SPAN(‘‘‘’”) “” BREAK(“”‘”) . LANLABPAT<9>
```
+ STARTP SCAN() "\" RPOS(0) = :F(LANLABERR)S(RETURN) :F(PSCANERR) PRE16050
+ |
+ ***STRUCTURE PARSER -- NC PARAMETERS
+ |  THE WORKING STRING ACTS AS A SYNTACTIC STACK
+ |  TRY TO MATCH A LABEL
+ |  START WORKING 'LABEL' RPOS(0) :F(NXT.1) PRE16140
+ |  CALL SEMANTIC ROUTINE 0, SCAN, AND RETURN TO START
+ |  ROUTO() PRE16170
+ |  SCAN AND BRANCH
+ |  SCAN() :S(START)F(PSCANERR) PRE16220
+ |  TRY TO MATCH AN ENDCASE
+ |  NXT.1 WORKING 'ENDCASE' RPOS(0) :F(NXT.2) PRE16260
+ |  SCAN AND BRANCH
+ |  SCAN() :S(ECASESTMT)F(PSCANERR) PRE16330
+ |  TRY TO MATCH AN ENDF
+ |  NXT.2 WORKING 'ENDIF' RPOS(0) :F(NXT.3) PRE16340
+ |  SCAN AND BRANCH
+ |  SCAN() :S(EIFSTMT)F(PSCANERR) PRE16400
+ |  TRY TO MATCH AN ENDOWHILE
+ |  NXT.3 WORKING 'ENDWHILE' RPOS(0) :F(NXT.4) PRE16420
+ |  SCAN AND BRANCH
+ |  SCAN() :S(EWHILESTMT)F(PSCANERR) PRE16470
+ |  TRY TO MATCH AN ENDOREPEAT
+ |
```
NXT.4   WORKING 'ENDREPEAT' RPOS(0) : F(NXT.5)
* SCAN AND BRANCH
* SCAN() : S(EREPEATSTMT) F(PSCANERR)
* TRY TO MATCH AN ENDDDO
NXT.5   WORKING 'ENDDDO' RPOS(0) : F(NXT.6)
* SCAN AND BRANCH
* SCAN() : S(EDOSTMT) F(PSCANFRR)
* TRY TO MATCH AN ENDDDOGROUP
NXT.6   WORKING 'ENDDDOGROUP' RPOS(0) : F(NXT.7)
* SCAN AND BRANCH
* SCAN() : S(EDOGRPSTMT) F(PSCANERR)
* TRY TO MATCH AN ELSE
NXT.7   WORKING 'ELSE' RPOS(0) : F(NXT.8)
* CALL SEMANTIC ROUTINE 1, SCAN AND RETURN TO START
* ROUT1()
* SCAN() : S(START) F(PSCANERR)
* TRY TO MATCH A QUITLOOP
NXT.8   WORKING 'QUITLOOP' RPOS(0) : F(NXT.9)
* SCAN AND BRANCH
* SCAN() : S(QUITSTMT) F(PSCANERR)
* TRY TO MATCH A CASE
NXT.9   WORKING 'CASE' RPOS(0) : F(NXT.10)
* SCAN AND BRANCH
* SCAN() : S(CASESTMT) F(PSCANERR)
PRE16490
PRE16500
PRE16510
PRE16520
PRE16530
PRE16540
PRE16550
PRE16560
PRE16570
PRE16580
PRE16590
PRE16600
PRE16610
PRE16620
PRE16630
PRE16640
PRE16650
PRE16660
PRE16670
PRE16680
PRE16690
PRE16700
PRE16710
PRE16720
PRE16730
PRE16740
PRE16750
PRE16760
PRE16770
PRE16780
PRE16790
PRE16800
PRE16810
PRE16820
PRE16830
PRE16840
PRE16850
PRE16860
PRE16870
PRE16880
PRE16890
PRE16900
PRE16910
PRE16920
PRE16930
PRE16940
PRE16950
PRE16960
TRY TO MATCH A DO

NXT.10 WORKING 'DO' RPOS(0) :F(NXT.11)

* SCAN AND BRANCH

NXT.11 WORKING 'DO GROUP' RPOS(0) :F(NXT.12)

* SCAN AND BRANCH

NXT.12 WORKING 'IF' RPOS(0) :F(NXT.13)

* SCAN AND BRANCH

NXT.13 WORKING 'REPEAT' RPOS(0) :F(NXT.14)

* SCAN AND BRANCH

NXT.14 WORKING 'WHILE' RPOS(0) :F(NXT.15)

* SCAN AND BRANCH

TRY TO MATCH A STRUCTURE LIST AND END OF FILE; IF MATCHED,
RETURN TO CALLING ROUTINE

NXT.15 WORKING 'STRLIST EOF' = 'PROG' :S(RETURN)

* TRY TO MATCH AN UNIDENTIFIED TO END OF STATEMENT; IF NOT
MATCHED TAKE ERROR EXIT
WORKING 'UEOS' RPOS(0) = 'STRUC' :F(PUNIDERR)

CALL SEMANTIC ROUTINE 01 AND BRANCH TO STRUCTURE SECTION

ROUT1()

CASE HAS BEEN RECOGNIZED, REDUCE TO CASESTM; CALL SEMANTIC ROUTINE 2, SCAN AND RETURN TO START

CASESTM WORKING 'CASE UEOS' RPOS(0) = 'CASESTM' :F(PCASEERRR)

ROUT2()

SCAN()

S(START)F(RETURN)

DO HAS BEEN RECOGNIZED, REDUCE TO DOSTM; CALL SEMANTIC ROUTINE 3, SCAN AND RETURN TO START

DOSTM WORKING 'DO UEOS' RPOS(0) = 'DOSTM' :F(PDOERR)

ROUT3()

SCAN()

S(START)F(RETURN)

DOGROUP HAS BEEN RECOGNIZED, REDUCE TO DOGROUPSTM; CALL SEMANTIC ROUTINE 4, SCAN AND RETURN TO START

DOGROUPSTM WORKING 'DOGROUP UEOS' RPOS(0) = DOGROUPSTM

ROUT4()

SCAN()

S(START)F(RETURN)

IF HAS BEEN RECOGNIZED, REDUCE TO IFTHEN; CALL SEMANTIC ROUTINE 5, SCAN AND RETURN TO START

IFSTM WORKING 'IF UTHEN' RPOS(0) = 'IFTHEN' :F(PIFERR)

ROUT5()

SCAN()

S(START)F(PSCANERRP)

REPEAT HAS BEEN RECOGNIZED, REDUCE TO REPUNTIL; SCAN

REPEATSTM WORKING 'REPEAT UNTIL' RPOS(0) = REPUNTIL

ROUT6()

SCAN()

S(START)F(RETURN)

