Application Program GY20-0567-1

CALL/360 - OS PL/I

System Manual - Volume I

Program Number 360A-CX-45X

The CALL/360-OS PL/I compiler (to be used with the CALL/360-OS system on an IBM System/360 Model 50 or higher) is described in the four volumes of this publication. The publication is addressed to system programmers and program systems representatives who require a detailed knowledge of the compiler. It contains a general overview of the compiler and detailed information on the compiler and runtime routines and macros that perform required functions. Additional information required to understand CALL/360-OS PL/I compiler operations is provided in several appendices.

Volume I contains a general description of the compiler, a section on compiler operations, and a directory to compiler routines.

PREFACE

This four-volume publication is addressed to system programmers and program systems representative who require a detailed knowledge of the CALL/360-OS PL/I compiler. It contains a general overview of the compiler and information on the compiler and runtime routines and macros that perform required functions.

Additional information required to understand CALL/360-OS PL/I compiler operations is provided in several appendices (see Volume IV). Appendix J contains tables of macro/module calls and summary listings of CALL/360-OS compiler and runtime routines, indicating where each routine is described in this publication.

The reader of this manual should have access to the *CALL/360-OS PL/I Language Reference Manual* (GH20-0700) and the *CALL/360-OS Terminal Operations Manual* (GH20-0787).

Terminal Equivalence

Terminals which are equivalent to those explicitly supported may also function satisfactorily. The customer is responsible for establishing equivalency. IBM assumes no responsibility for the impact that any changes to the IBM-supplied products or programs may have on such terminals.

Second Edition (January 1971)

This edition, GY20-0567-1, is a major revision obsoleting GY20-0567-0. It applies to Version 1, Modification Level 1, of CALL/360-OS and to all subsequent versions and modifications until otherwise indicated in new editions or Technical Newsletters.

Technical changes to text are indicated by vertical lines in the left margin. A revised illustration or chart is indicated by the symbol • to the left of the caption.

Changes are continually made to the information herein. Therefore, before using this publication, consult the latest System/360 SRL Newsletter (GN20-0360) for the editions that are applicable and current.

Copies of this and other IBM publications can be obtained through IBM branch offices. A form has been provided at the back of this publication for readers' comments. If the form has been removed, address comments to: IBM Corporation, Technical Publications Department, 112 East Post Road, White Plains, New York 10601.

CONTENTS - VOLUME I

FIGURES - VOLUME I

CONTENTS - VOLUME II

FIGURES - VOLUME II

CONTENTS - VOLUME III

CONTENTS - VOLUME IV

FIGURES - VOLUME IV

SECTION 1 - GENERAL DESCRIPTION

The CALL/360-OS PL/I compiler (to be used with the CALL/360-OS system on an IBM System/360 Model 50 or higher) is described in this manual. The compiler accepts, as input, a complete source program written in the CALL/360-OS PL/I language. The language (described in the CALL/360-OS PL/I Language Reference Manual) is oriented to a scientific user. It was selected on the basis of its utility to such a user, as well as the ease with which it could be implemented. The compiler produces a machine-language (object) program directly into memory and, if no severe errors were detected during compilation, initiates execution of the program.

This section of the manual presents the CALL/360-OS PL/I compiler as a whole and presents the overall logic of its implementation. It is intended for a reader having some knowledge of the time-sharing system under which it will run and of the CALL/360-OS PL/I language. Furthermore, it is designed for a reader who desires a broad picture of the design of the compiler.

PHILOSOPHY

The CALL/360-OS System is designed to support a large number of users concurrently interacting with the computer through typewriter-like input/output terminals. Since each user is in effect conversing with the system, a basic requirement is that the system respond within a reasonable time. In order to support these users with a limited amount of high-speed computer memory, some of the modules are stored temporarily in online storage units (disks) and called in for execution as needed. Since the number of modules that can be contained in high-speed storage at any one time is dependent upon their sizes, they should be as small as possible. Compiler modules that are loaded from disk for execution will not always be loaded to the same core location; therefore, they must be dynamically relocatable.

While the terminal user is entering his source text into the computer, he is in communication with the CALL/360-OS Executive (not with the compiler). After the complete text has been entered and the user has requested that the program be executed, the compiler is invoked. Once invoked, the compiler processes the complete source text, generating the object code into an area of memory allocated by the Executive.

OBJECTIVES

The compiler is designed to achieve the following goals:

1. To be as small as possible.
2. To be as fast as possible.
3. To provide a compilation facility for the CALL/360-OS PL/I language, with good diagnostic capability.
4. To produce efficient object code.

To a certain extent, these goals are mutually exclusive. A small, fast compiler is usually not able to produce efficient object code. Also, the extent of the language facilities supported by a small, fast compiler is usually very limited. The design of the CALL/360-OS PL/I compiler gives first priority to objectives 1 and 2, and secondary priority to objective 3. It strives for the best possible attainment of objective 4 within these constraints.

FACILITIES

The CALL/360-OS PL/I compiler operates in conjunction with and is controlled by the CALL/360-OS Executive. When the compiler is invoked, it processes and translates a CALL/360-OS PL/I source program directly into machine-readable object code (in main storage). The compiler does not produce a binary deck or any hard copy listings of the generated object code. Nor is it possible to save or store the compiled code immediately after compilation. Compilation and execution are invoked simultaneously (unless severe errors cause cessation of processing).

The command that the user must give to initiate both compilation and execution is the CALL/360-OS system command RUN. (See the CALL/360-OS Terminal Operations Manual, Form Number GH20-0787.) This command initiates a sort, if necessary, followed by compilation and execution of the user program. After being invoked, the CALL/360-OS PL/I compiler provides diagnostic messages on the user terminal for any errors detected during compilation. (See "Compiler-Time and Runtime Error Messages" in this section and Appendix F of the CALL/360-OS PL/I Language Reference Manual, Form Number GH20-0700.) When compilation and execution have been completed or terminated because of an error condition, the user can utilize available editing facilities to correct his program if necessary. Then he can enter another RUN command to recompile and reexecute his program.

A user's program consists of a single external procedure and the runtime library routines needed to support it. Although the external procedure may contain internal procedures, there are no facilities for linking separately compiled external procedures.

In addition to the compiler itself, a package of library routines is provided to support the object program at execution time. These routines are described more fully in later sections of this manual.

APPROACH

To meet the primary objectives of minimum size and maximum speed, the CALL/360-OS PL/I compiler has been designed to translate a CALL/360-OS PL/I source program in one pass over the source statements. That is, code is generated for each statement at the time and in the order it is encountered in the source stream. This does not prevent the compiler from scanning a statement several times during the compilation, but each statement is fully processed before processing of the next statement is begun.

One phase of the compiler, the source text, and the target area for the object program are completely contained in core memory during a compilation. None of these areas is overlaid. This approach places a practical limitation on the size of the program that can be compiled but avoids the necessity to include costly spill and overlay logic in the compiler.

The compiler itself is both reenterable and relocatable. These characteristics are necessary because CALL/360-OS, in essence, performs multiprocessing within its own partition in addition to handling and servicing interrupts and commands from any of a number of terminals.

Main storage is allocated for all of the resident modules and subroutines, I/O appendages, tables, CPID (control program interrupt dispatcher), an overlay buffer for nonresident modules, terminal and disk I/O buffers, compiler area, new-job-area-1, new-job-area-2 (optional), old-job-area, and any necessary OS/360 control blocks. At any given time, a number of user jobs may be in process. Furthermore, these jobs may require different compilers (in which case, one compiler may be overlaid by another compiler). Typical organization of main storage when the CALL/360-OS PL/I compiler has been invoked is shown schematically in Figure 1-1. In this example, a computer having 512K of main storage is assumed.

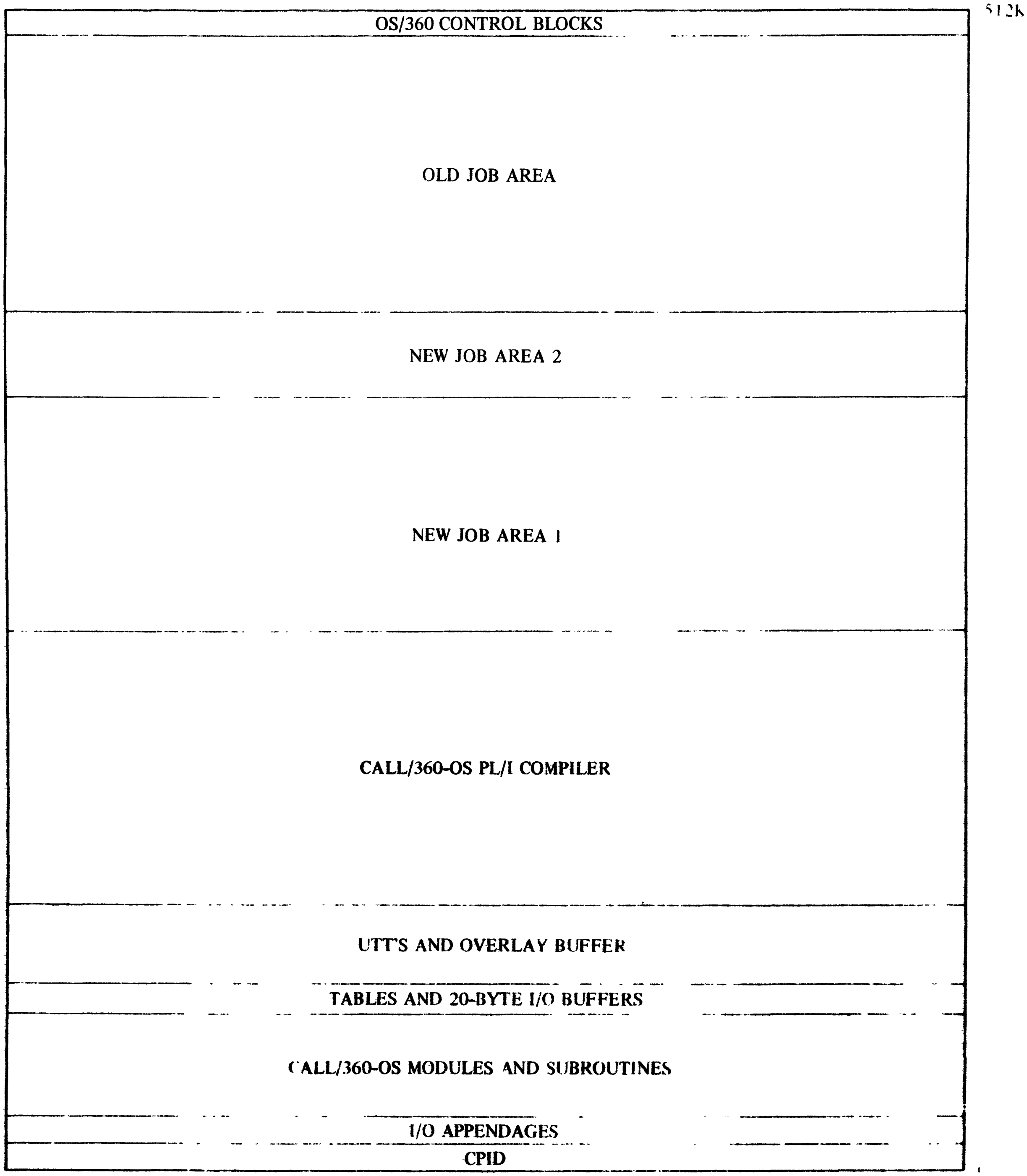


Figure 1-1. A Layout of Main Storage for CALL/360-OS PL/I

CALL/360-OS facilities also include time-slicing for user jobs. When the time allotted to a specific user job has expired or when a disk I/O operation or a terminal request for data occurs, the user job is swapped out of core to his disk work swap area (which was allotted to him at sign-on time). Depending on the condition forcing swap out, the user job is again queued for one of the work areas in core. It may or may not be read into the same main storage locations that it occupied previously. For this reason, the object code produced by the compiler must, like the compiler, be relocatable.

Upon initial entry into the compiler, the Executive supplies the compiler with two parameters: the address of the current location of the compiler and the address of the target area for the user's object program. These addresses are the first locations of the compiler area and the user work area, respectively. A communications area containing other information needed by the compiler is at the start of the target area. At the end of the target area is the complete source text to be compiled.

Within the communications area is the address of a user terminal table (UTT) entry for this user. The UTT entry contains, among other items, the amount of storage that has been allocated to this user and the location and size of the source text.

Phases 1 and 2 of the compiler operate on the source program and produce an executable object program in the same user work area. The compiler uses part of this area for its own work space. Upon completion of compilation, the object program is invoked immediately by the compiler (unless serious errors were detected during compilation).

The compiler area holds either phase 1 or phase 2 of the compiler. Since both phases of the compiler are reenterable and relocatable, areas within the compiler itself are not modifiable.

As noted previously, the compiler may process several jobs concurrently on a time-slicing basis. In such a situation, the compiler operates on multiple user areas. Each work area and the compiler area are subject to relocation (as well as swapping) on an individual basis. If the Executive requires an area being utilized as a user work area, it saves the content of the area on disk storage and then relocates it. If the Executive requires the compiler area, it need not save the content of the area, since the compiler is reentrant.

COMPILER ORGANIZATION

The routines of the compiler may be grouped within ten categories:

1. Executive routines, consisting of the initializers for both phase 1 and phase 2 of the compiler.
2. Entokening generators responsible for analyzing the source program for syntactic units.
3. General statement processors responsible for directing the syntactic and semantic analyses of CALL/360-OS PL/I source statements.
4. Declaration processing responsible for creating and finding definitions for all declarations and identifiers in the source program.
5. I/O statement processors.

6. Expression processing responsible for analyzing all expressions in the source program.

7. Code generators, which (upon direction of other compiler routines) produce the object code to be generated.

8. Wrap-up, which loads the required library routines and otherwise prepares the user area for execution.

9. Miscellaneous support routines.

10. Executive interface.

The processing of each statement is directed by the Controller routine. It invokes the Entoken routine to analyze the characters comprising the source text and to identify the significant tokens (that is, delimiters, constants, and identifiers) of the next statement in sequence. An entry is placed into a token table for each token in the statement, which is then analyzed by the Controller to determine which statement generator is to be invoked. The invoked generator performs a syntactic and semantic check of the validity of the statement and directs generation of the required object code.

The other areas perform functions that are common to several of the other routines (such as code generation for expressions, identifier resolution, label processing, and storage allocation). Figures 1-2 and 1-3 show the functional relationships of routines of the CALL/360-OS PL/I compiler.

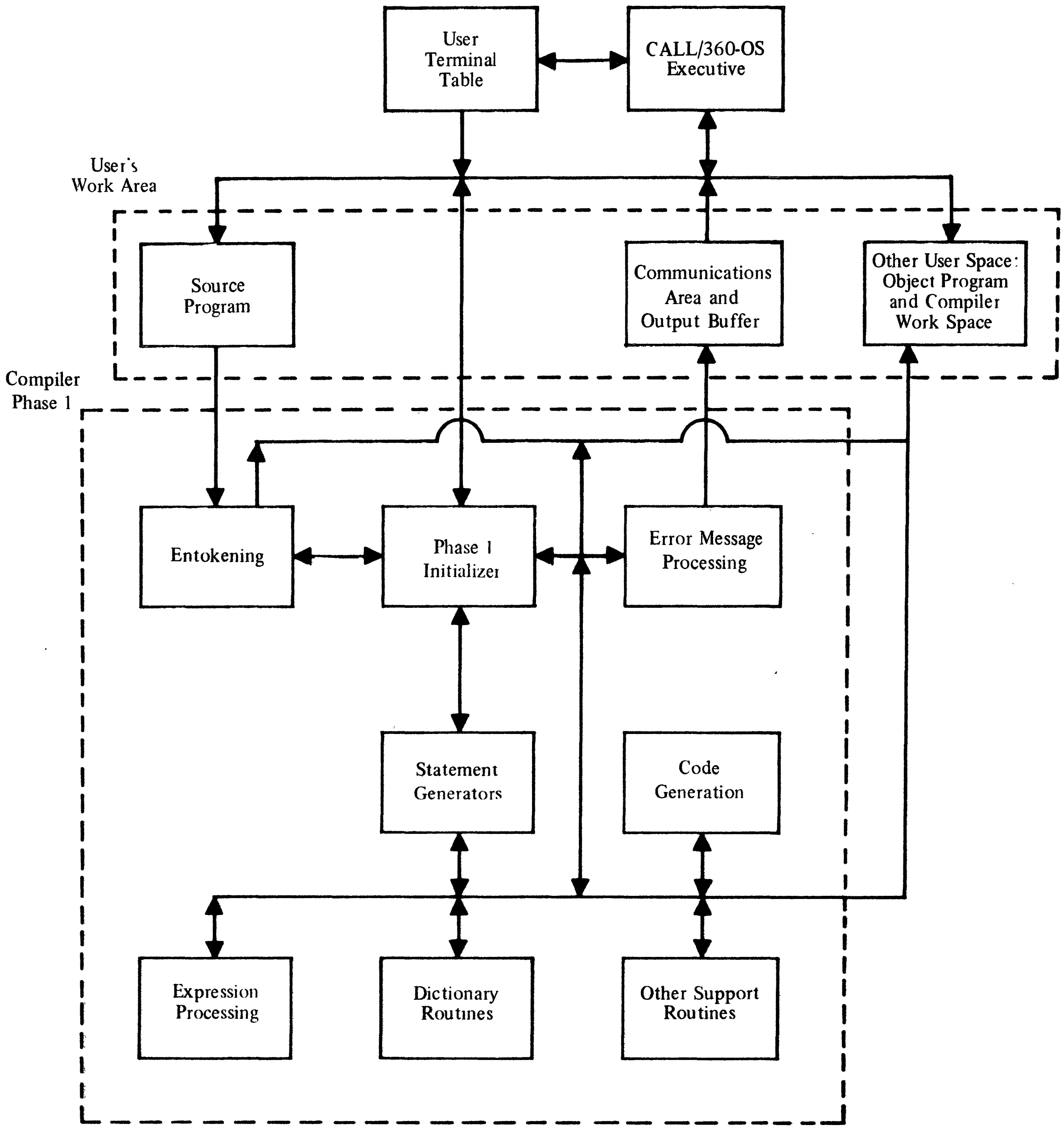


Figure 1-2. Functional Organization of CALL/360-OS PL/I Compiler, Phase 1

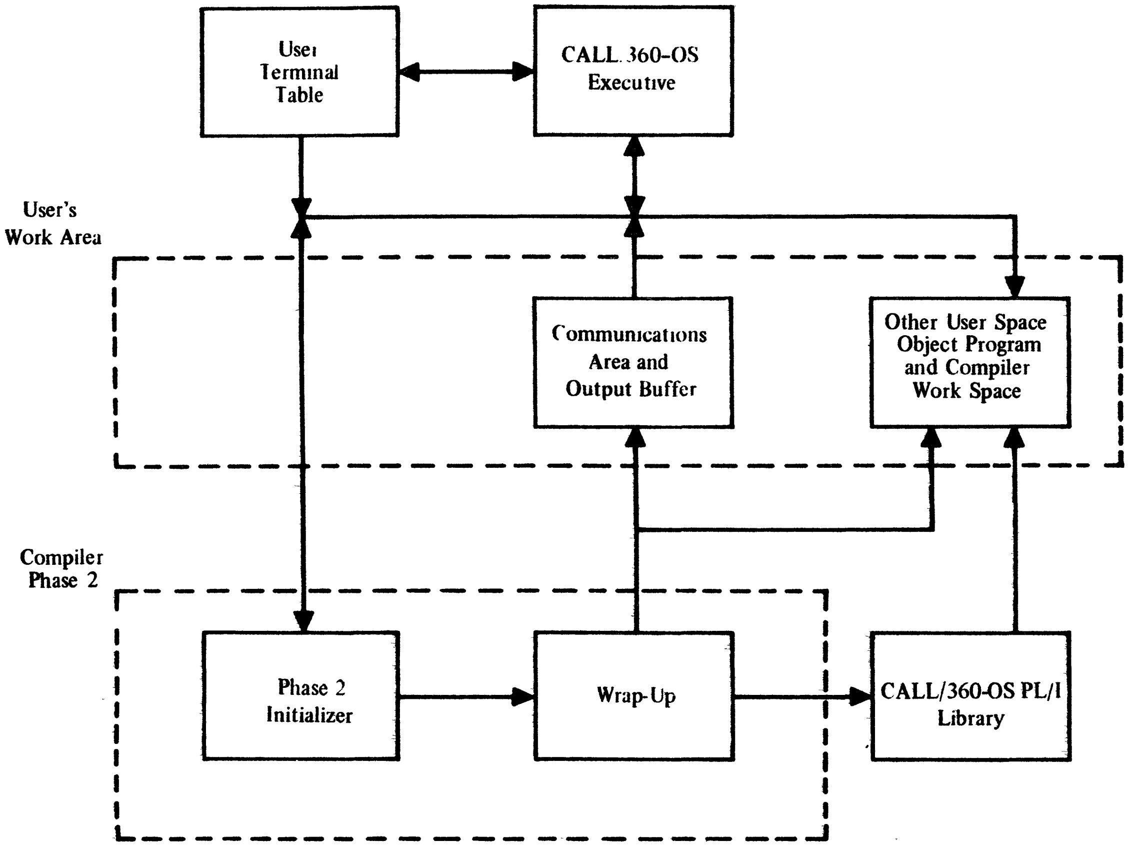


Figure 1-3. Functional Organization of CALL/360-OS PL/I Compiler, Phase 2

DATA ORGANIZATION

The requirement that the compiler must be reenterable implies that all of its working storage must be in the user work area. Since the location of this area may change during compilation, all addresses of items within this area must be relative to the base address of the area or contained in a special address constant area. The free space in the user work area is used by the compiler for all of its working storage and for the object program. The object program is built from the beginning of this area (following the communications area) to the end, and the working storage is allocated from the end (preceding the source text) to the beginning. If these two areas overlap, the work area is insufficient in size. The compiler returns to the CALL/360-OS Executive with a request for more space.

Most of the compiler's data is maintained in either expandable tables or threaded lists. If it is stored as an expandable table, the entries are stored consecutively in an area of fixed length. When one table area fills, that area is linked to another area (where another set of consecutive entries is started). If the compiler data is stored in the form of a threaded list, entries are not necessarily in consecutive locations. Each entry contains a pointer to the next entry in sequence, wherever it is located. The first byte of an expandable table area or the first byte of an entry in a threaded list contains an identification code. This code byte is often used by the compiler. However, its primary purpose is to simplify the checkout and maintenance of the compiler.

The compiler's working storage area consists of both a fixed-size and a variable-size work area. Space for the fixed area is set up by the Phase 1 Initializer routine. This fixed area is allocated backwards from the beginning of the user work area. Space for the variable area, which consists, primarily, of expandable tables and threaded lists, is obtained as needed during the compilation process. This area extends backwards from the beginning of the fixed area.

The fixed-size working storage area is divided into three main parts; a C-area or compiler adcon area, a P-area or program adcon area, and a G-area or general area.

The C-area contains addresses of compiler modules, locations within these compiler modules, and register save areas for C1, C2, and C3 (registers 8-10). Registers C1, C2, and C3 are used exclusively to provide compiler module cover and for linkage between compiler modules.

The P-area contains addresses of locations within the user work area and register save areas for P0, P1, P2, P3, P4, and P5 (registers 11-1, excluding 0). These registers are used exclusively for referencing locations within the user work area.

The G-area is used for all other purposes. It does not contain any adcons, but it does contain pointer words which are displacements to locations within the user work area. It also contains save areas for registers G0, G2, G3, G4, G5, G6, and G7 (registers 0 and 2-7). These registers cannot contain actual addresses when the program is swappable.

When either phase 1 or phase 2 of the compiler is relocated, registers C1, C2, and C3 and the adcons in the compiler adcon area of fixed working storage (C-area) are adjusted by the amount of the relocation.

When the user work area is relocated, registers P0, P1, P2, P3, P4, and P5 and the adcons in the program adcon area of fixed working storage (P-area) are adjusted by the amount of relocation.

No other areas are affected by relocation during compilation. This includes the G-area; registers G0, G2, G3, G4, G5, G6, and G7; the object code being generated; and the variable-size working storage area.

After the END statement for the external procedure has been processed, phase 2 of the compiler is invoked. Phase 2 is brought in by the Executive and overlays the code for phase 1. The function of phase 1 was primarily to generate machine-language (object) code for CALL/360-OS PL/I source-program statements. The function of phase 2 is to set up the items referenced by the object code. Constants, library routines, and adcon areas are set by processing the tables generated during phase 1. Space is reserved for necessary input/output buffers, and execution of the user's object program is initiated.

The execution (runtime) phase involves only one area, namely, the user work area. When this area is relocated during runtime, registers P0 through P9 (registers 6-15), the adcons in the address constant area, and the adcons in the relocatable library work space are adjusted by the amount of the relocation. The execution phase is self-contained. That is, all library routines have been loaded and there are no overlay facilities.

The layouts of the user work area during compilation and during execution are shown in Figure 1-4. (For details, see Appendix E.)

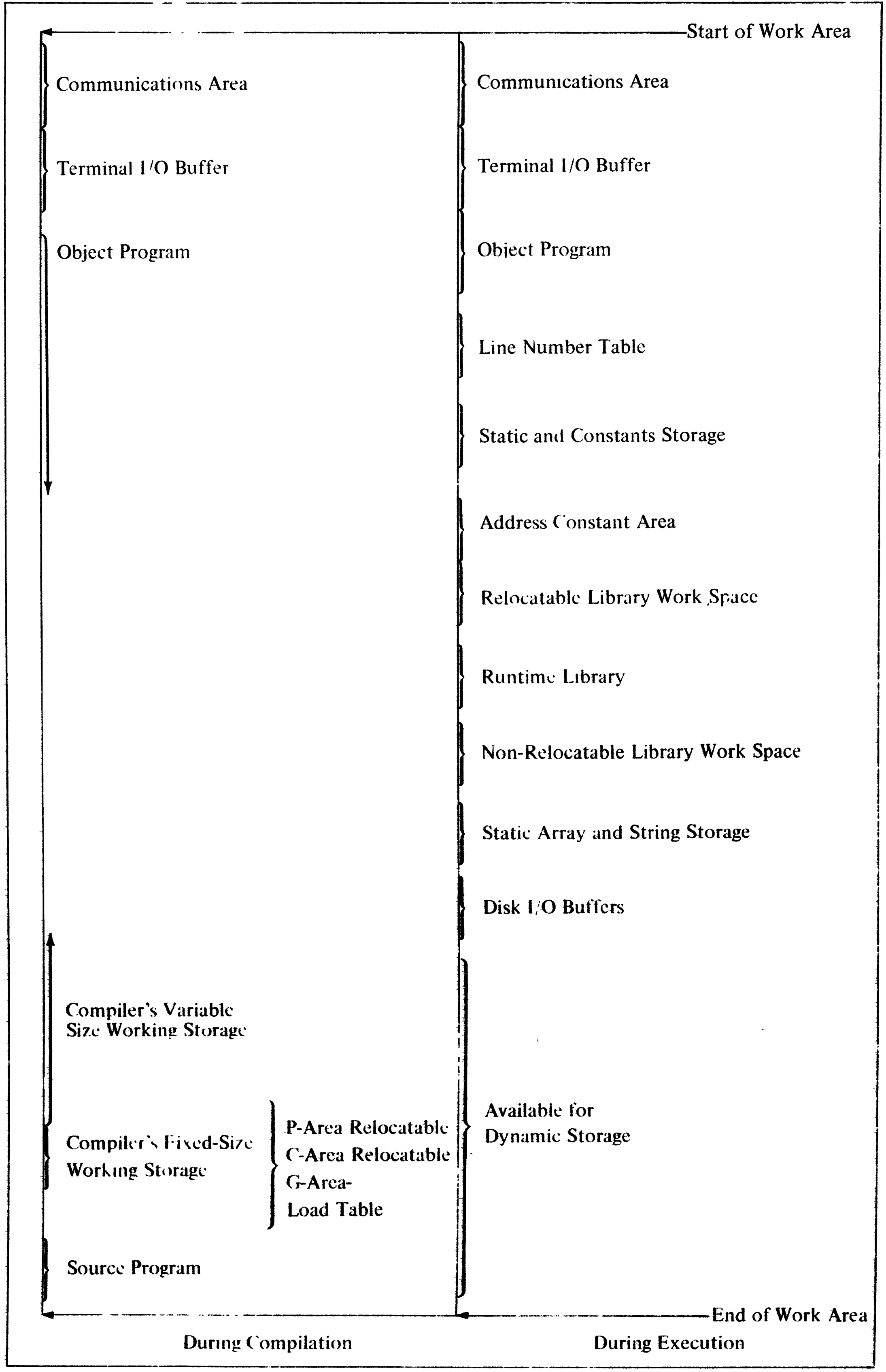


Figure 1-4. Layouts of User Work Area

NAMING CONVENTIONS

Runtime library routines are named IHExxx where xxx is a three- to five-character mnemonic indicating the routine's function. Phase 1 and phase 2 compiler modules are named $xxx where xxx is a one- to seven-character mnemonic indicating the routine's function.

Note: The CALL/360-OS PL/I compiler and runtime modules follow the naming conventions stated above. For clarity, these conventions are used throughout this manual. However, there are some exceptions in the member names assigned to certain routines when stored in CALL/360-OS PL/I system libraries. These exceptions are noted in Figure J-4 and under "CALL/360-OS PL/I Runtime Library" in Appendix J of Volume IV.

COMPILER LIMITATIONS

Because all tables and the object program are kept in main storage during program compilation, compilation may be terminated because the user work area is too small. This is the only valid (that is, source-program related) reason for termination of compilation. Compilation may also be terminated if the CALL/360-OS PL/I compiler discovers an "impossible" logic path or because a program interrupt occurs. Such a condition is due to either a machine error or an error in the CALL/360-OS PL/I compiler.

Before compilation is terminated because of insufficient work space, CALL/360-OS PL/I requests additional main storage from the Executive. However, for a given computer size, there is a maximum work space that the Executive can provide. At present, on a machine having 512K bytes of main storage, a user work area can be up to 112K bytes in size. (See Appendix I for details on the maximum size of a program to be compiled on a 512K computer.)

A programmer, by specifying an out of range value for the subscript of an array, may destroy the contents of his static array and string, disk buffer, or dynamic storage area. Complete storage protection is not provided for these areas. As an example, assume that an element of an array A is specified outside of the area for which the array has been declared. That is, a program may contain the following statement.

```
    A(I)=10;
```

Assume that, when this statement is executed, the current value of I is outside of the range declared for the subscript of A. This specification can cause data to be written in one of the three areas mentioned above. An error message is generated only if the value 10 would be moved into an area other than one of those three.

COMPILER-TIME AND RUNTIME ERROR MESSAGES

During the compilation process, the CALL/360-OS PL/I compiler modules analyze source statements of the program being compiled. If incorrect use of the language is detected, an error message identifying the error condition is printed out at the terminal. (See Appendix F of the CALL/360-OS PL/I Language Reference Manual for a description of CALL/360-OS PL/I compilation and execution error messages. These messages are also listed in Appendix H of this manual.)

A severity code is associated with each compiler-time error message. If any statement causes a message having severity code 3 to be generated, the compiled program is not executed.

If an error condition arises during program execution, an error message is printed (unless the programmer has indicated alternative action by means of an ON statement). Error conditions are of two types. Some cause execution to be terminated immediately, but others permit execution to continue.

SECTION 2 - COMPILER OPERATIONS

OVERVIEW

The CALL/360-OS PL/I compiler is a one-pass, two-phase compiler--one-pass, since it processes each statement completely before going to the next; two-phase, because it is separated into two parts, phase 1 and phase 2.

Phase 1 takes one pass over the source code and translates source statements to object code. During this pass, the entokening routines translate the source stream into intermediate syntactic units called tokens. The tokens are an unarranged, concise description of the source statements. The expression processing routines transform these tokens to another intermediate form called triads. Actually, triads are the original source-program operators and operands rearranged so that the specified operations will be executed in proper order. The triads are converted to final object code by the code generator segment of phase 1.

After phase 1 has processed the last statement, it issues a request to the CALL/360-OS Executive to bring in phase 2. The Executive overlays phase 1 with phase 2. Phase 2 performs wrap-up and loads the library routines.

Figure 2-1 depicts the internal compiler logic of phases 1 and 2. As noted above, each statement is completely processed before processing of the next statement begins.

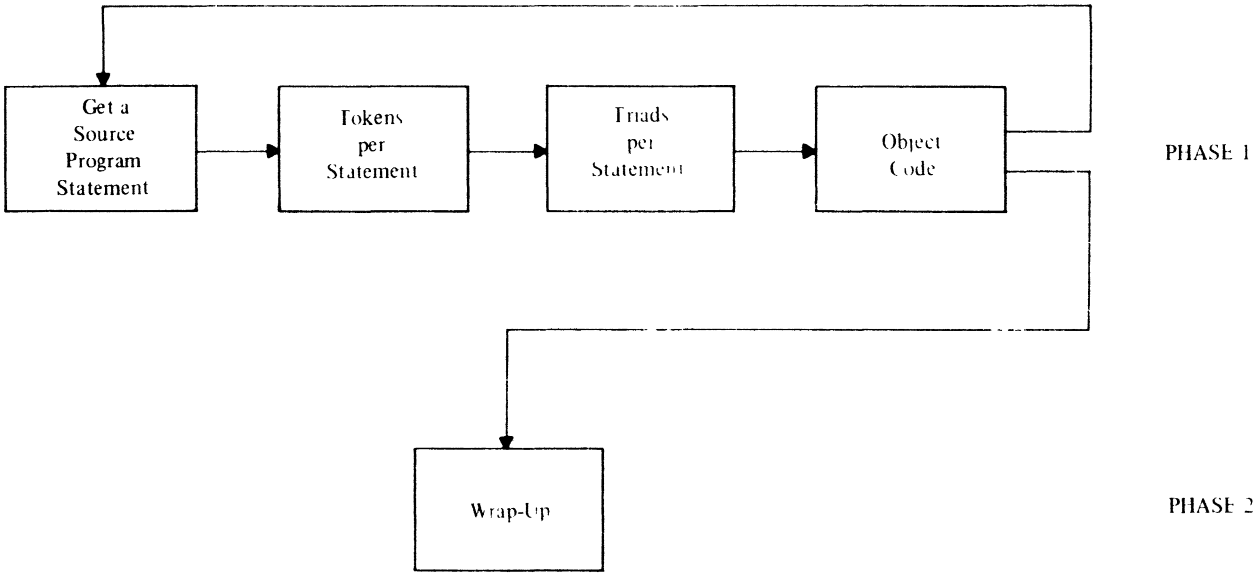


Figure 2-1. Internal Compiler Logic

PHASE 1

After initialization of the compiler work area, which is performed by the Phase 1 Initializer ($CCONT), control is transferred to the Controller ($CNT). The Controller directs the entokening of statements, analyzes most label prefixes, and directs control to statement processors and code generation routines.

The usual sequence of events is:

1. The Controller calls the Entoken routine ($ATKN) to entoken a statement.

2. The Controller examines the statement tokens to determine whether the statement is a keyword type or assignment statement. If it is a keyword type, the first token of the statement, excluding any possible label prefix token, determines the keyword type. The Controller directs control to the statement processor which handles that keyword type. Any statement that is not a keyword type statement is an assignment statement; the Assignment Generator ($ACGEN) is called.

   Note: Assignment statements may begin with keywords (for example, IF=1). These statements are recognized as assignment statements.

3. The called routine processes the tokens of the statement to create triads and then returns control to the Controller.

4. The Controller then directs control to the code generation routines to produce object code from the triads.

ENTOKENING

The primary function of entokening is to convert the syntactic units of each source statement to tokens. Each token is one word long. The syntactic units that may be encountered are:

- Identifiers
- Keywords
- Delimiters such as + , (* = or ;
- Constants
- Comments

The token created for an identifier or keyword references the dictionary name list entry for that item. If there is no dictionary name list entry, one is made. (See the Search-Insert routine ($FIND) for details.)

The tokens created for delimiters (with the exception of the left parenthesis token) are self-contained. That is, there are no pointers to other items in the token word. A left parenthesis token points to the corresponding right parenthesis token. When a constant is encountered, a constant attribute entry is created and the generated token references this attribute. No tokens are created for comments.

Entokening is controlled by the Entoken routine ($ATKN). The Scan Index routine ($ASIDX) is used to advance to the next source character. The Search-Insert routine ($FIND) is called to search for a dictionary name list entry and to create an entry if there is none. The Get Non-Blank routine ($FNB) is called to advance to the next non-blank source character. The relationship of these routines is shown in Figure 2-2.

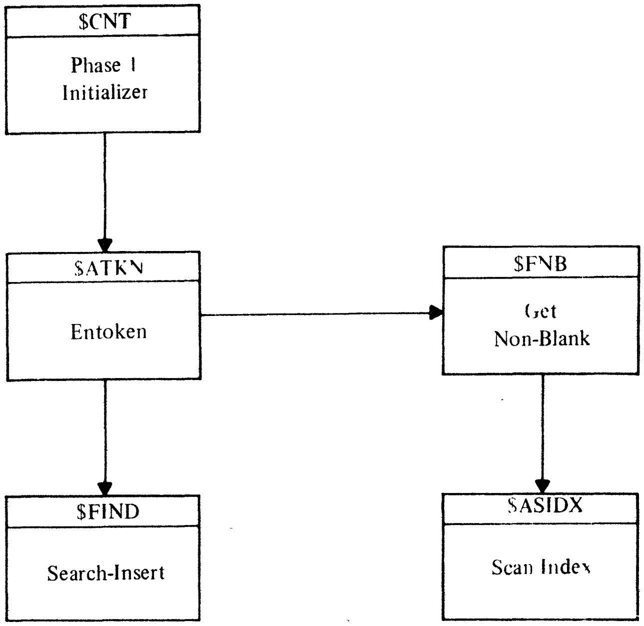


Figure 2-2. The Entokening Process

GENERAL STATEMENT PROCESSING

General statement processors are called by the Controller ($CNT). If a keyword is found in a statement, $CNT calls a statement processor as follows.

Keyword	Routine
PROC or PROCEDURE	PROC Generator ($APRC)
BEGIN	BEGIN Generator ($BEGIN)
GO or GOTO	GO Generator ($BRNH)
CALL	CALL Generator ($CALL)
IF	IF Generator ($CIF)
ON	ON Generator ($CON)
STOP	STOP Generator ($CSTOP)
REVERT	REVERT Generator ($CRVT)
DO	DO Generator ($DOG)
RETURN	RETURN Generator ($DRET)
END	END Generator ($EDGN)

If no keyword is found, the statement is an assignment statement. The Assignment Generator routine ($ACGEN) is called to process this statement. (The statement processors listed above do not include those which handle I/O or declaration processing. These routines are discussed later in this manual.)

The processing performed by each of the routines listed above is unique, but, in general, the routines examine tokens and build appropriate triads. The statement processors also check for logical consistency. The keyword routines and the Assignment Generator call the Expression Processor Controller ($NEXP) to process:

- Assignment statements (called by Assignment Generator)
- Expressions such as A=5 in the statement

  IF A=5 THEN GO TO HFILE;

- CALL statement entry names and argument lists

Other routines discussed together with the general statement processors are:

Routine	Caller and Function
On-Unit Specification Analyzer ($BONSA)	Called by the ON Generator ($CON) to check the legality of an ON-condition.
DO-Loop Triad Builder ($DEXP)	Called by the DO Generator ($DOG) and the Expander ($EXPND). Its function is to build the triads which initialize the DO index, increment the DO index, and test for DO-loop termination.
Process End of ELSE ($ENDES)	Called by the Controller ($CNT) when it detects the end of an IF statement. (Note that an IF statement may or may not be terminated by an ELSE clause.) This routine resolves branches to the end of the IF statement.
Process End of ON ($ENDON)	Called by the Controller ($CNT) when it detects the end of processing to be performed as the result of an ON-condition.
Expander ($EXPND)	Called by the Assignment Generator ($ACGEN) when it detects an array expression. This routine generates one DO triad for each dimension of the array.
END Expand ($EYPND)	Called by the Assignment Generator to end the range of DO's created by the Expander ($EXPND).

DECLARATION PROCESSING

Information about an identifier is contained in its dictionary name entry and its dictionary attribute entry(s). These entries are in the dictionary name list and dictionary attribute list respectively. An identifier may be used or declared differently in different blocks; thus, there may be more than one dictionary attribute entry per identifier. However, there can be at most one dictionary attribute entry for an identifier per block.

A dictionary attribute entry is constructed for an identifier when its attributes are specified either in a DECLARE statement or by the identifier's contextual usage. The contextual usage may either cause a new dictionary attribute entry to be created or cause a description to be added in an existing dictionary attribute entry.

Note: Dictionary attribute entries are also made for constants. These entries are formed in a section of the token table and are discarded after the statement has been processed.

Four types of declarations are allowed:

- Explicit
- Implicit
- Contextual
- Tentative

An explicit declaration is made when a label appears on a statement or when an identifier appears in a DECLARE statement (except for those entry names that are not also parameters).

An implicit declaration is created for any use, in an arithmetic position, of an identifier that is not followed by a left parenthesis and which has no previous declaration in the scope of usage.

Contextual declarations are made for identifiers used in file options in GET, PUT, OPEN, or CLOSE statements and for declarations of non-parameter entry names in DECLARE statements.

A tentative declaration is made for each parameter encountered in the processing of a PROCEDURE statement; for a scalar not followed by a left parenthesis, when it is used in a position of a label and there is no other declaration in the current block (such as GO TO A, where A has not been encountered previously); or for a scalar followed by a left parenthesis, when it is used in an arithmetic position and there is no previous declaration in the scope of usage.

The four declaration categories listed above do not correspond directly to those discussed in the CALL/360-OS PL/I Language Reference Manual. The language reference manual discusses only implicit, contextual, and explicit declarations. The tentative type was added because of the one-pass nature of the CALL/360-OS PL/I compiler. It is only significant internally. The contextual definition of the entry name not a parameter in the DECLARE statement is also at variance with the CALL/360-OS PL/I Language Reference Manual. Again, this is an internal convenience, since the declaration will become explicit when the procedure with that name is encountered. Classification of an identifier as explicit, implicit, or contextual indicates its scope. A more complete discussion of declarations is contained in the language reference manual.

The routines of the CALL/360-OS PL/I compiler that perform declaration processing are:

- DCL Generator ($DCLGN)
- Attribute Analysis ($ABAL)
- Attribute Node Creation ($ANCRE)
- Locate Identifier ($FSYM)
- Locate Variable ($FVAR)
- Label Processor ($BLPRC)

The DCL Generator ($DCLGN) is called as the result of a DECLARE statement. It directs the analysis and encoding of attributes for identifiers in that statement. In doing so, it calls the other declaration processing routines as subroutines.

$ABAL analyzes and encodes an identifier's attributes; $ANCRE creates a dictionary attribute entry; and $FSYM determines whether a dictionary attribute entry has previously been created in the block for this entry. If a previous declaration conflicts with the present declaration, an error message is issued.

As noted above, the Attribute Analysis routine ($ABAL) is called by $DCLGN to analyze and encode the attributes of an identifier. It is also called by the PROC Generator ($APRC) to process any RETURNS attributes. The results of this routine are placed in the attribute table ($ABTBL). This table should not be confused with the dictionary attribute list (A list). (For the format of the attribute table, see Appendix A.)

The Attribute Node Creation routine ($ANCRE) is also called by either $DCLGN or $APRC. Its function is to create a dictionary attribute node using the information placed in the attribute table by $ABAL. If the identifier is dimensioned, a skeletal dope vector is built. If the array bounds are constant, the dope vector bounds are initialized. If the array bounds are variable, code that will calculate the dope vector bounds at execution time is generated.

The Label Processor routine ($BLPRC) is called to process all label prefixes. It creates a dictionary attribute entry for the statement label or entry name and resolves any tentative declaration previously made (for example, as a result of a GOTO statement).

The Locate Identifier routine ($FSYM), although grouped with the declaration processing routines, is not used solely for declaration processing. Inputs to this routine are an identifier token and a block number. $FSYM determines whether the identifier has been previously declared or used in the block indicated or in a block which is internal to the indicated block.

The Locate Variable routine ($FVAR) is not referenced by other declaration processing routines. It is called by the CALL, OPEN/CLOSE, GET, PUT, or Format List Generator when certain variables are encountered. It is also called by the Expression Processor Controller ($NEXP) for each variable which it encounters. This routine is used to locate the most recent or the current dictionary attribute entry for a variable, and, if there is none, to create a definition for it. If there is more than one dictionary attribute entry for an identifier, the most recent entry is the entry with the highest block number.

INPUT/OUTPUT STATEMENT PROCESSING

CALL/360-OS PL/I programs have the ability to transmit or receive stream-oriented data (that is, data in the form of a continuous stream of characters). This form of data contrasts with record I/O, which is not supported in CALL/360-OS PL/I. Data may be transmitted or received from two types of devices: remote terminals and disk units. A program may address only one terminal, namely, the one at which the program was entered. An unlimited number of disk files may be processed; however, only four may be open at the same time. For both terminal and disk, the data stream may be specified as list-, data-, or edit-directed.

The I/O routines process the OPEN, CLOSE, GET, PUT, and FORMAT statements. (GET, PUT, and FORMAT are used to address both terminal and disk; OPEN and CLOSE pertain only to disk.) The relationships of the I/O routines are illustrated in Figures 2-3, 2-4, and 2-5. Details are given in the paragraphs which follow.

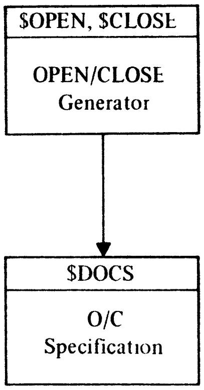


Figure 2-3. OPEN/CLOSE Processing

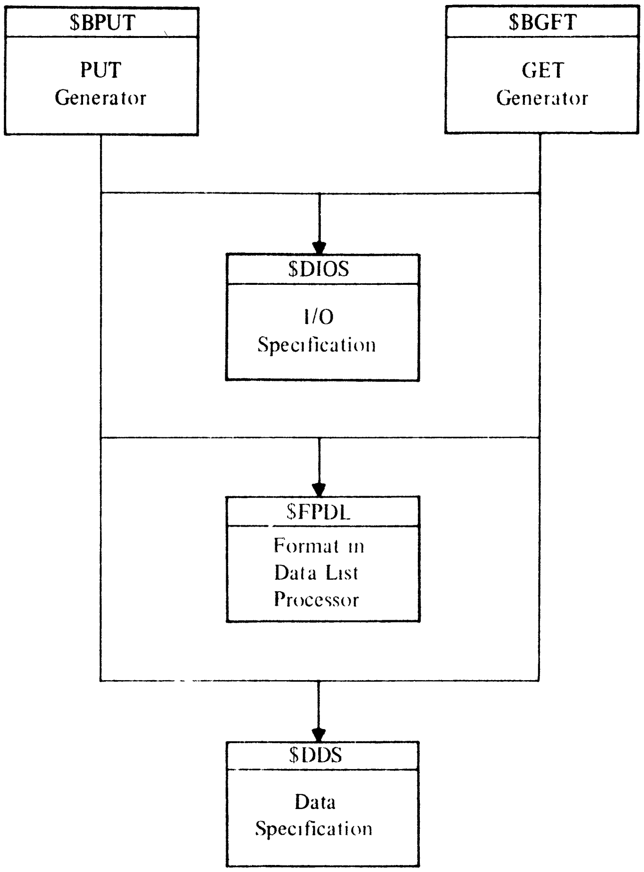


Figure 2-4. GET/PUT Processing

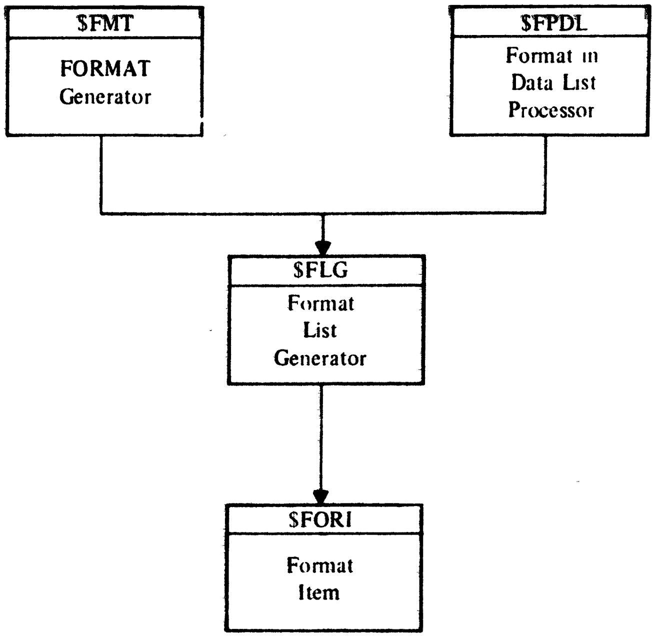


Figure 2-5. Format List Processing

In processing I/O statements, the compiler generates calls to library runtime routines. The library runtime routines associated with list-, data-, and edit-directed types of I/O are shown in Figures 2-6, 2-7, and 2-8.

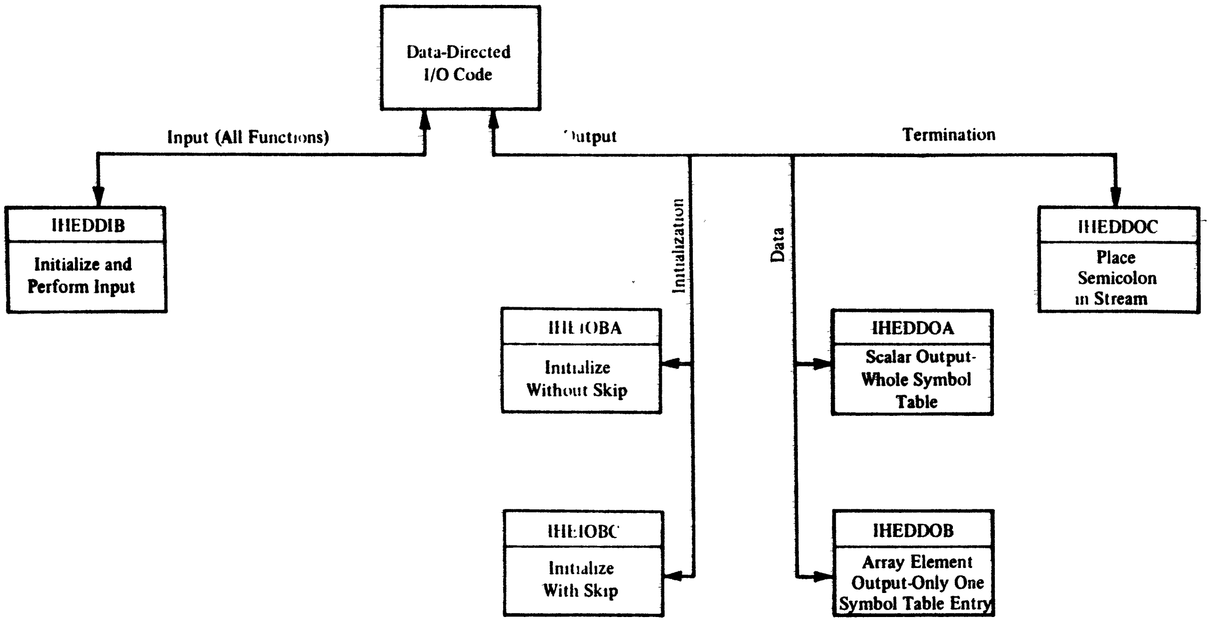


Figure 2-6. Data-Directed I/O Flow of Control

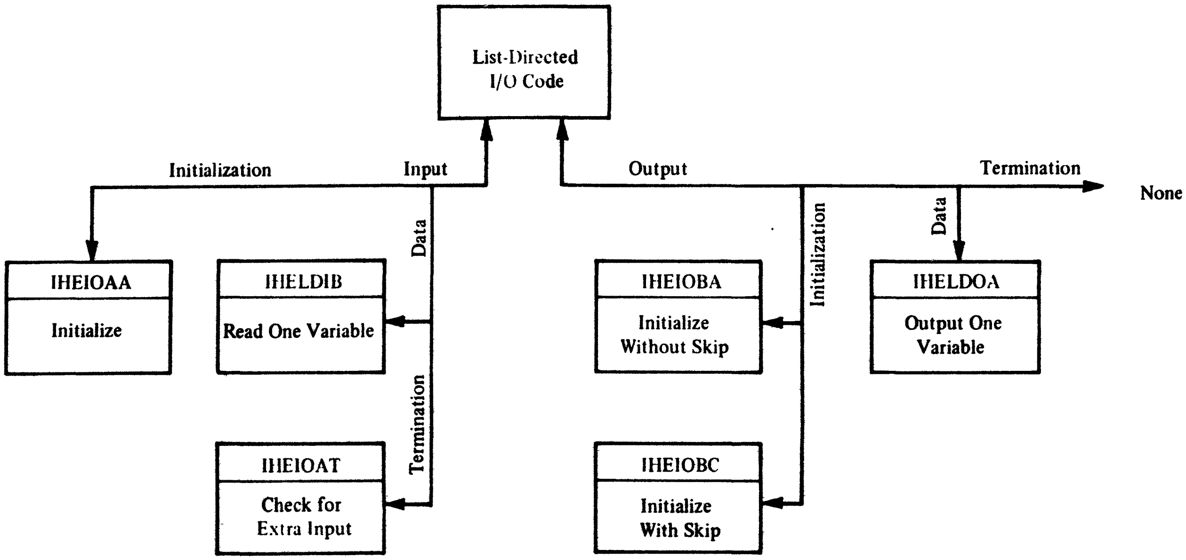


Figure 2-7. List-Directed I/O Flow of Control

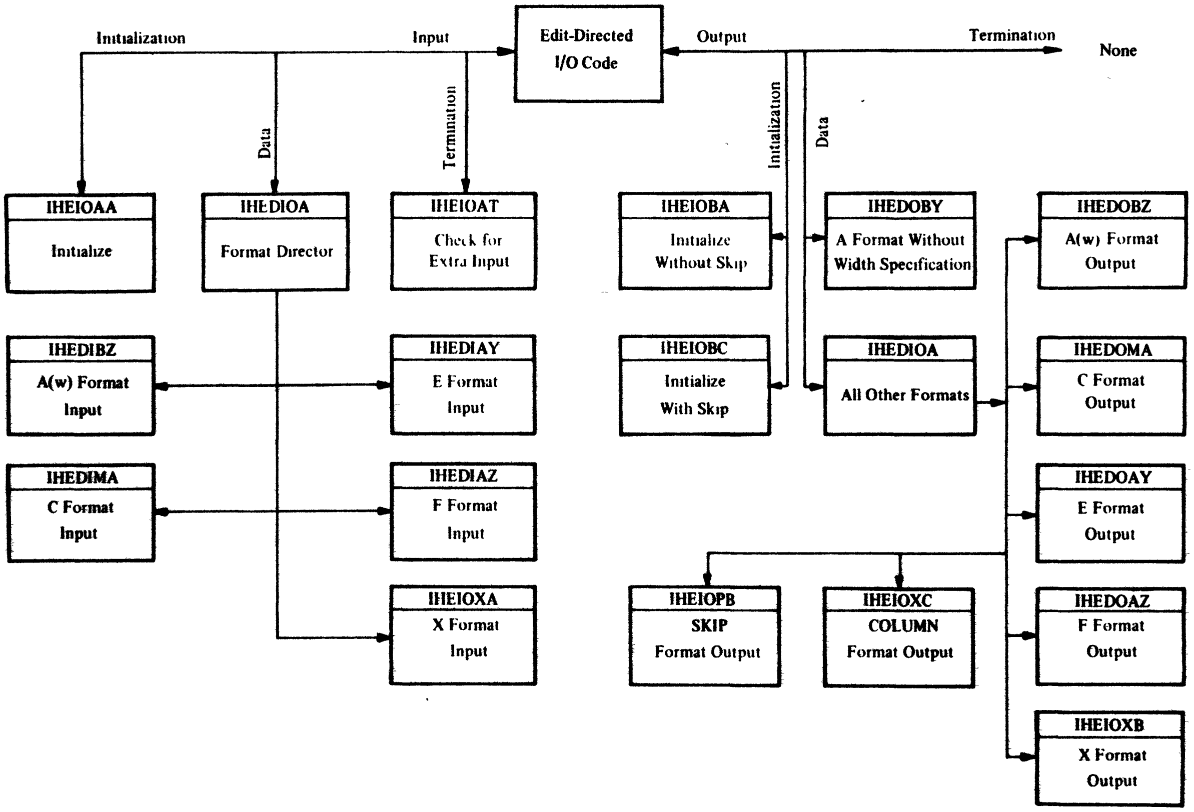


Figure 2-8. Edit-Directed I/O Flow of Control

The Phase 1 Initializer ($CCONT) calls the OPEN/CLOSE Generator ($OPEN, $CLOSE) when it encounters an OPEN or CLOSE statement. The OPEN/CLOSE Generator calls the O/C Specification routine ($DOCS) to scan the statement and determine which options are present. (Available options are the INPUT, OUTPUT, FILE, and TITLE attributes.)

If the statement is CLOSE, code is generated to:

1. Obtain the location of the file control interface block (FCIB)
2. Call the library runtime Close routine (IHECLOSE).

If the statement is OPEN, space is obtained in the initialization table (I table) for a file control interface block (FCIB) if the file was not previously opened or declared. When phase 2 of the compiler processes the initialization table, it will move this FCIB to static and constants storage. Code is generated to:

1. Obtain the location of the FCIB
2. Generate instructions to transfer around the rest of the code generated for OPEN if the file is already open
3. If the TITLE pointer is null, generate code to move the filename into the title field of the FCIB. Otherwise, call the Expression Processor Controller ($NEXP) to generate code to evaluate and move the title expression into the title field of the FCIB.
4. Generate code to call the library runtime Open routine (IHEOPEN).

Except for PUT DATA and GET DATA statements, processing of a GET or PUT statement causes the following object code to be generated:

1. A call to an initialization routine
2. Instructions to put the location of a data element descriptor (DED) and the variable into the file control block (FCB). If the I/O type is EDIT, instructions to put the location of a format element descriptor (FED) into the FCB are also generated.
3. A call to a routine which reads or writes a data item.

Steps 2 and 3 are repeated for each item in the data list. After all items have been processed, code to call a termination routine may be generated. The DED's are generated at compile time and are moved into the static and constants area when phase 2 of the compiler processes the initialization table. If possible, the same procedure is used for the FED's. If an FED is too complex to evaluate at compile time, instructions to evaluate it at runtime will be generated.

If the statement is a GET DATA or PUT DATA statement, a symbol table is built; one item is entered for each item in the data list. (See Appendix E.) If no data list is present, a symbol table entry for each identifier in the block will be generated at epilogue time. A symbol table entry consists of a DED and the EBCDIC name for an item. In addition, code is generated to call the appropriate library runtime routine(s).

Both the GET Generator and PUT Generator call the I/O Specification routine ($DIOS), which scans a statement and locates FILE, SKIP, DATA, LIST, and EDIT keywords. These generators also call the Data Specification routine ($DDS) to process the data list for every statement other than a GET DATA statement. For all statements except GET DATA and PUT DATA, each item in the data list is processed as follows:

1. A DED is constructed.
2. Code is generated to put the location of the DED into the FCB.

3. Unless the I/O specification is EDIT, a call to the proper library runtime routine is generated. If the I/O specification is EDIT, a branch and link is generated to part of the code generated to process the format list of the statement. This code will put the location of the proper FED in the FCB and call the proper library runtime routine.

These three steps will be repeated until the data list is exhausted. If the statement is PUT DATA, an entry will be made in a symbol table for each item in the data list and a call to either IHEDDOA or IHEDDOB of the Data-Directed Output routine will be generated for each symbol table.

The $FLG, $FMT, $FORI, and $FPDL routines handle the format lists for edit-directed I/O. $FPDL is called by the GET Generator ($BGET) or PUT Generator ($BPUT) to process a format list for edit-directed I/O. The processing of a format list will cause the generation of code (to be called as if it were a subroutine). Therefore, the first function of $FPDL is to generate a branch around the code for the individual format items. This branch will be directed to the beginning of the code which will be generated to process the data list. Then, $FLG is called to direct the generation of code for the individual format items. On return from $FLG, $FPDL generates a branch back to the code generated for the first format item. At object time, this branch will be executed if the number of scalar data items in the data list is greater than the number of associated items in the format list. Then, $FPDL resolves the branch around the code for the format list.

$FLG performs the syntax analysis and code generation for a format list. For each format item, it generates the following code:

1. Save the location from where called.

2. Put the location of the FED into the FCB.

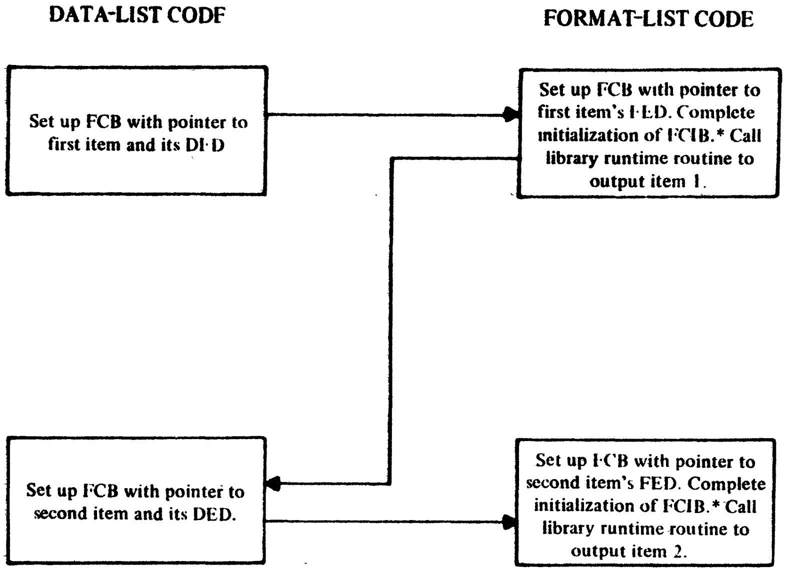
3. Call the appropriate library runtime routine.

4. Return to the location from which called with a branch and link instruction. This location will contain code generated for the next item in the data list.

$FLG calls $FORI to analyze and generate any code necessary for expressions in a format specification and to create a FED.

$FMT is not called by any I/O statement processing routine. When the Phase 1 Initializer ($CCONT) encounters a FORMAT keyword, $FMT is called to create a remote format list. The generated code is similar to that generated for a format list in a GET or PUT statement. Like $FPDL, $FMT calls $FLG to process the format list.

The code generated for a format list is driven by the code generated for the data list as illustrated in Figure 2-9. In this example, the data list contains two items and the format list has two entries describing these items.



*Note: This code will not be generated if the fields describing the format item are simple constants (for example, (5,2)).

Figure 2-9. Relationship of the Code Generated for the Data List and Format List

EXPRESSION PROCESSING

Certain common syntactic units are processed by the routines grouped as expression processing routines. Generally, the function of expression processing is to generate triads from syntactic units which are represented by tokens. Expression processing is invoked by calling the Expression Processor Controller ($NEXP). A call is generated in any of the following cases:

1. By a statement processor to evaluate an expression. For example, the following statement contains an expression to be evaluated:

   ```
   IF A<1 THEN GO TO KFILE;
   ```

   $NEXP is called by the IF Generator to evaluate the expression A<1.

2. By the CALL Generator ($CALL) to process the CALL statement entry name and the argument list.

3. By the Assignment Generator ($ACGEN) to process the entire assignment statement. An example of such a statement is:

   ```
   A = (B*(C+D) + MAX(A,B))/16;
   ```

The routines concerned with expression processing are:

Argument Operand Processor ($NATTP)
CALL Generator ($NCALL)
Cross-Section Dope Vector Build ($NCSDV)
Expression Processor Controller ($NEXP)
Dimension Multiplier Generator ($NMULT)
Convert Operand ($NOPCV)
Operator Stack Processor ($NOPRT)
Operand Set-Up ($NPRE)
Generate Triad ($TRIAD, $STRD)

Expression processors operate on the input tokens and generate triads. The processed tokens are the unarranged representation of the source stream and consist of operator and operand tokens. Two operands and an operator form a triad. An operand, itself, can be another triad.

The two primary routines of expression processing are the Expression Processor Controller ($NEXP) and the Operator Stack Processor ($NOPRT). $NEXP processes the input token stream and makes entries to the expression, operator, and operand stacks. $NOPRT operates on these stacks and creates triads. The items in these stacks are processed on a LIFO (last in, first out) basis. Figures 2-10 and 2-11 are simplified flow diagrams of $NEXP and $NOPRT.

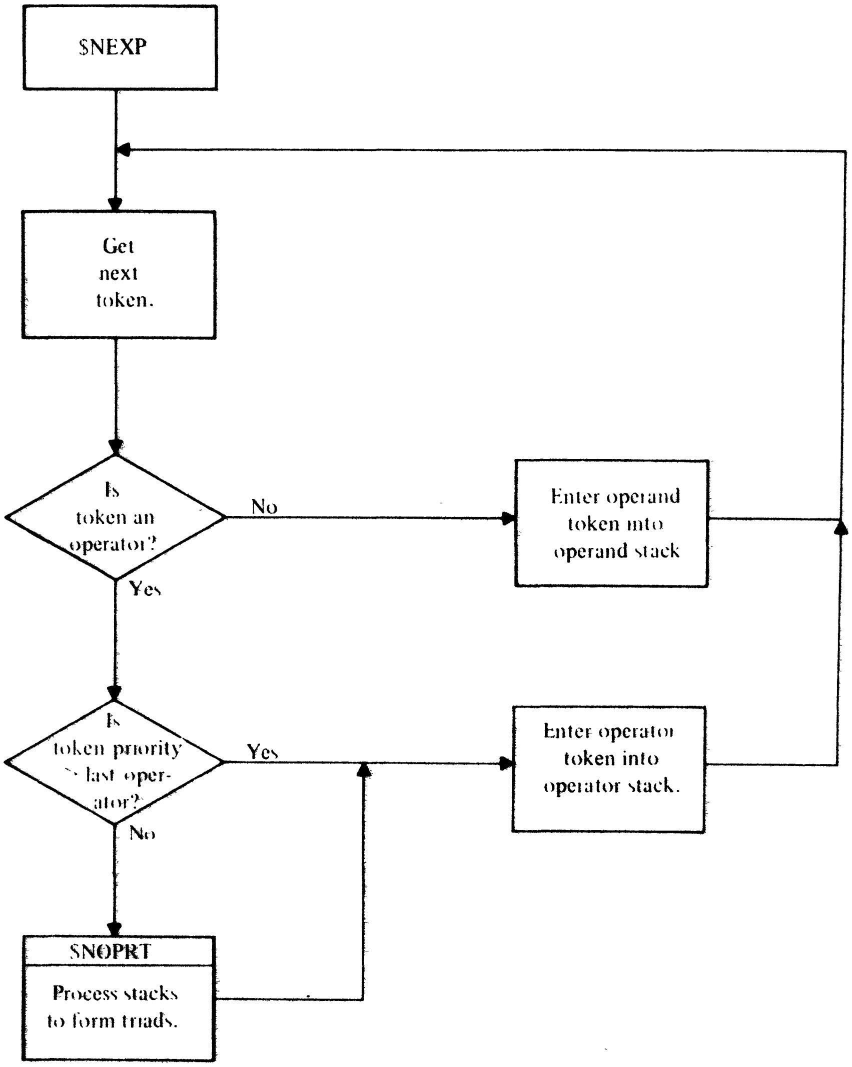


Figure 2-10. Simplified Logic Diagram of the Expression Processor Controller ($NEXP)

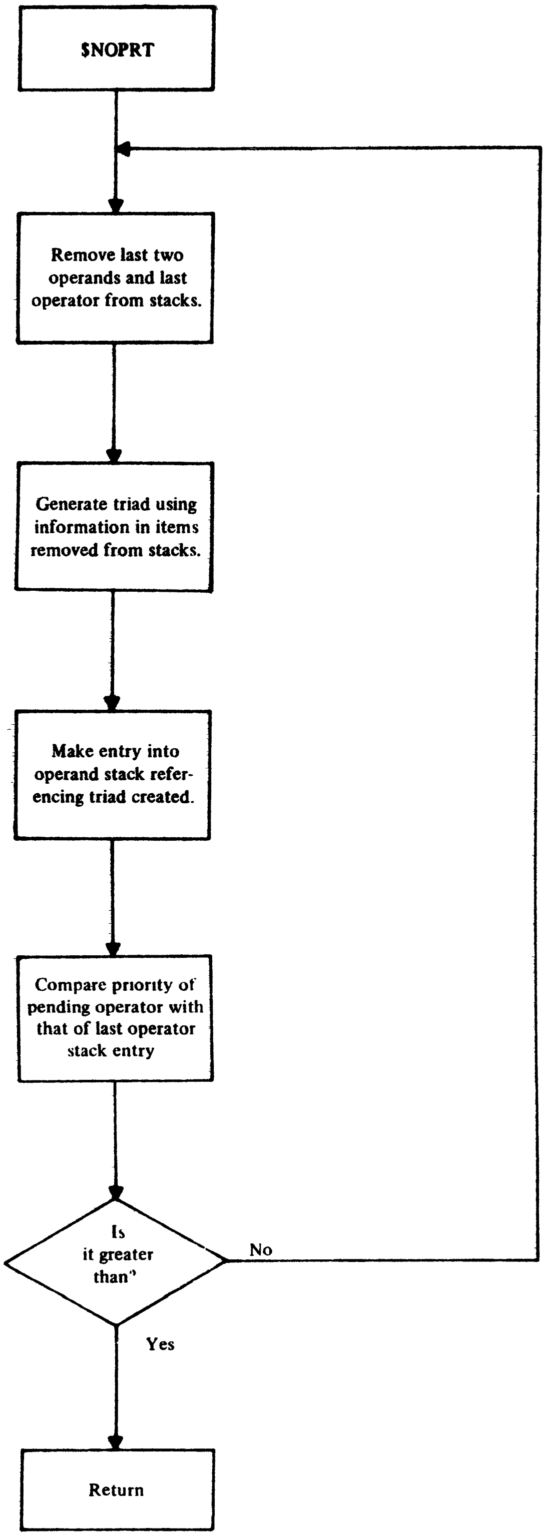


Figure 2-11. Simplified Logic Diagram of the Operator Stack Processor ($NOPRT)

To analyze the functioning of these routines, consider the following assignment statement:

A = 5*(B + C - D);

At entry to expression processing, each of the syntactic units has been converted to a token (A, B, C, and D, to operand tokens; all others, to operator tokens). The operator tokens have the following priorities:

Token	Priority
=	01
*	09
(	13
+	18
-	18
)	12
;	00

These priorities are expressed as two-digit numbers. The first digit represents the parenthesis level; the second represents the priority contribution of the operator. The order of evaluation is dependent upon the operator order in the expression and the operator priority. Operators and operands are added to their respective stacks until an operator having priority equal to or less than the priority of the previous operator stack entry is encountered. At this point, a triad is formed from the operator and operand stacks. In the case of a simple expression (such as the above example), the triad is formed from the last entry in the operator stack and the last two entries in the operand stack.

Note: The entries to the operator and operand stacks are an expanded form of the original tokens. See the discussion of these stacks in Appendix B.

Expression processing proceeds as follows.

1. Control is acquired by $NEXP with the token pointer at A. After initialization, the contents of the expression, operator, and operand stacks and the triad table are:

Expression Stack (V Table)	Operator Stack (X Table)	Operand Stack (Y Table)	Triad Table (Z Table)
Assign	Null	Empty	Empty

2. After $NEXP processes tokens A to *, the tables contain:

Expression Stack (V Table)	Operator Stack (X Table)	Operand Stack (Y Table)	Triad Table (Z Table)
Assign	Null	A	Empty
	=	5	
	*		

3. The left parenthesis generates an expression entry in the expression stack and a left parenthesis entry in the operator stack. After $NEXP has processed token C, the tables contain:

Expression Stack (V Table)	Operator Stack (X Table)	Operand Stack (Y Table)	Triad Table (Z Table)
Assign	Null	A	Empty
Expression	=	5	
	*	B	
	(	C	
	+		

4. Up to this point, no triads have been developed because the operators encountered so far are in order of increasing priority. The next token is a minus sign, which is equal in priority to a plus sign. This causes $NEXP to call $NOPRT. $NOPRT removes one operator and two operands from their respective tables and uses them to generate the triad B + C. A reference to this triad is placed in the operand stack. At this point, the tables contain:

Expression Stack (V Table)	Operator Stack (X Table)	Operand Stack (Y Table)	Triad Table (Z Table)
Assign	Null	A	T1
Expression	=	5	
	*	T1	
	(		
	-		

where T1 and similar entries below represent triads.

5. When the right parenthesis is encountered, triads are developed until the preceding left parenthesis is removed. Then, the tables contain:

Expression Stack (V Table)	Operator Stack (X Table)	Operand Stack (Y Table)	Triad Table (Z Table)
Assign	Null	A	T1
	=	5	T2
	*	T2	

6. The semicolon causes triad generation to be completed. The expression, operator, and operand stacks are now irrelevant. The triad table contains two additional entries; its contents are shown below:

Triad Table
(Z Table)

Triad	Left Operand	Operation	Right Operand
T1	B	+	C
T2	T1	-	D
T3	5	*	T2
T4	A	=	T3

Control will be returned to the calling routine (in this case, the Assignment Generator ($ACGEN)). Then code generation will be invoked by calling the Triad Code Generator ($TCODE). Object code will be generated by processing the triad table.

The triads generated in the previous example assume that neither conversion nor scaling is necessary. If conversions or scaling are required, additional triads have to be generated to perform these operations. Scaling and/or conversions are necessary when attributes of the operands differ. In the example above, if B were floating-point and C were fixed-point, C would be converted to floating-point before it was added to B. Situations in which conversions and scaling are performed are discussed in detail in the CALL/360-OS PL/I Language Reference Manual.

Other routines are also involved in expression processing. The Argument Operand Processor ($NATTP) is called by $NOPRT when processing the arguments of a CALL statement or a function reference. It extracts the attributes of these arguments. If an attribute of an argument does not agree with that of the corresponding parameter, $NOPRT calls the Convert Operand routine ($NOPCV) to perform the conversion, if possible.

The Call Generator ($NCALL) is called to generate the triads needed to call a function or subprogram.

The Cross-Section Dope Vector Build routine ($NCSDV) is called by $NEXP if an array cross-section is used as an argument of a CALL statement or a function reference. This routine builds a dope vector for the array cross-section.

The Dimension Multiplier Generator ($NMULT) is called by $NOPRT. It generates the triad required to multiply a subscript value by the dimension multiplier associated with the current array dimension. Thus, it performs part of the computation necessary to locate an array element. (See "Dope Vectors" in Appendix D for a description of the algorithm used to locate an element of an array.)

The Convert Operand routine ($NOPCV) is called to convert or scale an operand. If the argument is a source constant and the desired conversion is arithmetic to arithmetic, its attribute entry is changed to indicate the desired attributes. Later, when $NCONS is called (by $NOPRT, $NEXP, or $TRIAD), this constant will be converted to the desired form. Thus, no object code need be generated to perform the conversion or scaling. If the operand is not such a source constant, any triads needed to achieve the desired type and scale are generated.

The Operand Set-Up routine ($NPRE) is called by $NEXP or $NOPRT to move the top entry of the operand stack to an operand area. It then obtains the type, sign, scale, precision, and length attributes of the operand and moves them into this operand area. When the calling routine is $NEXP, these attributes represent the result operand. For all other types of operands, the calling routine is $NOPRT.

The Generate Triad routine ($TRIAD, $STRD) is called to generate a single entry in the triad table. The calling routine may be one of the expression processing routines ($NOPRT, $NOPCV, $NMULT, $NEXP, or $NCALL) or certain of the general statement processor routines.

Expression processing may be considered from another aspect, namely, that of the tables with which it interfaces. This relationship is shown in Figure 2-12. All illustrated tables have been mentioned except the library load table (L table) and the constant table (C table). Entries are made in the library load table if a call to a runtime library subroutine is generated. (Such calls do not, of course, include calls of and references to internal procedures.) Entries may be made into the constant table when the Constant Processor ($NCONS) is called to direct conversion of a source constant to a desired form, if necessary. If the converted form is not already in the constant table, it will be entered. (The tables are explained in detail in Appendix B.)

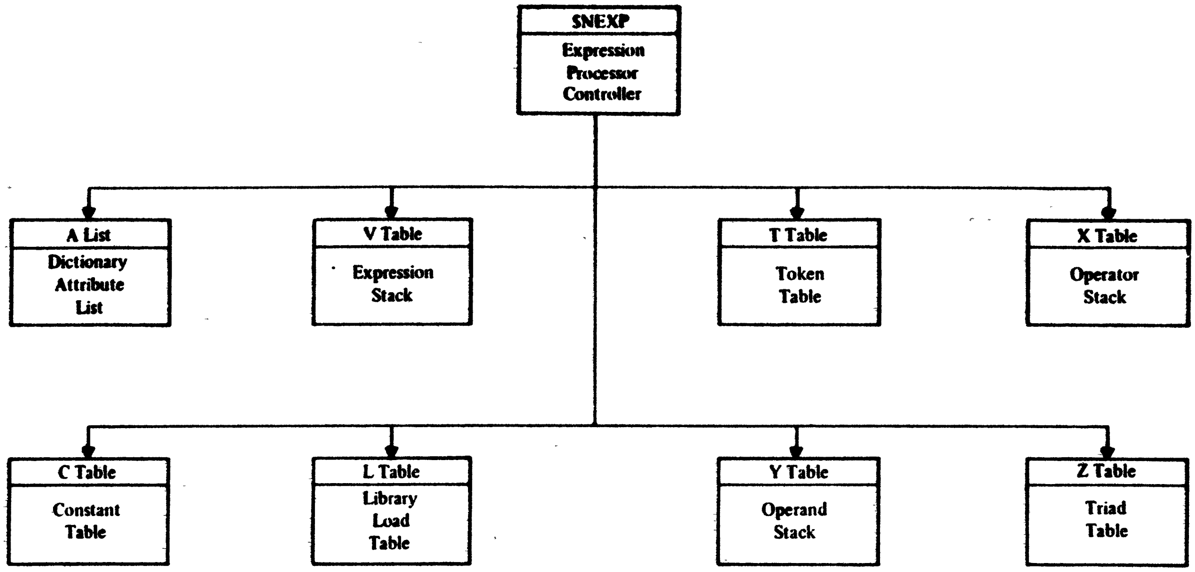


Figure 2-12. Expression Processor Tables

CODE GENERATION

The function of code generation is to produce final object code from the triads produced by expression processing and/or the statement processors. Code generation is invoked by the Phase 1 Initializer ($CCONT) after the triads for a statement (or, in certain cases, part of a statement) have been generated. Figure 2-13 illustrates the relationship between the code generation modules.

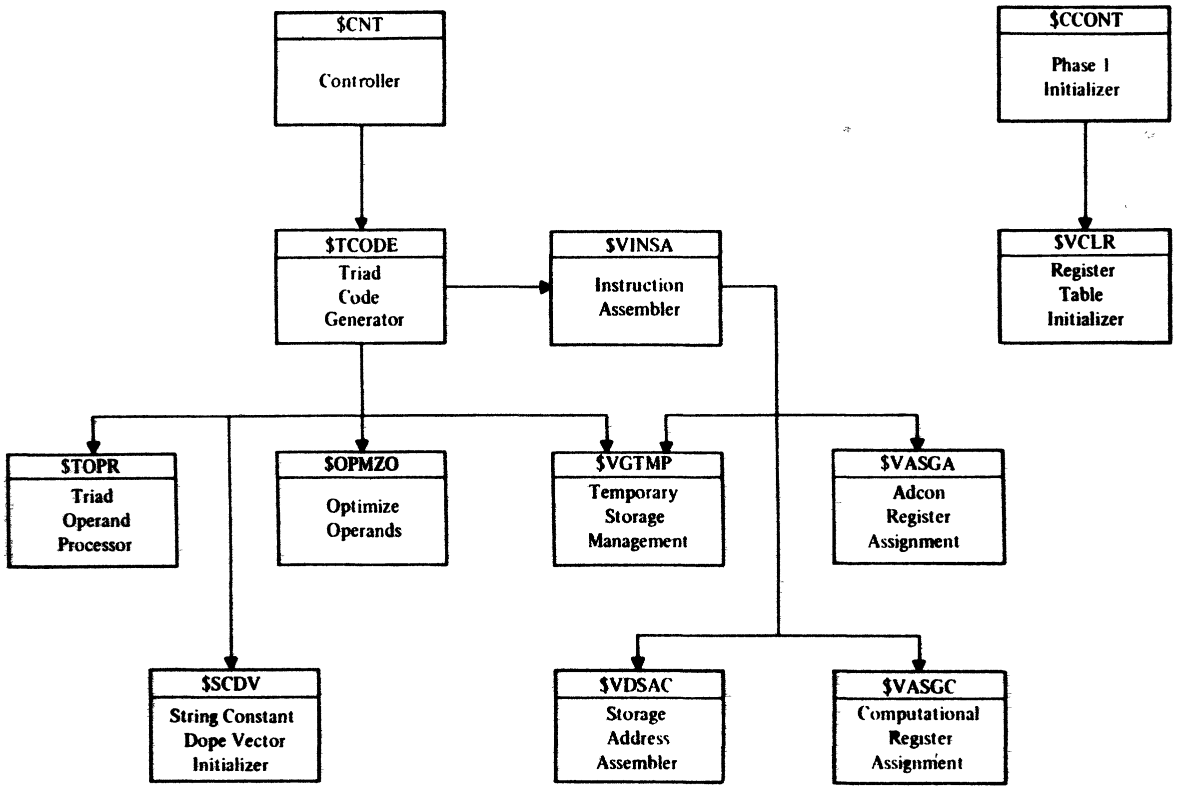


Figure 2-13. Code Generation

The two primary routines of code generation are Triad Code Generator ($TCODE) and Instruction Assembler ($VINSA). Code generation is invoked by calling $TCODE which processes the triad table entries in order, starting with the first entry and continuing until all triads have been processed. Figure 2-14 shows the tables that are referred to by the $TCODE module. In the illustration, those tables connected by broken lines are actually assembled as part of $TCODE.

The Triad Operand Processor ($TOPR) is called to put information describing the two operands of a triad into the data parameter table ($PTO). (For a description of this table, see "Compiler Variables" in Appendix A.) The triad operator directs a branch to the section of $TCODE which controls its processing. In general, this processing consists of:

1. Determining the sign of the operands and the sign of the result of execution of the triad code. Signs may be adjusted to optimize the required instruction sequence.

2. Arranging the operands, when possible, to insure optimum code generation.

3. Selecting the set of symbolic instructions in the symbolic instruction table (M table) to be used by $VINSA in generating the code for the triad. As shown in Figure 2-14, the symbolic instruction table is part of the $TCODE module. (For its description, see Appendix B.)

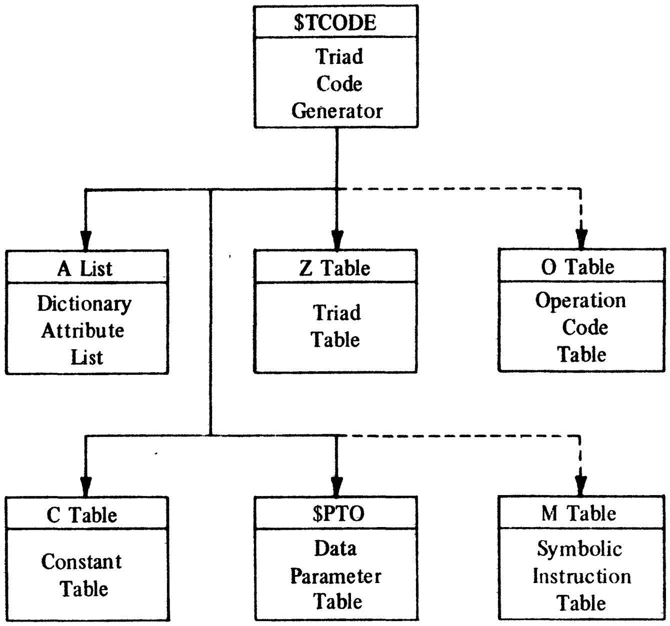


Figure 2-14. Tables Used by $TCODE

$TCODE then calls $VINSA, passing to it the location of the selected set of symbolic instructions. These symbolic instructions represent, in a general way, the object code which will be produced for that triad. The function of $VINSA is to interpret, modify, and, in some cases, optimize these general instructions into specific, machine-language instructions. The general processing performed by $VINSA for most instructions is:

1. Using the information passed about the operands and the operation to be performed, change the operation code, if it is modifiable, to the System/360 machine code for that instruction. For example, if LOAD were the general operation passed, it would be changed to the System/360 op code L, LE, or LD (depending on whether the referenced item was fixed point, short float, or long float).

2. Generating the remainder of the instruction.

Besides operations that conform closely to machine operations, $VINSA processes a class of instructions that involve pseudo-operations. Examples of pseudo-operations are:

1. Load Complex - This pseudo-operation may generate code which will cause two registers to be loaded, one with the real part of a number and the other with the imaginary part of the number. The registers need not be loaded if the items are already in registers.

2. Save Volatile Registers - This pseudo-operation is used prior to the generation of calls to object-program library routines. It causes the generation of code to store fixed registers 0 and 1 and floating-point registers 0 and 2 (which are volatile across library calls) in temporary storage if they contain active intermediate results.

Figure 2-15 describes the relationship of the Instruction Assembler ($VINSA) and its tables and output.

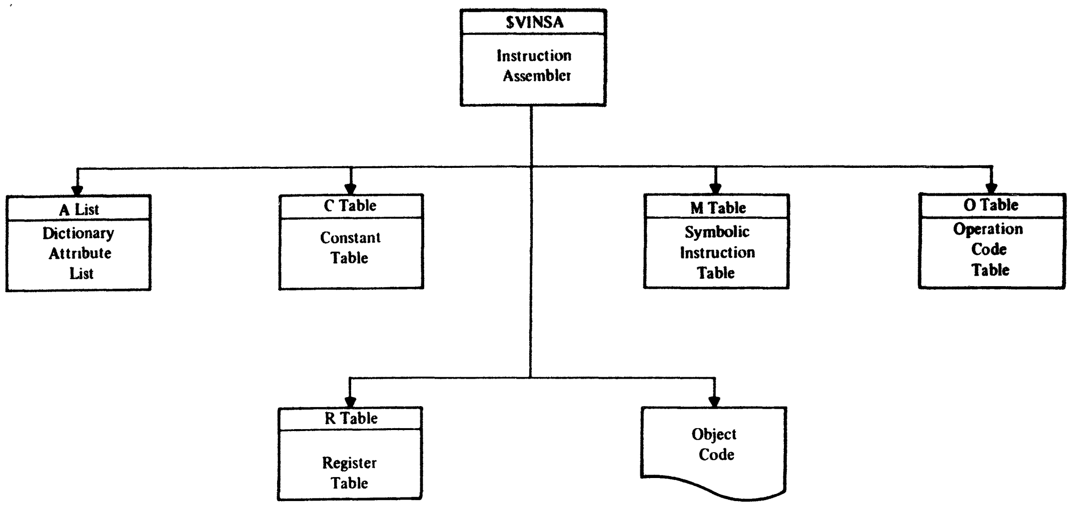


Figure 2-15. Tables of Instruction Assembler ($VINSA)

The Optimize Operands routine ($OPMZO) is called by $TCODE to determine whether either operand is in a register and, if so, under certain conditions, to interchange the operands. For example, assume that the operation to be performed is to add B to A, that B is in a register, and that A is not. The operation will be changed to add A to B, thereby eliminating the need to load A into a register (and B is already there).

The Triad Operand Processor ($TOPR) is called by $TCODE to put information concerning the two operands of a triad into the data parameter table ($PTO). This information includes the core address, core sign, register address, register sign, and length of the operand.

The Storage Address Assembler ($VDSAC) is called by $VINSA to convert the compiler's internal representation of a storage address (base code and displacement) into a machine address (base register and 12-bit displacement). If the storage address is not within 4096 bytes of any base register, instructions to set up a base register are also generated.

The Adcon Register Assignment routine ($VASGA) and the Computational Register Assignment routine ($VASGC) are used in register manipulation. Part of the initialization to run a job is to set up certain registers which are never modified during program execution (base registers for the object code and bases of frequently referenced areas). The remainder of the registers (adcon registers that are adjusted if the program is relocated and computational registers that are not) are set and modified as needed. $VASGA is called to obtain the least-used adcon register and $VASGC is called to obtain either a single or a double computational register. $VASGC makes its selection so that optimum code will be generated. For example, a free register will be chosen in preference to one that contains a value which may be needed later.

The Register Table Initializer ($VCLR) is called by the Phase 1 Initializer ($CCONT). It obtains and initializes an area for the register table. Only code generation routines use the register table. The Temporary Storage Management routine ($VGTMP) is called to obtain the base code and displacement of an area of temporary storage. The String Constant Dope Vector Initializer ($SCDV) is called by $TCODE to create a string dope vector when it encounters a call of the string to an arithmetic function which references a constant.

PHASE 2 OR WRAP-UP PHASE

After the END Generator ($EDGN) of phase 1 has determined that the last source statement has been processed, it calls the Executive to request initialization of phase 2. The Executive loads the phase 2 modules and the runtime library routines in the area formerly occupied by the phase 1 compiler routines. Then it transfers control to the Phase 2 Initializer ($WCONT). This routine sets up the adcons pertaining to the phase 2 compiler routines in the fixed area of the compiler work space (see Appendix A for a description of this area). Then it transfers control to the Compilation Wrap-Up Driver ($MCWU). $MCWU directs the wrap-up process. When wrap-up is complete, it calls the Executive to request execution. Figure 2-16 shows the relationship between the various wrap-up routines.

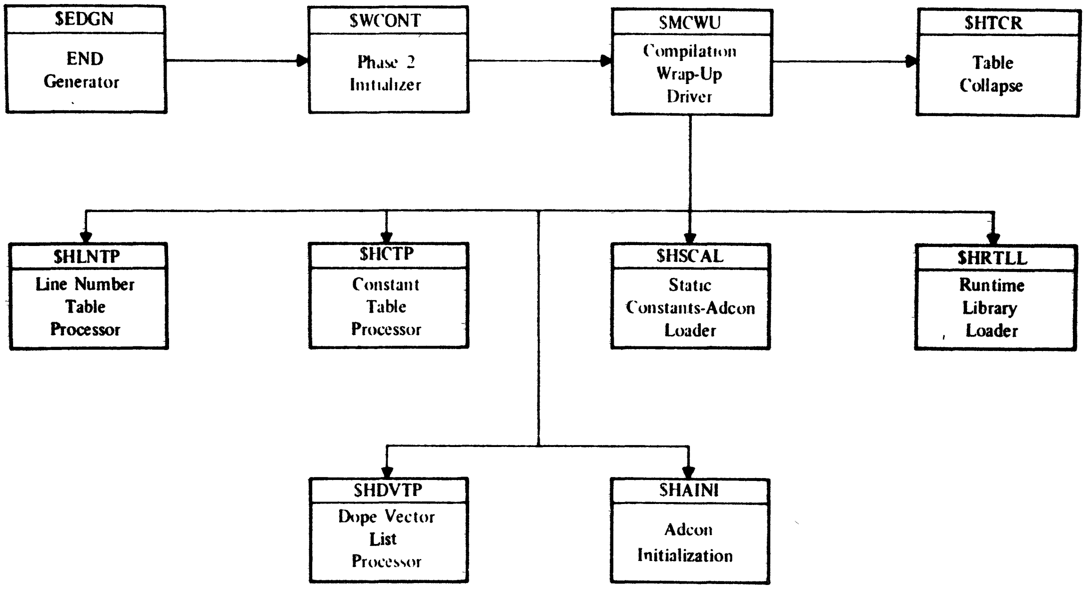


Figure 2-16. Wrap-Up Routines

The function of phase 1 was primarily to generate machine-language (object) code for CALL/360-OS PL/I source-program statements. The function of phase 2 is to set up the items referenced by the object code. Constants, library routines, and adcons are set by processing the tables generated during phase 1. Figure 2-17 shows a cumulative correspondence between the tables processed and the storage layout of the user's work space. The routines are discussed below in the order that they are called by $MCWU.

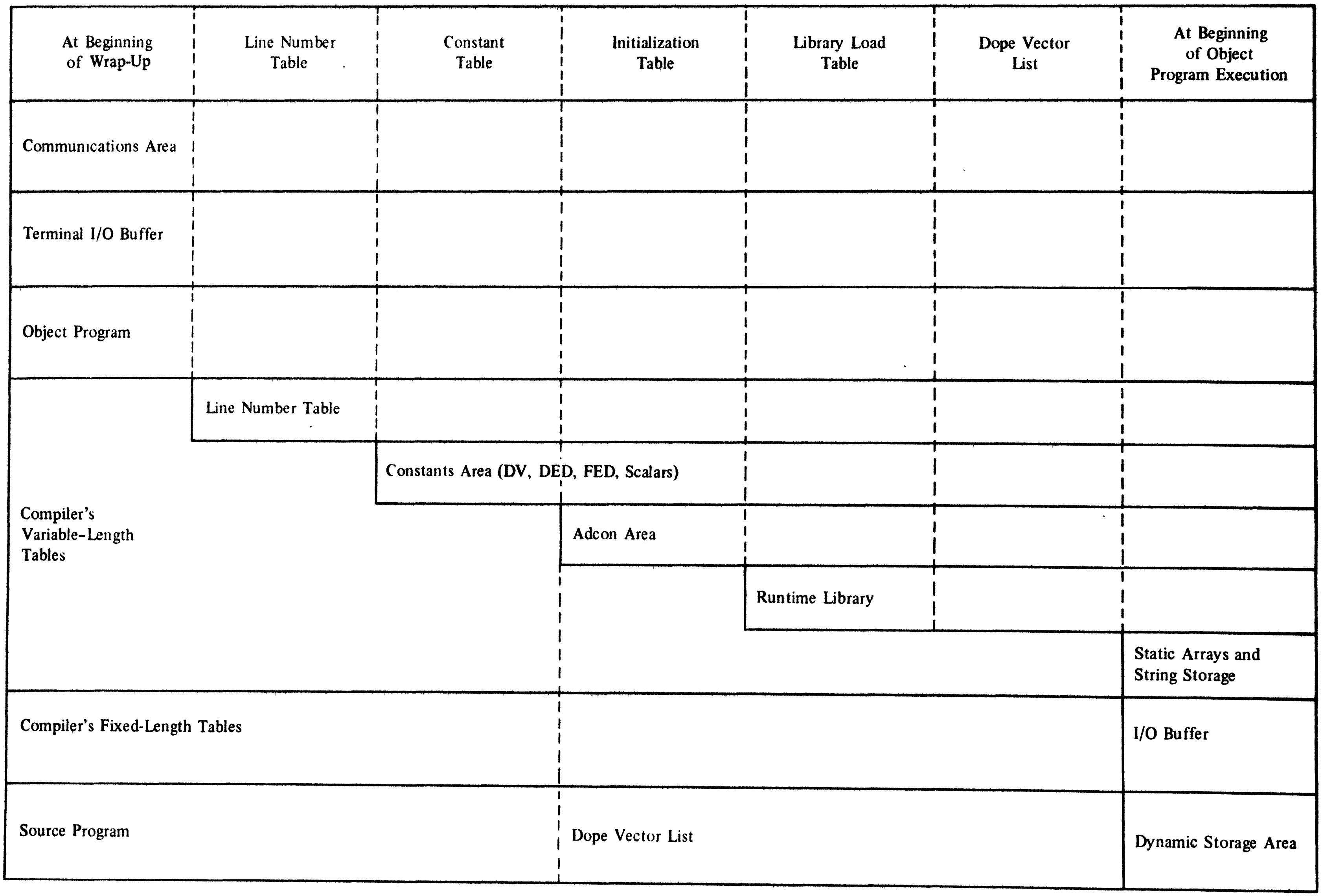


Figure 2-17. Storage Allocation During Compiler Wrap-Up

The Line Number Table Processor ($HLNTP) processes and moves the line number table (D table) adjacent to and immediately following the generated code. Each entry in the line number table correlates the number of a source statement with the object code displacement of that statement. This table is used for diagnostic and error message purposes.

The Constant Table Processor ($HCTP) processes the constant table (C table) and moves the constants to the static and constants area (adjacent to the line number table).

The Static Constants-Adcon Loader ($HSCAL) processes the initialization table (I table) and dope vector list (J list). Depending on the entry type, the item is entered in the adcon or static and constants area. If an item entered in static and constants storage is a dope vector that will reference an item in static array and string storage, an entry pointing to the dope vector is made in the dope vector list. Note that at this time the adcons are not true addresses, but merely pointers or displacements.

The Runtime Library Loader ($HRTLL) determines all library routines required by the program. These routines are moved from the compiler area to follow the adcons in the work area. Space is allocated in working storage for the work areas needed by the library runtime routines. The work areas are either relocatable or non-relocatable.

The Dope Vector List Processor ($HDVTP) processes the dope vectors referenced by the dope vector list. Its function is to compute the virtual location of a string or an array. (This could not be done earlier because the starting location of static array and string storage was unknown.)

The Adcon Initialization routine ($HAINI) converts all adcons from their base code and displacement form to true addresses and resets the user-area relocation constants in the communications area. To prevent the work area from being swapped during this conversion, the swap flag (SWPFLG) of the communications area is set.

The Table Collapse routine ($HTCR) is called if, at any time during the compiler wrap-up phase, prior to the loading of the runtime library, there is insufficient free space in the variable area. $HTCR collapses the C table, D table, I table, and J list. Collapsing consists of moving the unprocessed table or list entries into higher core storage (that is, into locations previously required for table or list entries that have been processed).

SUPPORT

The support routines provide services needed by many routines of the CALL/360-OS PL/I compiler. The three routines which handle error conditions are Array Expression Error ($AREXP), Compiler Error ($CERR), and Error Message Editor ($XERR). The $AREXP routine is of minor importance. It is called when an array expression is in an illegal position.

Control is passed to $CERR when an error so severe that compilation must be terminated is detected. All System/360 program interrupts are in this group, since no code in the compiler should cause program interrupt. When such an interrupt occurs, the Executive transfers to ARINTRP of the communications area. Code beginning at ARINTRP subsequently transfers to $CERR. In addition, numerous compiler modules transfer directly to $CERR when they detect a severe error necessitating compilation termination. $CERR prints out a message pointing to the error and terminates compilation.

$XERR is called for less serious errors. An error message number and one or more parameters are passed to $XERR. Using this information, $XERR prints out an error message identifying the error in a terminal user's program.

Other support routines perform vital functions. Output Director ($GPUT) is called to output a 120-character line to the terminal buffer. The Constant Processor ($NCONS, $NCON) is called when generated code must reference a constant. This routine enters the constant in the constant table and computes the location in static and constants storage that will contain this constant at runtime. Library Search ($NLSIB) is called when a statement references a runtime library routine. $NLSIB makes an entry in the library load table which indicates to the Runtime Library Loader ($HRTLL) that the referenced runtime library routine must be loaded. The Convert Constant routine ($NCVT) is called to convert a source constant to a required internal form.

The Segment Management routines ($WBACK, $WCTCT, $WEXP, and $WSTEP) provide the bookkeeping services necessary to support compiler macros which manipulate expandable table segments. In general, these macros address nodes within an expandable table segment. In advancing or stepping back from one node to the next, a table segment boundary may be exceeded. In such a case, the code generated by one of these macros will call the proper segment management routine to proceed from one table segment to another.

The Get Next Triad Entry routine ($GTRIAD) is called to get the next available entry in the triad table. Calling this routine serves the same function as issuing a GNODE macro addressing the triad table. Use of $GTRIAD is generally preferable because less space is required for the necessary code.

EXECUTIVE INTERFACE

Interface between the Executive of the CALL/360-OS system and the CALL/360-OS PL/I compiler is performed by means of the user terminal table (UTT), communications area, and the SVC Director module ($SVC). The Executive and the compiler communicate with one another by setting areas in the UTT and the communications area. The compiler directs requests to the Executive by calling $SVC.

SECTION 3 - CALL/360-OS PL/I COMPILER ROUTINE DIRECTORY

INTRODUCTION

The CALL/360-OS PL/I compiler is functionally divided into the following parts:

- Controllers
- Entokening
- General Statement Processors
- Declaration Processing
- I/O Statement Processing
- Expression Processing
- Code Generator
- Wrap-Up
- Support
- Executive Interface

Each CALL/360-OS PL/I compiler routine is described in this section under one of the headings listed above.

PART 1 - CONTROLLERS

The following routines are the controllers of the CALL/360-OS PL/I compiler. They are discussed in alphabetic order, according to their mnemonics, on succeeding pages. Detailed logic diagrams for the routines appear at the end of this subsection.

Statement Category ($CATEG)
Phase 1 Initializer ($CCONT)
Controller ($CNT)
Phase 2 Initializer ($WCONT)

TITLE: STATEMENT CATEGORY ($CATEG)

Program Definition

Purpose and Usage

The Statement Category routine determines whether or not the next statement in the token table is an assignment statement.

Description

This routine searches the token table up to the semicolon token for any of the following token combinations:

ident	ident
constant	ident
)	ident
keyword	ident

If any of these combinations occur, the statement is not an assignment.

If none of the above combinations occur at zero parenthesis level, and at least one equal sign does occur at this level, the statement is an assignment statement.

Errors Detected

None

Local Variables

None

Program Interface

Entry Points

$CATEG. Expects $PTR to be set to location to begin checking.

Exit Conditions

Normal exit only. Returns type of statement code in register G0; if greater than zero, assignment.

Routines Called

$CERR	Compiler Error

Global Variables

T Table	Token Table
$PTR	Token Table Pointer

Logic Diagram

Chart 1 shows the detailed logic diagram for the Statement Category routine.

TITLE: PHASE 1 INITIALIZER ($CCONT)

Program Definition

Purpose and Usage

The Phase 1 Initializer receives control from the CALL/360-OS Executive and initializes everything required to begin compilation.

Description

Upon receiving control from the CALL/360-OS Executive, $CCONT locates the user's work space allocated for the compilation and establishes registers to address it. The lengths of the compiler's relocatable data area, and the addresses of the compiler's relocatable registers are communicated to the CALL/360-OS Executive during the first few instructions, which are guaranteed not to be interrupted.

Instructions are established to receive control of arithmetic interrupts. Then the values in the adcon area which address entry points in the compiler code are relocated by the value of the compiler origin. Initial values are moved into all non-relocatable global variables which require them.

The first segments of all expandable tables are now constructed, and the necessary control pointers are initialized for them. In conjunction with this activity, the first node of the expression stack is obtained, and the first node in the constant table is cleared to zero. The names of the Error Message Editor and SVC Director routines are then entered into the library load table. These two object program library routines are always present, regardless of whether they are directly invoked or not.

Attribute nodes and name list entries are next created for all built-in function names and keywords. The operator stack is reset, and the register table is constructed and initialized. Control then passes to the Controller to initiate the compilation process.

Errors Detected

None

Local Variables

None

Program Interface

Entry Points

Control is transferred to $CCONT via a special interface with the CALL/360-OS Executive, which branches to the first byte of the first-phase load module with the entry point address in register 7 and the address of the user's area in register 12.

Exit Conditions

$CCONT is effectively inline; control passes to the Controller ($CNT) and does not return.

Routines Called

$CNT	Controller
$NLSIB	Library Search
$VCLR	Register Table Initializer
$FIND	Search-Insert
$WEXP	Segment Management
$WCTCT	Segment Management

Global Variables

All global variables requiring initial values and tables are initialized.

Logic Diagram

Chart 2 shows the detailed logic diagram for the Phase 1 Initializer routine.

TITLE: CONTROLLER ($CNT)

Program Definition

Purpose and Usage

The Controller directs the entokening of statements, analyzes most label prefixes, handles detection of compound statement units, and directs control to statement processors.

Description

The Controller, upon initial entry, calls the Entoken routine at its second entry point so that it can initialize itself and then entoken the first statement. A check is made to see if the first statement is an external procedure statement. If not, one is supplied.

Each statement in the program is checked for the presence of a label. If one is present, the label switch ($CLBLS) is set ON, a pointer ($CLPTR) is set to the label, and the token table pointer ($PTR) is advanced to the token following the label.

The status of the compound statement switch ($CSS) is then checked to see if any units have just been terminated. The Controller uses the status of $CSS to determine the program structure caused by IF and ON statements. $CSS is a four-position switch: OFF, THEN, ELSE, and ON.

If a THEN unit has just been terminated and no ELSE unit is present, the branch around the ELSE unit is not generated. Otherwise, processing continues as at the beginning of a THEN unit. The word THEN or ELSE is skipped, and it is determined whether or not a label is present on the unit. The unit is then checked for legality. The following statements are allowable in a THEN or ELSE unit:

- assignment
- BEGIN
- CALL
- DO
- GET
- GO TO
- IF
- null
- ON
- PUT
- RETURN
- REVERT
- STOP

The Controller uses the status of a first executable statement to determine whether a DECLARE statement precedes the first executable statement of the block. If it does not, the statement must be taken out of line and added to the prologue code chain. If it does, the statement can be executed in order with no need for branches linking it to a chain. This optimization requires that immediately before generating code for the first executable statement, the prologue code chain is started and the end of prologue instructions generated.

The syntactic and semantic analysis of the statements is performed by the individual statement generators, which also generate the triads required for the statements. The Controller determines to which statement generator to transfer control.

If the first token of the statement is not a statement keyword, the statement must be an assignment. If it is a statement keyword, Statement Category subroutine ($CATEG) is called to determine whether it is an assignment. If it is not an assignment statement, the type of statement is determined by the first token of the statement. Since all statement generators expect $PTR to point to the first token in the statement body, it is advanced for nonassignment statements.

If the statement is a DECLARE, FORMAT, END, PROCEDURE, or BEGIN, the statement generator processes the label. Otherwise, the Controller calls the Label Processor subroutine ($BLPRC) to define the label on the statement.

Upon return from the ON Generator ($CON), it is necessary to determine whether the on-unit statement is legal. (The ON Generator made sure it does not have a label.) The legal statements are the following:

assignment	GO TO	REVERT
CALL	null	STOP
GET	PUT	BEGIN

Errors Detected

PROCEDURE STATEMENT SUPPLIED (1)
ILLEGAL '___' STATEMENT--NULL CLAUSE SUPPLIED (2)
'___' NOT STATEMENT TYPE, IGNORED (3)
NOT STATEMENT TYPE--ASSIGNMENT ASSUMED (4)
MAXIMUM NO. OF BLOCKS EXCEEDED (24)

Local Variables

None

Program Interface

Entry Points

One entry point from the Phase 1 Initializer ($CCONT).

Exit Conditions

No exit. At end of compilation, END Generator ($EDGN) directs transition to phase 2.

Routines Called

$TCODE	Triad Code Generator
$GTRIAD	Get Next Triad Entry
$ATKN	Entoken
$ENDON	Process End of ON
$ENDES	Process End of ELSE
$CATEG	Statement Category
$BLPRC	Label Processor
$FIND	Search-Insert
$WEXP	Segment Management
$XERR	Error Message Editor
	All Statement Generators

Global Variables

$CSS	Compound Statement Switch
$CLBLS	Label Switch
$EOS	End of Source Switch
T Table	Token Table
$PTR	Token Table Pointer
$CLPTR	Label Pointer

Logic Diagram

Chart 3 shows the detailed logic diagram for the Controller routine.

TITLE: PHASE 2 INITIALIZER ($WCONT)

Program Definition

Purpose and Usage

The Phase 2 Initializer initiates the second or wrap-up phase of the compiler.

Description

$WCONT begins by relocating the values of all adcons for the entry points of compiler routines contained in the second phase. It then tests the highest severity code produced by diagnostic messages during the first phase of compilation. If fatal errors have occurred, wrap-up and execution are prevented, and control returns to the Executive via an SVC. If execution is to be allowed, the Compilation Wrap-Up Driver routine ($MCWU) is called to complete loading and initiate execution.

Errors Detected

None

Local Variables

None

Program Interface

Entry Points

Control is transferred to $WCONT via a special SVC interface with the CALL/360-OS Executive, which branches to the first byte of the second-phase load module. Register contents are the same as at the end of the first phase.

Exit Conditions

$WCONT is inline; it passes control to the Compilation Wrap-Up Driver routine ($MCWU) and does not return.

Routines Called

$SVC	SVC Director
$MCWU	Compilation Wrap-Up Driver
$ATKN	Entoken
$GPUT	Output Director
$FNB	Get Non-Blank
$CERR	Compiler Error
$WSTEP	Segment Management

Global Variables

$SEVCO	Highest Severity Code
$SCNX	Scan Index
$PTR	Token Table Pointer
$BASE	Base Address of Compiler

The following adcons in the C-area of the communications area are the locations of the routines used in phase 2. The values in these locations are adjusted by $WCONT.

@AADUM
@AAINT
@ASIDX
@ATKN
@ATKN2
@CERR
@FIND
@FNB
@GPUT
@GTRA
@HAINI
@HCTP
@HDVTP
@HLNTP
@HRTLL
@HSCAL
@HTCR
@MCWU
@STEP
@SVC
@WCTCT
@WEXP

Logic Diagram

Chart 4 shows the detailed logic diagram for the Phase 2 Initializer routine.

PART 1 LOGIC DIAGRAMS

The detailed logic diagrams for the controllers of the CALL/360-OS PL/I compiler follow.

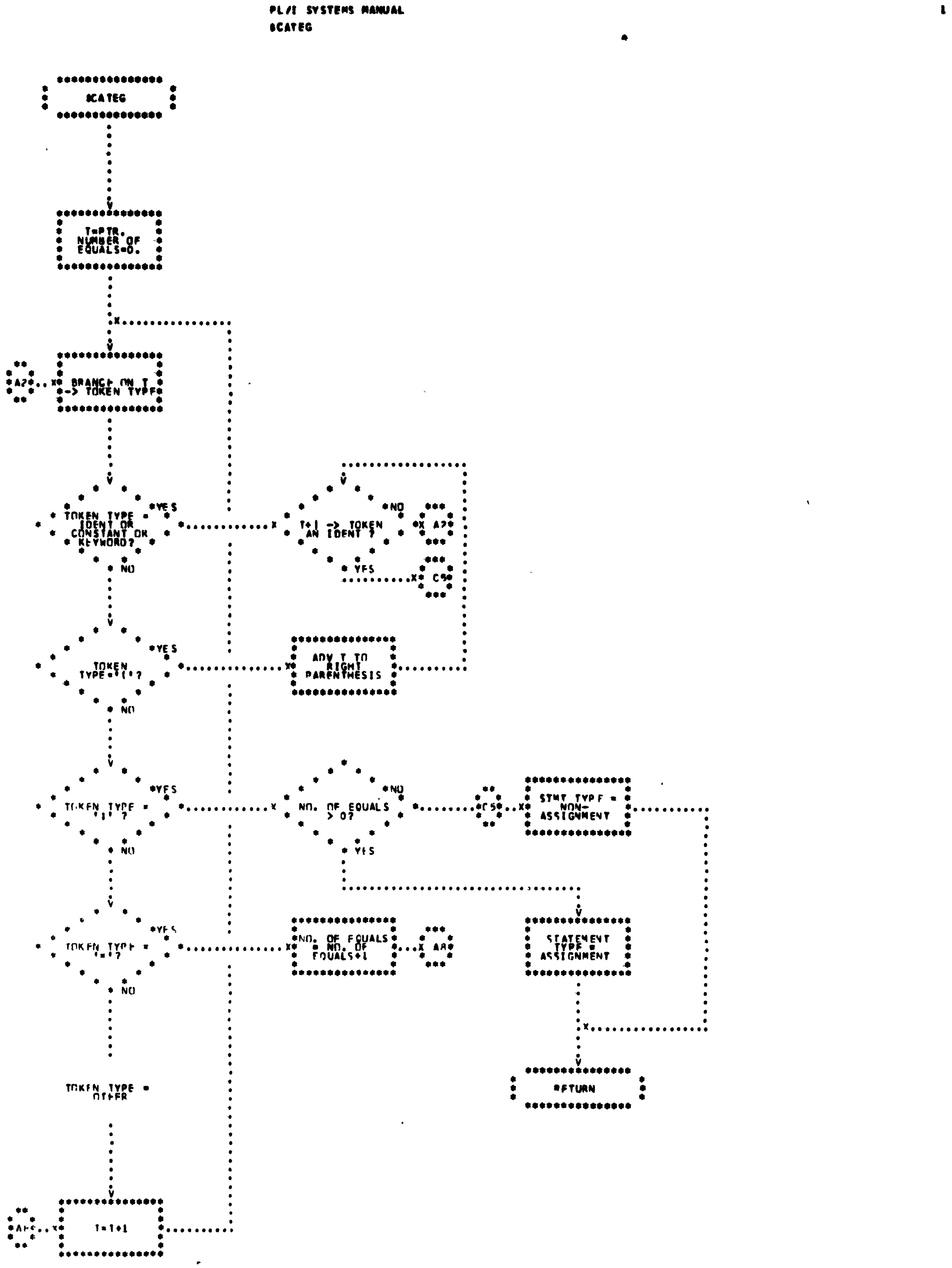


Chart 1. Statement Category

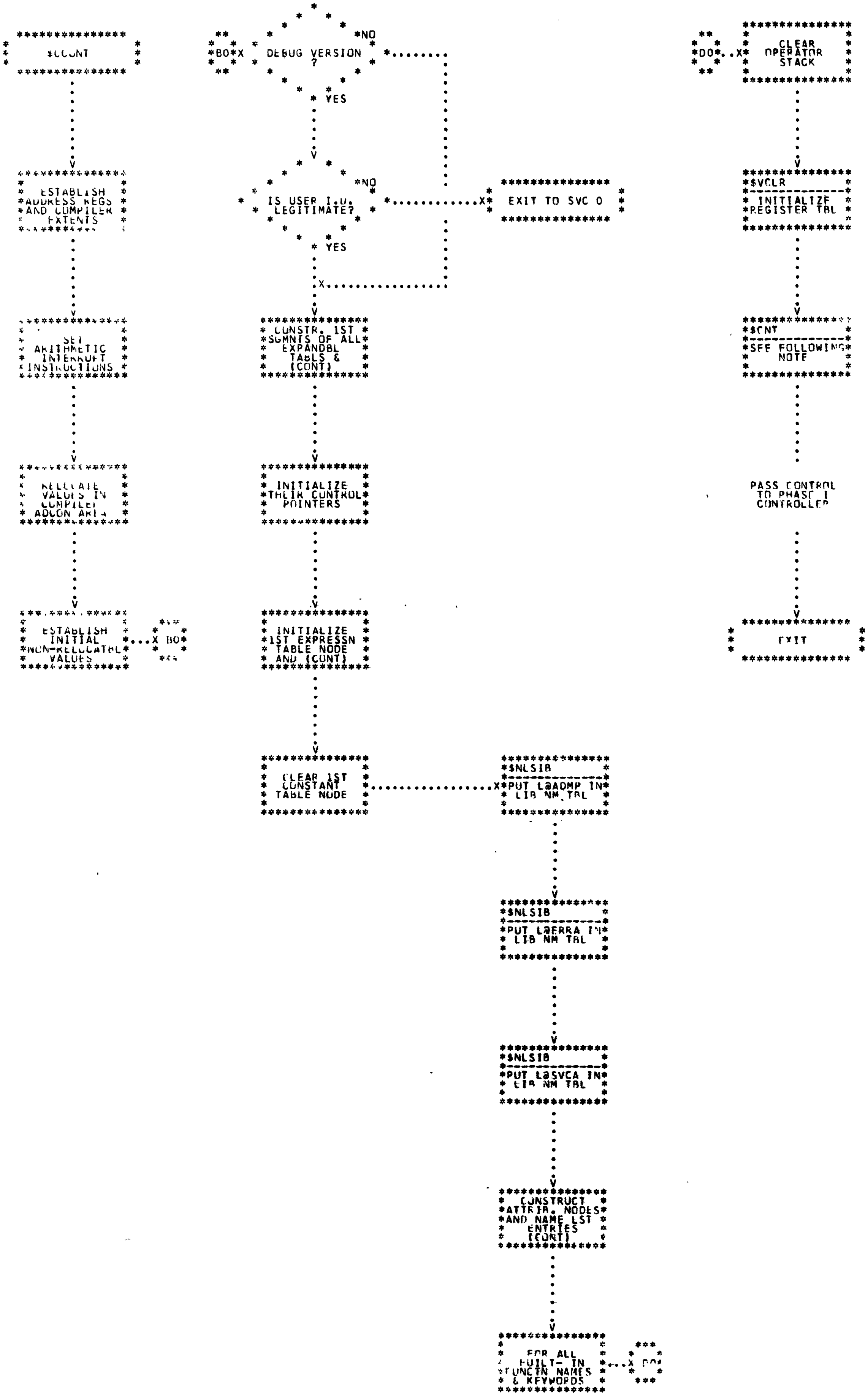


Chart 2. Phase 1 Initializer

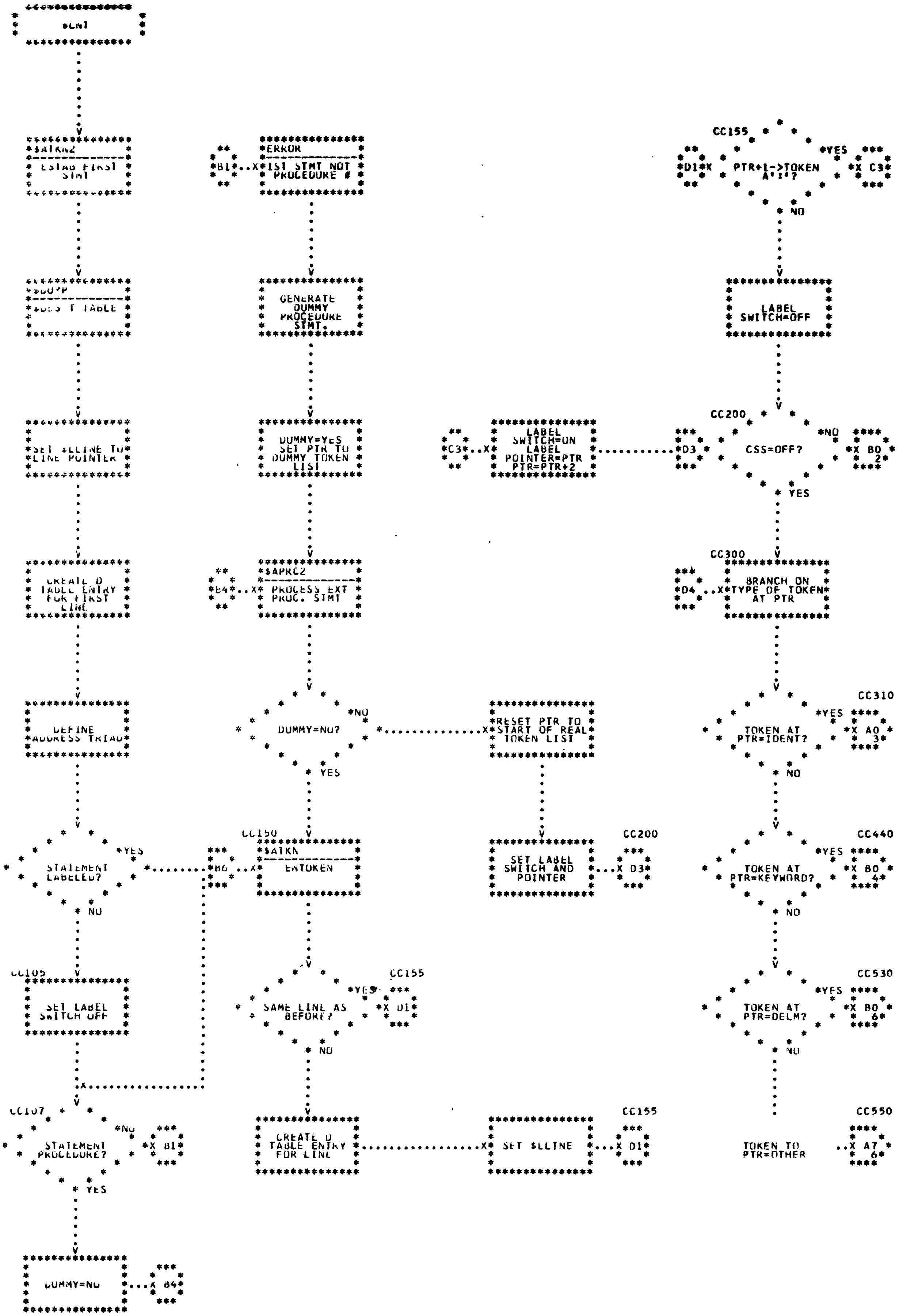


Chart 3. Controller (Page 1 of 6)

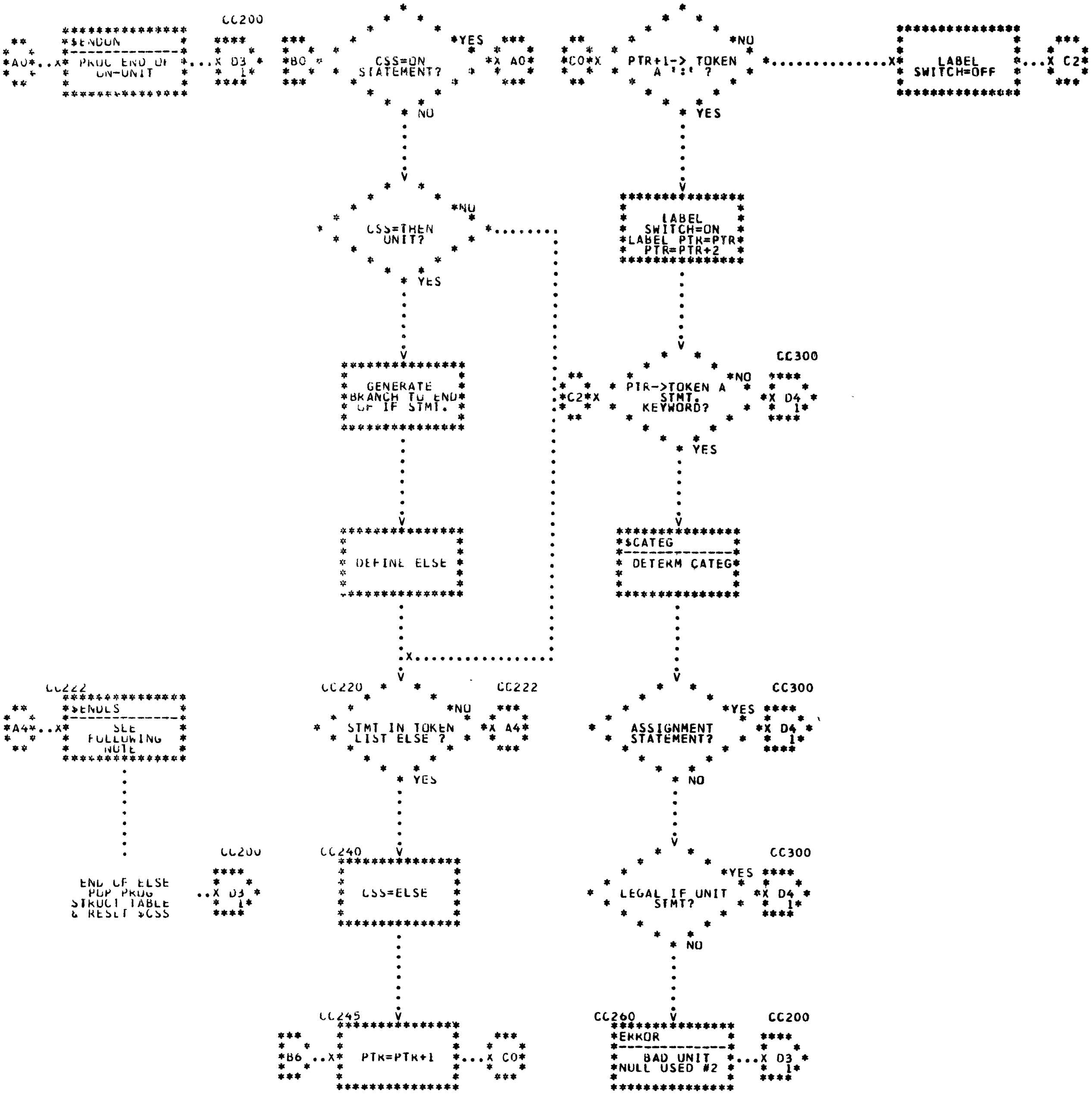


Chart 3. Controller (Page 2 of 6)

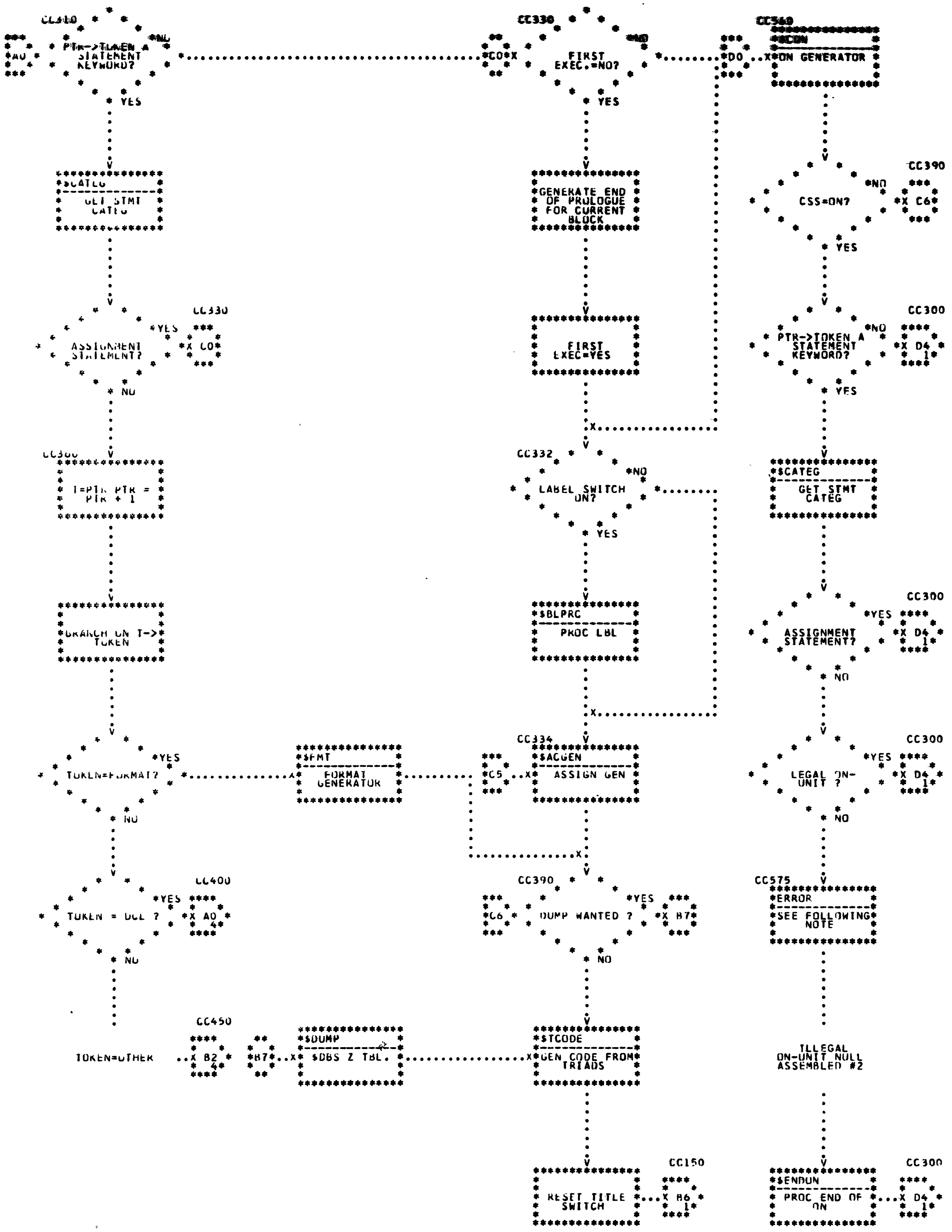


Chart 3. Controller (Page 3 of 6)

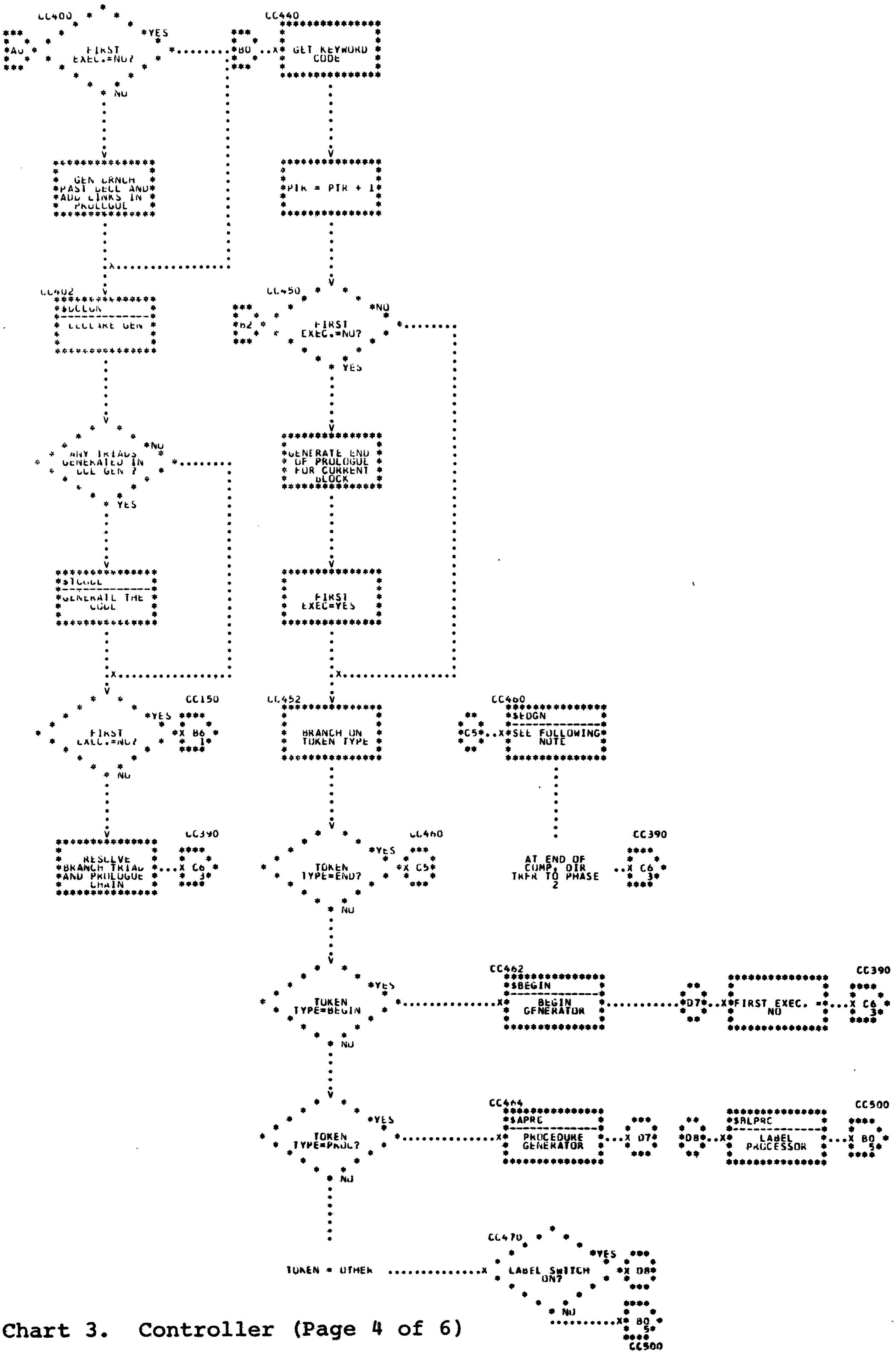


Chart 3. Controller (Page 4 of 6)

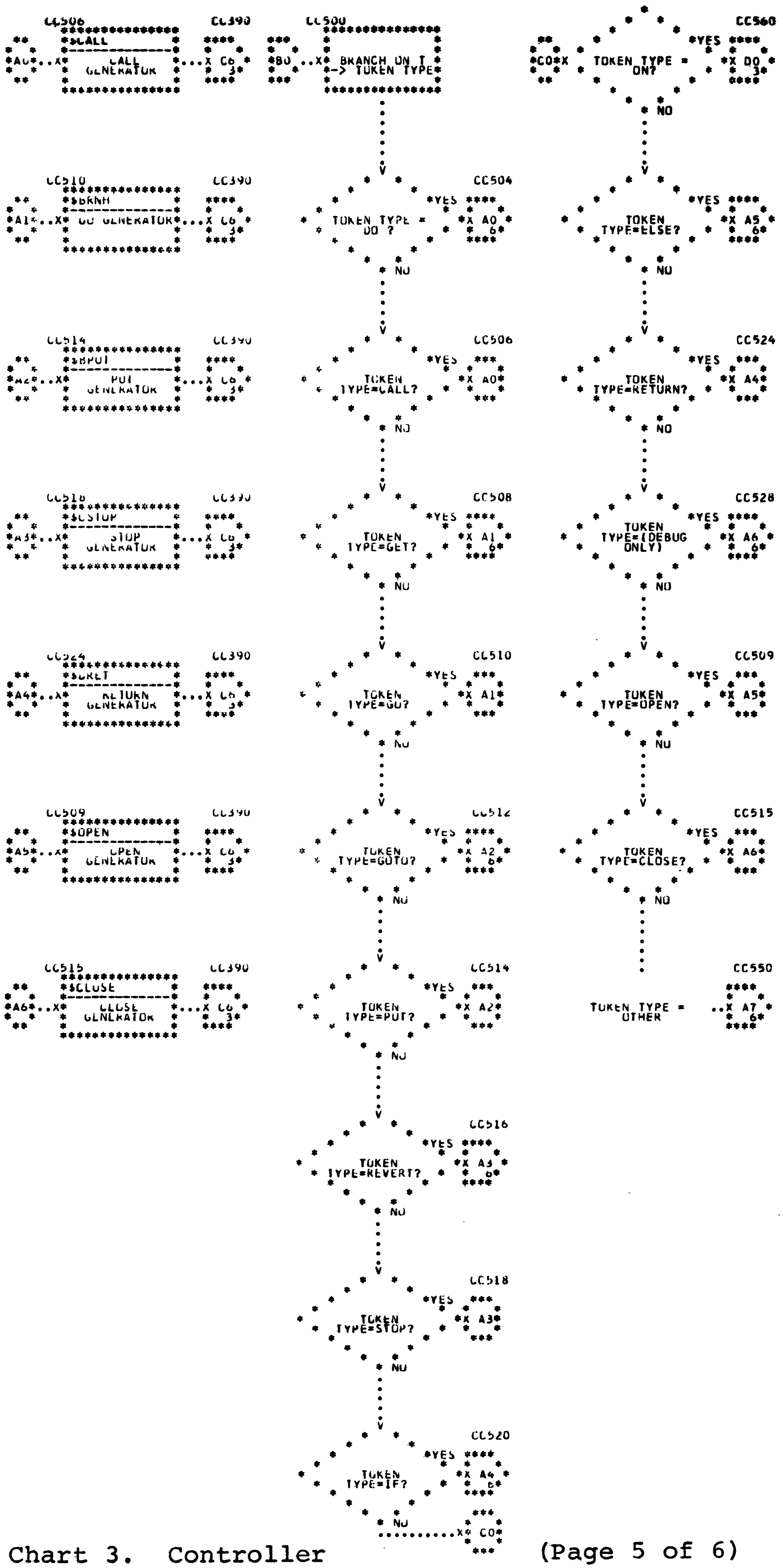


Chart 3. Controller (Page 5 of 6)

$CNT

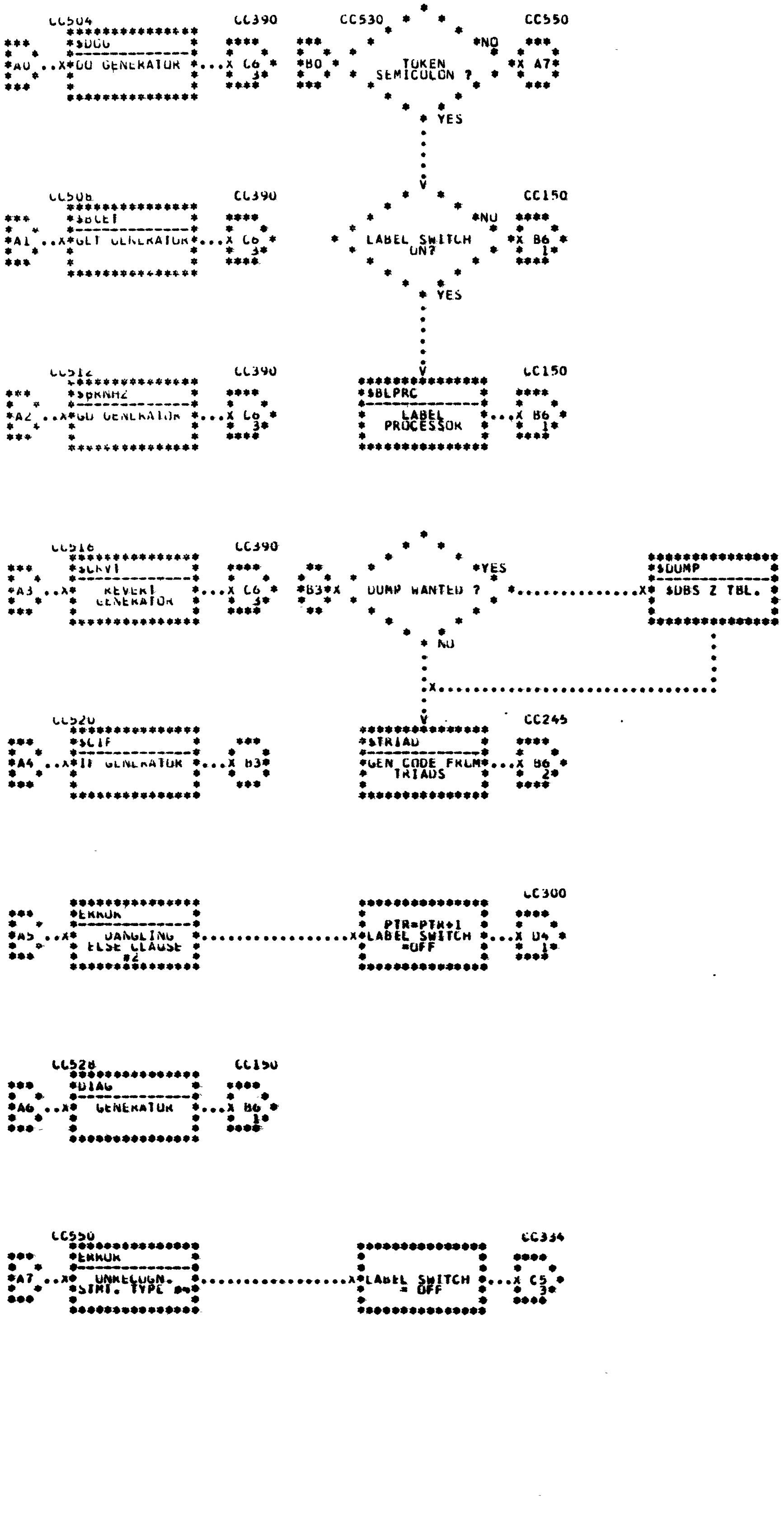


Chart 3. Controller (Page 6 of 6)

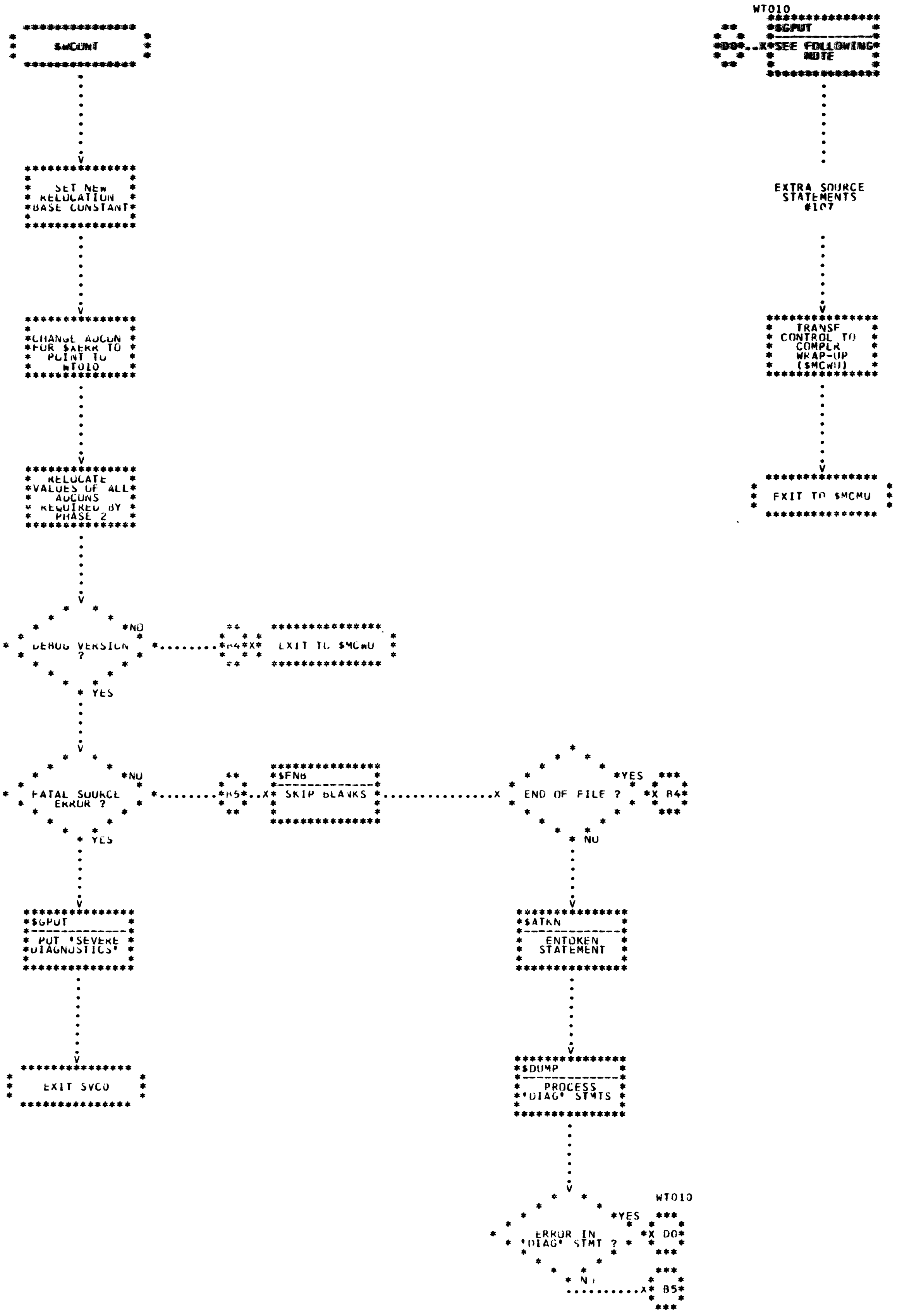


Chart 4. Phase 2 Initializer

PART 2 - ENTOKENING

The following routines perform entokening functions of CALL/360-OS PL/I. They are discussed in alphabetic order, according to their mnemonics, on succeeding pages. Detailed logic diagrams for the routines appear at the end of this subsection.