REPUNTIL HAS BEEN RECOGNIZED, REDUCE TO REPEATSTM; CALL SEMANTIC ROUTINE 6, SCAN AND RETURN TO START

REPSTM WORKING 'REPUNTIL UEOS' RPOS(0) = REPEATSTM

ROUT6()
* WHILE HAS BEEN RECOGNIZED, REDUCE TO WHILESTM; CALL SEMANTIC
  ROUTINE 7, SCAN AND RETURN TO START
  WHILESTM  WORKING 'WHILE UEOS' RPOS(0) = :F(PWHERR)
  ROUT7()
  SCAN() :S(START IF(RETURN)
  QUITLOOP HAS BEEN RECOGNIZED, REDUCE TO STRUCTURE; CALL SEMANTIC
  ROUTINE 8 AND BRANCH TO STRUCTURE SECTION
  QUITSTM  WORKING 'QUIT LOOP UEOS' RPOS(0) = :F(PQUERR)
  ROUT8() :S(STRUCTURE)
  REDUCTION OF CASE-EN CASE TO STRUCTURE; CALL SEMANTIC
  ROUTINE 9 AND BRANCH TO STRUCTURE SECTION
  CASESTM  WORKING 'CASESTM STRLIST ENCASE UEOS'
  RPOS(0) = 'STRUC'
  ROUT9() :F(PECASEERR)
  :S(STRUCTURE)
  REDUCTION OF IF-THEN-ENDIF TO STRUCTURE; CALL SEMANTIC ROUTINE
  10 AND BRANCH TO STRUCTURE SECTION
  EIFSTM  WORKING 'IFTHEC ENDIF UEOS' RPOS(0) = :F(NXT.16)
  ROUT10() :S(STRUCTURE)
  REDUCTION OF IF-THEN-ELSE-ENDIF TO STRUCTURE; CALL SEMANTIC ROUTINE
  10 AND BRANCH TO STRUCTURE SECTION
  ELSESTM  WORKING 'IFTHELSE ENDIF UEOS' RPOS(0) = :F(PIFTHEERR)
  ROUT11() :S(STRUCTURE)
  REDUCTION OF WHILE-ENDWHILE TO STRUCTURE; CALL SEMANTIC
  ROUTINE 11 AND BRANCH TO STRUCTURE SECTION
  FWHILESTM  WORKING 'WHILESTM STRLIST ENDMWHILE UEOS'
  RPOS(0) = 'STRUC'
  ROUT11() :F(PEWHERR)
  :S(STRUCTURE)
  REDUCTION OF REPEAT-ENDREPEAT TO STRUCTURE; CALL SEMANTIC
  ROUTINE 12 AND BRANCH TO STRUCTURE SECTION
  ERFPEATSTM  WORKING 'REPEATSTM STRLIST ENDRPEAT UEOS' RPOS(0) =
ROUT12()   'STRUC' :F(PEREPPER)  PFF18410
            :{STRUCTURE}
            |     REDUCTION OF DO-ENDDO TO STRUCTURE; CALL SEMANTIC ROUTINE
            |     13 AND BRANCH TO STRUCTURE SECTION
            |     RETURN TO START
            +   ENDSTMT  WORKING 'DCSTMT STRLIST ENDDO UEOS'
            +   RPOS(0) = 'STRUC' :F(PEDJERR)
            |       ROUT13() :{STRUCTURE}
            |       |     REDUCTION OF DOGROUP-ENDDOGROUP TO STRUCTURE; CALL SEMANTIC
            |       |     ROUTINE 14 AND BRANCH TO STRUCTURE SECTION
            |       |   ENDOGRPSTMT  WORKING 'DOGROUPSTMT STRLIST ENDDOGROUP UEOS'
            |       |   RPOS(0) = 'STRUC' :F(PEDGRPERR)
            |       |   ROUT14() :{STRUCTURE}
            |       |         LABEL AND A STRUCTURE REDUCE TO STRUCTURE; RETURN TO STRUCTURE
            |       |         SECTION
            |       |   STRUCTURE  WORKING 'LABEL STRUC' RPOS(0) =
            |       |         'STRUC' :S(STRUCTURE)
            |       |         ELSE AND STRUCTURE REDUCE TO ELSESTRUCTURE; RETURN TO STRUCTURE
            |       |         SECTION
            |       |   WORKING 'ELSE STRUC' RPOS(0) = 'ELSESTRUC'
            |       |         :S(STRUCTURE)
            |       |   LABEL AND ELSE STRUCTURE REDUCE TO ELSESTRUCTURE; RETURN TO STRUCTURE
            |       |         SECTION
            |       |   WORKING 'LABEL ELSESTRUC' RPOS(0) =
            |       |         'ELSESTRUC' :S(STRUCTURE)
            |       |         IFTHEN CLAUSE AND ELSESTRUCTURE REDUCE TO IFTHELSE; SCAN AND
            |       |         RETURN TO START
            |       |   WORKING 'IFTHCL ELSESTRUC' RPOS(0) =
            |       |         'IFTHELSE'
            |       |         :F(NXT.17)
            |       |         SCAN() :S(START) :F(RETURN)
            |       |         IF-THEN AND STRUCTURE REDUCE TO IF THEN CLAUSE; SCAN AND
            |       |         RETURN TO START
            +   NXT.17  WORKING 'IFTHEN STRUC' RPOS(0) =
            |       |         'IFTHCL' :F(NXT.18)
            |       |         PRE18880
* SCAN() : S(START) F(RETURN) PRE18890
* STRLIST AND STRUCTURE REDUCE TO STRLIST; CALL SEMANTIC ROUTINE
* 15, SCAN AND RETURN TO START
* NXT.18 WORKING 'STRLIST STRUCT RPOS(0) = ' : F(NXT.19)
* + 'STRLIST'
* ROUT15() SCAN() : (START) PRE18940
* STRUC REDUCES TO STRLIST; CALL SEMANTIC ROUTINE 15, SCAN
* AND RETURN TO START
* NXT.19 WORKING 'STRUCT RPOS(0) = ' : F(PCSTRERR)
* ROUT15() SCAN() : (START) PRE19010
* MACROSCAN MACCALL POS(0) '<MACRO CALL>': = F(MACCALLERR)
* ROUTINE TO SCAN A MACRO DEFINITION
* ONE PARAMETER -- MACCALL, THE INPUT CARD (OR FIRST OF TWO)
* SPECIFYING A MACRO CALLING FORM
* *********************
* BREAK OUT THE MACRO NAME
* MACCALL BREAK('') . NAME '(' = F(MACCALLERR)
* BREAK OUT THE PARAMETER LIST, READING IN ANOTHER CARD IF NECESSARY
* MACCALL BREAK('') . PLIST ')' RPOS(0) : S(GETMPARMS)
* MACCALL = MACCALL TRIM(INPUT)
* MACCALL BREAK('') . PLIST ')' RPOS(0) : F(MACCALLERR)
* BREAK OUT THE INDIVIDUAL PARAMETERS; PLACE THE NUMREP OF
* PARAMETERS IN MPARM<0> AND THE PARAMETERS IN ELEMENTS 1-6
* OF MPARM
* GETMPARMS
* I = 0
* I = I + 1
* PLIST BREAK('') . MPARM<1> ',' = S(MACPARMS)
* PLIST REM . MPARM<1>
* MPARM<0> = I
* MACDEF = TRIM(INPUT)
* MACDLF '<MACRO DEFINITION>': = RPOS(0) : F(MACDEFERR)
* READ THE MACRO DEFINITION AND PLACE IT IN MACREPSTR

MACREP
MACDEF = INPUT
MACDEF POS(0) 'ENC MACRO DEFINITION' SPAN(' ',') RPOS(0)
MACREPSTR = MACREPSTR MACDEF
MACDEF = S(ENDMACDEF) MACREP

* ADD THE MACRONAME TO THE MACRONAME PATTERN
ENDMACDEF MACRONAME = MACRONAME P NAME

* START THE PARAMETER PATTERN WITH A SERIES OF 81 BLANKS
PARM = DUPL(' ',81)

* ADD THE PARAMETERS TO THE PARAMETER PATTERN
PARMPAT PARM = PARM | MPARM<i>
I = GT(I,1) 1 - 1

* BREAK THE MACRO DEFINITION INTO COMPONENTS AND PLACE THEM IN
* ELEMENTS 1-20 OF MPAT

MACDEFLOOP I = I + 2
MACREPSTR ADD . COPY PARM . PARAM = F(MACDEFEND)
MACREPSTR POS(0) SPAN(' ',' ') . BLKS =
MPAT<i> = COPY
MPAT<i + 1> = PARM BLKS
BLKS = GT(SIZE(MACREPSTR),0) :S(MACDEFLOOP)F(MACDEFLEN)

MACDEFEND MPAT<i> = MACREPSTR

* PLACE THE NUMBER OF COMPONENTS IN MPAT<i>
MPAT<i> = I

MACDEFLEN MPAT<i> = I + 1

* STORE THE MPAT ARRAY IN ELEMENT 'MACRONAME' OF TABLE MACROTAB
* STORE THE MPARM ARRAY IN ELEMENT 'MACRONAME P' OF TABLE MACROTAB
* AND RETURN TO CALLING ROUTINE

RETMACDEF MACPCTABNAME> = COPY(MPAT)
MACROTABNAME 'P' > = COPY(MPARM)
FINOUT = NAME ' MACRO DEFINITION PROCESSED.'
FINOUT = ' ' FINOUT = ' ' :
RETMACDEF

SCANDEFS &ANCHOR = 1
ROUTINE TO SCAN DEFINITIONS AND CALL APPROPRIATE
INDIVIDUAL SCANNING Routines

ONE PARAMETER -- CARD; CARD IS THE FIRST CARD OF THE
FIRST DEFINITION

I = 1

DEFNAME = 'NULL STATEMENT' | 'GO TO STATEMENT' |
| 'IF THEN GO TO STATEMENT' | 'ASSIGN MINUS STATEMENT' |
| 'ASSIGN PLUS STATEMENT' | 'DECLARATION STATEMENT' |
| 'LITERAL' | 'COMMENT' | 'UNIQUE LABEL' |
| 'UNIQUE IDENTIFIER' | 'GREATER THAN' | 'NOT' |
| 'LESS THAN OR EQUAL' | 'GREATER THAN OR EQUAL' |
| 'NOT EQUAL' | 'CONTINUATION' | 'END OF STATEMENT' |
| 'IGNORE COLUMNS' | 'LANGLABEL' |

DETERMINE DEFINITION NAME

NEXTDEF CARD 'DEFNAME . SELECT ' > :: = :
SELECT ARB . STMTTYPE 'STATEMENT' RPOS(0) :
F(LEGALDEF)

BRANCH IF A STATEMENT DEFINITION

STMTTYPE 'END OF' RPOS(0) :
F(NOTEOS)

IF END OF STATEMENT DEFINITION, TRIM TRAILING BLANKS
AND BRANCH

CARD = TRIM(CARD)

:(EOSSCAN)

STATEMENT PROTOTYPES REQUIRE TWO INPUT CARDS, SO READ
ANOTHER CARD AND TRIM THE TRAILING BLANKS FROM THE
ENTIRE DEFINITION

NOTEOS CARD = TRIM(CARD INPUT)

BRANCH TO THE APPROPRIATE LABEL

STMTTYPE 'NULL' RPOS(0) :
S(NULLSCAN)
STMTTYPE 'GO TO' RPOS(0) :
S(GOTOSCAN)
STMTTYPE 'IF THEN GO TO' RPOS(0) :
S(IFSCAN)
STMTTYPE 'ASSIGN MINUS' RPOS(0) :
S(MINUSSCAN)
STMTTYPE 'ASSIGN PLUS' RPOS(0) :
S(PLUSSCAN)
ERROR IF NOT A VALID STATEMENT TYPE IN THE DEFINITION NAME

DECLAREATION RPOS(:) :S(DCLSCAN)F(ILLEGALDEF)

IF LANGUAGE LABEL DEFINITION, BRANCH

NOTASTMT SELECT 'LANGLABEL' RPOS(:) :S(LANGLABELSCAN)

ALL OTHER DEFINITIONS REQUIRE ONLY ONE CARD, SO TRIM THE
TRAILING BLANKS FROM THIS CARD

CARD = TRIM(CARD)

BRANCH APPROPRIATELY BASED ON DEFINITION NAME

SELECT 'BREAK' RPOS(:) :E($SEL('SCAN'))
SELECT 'UNIQUE' RPOS(:) :S($UNYTE('SCAN'))
SELECT 'IGNORECOLUMNS' RPOS(:) :S(IGNSCAN)
SELECT 'NOT EQUAL' RPOS(:) :S(NEQSCAN)
SELECT 'GREATER THAN' RPOS(:) :S(GTHSCAN)
SELECT 'LESS THAN OR EQUAL' RPOS(:) :S(GEQSCAN)

ILLEGALDEF OUTPUT = 'CARD WITH INVALID DEFINITION NAME OR CARD NOT '
'CONTAINING A DEFINITION ENCOUNTERED -> '

RETURN IF THIS SHOULD HAVE BEEN THE LAST DEFINITION

FOR(1,19)

OUTSKIPMSG OUTPUT = 'SKIPPING TO NEXT DEFINITION'

INCREMENT DEFINITION COUNTER AND LOCATE NEXT CARD
CONTAINING A DEFINITION NAME IN ANGLE CARD BRACKETS

1 = 1 + 1
CARD = INPUT
CARD '<' :S(NEXTDEF)F(SKIPCARD)

DETERMINE IF THE ENTIRE OPENING CHARACTER SET IS CONTAINED ON
THE FIRST CARD; IF SO, BRANCH TO LASTSET; IF NOT, READ
IN ANOTHER CARD AND TRIM THE TRAILING BLANKS FROM THE ENTIRE
DEFINITION