Increment Scan Index ($ASIDX)
Entoken ($ATKN, $ATKN2)
Search-Insert ($FIND)
Get Non-Blank ($FNB)

TITLE: INCREMENT SCAN INDEX ($ASIDX)

Program Definition

Purpose and Usage

Increment Scan Index is used to advance the scan index to the next character in the source stream. If the next character is on the next line, a new line token is inserted in the token table.

Description

If the scan index is not currently pointing to a new line character, it is simply advanced by one. If it is pointing to a new line character, the next line is accessed and a new line token created for it. The scan index is then set to the first non-blank character on the new line following the line number.

If there are no more lines and compilation has been completed, the scan index is set to point to the end-of-file character. Otherwise, a semicolon is supplied. Future END statements needed are provided.

Errors Detected

PROGRAM INCOMPLETE--REQUIRED END STATEMENTS SUPPLIED. (85)
LINE NUMBER NOT FOLLOWED BY BLANK. (106)

Local Variables

ASEND	END statement

Program Interface

Entry Points

Nonstandard entry. C1 return, C2 entry.

$ASIDX. No formal parameters. Scan index in P5. Pointer to next available location in token table in G4. Address of token table in P4. Displacement of next source character from beginning of line in G3.

Exit Conditions

Normal exit only. Scan index incremented. G2 destroyed.

Routines Called

$WEXP	Segment Management
$XERR	Error Message Editor

Global Variables

T Table	Token Table
$CHRFG	Building Character String Switch
$EOS	End of Source Switch
$CCF	Compilation Completed Flag
	Source Program

Comments

This routine assumes that the first significant character in the source line is the first non-blank following the line number. Note that this affects character strings only.

Logic Diagram: See Chart 5

TITLE: ENTOKEN ($ATKN, $ATKN2)

Program Definition

Purpose and Usage

The Entoken routine divides the source stream into syntactic units (tokens) up to and including the next semicolon token. During this process of division, the routine balances parentheses.

Description

The Entoken routine creates token table entries for all syntactic units in the source stream from the current position in the source to the next semicolon token.

The Entoken routine recognizes the following syntactic units:

1. Identifiers and keywords
2. Delimiters (including composite)
3. Constants
4. Comments

With several keyword exceptions, an identifier or keyword is any string of up to eight alphameric characters, the first of which is alphabetic.

The following characters act as delimiters.

+
-
*
/
↑
**

>
¬>
≥
>=
=
≠

<
¬<
≤
<=
¬=
|

(
)
;
:
._
,

(The blank delimiter is not needed after the entokening process; therefore, it is not represented in the list of delimiters for which token table entries are made.)

Constants are of three types: character-string, fixed-decimal, and floating-decimal. Since comments are not significant syntactic units, they are skipped during entokening.

Whenever an identifier token is formed, a search is made of the dictionary name list (N list) to determine whether a name entry is present for that token. If none is present, an entry is created. The token type and a pointer to the name entry for the token are then added to the token table.

Whenever a constant token is formed, a constant attribute entry is created which contains a pointer to the name and the attributes determined during entokening.

Each non-left parenthesis token contains a priority indicator which is a combination of the parenthesis count and the normal priority of the operator.

The Entoken routine balances parentheses by maintaining a push-down (last in, first out) list of all unbalanced left parentheses. Whenever a right parenthesis occurs, the top left parenthesis is removed from the list and its token in the token table is made to point to the right parenthesis. This list is kept in the unbalanced left parenthesis tokens. The left parenthesis token at the head of this list is pointed to by ATLPRN (see "Local Variables"). The unbalanced left parenthesis tokens are chained together by pointers within the tokens.

Upon adding a semicolon token to the token table, the entokening of one statement is complete. The Entoken routine returns to the CALL/360-OS PL/I Controller.

Errors Detected

EXTRA ')', IGNORED. (5)
'||' NOT SUPPORTED--CHANGED TO '|'. (6)
IDENTIFIER TRUNCATED TO 8 CHARS. (7)
EXPONENT MISSING. (8)
CONSTANT NOT SUPPORTED--DECIMAL USED. (9)
DELIMITER OR SEPARATOR MUST FOLLOW CONSTANT. (10)
BIT STRINGS NOT SUPPORTED--CHARACTER USED. (11)
'___' NOT SUPPORTED--BLANK ASSUMED. (12)
')' SUPPLIED BEFORE ';'. (13)
'___' ILLEGAL DELIMITER--IGNORED. (14)
INCOMPLETE COMMENT OR CHARACTER STRING CONSTANT. (103)
LINE NUMBER NOT FOLLOWED BY BLANK. (106)
STRING TOO LONG--FIRST 255 USED. (115)

Local Variables

ATLPRN	Last Unbalanced Left Parenthesis
ATCLNT	Parenthesis Depth Count
ATCMMA	Comma Count for Current Parenthesis Set
ATNMLG	Length of Identifier

Program Interface

Entry Points

$ATKN. No formal parameters. Entokens one statement.

$ATKN2. No formal parameters. Entokens one statement. Used for initial entry only.

Exit Conditions

Normal exit only. One statement entokened.

Routines Called

$FNB	Get Non-Blank
$FIND	Search-Insert
$ASIDX	Increment Scan Index
$SVC	SVC Director
$XERR	Error Message Editor
$WEXP	Segment Management
$WCTCT	Segment Management

Global Variables

N List	Dictionary Name List
T Table	Token Table
$PTR	Token Table Pointer
$SCNX	Scan Index
$CHRFG	Building Character String Switch
$TAREA	Translate Area
$LLINE	Last Line Number Pointer
$TSOFF	Address of Byte within Source Line
$PSCRT	Scratch Adcon Pair
$TSOFF+4	Address of Beginning of Source Line

Comments

Each translate for a certain type of character advances scan index to that character.

Logic Diagram

Chart 6 shows the detailed logic diagram for the Entoken routine.

TITLE: SEARCH-INSERT ($FIND)

Program Definition

Purpose and Usage

Using an EBCDIC character supplied as input, Search-Insert searches the dictionary name list (N list) for the presence of an entry for the identifier and, if one is present, returns a pointer to it. If no dictionary name entry is present, one is created and inserted.

Description

Each entry in the dictionary hash table points to the head of the list of dictionary name entries which hash to this location. This list is maintained in sorted order, from low to high, with one- through four-character items appearing before all five- through eight-character items.

The EBCDIC form of the identifier is hashed to obtain an index to the dictionary hash table (H table). A pointer L (for last entry) is set to point to this location in the hash table. An index P is set to the contents of this hash table location or the location of the first dictionary name entry in the list. This will be zero if there are no items in the list.

The hash algorithm consists of taking the first four characters of the identifier, padded with blanks if necessary, and dividing by a constant. The remainder is used as an index into the hash table.

If an identifier is equal to the one in the name entry pointed to by P, it is assumed to be the required identifier. If the identifier is less than the name entry, a new entry is created and placed in the list between L and P. If the identifier is greater than the name entry, L is set to P, and P is advanced to the next entry. If P points to the name entry, the entire list has been searched and a new entry is inserted after L. Otherwise, the process continues as before.

This method of searching creates ordered name lists for faster searching.

Errors Detected

None

Local Variables

FINME	Name to be Searched

Program Interface

Entry Points

$FIND. Needs an identifier for which to search. G6 has its length in characters. P3 has its address.

Exit Conditions

Normal exit only. Returns a pointer in G0 to the dictionary name entry.

Routines Called

None

Global Variables

H Table	Dictionary Hash Table
N List	Dictionary Name List

Logic Diagram

Chart 7 shows the detailed logic diagram for the Search-Insert routine.

TITLE: GET NON-BLANK ($FNB)

Program Definition

Purpose and Usage

Get Non-Blank is used to advance the scan index to the next non-blank character in the source program. During this process, it skips blanks and comments.

Description

A search is made for the next non-blank character in the current source line. If it is a new line character, Increment Scan Index is called to advance to the next line, where the search continues.

If the next non-blank character is a slash, a check for a comment must be made and, if present, the comment skipped. Upon exit from this routine, scan index is pointing to the next non-blank, non-new-line character in the source program.

Errors Detected

INCOMPLETE COMMENT OR CHARACTER STRING CONSTANT. (103)

Local Variables

None

Program Interface

Entry Points

Nonstandard entry. C1 return, C2 entry.

$FNB. No formal parameters. Scan index in P5.

Exit Conditions

Normal exit only. Scan index in P5. G2 destroyed.

Routines Called

$ASIDX	Increment Scan Index
$XERR	Error Message Editor

Global Variables

$TAREA	Translate Area

Logic Diagram

Chart 8 shows the detailed logic diagram for the Get Non-Blank routine.

PART 2 LOGIC DIAGRAMS

The detailed logic diagrams for the routines that perform entokening follow.

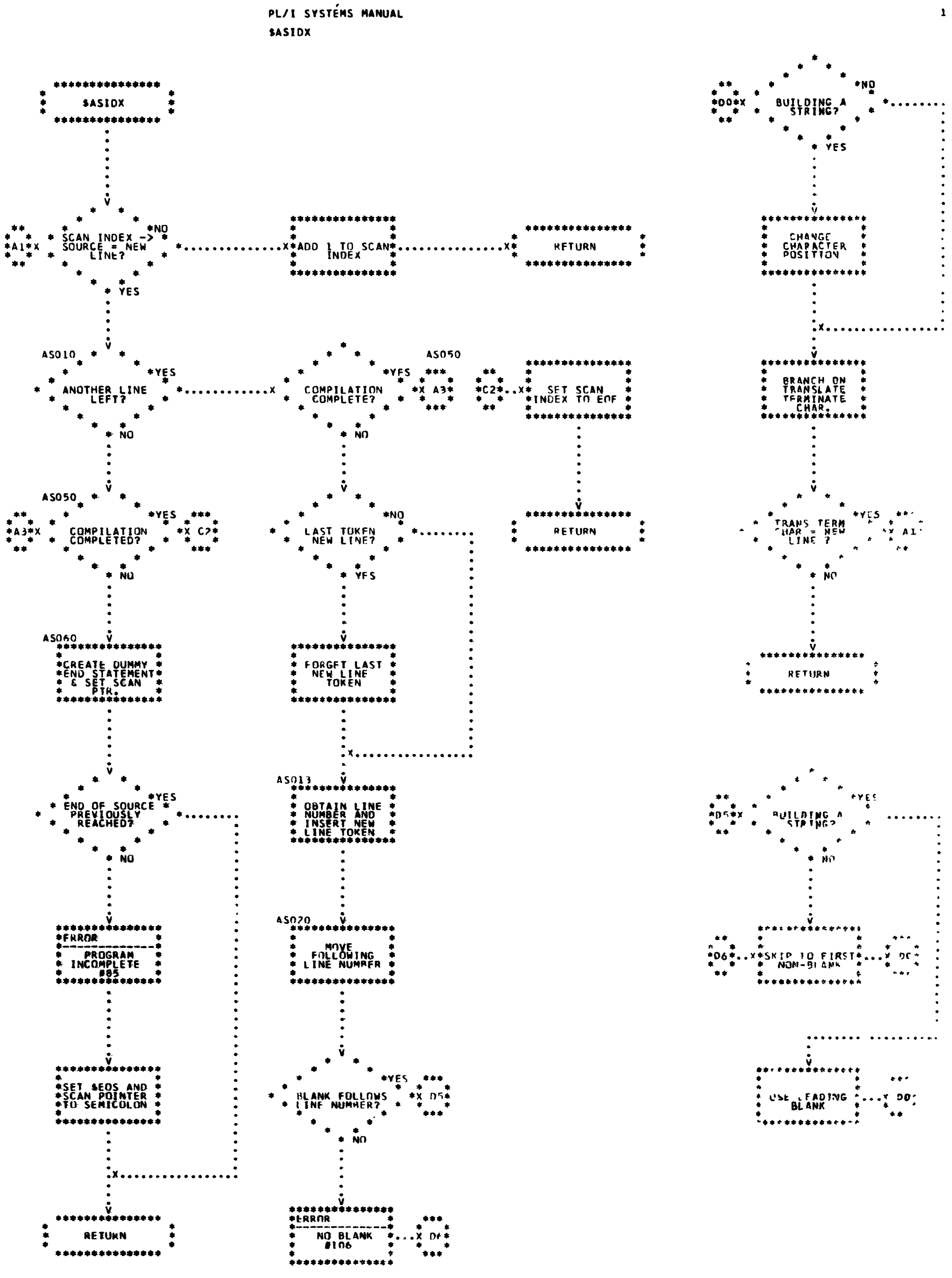


Chart 5. Increment Scan Index

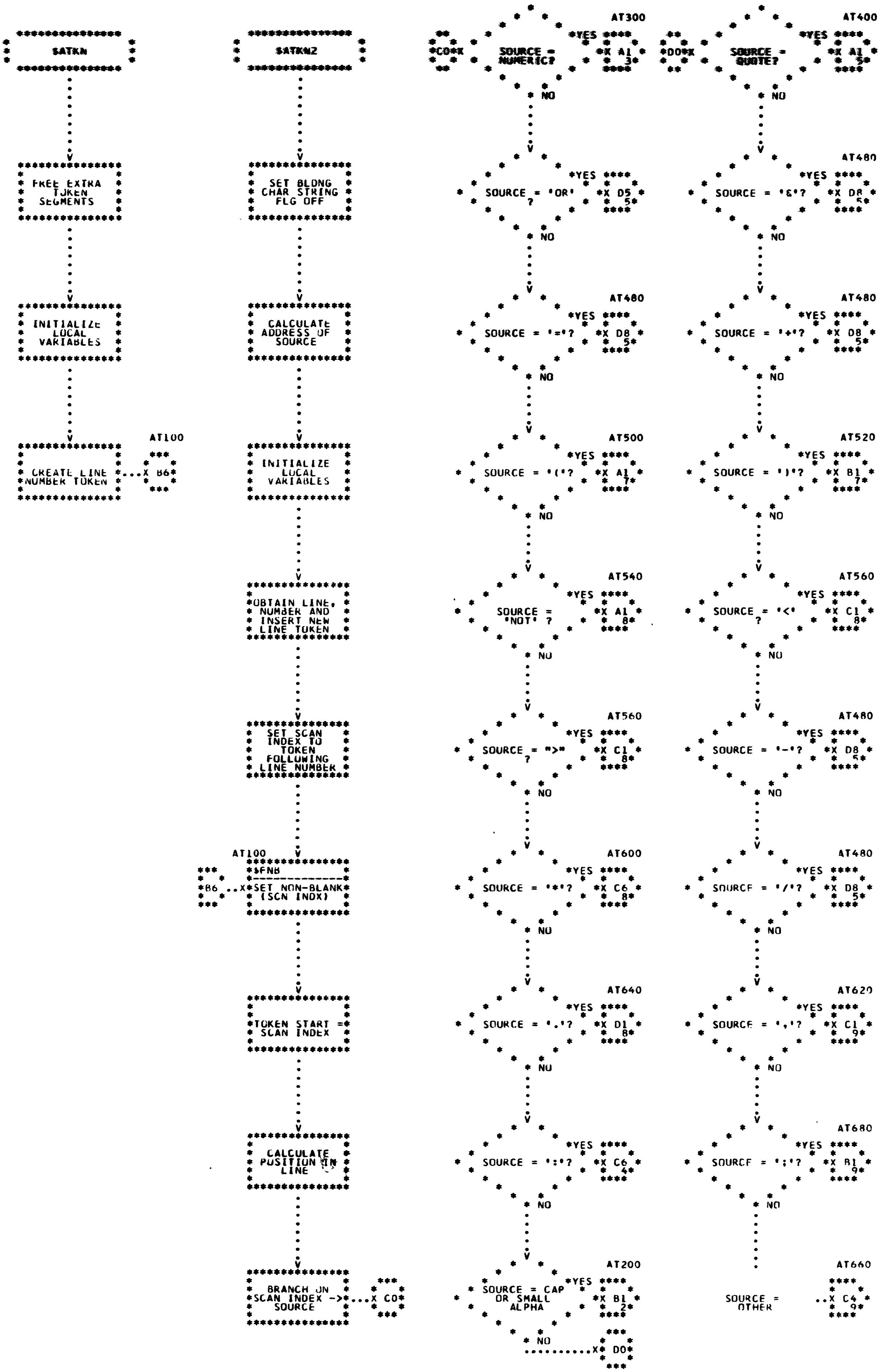


Chart 6. Entoken (Page 1 of 9)

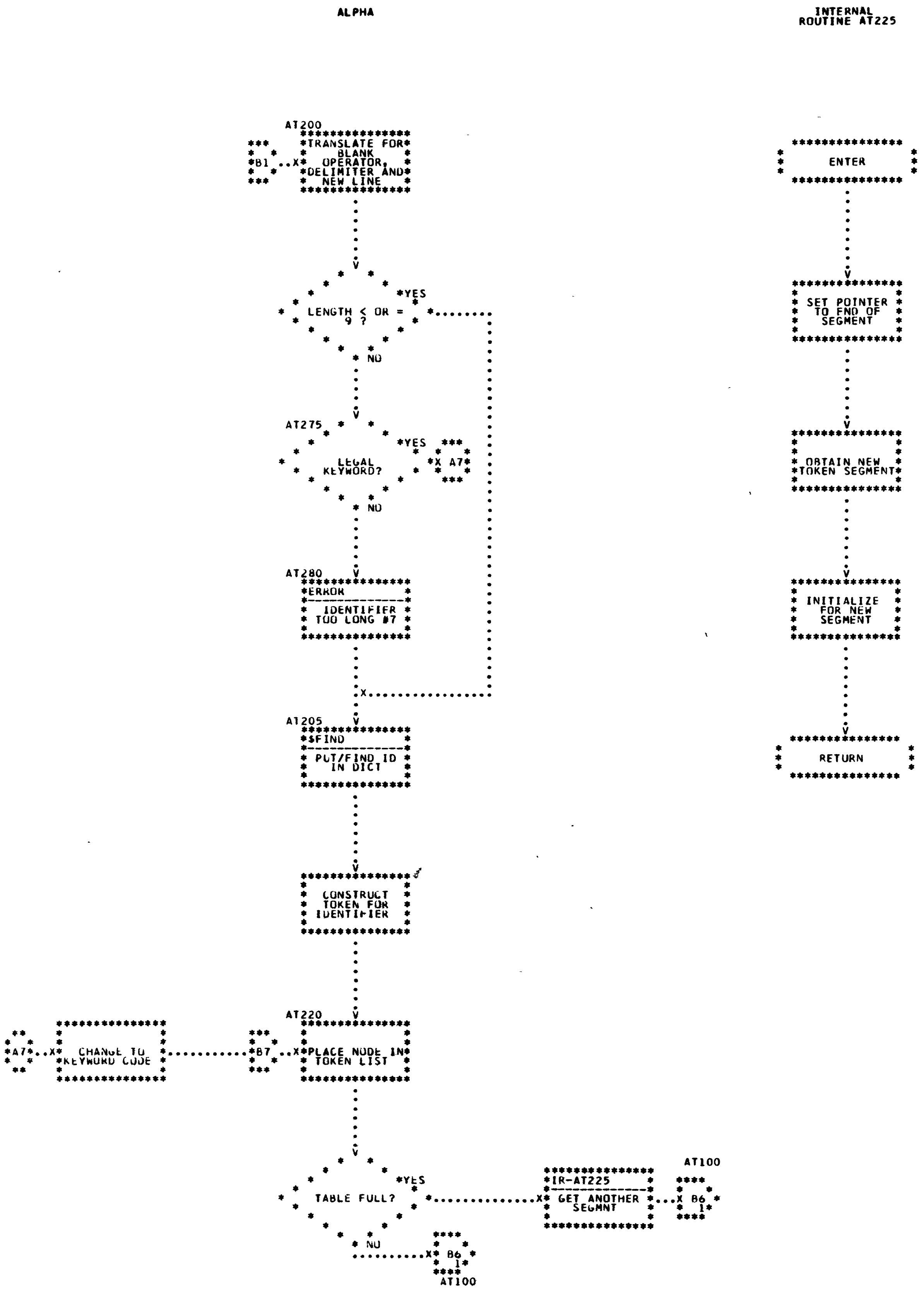


Chart 6. Entoken (Page 2 of 9)

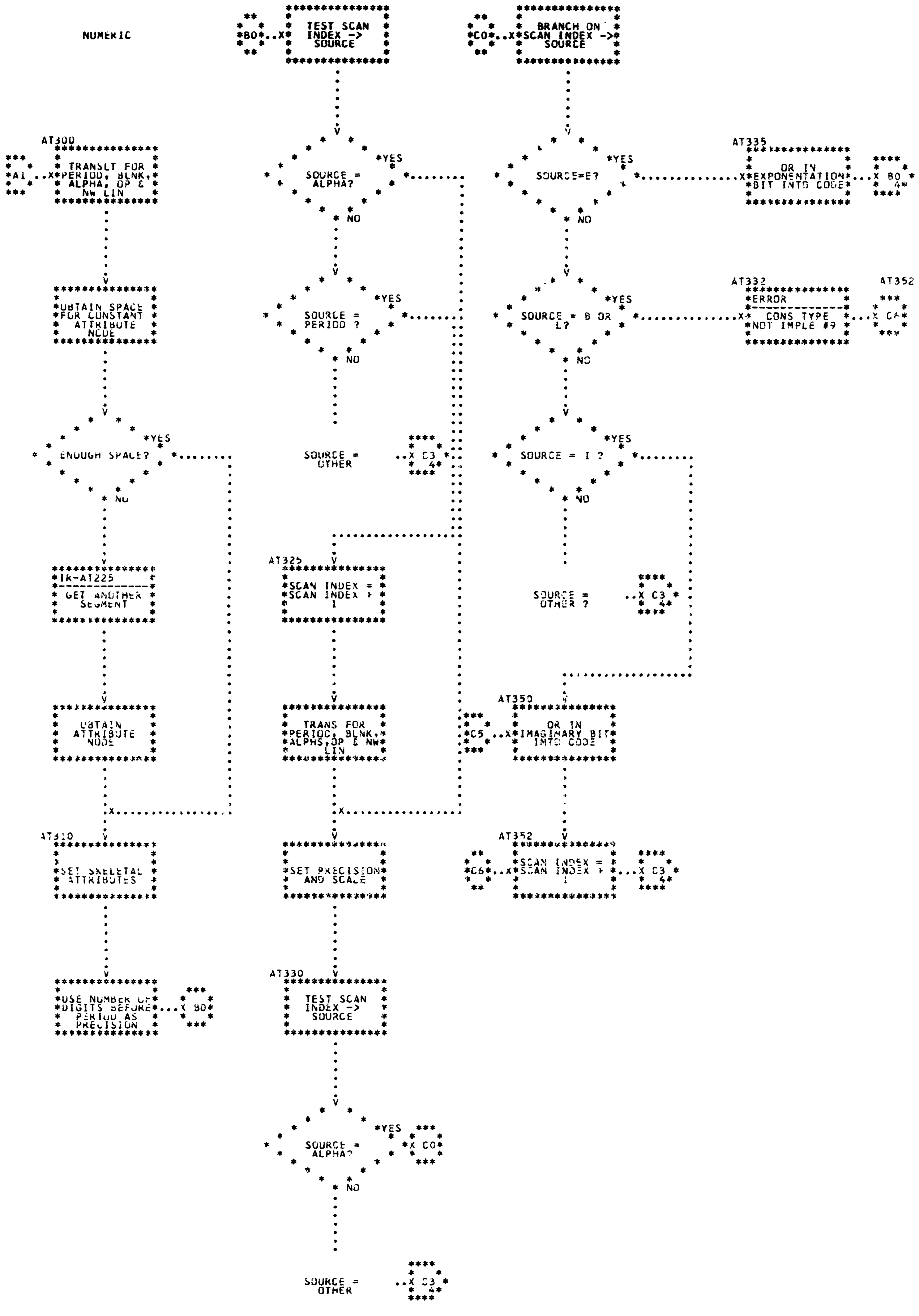


Chart 6. Entoken (Page 3 of 9)

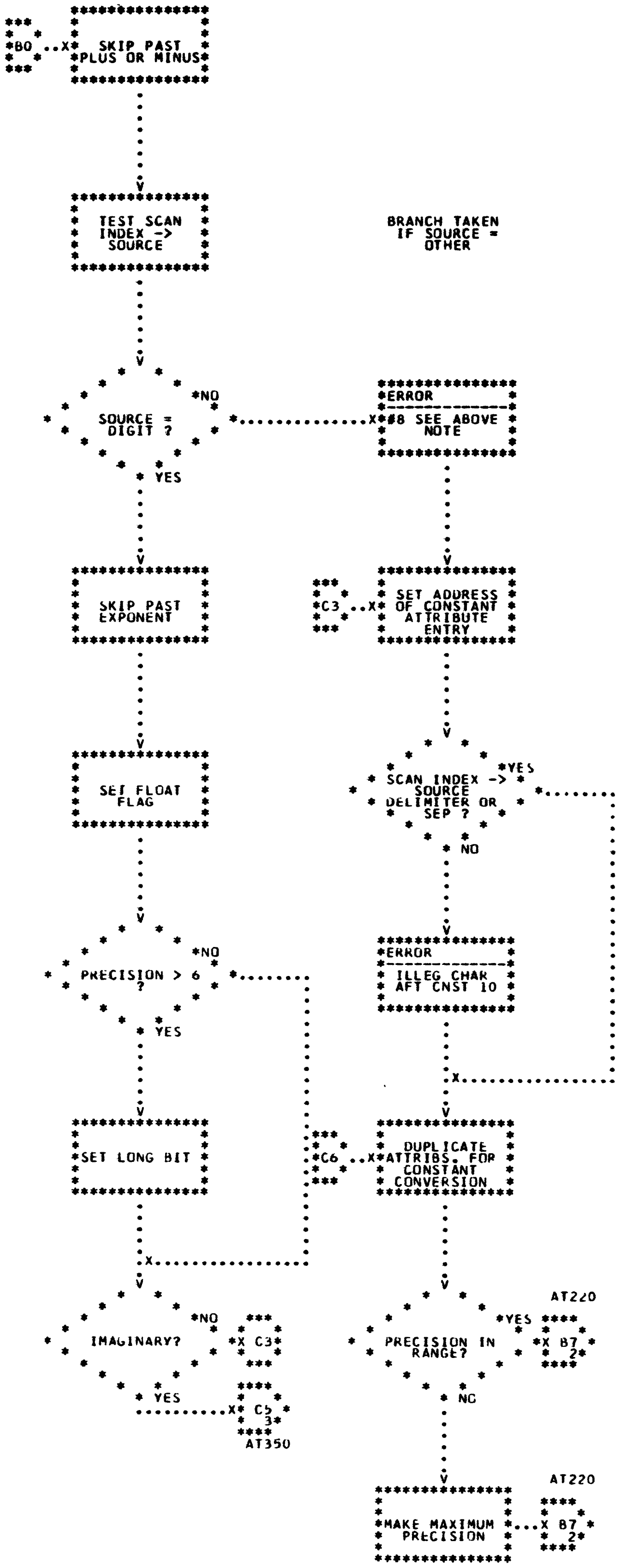


Chart 6. Entoken (Page 4 of 9)

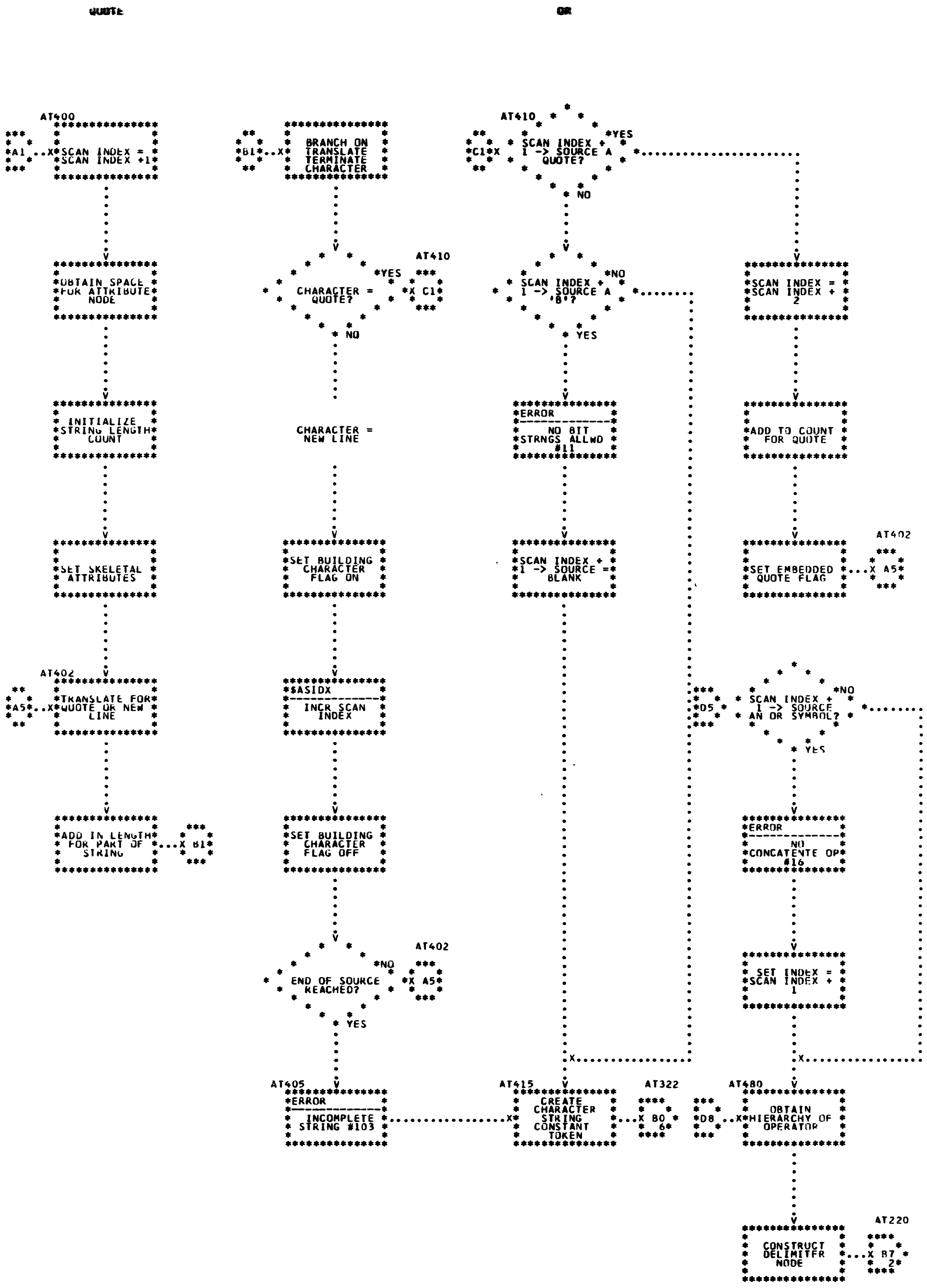
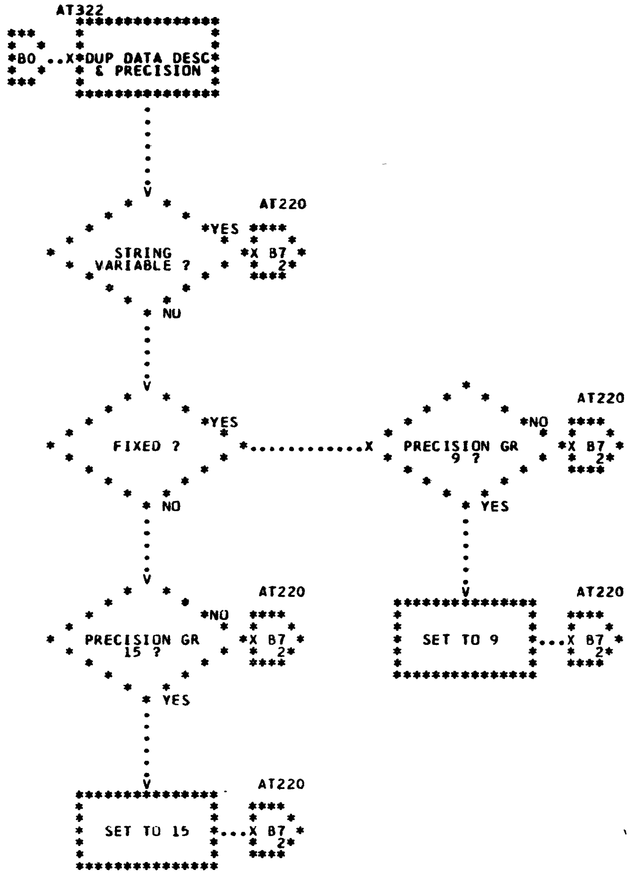


Chart 6. Entoken (Page 5 of 9)


AT322
B0
DUP DATA DESC & PRECISION
STRING VARIABLE ?
YES
AT220
B7 2
NO
FIXED ?
YES
PRECISION GR 9 ?
NO
AT220
B7 2
YES
NO
PRECISION GR 15 ?
NO
AT220
B7 2
YES
SET TO 9
AT220
B7 2
SET TO 15
AT220
B7 2

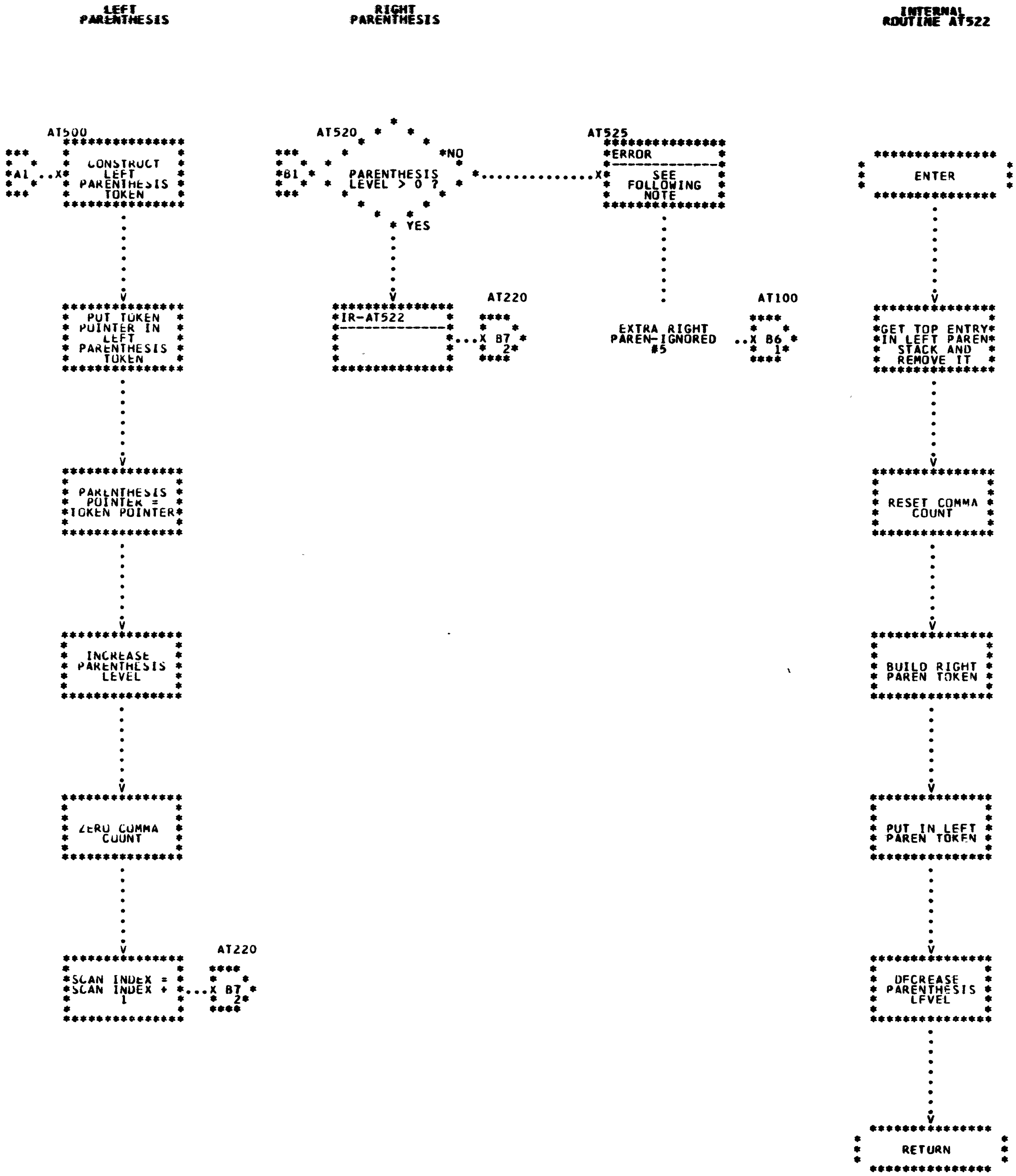
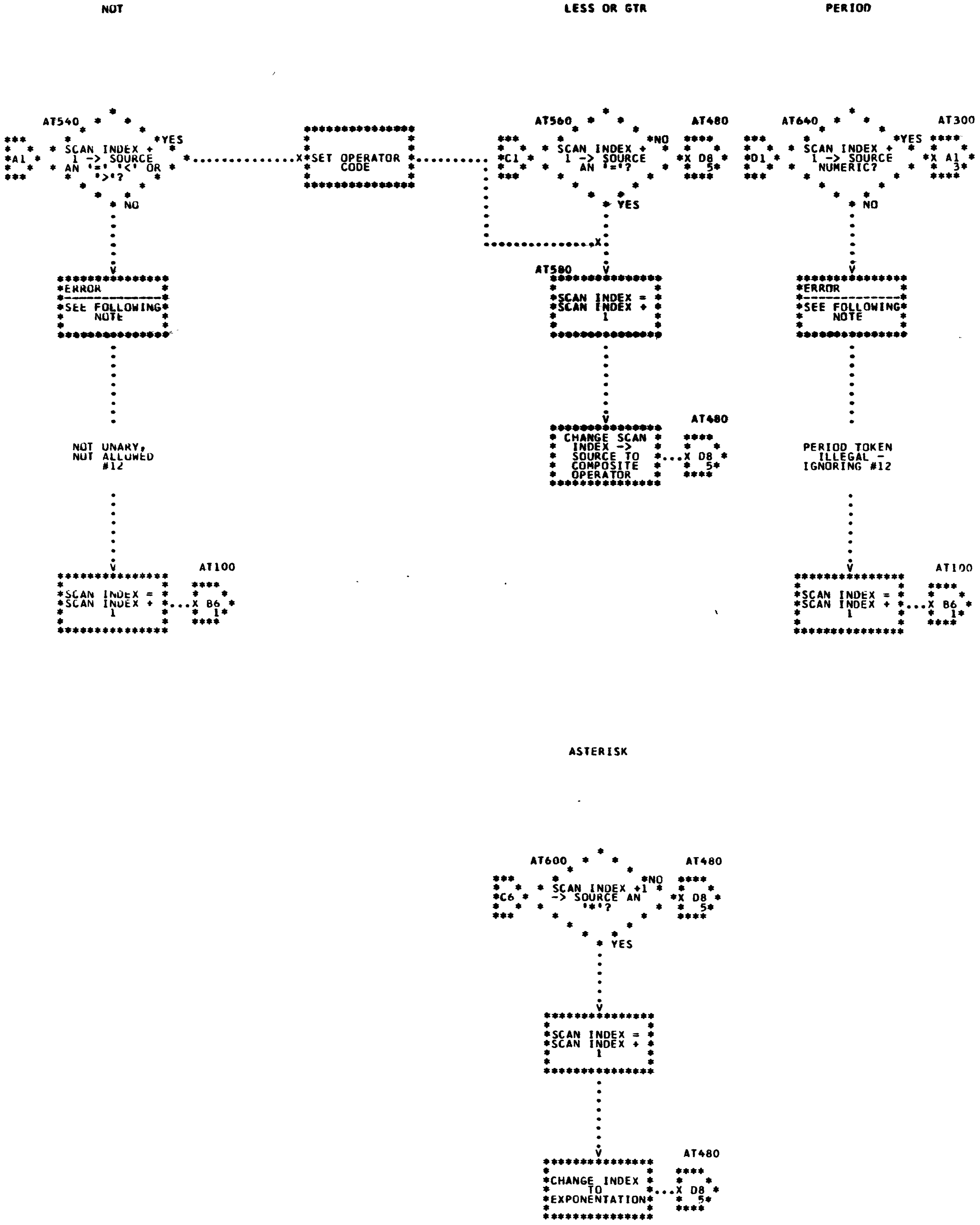


Chart 6. Entoken (Page 7 of 9)


NOT
LESS OR GTR
PERIOD
AT540
A1
SCAN INDEX + 1 -> SOURCE AN '=' '<' OR '>'?
YES
NO
SET OPERATOR CODE
ERROR
SEE FOLLOWING NOTE
NOT UNARY, NOT ALLOWED #12
SCAN INDEX = SCAN INDEX + 1
AT100
B6 1
AT560
C1
SCAN INDEX + 1 -> SOURCE AN '='?
NO
YES
AT480
D8 5
AT580
SCAN INDEX = SCAN INDEX + 1
CHANGE SCAN INDEX -> SOURCE TO COMPOSITE OPERATOR
AT480
D8 5
AT640
D1
SCAN INDEX + 1 -> SOURCE NUMERIC?
YES
NO
AT300
A1 3
ERROR
SEE FOLLOWING NOTE
PERIOD TOKEN ILLEGAL - IGNORING #12
SCAN INDEX = SCAN INDEX + 1
AT100
B6 1
ASTERISK
AT600
C6
SCAN INDEX +1 -> SOURCE AN '*'?
NO
YES
AT480
D8 5
SCAN INDEX = SCAN INDEX + 1
CHANGE INDEX TO EXPONENTATION
AT480
D8 5

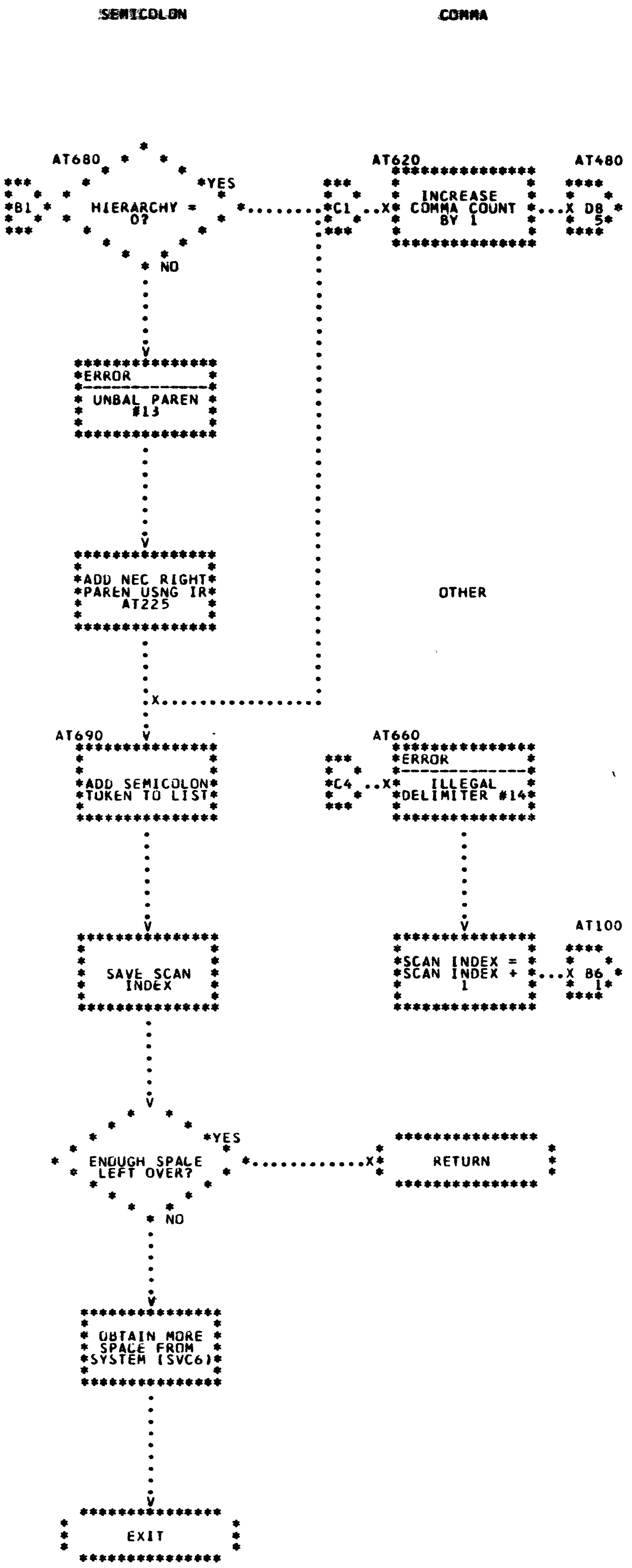


Chart 6. Entoken (Page 9 of 9)

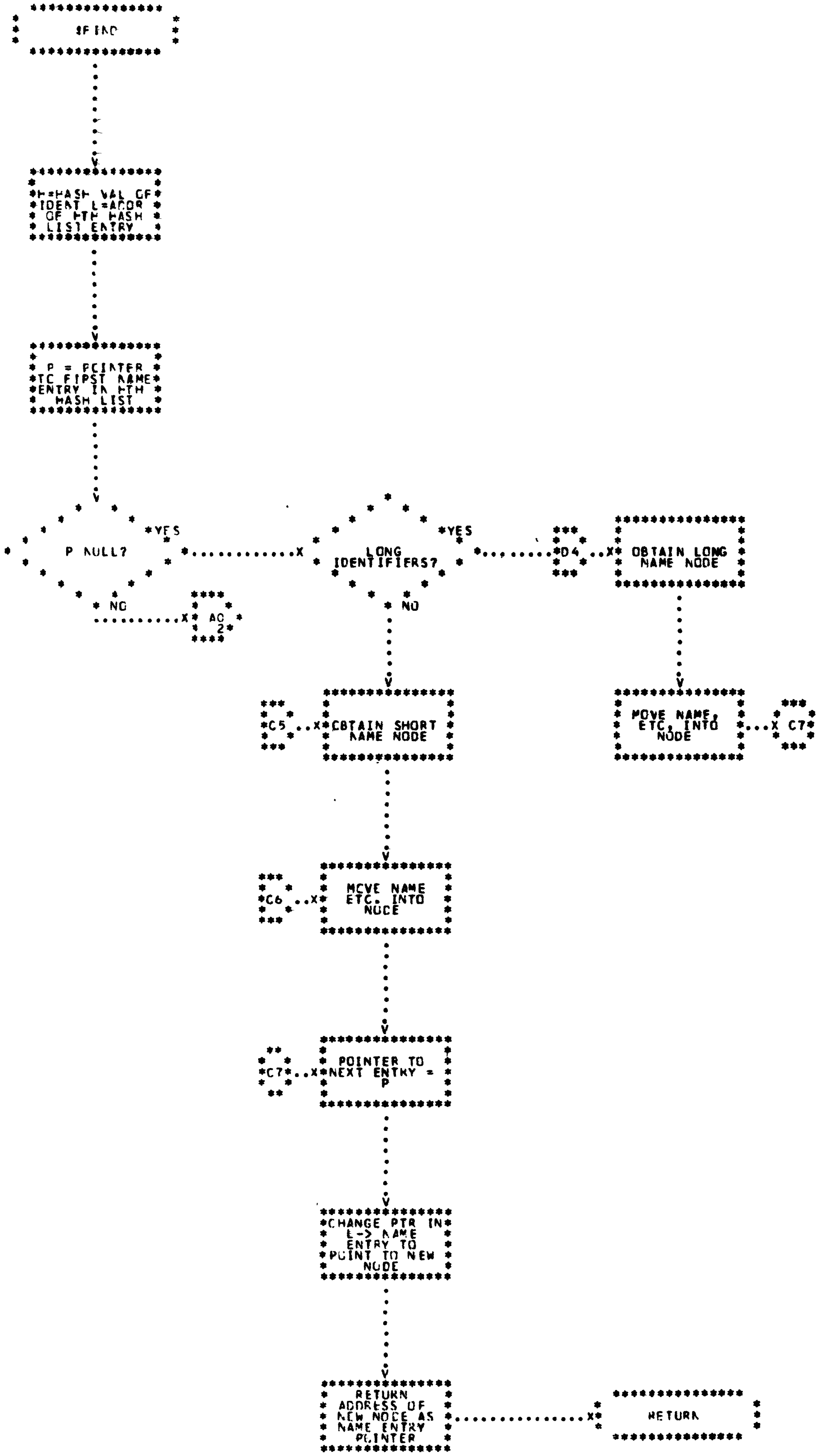


Chart 7. Search Insert (Page 1 of 2)

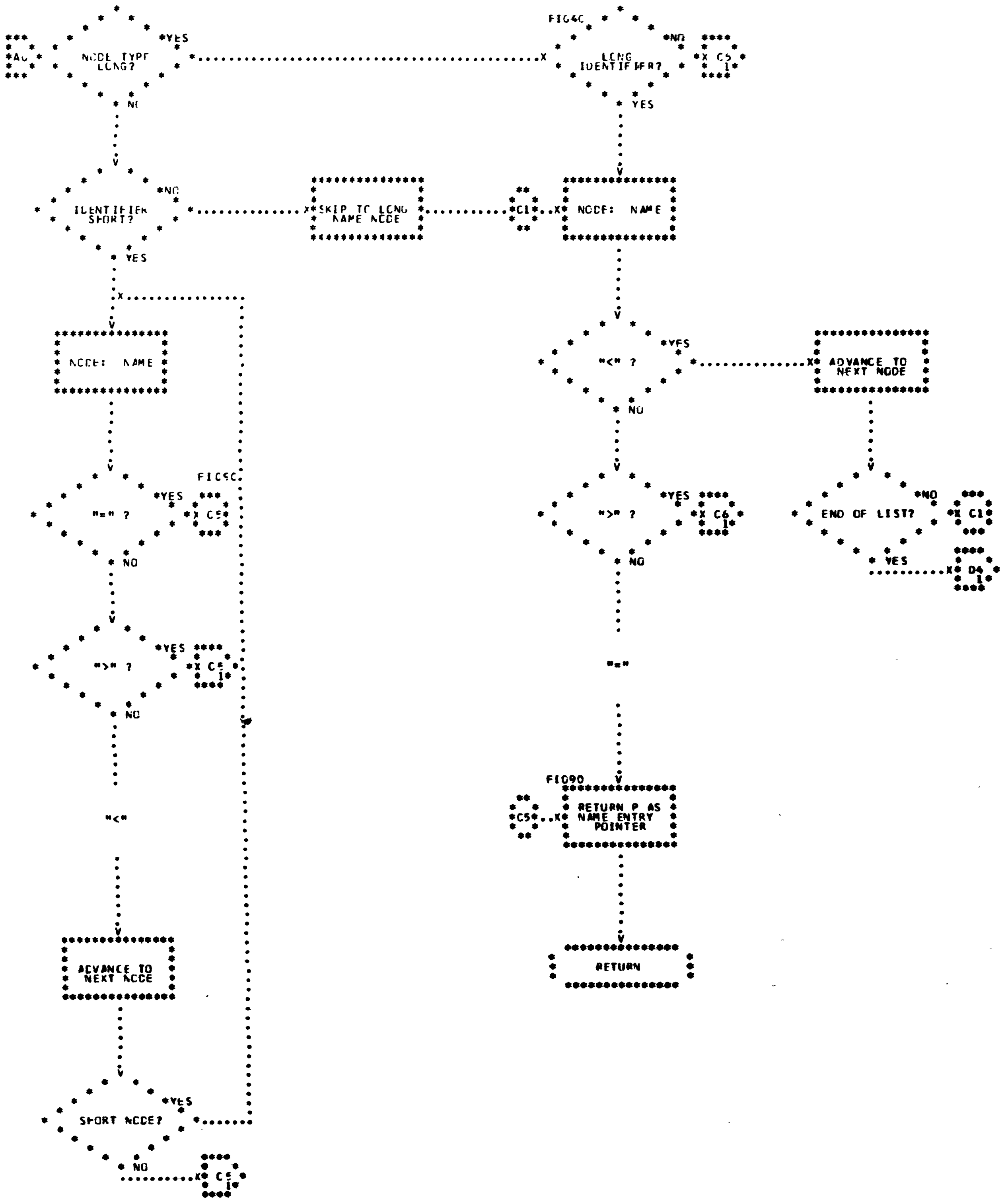


Chart 7. Search Insert (Page 2 of 2)

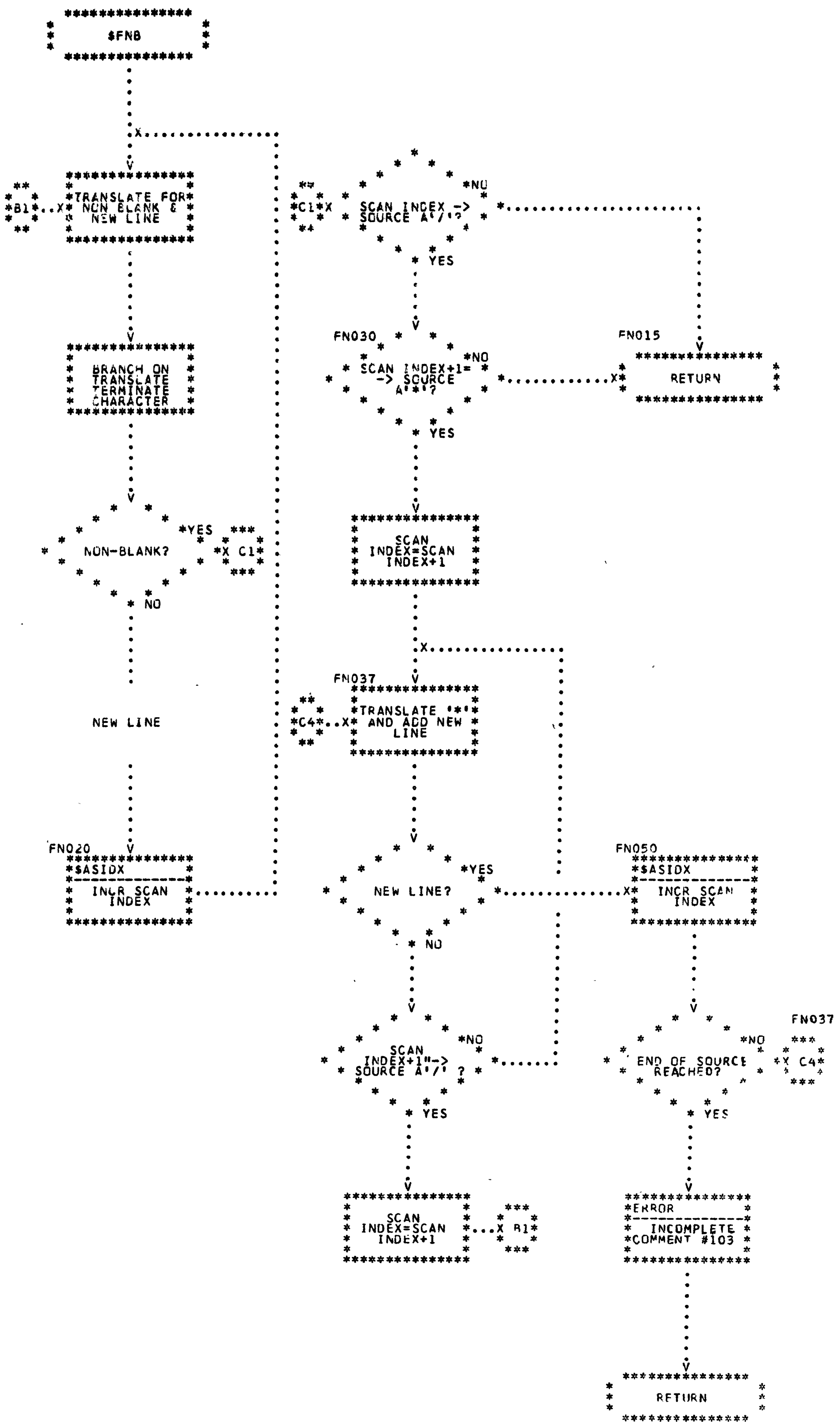


Chart 8. Get Non-Blank

PART 3 - GENERAL STATEMENT PROCESSORS

CALL/360-OS PL/I general statement processors are described in this subsection. The routines are discussed in alphabetic order, according to their mnemonics, as shown below. Detailed logic diagrams for the routines appear at the end of the subsection.

Assignment Generator ($ACGEN)
PROC Generator ($APRC, $APRC2)
BEGIN Generator ($BEGIN)
On-Unit Specification Analyzer ($BONSA)
GO Generator ($BRNH, $BRNH2)
CALL Generator ($CALL)
IF Generator ($CIF)
ON Generator ($CON)
REVERT Generator ($CRVT)
STOP Generator ($CSTOP)
DO-Loop Triad Builder ($DEXP)
DO Generator ($DOG, $DOG2)
RETURN Generator ($DRET)
END Generator ($EDGN, $EDGN2)
Process End of ELSE ($ENDES)
Process End of ON ($ENDON)
Expander ($EXPND)
END Expand ($EYPND)

TITLE: ASSIGNMENT GENERATOR ($ACGEN)

Program Definition

Purpose and Usage

The Assignment Generator prepares an assignment statement for analysis by the Expression Processor Controller.

Description

The assignment symbol in the assignment statement is changed from an ordinary equal sign to an assignment type equal. This allows the Expression Processor Controller to translate the whole statement and perform the assignment.

If the statement is an array assignment, the statement is expanded and reprocessed by the Expression Processor Controller. Then the expansion is terminated.

Errors Detected

ILLEGAL ASSIGNMENT STATEMENT. (15)
ERROR AT '___'. (16)

Local Variables

None

Program Interface

Entry Points

$ACGEN. No formal parameters. Expects $PTR to point to first token in statement.

Exit Conditions

Normal exit only.

Routines Called

$NEXP	Expression Processor Controller
$CERR	Compiler Error
$EXPND	Expander
$EYPND	END Expand
$XERR	Error Message Editor

Global Variables

T Table	Token Table
$PTR	Token Table Pointer

Logic Diagram

Chart 9 shows the detailed logic diagram for the Assignment Generator routine.

TITLE: PROC GENERATOR ($APRC, $APRC2)

Program Definition

Purpose and Usage

The PROC Generator analyzes the syntax, creates parameter declarations, and generates triads for PROCEDURE statements.

Description

PROC Generator checks to make sure that a label is present on the PROCEDURE statement. If a parameter list is present, tentative default attributes are created for the parameters. If RETURNS attributes are specified, they are analyzed and used in creating these attributes. If no RETURNS attributes are supplied, default attributes are created using the procedure name.

The code generated for an internal procedure consists of a branch around the procedure and the prologue code. Entries for the block are made in both the program structure table and block information table.

If the procedure is the external procedure, a second entry point is used. This bypasses the main analysis, because the only information pertinent to an external procedure is the possible presence of the OPTIONS (MAIN) attribute.

Errors Detected

ERROR AT '___'. (16)
'___' WHERE ',' EXPECTED--SKIPPING TO '___'. (32)
LABEL MISSING. (46)
ILLEGAL RETURNS ATTRIBUTES, DEFAULT RETURNS ATTRIBUTES USED. (47)
ILLEGAL OPTION ON EXTERNAL PROCEDURE STATEMENT. (48)
MULTIPLE DECLARATION FOR '___'--THIS DECLARATION USED. (56)

Local Variables

APBIT	Attribute Bit Mask
APFIB	FIB Pointer
APTRD	Dummy Name Token
APATT	Pointer to ENTRY Attribute
APRET	Pointer to RETURNS Attribute
APPRCT	Type of Procedure (0 = External)

Program Interface

Entry Points

$APRC. No formal parameters. Expects $PTR to point to token following PROC. For internal procedures.

$APRC2. No formal parameters. Expects $PTR to point to token following PROC. For external procedures.

Exit Conditions

Normal exit only. $PTR points to semicolon.

Routines Called

$ANCRE	Attribute Node Creation
$ABAL	Attribute Analysis
$GTRIAD	Get Next Triad Entry

$BLPRC	Label Processor
$FIND	Search-Insert
$WEXP	Segment Management
$XERR	Error Message Editor

Global Variables

T Table	Token Table
$PTR	Token Table Pointer
P Table	Program Structure Table
B Table	Block Information Table
A List	Dictionary Attribute List
$CLBLS	Label Switch
$DCNME	Name of Current Identifier

Logic Diagram

Chart 10 shows the detailed logic diagram for the PROC Generator routine.

TITLE: BEGIN GENERATOR ($BEGIN)

Program Definition

Purpose and Usage

The BEGIN Generator checks the syntax of a BEGIN statement and generates the initial part of the prologue for the block.

Description

If a label is present on the statement, it is processed. The syntax is checked by insuring that a semicolon immediately follows the word BEGIN. Entries for the block are made in the program structure and block information tables. A triad for the initial part of the prologue for a begin block is created.

Errors Detected

ERROR AT '___'. (16)

Local Variables

None

Program Interface

Entry Points

$BEGIN. No formal parameters. Expects $PTR to point to token following BEGIN.

Exit Conditions

Normal exit only. $PTR points to semicolon.

Routines Called

$BLPRC	Label Processor
$GTRIAD	Get Next Triad Entry
$WEXP	Segment Management
$XERR	Error Message Editor

Global Variables

T Table	Token Table
$PTR	Token Table Pointer
P Table	Program Structure Table
B Table	Block Information Table

Logic Diagram

Chart 11 shows the detailed logic diagram for the BEGIN Generator routine.

TITLE: ON-UNIT SPECIFICATION ANALYZER ($BONSA)

Program Definition

Purpose and Usage

The On-Unit Specification Analyzer checks the ON-condition for legality and returns a code indicating the type of unit.

Description

The type of token is checked and its keyword code obtained. If not legal for an ON-condition, a message is given. If the code is for ENDFILE, a check is made for (filename) and a pointer to the filename is returned.

Errors Detected

ERROR AT '___'. (16)
FILE NAME MISSING. (40)
'___' NOT FILENAME. (41)
UNRECOGNIZABLE ON-CONDITION. (44)

Local Variables

None

Program Interface

Entry Points

$BONSA. No formal parameters. Expects $PTR to point to condition.

Exit Conditions

Normal exit only. $PTR points to token following condition. Register G2 indicates type of unit detected:

G2 = 0	Unrecognizable
G2 = 4	ERROR
G2 = 8	FIXEDOVERFLOW
G2 = 12	OVERFLOW
G2 = 16	UNDERFLOW
G2 = 20	ZERODIVIDE
G2 = 24	ENDFILE (filename)

Filename pointer stored in $FILON.

Routines Called

$FVAR	Locate Variable
$XERR	Error Message Editor

Global Variables

T Table	Token Table
N List	Dictionary Name List
$PTR	Token Table Pointer
$FILON	On-Unit Flag and FCIB Pointer

Logic Diagram

Chart 12 shows the detailed logic diagram for the On-Unit Specification Analyzer.

TITLE: GO GENERATOR ($BRNH, $BRNH2)

Program Definition

Purpose and Usage

The GO Generator syntactically and semantically analyzes GO TO statements and generates code to perform the indicated functions.

Description

Five situations may occur during compilation of a GO TO statement:

1. Forward references to statement-label constants.
2. Backward reference to statement-label constant in same block.
3. Backward reference to statement-label constant in containing block.
4. Reference to a statement-label variable in same block or array label.
5. Reference to a statement-label variable in containing block.

The second and fourth situations present special cases. A branch to the statement referenced (or to the GO TO Interpreter (IHESAF) if a statement-label variable) is required.

In any other situation, the GO Generator leaves sufficient space in the object code file to insert instructions. Information is placed in this area to allow proper code generation later. The proper code is inserted into this area when it is determined which of the three remaining situations exists.

Errors Detected

ERROR AT '___' (16)
'TO' MISSING AFTER 'GO' (36)
ILLEGAL STATEMENT LABEL--STATEMENT IGNORED (37)

Local Variables

None

Program Interface

Entry Points

$BRNH. No formal parameters. Expects $PTR to point to keyword token TO.

$BRNH2. No formal parameters. Expects $PTR to point to token following keyword GOTO.

Exit Conditions

Normal exit only. $PTR points to semicolon token.

Routines Called

$CERR	Compiler Error
$FSYM	Locate Identifier
$NEXP	Expression Processor Controller
$GTRIAD	Get Next Triad Entry
$XERR	Error Message Editor
$AREXP	Array Expression Error

Global Variables

T Table	Token Table
$PTR	Token Table Pointer
A List	Dictionary Attribute List

Logic Diagram

Chart 13 shows the detailed logic diagram for the GO Generator routine.

TITLE: CALL GENERATOR ($CALL)

Program Definition

Purpose and Usage

The CALL Generator analyzes the syntax and directs generation of the triads for a CALL statement.

Description

Before passing the entry name to the Expression Processor Controller for processing, the CALL Generator ensures that the entry name is properly declared and that the syntax of the statement is correct.

An entry name of $RESET that is not declared as an entry point is defined to be the special CALL/360-OS feature CALL $RESET. In this case the identifier following the left parenthesis must be a filename. A triad is generated to call the Reset Disk Files routine (IHERSET) to execute the reset operation. The address of the file control interface block (FCIB) is passed in R13.

Errors Detected

ERROR AT '___'. (16)
'___' NOT ENTRY NAME. (17)
'___' NOT FILE NAME. (41)

Local Variables

None

Program Interface

Entry Points

$CALL. No formal parameters. Expects $PTR to point to entry name.

Exit Conditions

Normal exit only.

Routines Called

$FVAR	Locate Variable
$NEXP	Expression Processor Controller
$GTRIAD	Get Next Triad Entry
$XERR	Error Message Editor

Global Variables

Z Table	Triad Table
T Table	Token Table
$PTR	Token Table Pointer

Logic Diagram

Chart 14 shows the detailed logic diagram for the CALL Generator routine.

TITLE: IF GENERATOR ($CIF)

Program Definition

Purpose and Usage

The IF Generator analyzes the syntax and directs the generation of triads for an IF statement.

Description

The IF Generator directs the translation of the logical expression in an IF statement and tests the result and generates a conditional branch to the ELSE clause. An entry is made for the IF statement in the program structure table and the compound statement switch is set so that a THEN clause is expected.

A check is performed to insure that a THEN clause follows. If the word THEN does not immediately follow the expression, a search is made to find it. If it is not present, a null-statement THEN unit is formed.

Errors Detected

ERROR AT '___'. (16)
'THEN' CLAUSE MISSING, NULL ASSUMED. (39)

Local Variables

CITHEN	Name THEN

Program Interface

Entry Points

$CIF. No formal parameters. Expects $PTR to point to token following IF.

Exit Conditions

Normal exit only. $PTR points to token following THEN.

Routines Called

$GTRIAD	Get Next Triad Entry
$NEXP	Expression Processor Controller
$XERR	Error Message Editor
$AREXP	Array Expression Error
$WCTCT	Segment Management
$WEXP	Segment Management

Global Variables

T Table	Token Table
$PTR	Token Table Pointer
P Table	Program Structure Table

Logic Diagram

Chart 15 shows the detailed logic diagram for the IF Generator routine.

TITLE: ON GENERATOR ($CON)

Program Definition

Purpose and Usage

The ON Generator generates code to establish ON-condition addresses. It also analyzes the syntax of the ON statement and creates an entry in the program structure table (P table) for the ON statement.

Description

The conditions for which ON statements are allowed are the following:

ERROR
FIXEDOVERFLOW
OVERFLOW
UNDERFLOW
ZERODIVIDE
ENDFILE (filename)

ON Generator makes sure that the on-unit is an unlabeled statement. The Controller makes sure it is a legal unit.

Since all temporaries in the on-unit use special temporaries, each on-unit is handled as a procedure block and creates a new entry in the temporary storage table (S table) which is removed upon completion of the on-unit.

When an ENDFILE on-unit is encountered, the following special steps occur:

1. Space is obtained for the on-unit parameter list in static and constants area. The address of the parameter list is saved in the block information table (B table). (For the format of the on-unit parameter list, see Appendix B.)
2. The number of ENDFILEs encountered for the current block (B$FILE) is incremented.
3. Code is generated to initialize the on-unit parameter list, including the 20-byte on-unit adcon area, address of FCIB. An ENDFILE table entry is initialized in the END Generator. (See Appendix B.)
4. Code is generated to call the On-ENDFILE and REVERT Initializer (IHEONREV) at entry-point IHEONUN and pass the address of the on-unit parameter list in R13.

Errors Detected

ERROR AT '___'. (16)
LABEL ILLEGAL HERE--IGNORED. (45)

Local Variables

None

Program Interface

Entry Points

$CON. No formal parameters. Expects $PTR to point to token following ON.

Exit Conditions

Normal exit only. $PTR points to first token of the on-unit in the ON statement or to semicolon ending the ON statement. (For details regarding on-units, see the CALL/360-OS PL/I Language Reference Manual.)

Routines Called

$BONSA	On-Unit Specification Analyzer
$GTRIAD	Get Next Triad Entry
$CATEG	Statement Category
$NCONS	Constant Processor
$XERR	Error Message Editor
$WEXP	Segment Management

Global Variables

S Table	Temporary Storage Table
T Table	Token Table
$PTR	Token Table Pointer
P Table	Program Structure Table
$CSS	Compound Statement Switch
B Table	Block Information Table
$FILON	On-Unit Flag and FCIB Pointer

Logic Diagram

Chart 16 shows the detailed logic diagram for the ON Generator routine.

TITLE: REVERT GENERATOR ($CRVT)

Program Definition

Purpose and Usage

The REVERT Generator analyzes the syntax and generates the code associated with the REVERT statement.

Description

The conditions for which REVERT statements are allowed are:

ERROR
FIXEDOVERFLOW
OVERFLOW
UNDERFLOW
ZERODIVIDE
ENDFILE (filename)

For ENDFILE condition, code is generated to call the On-ENDFILE and REVERT Initializer library subroutine (IHEONREV) at entry point IHEREVT. The address of the file control interface block (FCIB) is passed to the subroutine in R13. All other conditions store revert code directly into the DSA for the particular type of on-unit.

Errors Detected

ERROR AT '___'. (16)

Local Variables

None

Program Interface

Entry Points

$CRVT. No formal parameters. Expects $PTR to point to token following REVERT.

Exit Conditions

Normal exit only. $PTR points to semicolon.

Routines Called

$BONSA	On-Unit Specification Analyzer
$GTRIAD	Get Next Triad Entry
$NCONS	Constant Processor
$XERR	Error Message Editor

Global Variables

T Table	Token Table
$PTR	Token Table Pointer
$FILON	On-Unit Flag and FCIB Pointer

Logic Diagram

Chart 17 shows the detailed logic diagram for the REVERT Generator routine.

TITLE: STOP GENERATOR ($CSTOP)

Program Definition

Purpose and Usage

The STOP Generator analyzes the syntax and generates triads for the STOP statement.

Description

The functions of a STOP statement are performed by a branch to the prologue of the external procedure.

Errors Detected

ERROR AT '___'. (16)

Local Variables

None

Program Interface

Entry Points

$CSTOP. No formal parameters. Expects $PTR to point to token following STOP.

Exit Conditions

Normal exit only. $PTR points to semicolon token.

Routines Called

$GTRIAD	Get Next Triad Entry
$XERR	Error Message Editor

Global Variables

B Table	Block Information Table
T Table	Token Table
$PTR	Token Table Pointer

Logic Diagram

Chart 18 shows the detailed logic diagram for the STOP Generator routine.

TITLE: DO-LOOP TRIAD BUILDER ($DEXP)

Program Definition

Purpose and Usage

The DO-Loop Triad Builder is used to build the triads required for an iterative DO-loop to initialize the DO index, to increment the DO index, and to test the DO index for loop termination.

Description

The elements of the DO statement along with their attributes are set on entrance to this routine in $DOLHS (the DO index), $DORHS (the initial value), $DOBY (the increment value), and $DOTO (the final value). If the increment was not specified in the DO statement, the constant one is substituted. Only the required portions of the DO-loop control code are generated. Dummy statements are constructed from the elements of the DO statement by placing the operators and operands in the operator and operand stacks. The stacks are then processed to form the required triads.

Errors Detected

None

Local Variables

DEFLG	Increment type flag: 0 = constant increment 1 = variable increment
DETEST	Used to resolve FIB's to the code that tests for the end of the loop. Contains the head pointer to the FIB's which branch to this code.
DEPLUS	Used to resolve FIB's to the code that tests for the end of the loop if the increment is plus. Contains the head pointer to the FIB's that branch to this code.

Program Interface

Entry Points

$DEXP

Calling Conditions

All communication is through the global variables.

Exit Conditions

Control returns to the caller immediately following the invoking call. No specific output values are returned.

Routines Called

$WEXP	Segment Management
$NCONS	Constant Processor
$NOPRT	Operator Stack Processor
$WCTCT	Segment Management
$GTRIAD	Get Next Triad Entry

Global Variables

$CSS	Compound Statement Switch
$DISPL	Displacement from Variable Table Address to Fixed Tables
$NTCUR	Number of Last Triad Generated
$PRIOR	Operator Priority
$DORHS	DO Statement Right-Hand Side
$DOLHS	DO Statement Left-Hand Side
$DOSWT	DO Statement Switch
$DOBY	DO BY Clause
$DOTO	DO TO Clause
A List	Dictionary Attribute List
C Table	Constant Table
P Table	Program Structure Table
X Table	Operator Stack
Y Table	Operand Stack
Z Table	Triad Table

Logic Diagram

Chart 19 shows the detailed logic diagram for the DO-Loop Triad Builder routine.

TITLE: DO GENERATOR ($DOG, $DOG2)

Program Definition

Purpose and Usage

The DO Generator generates triads to perform the functions indicated by the statement, checks the syntax of the statement, and creates an entry in the program structure table for the statement.

Description

If the specification is empty, a noniterative DO entry is created in the program structure table. No code is generated for this form of a noniterative DO statement.

If the specification contains only a WHILE clause, an iterative DO entry is created. The address of the test instructions is established, and the logical expression is evaluated and tested.

If an assignment to an index variable is given, the address and type of the index variable are obtained and saved. The initial value expression is evaluated, and its type and address are saved.

Any BY or TO clauses are then evaluated and their values are assigned to temporary variables. If more than one clause of each type is present, an error is generated. If no BY clause was given, a BY value of 1 is provided, unless no TO value was given (in which case the assignment is noniterative and special analysis follows the assignment).

The DO-Loop Triad Builder routine ($DEXP) generates the setting, incrementing, and testing instructions associated with the DO.

If a WHILE specification is present, code is generated for the logical expression and the value tested. After this, or if no WHILE was given, the program structure table is pushed down and a return performed.

All iterative DO's are repeated unless only an assignment and a WHILE clause are present. In this case the iterative DO is nonrepeating.

Code is generated only for those sections of the DO that are present. The triad table for a complete DO will look like this:

```
                  Calculate address of index variable
                  Calculate starting value expression (exp1)
                  Calculate TO value expression (exp2)
                  Calculate BY value expression (exp3)
                  Assign exp1 to index variable
    Inc           Define address - Push down temp level
                  Add exp3 to index variable
    Test          RFIB
                  Compare exp3 to zero
                  FIB  if negative go to Neg test
                  Compare index to exp2
                  FIB  less than or equal to loop
                  FIB  end of loop
    Neg test      RFIB
                  Compare index to exp2
                  FIB  greater than or equal to loop
                  FIB  end of loop
    Loop          RFIB
                  Calculate WHILE clause
                  FIB  false to end of loop
```

Errors Detected

ERROR AT '___'. (16)
ILLEGAL 'DO' INDEX. (22)
ILLEGAL 'WHILE' CLAUSE. (25)
DUPLICATE '___' CLAUSE. (26)
ITERATIVE 'DO' REQUIRED. (38)

Local Variables

DOTYPE	Check for Proper DO Terminator

Program Interface

Entry Points

$DOG. Expand DO statement. Expects $PTR to be set to token after DO.

$DOG2. Expand DO specification in I/O statement. Expects $PTR to be set to token after DO.

Exit Conditions

Upon encountering a semicolon token or a right-parenthesis token. Both exit points leave $PTR set to token that caused exit.

Routines Called

$NEXP	Expression Processor Controller
$DEXP	DO-Loop Triad Builder
$GTRIAD	Get Next Triad Entry
$AREXP	Array Expression Error
$XERR	Error Message Editor
$WCTCT	Segment Management
$WEXP	Segment Management

Global Variables

T Table	Token Table
$PTR	Token Table Pointer
P Table	Program Structure Table
$DOLHS	DO Statement Left-Hand Side
$DORHS	DO Statement Right-Hand Side
$DOTO	DO TO Clause
$DOBY	DO BY Clause
$DOSWT	DO Statement Switch

Logic Diagram

Chart 20 shows the detailed logic diagram for the DO Generator routine.

TITLE: RETURN GENERATOR ($DRET)

Program Definition

Purpose and Usage

The RETURN Generator analyzes the syntax and generates the code necessary for the RETURN statement.

Description

RETURN Generator makes sure that the RETURN statement is not contained within an on-unit and that, if immediately contained in the external procedure, it has no return expression. If it is in a begin block, it is treated as appearing in the containing procedure.

The return expression, if any, is calculated and stored in the dummy parameter passed for the return value. Code is generated to branch to the procedure block's epilogue.

Errors Detected

ERROR AT '___'. (16)
RETURN STATEMENT ILLEGAL IN ON-UNIT. (51)
RETURNS ATTRIBUTE ILLEGAL IN EXTERNAL PROCEDURE--SKIPPING TO ';'. (52)

Local Variables

DRTKN	Return Assignment Tokens

Program Interface

Entry Points

$DRET. No formal parameters. Expects $PTR to point to token following RETURN.

Exit Conditions

Normal exit only. $PTR points to semicolon token.

Routines Called

$NEXP	Expression Processor Controller
$GTRIAD	Get Next Triad Entry
$AREXP	Array Expression Error
$XERR	Error Message Editor
$WBACK	Segment Management

Global Variables

T Table	Token Table
$PTR	Token Table Pointer
P Table	Program Structure Table
B Table	Block Information Table

Logic Diagram

Chart 21 shows the detailed logic diagram for the RETURN Generator routine.

TITLE: END GENERATOR ($EDGN, $EDGN2)

Program Definition

Purpose and Usage

The END Generator determines which, and how many, DO, BEGIN, and PROC statements are being closed and whether any IF statements are being determined. It also generates the triads and performs the housekeeping associated with the closings.

Description

If no label follows the word END, then the statement on top of the program structure table is being terminated. If a label follows END, then each statement down to and including the statement with that label is terminated. If a label prefix is present on the END statement, its processing must be delayed until the last group or block is ended.

When a begin or procedure block is being closed, some work, besides generating the epilogue code, needs to be done. If the external procedure is being closed, compilation is completed and it is necessary to finish housekeeping tasks such as establishing static storage and library linkage. Once this work is completed, the object program can be executed.

At the end of an internal block, all declarations made in the block are removed from the dictionary attribute list (A list). This is a simple procedure for variables but more complex for entry names and statement-label constants. At the end of each procedure, all temporaries for the procedure are removed from the temporary storage table (S table).

If the dictionary attribute list shows entry names or statement-label constants tentatively, rather than explicitly defined in a block being closed, it is necessary to search for an explicit definition in the immediately containing block. If no explicit definition is found, the tentative definition is changed to an explicit definition in the containing block.

If this tentative definition is for a FORMAT statement label, no search is made of the containing block and an error message for an undefined format is generated. All explicit entry-name or statement-label-constant definitions are removed from the dictionary attribute list.

Each ENDFILE on-unit encountered in the block being ended requires eight bytes in the DSA for the block to build the ENDFILE table. This area is initialized by a call to the IHEONUN entry point of the On-ENDFILE and REVERT Initializer (IHEONREV) at runtime.

Errors Detected

ERROR AT '___'. (16)
'___' AFTER 'END' ILLEGAL--IGNORED. (27)
UNDEFINED FORMAT. (28)
ILLEGAL USE OF '___'. (29)
SOURCE STMTS AFTER END OF PROGRAM IGNORED. (107)
SEVERE DIAGNOSTICS, EXECUTION PREVENTED. (108)

Local Variables

EDLAST	Last Statement Switch (0 = No)
EDENTY	Entry Point Code
EDFIB	FIB Around Symbol Table
EDBLKT	Block End Switch:

X '01'	RFIB Around Block
X '40'	External Procedure
X '20'	Begin Block
X '80'	Symbol Table Switch

Program Interface

Entry Points

$EDGN. No formal parameters. Expects $PTR to point to token following END.

$EDGN2. No formal parameters. Does not use token table.

Exit Conditions

Normal exit. $PTR points to semicolon and also end of compilation exit.

Normal exit only. One iterative DO closed.

Routines Called

$CERR	Compiler Error
$BLPRC	Label Processor
$ENDES	Process End of ELSE
$ENDON	Process End of ON
$GTRIAD	Get Next Triad Entry
$TCODE	Triad Code Generator
$FNB	Get Non-Blank
$SVC	SVC Director
$XERR	Error Message Editor
$WCTCT	Segment Management
$WEXP	Segment Management

Global Variables

$PTR	Token Table Pointer
T Table	Token Table
P Table	Program Structure Table
A List	Dictionary Attribute List
N List	Dictionary Name List
$CSS	Compound Statement Switch
B Table	Block Information Table
S Table	Temporary Storage Table

Logic Diagram

Chart 22 shows the detailed logic diagram for the END Generator routine.

TITLE: PROCESS END OF ELSE ($ENDES)

Program Definition

Purpose and Usage

The Process End of ELSE routine does the processing necessary at the end of an ELSE unit.

Description

If necessary, the address of the end of the IF statement is defined. The program structure table entry for the IF statement is removed and the compound statement switch reset.

Errors Detected

None

Local Variables

None

Program Interface

Entry Points

$ENDES. No formal parameters.

Exit Conditions

Normal exit only.

Routines Called

$GTRIAD	Get Next Triad Entry
$WCTCT	Segment Management

Global Variables

P Table	Program Structure Table
$CSS	Compound Statement Switch

Logic Diagram

Chart 23 shows the detailed logic diagram for the Process End of ELSE routine.

TITLE: PROCESS END OF ON ($ENDON)

Program Definition

Purpose and Usage

The Process End of ON routine does the processing necessary at the end of an on-unit.

Description

The address of the end of the on-unit is defined and the entry in the temporary storage table made for the ON statement is removed. The program structure table entry for the ON statement is removed and the compound statement switch reset.

Errors Detected

None

Local Variables

None

Program Interface

Entry Points

$ENDON. No formal parameters.

Exit Conditions

Normal exit only.

Routines Called

$GTRIAD	Get Next Triad Entry
$CERR	Compiler Error
$WCTCT	Segment Management

Global Variables

P Table	Program Structure Table
$CSS	Compound Statement Switch

Logic Diagram

Chart 24 shows the detailed logic diagram for the Process End of ON routine.

TITLE: EXPANDER ($EXPND)

Program Definition

Purpose and Usage

The Expander determines the dimensionality of an array expression, generates the DO statements for the expression, and builds a list of the temporary variables used for the indices.

Description

This routine is called after the Expression Processor Controller has detected the presence of an array expression. A pointer to the asterisk subscript or array name that caused detection of the array expression, a subscript number (called SUBNUM in the logic diagram), and the pointer to the dictionary attribute list entry for the array are available. The subscript number is set to zero if an array name caused detection of the array.

The Expander proceeds to remove from the triad table any triads generated by the Expression Processor Controller prior to detection. It determines if this is an array or a cross-section reference.

For each dimension, a fixed-point temporary variable is obtained for the bounds and a DO statement is generated.

If the expression is an array, the number of dimensions is checked before proceeding. If it is a cross-section, the subscripts are scanned for the next asterisk subscript. The subscript number is properly advanced during the scan.

The generated substitution list is in the form of a token list from the left parenthesis for a subscript to the right parenthesis.

Errors Detected

None

Local Variables

EXDIM	Number of Dimensions in Array

Program Interface

Entry Points

$EXPND. Expects $EXPNS and $PTR from Expression Processor Controller.

Exit Conditions

Normal exit only.

Routines Called

$DEXP	DO-Loop Triad Builder
$GTRIAD	Get Next Triad Entry
$WCTCT	Segment Management
$WEXP	Segment Management

Global Variables

T Table	Token Table
Q Table	Subscript Substitution Table
$EXPCT	Expansion Count
$EXPNS	Expression Processor to Array Expansion Routine Information

Logic Diagram

Chart 25 shows the detailed logic diagram for the Expander routine.

TITLE: END EXPAND ($EYPND)

Program Definition

Purpose and Usage

The END Expand routine generates the END statements necessary to complete DO-loops generated by the Expander routine.

Description

For each dimension in the array expression, a call is made to the END Generator to complete a DO statement. After each entry, the temporary variable used for the DO index is released and the corresponding entry in the subscript substitution table is set to null.

Errors Detected

None

Local Variables

None

Program Interface

Entry Points

$EYPND. No formal parameters.

Exit Conditions

Normal exit only.

Routines Called

$EDGN	END Generator
$WCTCT	Segment Management

Global Variables

$EXPCT	Expansion Count
Q Table	Subscript Substitution Table

Logic Diagram

Chart 26 shows the detailed logic diagram for the END Expand routine.

PART 3 LOGIC DIAGRAMS

The detailed logic diagrams for CALL/360-OS PL/I statement processors follow.

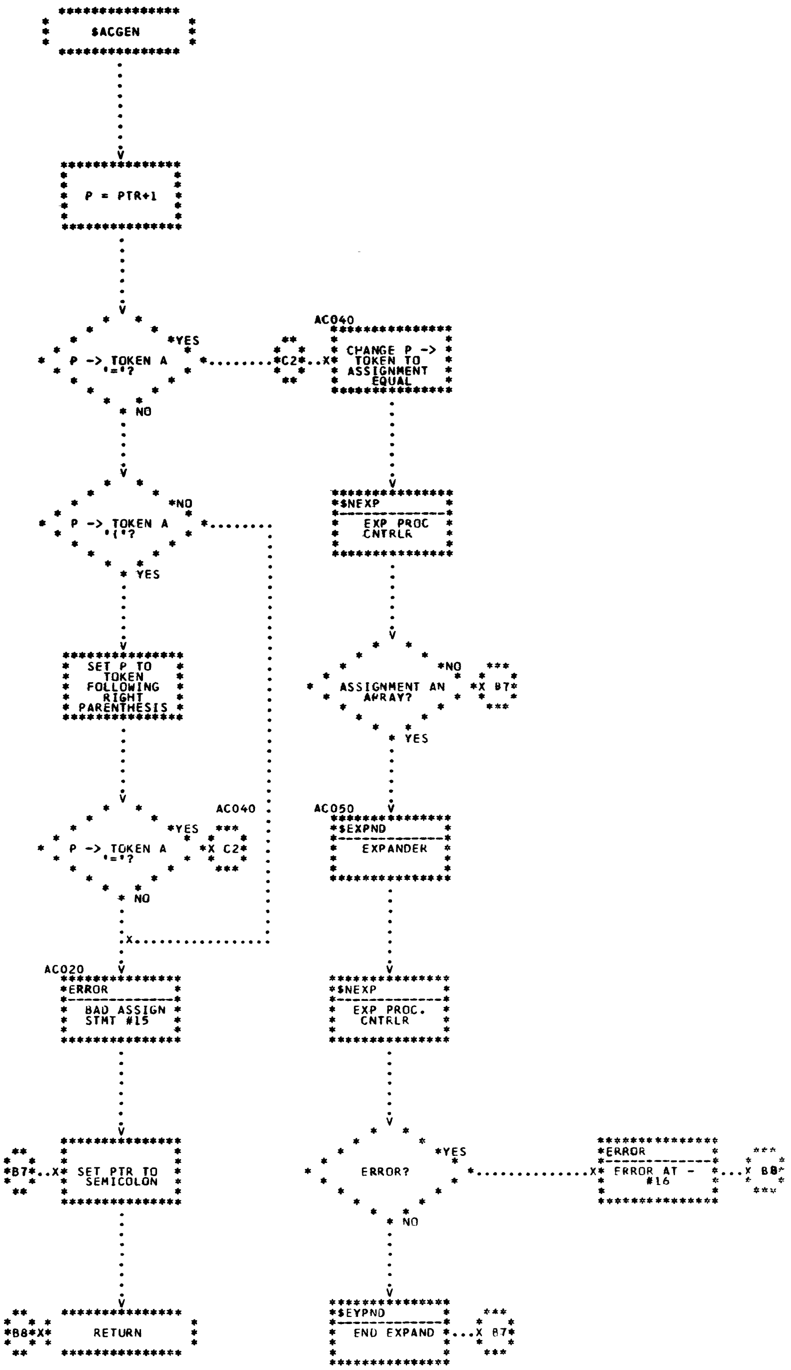


Chart 9. Assignment Generator

$APRC & $APRC2

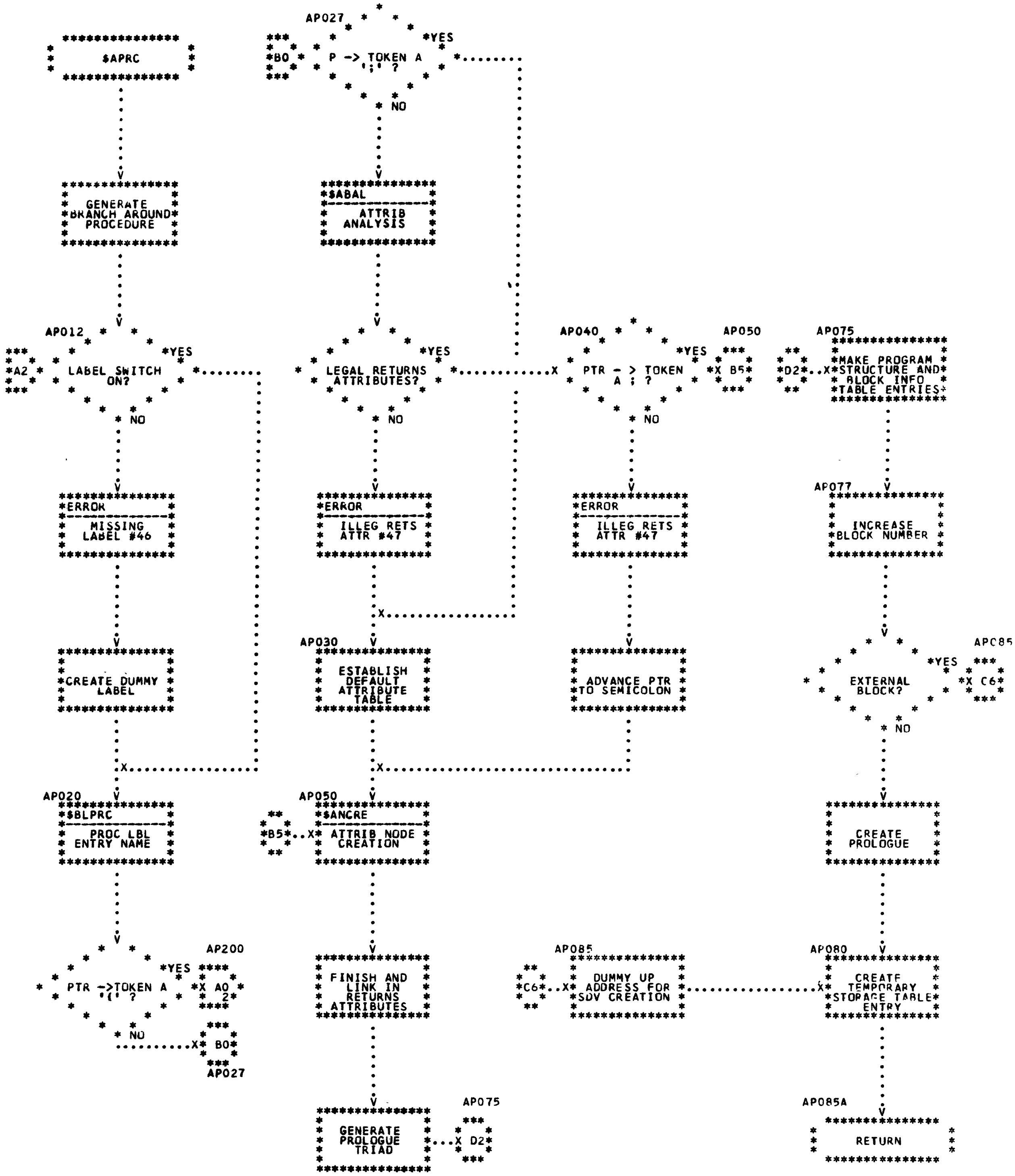


Chart 10. PROC Generator (Page 1 of 3)

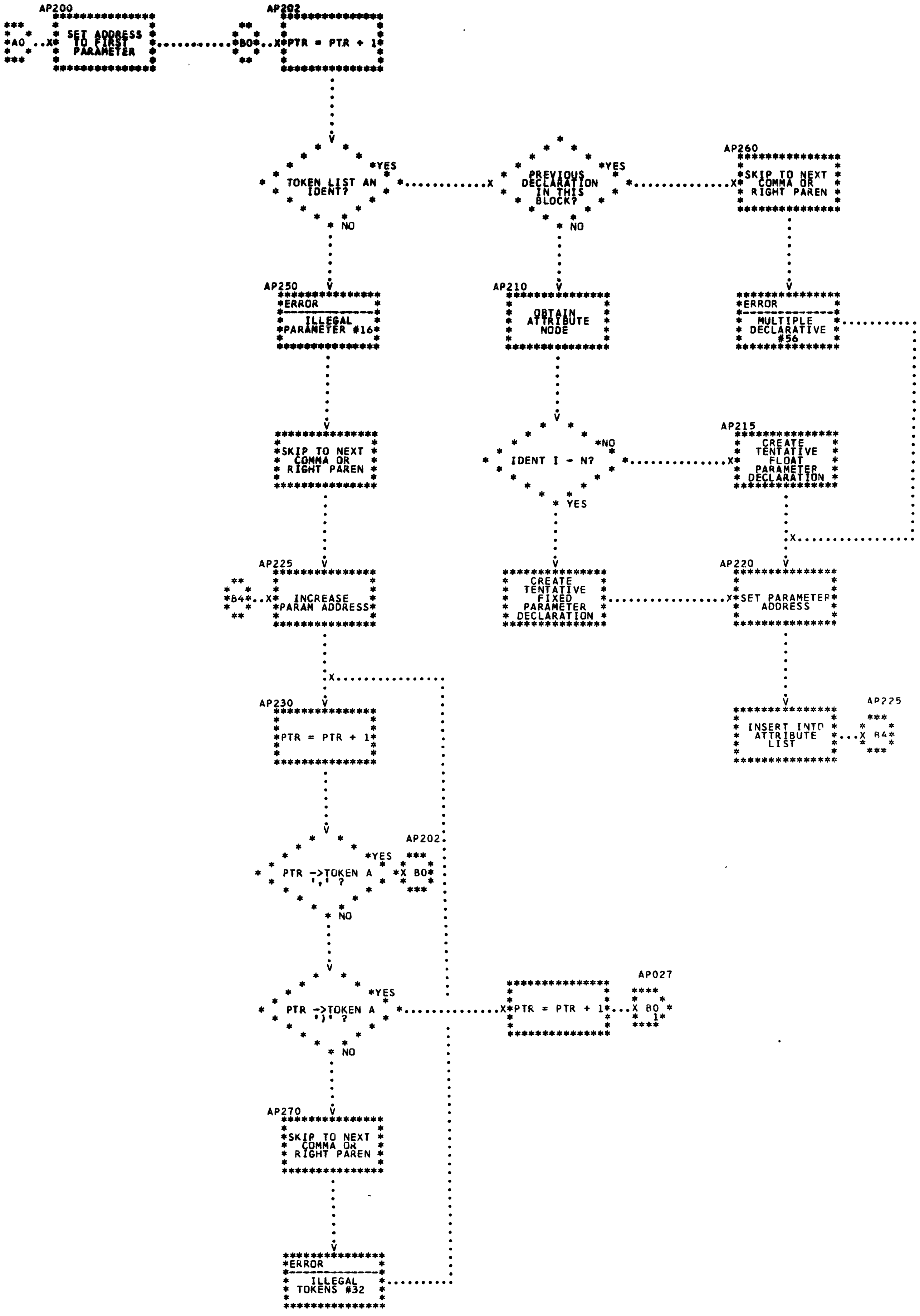


Chart 10. PROC Generator (Page 2 of 3)

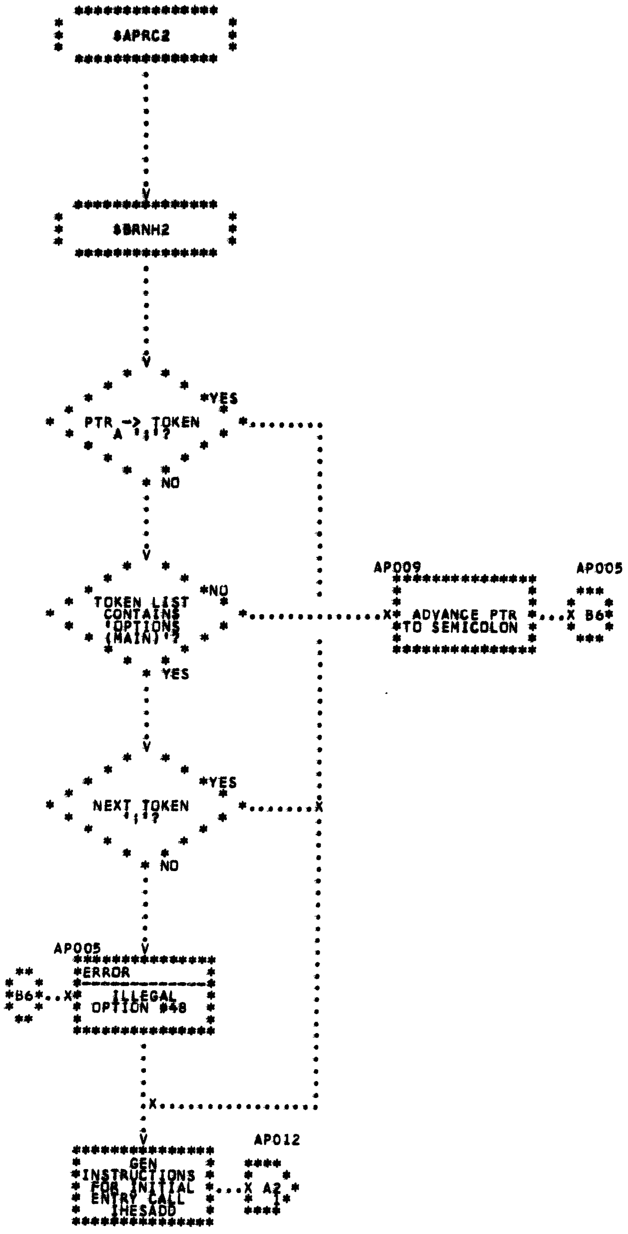


Chart 10. PROC Generator (Page 3 of 3)

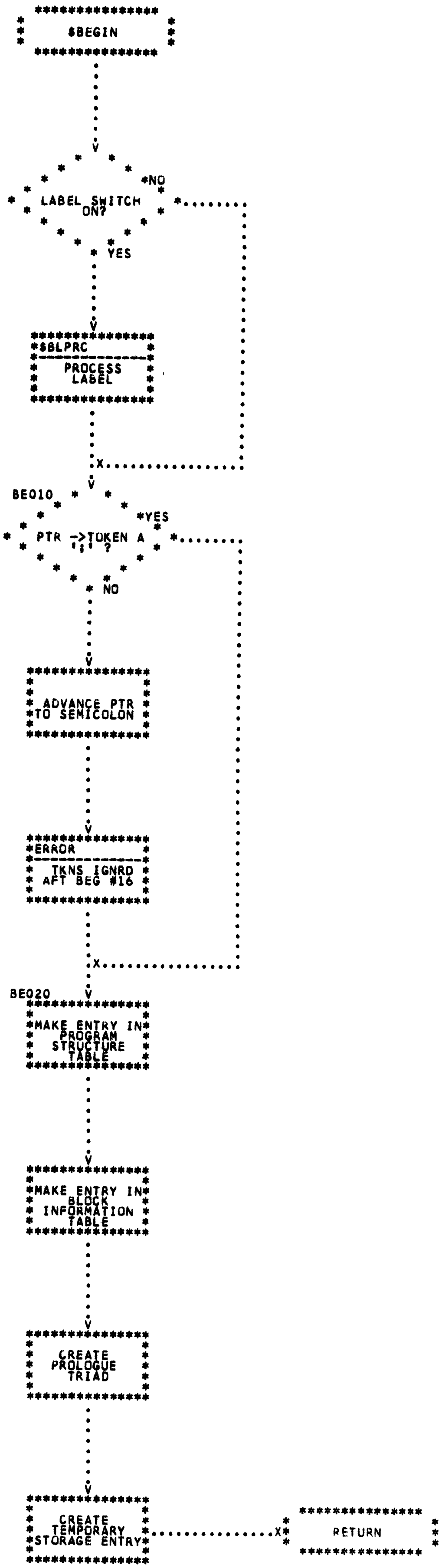


Chart 11. BEGIN Generator

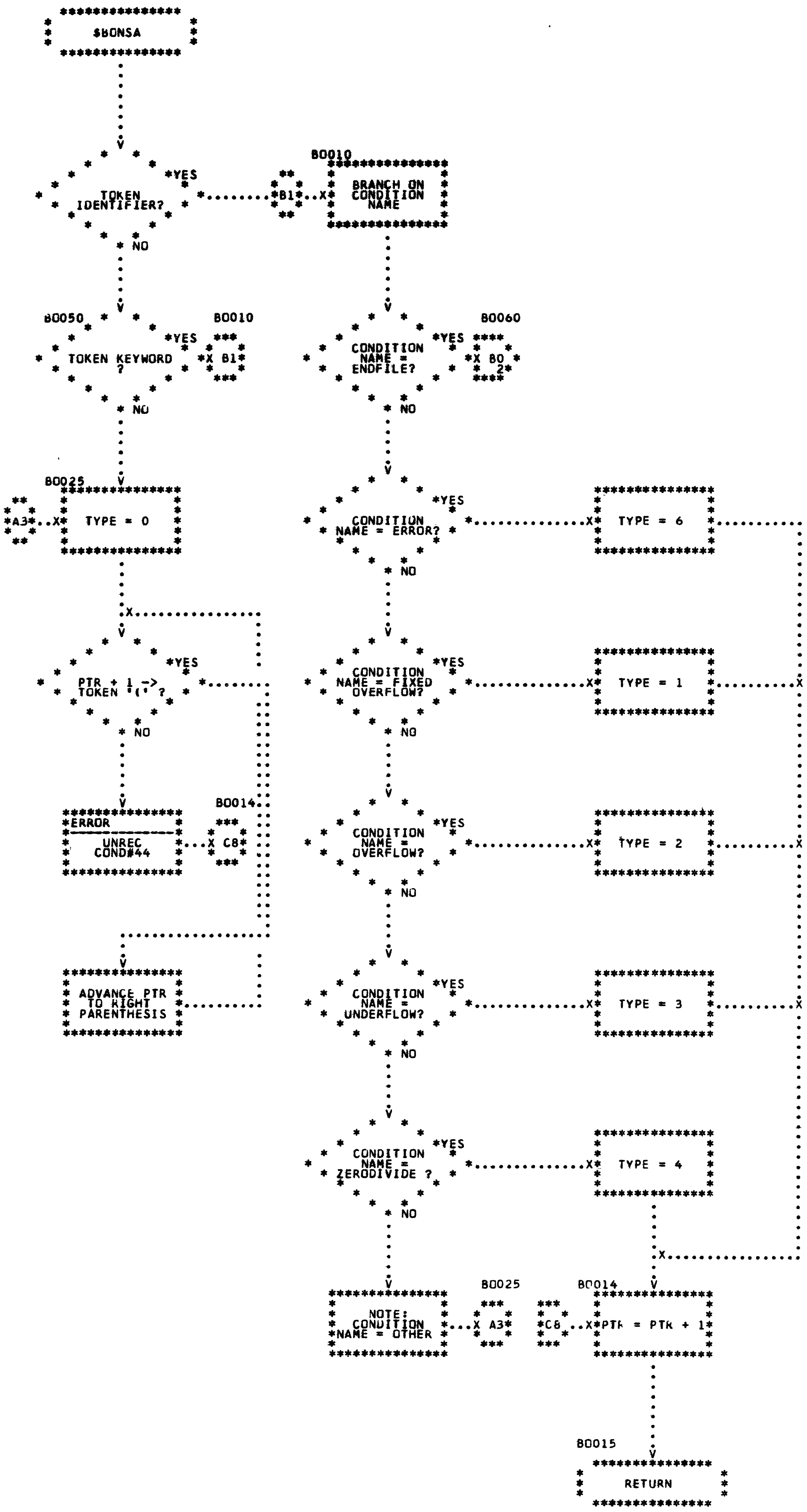


Chart 12. On-Unit Specification Analyzer (Page 1 of 2)

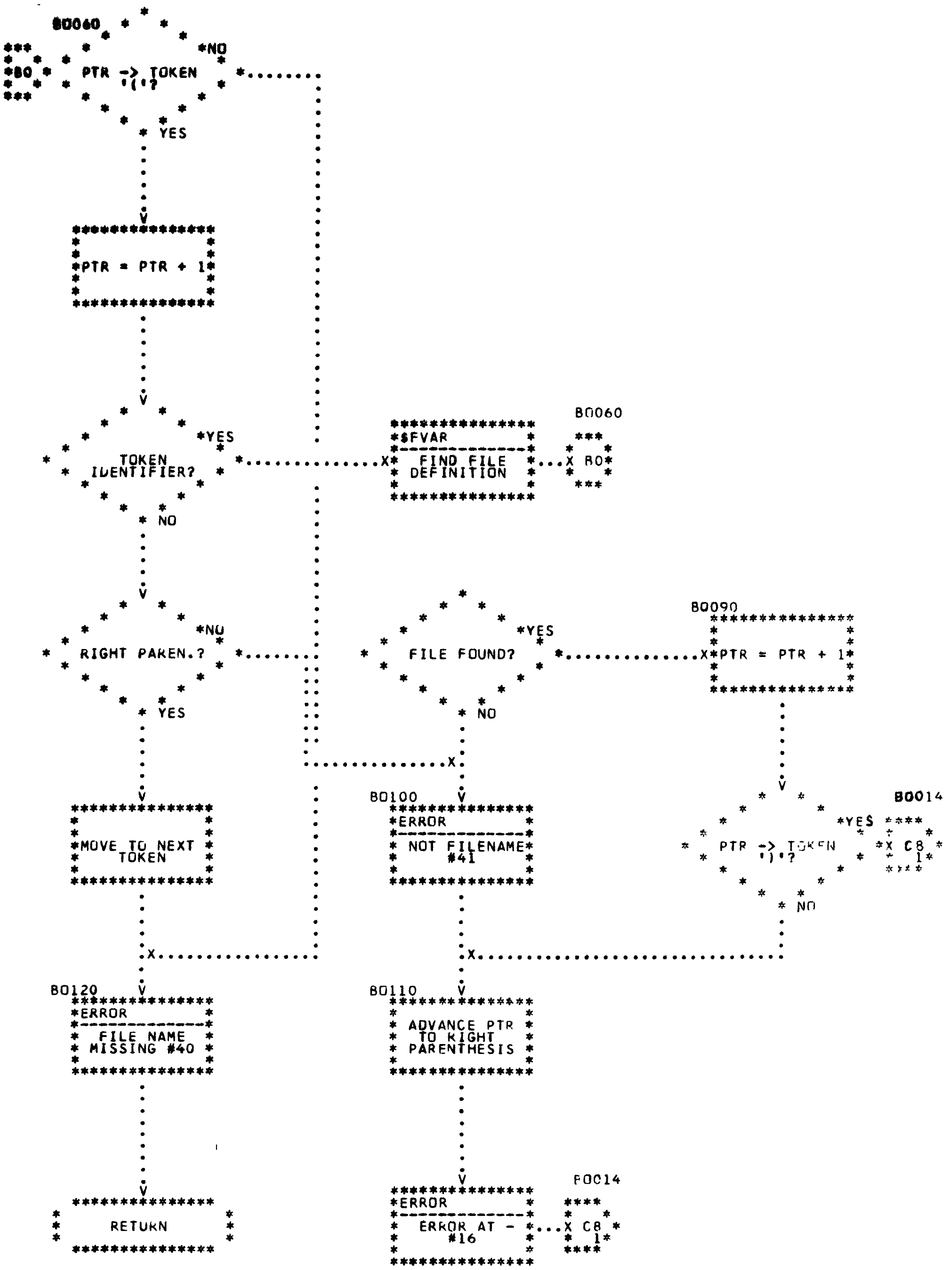


Chart 12. On-Unit Specification Analyzer (Page 2 of 2)

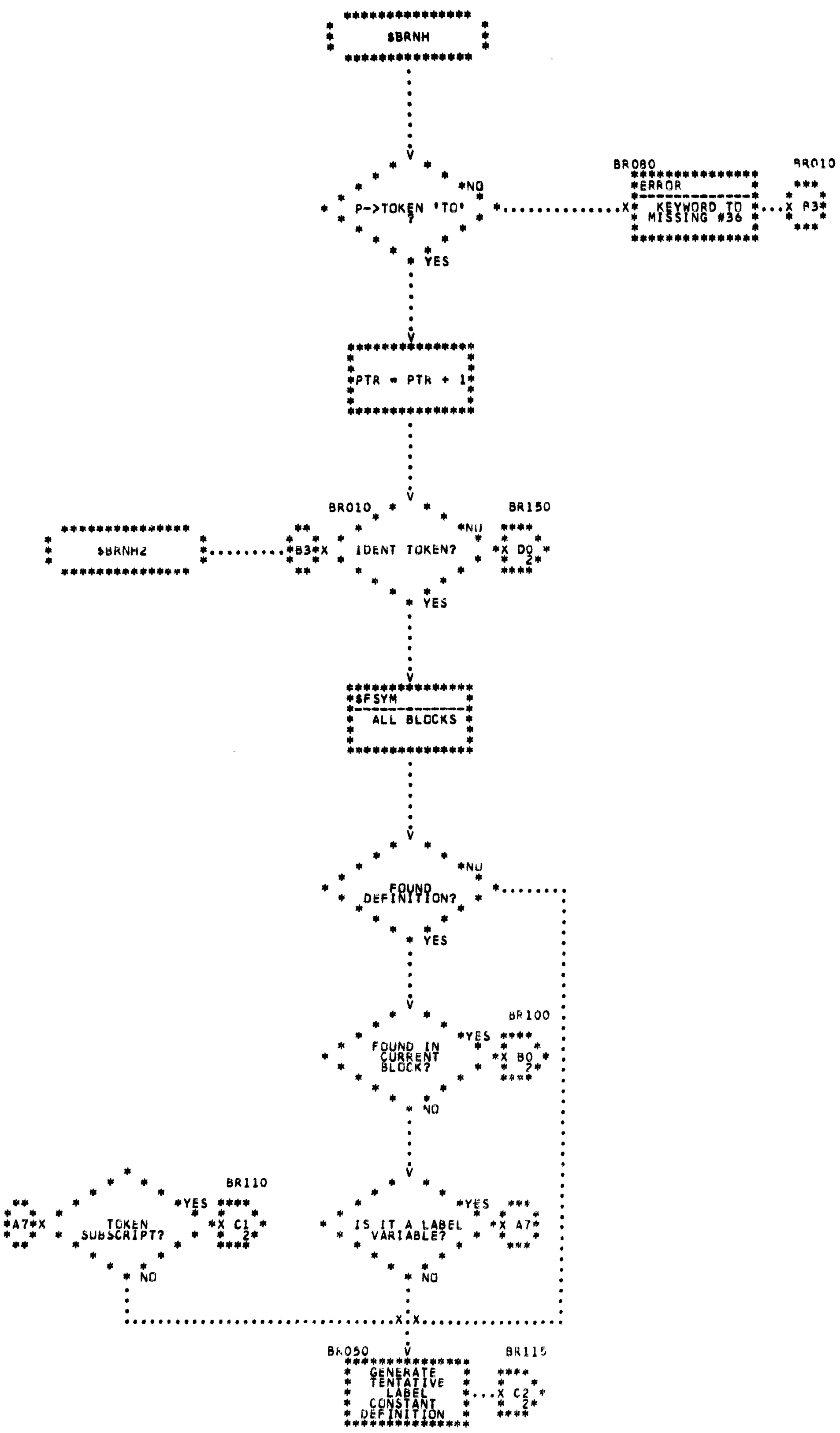


Chart 13. GO Generator (Page 1 of 2)

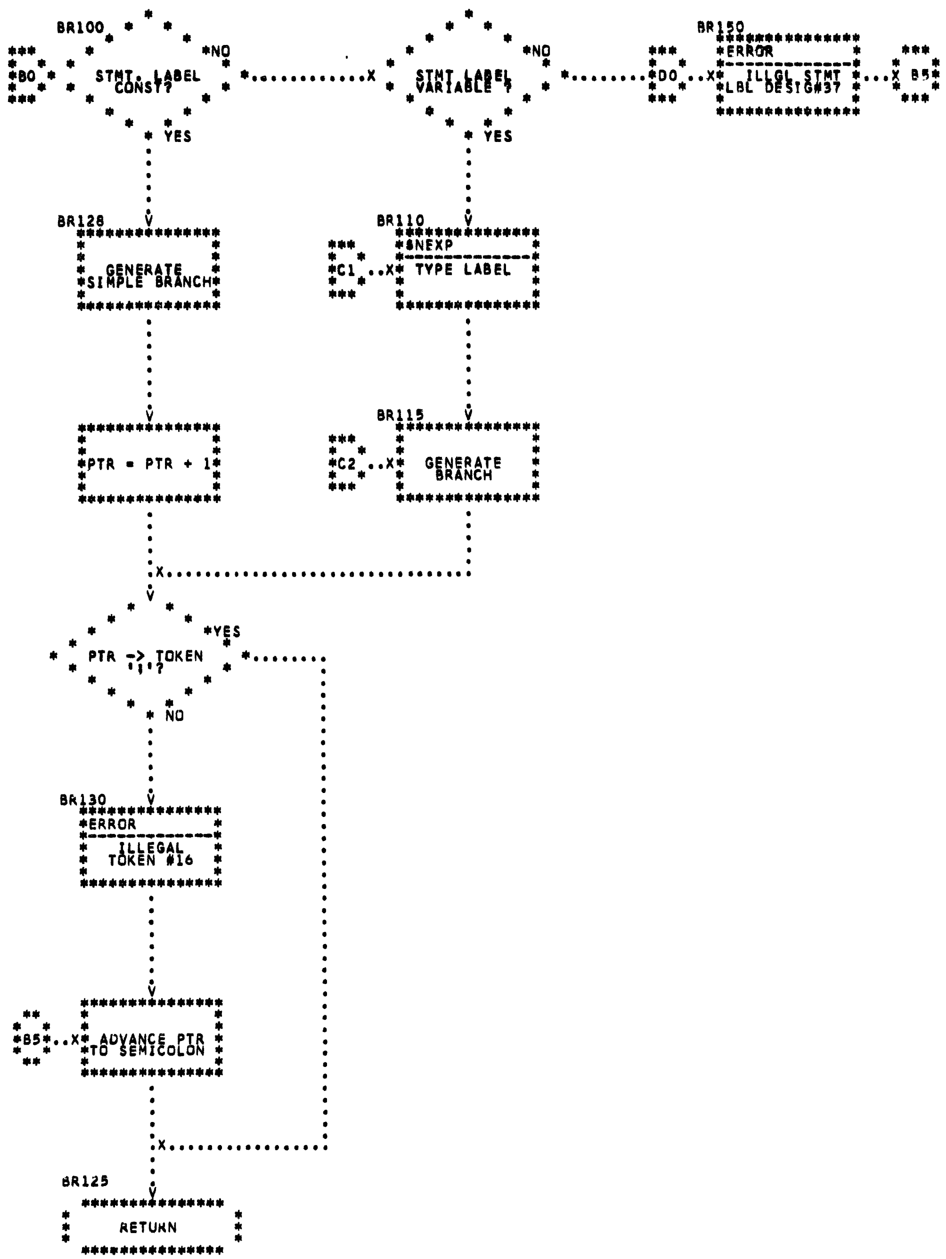


Chart 13. GO Generator (Page 2 of 2)

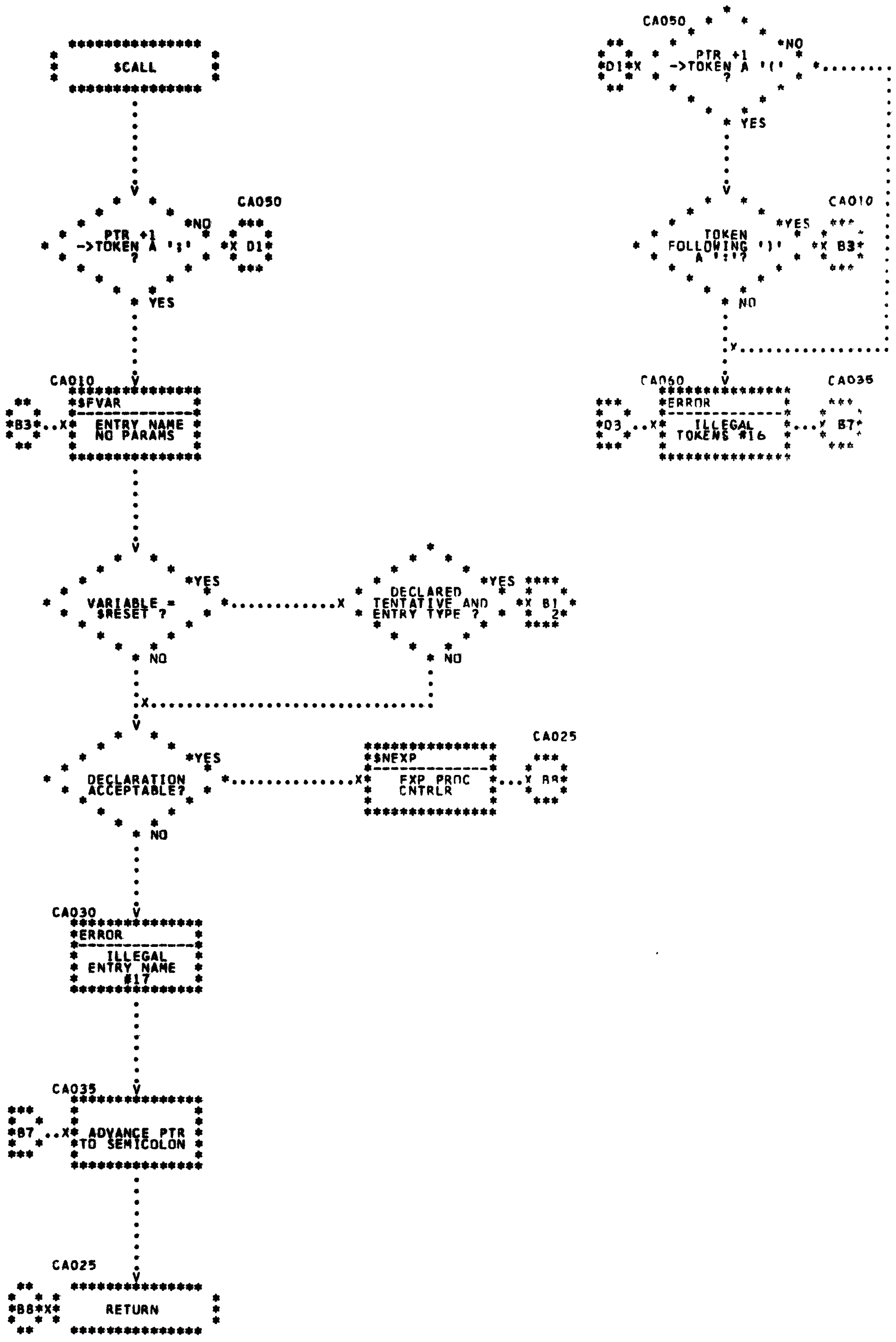


Chart 14. CALL Generator (Page 1 of 2)

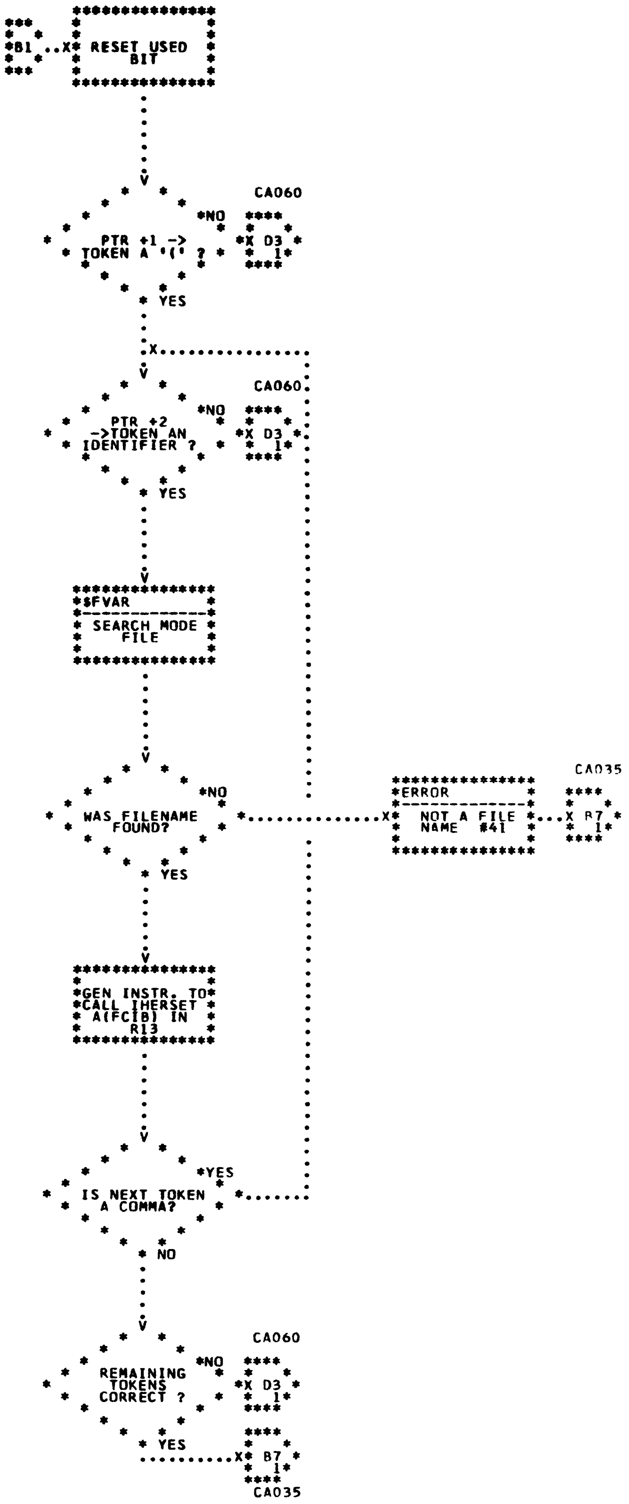


Chart 14. CALL Generator (Page 2 of 2)

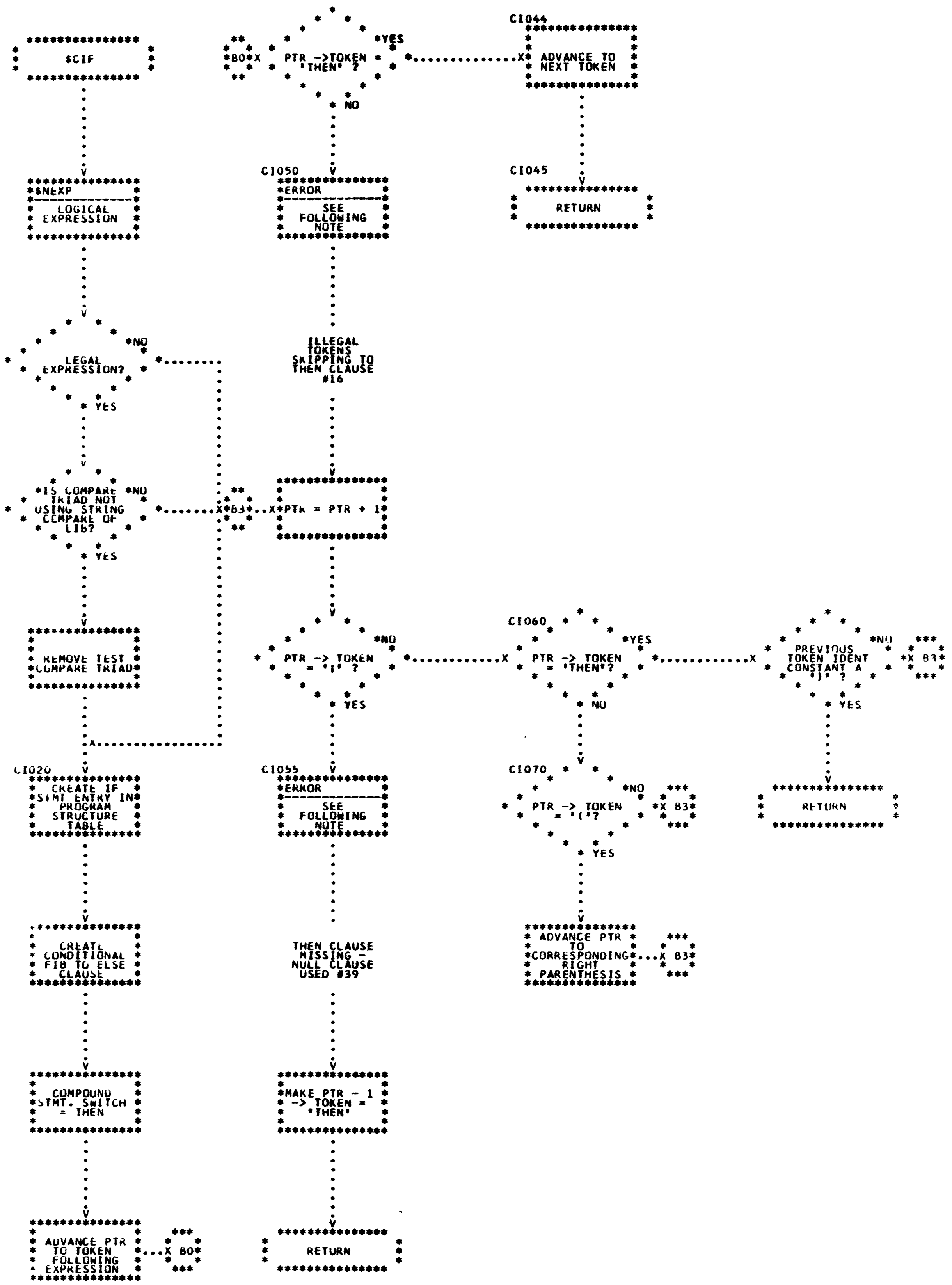


Chart 15. IF Generator

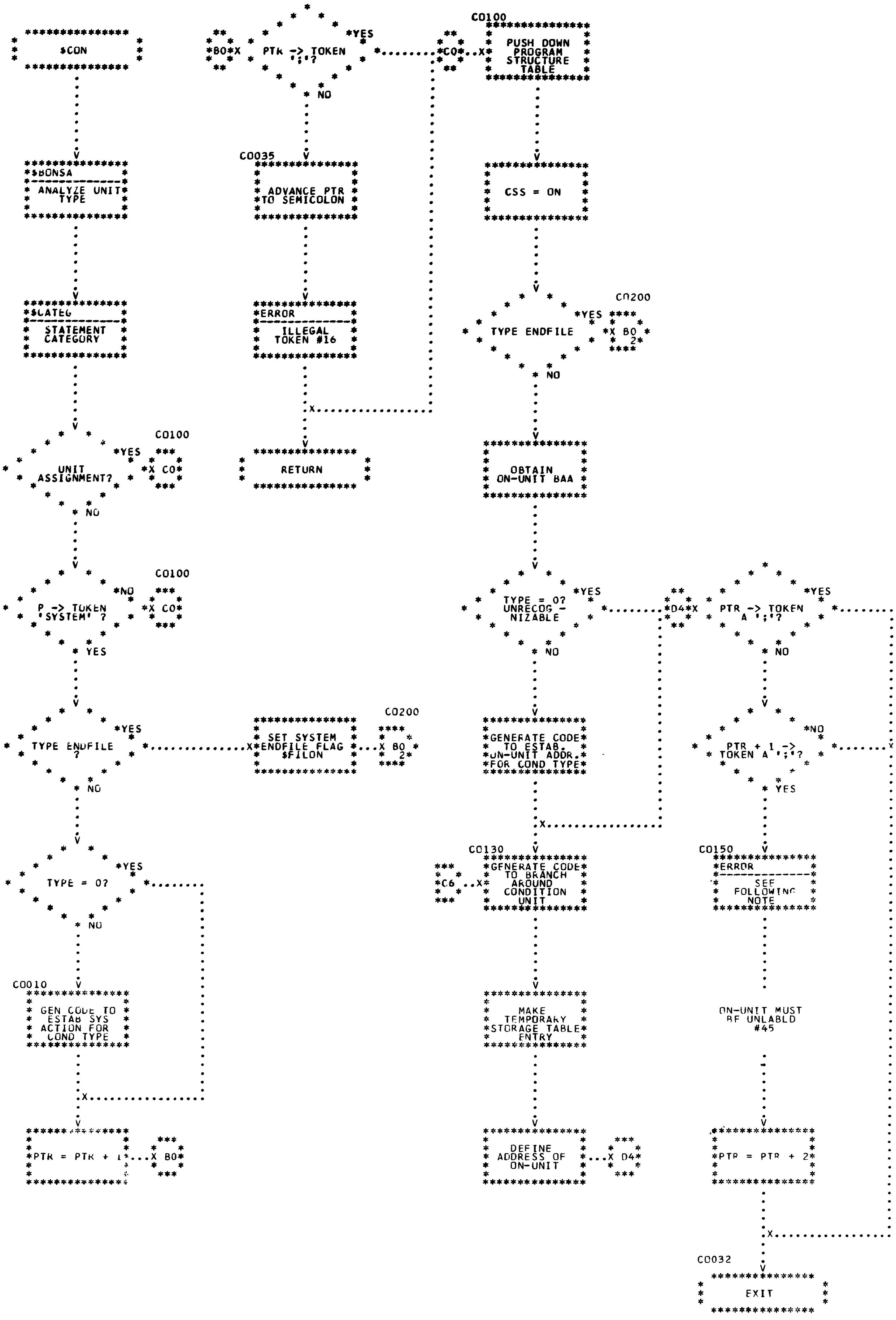


Chart 16. ON Generator (Page 1 of 2)