PRE20720
* LANGLABELSCAN CARD BREAK(' ') ' ' BREAK(' ') ' ' REM . REST
   REST SPAN(' ') =
   REST "": BREAK("": "": "": "": "": :S(LASTSET)
NEEDONE CARD = TRIM(CARD INPUT)
* CALL THE LANGUAGE LABEL DEFINITION SCANNER
CALLSCAN SCANLANLABDEF(CARD) : (NEXTIN)
* DETERMINE IF THE ENTIRE REMAINING CHARACTER SET IS CONTAINED
ON THE FIRST CARD: IF NOT, BRANCH TO READ IN ANOTHER CARD
* AND TRIM THE TRAILING BLANKS: IF SO, TRIM THE TRAILING
* BLANKS AND BRANCH TO CALL THE SCANNING ROUTINE
LASTSET REST SPAN(' ') =
   REST "": BREAK("": "": "": "": "": :F(NEEDONE)
   CARD = TRIM(CARD)
   : (CALLSCAN)
   : (NEXTIN)
NOTSCN SCANOPERDEF('NOT',CARD)
CONTINUATIONSCAN SCANCONTDEF(CARD)
COMMENTSSCAN SCANCOMDEF(CARD)
LITERALSSCAN SCANLITDEF(CARD)
NULLSCAN SCANNULLDEF(CARD)
GOTOSCAN SCANGOTDEF(CARD)
IFSCAN SCANIFDEF(CARD)
MINUSSCAN SCANMINUSDEF(CARD)
PLUSSCAN SCANPLUSDEF(CARD)
DCLSCAN SCANDCLDEF(CARD)
LABELSCAN SCANLABELDEF(CARD)
IDENTIFIERSSCAN SCANVARDEF(CARD)
EOSSCAN SCANEOSSDEF(CARD)
IGNSCAN SCANIGSCLDEF(CARD)
NEGSCN SCANOPERDEF('NEG',CARD)
GTHANSCAN SCANOPERDEF('GTHAN',CARD)
GEUSCAN SCANOPERDEF('GEU',CARD)
LEUSCAN SCANOPERDEF('LEU',CARD)
* RETURN IF THIS WAS THE LAST DEFINITION
NEXTIN EQ(11,19)
ANCHOR = 0
* INCREMENT DEFINITION COUNTER AND READ ANOTHER CARD
INCDEFNCTR I = I + 1
CARD = INPUT
SCANOPERDEF OPERDEF "": ARR . $TYPEDEF "": RpOs(i) =
   : (NEXTDEF)
   : (RETURN)
   :F($($TYPEDEF 'DEFERR'))
SCANOPERDEF =

: (RETURN)  

SCANVARDEF

 ROUTINE TO SCAN DEFINITION OF UNIQUE VARIABLE

LINE PARAMETER -- VARDEF

VARDEF BREAK(' ') . VARPAT<1> SPAN(' ') = :F(VARERROR)
VARDEF BREAK(' ') . VARPAT<2> = :F(NOVAREND)
VARDEF SPAN(' ') =
VARPAT<3> = VARDEF
VARDEF =
VARPAT<3> =
VARPAT<2> = VARDEF
VARDEF =

NOVAREND

: (TESTVARCOL)

TESTVARCOL VARPAT<2> SPAN('0123456789') RPOS(0) = :F(VARERROR)

SCANNULLDEF

: (RETURN)

ROUTINE TO SCAN DEFINITION OF NULL STATEMENT

LINE PARAMETER -- NULLDEF

I = -1
J = 0

NULLDEFLOOP I = I + 2
NULLDEF ARB . COPY 'LAB' =
NULLDEF POS(0) SPAN(' ') . BLKS =
J = J + 1
NULLPAT<i> = COPY

SAVE PARAMETER NAME AND BLANKS FOLLOWING IT

NULLPAT<i> = 'LAB' BLKS
BLKS =
GT(SIZE(NULLDEF), 0)
**SCANGETDEF**

* ROUTINE TO SCAN DEFINITION OF GO TO STATEMENT

* ONE PARAMETER -- GOTODEF

**

PARM = 'LAB' | 'DES'
I = -1
J = 0
GOTODEFFLOOP I = I + 2
GOTODEF ARB, COPY PARM, PARAM =
GOTODEF POS(0) SPAN(1, *), BLKS =
J = J + 1
GOTOPAT<1> = COPY

* SAVE PARAMETER NAME AND BLANKS FOLLOWING IT

GOTOPAT<1 + 1> = PARAM BLKS
BLKS =
GTSIZE(GOTODEF), 0

**

GOTODEFEND GOTOPAT<1> = GOTODEF
GOTOPAT<0> = I
GOTODEF =
GOTODEFFLEN GOTOPAT<0> = I + 1
ENDGOTODEF SCANGETDEF =

**

TEST FOR ALL PARAMETERS SPECIFIED

* EO(J, 2)

**

SCANIFDEF

**

S(NULDELFOOP)F(NULDELFLN)
ROUTINE TO SCAN DEFINITION OF IF THEN GO TO STATEMENT
ONE PARAMETER -- IFDEF

PARM = 'LAB' | 'VAL1' | 'RL' | 'VAL2' | 'DES'
J = -1
J = 0
IFDEFLOOP I = I + 2
IFDEF AND COPY PARM PARAM = F(IFDEFEND)
IFDEF POS(O) SPAN(* *) BLKS =
J = J + 1
IFPAT<1> = COPY

SAVE PARAMETER NAME AND BLANKS FOLLOWING IT
IFPAT<1 + 1> = PARAM BLKS
BLKS = GT(SIZE(IFDEF),0)

IFDEFEND IFPAT<1> = IFDEF
IFPAT<0> = 1
IFDEF = ENDIFDEF
IFDEFLEN IFPAT<0> = 1 + 1
ENDIFDEF SCANIFDEF =

TEST FOR ALL PARAMETERS SPECIFIED
EQ(I,J,5)
RETURN F(IFPARMERR)

ROUTINE TO SCAN DEFINITION OF DECLARATION STATEMENT
ONE PARAMETER -- DCLDEF

I = -1
J = 0
DCLDEFLOOP I = I + 2
DCLDEF AND COPY 'VARNAM' = F(DCLDEFEND)
DCLDEF POS(O) SPAN(* *) BLKS =
J = J + 1
DCLPAT<i> = COPY

* SAVE PARAMETER NAME AND BLANKS FOLLOWING IT
DCLPAT<i + 1> = 'VARNAM' BLKS
BLKS = GT(SIZE(DCLDEF),0)
DCLDEFEND DCLPAT<i> = DCLDEF
DCLPAT<i> = 1
DCLDEF = DCLDEF = (ENDDCLDEF)
DCLDEFLEN DCLPAT<i> = 1 + 1
ENDDCLDEF SCANDCLDEF = (RETURN)
SCANMINUSDEF

ROUTINE TO SCAN DEFINITION OF ASSIGN MINUS STATEMENT
ONE PARAMETER -- MINUSDEF

PARM = 'LAB' | 'VAR' | 'VAL1' | 'VAL2'
I = -1
J = 0
MINUSDEFLOOP J = I + 2
MINUSDEF ARB . COPY PARM . PARAM = F(MINUSDEFEND)
MINUSDEF POS(0) SPAN(' ') . BLKS =
J = J + 1
MINUSPAT<i> = COPY

* SAVE PARAMETER NAME AND BLANKS FOLLOWING IT
* MINUSPAT<i + 1> = PARAM BLKS
BLKS = GT(SIZE(MINUSDEF),0)
MINUSDEFEND MINUSPAT<i> = MINUSDEF
MINUSPAT<i> = 1
MINUSDEF = (ENDMINUSDEF)
MINUSDEFLEN MINUSPAT<i> = 1 + 1
ENDMINUSDEF SCANMINUSDEF =

* TEST FOR ALL PARAMETERS SPECIFIED
EQ(J,4)
**SCANPLUSDEF**

```
: S(RETURN)F(MINPARMERR)
```

**ROUTINE TO SCAN DEFINITION OF ASSIGN PLUS STATEMENT**

**ONE PARAMETER -- PLUSDEF**

```
* PARM = 'LAB' | 'VAR' | 'VAL1' | 'VAL2'
* I = -1
* J = 0
```

**PLUSDEFLOOP**

```
I = I + 2
PLUSDEF ARB . COPY PARM . PARAM = F(PLUSDEFEND)
PLUSOFF POS(I) SPAN(‘ ’ ) . BLKS =
J = J + 1
PLUSPAT<I> = COPY
```

**SAVE PARAMETER NAME AND BLANKS FOLLOWING IT**

```
PLUSPAT<I + 1> = PARAM BLKS
BLKS =
GT(SIZE(PLUSDEF),0)
```

**PLUSDEFEND**

```
PLUSPAT<I> = PLUSDEF
PLUSPAT<Q> = I
PLUSDEF =
PLUSDEFLEN PLUSPAT<Q> = I + 1
ENDPLUSDEF SCANPLUSDEF =
```

**TEST FOR ALL PARAMETERS SPECIFIED**

```
EQ(J,4)
```

**SCANLABFLDEF**

```
: S(RETURN)F(PLUSPARAMERR)
```

**ROUTINE TO SCAN DEFINITION OF UNIQUE LABEL**

**ONE PARAMETER -- LABELDEF**

```
* LABELDEF BREAK(‘ ’ ) . LABELPAT<I> SPAN(‘ ’ ) = F(LABERROR)
```

PRE22890
PRE22900
PRE22910
PRE22911
PRE22912
PRE22920
PRE22920
PRE22930
PRE22940
PRE22950
PRE22951
PRE22952
PRE22960
PRE22970
PRE22980
PRE22990
PRE23000
PRE23010
PRE23019
PRE23020
PRE23030
PRE23040
PRE23050
PRE23060
PRE23070
PRE23080
PRE23090
PRE23100
PRE23110
PRE23120
PRE23130
PRE23140
PRE23150
PRE23160
PRE23170
PRE23180
PRE23190
PRE23200
PRE23210
PRE23220
PRE23230
PRE23240
PRE23250
PRE23260
PRE23270
PRE23280
PRE23290
PRE23291
PRE23292
PRE23300
PRE23340
PRE23350
PRE23360
PRE23370
PRE23380
LABELDEF BREAK(' ') LABELPAT<2> = :F(NOLABEL)
LABELDEF SPAN(' ') =
LABELPAT<3> = LABELDEF
LABELDEF =
LABELPAT<3> = LABELDEF
LABELPAT<2> = LABELDEF
LABELDEF =
*
TEST FOR COLUMN SPECIFICATION CONTAINING ONLY DIGITS
TESTLABCOL LABELPAT<2> SPAN('0123456789') RPOS(0) :F(LABERROR)
SCANLABELDEF = :F(RETURN)
ROUT0 LAOLAPAT<2> 'COL' LABELNAME = SUBSTR(LABELNAME,1,SIZE(LABELNAME) - 1)
ROUT01 STRING = CRRNULL(LABELNAME)
*
ROUTINE FOR LABEL PROCESSING -- NO PARAMETERS
*
ROUTINE FOR UNIDENTIFIED-TO-END-OF-STATEMENT AND MACRO CALL
PROCESSING -- NO PARAMETERS
*
DETERMINE IF THIS UEOS WAS A MACRO CALL
*
CURSTMT SPAN(' ') MACNAMEPAT : SELMAC ('';
BREAK('')'; PARMLIST');
*
CURSTMT MACNAMEPAT ; SELMAC (''; BREAK('')'; PARMLIST
* ;
MACROCALL I = 0
*
PROCESS A MACRO CALL
*