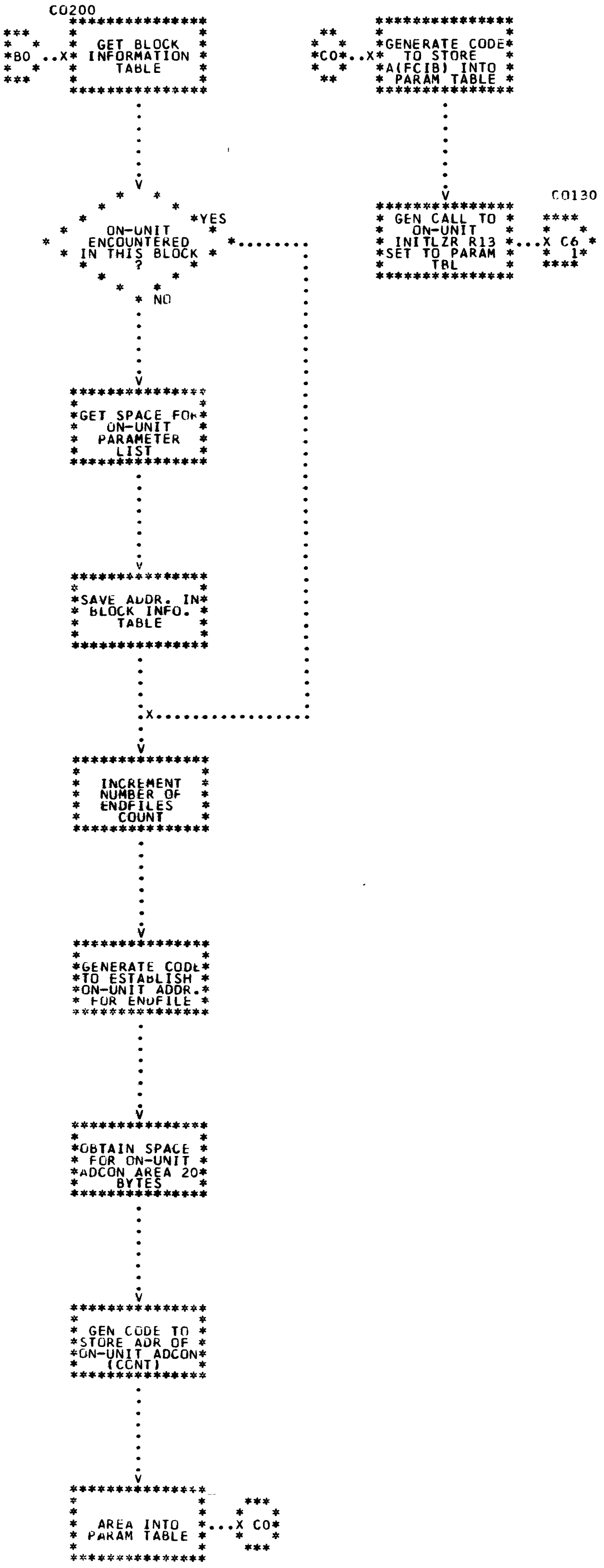


Chart 16. ON Generator (Page 2 of 2)

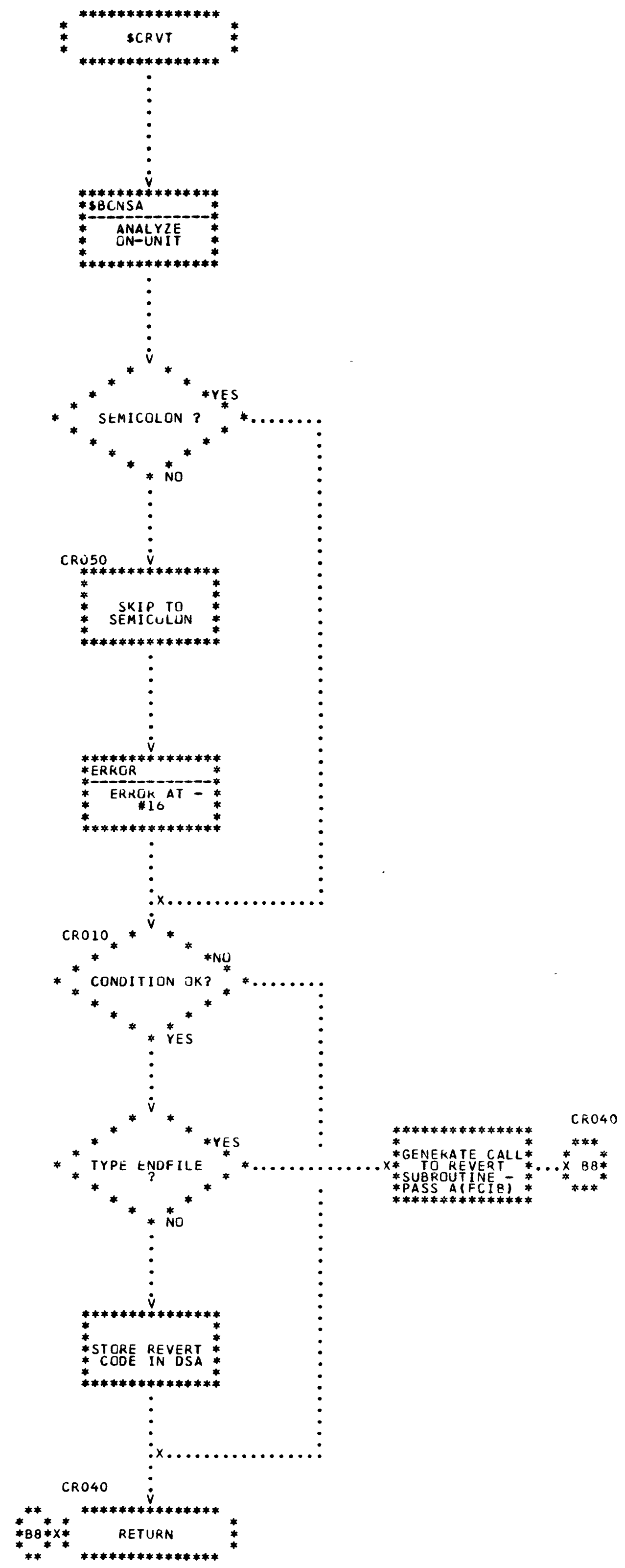


Chart 17. REVERT Generator

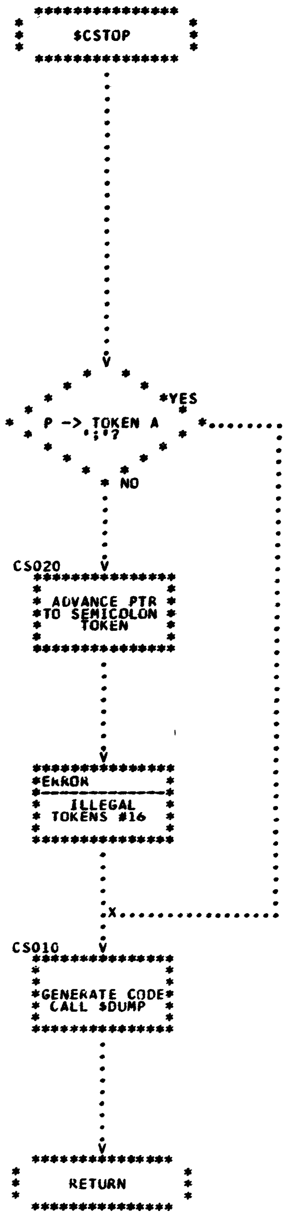


Chart 18. STOP Generator

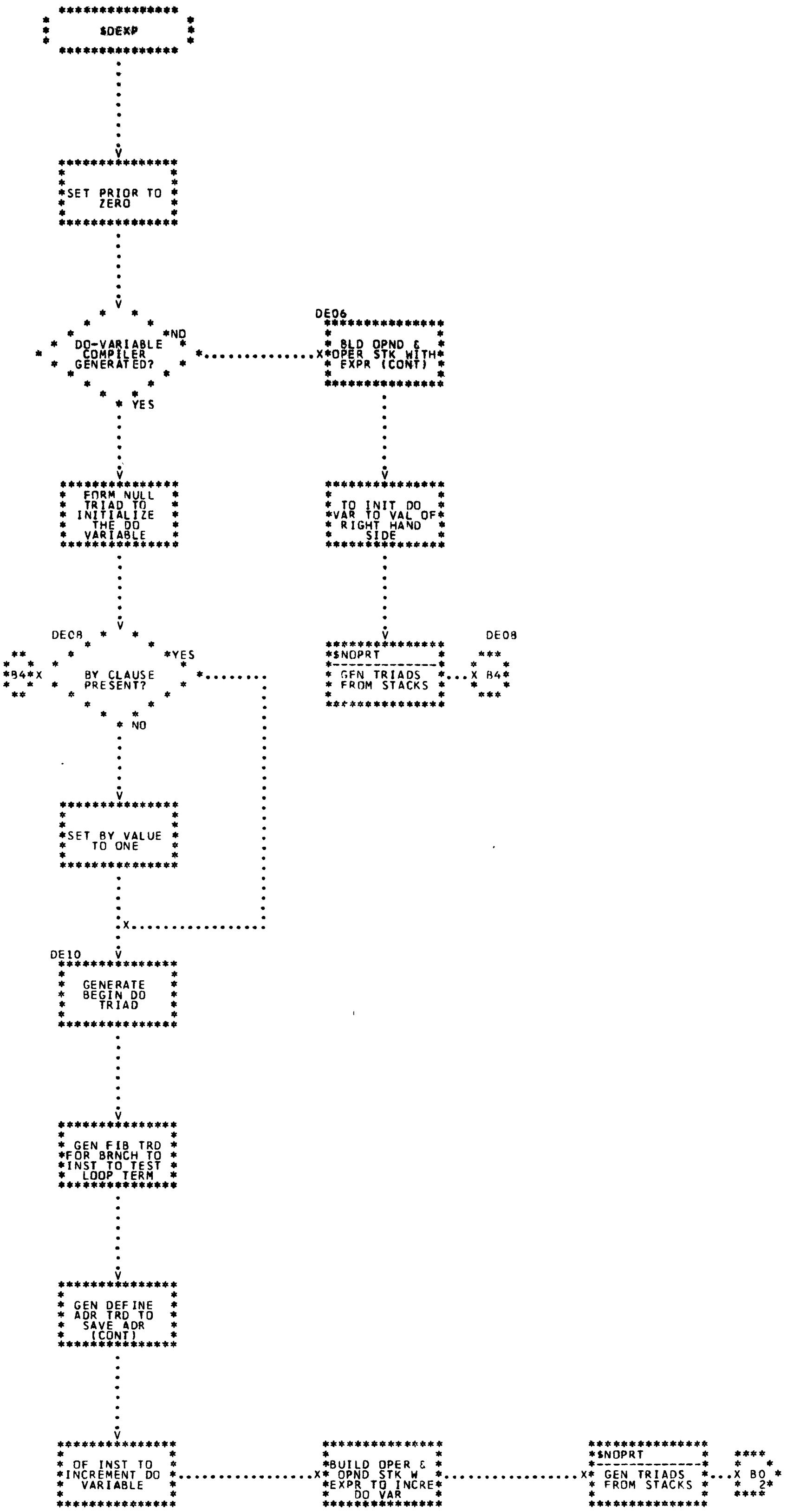


Chart 19. DO-Loop Triad Builder (Page 1 of 2)

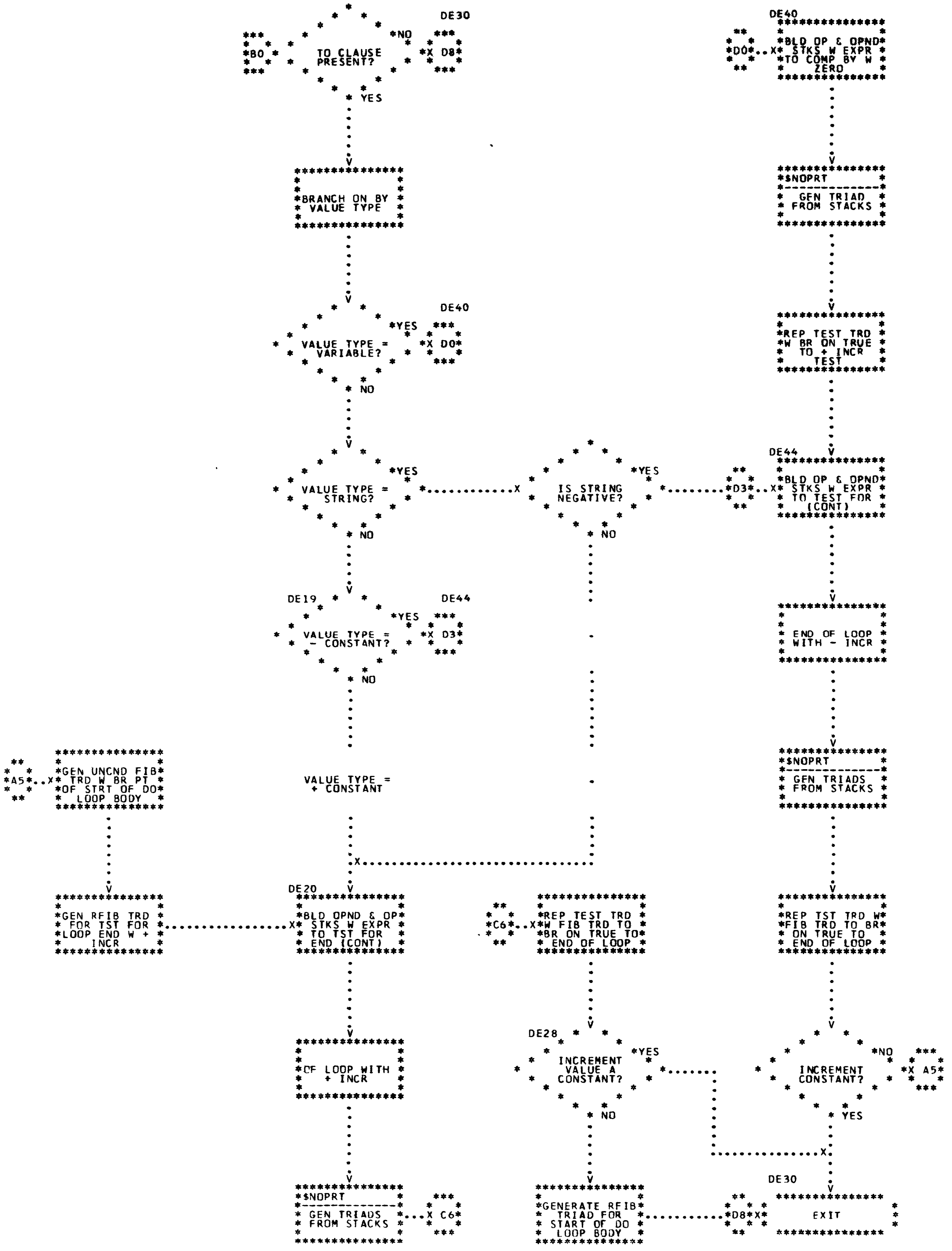


Chart 19. Do-Loop Triad Builder (Page 2 of 2)

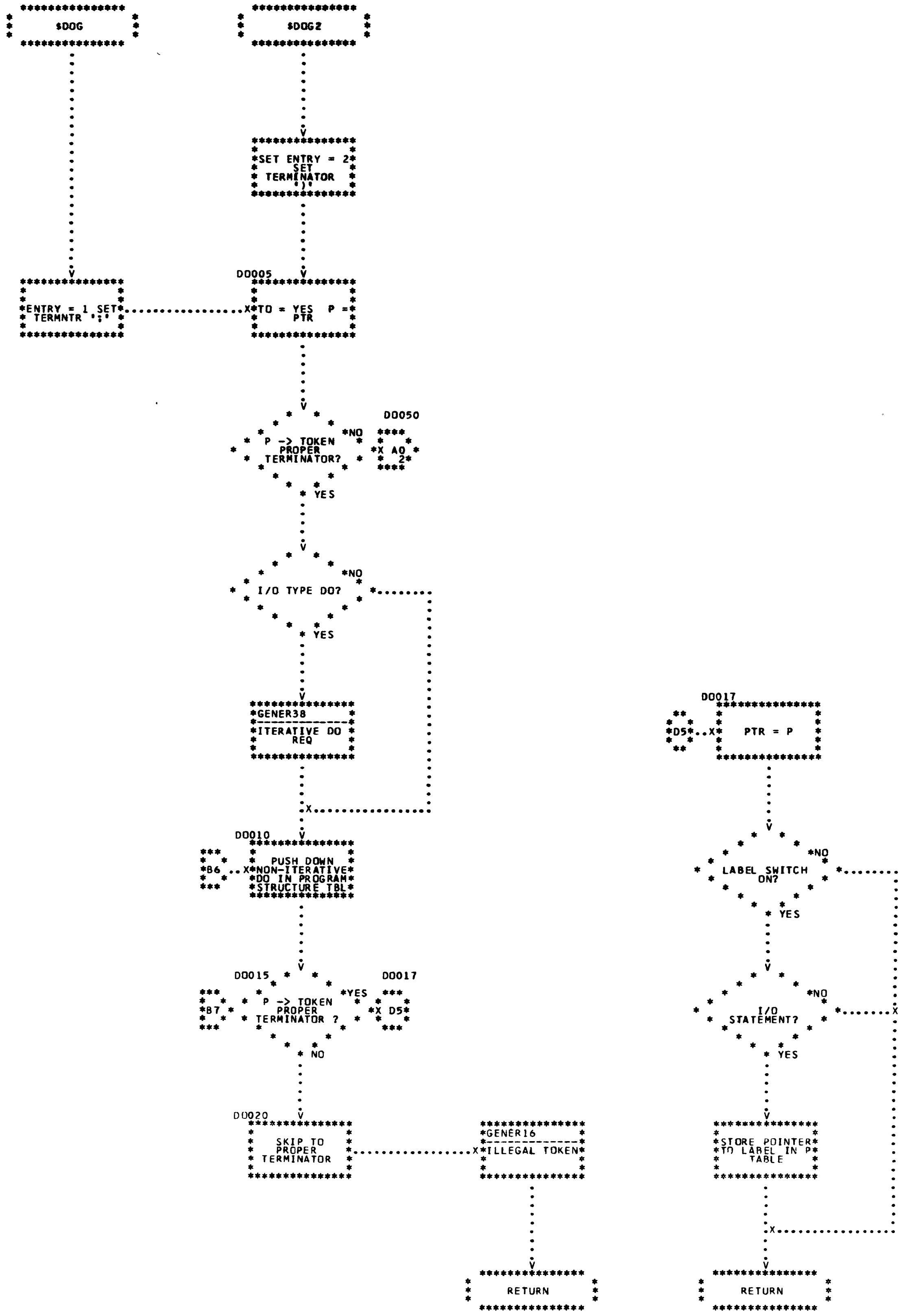


Chart 20. DO Generator (Page 1 of 4)

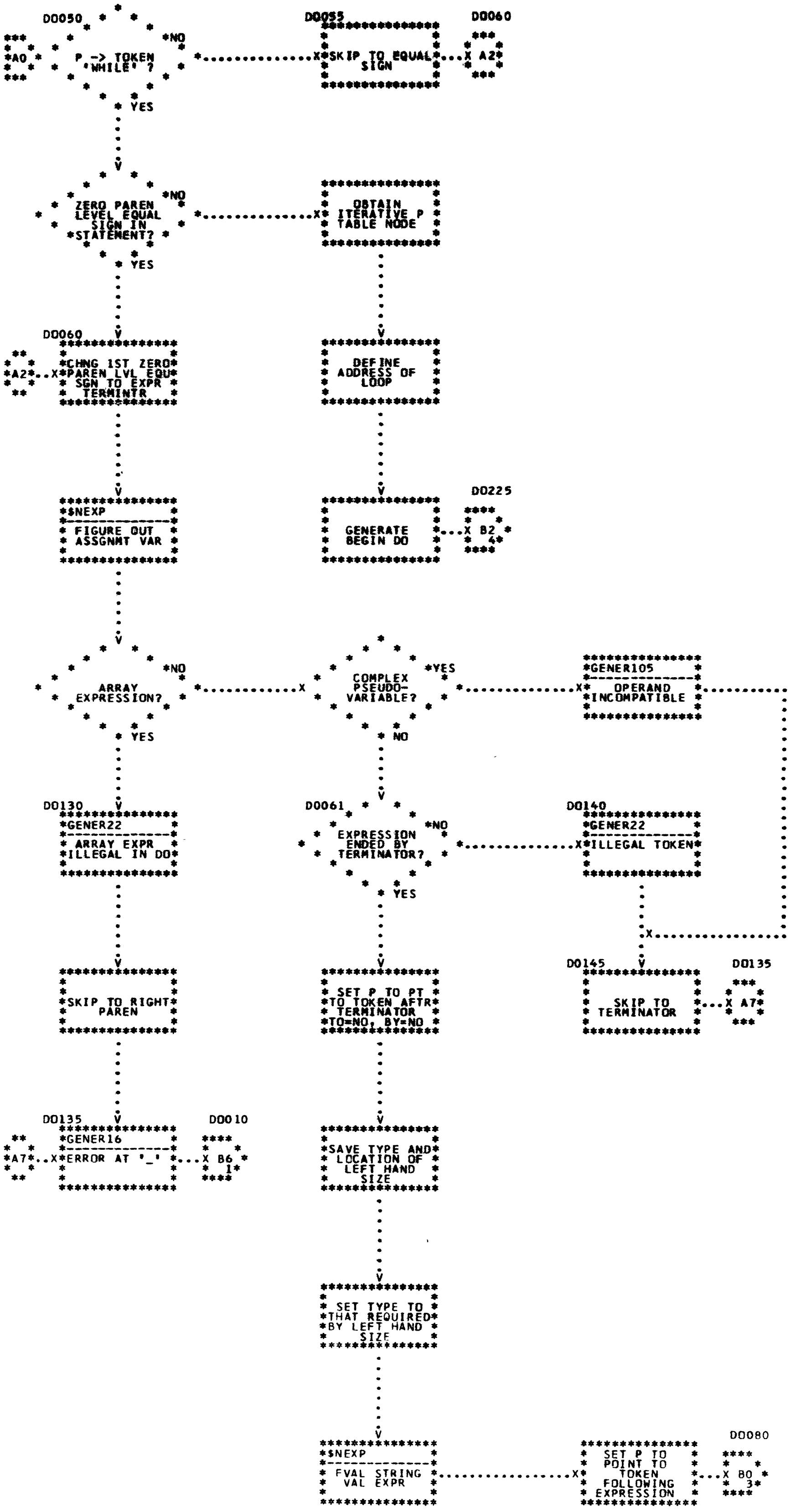


Chart 20. DO Generator (Page 2 of 4)

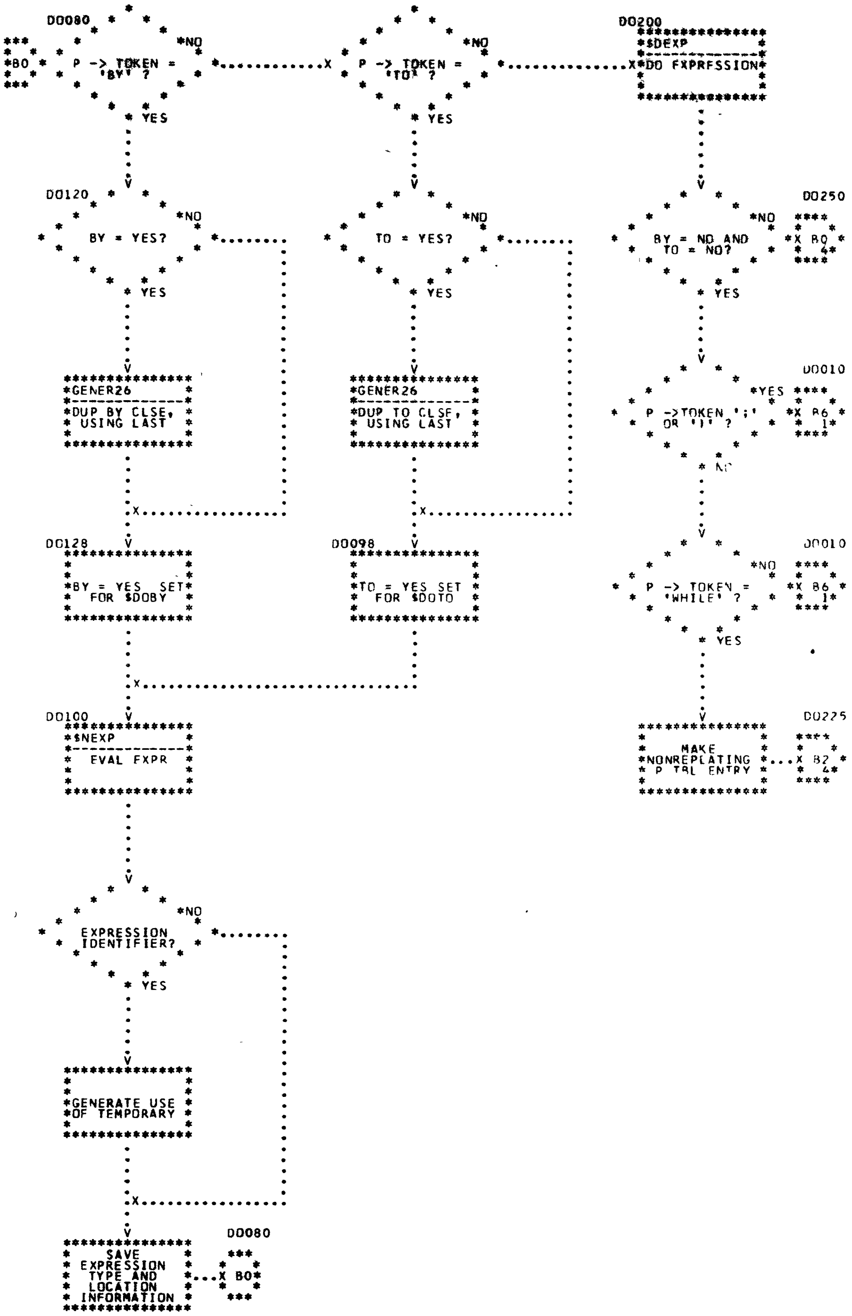


Chart 20. DO Generator (Page 3 of 4)

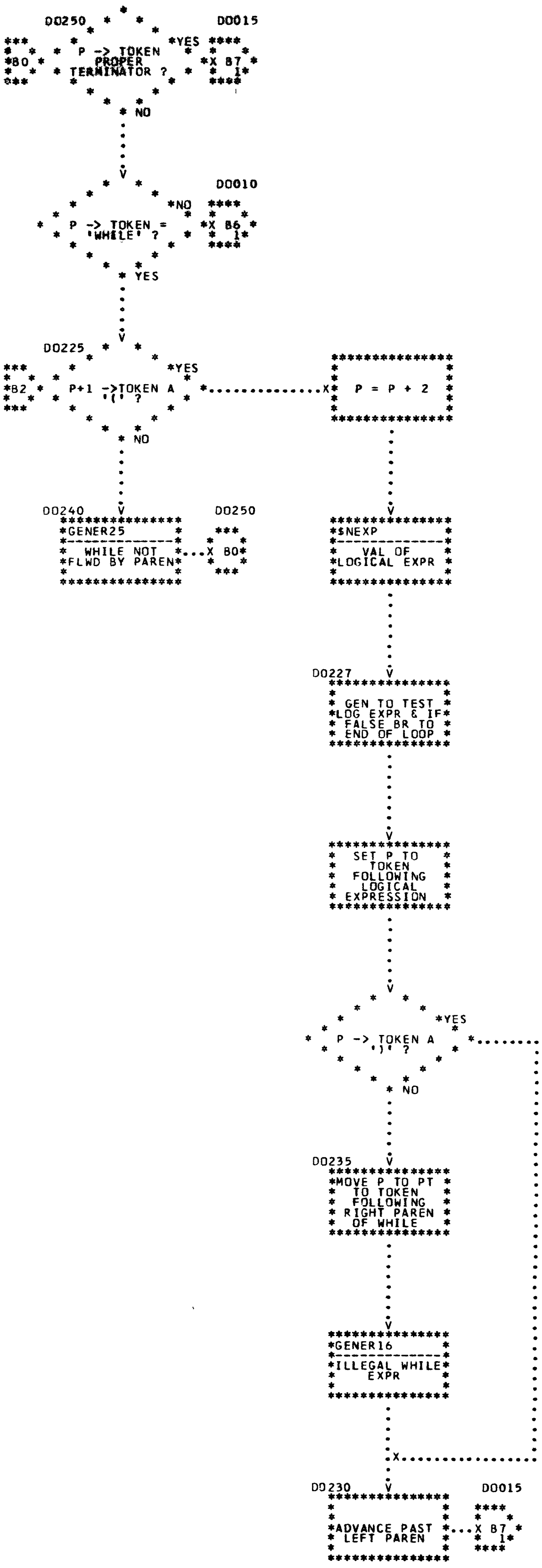


Chart 20. DO Generator (Page 4 of 4)

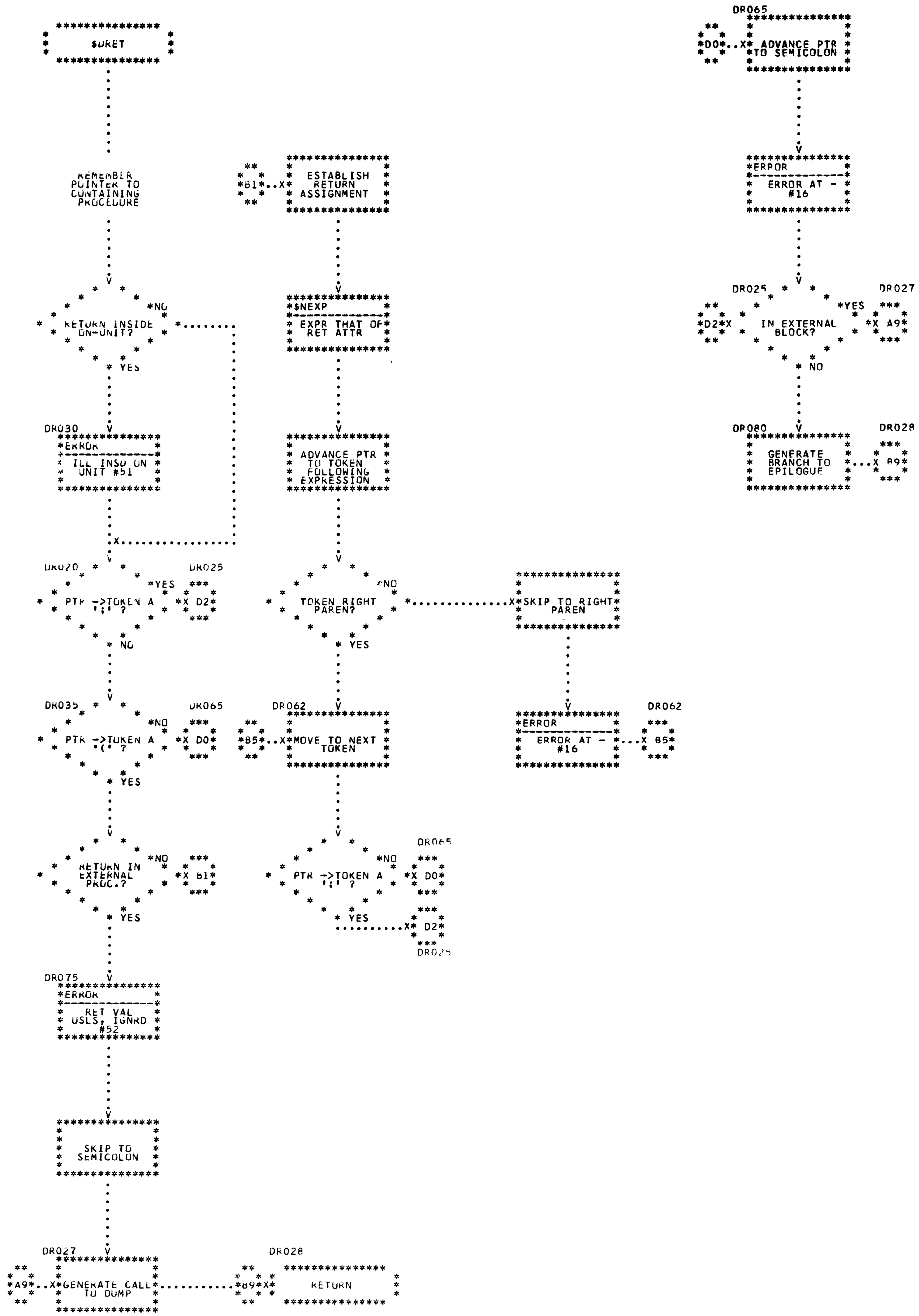
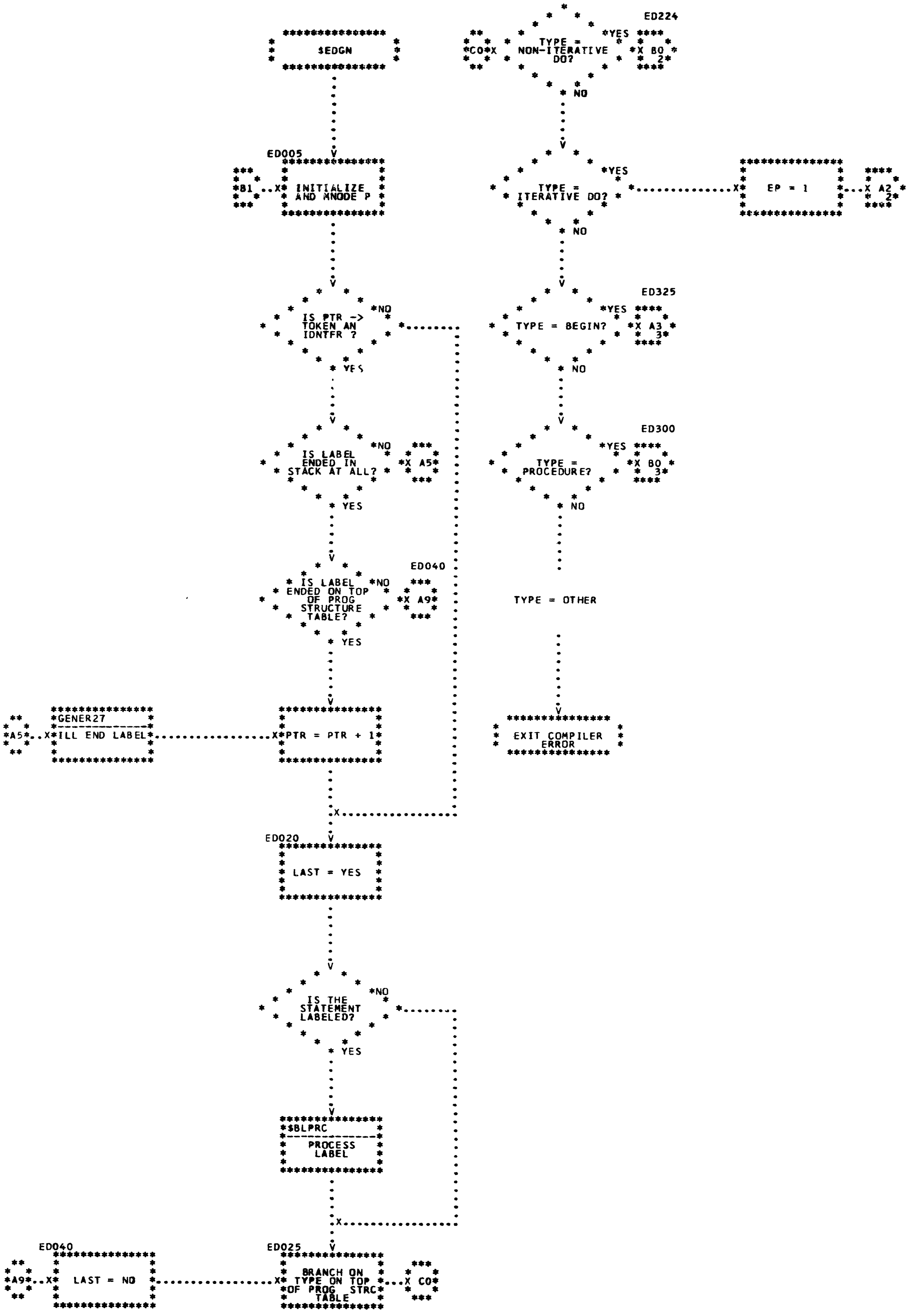


Chart 21. RETURN Generator


$EDGN
ED005
B1
INITIALIZE AND MNODE P
IS PTR -> TOKEN AN IDNTFR ?
NO
YES
IS LABEL ENDED IN STACK AT ALL?
NO
A5
YES
ED040
IS LABEL ENDED ON TOP OF PROG STRUCTURE TABLE?
NO
A9
YES
A5
GENER27
ILL END LABEL
PTR = PTR + 1
ED020
LAST = YES
IS THE STATEMENT LABELED?
NO
YES
$BLPRC
PROCESS LABEL
ED040
A9
LAST = NO
ED025
BRANCH ON TYPE ON TOP OF PROG STRC TABLE
C0
ED224
C0
TYPE = NON-ITERATIVE DO?
YES
B0 2
NO
TYPE = ITERATIVE DO?
YES
EP = 1
A2 2
NO
ED325
TYPE = BEGIN?
YES
A3 3
NO
ED300
TYPE = PROCEDURE?
YES
B0 3
NO
TYPE = OTHER
EXIT COMPILER ERROR

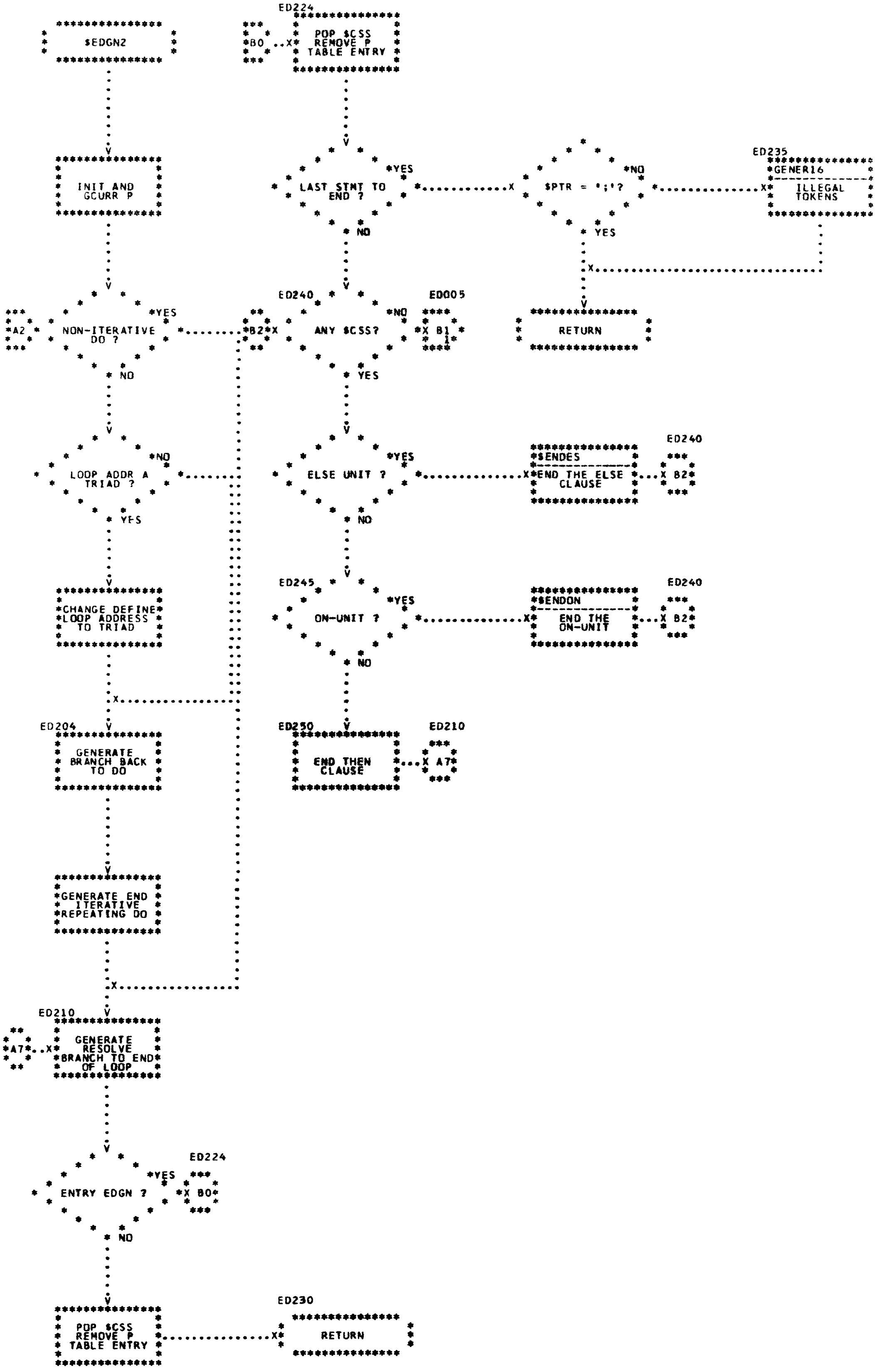


Chart 22. END Generator (Page 2 of 6)

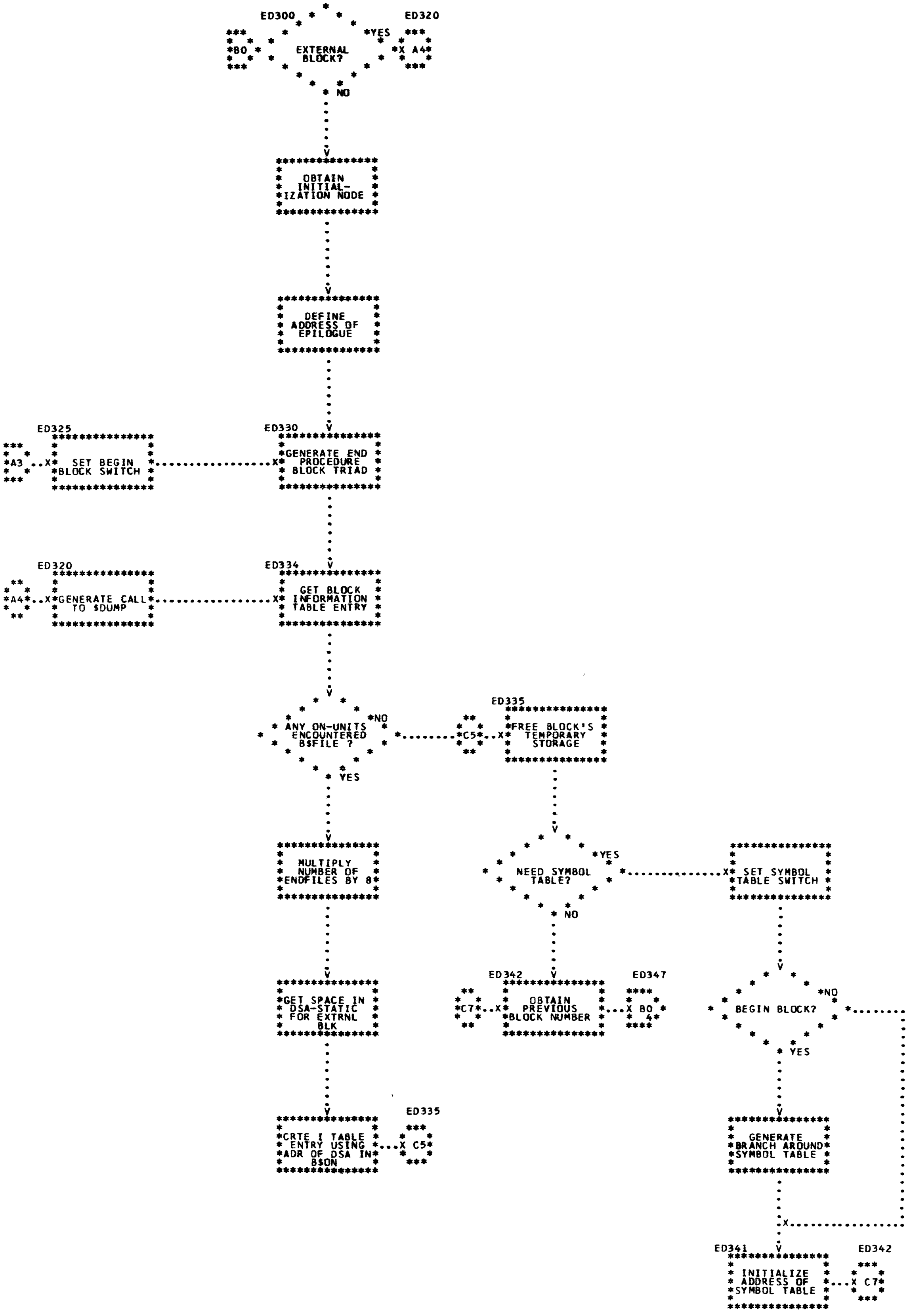


Chart 22. END Generator (Page 3 of 6)

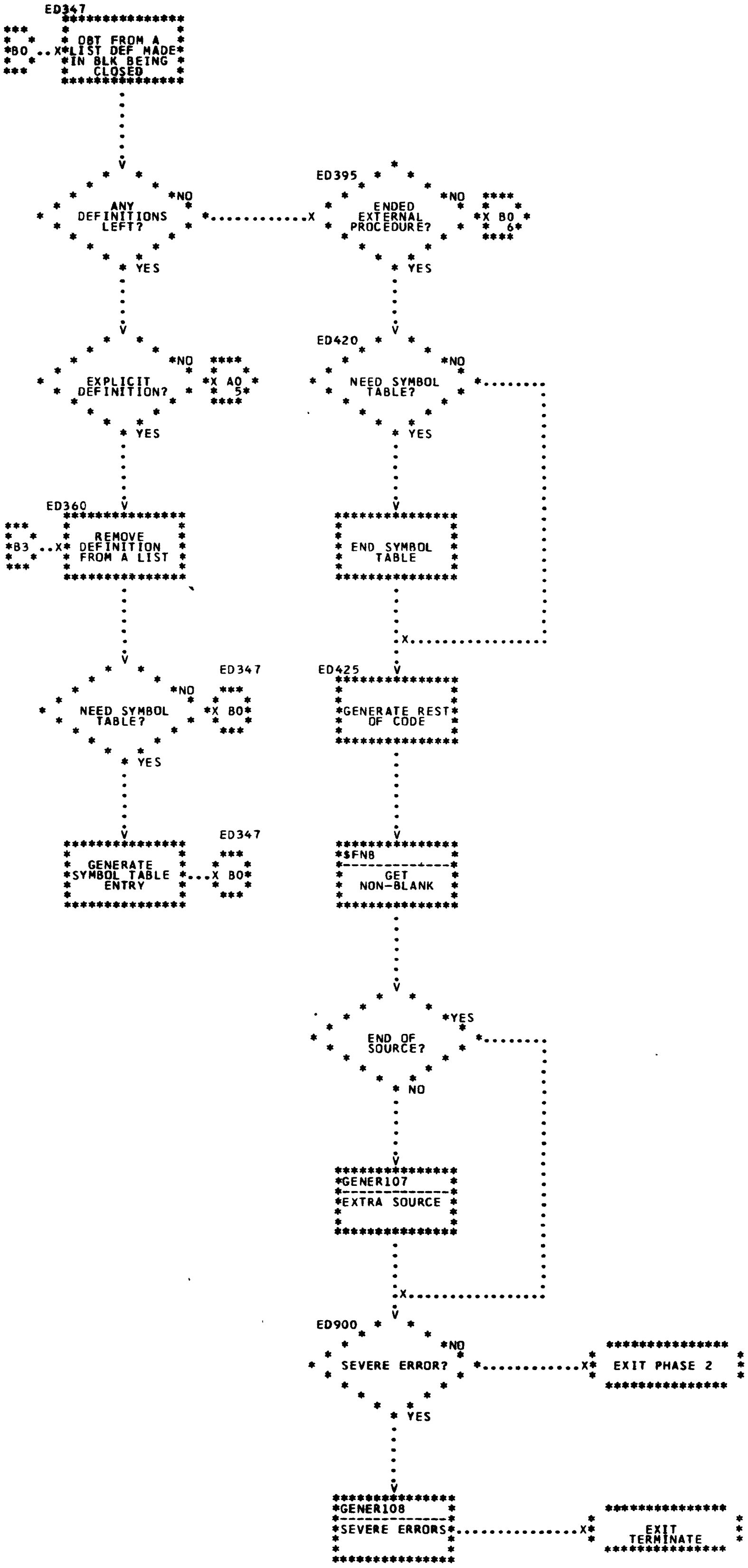


Chart 22. END Generator (Page 4 of 6)

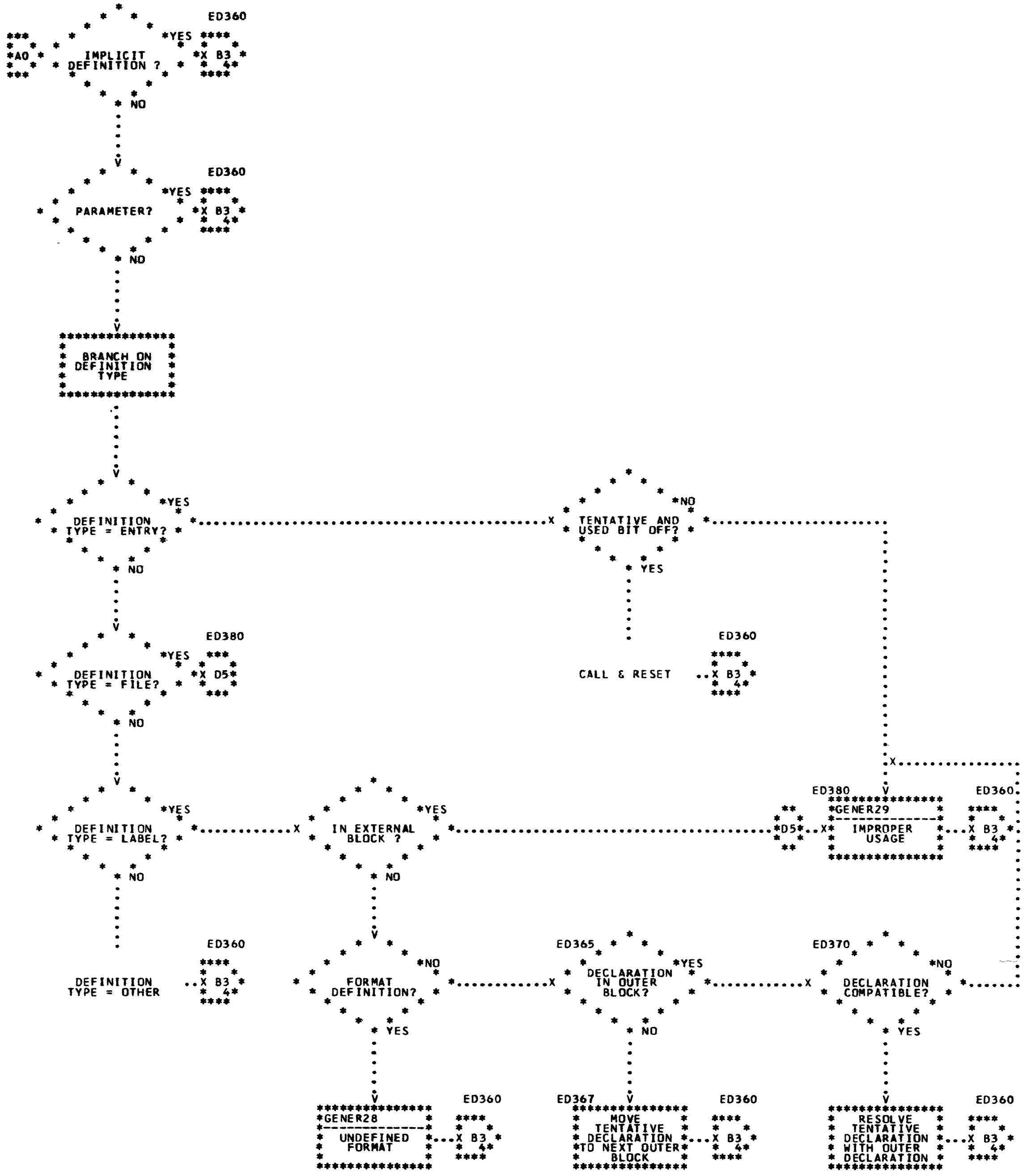


Chart 22. END Generator (Page 5 of 6)

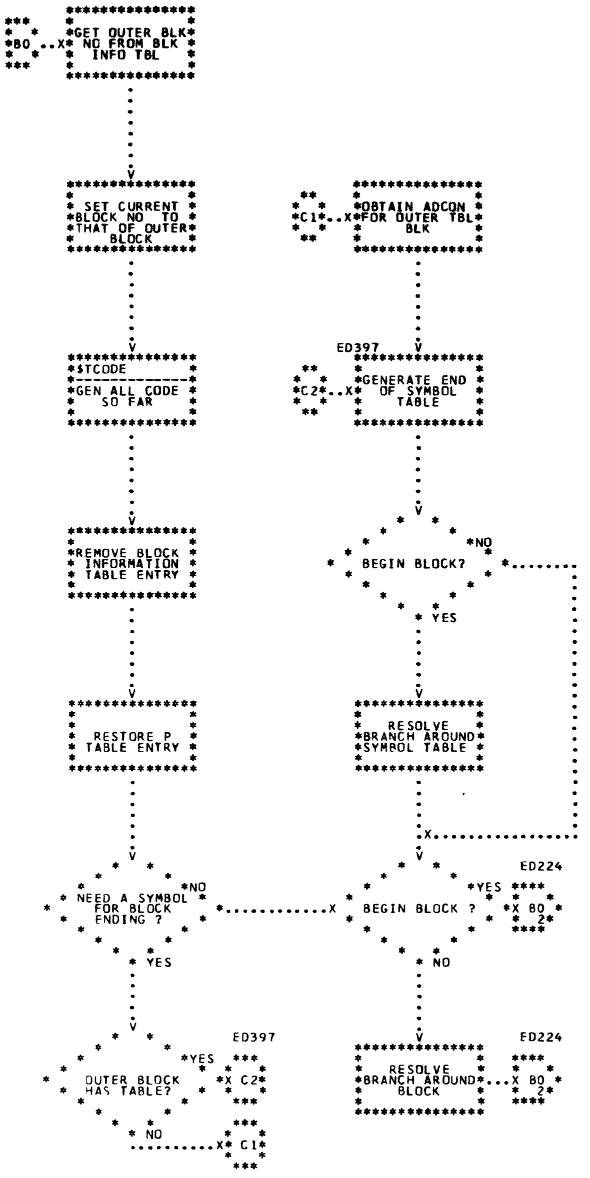


Chart 22. END Generator (Page 6 of 6)

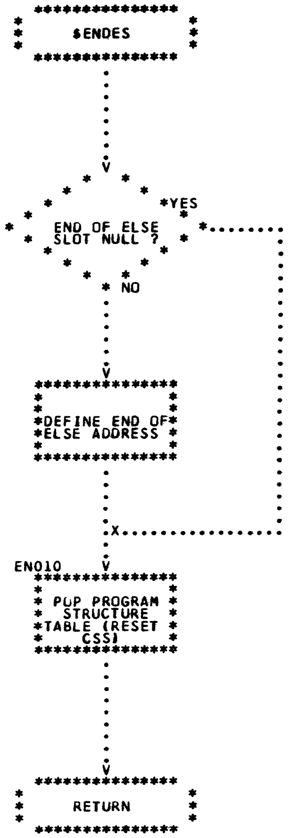


Chart 23. Process End of ELSE

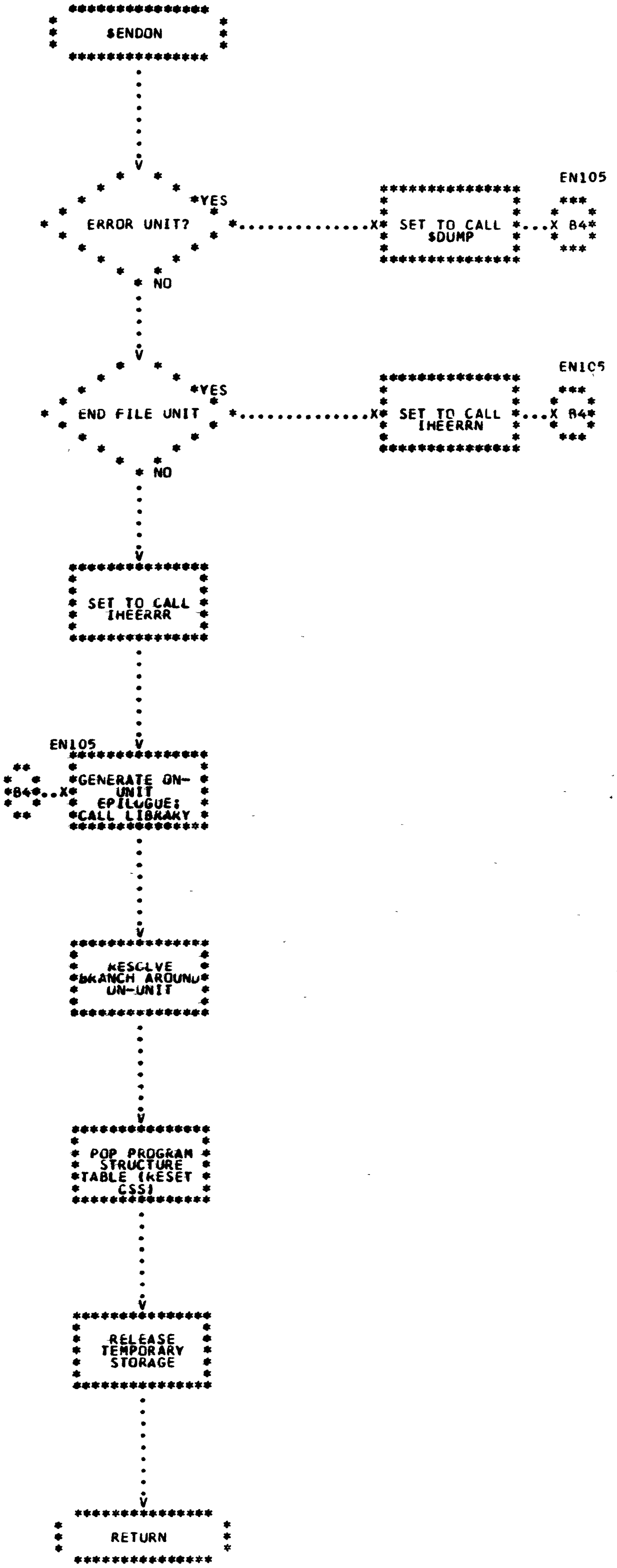
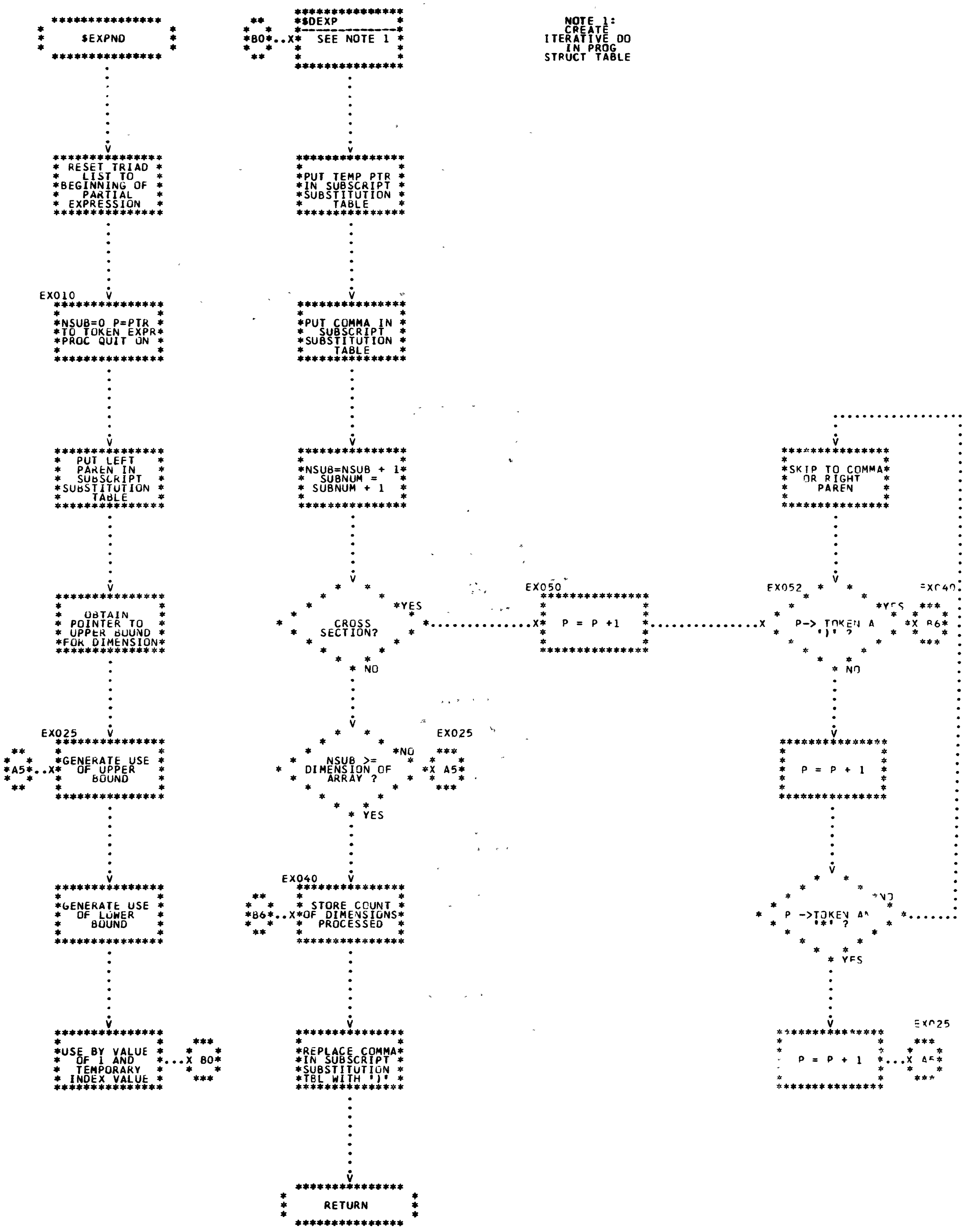


Chart 24. Process End of ON


$EXPND
RESET TRIAD LIST TO BEGINNING OF PARTIAL EXPRESSION
EX010
NSUB=0 P=PTR TO TOKEN EXPR PROC QUIT ON
PUT LEFT PAREN IN SUBSCRIPT SUBSTITUTION TABLE
OBTAIN POINTER TO UPPER BOUND FOR DIMENSION
EX025
A5
GENERATE USE OF UPPER BOUND
GENERATE USE OF LOWER BOUND
USE BY VALUE OF 1 AND TEMPORARY INDEX VALUE
B0
B0
$DEXP
SEE NOTE 1
PUT TEMP PTR IN SUBSCRIPT SUBSTITUTION TABLE
PUT COMMA IN SUBSCRIPT SUBSTITUTION TABLE
NSUB=NSUB + 1 SUBNUM = SUBNUM + 1
CROSS SECTION?
YES
NO
NSUB >= DIMENSION OF ARRAY ?
NO
YES
EX025
A5
EX040
B6
STORE COUNT OF DIMENSIONS PROCESSED
REPLACE COMMA IN SUBSCRIPT SUBSTITUTION TBL WITH ')'
RETURN
NOTE 1: CREATE ITERATIVE DO IN PROG STRUCT TABLE
EX050
P = P +1
EX052
P-> TOKEN A ')' ?
YES
EX040
B6
NO
P = P + 1
P ->TOKEN A '*' ?
NO
YES
SKIP TO COMMA OR RIGHT PAREN
P = P + 1
EX025
A5

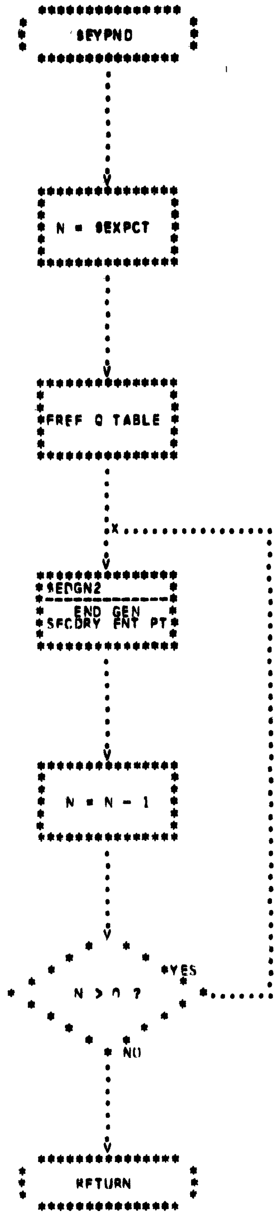


Chart 26. END Expand

PART 4 - DECLARATION PROCESSING

The routines described in this subsection are concerned primarily with creating and finding definitions (declarations) of variables. Four types of declarations are allowed:

- explicit
- implicit
- contextual
- tentative

Understanding of these types of declarations is essential to interpretation of the declaration-processing routines. The types are discussed under "Declaration Processing" in Section 2.

The routines concerned primarily with declaration processing are listed below and described in alphabetic order according to their mnemonics on succeeding pages. Detailed logic diagrams for the routines appear at the end of the subsection.

- Attribute Analysis ($ABAL)
- Attribute Node Creation ($ANCRE)
- Label Processor ($BLPRC)
- DCL Generator ($DCLGN)
- Locate Identifier ($FSYM)
- Locate Variable ($FVAR)

Some attribute entries are processed in routines described in other subsections (for example, PROC Generator). However, such routines are not concerned primarily with declaration processing.

TITLE: ATTRIBUTE ANALYSIS ($ABAL)

Program Definition

Purpose and Usage

The Attribute Analysis routine is used by the DCL Generator to prepare a bit mask and table of option list pointers that represent the explicitly declared attributes of an identifier.

Description

Each attribute that it is possible to explicitly declare for an identifier is assigned a bit position in an attribute bit string. This bit string must be long enough to contain one bit for each attribute and for each optional list. If the bit denoting an attribute is one, the attribute was present; if zero, the attribute was not present. Each attribute list is assigned a pointer that points to the left parenthesis token of the tokens that describe the list. Thus, the attribute table consists of the attribute bit string and the pointers. If an attribute was specified without its list, a corresponding bit is set but the pointer for the attribute list is zero. (For a description of the attribute table, see "Compiler Variables" in Appendix A.)

When an attribute is encountered, an attribute conflict mask for the attribute is obtained. This conflict mask is the row of the attribute conflict matrix (see Figure 3-1) that corresponds to the attribute. By performing a logical AND of the attribute bit string and the conflict mask, it is possible to determine whether the new attribute conflicts with any previous attributes. If not, the bit for the attribute in the bit string is set to one. If the attribute is of the type that may have a list, a check is made to see if one is present. If not, and the list is only optional, processing advances. Otherwise, the pointer for the list is filled and the list is skipped.

This routine finishes processing at the end of a continuous, unfactored set of attributes.

Errors Detected

ERROR AT '___'. (16)
ILLEGAL LIST AFTER ATTRIBUTE '___' FOR IDENTIFIER '___'. (58)
FOR IDENTIFIER '___' ATTRIBUTE '___' CONFLICTS WITH
PREVIOUS ATTRIBUTES. (59)
LIST MISSING AFTER ATTRIBUTE '___'. (60)

Local Variables

ABTB	Attribute Conflict Matrix

Program Interface

Entry Points

One entry point accepting a pointer ($PTR) to a token in the token table.

Exit Conditions

Exit upon detection of a comma, semicolon, or right-parenthesis token. $PTR points to this token.

	File	Input	Output	Print	Static	Automatic	Label	Environment	Returns	Character	Entry	Fixed	Float	Complex	Real	Dimension	Entry Param. List	Precision	Previous Attribute
File					X	X	X		X	X	X	X	X	X	X	X	X	X	
Input			X	X	X	X	X		X	X	X	X	X	X	X	X	X	X	
Output		X			X	X	X		X	X	X	X	X	X	X	X	X	X	
Print		X			X	X	X		X	X	X	X	X	X	X	X	X	X	
Static	X	X	X	X		X		X	X		X						X		
Automatic	X	X	X	X	X			X	X		X						X		
Label	X	X	X	X				X	X	X	X	X	X	X	X		X	X	
Environment					X	X	X	X	X	X	X	X	X	X	X	X	X	X	
Returns	X	X	X	X	X	X	X	X	X	X		X	X	X	X	X		X	
Character	X	X	X	X			X	X	X	X	X	X	X	X	X		X	X	
Entry	X	X	X	X	X	X	X	X		X		X	X	X	X	X	X	X	
Fixed	X	X	X	X			X	X	X	X	X		X				X		
Float	X	X	X	X			X	X	X	X	X	X					X		
Complex	X	X	X	X			X	X	X	X	X				X		X		
Real	X	X	X	X			X	X	X	X	X			X			X		
Dimension	X	X	X	X				X	X		X					X	X		X
Entry Param. List	X	X	X	X	X	X	X	X		X		X	X	X	X	X	X	X	
Precision	X	X	X	X			X	X	X	X	X						X	X	

Figure 3-1. Attribute Conflict Matrix

Routines Called

$XERR	Error Message Editor

Global Variables

$DCNME	Name of Current Identifier
A List	Dictionary Attribute List
$ABTBL	Attribute Table
T Table	Token Table

Comments

Ease of access of conflict masks depends on convenient ordering of keyword codes. Order desired is continuous codes in the following order:

FILE
INPUT
OUTPUT
PRINT
STATIC
AUTOMATIC
LABEL
ENVIRONMENT
RETURNS
CHARACTER
ENTRY
FIXED
FLOAT
COMPLEX
REAL

Logic Diagram

Chart 27 shows the detailed logic diagram for the Attribute Analysis routine.