**BREAK OUT THE ACTUAL ARGUMENTS**

**BREAKPARMS I = I + 1**

**PARMLIST BREAK(';', ')'), CPARM:<I>,';' : S(BREAKPARMS)**

**PARMLIST REM . CPARM:<I> : F(MACCALLPERR)**

**ACTARG = I**

**RETRIEVE THE MACRO PATTERN AND NUMBER OF COMPONENTS INTO MPAT**

**MPAT = COPY(MACROTAB<SELMAC>)**

**MPARM = COPY(MACROTAB<SELMAC 'P'>)**

**I = MPARM<0>**

**START THE PARAMETER PATTERN WITH A SERIES OF 81 BLANKS**

**PARM = DUPL(' ', 81)**

**ADD THE PARAMETERS TO THE PARAMETER PATTERN**

**MACPARMMPAT PARM = PARM | MPARM:<I>**

**I = GT(I, 1) | - 1**

**MACSTR = S(MACPARMMPAT)**

**COPY THE STATEMENT COMPONENTS AND REPLACE THE PARAMETERS WITH THE ARGUMENTS**

**I = 0**

**STMTLEN = MPAT<0>**

**LE(STMTLEN, 0) : S(ROUTEOLERR)**

**MACLOOP LT(I, STMTLEN) : F(FAOONE)**

**I = I + 1**

**MPAT:<I> PARM . SELECTPAM SPAN(' ', ')') RPOS(0)**

**+ MPAT:<I> PARM . SELECTPAM RPOS(0) : S(COPYMACPARM)**

**J = 0**

**INCR.J J = LT(J, ACTARG) J + 1**

**SELECTPAM MPARM:<J> : F(MACARGERR)**

**MACSTR = MACSTR CPARM:<J> : (MACLOOP)**

**COPYMACPARM J = 0**

**INCR2.J J = LT(J, ACTARG) J + 1**

**SELECTPAM MPARM:<J> : F(MACAPGERR)**

**MACSTR = MACSTR CPARM:<J> DUPL(' ', SIZE(MPAT:<I>), 0)**

**LT(SIZE(MPAT:<I>)) = SIZE(CPARM:<J>), 0) : S(MACBLKERR)**

**MACSTR = MACSTR CPARM:<J> DUPL(' ', SIZE(MPAT:<I>), 0)**

**LT(SIZE(CPARM:<J>))** : (MACLOOP)
COPYHAL
MACSTR = MACSTR MPAT<i> >

*: (MACLOOP)

* THESE ARE AN UNIDENTIFIED-TO-END-OF-STATEMENT

* NOT A MACRO CALL; INSERT THE CURSTMT; NULLIFY CURSTMT AND RETURN

COPYUEOS
INSERTSTMT(CURSTMT)
&ANCHOR = 0
CURSTMT = :
{RETURN}
MACDNE
INSERTCOM(CURSTMT)

* IS A MACRO CALL; INSERT THE CURSTMT AS A COMMENT; NULLIFY
* THE CURSTMT; INSERT THE MACRO REPLACEMENT TEXT AS ONE OR
* MORE STATEMENTS AND RETURN

CURSTMT =
INSERT(MACSTR)
&ANCHOR = 0
MACSTR = :
{RETURN}

ROUT1
&ANCHOR = 1

* ROUTINE FOR ELSE PROCESSING

* ERROR IF NOT IN CORRECT FORM FOR ELSE

INSTR = CURSTMT
INSTR 'SPAN(" ");
INSTR 'ELSE' = :
&ANCHOR = 0
CURSTMT 'ELSE' = 1;
&ANCHOR = 1

* POP THE LABEL TO BE RPNANCED TO ON FALSE FROM THE SEMANTIC STACK;
* CREATE THE LABEL TO BE PLACED AT THE END OF THE IF STRUCTURE;
* CREATE A STRING OF A GOTO STATEMENT AND A NULL STATEMENT

GEN1
POP_SEM(1,"VAR1")
VAR2 = UNILABEL()
STRING = LGOTO(UNILABEL(),VAR2)
CRNULL(VAR1)

PRE24110
PRE24120
PRE24121
PRE24122
PRE24123
PRE24124
PRE24125
PRE24126
PRE24127
PRE24130
PRE24140
PRE24150
PRE24160
PRE24170
PRE24180
PRE24190
PRE24200
PRE24210
PRE24220
PRE24230
PRE24240
PRE24250
PRE24260
PRE24270
PRE24280
PRE24290
PRE24291
PRE24300
PRE24310
PRE24320
PRE24330
PRE24340
PRE24350
PRE24360
PRE24370
PRE24380
PRE24390
PRE24400
PRE24410
PRE24420
PRE24430
PRE24440
PRE24450
PRE24460
PRE24470
PRE24480
INSERTCOM('ELSE KEYWORD PROCESSED.')

** INSERT THE STRING 
** INSERT (STRING)

** PUSH THE LABEL CREATED IN THIS ROUTINE ONTO THE SEMANTIC STACK 
** PUSH. SEM(1,VAR2)
 &ANCHOR = 0
 &ANCHOR = 1 : (RETURN)

ROUT2

** ROUTINE FOR CASE STATEMENT PROCESSING 
** ERROR IF NOT IN CORRECT FORM FOR CASE 
** INSTR = CURSTM T
 INSTR SPAN(' ') =
** BREAK OUT THE INDEX OF THE CASE 
** INSTR 'CASE OF ' BREAK(SEMDEL) . IND = :F(CASEERROR)
** CREATE 2 LABELS; CREATE A STRING OF AN IF STATEMENT; SET KTR TO 1

GEN2

VAR0 = UNILABEL()
VAR1 = UNILABEL()
INSTRUCT = INSTRUCT + 1
NO. OF CASES = NO. OF CASES + 1
\$('KTR': NO. OF CASES) = 1
STRING = $IF(UNILABEL(), INDUCT, \$('KTR': NO. OF CASES), VAR1)

** INSERT THE CURSTM T AS A COMMENT; NULLIFY CURSTM T 
** INSERTCOM(CURSTM T)
 CURSTM T =

** INSERT THE CREATED STRING; PUSH THE INDEX OF THE CASE AND THE 2 CREATED LABELS ONTO THE SEMANTIC STACK 
** INSERT (STRING)
 PUSH. SEM(3, IND, VAR1, VAR0)
 &ANCHOR = 0
 &ANCHOR = 1 : (RETURN)

ROUT3
ROUTINE FOR DO LOOP STATEMENT PROCESSING

ALL ERRORS IF NOT IN CORRECT FORM FOR DO LOOP

INSTR = CURSTMT
INSTR SPAN(' ', ') =

BREAK OUT THE INDEX, BEGINNING VALUE, ENDING VALUE, AND INCREMENT
VALUE OF THE DO LOOP

INSTR 'DO' 'BREAK(' ' ') ; IND ' = ' = :F(DDERROR)
INSTR ARB . BEGIN ' TO ' = :F(DDERROR)
INSTR ARB . END ' BY ' = :S(INCR)
INSTR BREAK(SEMDEL) . END = :F(DDERROR)
INC = 1
INSTR BREAK(SEMDEL) . INC = :F(DDERROR)

CREATE 1 LABEL AND 1 VARIABLE; CREATE A STRING OF 2 ASSIGN MINUS
STATEMENTS AND AN ASSIGN PLUS STATEMENT

GEN3
VAR1 = UNILABEL()
VAR2 = UNIVAR()
INSTRUCT = INSTRUCT + 1
STRING = CDMINUS(UNILABEL(), VAR2, END, INC)
        CDMINUS(UNILABEL(), IND, BEGIN, INC)
        CRPLUS(VAR1, IND, INC, INC)

INSERT THE CURSTMT AS A COMMENT; NULLIFY THE CURSTMT;
INSERT THE CREATED STRING

INSERTCCM(CURSTMT)
CURSTMT = INSERT(STRING)

PUSH THE INCREMENT, INDEX, AND VARIABLE AND LABEL CREATED IN
THESE ROUTINE INTO THE SEMANTIC STACK

PUSH .SEM(4, INC, IND, VAR2, VAR1)
&ANCHOR = 0
&ANCHOR = 1

RETURN
ROUTINE FOR DOGROUP STATEMENT PROCESSING

ERROR IF NOT IN CORRECT FORM FOR DOGROUP

INSTR = CURSTM1
INSTR SPAN('') =
INSTR 'DOGROUP' ANY(SEMDEL) = :F(DOGRPEMERROR)

INSERT CURSTM AS A COMMENT; NULLIFY CURSTM; RETURN

GEN4
INERTCOM(CURSTM1)
INACRUCT = INACTCRUCT + 1
CURSTM =
&ANCHOR = 0
(SELECT)
&ANCHOR = 1

ROUT5

ROUTINE FOR IF THEN STATEMENT PROCESSING

ERROR IF NOT IN CORRECT FORM FOR IF THEN

INSTR = CURSTM1
INSTR SPAN('') =
BREAK OUT THE CONDITION

INSTR 'IF ' ARB ' BOOL ' THEN ' = :F(IFERROR)

CREATE A LABEL FOR BRANCHING TO ON FALSE; CREATE A STRING
OF AN IF STATEMENT

GEN5
V派 = UNILABEL()
STRING = CRIP(UNILABEL(),NULL,NOT,(' BOOL '),V派)

INSERT CURSTM AS A COMMENT; NULLIFY CURSTM

INERTCOM(CURSTM1)
CURSTM =
INSERT THE STRING; PUSH THE LABEL CREATED IN THIS ROUTINE
INTO THE SEMANTIC STACK; RETURN

INSERT(STRING)
PUSH SEM(1, VAR1)
&ANCHOR = 0
&ANCHOR = 1
: (RETURN)

ROUT6
*
******************************************************************************
* ROUTINE FOR REPEAT STATEMENT PROCESSING
******************************************************************************
* ERROR IF NOT IN CORRECT FORM FOR REPEAT
* INSTR = CURSTM
* INSTR SPAN(1) =
* INSTR * REPEAT UNTIL ( =
* : F(REPPROR)
* BREAK CUT THE CONDITION
* INSTR ARR * BOOL ']' ANY(SEMD) =
* : F(REPPROR)
* CREATE A LABEL FOR A NULL STATEMENT; CREATE A STRING OF A NULL
* STATEMENT
* GEN6
* VAR1 = UNILABEL()
* INASTRUCT = INASTRUCT + 1
* STRING = CRNLLL(VAR1)
* INSRT CURSTM AS A COMMENT; NULLIFY CURSTM
* INSRTCOM(CURSTM)
* CURSTM =
* INSERT THE STRING; PUSH THE LABEL CREATED IN THIS ROUTINE AND THE
* CONDITION ONTO THE SEMANTIC STACK; RETURN
* INSERT STRING
* PUSH SEM(2, VAR1, BOOL)
* &ANCHOR = 0
* &ANCHOR = 1
* : (RETURN)

ROUT7
*
******************************************************************************
* ROUTINE FOR WHILE STATEMENT PROCESSING
******************************************************************************
* ERROR IF NOT IN CORRECT FCRM FCR WHILE
*
INSTR = CURSTM
INSTR SPAN(' ') =
INSTR 'WHILE (' =
*:F(WHERROR)

BREAK OUT THE CONDITION
INSTR ARB . BOOL '(' . ANY(SEMDEL) =
*:F(WHERROR)

CREATE 2 LABELS; CREATE A STRING OF AN IF STATEMENT

GEN7
VAR1 = UNILABEL()
VAR2 = UNILABEL()
INSTRUCT = INSTRUCT + 1
STRING = CRIF(VAR1, NULL, NOT, '(', 'BOOL '), VAR2)

INSERT CURSTM AS A COMMENT; NULLIFY CURSTM

INSERTCOM(CURSTM)

CURSTM =

INSERT THE STRING; PUSH THE 2 LABELS CREATED IN THIS
ROUTINE INTO THE SEMANTIC STACK; RETURN

INSERT(STRING)
PUSH(SEM1, VAR2, VAR1)
&ANCHOR = 0
&ANCHOR = 1
*:RETURN

ROUT8

_ROUTINE FOR QUITLOOP STATEMENT PROCESSING

_ERROR IF NOT IN CORRECT FCR FOR QUITLOOP

INSTR = CURSTM
INSTR SPAN(' ') =
INSTRUCT 'QUITLOOP' ANY(SEMDEL) =
*:F(QUITERROR)

CREATE A LABEL FOR BRANCHING OUT OF THE LOOP; CREATE A
STRING OF A GO TO STATEMENT

GFIN8
VAR1 = UNILABEL()
STRING = CRGOTO(UNILABEL(), VAR1)

INSERT CURSTM AS A COMMENT; NULLIFY CURSTM

PRE26170
PRE26180
PRE26190
PRE26200
PRE26210
PRE26220
PRE26230
PRE26240
PRE26250
PRE26260
PRE26270
PRE26280
PRE26290
PRE26300
PRE26310
PRE26320
PRE26330
PRE26340
PRE26350
PRE26360
PRE26370
PRE26380
PRE26390
PRE26400
PRE26410
PRE26420
PRE26430
PRE26440
PRE26441
PRE26442
PRE26450
PRE26460
PRE26461
PRE26462
PRE26470
PRE26480
PRE26490
PRE26500
PRE26510
PRE26520
PRE26530
PRE26540
PRE26550
PRE26560
PRE26570
PRE26580
PRE26590
PRE26600
**ROUTINE FOR ENDCALL PROCESSING**

**ERROR IF NOT IN CORRECT FORM FOR ENDCALL**

INSTR = CURSTM
INSTR 'ENDCALL' ANY(SEMDEL) = :IF(ECASEERROR)

POP 2 LABELS AND THE INDEX OF THE CASE OFF THE SEMANTIC STACK

GEN9

CREATE A STRING OF 2 NULL STATEMENTS; RESET KTR TO 0

STRING = CRNULL(VAR1) CRNULL(VAR0)
+ ('KTR', NO. OF CASES) = 0
+ NO. OF CASES = NO. OF CASES - 1
+ INSTRUCT = INSTRUCT - 1

INSEPT CURSTM AS A COMMENT; NULLIFY CURSTM

INSEPTCOM(CURSTM)
CURSTM =

INSEPT THE STRING; RETURN

INSEPT(STRING)
&ANCHOR = 0
CURSTM = '
CURSTM =

**ROUTINE FOR ENDFR PROCESSING**
* ERROR IF NOT IN CORRECT FORM FOR ENDFMT
* INSTR = CURSTMFT
* INSTR SPAN(' ') =
* INSTR 'ENDIF' ANY(SEDHEM) = :F(EIFERROR)
* POP AN ITEM OFF THE SEMANTIC STACK; BRANCH IF IT ISN'T 'QUIT'
* GEN10 POP SEM(1, 'VAR1')
* VAR1 'QUIT' RPOS(0) :F(NotaQUIT)
* THE ITEM POPPED WAS 'QUIT', SO POP 2 MORE ITEMS OFF AND PUSH THE
* 'QUIT' AND FOLLOWING ITEM BACK ONTO THE STACK, LEAVING A
* LABEL AVAILABLE
* POP SEM(2, 'VAR2', 'VAR1')
* PUSH SEM(2, 'VAR2', 'QUIT')
* CREATE A STRING OF A NULL STATEMENT
* NOTAQUIT STRING = CRNULL(VAR1)
* INSERT CURSTMFT AS A COMMENT; NULLIFY CURSTMFT
* INSERTCOM(CURSTMFT)
* CURSTMFT =
* INSERT THE STRING; RETURN
* INSERT(STRING)
* GANCHOR = 0
* GANCHOR = 1 :(RETURN)

** ROUTINE FOR ENDFMT PROCESSING**

** ERROR IF NOT IN CORRECT FORM FOR ENDFMT**
* INSTR = CURSTMFT
* INSTR SPAN(' ') =
* INSTR 'ENDFMT' ANY(SEDHEM) = :F(EWHILEERROR)
SET QUITFLAG TO 0; POP 2 ITEMS OFF THE SEMANTIC STACK;
  BRANCH IF THE FIRST ITEM ISN'T 'QUIT'
GEN11
  QUITFLAG = 0
  POP,SEM(2,'VAR1','VAR2')
  INSTRUCT = INSTRUCT - 1
  IF QUIT* POP(0)
       :F(ELSE11)
  THE FIRST ITEM WAS 'QUIT', SO ASSIGN THE SECOND ITEM TO QUITLAB
       AND SET QUITFLAG TO 1
       QUITLAB = VAR2
       QUITFLAG = 1
  POP 2 LABELS OFF THE SEMANTIC STACK
  PCP,SEM(2,'VAR1','VAR2')
  CREATE A STRING OF A GO TO STATEMENT AND A NULL STATEMENT
ELSE11
  STRING = CRGOTO(UNILABEL(),VAR1) CRNULL(VAR2)
  INSERT CURSTM AS A COMMENT; NULLIFY CURSTM
  INSERTCOM(CURSTM)
  CURSTM =
  INSERT THE STRING; RETURN IF QUITFLAG IS 0
       INSERT STRING
       NL(QUITFLAG,1)
       &ANCHOR = 0
       :F(QUITINWHILE)
       :F(RETURN)
  QUITFLAG IS 1, CREATE A STRING OF A NULL STATEMENT; INSERT
  THE STRING; RETURN
QUITINWHILE
  STRING = CRNULL(QUITLAB)
  INSERT STRING
  &ANCHOR = 0
  :F(RETURN)
ROUT12
  &ANCHOR = 1

*****************************************************************************
* ROUTINE FOR ENDREPEAT PROCESSING
*****************************************************************************
* ERROR IF NOT IN CORRECT FORM FOR ENDREPEAT
* INSTR = CURSTMT
* INSTR SPAN(, ) = ANY(SEMDEL) = :F(EREERROR)
* SET QUITFLAG TO 0; POP 2 ITEMS OFF THE SEMANTIC STACK; BRANCH
* IF THE FIRST ITEM ISN'T 'QUIT'
* GEN12
* QUITFLAG = 0
* PCP.SEM(2,'BCOL','VARI')
* INASTRUCT = INASTRUCT - 1
* BOOL 'QUIT' RPOS(0) :F(ELSE12)
* THE FIRST ITEM WAS 'QUIT', SO ASSIGN THE SECOND ITEM TO QUITLAB
* AND SET QUITFLAG TO 1
* QUITLAB = VARI
* QUITFLAG = 1
* POP A CONDITION AND A LABEL OFF THE SEMANTIC STACK; CREATE A
* STRING OF AN IF STATEMENT
* PCP.SEM(2,'BCOL','VARI')
* STRING = CRIF(UNILABEL(),NULL,NOT,' ','BOOL',')',VARI)
* ELSE12
* INSERT CURSTMT AS A COMMENT; NULLIFY CURSTMT
* INSERTCOM(CURSTMT)
* CURSTMT =
* INSERT THE STRING; RETURN IF QUITFLAG IS 0
* INSERT(STRING)
* NE(QUITFLAG,1)
* &ANCHOR = 0 :F(QUITINREP)
* (RETURN)
* QUITFLAG IS 1; CREATE A STRING OF A NULL STATEMENT; INSERT THE
* STRING; RETURN
* QUITINREP
* STRING = CRNNULL(QUITLAB)
* INSERT(STRING)
* &ANCHOR = 0 :F(RETURN)
* ROUT13
* &ANCHOR = 1
* ROUTINE FOR ENDNO PROCESSING
**ERROR IF NOT IN CORRECT FORM FOR ENDDO**

```
INSTR = CURSTM
INSTR SPA('') =
INSTR 'ENDDO' ANY(SEMDEL) = :F(EDERROR)
```