TITLE: ATTRIBUTE NODE CREATION ($ANCRE)

Program Definition

Purpose and Usage

The Attribute Node Creation routine translates the information in the attribute table ($ABTBL) into a dictionary attribute node. However, it does not enter this node into the dictionary attribute list.

Description

Because file and entry name declarations require special handling, the first distinction made after entering this routine is whether or not a variable is being declared. This distinction is made using a logical AND of the attribute bit string and a classification mask. This classification mask and other classification masks used by the Attribute Node Creation routine are shown in Figure 3-2. (For a description of the attribute table, see "Compiler Variables" in Appendix A.)

Mask Classifications	File	Input	Output	Print	Static	Automatic	Label	Environment	Returns	Character	Entry	Fixed	Float	Complex	Real	Dimension	Entry Parameter List	Precision
Mask 1: Not Arithmetic Class	X	X	X	X			X	X	X	X	X						X	
Mask 2: Entry Class									X		X						X	
Mask 3: File Class	X	X	X	X				X										
Mask 4: Not Variable Class	X	X	X	X				X	X		X						X	
Mask 5: Mode and Scale												X	X	X	X			
Mask 6: Mode														X	X			
Mask 7: Scale												X	X					

Figure 3-2. Classification Masks

If the declared entry is not a variable, it is determined whether a file or an entry name is being declared. In either case, space for a dictionary attribute node is obtained, special attributes are encoded, and transfer is made to the section of the routine that encodes the common attributes shared by all declarations.

Since entry name declarations may have entry name declarations within them, a push-down list is maintained which indicates the processing state when a nested entry name declaration was encountered. This entry name declaration list allows this routine to be pseudo-recursive. The contents of the list entries are:

1. Pointer to the entry attribute node being processed (P$SENT).
2. Number of parameters previously processed (P$ZNP).
3. Pointer to last parameter attribute node processed (P$ZPRM).
4. State of processing (RETURNS attributes or parameter list (P$ZSP)).
5. Pointer to current position in entokening of parameter list (P$ZETK).

For convenience in storage, this push-down list is kept in the program structure table. The top node of the push-down list is kept in the local variable ANPDL. (For a description of the entry name declaration list, see Appendix B.)

If a file declaration for SYSIN or SYSPRINT is made, a search is made to see if there is a corresponding external definition (in block zero). If not, one is created that is the same as the definition just made. If so, the new definition is compared with the corresponding definition for consistency.

If a variable is declared, space is obtained for a dictionary attribute node. The size of the space depends on whether the variable is dimensioned. If it is, the dimension list is analyzed at this time. Checks are made to ensure that all the specified bounds are legal. If necessary, code is generated to evaluate the bounds at execution time. A skeleton dope vector is established for each dimension variable. All bounds of constant value are initialized into this skeleton dope vector.

If the variable is arithmetic, any precision, mode, or scale defaults are added. The arithmetic attributes are then encoded and transfer is made to the common attributes section (see below).

If the variable is character, its string length is evaluated in a manner similar to dimensions. Any string attributes are established before taking care of the common attributes.

The common attributes section establishes attributes such as scope, type of declaration, and block in which information was declared. All actions to complete the attribute node are performed.

Errors Detected

ERROR AT '___'.(16)
ILLEGAL USE OF '___' ATTRIBUTE. (20)
ILLEGAL RETURNS ATTRIBUTES--DEFAULT RETURNS ATTRIBUTES USED. (47)
USE OF '___' HERE CONFLICTS WITH PREVIOUS USAGE. (57)
PRECISION ATTRIBUTE IS ILLEGAL--DEFAULT USED. (62)
ILLEGAL PARAMETER ATTRIBUTES FOR '___'. (63)
ILLEGAL SCALE FACTOR FOR '___'--IGNORED. (64)
NOT ALL DIMENSION EXPRESSIONS ARE CONSTANTS. (65)
FOR STRING '___' LENGTH NOT A CONSTANT. (66)
'___' HAS ILLEGAL LENGTH--225 USED. (67)
'___' HAS ILLEGAL '*' DIMENSION OR STRING LENGTH. (68)
ATTRIBUTE FOR FILE '___' CONFLICTS WITH PREVIOUS DECLARATION OR USE. (69)
IMPROPER ARRAY BOUND. (104)

Local Variables

ANRPT	Internal Routine Return Point
ANTPTR	Entry Value of $PTR
Register G3	Address of Attribute Node
ANPDL and ANPLEV	Entry Name Declaration List Pointer
ANDVL	Dope Vector Length

Program Interface

Entry Points

One entry point. Only input required is an attribute table and $DCNME.

Exit Conditions

Normal exit only. Output consists of a pointer to a dictionary attribute node in register G0.

Routines Called

$ABAL	Attribute Analysis
$CERR	Compiler Error
$NEXP	Expression Processor Controller
$STRD	Generate Triad
$GTRIAD	Get Next Triad Entry
$NCONS	Constant Processor
$AREXP	Array Expression Error
$XERR	Error Message Editor
$WCTCT	Segment Management
$WEXP	Segment Management

Global Variables

$DCNME	Name of Current Identifier
$APARM	Switch Determining Whether Identifier is Parameter
A List	Dictionary Attribute List
I Table	Initialization Table
J List	Dope Vector List

Logic Diagram

Chart 28 shows the detailed logic diagram for the Attribute Node Creation routine.

TITLE: LABEL PROCESSOR ($BLPRC)

Program Definition

Purpose and Usage

The Label Processor defines the address of a statement label or entry name. It also resolves any tentative declaration previously made.

Description

Locate Identifier ($FSYM) is called to find any declaration in the current block of the identifier pointed to by the label pointer. If a previous declaration has been made, a check is made to see if it is tentative. If not, a multiple declaration exists (which is illegal). If the declaration was tentative, a check is made to ensure that its usage was compatible with the type of declaration being made. All tentative declarations within the proper scope are resolved.

An attribute node is made for the declaration. The definition is added to the dictionary attribute list and the address of the label is defined.

Errors Detected

MULTIPLE DECLARATION FOR '___'--THIS DECLARATION USED. (56)
USE OF '___' HERE CONFLICTS WITH PREVIOUS USAGE. (57)

Local Variables

None

Program Interface

Entry Points

$BLPRC. No formal parameters. This routine determines whether it is to define statement label, FORMAT statement label, or entry name by the value in $CLBLS.

Exit Conditions

Normal exit only. Register G0 has pointer to attribute entry.

Routines Called

$FSYM	Locate Identifier
$GTRIAD	Get Next Triad Entry
$XERR	Error Message Editor
$WEXP	Segment Management

Global Variables

$CLBLS	Label Switch
A List	Dictionary Attribute List
N List	Dictionary Name List
T Table	Token Table
I Table	Initialization Table
$CLPTR	Label Pointer

Logic Diagram

Chart 29 shows the detailed logic diagram for the Label Processor routine.

TITLE: DCL GENERATOR ($DCLGN)

Program Definition

Purpose and Usage

The DCL Generator directs the analysis and encoding of attributes for identifiers in a DECLARE statement, and adds the attribute entries to the dictionary attribute list (A list) upon completion of encoding. During this process, code is generated, if necessary, to allocate space for data and to construct dope vectors.

Description

The attributes explicitly declared for an identifier are maintained in a bit string, one bit assigned to each attribute. If an attribute has been specified, its bit is one; otherwise, the bit is zero. Before beginning any attribute analysis, the bit string where the attributes are accumulated is initialized to zeros. The Attribute Analysis routine ($ABAL) is called to analyze the attributes immediately following the identifier. On return, any factored attributes are analyzed, one factoring level at a time.

The DCL Generator keeps track of the factoring in effect by maintaining a push-down list of pointers to the right parenthesis token starting the factoring level. When the right parenthesis is reached, its pointer is removed from the push-down list and the attributes following it are skipped.

When all the attributes have been analyzed, a check is made to determine whether the identifier was previously declared or used in the current block. If so, and if it was not used only as a parameter specification, a flag is set to indicate possible multiple declaration or use before declaration.

At this point, the Attribute Node Creation routine ($ANCRE) is called to translate the bit string and a table of pointers to option lists into a dictionary attribute list entry and the necessary code. The entry just created is added to the dictionary attribute list unless the error flag is set; if set, the new definition is checked for compatibility with the previous declaration.

After an identifier has been completely encoded, translation of the next identifier in the token table begins.

Errors Detected

ERROR AT '___'. (16)
STRUCTURES NOT SUPPORTED--'___' IGNORED. (18)
PREVIOUS DECLARATION OR USE OF '___'. (21)

Local Variables

Push-down list kept in left parenthesis tokens in token table:

DCPDL	Push-Down List Head
DCPTR	Temporarily Saved Token Table Pointer
DCERR	Possible Error Flag
DCDEF	Check for Compatible Definitions

Program Interface

Entry Points

$DCLGN. No formal parameters.

Exit Conditions

Exit after translating all of DECLARE statement and reaching a semicolon token.

Routines Called

$ABAL	Attribute Analysis
$ANCRE	Attribute Node Creation
$FSYM	Locate Identifier
$XERR	Error Message Editor

Global Variables

$PTR	Token Table Pointer
T Table	Token Table
A List	Dictionary Attribute List
N List	Dictionary Name List
$CBKNO	Current Block Number
$DCNME	Name of Current Identifier
$APARM	Switch Determining Whether Identifier is Parameter

Logic Diagram

Chart 30 shows the detailed logic diagram for the DCL Generator routine.

TITLE: LOCATE IDENTIFIER ($FSYM)

Program Definition

Purpose and Usage

Locate Identifier finds the definition of an identifier that occurs within a block with a block number greater than that one specified.

Description

The dictionary attribute list for a given name entry is searched until a definition is found which had last usage or declaration with a block number greater than or equal to the block number specified for the search. This determines whether the identifier has been declared or used in the block or in a block which is internal to the indicated block. If one or more definitions is found, the one with the greatest block number is returned along with a success indication. If none is found, a failure is reported.

Errors Detected

None

Local Variables

None

Program Interface

Entry Points

$FSYM. No formal parameters. Needs pointer to a token table identifier entry in G7 and block number specifying where the declaration must be in G6.

Exit Conditions

Normal exit only. Returns a success indicator and, if successful, a pointer to the dictionary attribute list entry in G0.

Routines Called

$CERR	Compiler Error

Global Variables

T Table	Token Table
A List	Dictionary Attribute List
N List	Dictionary Name List

Logic Diagram

Chart 31 shows the detailed logic diagram for the Locate Identifier routine.

TITLE: LOCATE VARIABLE ($FVAR)

Program Definition

Purpose and Usage

Locate Variable is used to locate the current definition of a variable and, if none, to create a definition.

Description

If the input consists of a pointer to a dictionary name list entry, the dictionary attribute list for that name is searched for the current definition. If an arithmetic identifier is being sought, the search encompasses all blocks. If a label identifier is being sought, only the current block is searched. If the definition found is of type label, the mode of search is set to label for use by further searches.

If a definition is not found, a definition is created depending on the mode of search and the contents of the token table. The results for the various modes of search are as follows:

label - a tentative statement-label declaration is made in the current block.

file - a file declaration is made in the outermost block (as well as an external declaration).

arithmetic - if the token following the identifier in the token table is a left parenthesis, an entry name is declared; if it is not, an implicit variable declaration is made.

entry name - a check is made to determine whether a block adcon area (BAA) must be allocated.

Any created declaration is added to the dictionary name list and the pointer to the definition returned.

If a previously used declaration is returned, no check is made to insure that it meets the requested mode.

Errors Detected

None

Local Variables

None

Program Interface

Entry Points

$FVAR. No formal parameters. Requires a pointer to a token table entry in G3.

Exit Conditions

Normal exit only. Returns a pointer to a dictionary attribute list entry in G5.

Routines Called

$WEXP	Segment Management

Global Variables

T Table	Token Table
A List	Dictionary Attribute List
N List	Dictionary Name List
$NIDSI	Identifier Search Indicator

Logic Diagram

Chart 32 shows the detailed logic diagram for the Locate Variable routine.

PART 4 LOGIC DIAGRAMS

The detailed logic diagrams for the CALL/360-OS PL/I declaration-processing routines follow.

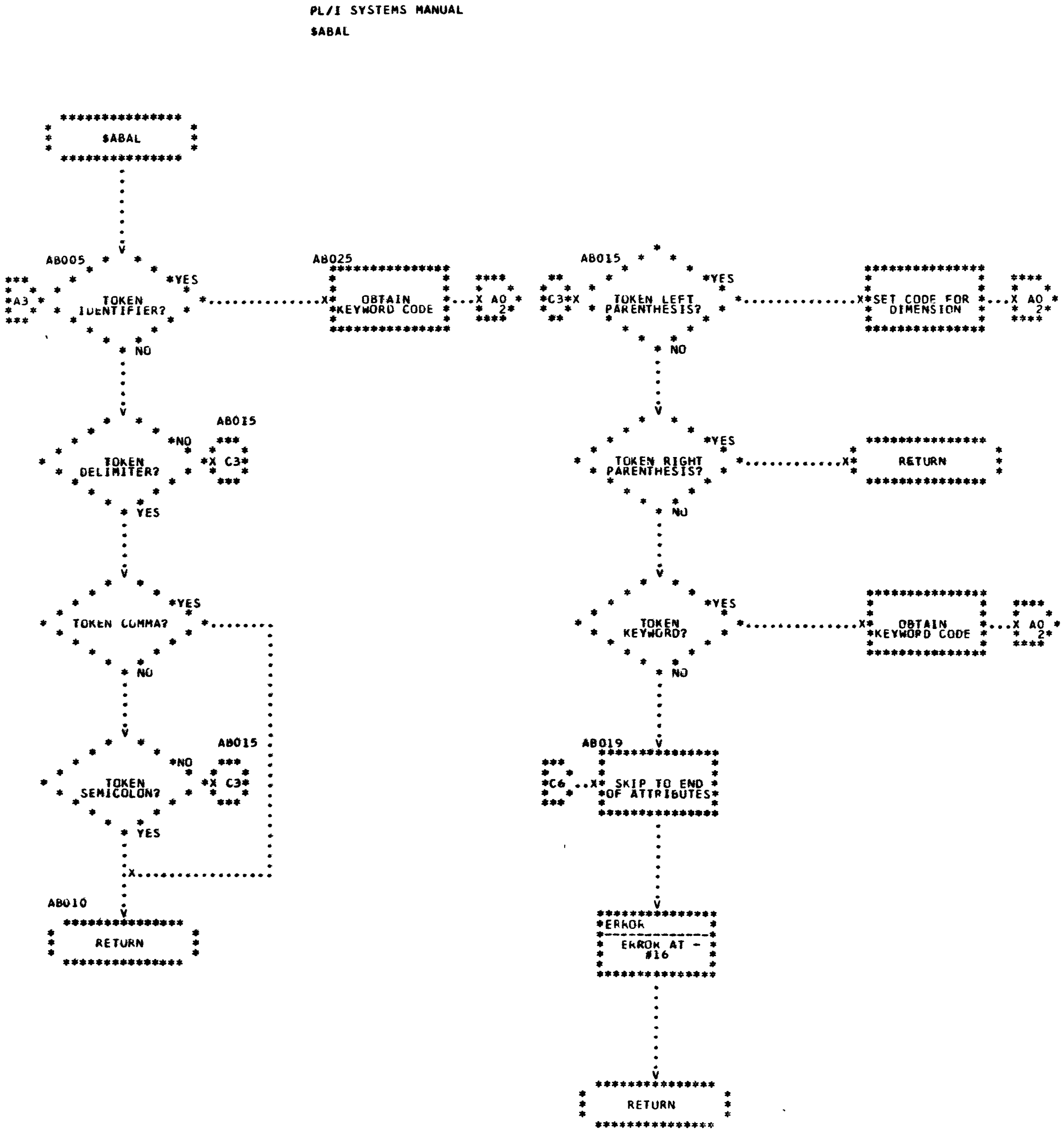


Chart 27. Attribute Analysis (Page 1 of 2)

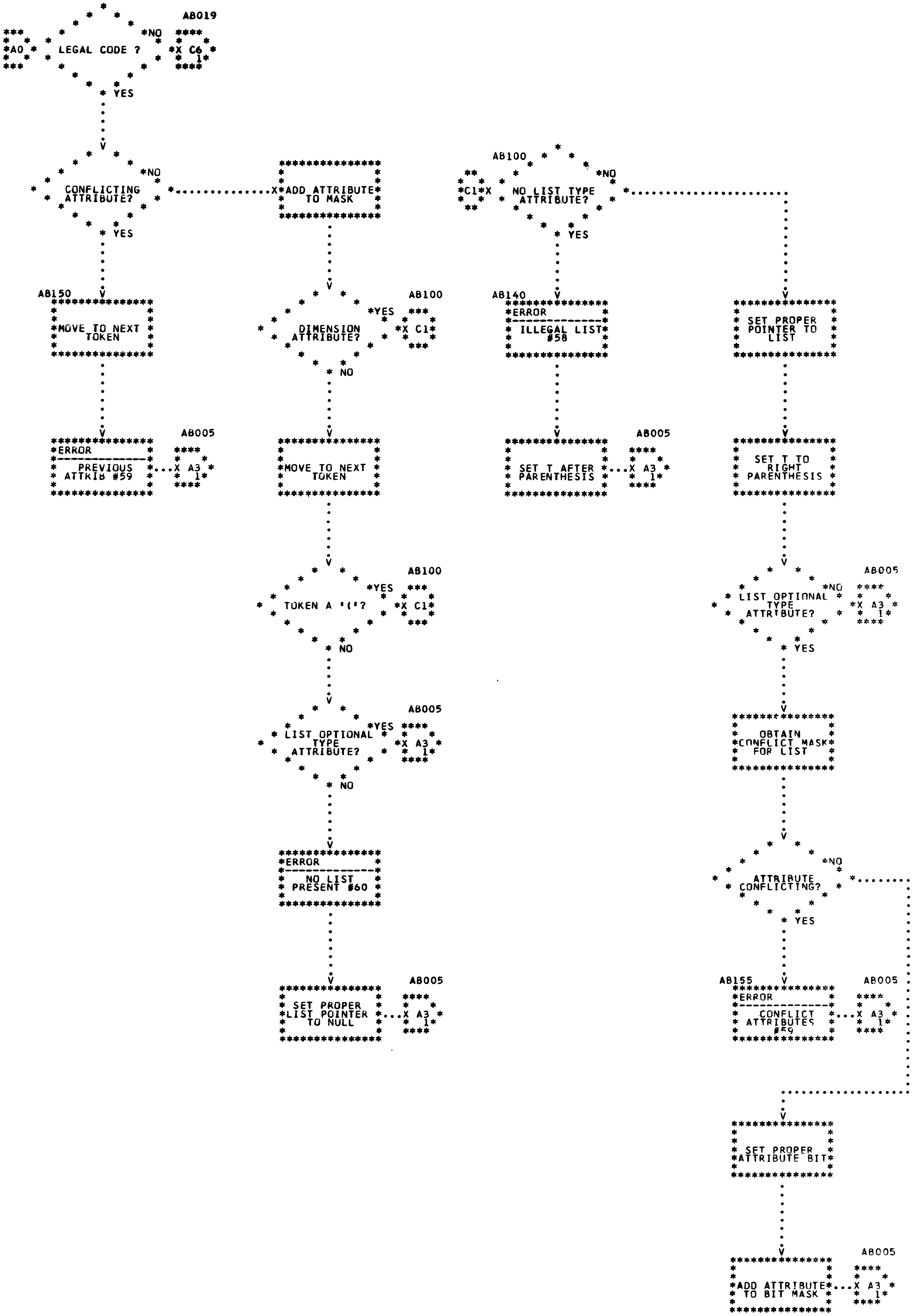


Chart 27. Attribute Analysis (Page 2 of 2)

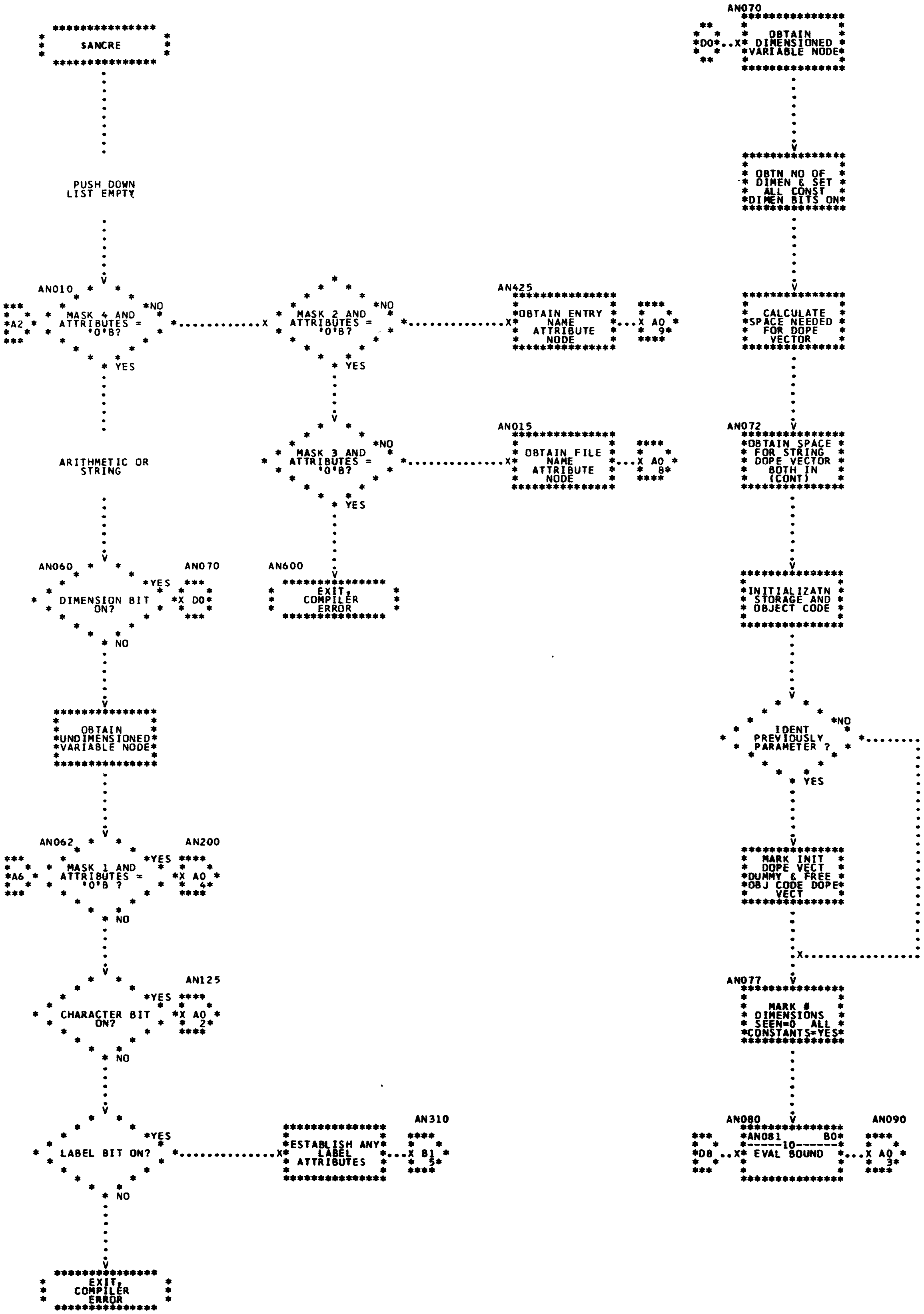


Chart 28. Attribute Node Creation (Page 1 of 10)

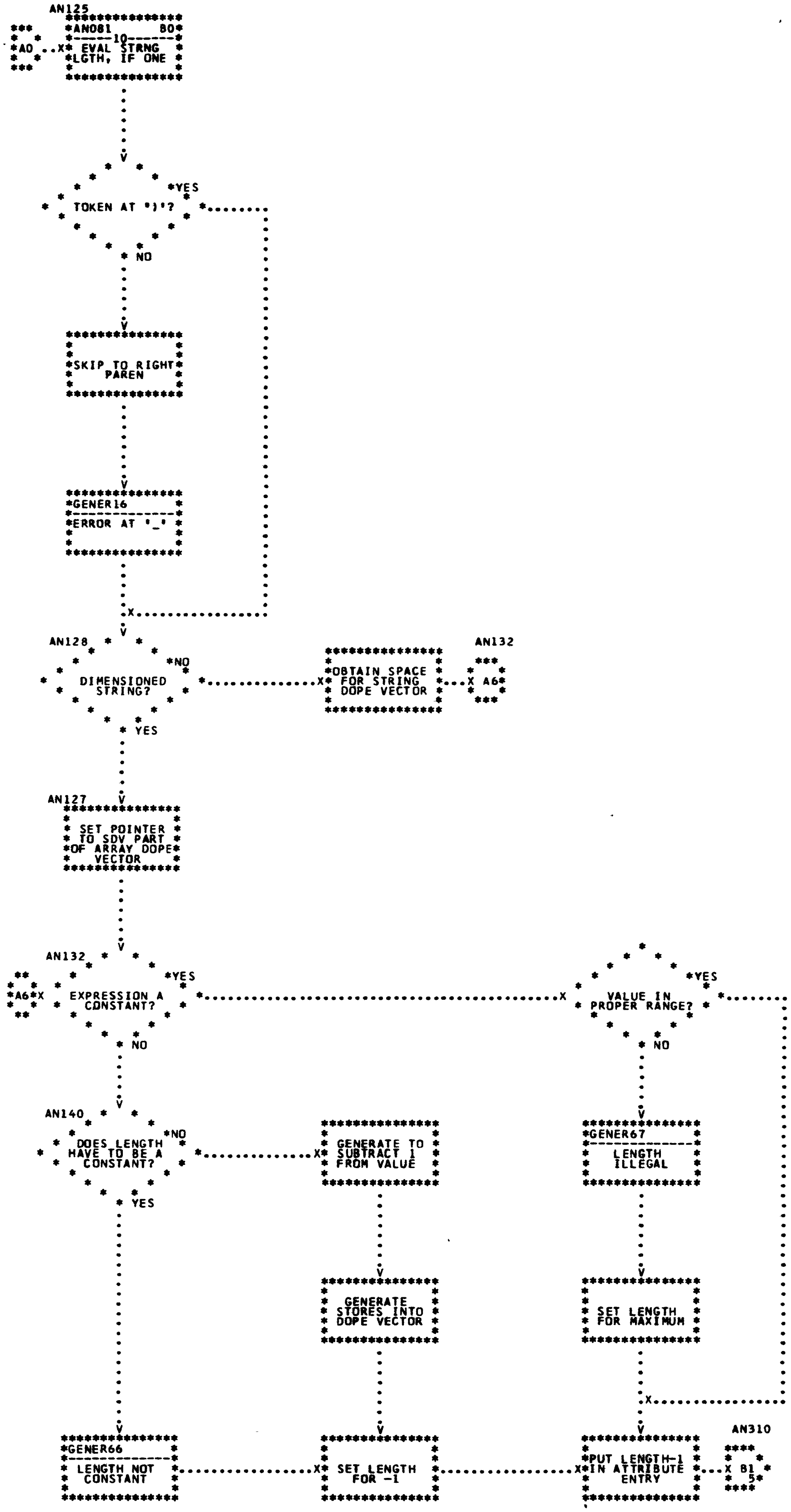


Chart 28. Attribute Node Creation (Page 2 of 10)

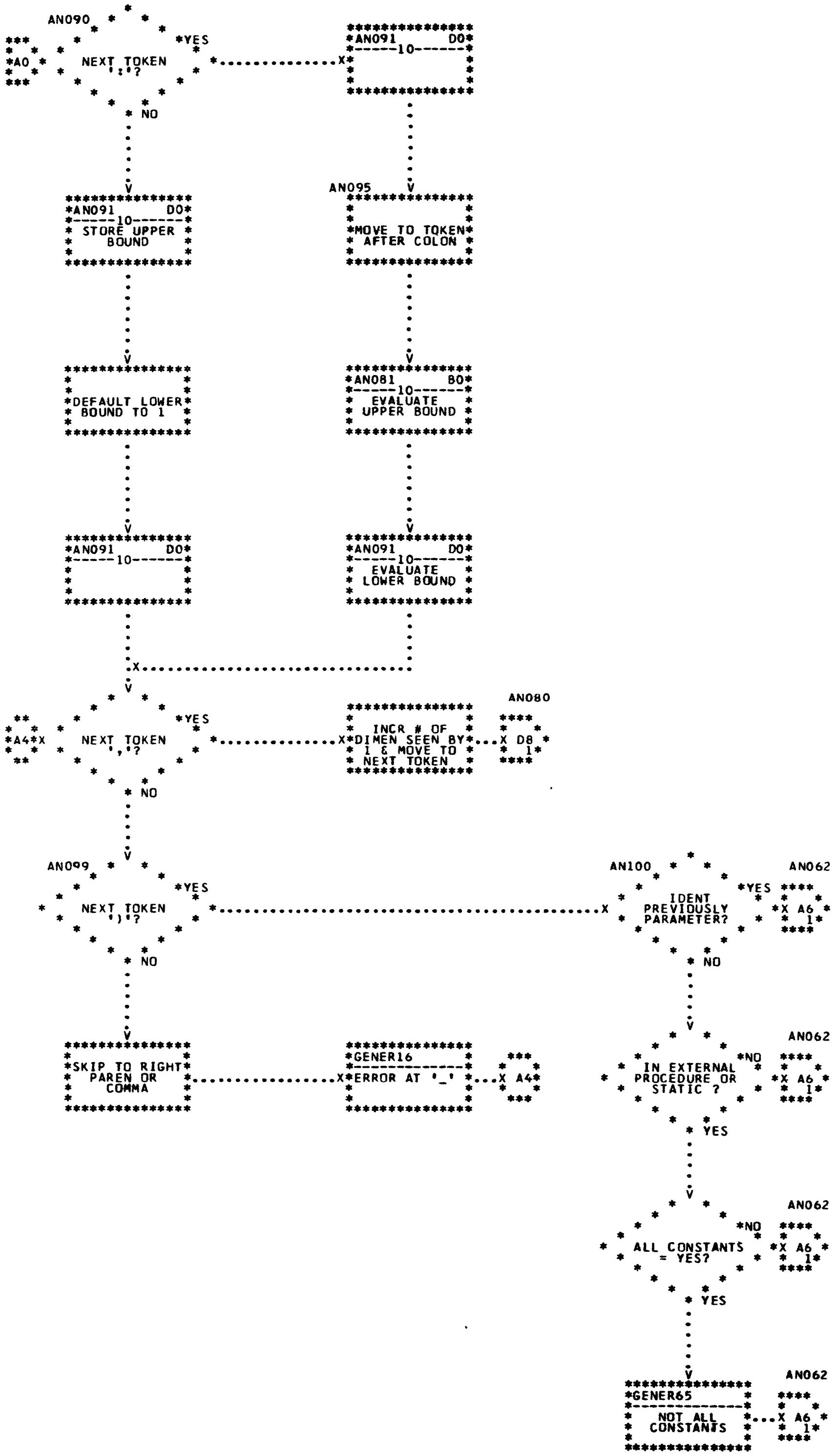


Chart 28. Attribute Node Creation (Page 3 of 10)

$ANCRE

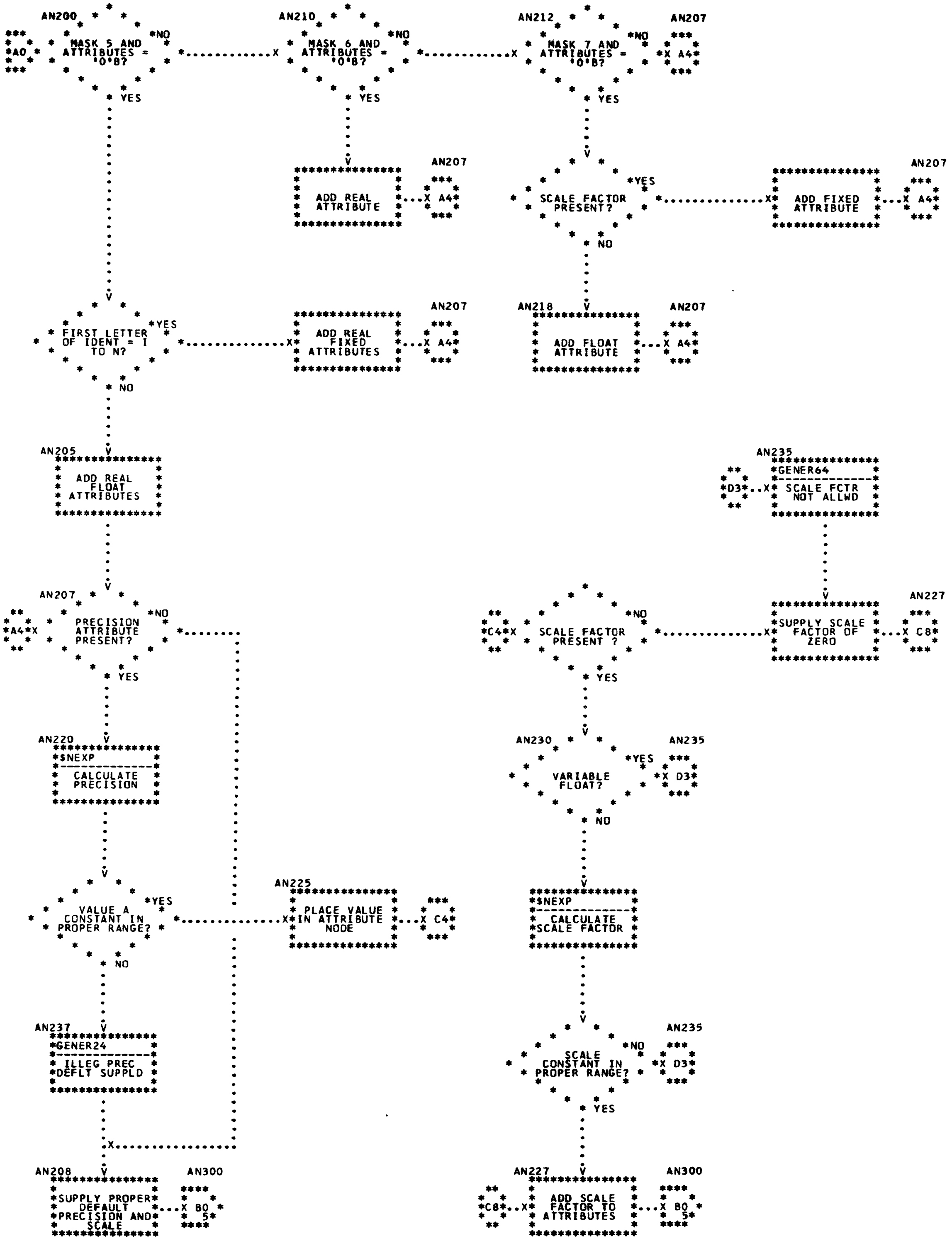
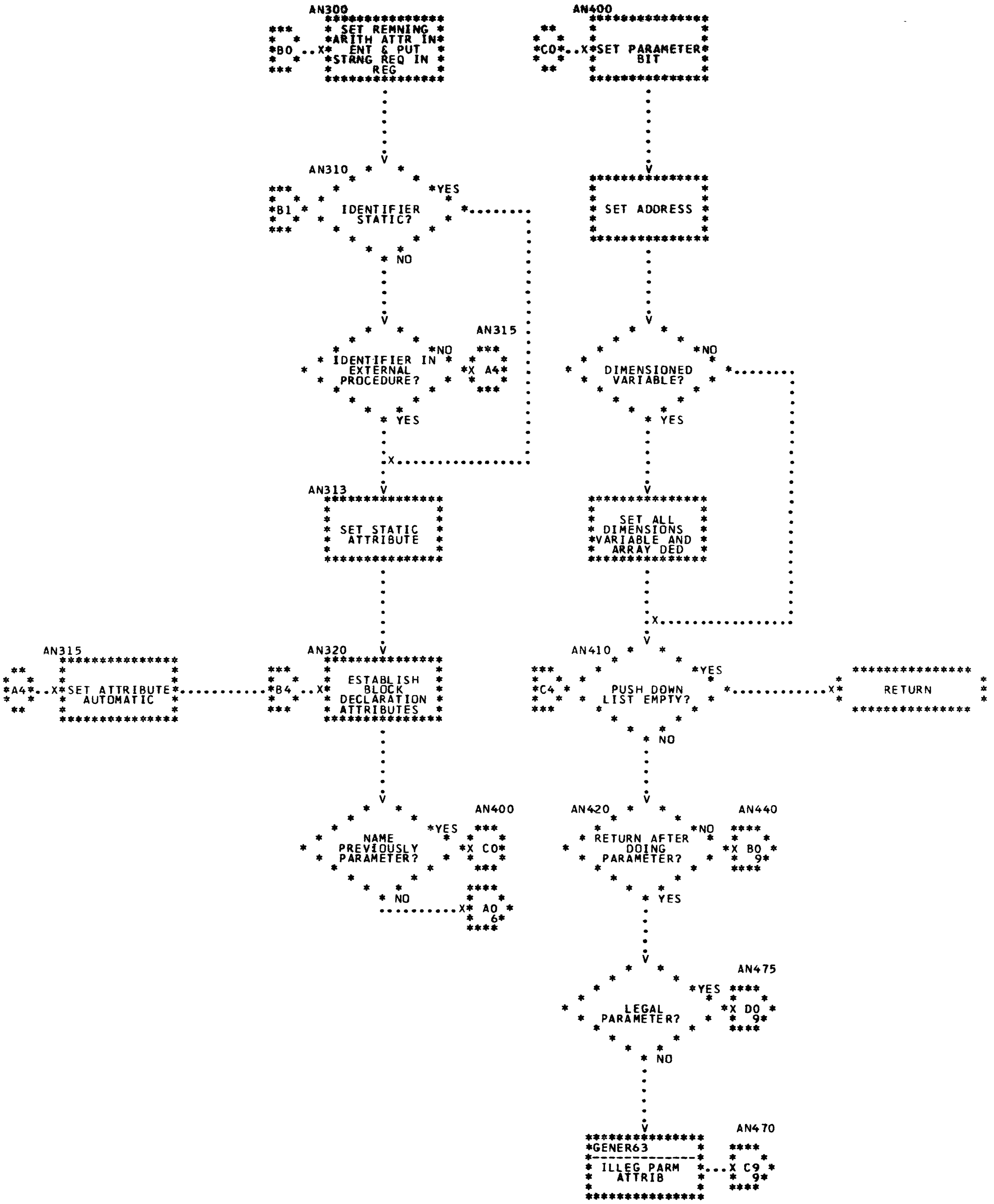


Chart 28. Attribute Node Creation (Page 4 of 10)


AN300
B0
SET REMNING ARITH ATTR IN ENT & PUT STRNG REQ IN REG
AN400
C0
SET PARAMETER BIT
AN310
B1
IDENTIFIER STATIC?
YES
NO
SET ADDRESS
AN315
IDENTIFIER IN EXTERNAL PROCEDURE?
NO
A4
YES
DIMENSIONED VARIABLE?
NO
YES
AN313
SET STATIC ATTRIBUTE
SET ALL DIMENSIONS VARIABLE AND ARRAY DED
AN315
A4
SET ATTRIBUTE AUTOMATIC
AN320
B4
ESTABLISH BLOCK DECLARATION ATTRIBUTES
AN410
C4
PUSH DOWN LIST EMPTY?
YES
RETURN
NO
AN400
NAME PREVIOUSLY PARAMETER?
YES
C0
NO
A0 6
AN420
RETURN AFTER DOING PARAMETER?
NO
AN440
B0 9
YES
AN475
LEGAL PARAMETER?
YES
D0 9
NO
AN470
GENER63
ILLEG PARM ATTRIB
C9 9

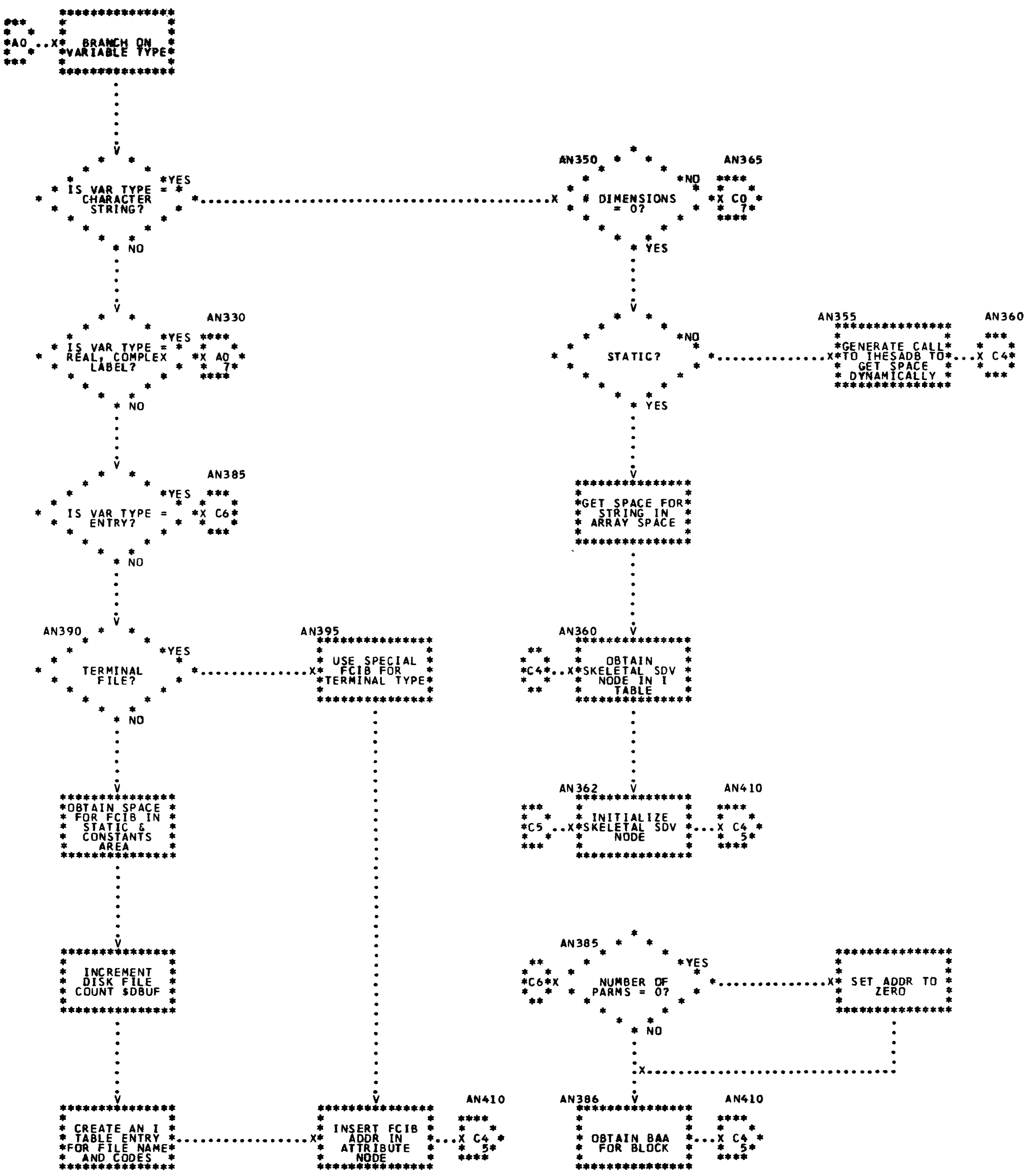


Chart 28. Attribute Node Creation (Page 6 of 10)

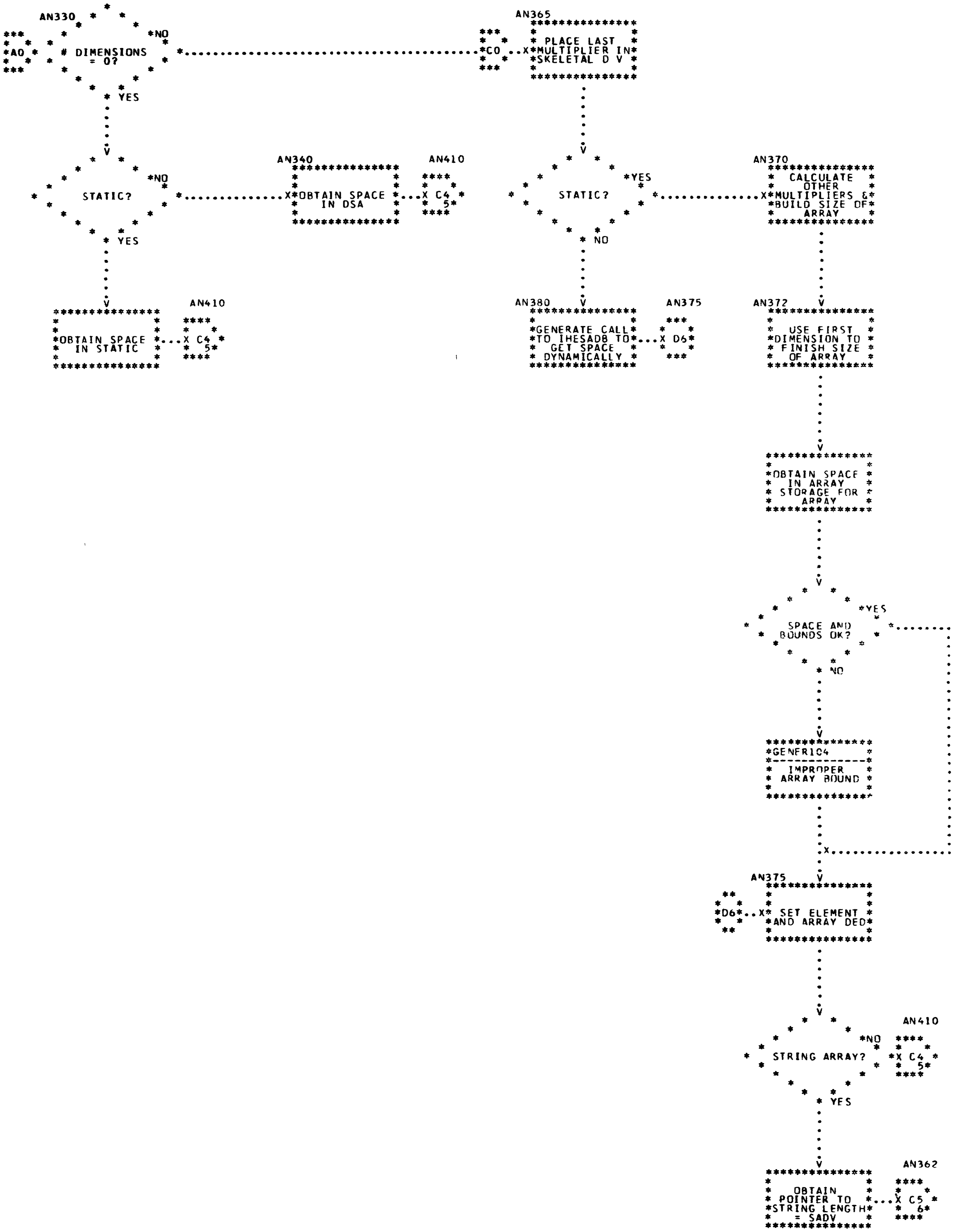


Chart 28. Attribute Node Creation (Page 7 of 10)

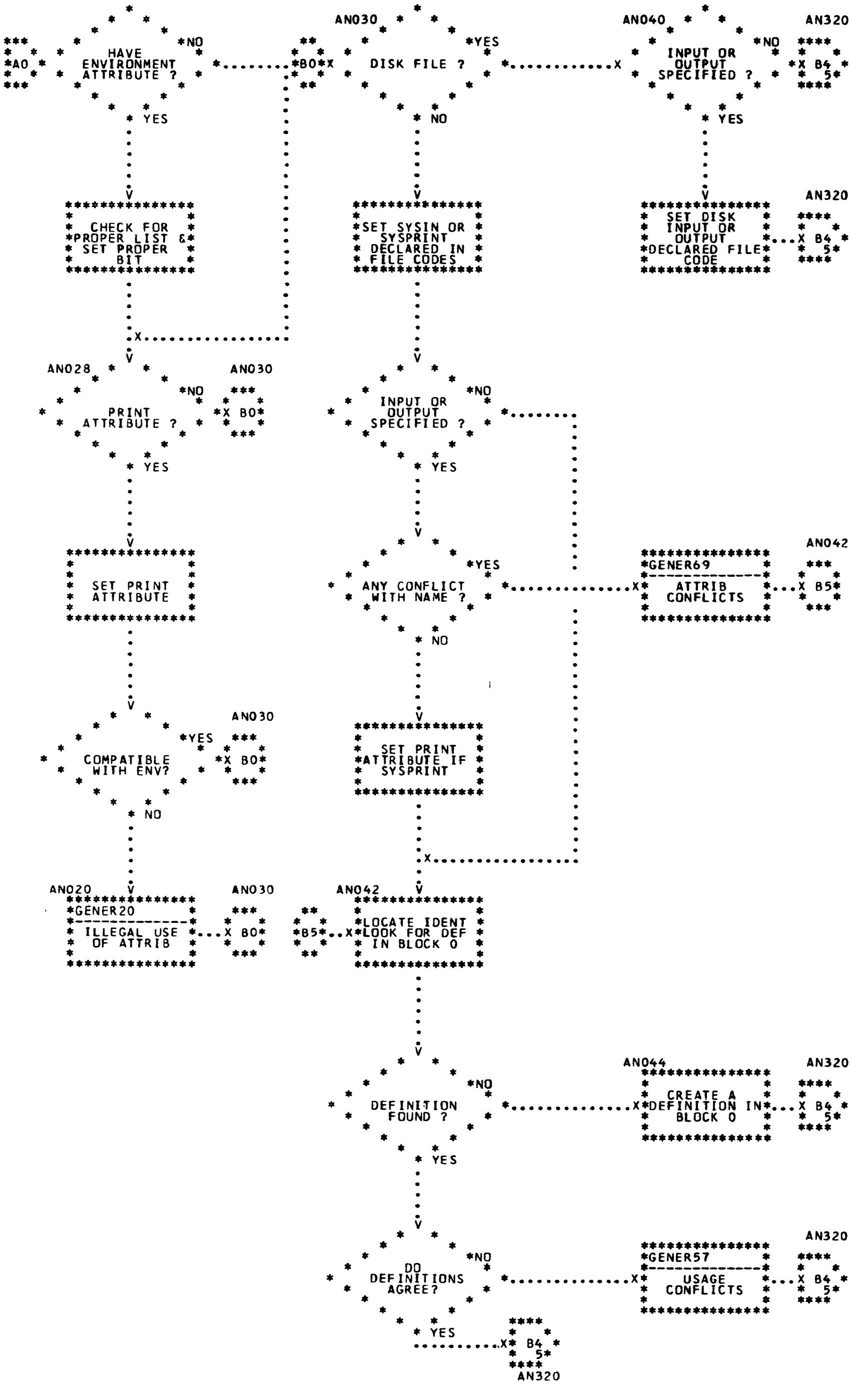


Chart 28. Attribute Node Creation (Page 8 of 10)

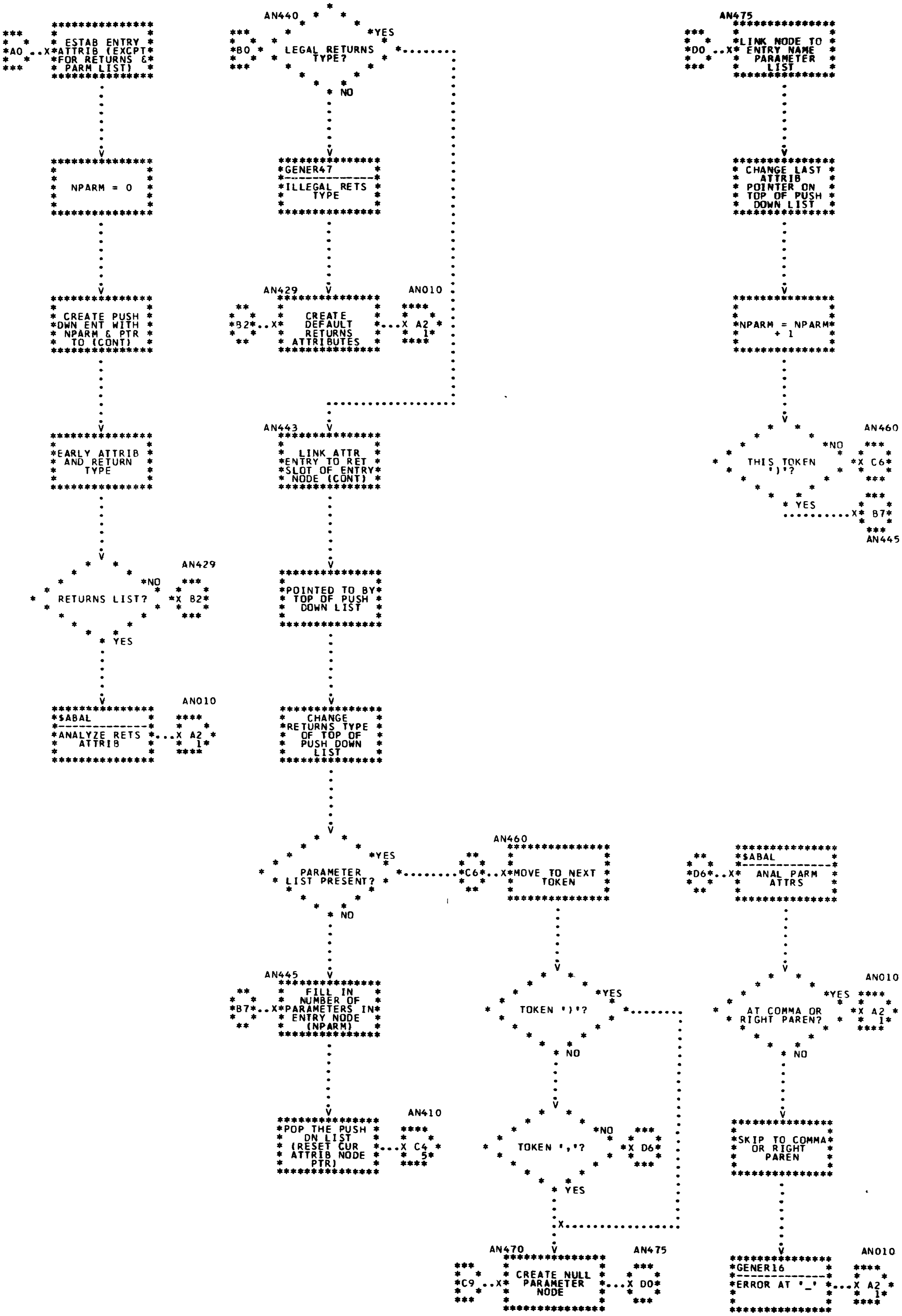


Chart 28. Attribute Node Creation (Page 9 of 10)

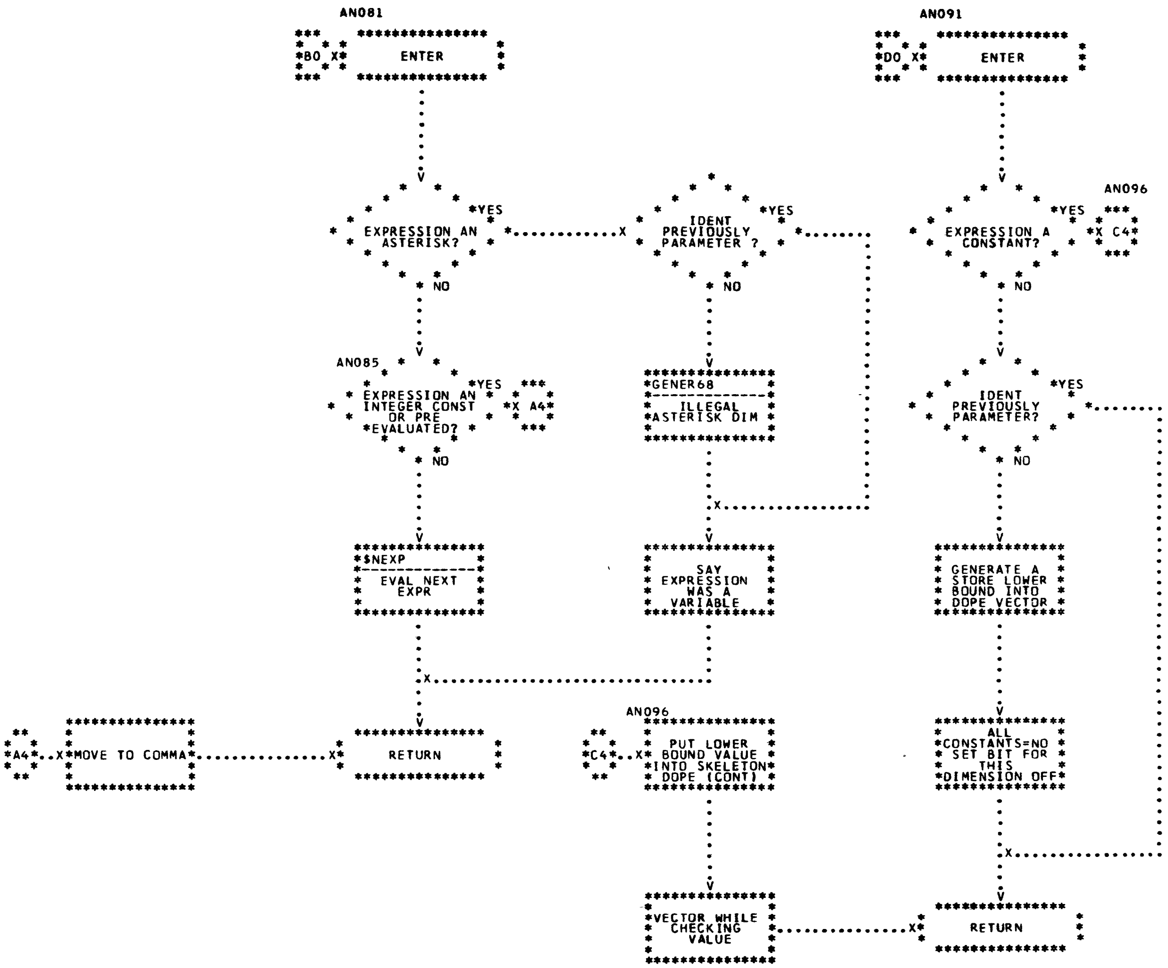


Chart 28. Attribute Node Creation (Page 10 of 10)

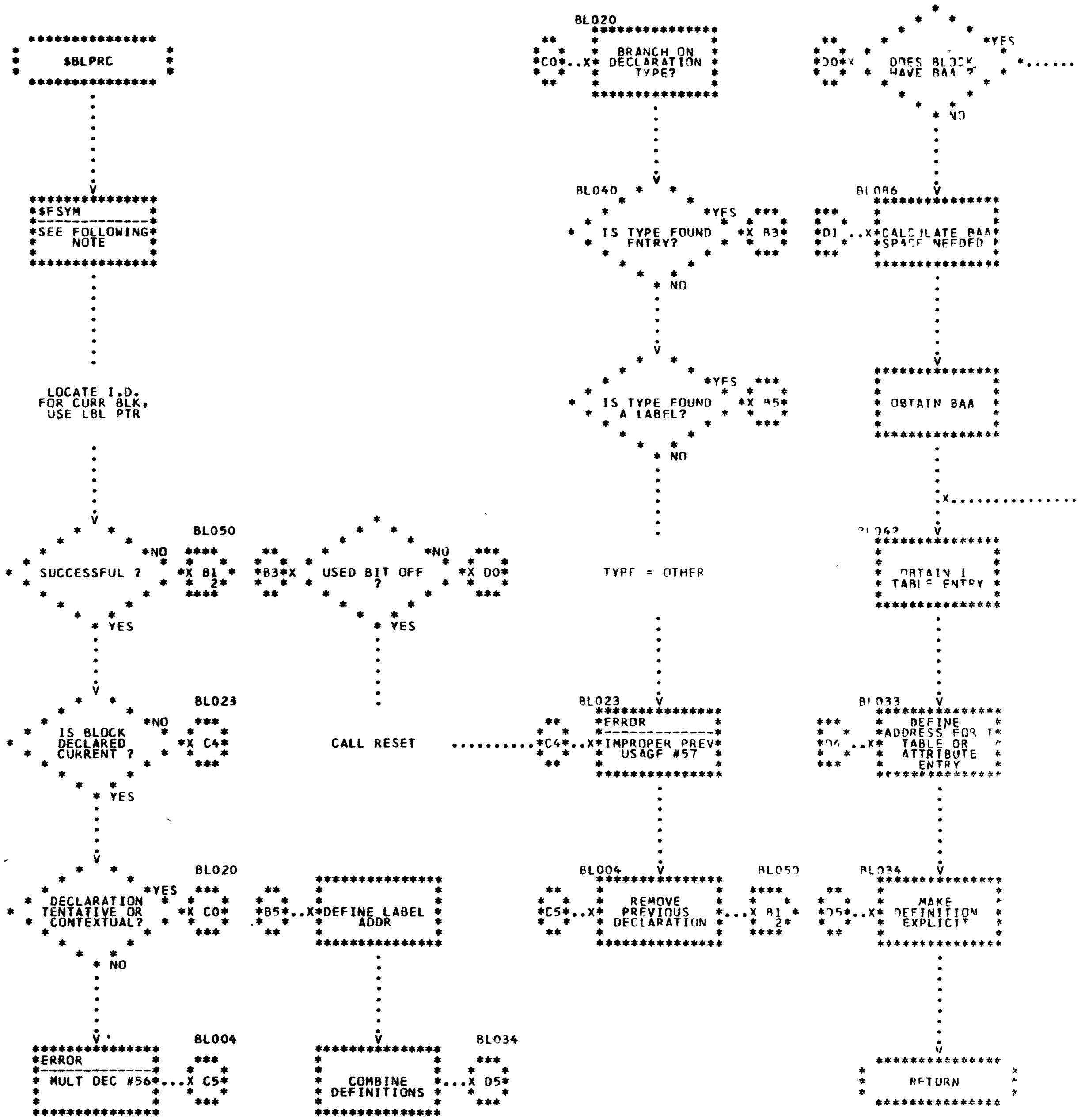


Chart 29. Label Processor (Page 1 of 2)

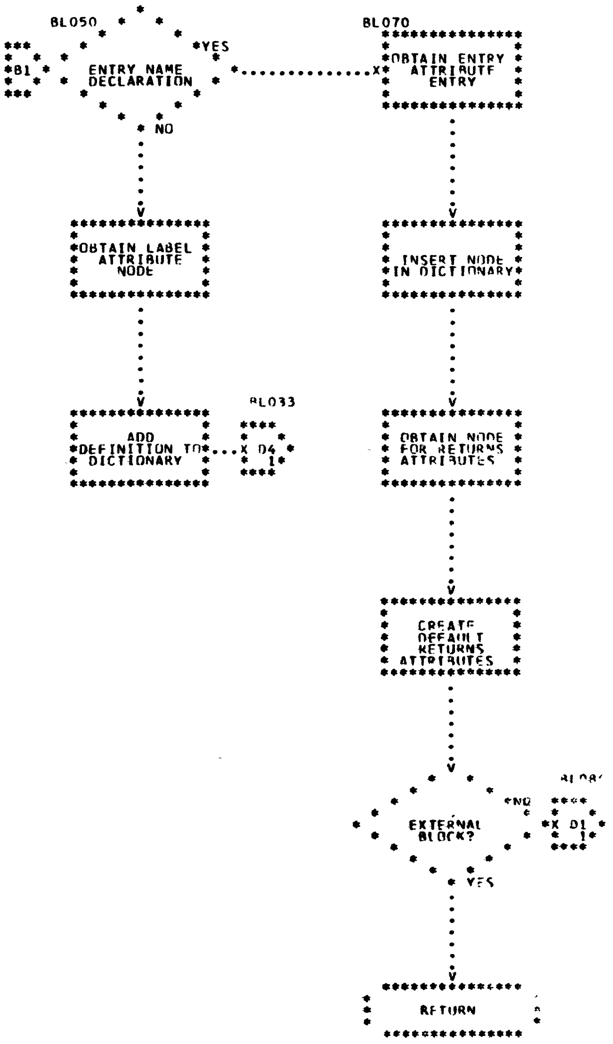


Chart 29. Label Processor (Page 2 of 2)

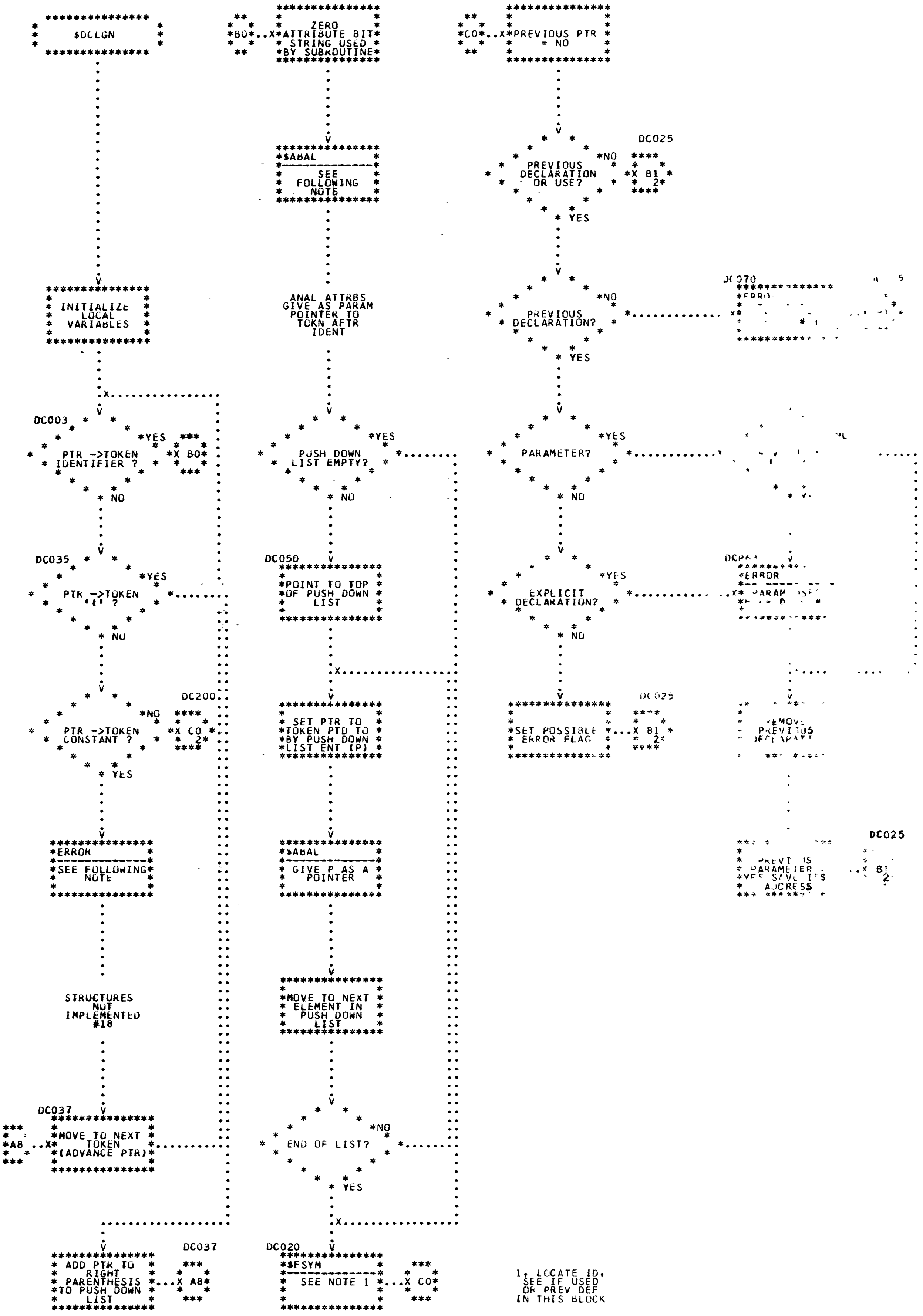


Chart 30. DCL Generator (Page 1 of 2)