**SET QUITFLAG TO 0; POP 4 ITEMS OFF THE SEMANTIC STACK; BRANCH IF THE FIRST ITEM ISN'T 'QUIT'**

```
QUITFLAG = 0
POP, SFM(4,'VAR1','VAR2','IND','INC')
INASTRUCT = INASTRUCT - 1
VAR1 'QUIT' KPOS(0) :FELSE13
```

**THE FIRST ITEM WAS 'QUIT', SO ASSIGN THE SECOND ITEM TO QUITLAB**

```
QUITLAB = VAR2
```

**ASSIGN THE THIRD AND FOURTH ITEMS TO VAR1 AND VAR2**

```
VAR1 = IND
VAR2 = INC
```

**POP 2 MORE ITEMS OFF THE SEMANTIC STACK AND SET QUITFLAG TO 1**

```
POP, SEM(2,'IND','INC')
QUITFLAG = 1
```

**CREATE 2 LABELS: CREATE A STRING OF 2 IF STATEMENTS, A GO TO STATEMENT, AN IF STATEMENT, AND AN ASSIGN PLUS STATEMENT**

```
ELSE13
VAR3 = UNILABEL()
VAR4 = UNILABEL()
STRING = CRIF(UNILABEL(),INC,GTHAN,'O',VAR3)
CRIF(UNILABEL(),IND,GEQU,VAR2,VAR1)
CRGTO(UNILABEL(),VAR4)
CRIF(VAR3,IND,LEQU,VAR2,VAR1)
CRPLUS(VAR4,IND,IND,INC)
```

**INSERT CURSTM AS A COMMENT; NULLIFY CURSTM**

```
INSERTCOM(CURSTM)
CURSTM =
```

**INSERT STRING; RETURN IF QUITFLAG IS 0**
* INSERT (STRING)
  NE (QUITFLAG, 1)
  $ANCHOR = 0
  :(QUITINDO)  :(RETURN)
* QUITINDO
  STRING = CNUL(QUITLAB)
  INSERT (STRING)
  $ANCHOR = 0
  :(RETURN)
ROUT14
  $ANCHOR = 1
* ROUTINE FOR ENDDGRUP PROCESSING
*---------------------------------------------------------------------------*
* ERROR IF NOT IN CORRECT FORM FOR ENDDGROUP
*---------------------------------------------------------------------------*
  INSTR = CURSTMT
  INSTR SPAN [" "] =
  INSTR 'ENDDGROUP' ANY(SEMLD) =
  :(F(NDOGRPERROR)
* INSERT CURSTMT AS A COMMENT; NULLIFY CURSTMT; RETURN
*---------------------------------------------------------------------------*
  INSERTCOM(CURSTMT)
  INASTRUCT = INASTRUCT - 1
  CURSTMT =
  $ANCHOR = 0
  :(RETURN)
ROUT15
  $ANCHOR = 1
* ROUTINE FOR STRUC OR STRLIST STRUC TO STRLIST REDUCTION
*---------------------------------------------------------------------------*
  RETURN IF KTR = 0 (NOT WITHIN A CASE STRUCTURE) OR IF
  INASTRUCT > NO. OF CASES (WITHIN A 'PRIMITIVE' STRUCTURE);
  OTHERWISE, INCREMENT KTR BY 1
*---------------------------------------------------------------------------*
GEN15
  GT(*KTR, NO. OF CASES), 0
  $ANCHOR = 0
  :(NACASE)
  :(RETURN)
INACASE
  GT(INASTRUCT, NO. OF CASES)
  $ANCHOR = 0
  :(AFTERSTRUC)
AFTERSTRUC
  $(*KTR, NO. OF CASES) = $(*KTR, NO. OF CASES) + 1
* CREATE A LABEL; POP 3 ITEMS OFF THE SEMANTIC STACK
* VAR3 = UNILABEL()
* POP(SEM(3,'VAR0','VAR1','IND'))
* CREATE A STRING OF A GO TO STATEMENT AND IF AN STATEMENT
* STRING = CRGOTO(UNILABEL(),VAR0)
* CRIP(VAR1,IND,NEQU,$KTR' NO.OF.CASES),VAR3
* INSERT A COMMENT ABOUT THIS ROUTINE'S RESULTS
* INSERTCOM('NEXT TWO STATEMENTS INSERTED FOR CASE PROCESSING')
* INSERT THE STRING; PUSH THE INDEX OF THE CASE AND THE 2
* LABELS CREATED IN THIS ROUTINE ONTO THE SEMANTIC STACK;
* RETURN
* INSERT(STRING)
* PUSH(SEM(3,IND,VAR3,VAR0)
* &ANCHOR = 0 : (RETURN)
INSERTCOM I = 1
* ROUTINE TO INSERT A STRING AS A COMMENT
* ONE PARAMETER -- CCMSTR, THE STRING TO BE MADE INTO A
* COMMENT AND INSERTED
* FIND THE FIRST AVAILABLE COLUMN FOR INSERTING THE COMMENT
* IGNPAT<0> 'SEQ'
* I = FQ(I,IGNPAT<1>) I + 1 : S(COLSEQ)
* COLSEQ GT(I,IGNPAT<2>)
* [T(I,IGNPAT<2>) : S(FOUNDCOLOPEN)
* INC.R.1 I = LE(I,IGNPAT<2>) I + 1 : S(INCR.1)
* FILL THE COLUMNS PRECEDING THE COLUMN FOUND WITH BLANKS,
* BRANCH BASED ON THE COMMENT DEFINITION
* FOUNDCOLOPEN OUTCOM = DUPL(' ',I - I)
* COMPAT<1> '10'
* A COMMENT BEGINS IN A CERTAIN COLUMN, FILL WITH BLANKS TO THAT
COLUMN

I = COMPAT<4> - I
OUTCOM = OUTCOM DUPL( ' ', I)
CCMPAT<1> '11' :F(ADDCOMSTR)

A COMMENT BEGINS WITH A CERTAIN STRING IN THAT PARTICULAR COLUMN, ADD THE STRING TO THE OUTPUT COMMENT STRING

ADDOPENSTR OUTCOM = OUTCOM COMPAT<3>

ADD THE PARAMETER STRING TO THE OUTPUT COMMENT STRING,
CHECK FOR THE OUTPUT STRING BEING LONGER THAN 80 CHARACTERS

ADDCOMSTR OUTCOM = OUTCOM COMSTR
GT(SIZE(OUTCOM),80) :F(NOTRUNCATE)

TRUNCATE TO 80 CHARACTERS

OUTCOM = SUBSTR(OUTCOM,1,80)
NOTRUNCATE CCMPAT<2> '01' :F(ENDCOMSTR)

A COMMENT ENDS IN A CERTAIN COLUMN, ADD BLANKS IF THE COMMENT IS TOO SHORT, TRUNCATE IF IT IS TOO LONG, BRANCH TO THE COMMENT FINISHED LABEL

GT(SIZE(OUTCOM),COMPAT<6>) :S(COLTRUNC)
OUTCOM = OUTCOM DUPL( ' ', 80 - SIZE(OUTCOM)) :F(COMFIN)

COLTRUNC OUTCOM = SUBSTR(OUTCOM,1,CCMPAT<6>) :F(COMFIN)
ENDCOMSTR COMPAT<2> '11' :S(COMENDSTRCOL)

A COMMENT ENDS WITH A CERTAIN STRING, IF THE STRING WON'T FIT ONTO THE OUTPUT STRING AND STILL HAVE THE STRING BE 80 CHARACTERS OR LESS, TRUNCATE THE CURRENT OUTPUT STRING

GT(SIZE(OUTCOM), 80 - SIZE(COMPAT<5>)) :F(NOSYMTRUNC)
OUTCOM = SUBSTR(OUTCOM,1,80 - SIZE(COMPAT<5>))

ADD THE TERMINATING STRING, SET I TO THE COLUMN NUMBER THE STRING MUST START IN TO END EXACTLY IN COLUMN 80, BRANCH ACCORDING TO THE IGNORE COLUMN DEFINITION SPECIFYING A SINGLE COLUMN OR A SEQUENCE OF COLUMNS

NOSYMTRUNC (OUTCOM = OUTCOM COMPAT<5>
I = 81 - SIZE(COMPAT<5>)
IGNPAT<> 'SEQ' :S(COLOSEQEND)

DECREMENT I IF THE STRING WOULD END IN AN IGNORE COLUMN AS IT
* IS AND BRANCH
* I = EQ(I, IGNPAT<1>) I - SIZE(COMPAT<5>) : S(FOUNDCOLCLOSE)
* DECREMENT I UNTIL THE STRING IS SURE TO END IN A NON-IGNORED
* COLUMN
* COLSEQEND GT(I, IGNPAT<2>) : S(FOUNDCOLCLOSE)
* (T(I + SIZE(COMPAT<5>) - 1, IGNPAT<1>) : S(FOUNDCOLCLOSE)
* DECR. I = GE(I + SIZE(COMPAT<5>) - 1, IGNPAT<1>) I - 1
* ADD THE COMMENT TERMINATING STRING TO THE OUTPUT STRING AND
* ADD BLANKS AS NECESSARY TO FILL 80 COLUMNS
* FOUNDCOLCLOSE OUTCOM = SUBSTR(OUTCOM, I, I - 1) COMPAT<5>
* + DUPL(' ', 80 - SIZE(OUTCOM)) : COMIN
* COMENDSTRCOL GT(SIZE(OUTCOM), COMPAT<6>) : S(COLSYMTRUNC)
* OUTCOM = OUTCOM DUPL(' ', COMPAT<6> - 1 - SIZE(OUTCOM))
* + COLSYMTRUNC OUTCOM = SUBSTR(OUTCOM, 1, COMPAT<6> - 1)
* ADDCOLSYM OUTCOM = OUTCOM COMPAT<5>
* + DUPL(' ', 80 - SIZE(OUTCOM) - SIZE(COMPAT<5>))
* PRINT AND PUNCH THE COMMENT AND RETURN
* COMIN OUTPUT = OUTCOM
* PUNCH = OUTCOM
* INCMPCTOM = (: RETURN)
* INSERTSTM OUTSTR =
* ROUTINE TO INSERT A STRING AS A STATEMENT -- IGNORED COLUMNS
* MUST BE SKIPPED
* UNF PARAMETER -- STRINSTR, THE STRING TO BE INSERTED
* AS A STATEMENT
* INSERTSTM =
* I = 0
* CHAKTR = 0
* SET TYPE APPROPRIATELY BASED ON A SINGLE COLUMN OR A
* SEQUENCE OF COLUMNS TO BE IGNORED
* INSERTSTM
IGNPAT<0> 'SEQ'
TYPE = 'SEQ'