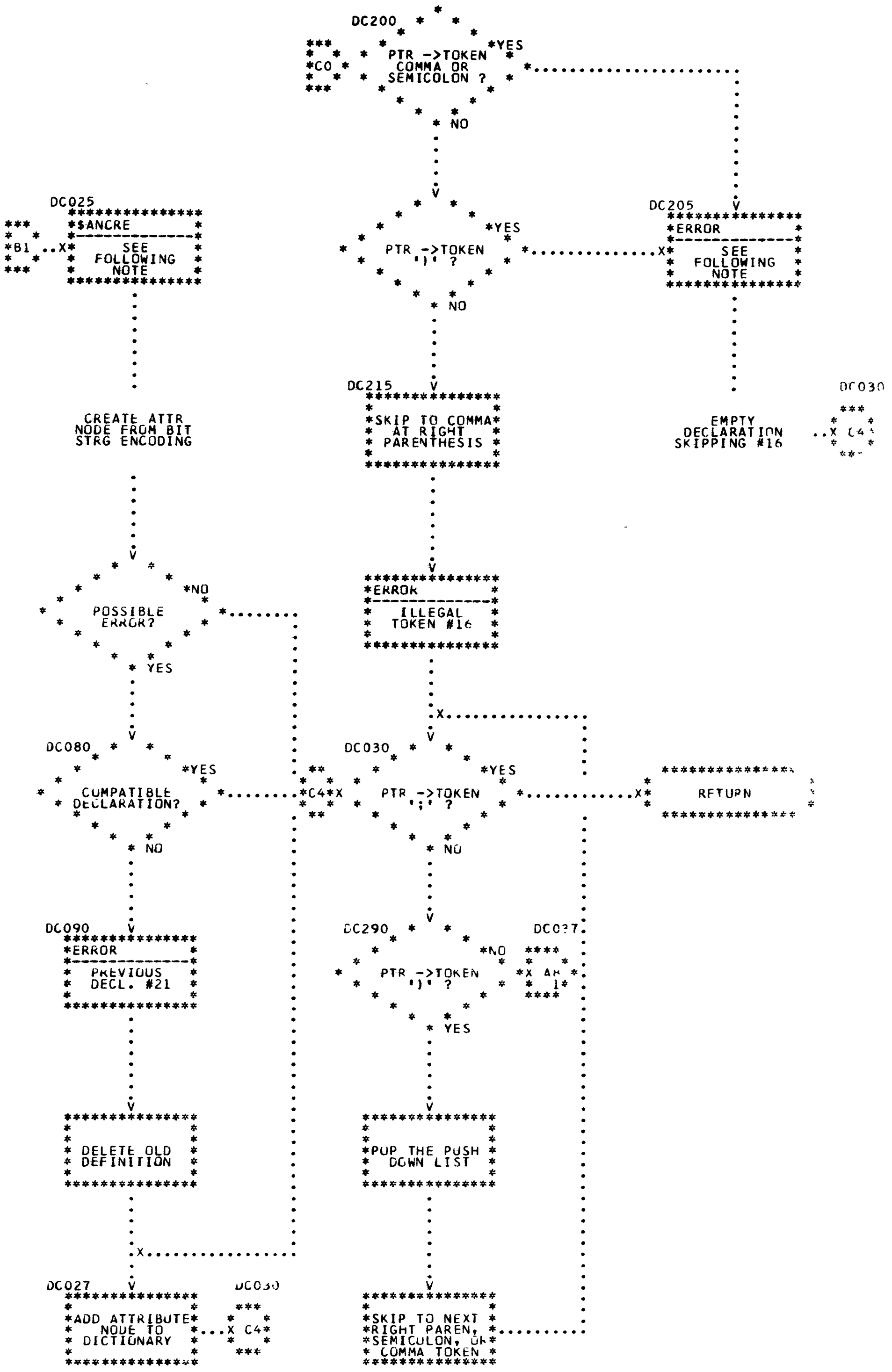


Chart 30. DCL Generator (Page 2 of 2)

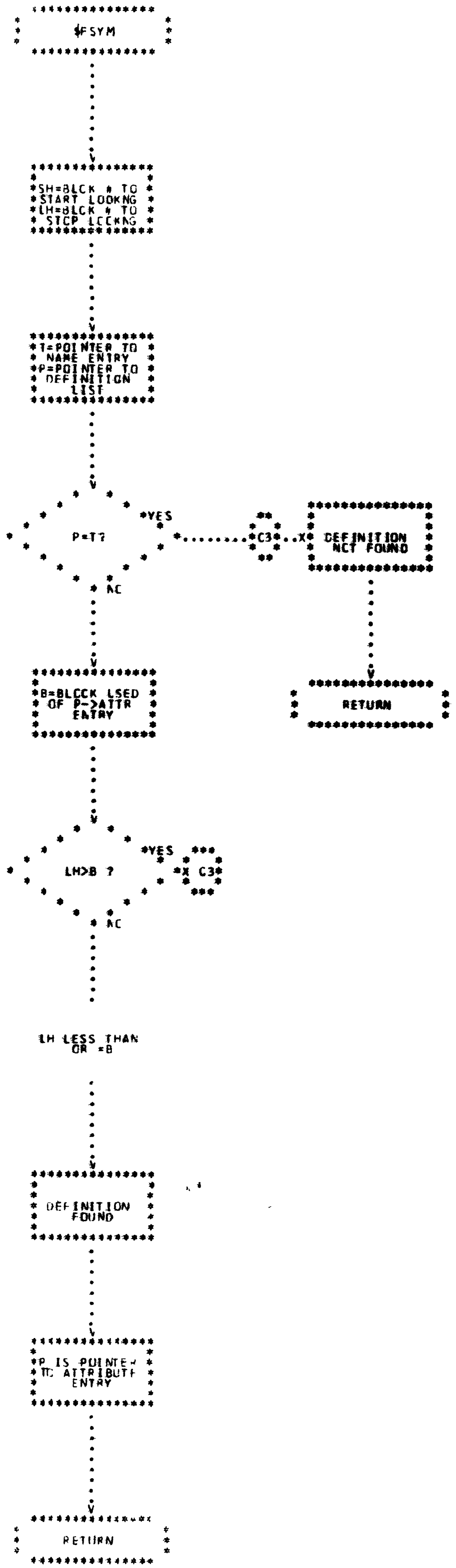


Chart 31. Locate Identifier

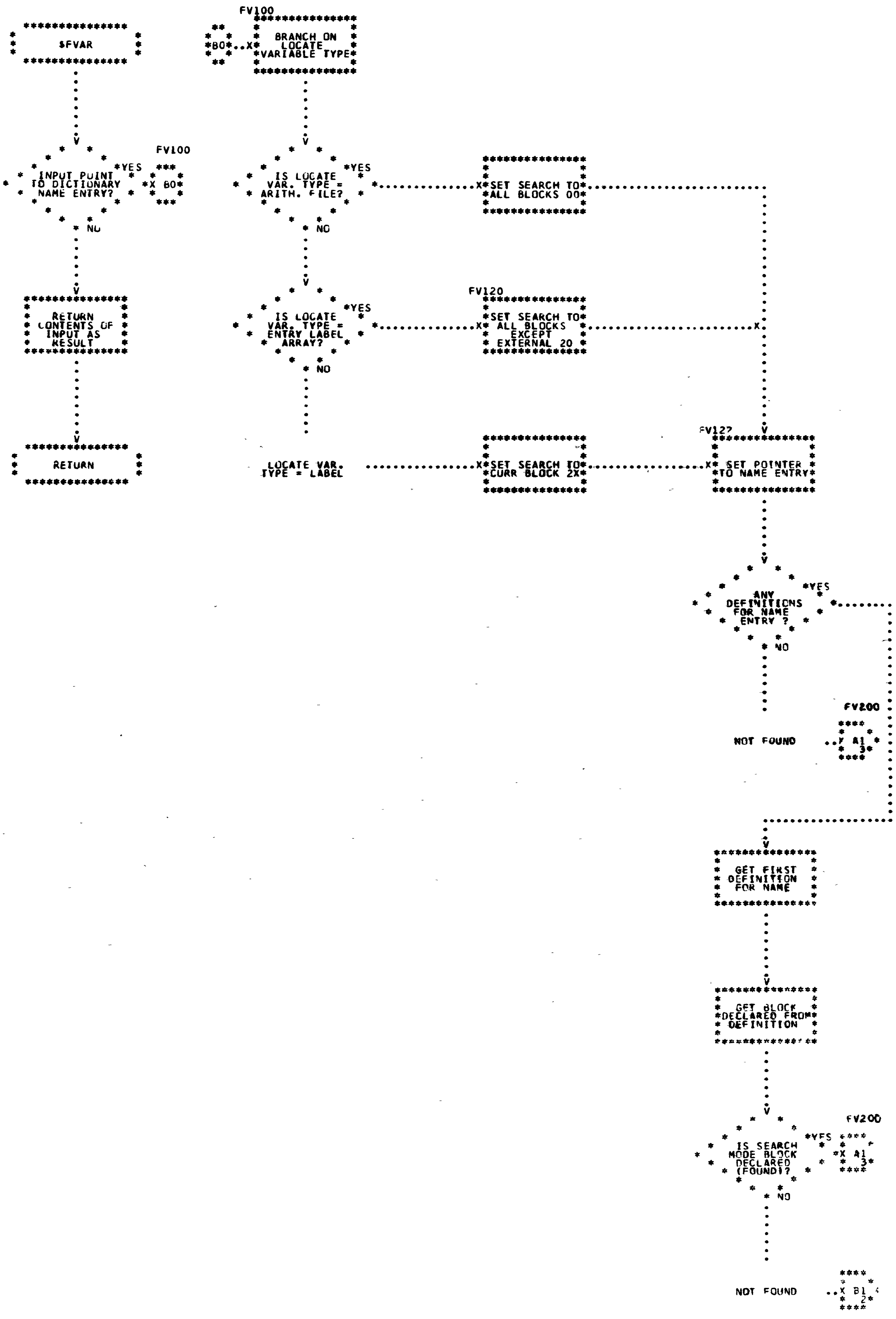


Chart 32. Locate Variable (Page 1 of 3)

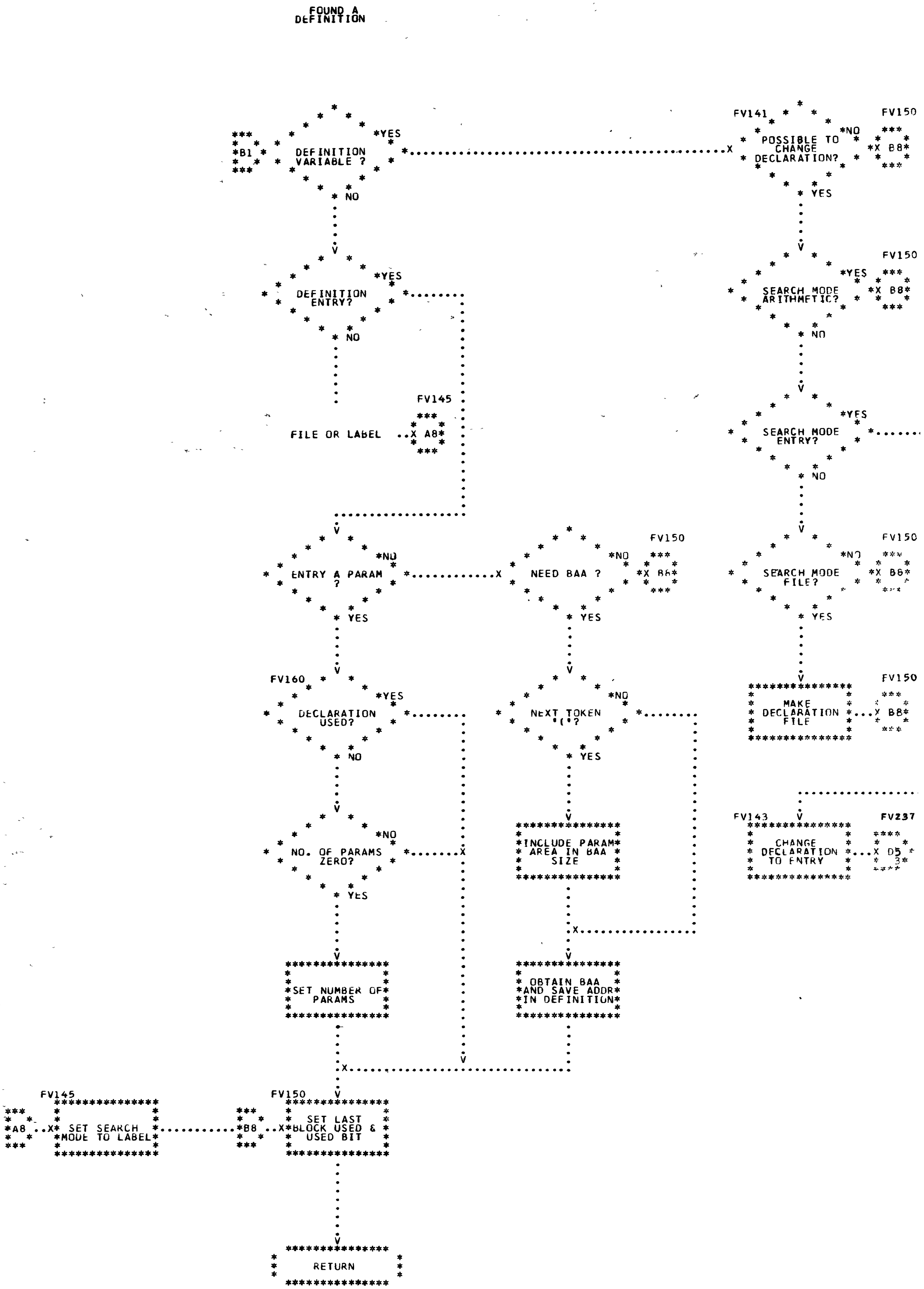


Chart 32. Locate Variable (Page 2 of 3)

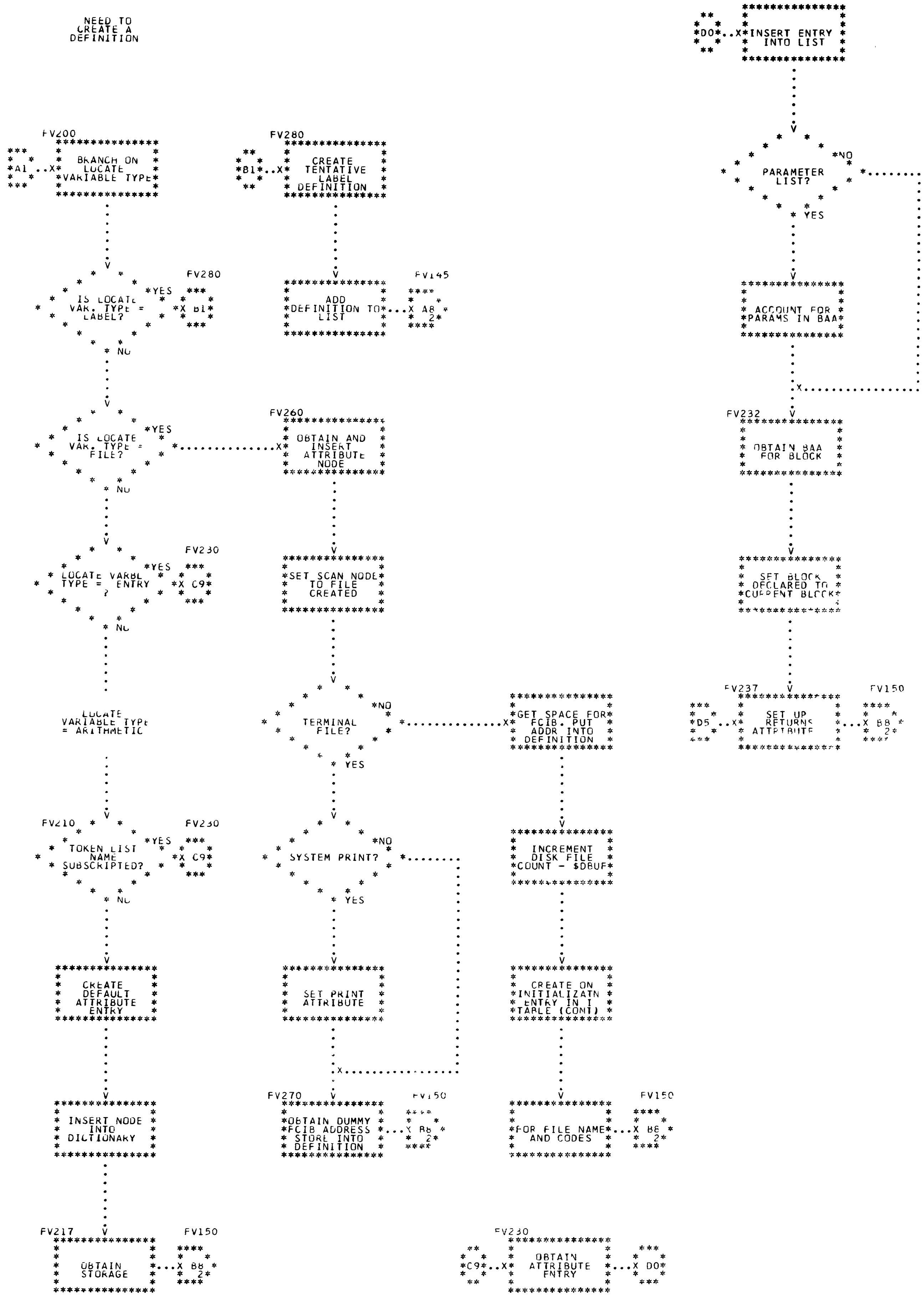


Chart 32. Locate Variable (Page 3 of 3)

PART 5 - I/O STATEMENT PROCESSING

The routines described in this subsection process the OPEN, CLOSE, GET, PUT, and FORMAT statements. These routines are discussed in alphabetic order according to their mnemonics, as shown below. Detailed logic diagrams for the routines appear at the end of the subsection.

GET Generator ($BGET)
PUT Generator ($BPUT)
Data Specification ($DDS)
I/O Specification ($DIOS)
O/C Specification ($DOCS)
Format List Generator ($FLG)
FORMAT Generator ($FMT)
Format Item ($FORI, $FORI2)
Format in Data List Processor ($FPDL)
OPEN/CLOSE Generator ($OPEN, $CLOSE)

TITLE: GET GENERATOR ($BGET)

Program Definition

Purpose and Usage

The GET Generator analyzes a GET statement and directs the generation of triads for the statement.

Description

This routine directs the identification of the options in a GET statement, namely, the file and data specification options. The I/O Specification routine ($DIOS) scans the statement looking for the keywords FILE, DATA, EDIT, LIST, and SKIP. Pointers (the file, data specification, and skip pointers) are set to indicate the location in the token table of the corresponding keywords.

Upon return from I/O Specification, a check is performed to see if the file option was present. If not, the default file of SYSIN is provided.

If the type of input is DATA, GET Generator proceeds to generate a symbol table triad for the data list, if one is present. If no data list is present, a marker is placed in the block information table so that a symbol table for the block will be generated at epilogue time. A triad is generated to perform a call to the Data-Directed Input routine (IHEDDIB). (For a description of the symbol table, see Appendix E.)

If the type of input is LIST or EDIT, triads are generated to call the List- or Edit-Directed GET Initiation and Termination routine (IHEIOAA) to initialize the operation. A switch indicating the type of I/O statement is set and the Data Specification routine ($DDS) is called to process the data list. On return, if the type of input is EDIT, the Format in Data List Processor routine ($FPDL) is called to process the format list. A triad is then generated to call IHEIOAT to terminate the operation.

Errors Detected

ERROR AT '___'. (16)
FILE NAME NOT INPUT FILE. (31)
ILLEGAL USE OF '___' IN DATA INPUT LIST. (33)
'SKIP' OPTION ILLEGAL HERE. (34)
ILLEGAL FILE DESIGNATION. (43)

Local Variables

BGNAME	Default Filename or FCB Address
BGPTR	Token Table Pointer to Semicolon
BGSYM	Symbol Table Triad Pointer

Program Interface

Entry Points

$BGET. No formal parameters, but requires a pointer to token table ($PTR).

Exit Conditions

Normal exit after translation. $PTR points to semicolon.

Routines Called

$DIOS	I/O Specification
$CERR	Compiler Error
$FVAR	Locate Variable
$DDS	Data Specification
$FPDL	Format in Data List Processor
$GTRIAD	Get Next Triad Entry
$FIND	Search-Insert
$XERR	Error Message Editor

Global Variables

$PTR	Token Table Pointer
$DSKIP	Skip Specification Pointer
$DFILE	File Specification Pointer
$DDATA	Data Specification Pointer
A List	Dictionary Attribute List
N List	Dictionary Name List
T Table	Token Table
$BIOTY	I/O Statement Type Indicator
$FCB	File Attribute Entry Pointer

Logic Diagram

Chart 33 shows the detailed logic diagram for the GET Generator routine.

TITLE: PUT GENERATOR ($BPUT)

Program Definition

Purpose and Usage

The PUT Generator analyzes a PUT statement and directs the generation of triads for the statement.

Description

This routine directs the identification of the location of the options in a PUT statement, namely, the file, skip, and data specification options. The I/O Specification routine ($DIOS) scans the statement looking for the keywords FILE, DATA, EDIT, LIST, and SKIP. Pointers (the file, data specification, and skip pointers) are set to indicate the location in the token table of the corresponding keywords.

Upon return from I/O Specification, a check is performed to see if the file option was present. If not, the default file of SYSPRINT is provided.

Triads are generated to call the Output Initialization with or without Skipping library routine (IHEIOBA or IHEIOBC) to initialize the operation without or with a skip respectively. A switch indicating the type of I/O statement is set and the Data Specification routine ($DDS) is called to process the data list. On return, if the type of output is EDIT, the Format in Data List Processor routine ($FPDL) is called to process the format list. If the output type is DATA, the triad is then generated to call the Data-Directed Output library routine (IHEDDOC) to terminate the operation.

Errors Detected

ERROR AT '___'. (16)
ILLEGAL FILE DESIGNATION. (43)
FILE NAME NOT OUTPUT FILE. (49)
NON-PRINT FILE--'SKIP' OPTION ILLEGAL. (50)
LIST MISSING AFTER '___'. (71)

Local Variables

BPPTR	Pointer to Semicolon Token
BPNME	Default Filename or FCB Address
BPFIB	FIB Around Symbol Table

Program Interface

Entry Points

$BPUT. No formal parameters, but requires a pointer to token table ($PTR).

Exit Conditions

Normal exit after translation; $PTR points to semicolon.

Routines Called

$NEXP	Expression Processor Controller
$DIOS	I/O Specifiction
$FVAR	Locate Variable
$DDS	Data Specification
$FPDL	Format in Data List Processor
$GTRIAD	Get Next Triad Entry
$FIND	Search-Insert
$NCONS	Constant Processor
$AREXP	Array Expression Error
$XERR	Error Message Editor

Global Variables

$PTR	Token Table Pointer
$DSKIP	Skip Specification Pointer
$DFILE	File Specification Pointer
$DDATA	Data Specification Pointer
N List	Dictionary Name List
A List	Dictionary Attribute List
T Table	Token Table
$BIOTY	I/O Statement Type Indicator
$FCB	File Attribute Entry Pointer

Logic Diagram

Chart 34 shows the detailed logic diagram for the PUT Generator routine.

TITLE: DATA SPECIFICATION ($DDS)

Program Definition

Purpose and Usage

This routine generates and directs the generation of triads to perform the operations required by the data list in an I/O statement.

Description

If an element in the data list is an expression that does not begin with a left parenthesis, the Expression Processor Controller is called to evaluate the expression. If the expression is an array expression, code is generated for it as though it were a repetitive specification from lower bounds to upper bounds by the Expander routine. Otherwise, a check is made to see if it is a legal expression type (e.g., not label).

If a left parenthesis is the first token in an element, whether the element is an expression enclosed in parentheses or a repetitive specification cannot be immediately determined. This question is resolved through a scan that provides a pointer to a comma if the left parenthesis began an expression, or to the token DO if the element is a repetitive specification.

If the element is a repetitive specification, the DO Generator is called to generate triads for the DO specification. The specification is then reanalyzed starting immediately from the right of the left parenthesis that began the repetitive specification. When the DO specification is reached again, an END statement is generated and the END Generator is called.

If the statement is PUT DATA, items within the expression are scanned and a symbol table entry is formed for each unsubscripted or subscripted scalar (Appendix E shows symbol table entry format). All contiguous unsubscripted scalars are entered into the same symbol table while a separate symbol table is constructed for each subscripted scalar.

Each symbol table is preceded by a call to the Data-Directed Output library routine (IHEDDOA for unsubscripted scalars or IHEDDOB for subscripted scalars) followed by a branch around the symbol table. The triads generated for one or more contiguous unsubscripted scalars are:

Call of library routine IHEDDOA
FIB (Forward Internal Branch)
Symbol table entry 1
.
.
.
Symbol table entry n
RFIB (Resolve Forward Internal Branch)

Note that a symbol table consists of a series of entries in generated code and is not a compiler-time table.

The triads generated for a subscripted scalar are:

Call of library routine IHEDDOB
FIB (Forward Internal Branch)
Symbol table entry 1
.
.
.
Symbol table entry n
RFIB (Resolve Forward Internal Branch)

For other types of statements, instructions are generated to set up the FCB for the operation. If the operation is EDIT, a branch and link to a format subroutine is generated. This subroutine is generated by the FORMAT Generator when it processes the FORMAT statement. If the operation is LIST, a call to the List- and Data-Directed Input library routine (IHELDIA) or the List-Directed Output library routine (IHELDOB) is generated. If the operation is GET, the expression is checked for consisting of only a scalar before the code is generated.

Any expansions and any implied DO's are ended. When the right parenthesis of the data list is reached, a return is made.

Errors Detected

ERROR AT '___'. (16)
ILLEGAL I/O EXPRESSION--SKIPPING TO '___'. (75)
ILLEGAL DATA OUTPUT ITEM. (76)
EXPRESSION ILLEGAL IN 'GET' DATA LIST. (77)

Local Variables

DDFIB	Branch Around Symbol Table
DDDATA	Data List Type Switch for PUT DATA
DDARRY	Array Switch
DDSYM	Symbol Table Address

Program Interface

Entry Points

$DDS. Needs data specification pointer ($DDATA) and statement type indicator ($BIOTY); no formal parameters.

Exit Conditions

Normal exit only.

Routines Called

$FVAR	Locate Variable
$NEXP	Expression Processor Controller
$EXPND	Expander
$DOG	DO Generator
$EDGN	END Generator
$EYPND	END Expand
$GTRIAD	Get Next Triad Entry
$SCDV	String Constant Dope Vector Initializer
$NCONS	Constant Processor
$WCTCT	Segment Management
$XERR	Error Message Editor

Global Variables

T Table	Token Table
$DDATA	Data Specification Pointer
$BIOTY	Statement Type Indicator
$FCBAD	FCB Base Address

Logic Diagram

Chart 35 shows the detailed logic diagram for the Data Specification routine.

TITLE: I/O SPECIFICATION ($DIOS)

Program Definition

Purpose and Usage

The I/O Specification routine is used by the GET Generator or PUT Generator to scan a statement and locate the file, skip, and data list sections of the statement.

Description

Each section of a GET or PUT statement consists of a keyword followed by tokens enclosed in parentheses. The SKIP and DATA keywords do not require parenthesized tokens, and the EDIT keyword requires two sets of parenthesized tokens.

This routine checks each keyword for the proper number of parenthesized tokens following it. No checking is performed on the tokens between the parentheses.

Three pointers can be set by this routine: $DFILE, $DDATA, and $DSKIP. $DFILE is set to point to the FILE token; $DSKIP is set to point to the SKIP token; and $DDATA is set to point to the EDIT, LIST, or DATA token.

Errors Detected

ERROR AT '___'. (16)
DUPLICATE '___' DESIGNATION--LAST USED. (70)
LIST MISSING AFTER '___'. (71)
DATA AND FORMAT LIST MISSING. (73)
FORMAT LIST MISSING. (74)

Local Variables

DIERR	Error Switch (0 = OFF)

Program Interface

Entry Points

$DIOS. No formal parameter but expects $PTR to point to token at which to begin scan.

Exit Condition

Normal exit. $PTR points to semicolon.

Routines Called

None

Global Variables

T Table	Token Table
$PTR	Token Table Pointer
$DSKIP	Skip Specification Pointer
$DDATA	Data Specification Pointer
$DFILE	File Specification Pointer

Logic Diagram

Chart 36 shows the detailed logic diagram for the I/O Specification routine.

TITLE: O/C SPECIFICATION ($DOCS)

Program Definition

Purpose and Usage

The O/C Specification routine is used by the OPEN/CLOSE Generator to scan a statement and locate the file, title, and input/output attribute sections of the statement.

Description

Each section of an OPEN or CLOSE statement consists of a keyword followed by tokens enclosed in parentheses (except for INPUT and OUTPUT keywords).

This routine checks each keyword for the proper number of parenthesized tokens following it. No checking is performed on the tokens between the parentheses.

Three pointers, which point to the file list, title list, and input or output token, are set by this routine. If a token in a particular category was not present, the pointer is null.

A flag is set if more than one option group is present in the statement (that is, for a multiple OPEN or CLOSE). This will inform the calling routine to return after processing the current option group.

Errors Detected

ERROR AT '___'. (16)
DUPLICATE '___' DESIGNATION--LAST USED. (70)
LIST MISSING AFTER '___'. (71)

Local Variables

DERR	Error Switch (0 = OFF)

Program Interface

Entry Points

$DOCS. No formal parameters but expects $PTR to point to token at which to begin scan.

Exit Conditions

Normal exit. $PTR points to either semicolon or comma.

Routines Called

$XERR	Error Message Editor

Global Variables

T Table	Token Table
$PTR	Token Table Pointer
$DFILE2	File Token Pointer
$DTITLE	Title Token Pointer
$DINOUT	Input/Output Token Pointer
$DOPT	Option Group Flag

Logic Diagram

Chart 37 shows the detailed logic diagram for the O/C Specification routine.

TITLE: FORMAT LIST GENERATOR ($FLG)

Program Definition

Purpose and Usage

The Format List Generator performs the syntax analysis and code generation for a format list.

Description

Because the code generated for a FORMAT statement may call the code generated for other FORMAT statements, it is necessary to ensure that any temporary results obtained for replication factors use locations that cannot be changed during the execution of another format or calculation of data list elements. Therefore, another level of temporary storage definitions is created during the processing of the format list.

Each format specification (except A without a length) creates a format element descriptor (FED). Each descriptor is one word long and contains a halfword integer value followed by two one-byte values. These three values are obtained from the expressions within a specification. This routine generates code to place the address of the FED in the file control block (FCB) and to branch to the proper library routine. A complex format specification consists of two E and/or F format specifications. Besides putting the addresses of each of the FED's in the FCB, the type (E or F) is also placed in the FCB. Thus the library receives only one call for a complex data item.

Since the code generated for format lists is driven by the code generated for data lists, the format list code must return to the data list after performing each data specification and, in this process, tell the data list code where to continue in the format list. This address and the FCB address are kept in the format adcon area ($FORAD).

If a remote format is referenced, three addresses must be passed to $FORAD:

1. Address of FCB
2. Return address to format list
3. Return address to data list

To resolve extent of replications, a variable called FLSWT is used to specify n item forms, and a variable called FLPLEV (parenthesis level) is used to specify n format-list forms. Figure 3-3 shows an expansion of a format list into code.

Format List = (F (3), (2*N) A(10), 5(F(3), SKIP, 3 E(10,3)))

Code Generated	Spec F (3)
	Change FMT address and return
α_2	DO-loop for 2*N
	Spec A(10)
	Change FMT address and return
	Branch to α_2
α_4	DO-loop for 5
	Spec F (3)
	Change FMT address and return
	Spec SKIP
	3
α_6	DO-loop for 3
	Spec E(10,3)
	Change FMT address and return
	Branch to α_6
	Branch to α_4

Figure 3-3. Format List Generation Example

Errors Detected

ERROR AT '___'. (16)
ILLEGAL USE OF '___'. (29)
ITERATION FACTOR NOT PARENTHESIZED. (78)
'___' ILLEGAL FORMAT ITEM--SKIPPING TO '___'. (79)
'___' INCOMPLETE FORMAT ITEM--SKIPPING TO '___'. (80)
ILLEGAL COMPLEX FORMAT ITEM--SKIPPING TO '___'. (81)
'___' NOT FORMAT LABEL--SKIPPING TO '___'. (82)

Local Variables

FLPLEV	DO Level
FLSWT	Replication Switch

<u>Program Interface</u>

Entry Points

$FLG. No formal parameters. Expects $PTR to point to left parenthesis of format list.

Exit Conditions

Normal exit only. $PTR points to right parenthesis of format list.

Routines Called

$DEXP	DO-Loop Triad Builder
$EDGN	END Generator
$NEXP	Expression Processor Controller
$FORI	Format Item
$FVAR	Locate Variable
$GTRIAD	Get Next Triad Entry
$NCONS	Constant Processor
$XERR	Error Message Editor
$AREXP	Array Expression Error
$WEXP	Segment Management

Global Variables

T Table	Token Table
$FED	Dummy FED
$PTR	Token Table Pointer
$FCBAD	FCB Base Address
$FORAD	Format Adcon Area

Logic Diagram

Chart 38 shows the detailed logic diagram for the Format List Generator routine.

TITLE: FORMAT GENERATOR ($FMT)

Program Definition

Purpose and Usage

The FORMAT Generator directs translation of a FORMAT statement into triads. It also generates the triads necessary to take the FORMAT statement out of the normal flow of control.

Description

Because this statement may be referenced during execution of other statements (GET, PUT, and FORMAT), temporaries used in the statement may not be used by any other statements.

The first instructions generated perform a branch to the statement which follows. Next the label on the FORMAT statement is defined and the location from which the format was entered is saved. After completion of the format, a return is made to this location.

Errors Detected

ERROR AT '___'. (16)
UNLABELED FORMAT STATEMENT. (30)

Local Variables

FMTFIB	Branch Around Format Triad Pointer

Program Interface

Entry Points

$FMT. Expects $PTR to point to token following the FORMAT statement.

Exit Conditions

Normal exit only. $PTR points to semicolon.

Routines Called

$BLPRC	Label Processor
$FLG	Formal List Generator
$GTRIAD	Get Next Triad Entry
$TCODE	Triad Code Generator
$XERR	Error Message Editor

Global Variables

T Table	Token Table
$PTR	Token Table Pointer
$CLBLS	Label Switch
$FORAD	Format Adcon Area

Comments

No checking is done at compile or execution time to insure that formats are not used recursively.

Logic Diagram

Chart 39 shows the detailed logic diagram for the FORMAT Generator routine.

TITLE: FORMAT ITEM ($FORI, $FORI2)

Program Definition

Purpose and Usage

The Format Item routine is used by the Format List Generator to analyze and generate any code necessary for expressions in a format specification. It creates an FED and returns the address of the FED.

Description

Expressions in the parentheses following the format type specification are evaluated. If constants, the value is placed in a dummy FED. If expressions, space in the object program is obtained for an FED and the value is stored in the FED space. (The FED space is initialized with any constant values.) If all expressions are constants, the FED is defined as a constant.

If the type specification is not E or F, there can be only one expression. If the type is F, there can be up to three expressions. Any expressions not present are set to zero. An E specification may contain either two or three expressions. If only two are present, the third is set to the value of the second plus one.

Errors Detected

ERROR AT '___'. (16)
FORMAT ITEM HAS INCORRECT NO. OF FIELDS. (83)

Local Variables: None

Program Interface

Entry Points

$FORI. No formal parameters. Expects pointer to left parenthesis of specification in G2 and type of format in G4.

$FORI2. No formal parameters. Defines only a constant FED.

Exit Conditions

Normal exit. G2 points to token following right parenthesis of specification.

Normal exit. G2 not touched.

Routines Called

$NEXP	Expression Processor Controller
$NCONS	Constant Processor
$GTRIAD	Get Next Triad Entry
$AREXP	Array Expression Error
$XERR	Error Message Editor
$WEXP	Segment Management

Global Variables

T Table	Token Table
$FED	Dummy FED
$FEDC	FED Constant Flag
$FEDNM	Number of FED Fields

Logic Diagram: See Chart 40.

TITLE: FORMAT IN DATA LIST PROCESSOR ($FPDL)

Program Definition

Purpose and Usage

The Format in Data List Processor is used by the GET Generator or PUT Generator to process a format list for edit-directed I/O.

Description

This routine advances the token pointer passed to the left parenthesis of the format list. A branch around the format is generated, and the address of the format is defined. After the format is translated by the Format List Generator ($FLG), this routine directs the generation of code necessary to branch back to the beginning of the format.

If the last format item was a data type specification, two triads loading the adcon registers are removed. A branch to the beginning of the format is generated. Otherwise, the return address register and FCB register must be loaded before branching back to the beginning of the format.

After the format is completed, the branch around the format is resolved, and the address of the format is saved.

Errors Detected

None

Local Variables

FPBRNH	Branch Around Format Triad Pointer

Program Interface

Entry Points

$FPDL. Register G2 has pointer to left parenthesis of data list in I/O statement.

Exit Conditions

Normal exit is to four bytes after call.
Exit immediately after call if no data list is present.

Routines Called

$FLG	Format List Generator
$GTRIAD	Get Next Triad Entry
$TCODE	Triad Code Generator
$WCTCT	Segment Management

Global Variables

T Table	Token Table

Logic Diagram

Chart 41 shows the detailed logic diagram for the Format in Data List Processor routine.

TITLE: OPEN/CLOSE GENERATOR ($OPEN, $CLOSE)

Program Definition

Purpose and Usage

The OPEN/CLOSE Generator analyzes an OPEN or CLOSE statement and directs the generation of triads for the statement.

Description

This routine directs the identification of the location of the options in an OPEN or CLOSE statement, namely, the file, input/output, and title options. The O/C Specification routine scans the statement looking for the keywords FILE, INPUT, OUTPUT, and TITLE. Pointers (the file, input/output, and title pointers) are set to indicate the location of the token list of the corresponding keywords.

Upon return from O/C Specification, a check is made to see if the file option was present. If not, an error condition exists. Triads are generated to load the address of the file control interface block. For CLOSE, a triad is generated to call the Close routine (IHECLOSE).

For the OPEN statement, triads are generated to test the OPEN/mode flag in the FCIB and, if open, branch around the rest of the code that will be generated for the OPEN statement.

If the input/output pointer is active, a triad is generated to set the appropriate file code bit in the FCIB. An error condition is detected if the INPUT or OUTPUT specification in the OPEN statement conflicts with that of a previous usage.

If the title pointer is null, an initialization entry is created that will use the filename for the title. If the title pointer is active, the Expression Processor Controller is called to evaluate the title. A dummy attribute for the title is created that has the address of the FCIB and target attributes of string and is eleven characters in length.

The Expression Processor Controller generates triads that move the eleven-character title to the FCIB. A triad to call the Open routine (IHEOPEN) is generated. For both OPEN and CLOSE, another option group can exist (that is, more than one file can be opened or closed). In this case, the entire process is repeated for the next group.

Errors Detected

ERROR AT '___'. (16)
FILE NAME MISSING. (40)
'___' NOT FILE NAME. (41)
ILLEGAL FILE DESIGNATION. (43)
USE OF '___' HERE CONFLICTS WITH PREVIOUS USAGE. (57)
ILLEGAL TITLE DESIGNATION. (110)

Local Variables

OPSW	OPEN/CLOSE Switch
OPPTR	Save Area for $PTR
OPTKN	Dummy Token Area for Title
OPTITL	Title Switch

Program Interface

Entry Points

$OPEN. No formal parameters. Requires a pointer to token table ($PTR).

$CLOSE. No formal parameters. Requires a pointer to token table ($PTR).

Exit Conditions

Normal exit only. $PTR points to semicolon.

Routines Called

$DOCS	O/C Specification
$FVAR	Locate Variable
$NEXP	Expression Processor Controller
$GTRIAD	Get Next Triad Entry
$TCODE	Triad Code Generator
$AREXP	Array Expression Error
$XERR	Error Message Editor

Global Variables

$PTR	Token Table Pointer
$DFILE2	File Token Locator
$DINOUT	Input/Output Token Locator
$DTITLE	Title Token Locator
$DOPT	Option Group Flag
T Table	Token Table
$TITLE	Dummy Title Attribute Entry
I Table	Initialization Table
A List	Dictionary Attribute List

Logic Diagram

Chart 42 shows the detailed logic diagram for the OPEN/CLOSE Generator routine.

PART 5 LOGIC DIAGRAMS

The detailed logic diagrams for the routines that process GET, ~~PUT~~, OPEN, CLOSE, and FORMAT statements follow.

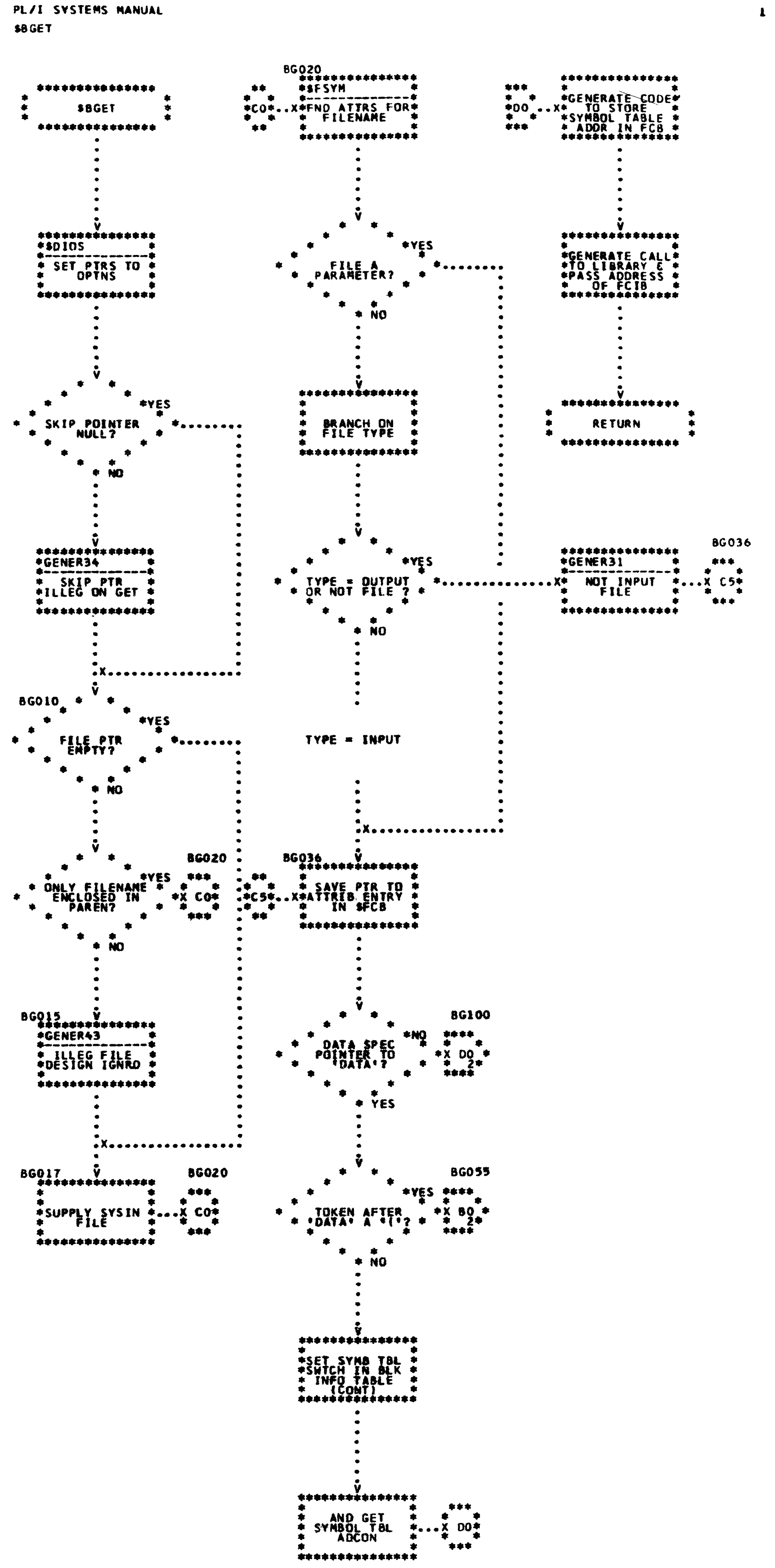
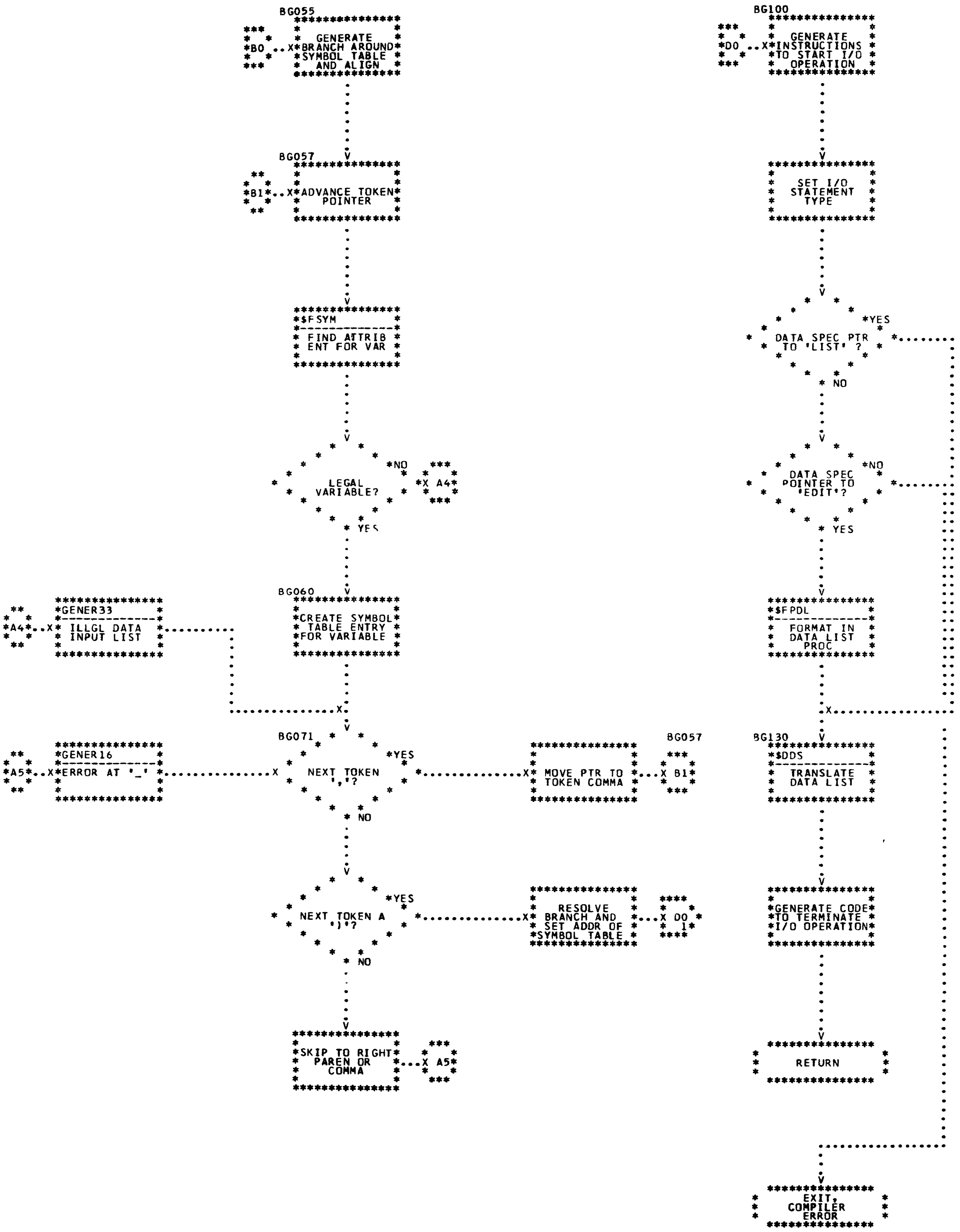


Chart 33. GET Generator (Page 1 of 2)


BG055
B0
GENERATE BRANCH AROUND SYMBOL TABLE AND ALIGN
BG057
B1
ADVANCE TOKEN POINTER
$FSYM
FIND ATTRIB ENT FOR VAR
LEGAL VARIABLE?
NO
A4
YES
BG060
CREATE SYMBOL TABLE ENTRY FOR VARIABLE
GENER33
A4
ILLGL DATA INPUT LIST
BG071
NEXT TOKEN ','?
YES
NO
GENER16
A5
ERROR AT '_'
MOVE PTR TO TOKEN COMMA
BG057
B1
NEXT TOKEN A ')'?
YES
NO
RESOLVE BRANCH AND SET ADDR OF SYMBOL TABLE
D0 1
SKIP TO RIGHT PAREN OR COMMA
A5
BG100
D0
GENERATE INSTRUCTIONS TO START I/O OPERATION
SET I/O STATEMENT TYPE
DATA SPEC PTR TO 'LIST' ?
YES
NO
DATA SPEC POINTER TO 'EDIT'?
NO
YES
$FPDL
FORMAT IN DATA LIST PROC
BG130
$DDS
TRANSLATE DATA LIST
GENERATE CODE TO TERMINATE I/O OPERATION
RETURN
EXIT, COMPILER ERROR

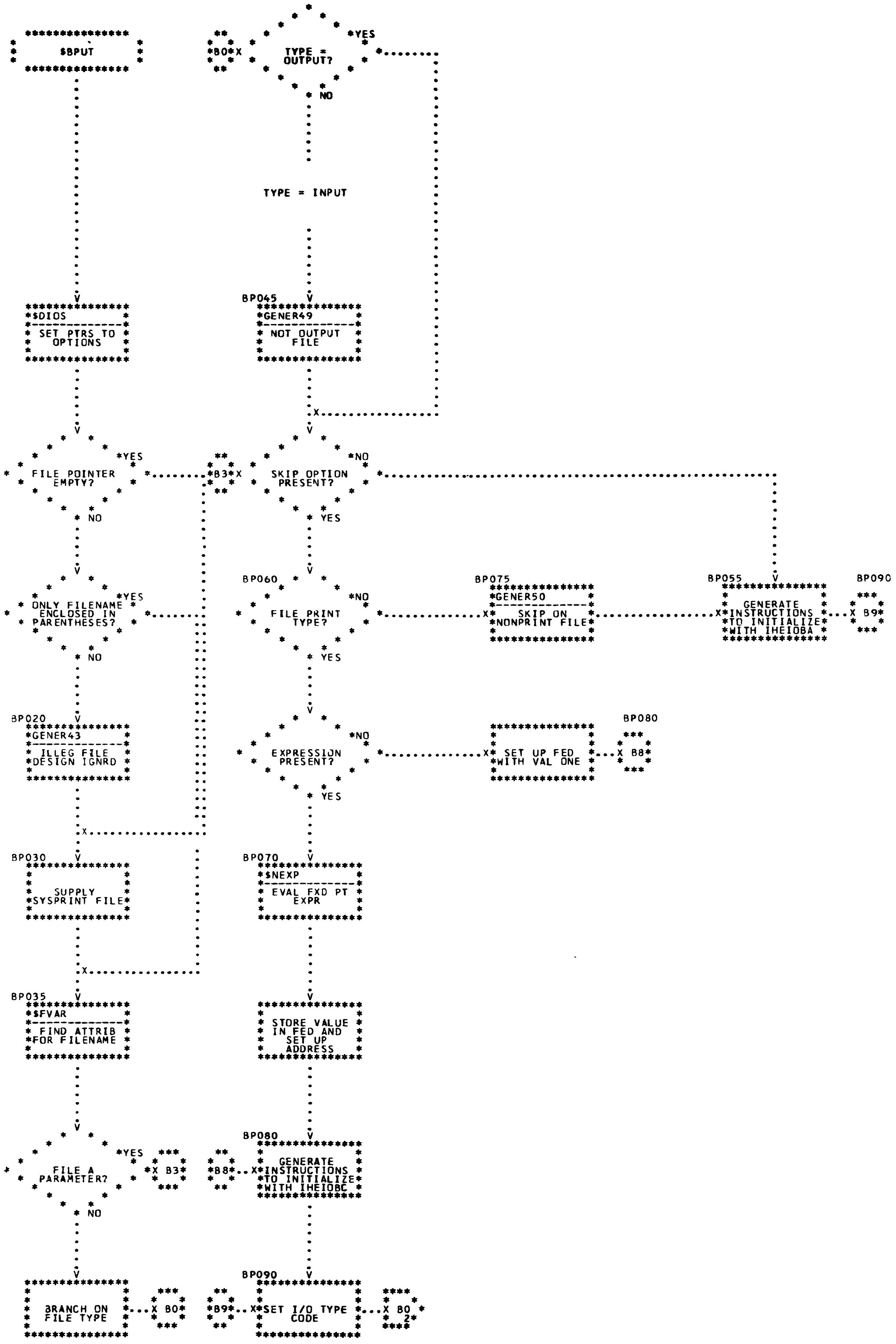


Chart 34. PUT Generator (Page 1 of 2)

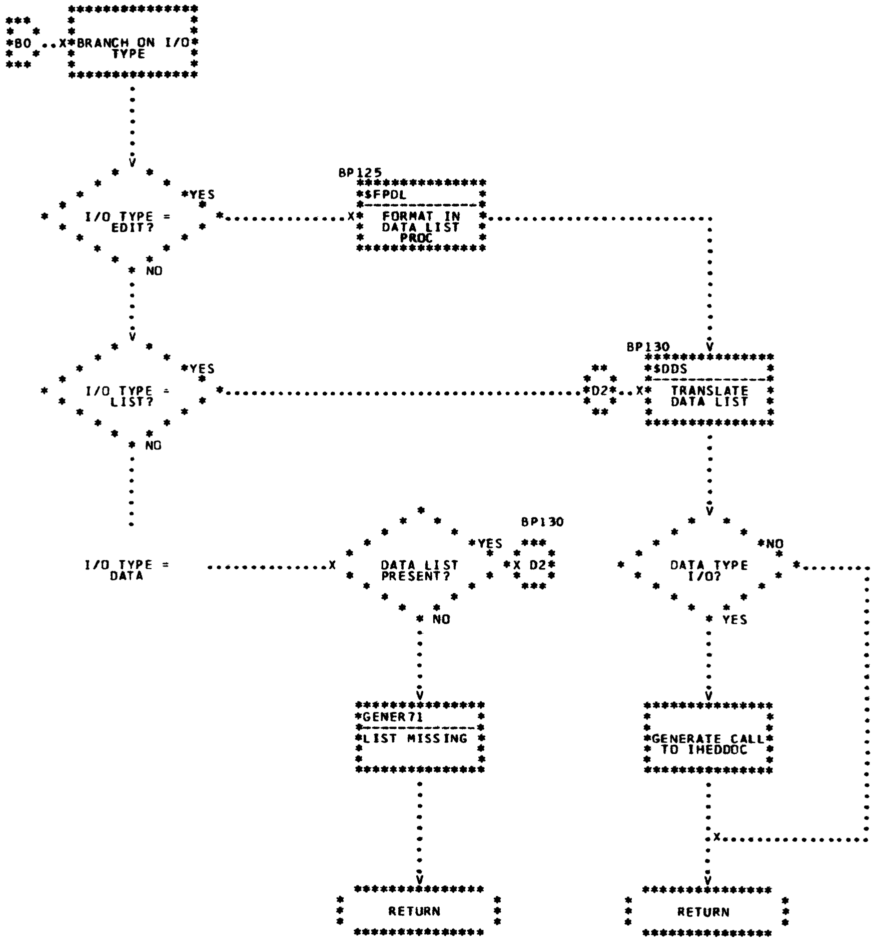


Chart 34. PUT Generator (Page 2 of 2)

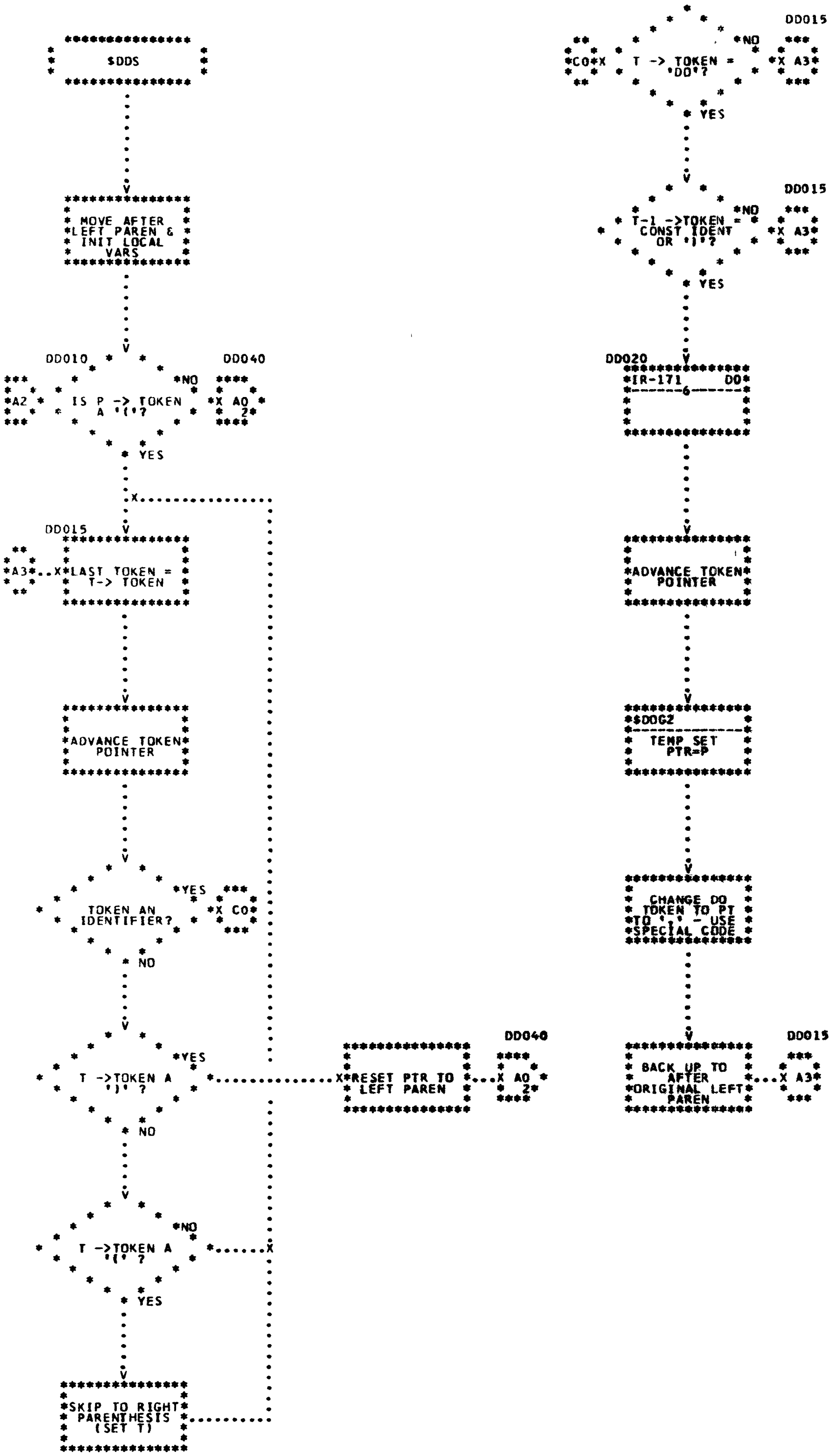


Chart 35. Data Specification (Page 1 of 6)

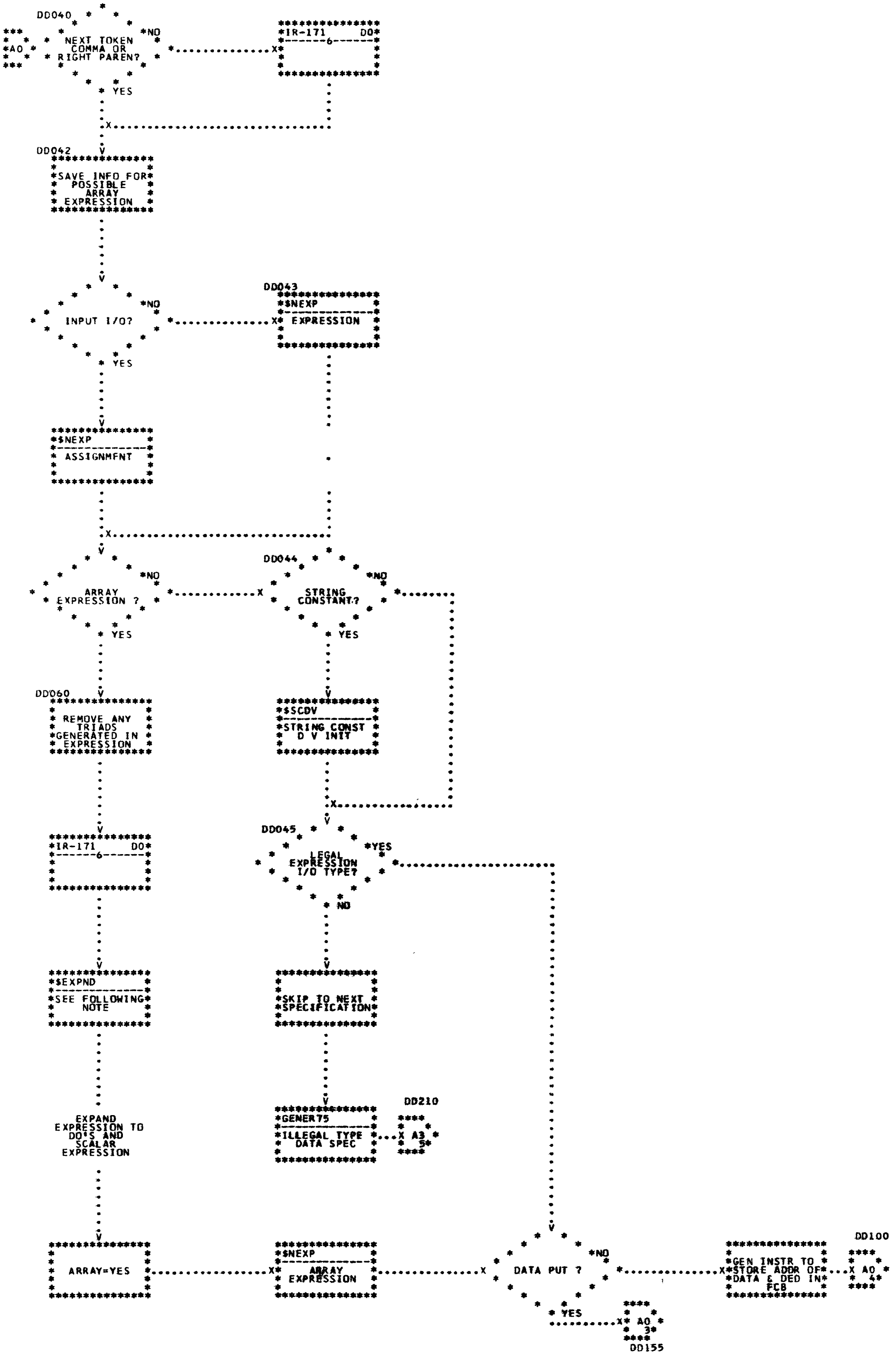


Chart 35. Data Specification (Page 2 of 6)

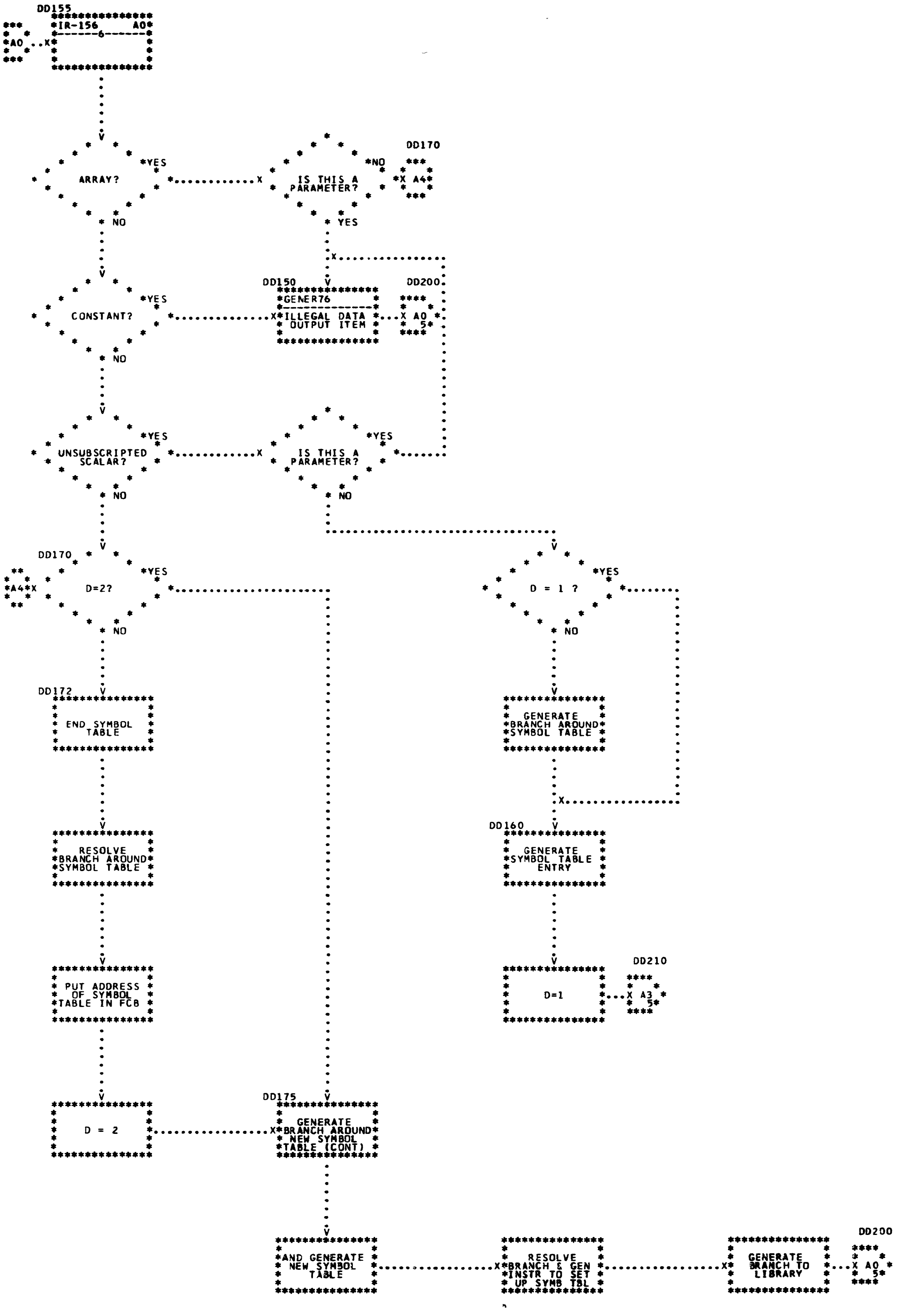


Chart 35. Data Specification (Page 3 of 6)

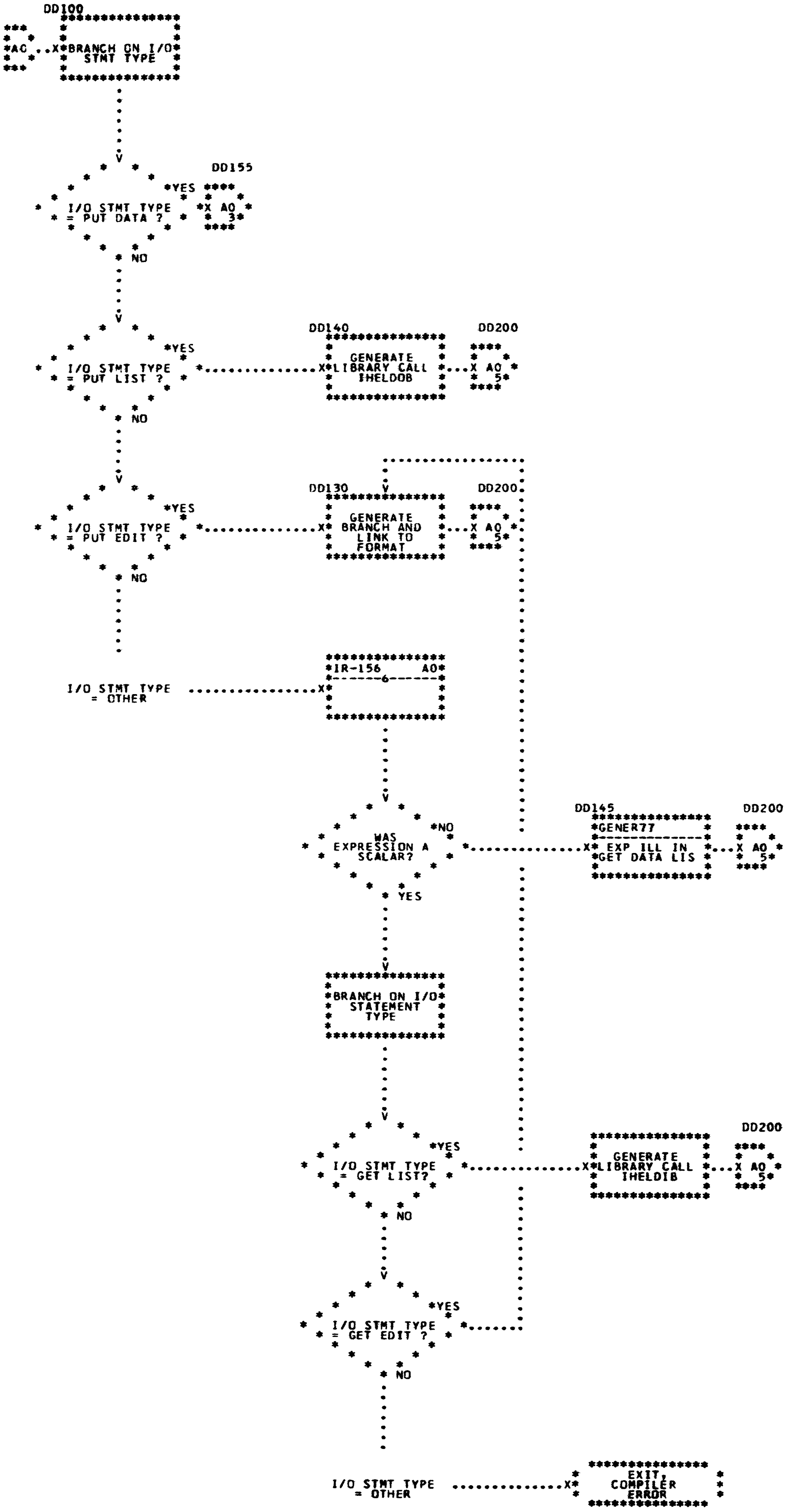


Chart 35. Data Specification (Page 4 of 6)