SINGLEIGN TYPE = 'ONE'

* CHARKTR COUNTS THE POSITION IN THE PARAMETER STRING
* INCREMENT CHARKTR = CHARKTR + 1
* I COUNTS THE POSITION IN THE OUTPUT STRING
* INCREMENT2 I = I + 1
  LE(CHARKTR, SIZE(STR.INSRT))
  LE(I,80)

* BRANCH TO CHECK FOR IGNORING COLUMNS

* CHECK FOR IGNORING THIS COLUMN WHEN ONLY ONE COLUMN IS IGNORED
  CKIGNONE NE(I,IGNPAT<1>)

* CHECK FOR IGNORING THIS COLUMN WHEN A SEQUENCE OF COLUMNS IS IGNORED
  CKIGNSEQ GT(I,IGNPAT<2>)
  LT(I,IGNPAT<1>)

* ADD A BLANK TO THE OUTPUT STRING IN AN IGNORED COLUMN AND BRANCH TO PROCESS THE NEXT CHARACTER
  IGNOREIT OUTSTR = OUTSTR ' ' :F(INCREMENT2)

* ADD THE CURRENT CHARACTER TO THE OUTPUT STRING AND BRANCH TO PROCESS THE NEXT CHARACTER
  GOODCOL OUTSTR = OUTSTR SUBSTR(STR.INSRT,CHARKTR,1) :F(INCREMENT)

* OUTPUT CARD IS FULL, PUNCH AND PRINT IT, REINITIALIZE COLUMN Counter AND OUTPUT STRING
  ENDCARD OUTPUT = OUTSTR
  PUNCH = OUTSTR
  I = 1
  OUTSTR =

* END OF PARAMETER STRING, ADJUST ENOUGH BLANKS TO FILL A CARD, PUNCH AND PRINT IT, AND RETURN
**ENDSTMTOUT**
OUTSTR = OUTSTR DUPL(' ',80 - SIZE(OUTSTR))
OUTPUT = OUTSTR
PUNCH = OUTSTR
:((RETURN)
**INSERT**
PUNCH = INSRAT.STR

**ROUTINE TO INSERT A STRING WHICH IS ALREADY IN THE FORM OF**
**A STATEMENT**

**ONE PARAMETER -- INSRAT.STR, THE STATEMENT TO BE INSERTED**

**PRINT AND PUNCH THE STATEMENT AND RETURN**
MOREOUT
GT(SIZE(INSRAT.STR),80)
OUTPUT = INSRAT.STR DUPL(' ',80 - SIZE(INSRAT.STR))
:((RETURN)
OUTAO
OUTPUT = SUBSTR(INSRAT.STR,1,80)
INSRAT.STR = SUBSTR(INSRAT.STR,81,SIZE(INSRAT.STR) - 80)
:((MOREOUT)

**CRIF**
CRIF =

**ROUTINE TO CREATE AN IF STATEMENT**

**FIVE PARAMETERS -- LAB, VAL1, RL, VAL2, AND DES**

PARNAME = 'LAB' | 'VAL1' | 'RL' | 'VAL2' | 'DES'
I = 0
STMTLEN = IFPAT<0>

**ERROR IF NUMBER OF STATEMENT COMPONENTS IS < 0**

CRIFLOOP
LE(STMTLEN,0)
LT(I,STMTLEN)
I = I + 1
IFPAT<1> PARNAME . SELECTPARM SPAN(' ') RPOS(0)
:((IFPAT<1> PARNAME . SELECTPARM RPOS(0) :((COPYCRIF)
CRIF = CRIF $SELECTPARM
:((CRIFLOOP)

**ERROR IF NOT ENOUGH BLANKS LEFT AFTER PARAMETER NAME IN PROTOTYPE**

PRE31050
PRE31060
PRE31070
PRE31080
PRE31090
PRE31100
PRE31110
PRE31120
PRE31130
PRE31140
PRE31150
PRE31160
PRE31170
PRE31180
PRE31190
PRE31200
PRE31210
PRE31220
PRE31230
PRE31240
PRE31250
PRE31260
PRE31270
PRE31280
PRE31290
PRE31291
PRE31292
PRE31293
PRE31294
PRE31295
PRE31296
PRE31297
PRE31298
PRE31299
PRE31300
PRE31310
PRE31320
PRE31330
PRE31340
PRE31350
PRE31360
PRE31370
PRE31380
PRE31390
PRE31400
PRE31410
PRE31420
PRE31430
PRE31440
COPYIFP ARM LT(SIZE(IFPAT<1>) - SIZE($SELECTPARM), 0) : S(IFBLKERR) PRE31450
* COPY PARAMETER AND ENOUGH BLANKS TO EQual FIELD LENGTH IN PROTOTYPE PRE31470
* CRIF = CRIF $SELECTPARM DUPL(' ', SIZE(IFPAT<1>) - SIZE($SELECTPARM)) : (CRIFLOOP) PRE31480
COPYCRIF CRIF = CRIF IFPAT<1> PRE31490
* ADJ BLANKS TO MAKE 80 COLUMNS, ERROR IF MORE THAN 80 COLUMNS PRE31500
* ALREADY PRE31510
IFRETSEQ CRIF = CRIF DUPL(' ', 80 - SIZE(CRIF)) : S(RETURN)F(CRIFERR) PRE31520
CRDCL CRDCL = PRE31530
* ROUTINE TO CREATE DECLARATION STATEMENT PRE31540
* ONE PARAMETER -- VARNAME PRE31550
* I = 0 PRE31560
STMTLEN = DCLPAT<0> PRE31570
* ERROR IF NUMBER OF STATEMENT COMPONENTS IS <= 0 PRE31580
PRE31590
CRDCLLOOP LT(I, STMTLEN) : S(CRDCLERR) PRE31591
F(DCLRETSEQ) PRE31600
I = I + 1 PRE31610
DCLPAT<1> 'VARNAME' SPAN(' ') RPOS(0) : S(COPYDCLPARAM) PRE31620
DCLPAT<1> 'VARNAME' RPOS(0) : F(COPYCRDCL) PRE31630
CRDCL = CRDCL VARNAME : (CRDCLLOOP) PRE31640
* ERROR IF NOT ENOUGH BLANKS LEFT AFTER PARAMETER NAME IN PROTOTYPE PRE31650
* PRE31660
PRE31670
PRE31680
COPYDCLPARAM LT(SIZE(DCLPAT<1>) - SIZE(VARNAME), 0) : S(DCLBLKERR) PRE31690
* COPY PARAMETER AND ENOUGH BLANKS TO EQual FIELD LENGTH IN PROTOTYPE PRE31710
* PRE31720
PRE31730
PRE31740
PRE31750
CRDCL = CRDCL VARNAME DUPL(' ', SIZE(DCLPAT<1>) - SIZE(VARNAME)) : (CRDCLLOOP) PRE31760
COPYCRDCL CRDCL = CRDCL DCLPAT<1> PRE31770
* ADJ BLANKS TO MAKE 80 COLUMNS, ERROR IF MORE THAN 80 COLUMNS PRE31780
* ALREADY PRE31790
PRE31800
PRE31810
PRE31820
PRE31830
PRE31840
PRE31850
PRE31860
PRE31870
PRE31880
DCL RETSEQ CRDCL = CRDCL DUPL(' ',80 - SIZE(CRDCL))
+ :S(RETURN)F(CRDCLERR)
CRMINUS CRMINUS =
\* * *
\* " ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
\* ROUTINE TO CREATE AN ASSIGN MINUS STATEMENT
\* FOUR PARAMETERS -- LAB, VAR, VAL1, AND VAL2
\* ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
\* PARMNAME = 'LAB' | 'VAR' | 'VAL1' | 'VAL2'
\* I = 0
\* STMTLEN = MINUSPAT<0>
\* ERROR IF NUMBER OF STATEMENT COMPONENTS IS <= 0
\* ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
\* LE(STMTLEN,0)
\* :S(CRMINERR)
\* CRMINUSLOOP LT(I,STMTLEN)
\* I = I + 1
\* MINUSPAT<I> PARMNAME : SELECTPARM SPAN[' ' ] RPOS<0>
\* :S(COPYMINPARM)
\* MINUSPAT<I> PARMNAME : SELECTPARM RPOS<0>
\* :F(COPYCRMINUS)
\* CRMINUS = CRMINUS $SELECTPARM
\* :F(CRMINUSLOOP)
\* ERROR IF NOT ENOUGH BLANKS LEFT AFTER PARAMETER NAME IN PROTOTYPE
\* COPYMINPARM LT(SIZE(MINUSPAT<I>) - SIZE($SELECTPARM),0)
\* :S(MINBLKERR)
\* COPY PARAMETER AND ENOUGH BLANKS TO EQUAL FIELD LENGTH IN PROTOTYPE
\* CRMINUS = CRMINUS $SELECTPARM DUPL(' ',SIZE(MINUSPAT<I>) -
\* SIZE($SELECTPARM))
\* :F(CRMINUSLOOP)
\* COPYCRMINUS CRMINUS = CRMINUS MINUSPAT<I>
\* :F(CRMINUSLOOP)
\* ADD BLANKS TO MAKE 80 COLUMNS, ERROR IF MORE THAN 80 COLUMNS
\* ALREADY
\* MINKRTSFQ CRMINUS = CRMINUS DUPL(' ',80 - SIZE(CRMINUS))
\* :S(RETURN)F(CRMINERR)
\* CRPLUS CRPLUS =
\* ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
\* ROUTINE TO CREATE AN ASSIGN PLUS STATEMENT
\* FOUR PARAMETERS -- LAB, VAR, VAL1, AND VAL2
\* ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
PARNAME = 'LAB' | 'VAR' | 'VAL1' | 'VAL2'
I = 0
STMTLEN = PLUSPAT<0>

*/
ERROR IF NUMBER OF STATEMENT COMPONENTS IS <= 0
*/
LE(STMTLEN,0)
CRPLUSLOOP LT(I,STMTLEN)
I = I + 1
PLUSPAT<1> PARNAME = SELECTPARM SPAN(' ') RPOS(0)
PLUSPAT<1> PARNAME = SELECTPARM RPOS(0)
CRPLUS = CRPLUS $SELECTPARM
CRPLUSLOOP:

*/
ERROR IF NOT ENOUGH BLANKS LEFT AFTER PARAMETER NAME IN PROTOTYPE
COPYPLUSPARM LT(SIZE(PLUSPAT<1>) - SIZE(SELECTPARM),0) : $PLUSBLKERR
COPY PARAMETER AND ENOUGH BLANKS TO EQUAL FIELD LENGTH IN PROTOTYPE
CRPLUS = CRPLUS $SELECTPARM DUPL( ' ',80 - SIZE(CRPLUS))
COPYCRPLUS CRPLUS = CRPLUS PLUSPAT<1> CRPLUSLOOP:

ADD BLANKS TO MAKE 80 COLUMNS, ERROR IF MORE THAN 80 COLUMNS ALREADY
PLUSRETSEQ CRPLUS = CRPLUS DUPL( ' ',80 - SIZE(CRPLUS))
CRNULL
CRNULL =

ROUTING TO CREATE A NULL STATEMENT
ONE PARAMETER -- LAB

I = 0
STMTLEN = NULLPAT<0>

ERROR IF NUMBER OF STATEMENT COMPONENTS IS <= 0
LE(STMTLEN,0): $CRNULLERR
CRNULLLOOP  LT(I,STMTLEN)  :F(NULLRETSEQ)  PRE32730
       I = I + 1  PRE32740
       NULLPAT<i>  'LAB'  SPAN(' ' )  RPOS(O)  :S(COPYNULLPARM)  PRE32750
       NULLPAT<i>  'LAB'  RPOS(O)  :F(COPYCRNULL)  PRE32760
       CRNULL = CRNULL LAB  :C(RNULLLOOP)  PRE32770
* ERROR IF NOT ENOUGH BLANKS LEFT AFTER PARAMETER NAME IN PROTOTYPE
COPYNULLPARM  LT(SIZE(NULLPAT<i>) - SIZE(LAB),0)  :S(NULLBLKERR)  PRE32780
* COPY PARAMETER AND ENOUGH BLANKS TO EQUAL FIELD LENGTH IN PROTOTYPE
* CRNULL = CRNULL LAB DULP(' ',SIZE(NULLPAT<i>) - SIZE(LAB))  :C(RNULLLOOP)  PRE32790
* COPYCRNULL  CRNULL = CRNULL NULLPAT<i>  :C(RNULLLOOP)  PRE32791
* ADD BLANKS TO MAKE 80 COLUMNS, ERROR IF MORE THAN 80 COLUMNS ALREADY
NULLRETSEQ  CRNULL = CRNULL DULP(' ',80 - SIZE(CRNULL))  :S(RETURN)F(CRNULLERR)  PRE32792
CRGOTO  CRGOTO =  PRE32793
*----------------------------------------------------------------------
* ROUTINE TO CREATE A GO TO STATEMENT
* TWO PARAMETERS -- LAB AND DES
*----------------------------------------------------------------------
* PARMNAME = 'LAB' | 'DES'
I = 0  PRE32794
STMTLEN = GOTO PAT<i>  PRE32795
* ERROR IF NUMBER OF STATEMENT COMPONENTS IS <= 0
* LE(STMTLEN,O)  :S(CRGOTOERR)  PRE33050
CRGOTOLOOP  LT(I,STMTLEN)  :F(GOTORETSEQ)  PRE33060
       I = I + 1  PRE33070
       GOTO PAT<i>  PARMNAME  .SELECTPARM SPAN(' ' )  RPOS(O)  :S(COPYGOTOPARM)  PRE33080
       GOTO PAT<i>  PARMNAME  .SELECTPARM RPOS(O)  :F(COPYCRGOTO)  PRE33090
       CRGOTO = CRGOTO  $SELECTPARM  :C(CRGOTOLOOP)  PRE33100
* ERROR IF NOT ENOUGH BLANKS LEFT AFTER PARAMETER NAME IN PROTOTYPE
COPYGOTO PARARM  LT(SIZE(GOTO PAT<i>) - SIZE($SELECTPARM),0)  :S(GOTOBLKERR)  PRE33110
* COPY PARAMETER AND ENOUGH BLANKS TO EQUAL FIELD LENGTH IN PROTOTYPE
* CRGOTO = CRGOTO $SELECTPARM DUPL(' ', SIZE(GOTOPAT<1>) - SIZE($SELECTPARM)) : (CRGOTOLOOP)
* COPYCRGOTO CRGOTO = CRGOTO GOTOPAT<1> : (CRGOTOLOOP)
* ADD BLANKS TO MAKE 80 COLUMNS, ERROR IF MORE THAN 80 COLUMNS ALREADY.
* GOTORETSEQ CRGOTO = CRGOTO DUPL(' ', 80 - SIZE(CRGOTO))
* UNILABEL UNILABEL = LABELPAT<1>
* ********** ROUTINE TO CREATE A UNIQUE LABEL **********
* NO PARAMETERS
* ********** COMPUTE NUMBER OF ZEROS TO IMBED AND ADD THEM TO THE LABEL'S UNIFORM BEGINNING **********
* I = LABELPAT<2> - SIZE(LABELNUM)
* LT(I,0) ADDLABZEROS UNILABEL = GT(I,0) UNILABEL '0'
* I = I - 1 : (ADDLABZEROS)
* ADD THE SIGNIFICANT DIGITS AND THE LABEL ENDING
* ENDLABZEROS UNILABEL = UNILABEL LABELNUM LABELPAT<3>
* INCREMENT THE NUMBER USED TO MAKE THE LABEL UNIQUE
* LABELNUM = LABELNUM + 1 : (RETURN)
* UNIVAR UNIVAR = VARPAT<1>
* ********** ROUTINE TO CREATE A UNIQUE VARIABLE **********
* NO PARAMETERS
* ********** COMPUTE NUMBER OF ZEROS TO IMBED AND ADD THEM TO THE VARIABLE'S
UNIFORM BEGINNING
I = VARPAT<2> - SIZE(VARNUM)
LT(I,0) :S(TODOMANYVARS)
ADDVARZEROS UNIVAR = GT(I,0) UNIVAR 'O'
I = I - 1 :F(ENDDVARZEROS)
ADD THE SIGNIFICANT DIGITS AND THE VARIABLE ENDING
ADD THE NEW VARIABLE TO THE LIST FOR LATER DECLARATION
AND INCREMENT THE INDEX INTO THAT LIST AND THE NUMBER
USED TO MAKE THE VARIABLE UNIQUE
VARIABLELIST<NEVTXT> = UNIVAR
NEXTVAR = NEXTVAR + 1
VARNUM = VARNUM + 1 :{RETURN}

PUP.SFM

ROUTINE TO PUP ONE TO SIX ITEMS FROM THE TOP OF THE
SEMANTIC STACK
TWO TO SEVEN PARAMETERS -- NOARGS, THE NUMBER OF ITEMS TO
BE POPPED; A1 - A6, THE NAMES OF THE VARIABLES (IN STRING
FORM) INTO WHICH THE POPPED VALUES ARE TO BE PLACED
I = 1
POP A VALUE FROM THE STACK AND ASSIGN IT TO A PARAMETER
POP 1(*('A' I)) = SEMSTACK<TOP> :F(EMPTYSTK)
DECREMENT THE POINTER TO THE TOP OF THE STACK
TOP = TOP - 1
IF NOT DONE POPPING, BRANCH TO POP THE NEXT VALUE
I = LT(I,NOARGS) I + 1 :S(PUP)
RETURN WITH A NULL STRING VALUE
PUP.SFM = :{RETURN}
PUSH.SEM

* ***************************************** PRE34020
  * Routine to push one to six items on the top of the 
  * semantic stack
  * Two to seven parameters -- NOARGS, the number of items to be 
  * pushed; A1 - A6, the values to be pushed onto the stack
  * ***************************************** PRE34030

  I = 1
  * Increment the pointer to the top of the stack
  PUSH TCP = TOP + 1
  * Push a value onto the stack
  SEMSTACK<TOP> = $('A' I) :F(TOOFULLSTK)
  * If not done pushing, branch to push the next value
  I = LT(I,NOARGS) I + 1 :S(PUSH)
  * Return with a null string value
  END PUSH.SEM = :
: (RETURN) PRE34040
PRE34050
PRE34060
PRE34070
PRE34080
PRE34090
PRE34100
PRE34110
PRE34120
PRE34130
PRE34140
PRE34150
PRE34160
PRE34170
PRE34180
PRE34190
PRE34200
PRE34210
PRE34220
PRE34230
PRE34240
PRE34250
PRE34260
PRE34270
A STRUCTURED PROGRAMMING PREPROCESSOR

FOR A VARIETY OF BASE LANGUAGES

by

Mary L. Love

B.S., Kansas State University, 1975

AN ABSTRACT OF A MASTER'S REPORT

submitted in partial fulfillment of the requirements for the degree

MASTER OF SCIENCE

Department of Computer Science

KANSAS STATE UNIVERSITY
Manhattan, Kansas

1977
ABSTRACT

This report describes the design and implementation of a preprocessor which is intended to allow and encourage the use of structured programming techniques in a variety of base languages. Eight structured programming control structures are implemented in the preprocessor which currently works with the PL/I, SNOBOL, and FORTRAN programming languages. The extended constructs are translated by the preprocessor into statements which are standard in the base language. The input provided by the user includes information about the base language syntax, optional macro definitions, and a program written in the extended version of the language. The extended version includes the features of the base language, user-defined macros, and the built-in structured programming control structures. The output produced by the preprocessor consists of a printed listing and a temporary disk file. The disk file may be used as input to the usual translator for the base language. The macro facility is a simple text replacement macro processor, included because it is not possible for one person to think of all the control structures which may be desirable at all times. The facility is crude and allows no semantic stack manipulation. A set of guidelines is provided for use in writing macro definitions to reduce the likelihood of writing macros which conflict with the principles of structured programming and, therefore, undermine the purpose of the preprocessor.