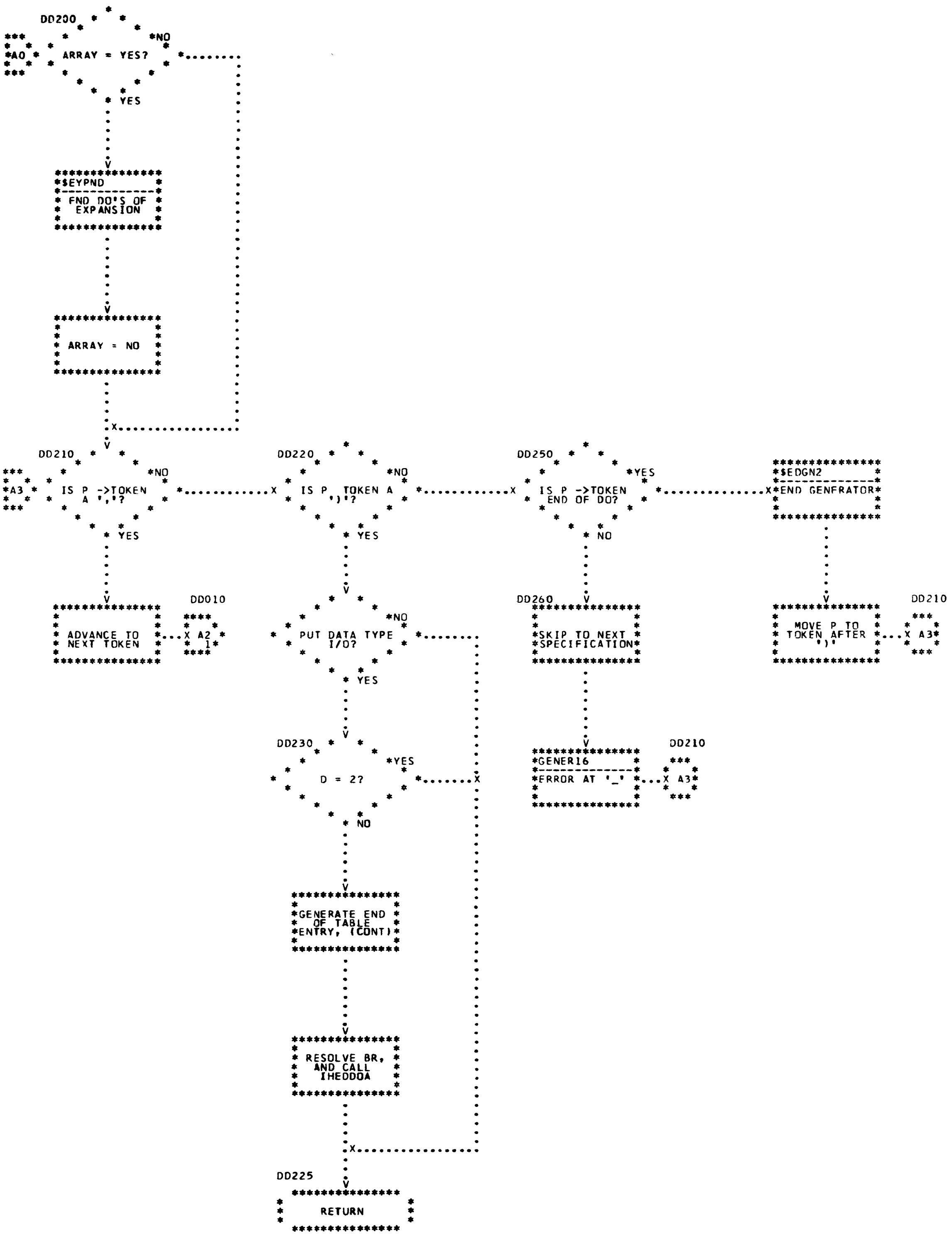


Chart 35. Data Specification (Page 5 of 6)

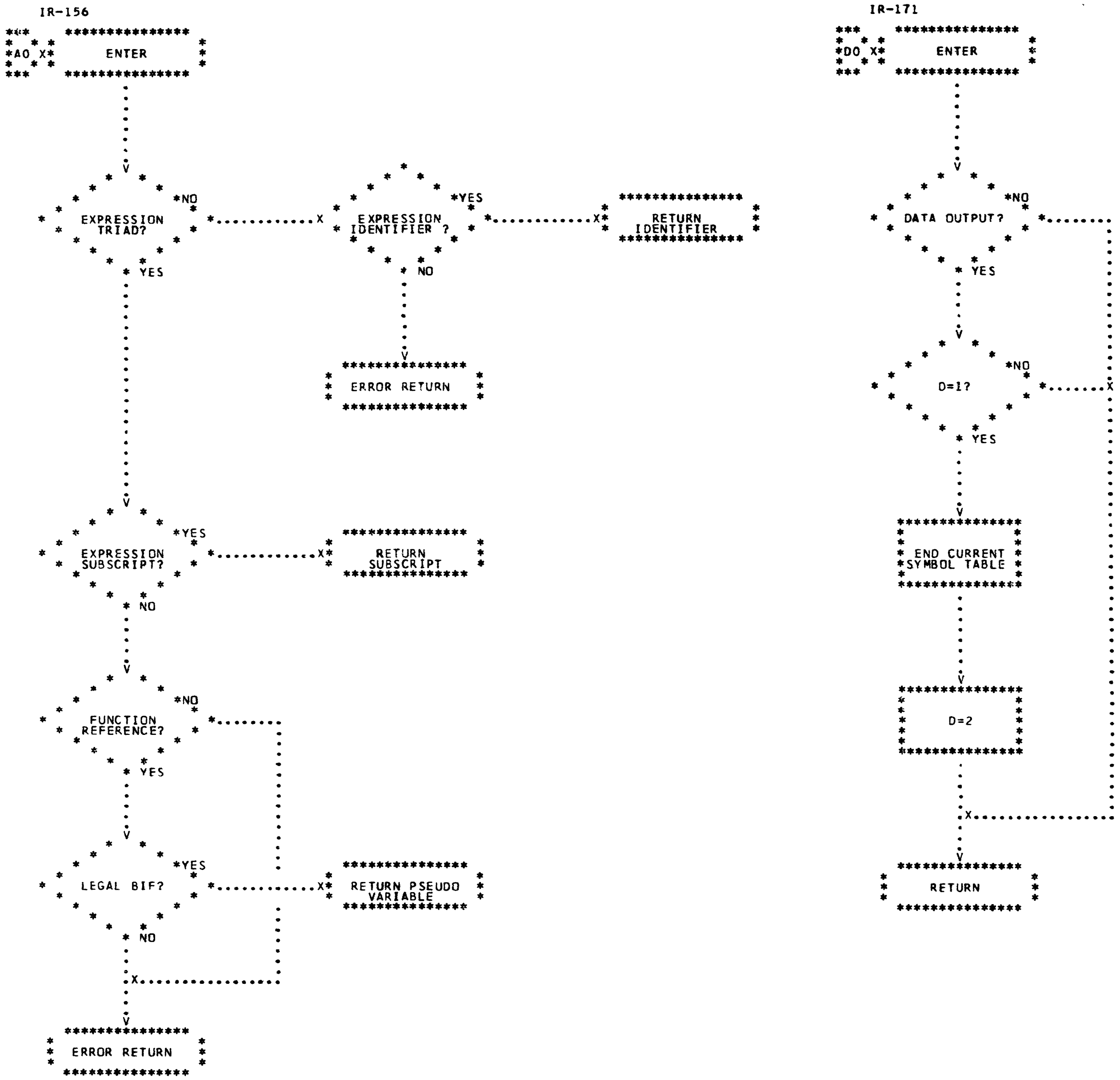


Chart 35. Data Specification (Page 6 of 6)

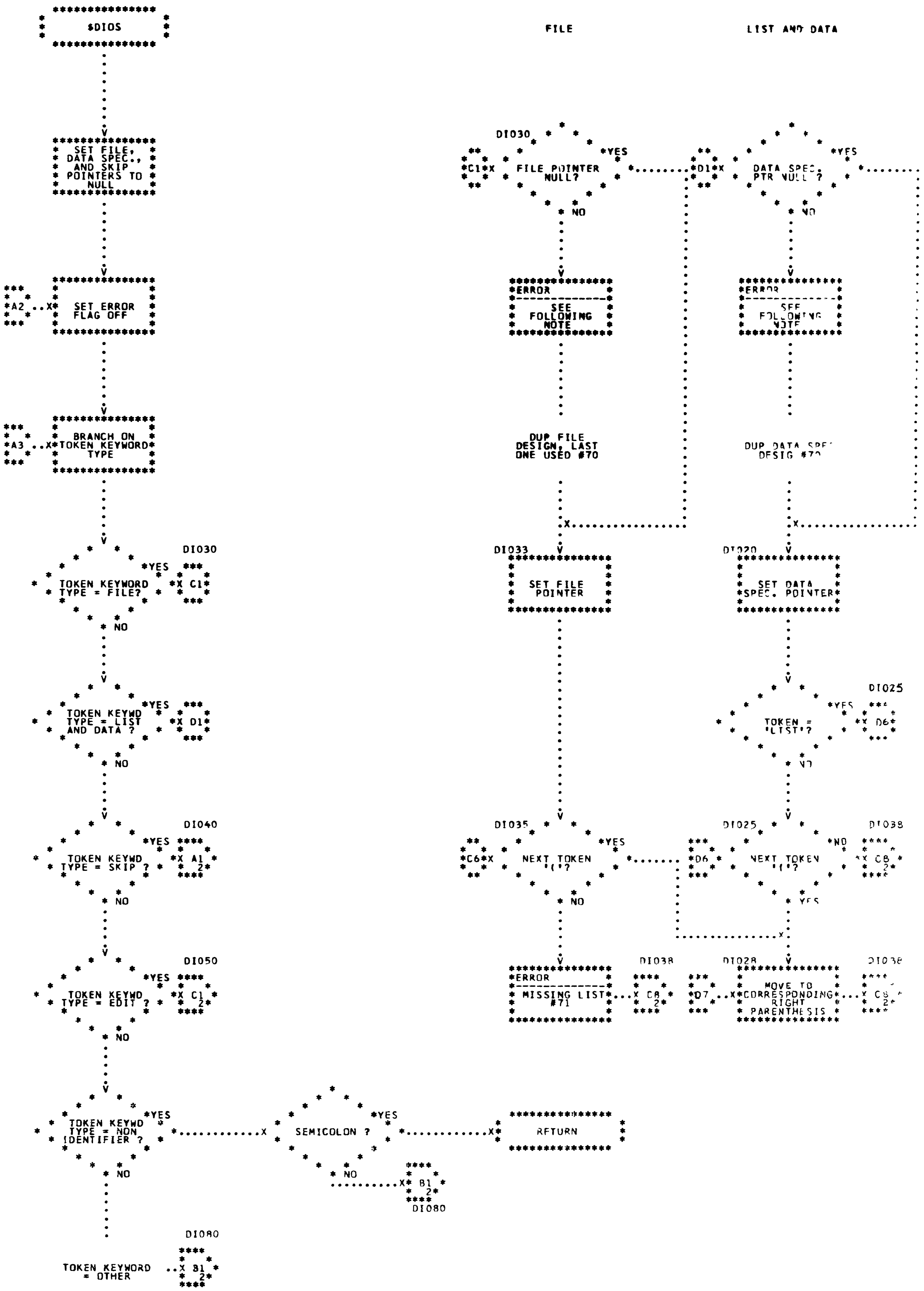


Chart 36. I/O Specification (Page 1 of 2)

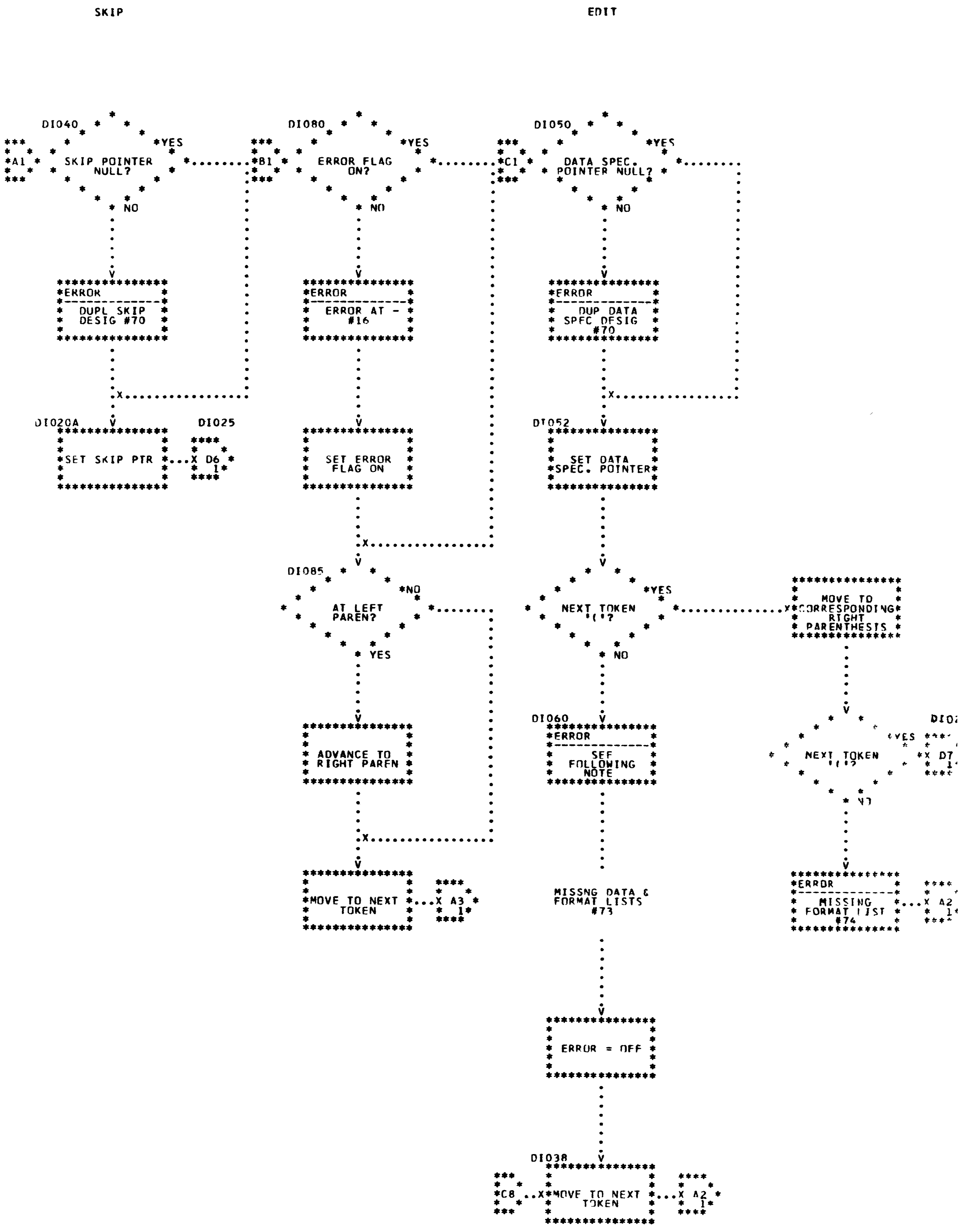


Chart 36. I/O Specification (Page 2 of 2)

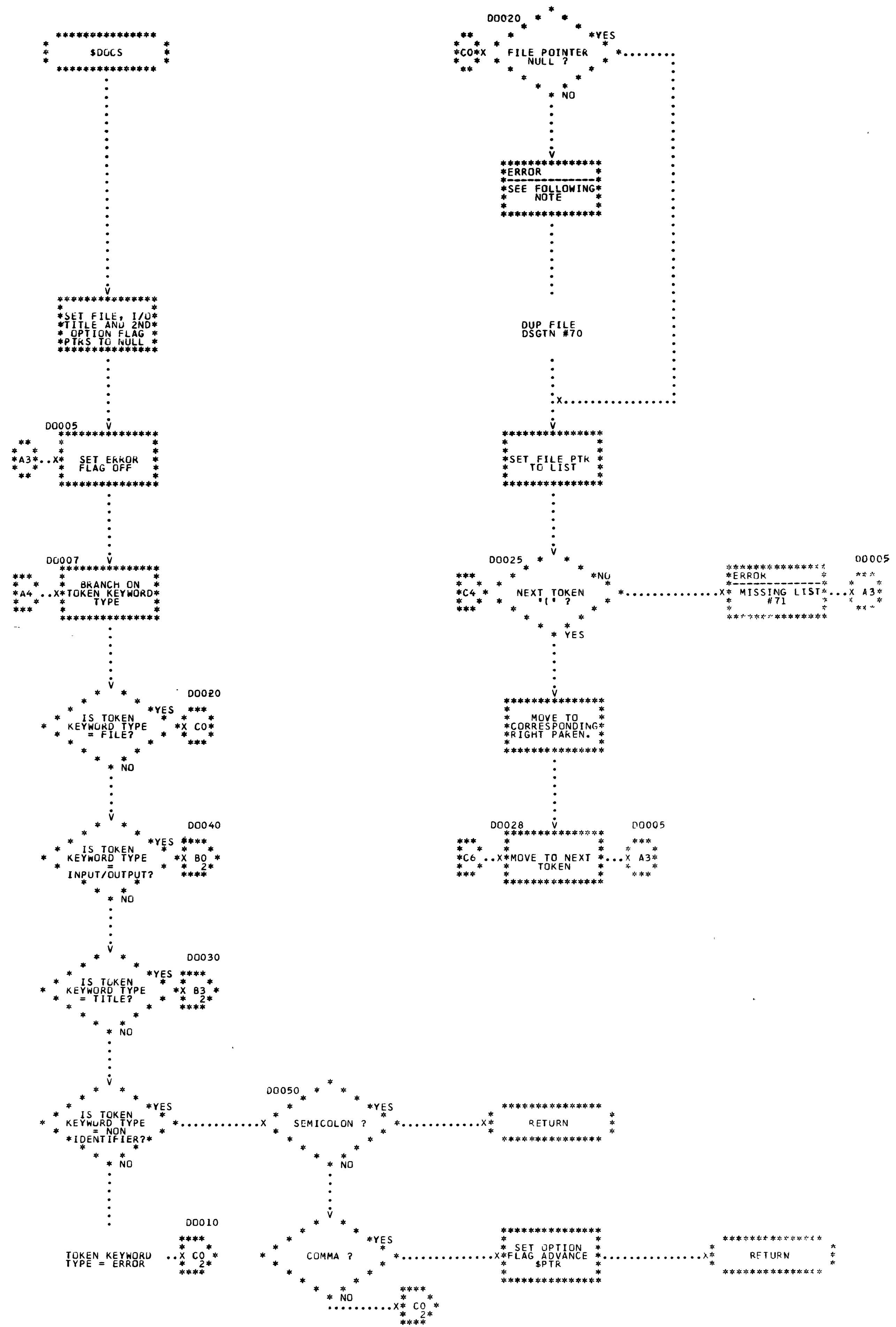


Chart 37. O/C Specification (Page 1 of 2)

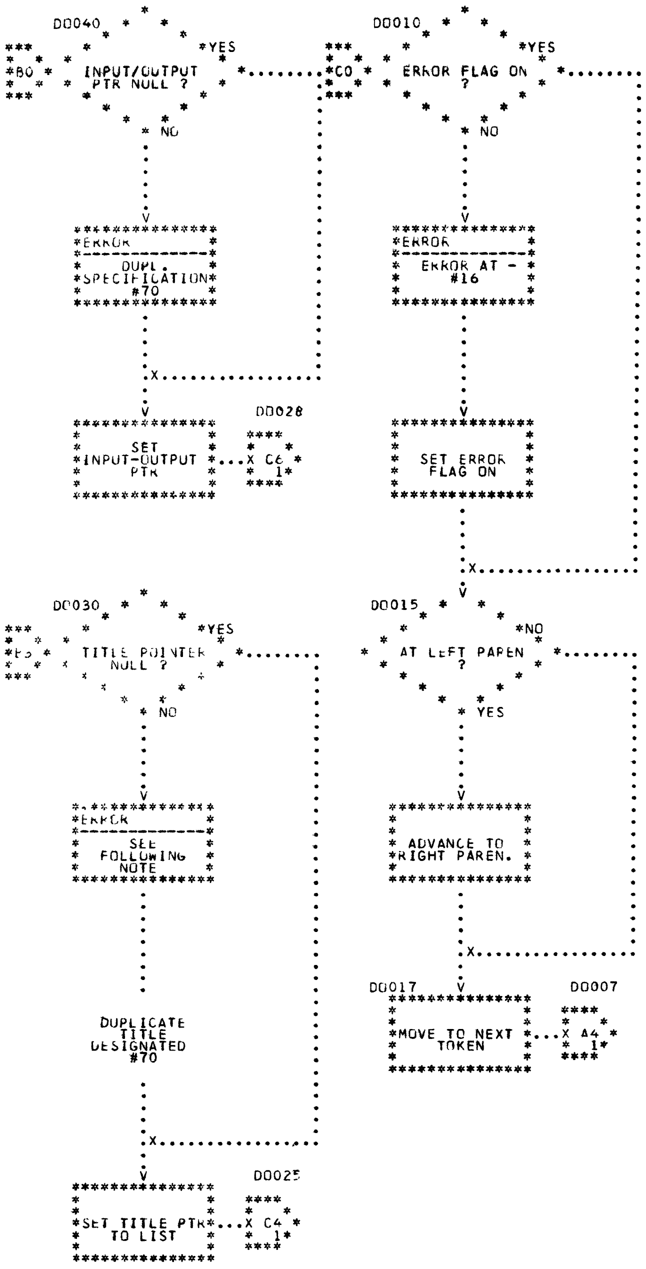


Chart 37. O/C Specification (Page 2 of 2)

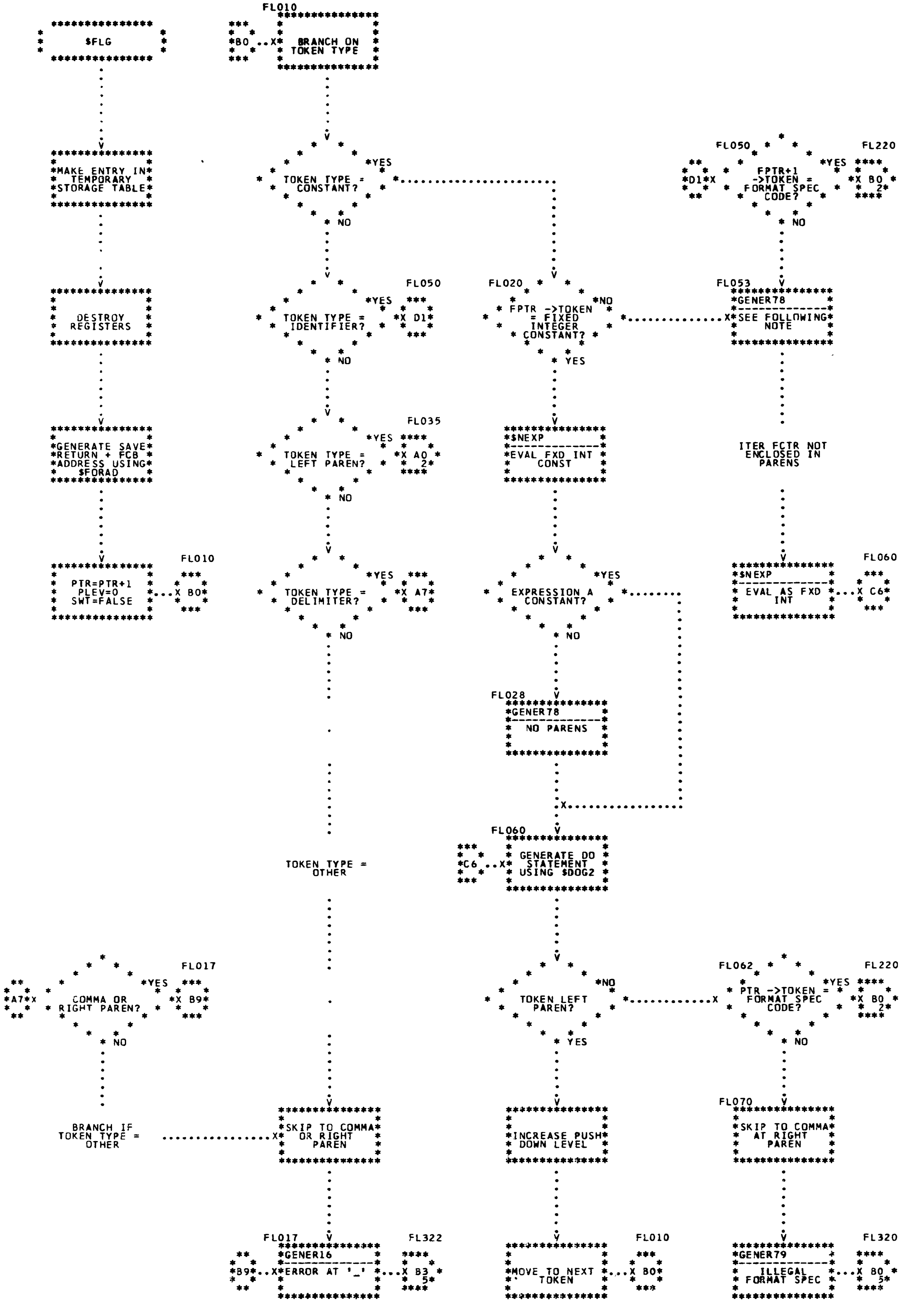


Chart 38. Format List Generator (Page 1 of 5)

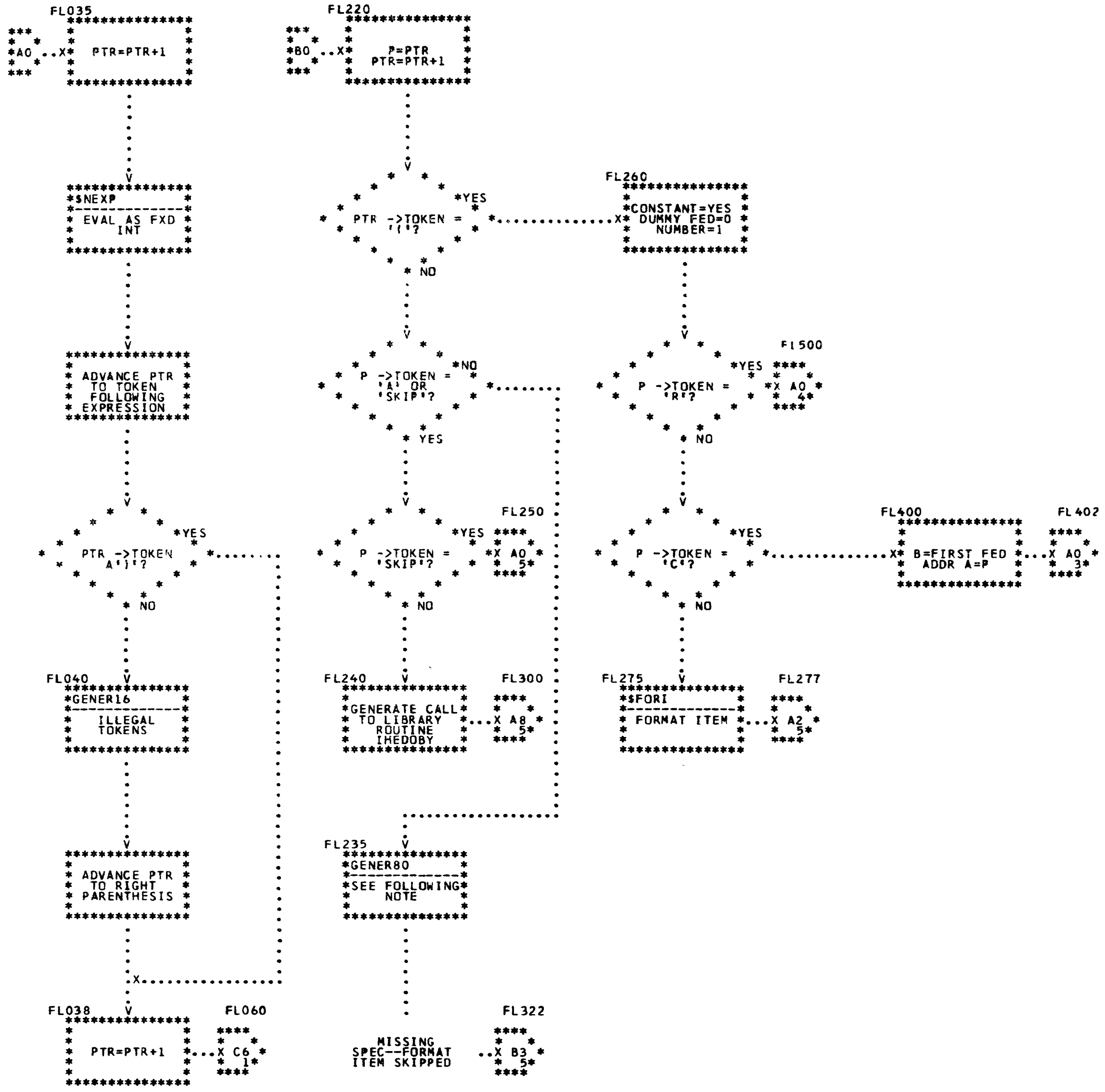
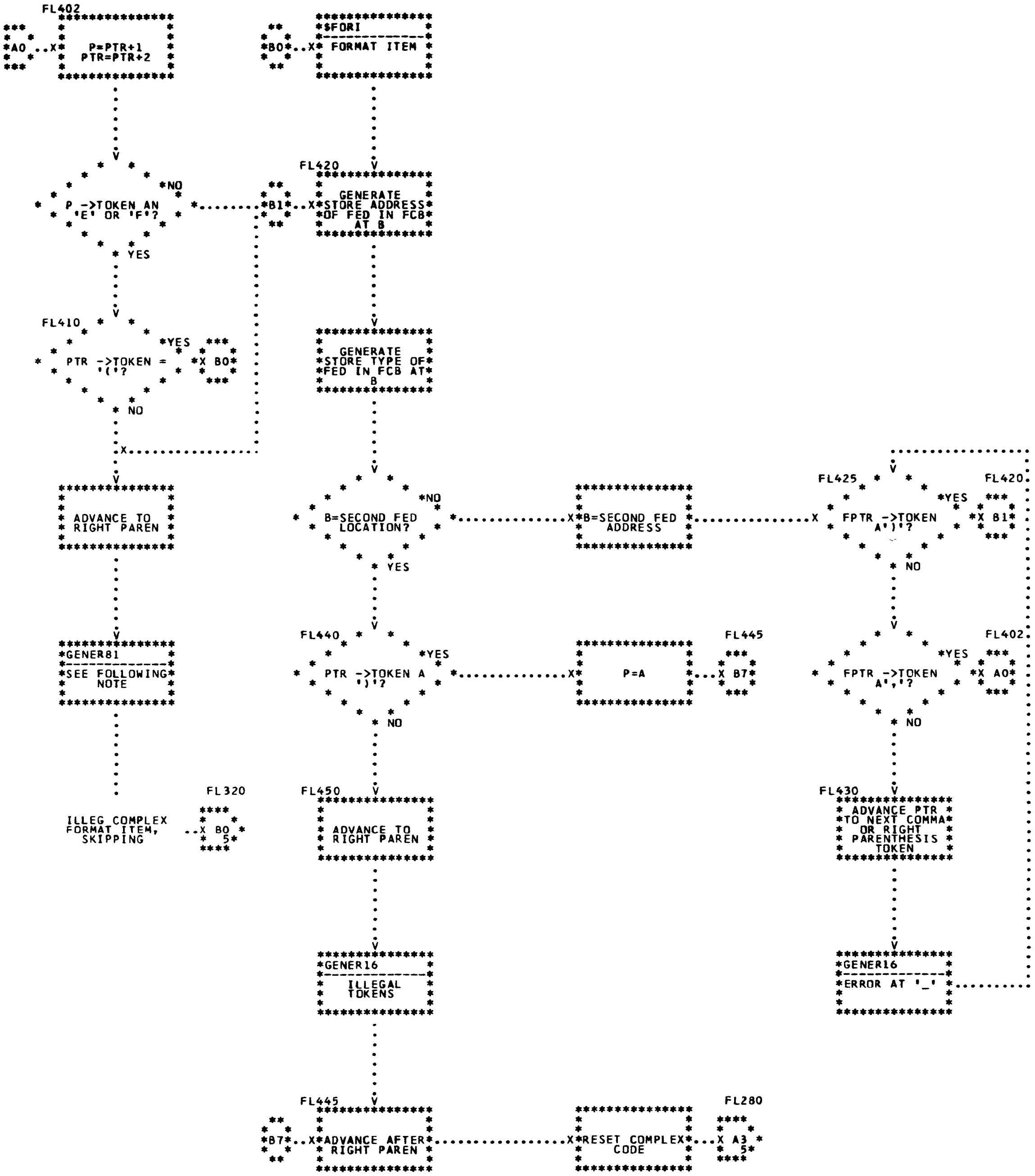


Chart 38. Format List Generator (Page 2 of 5)


FL402
A0
P=PTR+1
PTR=PTR+2
B0
$FORI
FORMAT ITEM
NO
P ->TOKEN AN 'E' OR 'F'?
YES
FL420
B1
GENERATE STORE ADDRESS OF FED IN FCB AT B
FL410
YES
PTR ->TOKEN = '('?
X B0
NO
GENERATE STORE TYPE OF FED IN FCB AT B
ADVANCE TO RIGHT PAREN
B=SECOND FED LOCATION?
NO
B=SECOND FED ADDRESS
FL425
FL420
YES
FPTR ->TOKEN A')'?
X B1
NO
YES
GENER81
SEE FOLLOWING NOTE
FL440
YES
PTR ->TOKEN A ')'?
NO
P=A
FL445
X B7
FL402
YES
FPTR ->TOKEN A','?
X A0
NO
ILLEG COMPLEX FORMAT ITEM, SKIPPING
FL320
X B0
5
FL450
ADVANCE TO RIGHT PAREN
FL430
ADVANCE PTR TO NEXT COMMA OR RIGHT PARENTHESIS TOKEN
GENER16
ILLEGAL TOKENS
GENER16
ERROR AT '_'
FL445
B7
ADVANCE AFTER RIGHT PAREN
RESET COMPLEX CODE
FL280
X A3
5

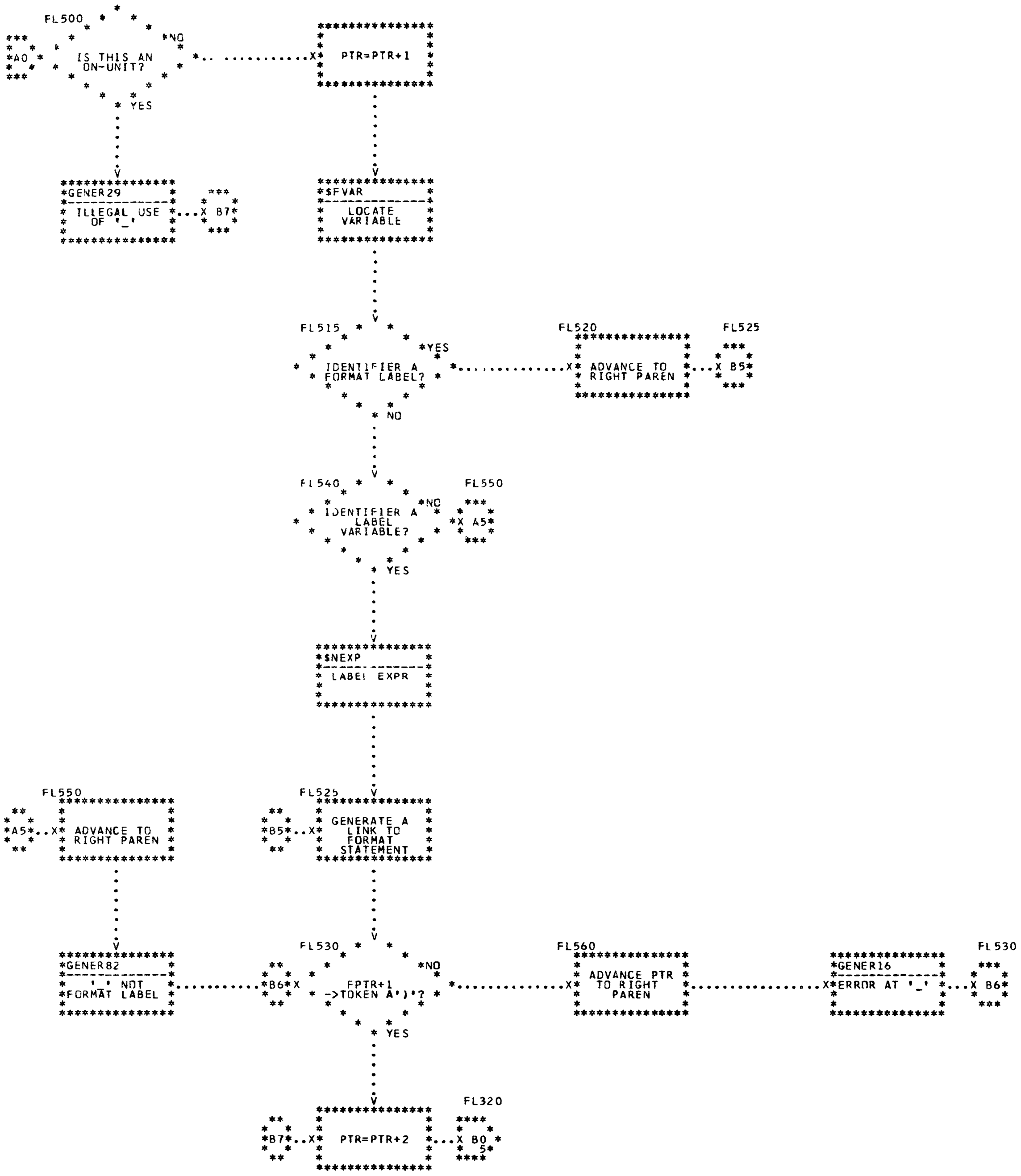


Chart 38. Format List Generator (Page 4 of 5)

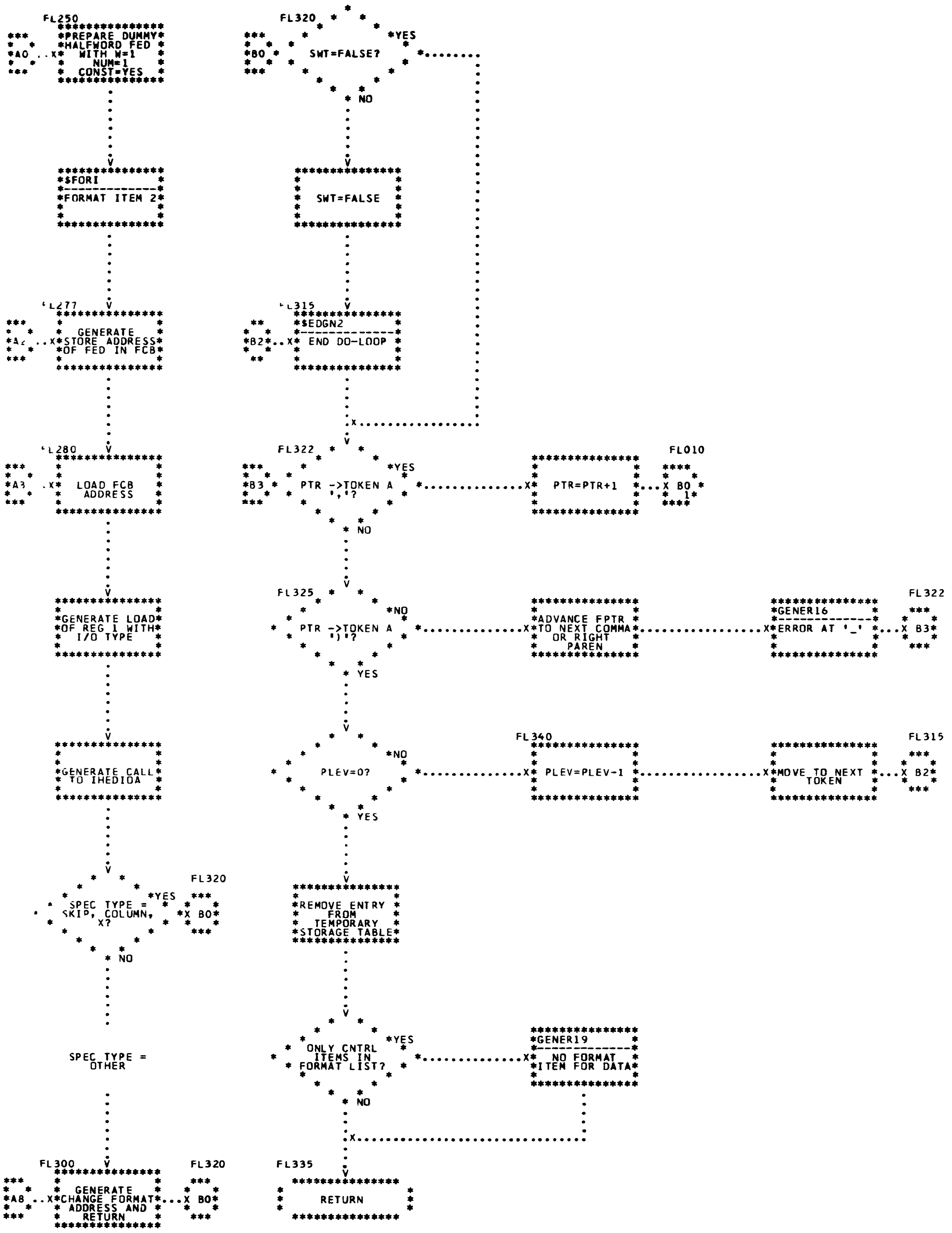


Chart 38. Format List Generator (Page 5 of 5)

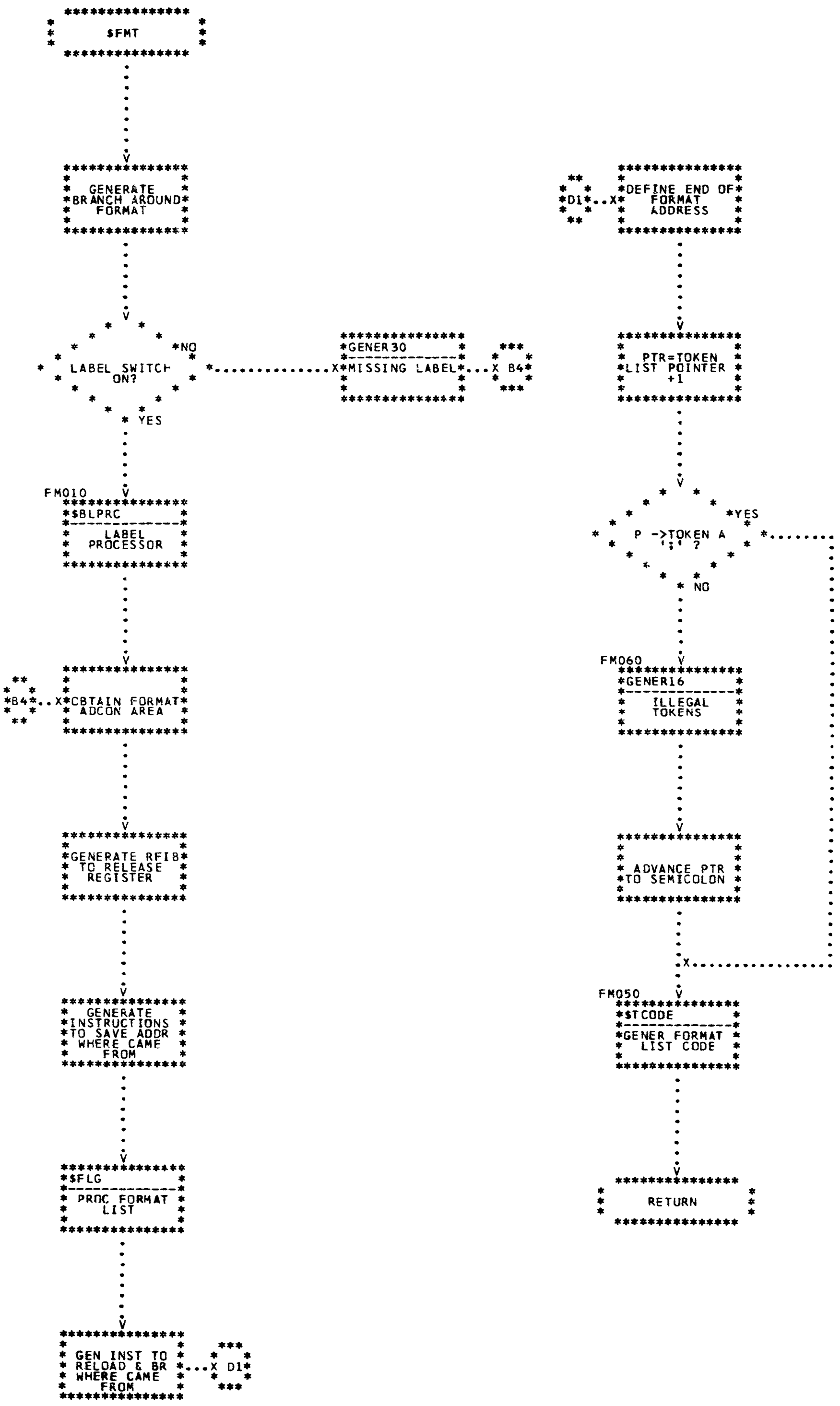


Chart 39. FORMAT Generator

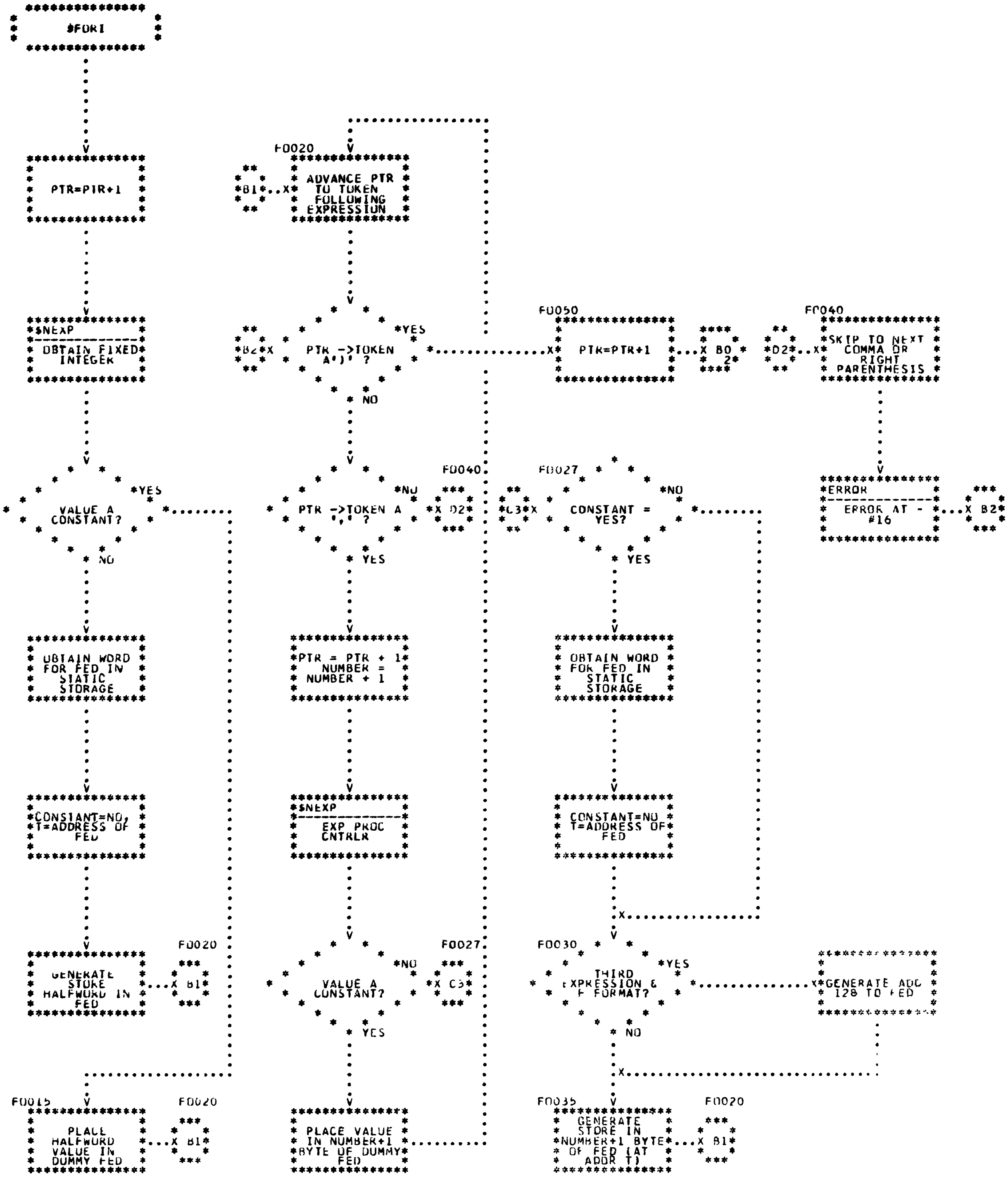


Chart 40. Format Item (Page 1 of 2)

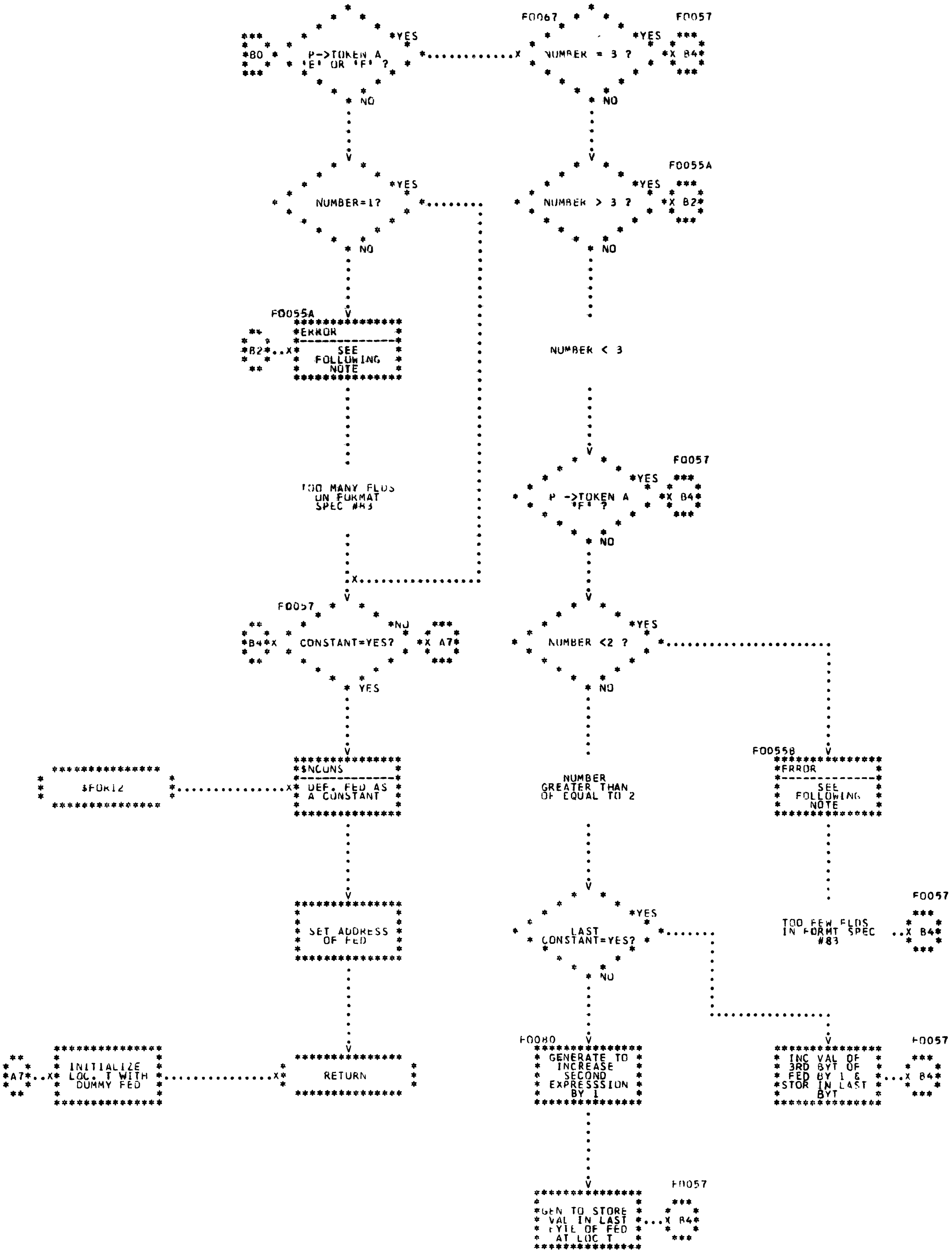


Chart 40. Format Item (Page 2 of 2)

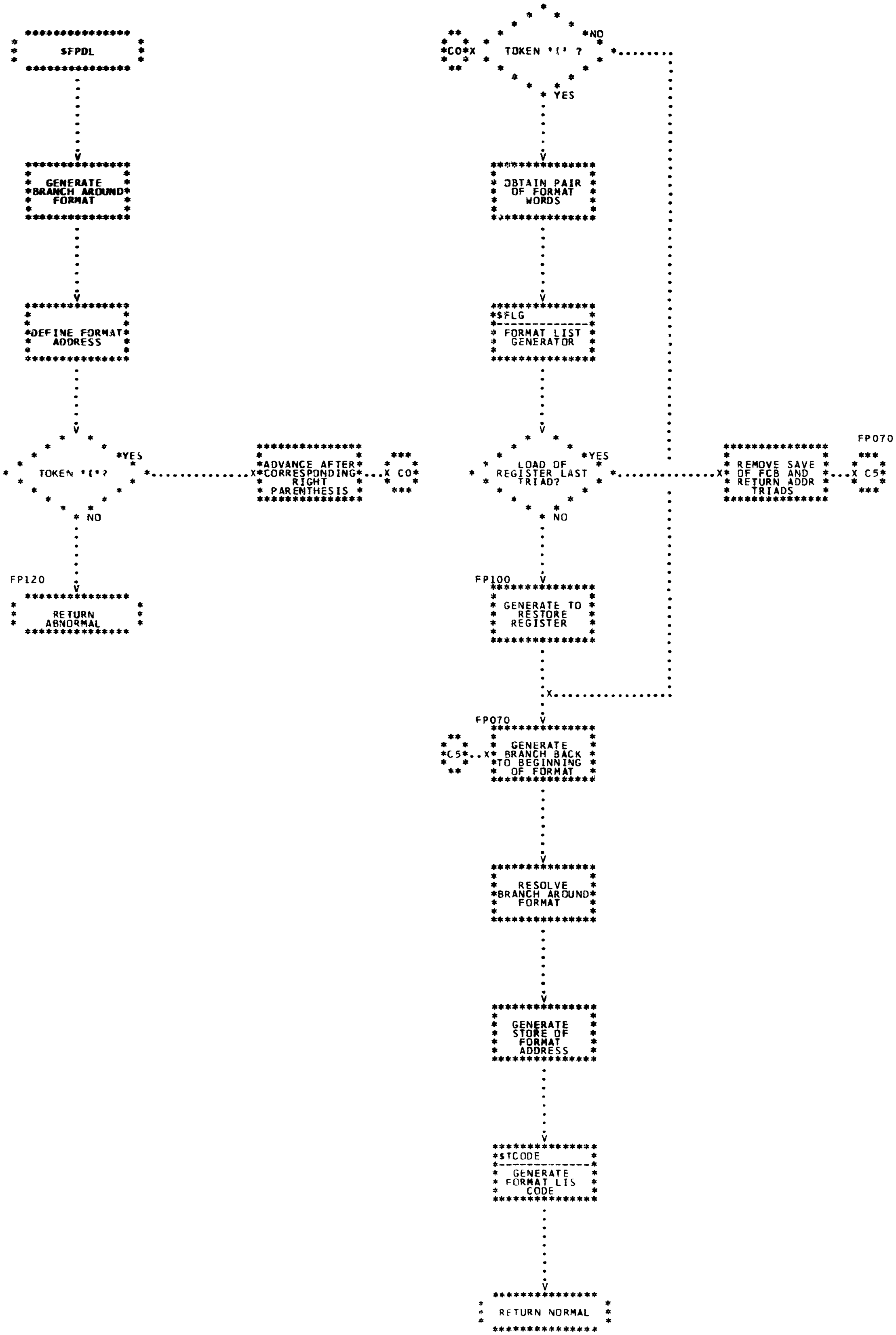


Chart 41. Format in Data List Processor

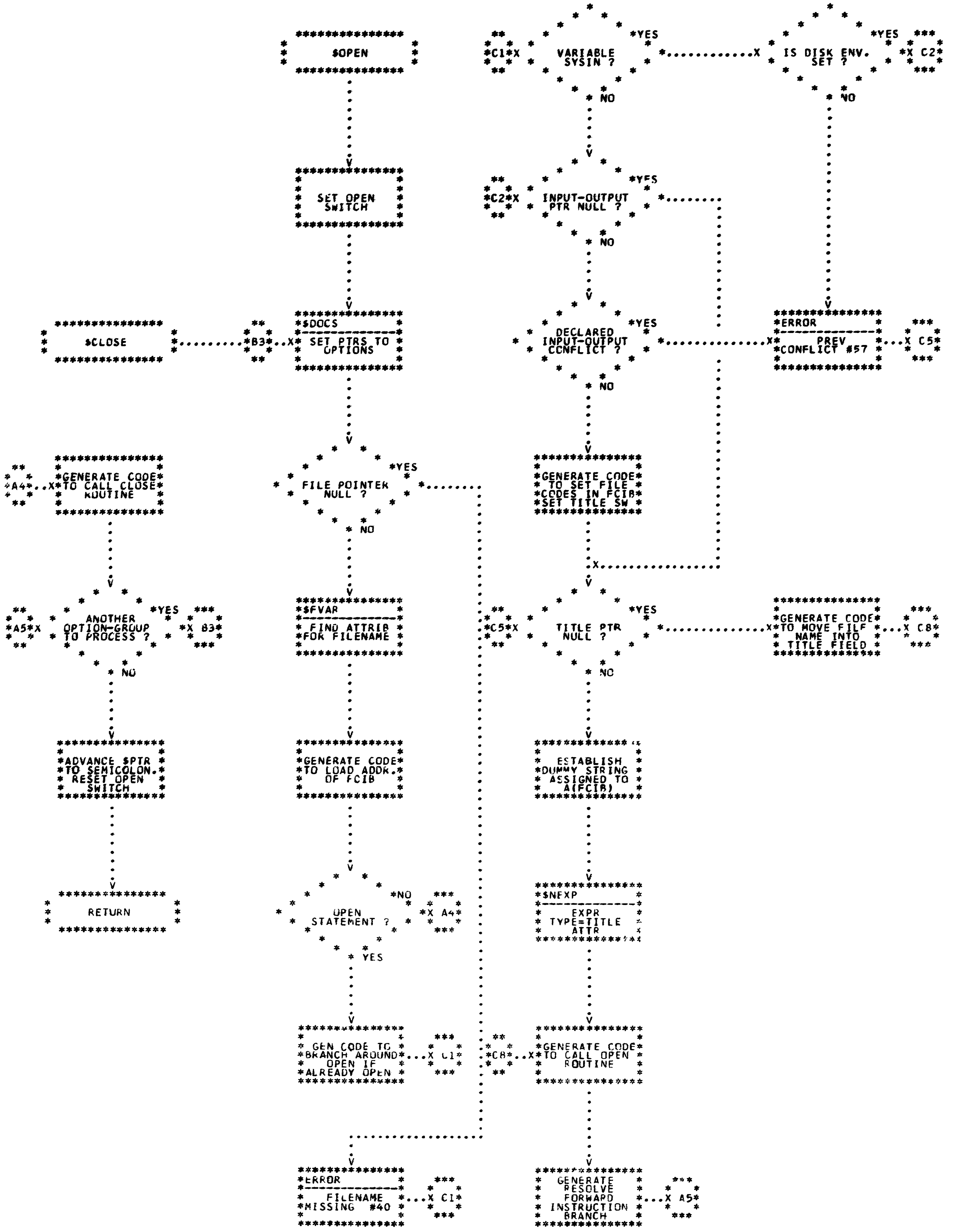


Chart 42. OPEN/CLOSE Generator

PART 6 - EXPRESSION PROCESSING

CALL/360-OS PL/I routines concerned primarily with expression processing are described in this subsection. The routines are discussed in alphabetic order, according to their mnemonics, as shown below. Detailed logic diagrams for the routines appear at the end of the subsection.

Argument Operand Processor ($NATTP)
Call Generator ($NCALL)
Cross-Section Dope Vector Build ($NCSDV)
Expression Processor Controller ($NEXP)
Dimension Multiplier Generator ($NMULT)
Convert Operand ($NOPCV)
Operator Stack Processor ($NOPRT)
Operand Set-Up ($NPRE)
Generate Triad ($TRIAD, $STRD)

TITLE: ARGUMENT OPERAND PROCESSOR ($NATTP)

Program Definition

Purpose and Usage

The Argument Operand Processor is used to obtain the attributes of an argument of a CALL or function reference.

Description

The argument list of a CALL or function reference is scanned twice. On the first scan, the highest type of the arguments and the number of array arguments are determined. This information is necessary in processing generic functions. On the second scan, the arguments are converted to the type required for the CALL. This routine is used to extract an argument's attributes in both scans. A flag indicates the scan number. During the first scan, a function and array argument are processed to substitute their return value attributes and array element attributes, respectively.

Errors Detected

None

Local Variables

None

Program Interface

Entry Points

$NATTP. Register G2 points to the entry of the operand stack which contains the argument token. The flag $NXFLG is set to zero for scan one and nonzero for scan two.

Exit Conditions

Return is to the calling routine immediately following the invoking call. The attributes of the argument are set in the left operand area.

Routines Called

None

Global Variable

$NAARG	Count of Number of Array Arguments
$NLOTM	Left Operand Type Mask
$NXFLG	Communication Flag
$NLOPR	Left Operand Arithmetic Precision
A List	Dictionary Attribute List
Y Table	Operand Stack

Logic Diagram

Chart 43 shows the detailed logic diagram for the Argument Operand Processor routine.

TITLE: CALL GENERATOR ($NCALL)

Program Definition

Purpose and Usage

The Call Generator generates the triads required to call a function or subprogram.

Description

A begin call triad is generated with the entry point and number of arguments as operands. An argument list triad is generated for each argument. If the entry point is to a fixed-point SUM, PROD, or POLY routine, a DED is generated for each argument. Each DED is passed as a separate argument immediately following the argument to which it applies. The operands for the arguments are obtained from the top entries of the operand stack; therefore, the arguments are placed in the argument list triads in the reverse order from which they appeared in the CALL or function reference. An end call triad is generated to terminate the argument list.

Errors Detected

None

Local Variables

NCNA	Left half is flag; if nonzero, the entry point is to a fixed-point SUM, PROD, or POLY routine. The right half contains the number of arguments in the CALL or function reference.
NCFN	Save area for registers P3 and P4

Program Interface

Entry Points

$NCALL. The number of arguments in the argument list is contained in register G0. The operands comprising the argument list are in the topmost entries of the operand stack, the last argument being at the top. Register P3 is the address of the operand area containing the description of the subprogram entry point.

Exit Conditions

Return is to the calling routine immediately following the invoking call. The entry point token has been replaced with a token referencing the end call triad.

Routines Called

$STRD	Generate Triad
$XERR	Error Message Editor
$WCTCT	Segment Management

Global Variables

$ERROR	Parameter List for Error Message Editor
$NOPI	Field of $NO
$NOTKN	Field of $NO
$NOTM	Field of $NO
$NTCUR	Number of Last Triad Generated
$NXOP	Stack Triad-Building Operand Area First Word

$NXTKN	Stack Triad-Building Operand Area Second Word
$NYOP	Stack Triad-Building Operand Area First Word
$NYTKN	Stack Triad-Building Operand Area Second Word
A List	Dictionary Attribute List
N List	Dictionary Name List
Y Table	Operand Stack

Logic Diagram

Chart 44 shows the detailed logic diagram for the Call Generator routine.

TITLE: CROSS-SECTION DOPE VECTOR BUILD ($NCSDV)

Program Definition

Purpose and Usage

The Cross-Section Dope Vector Build routine is invoked to process the subscript list of an array cross-section. The routine is invoked when the first subscript comprised of a single * is encountered in processing a subscript list within an argument list. For example, assume that the following subscript list is processed, with B having been previously defined as an array.

```
    INT(A,B(2,*,3),5)
```

The list of arguments of the function INT contains three arguments: A, B(2,*,3), and 5. The list of subscripts of the array B consists of three subscripts: 2, *, and 3.

All fields of the dope vector are constructed except the zero element address field and the number of dimensions field.

Description

The subscript list, starting immediately after the first * subscript, is scanned. The subscripts comprised of a single * token are counted. The dope vector is then allocated in static storage. The amount of storage required is a function of this count, the number of dimensions in the array cross-section.

Let the subscript list dimension positions containing a single * token be p1,p2,p3,...,pn. The data to be computed for each dimension of the array's cross-section dope vector is as follows:

cross-section lower bound-bound(i) = array lower bound p(i)

cross-section upper bound-bound(i) = array upper bound p(i)

cross-section multiplier-multiplier(i) = product of the array multiplier p(i) and all the multipliers on subscripts of the array between p(i) and p(i+1)

The subscript list is scanned a second time to generate the triads. The multipliers are computed and moved with the bounds into the cross-section dope vector.

If the array is for a string variable, triads are generated to move the length fields of the dope vectors to the cross-section dope vector.

Errors Detected

None

Local Variables

None

Program Interface

Entry Points

$NCSDV. Registers G2 and G4 point to the top of the operand and operator stacks, respectively. Register G3 points to the first token of the token table following the first * subscript of the cross-section reference.

Exit Conditions

Return is to the calling routine immediately following the invoking call. Register G3 is reset to its entrance value.

Routines Called

$GTRIAD	Get Next Triad Entry
$WEXP	Segment Management
$TRIAD, $STRD	Generate Triad
$WCTCT	Segment Management

Global Variables

$ASC	Offset to Next Available Byte in Static and Constants Area
$NLOPI	Expression Processing Results
$NLTKN	Expression Processing Results
$NROPN	Right Operand Area of Expression Processing
$NRPI	Field of $NRSLT
$NRTKN	Expression Processing Results
$NRTM	Field of $NRSLT
$NXTKN	Stack Triad-Building Operand Area Second Word
$NYOP	Stack Triad-Building Operand Area First Word
$NYPTR	Stack Triad-Building Operand Area Third Word
$NYTKN	Stack Triad-Building Operand Area Second Word
A List	Dictionary Attribute List
T Table	Token Table
V Table	Expression Stack
X Table	Operator Stack
Y Table	Operand Stack

Logic Diagram

Chart 45 shows the detailed logic diagram for the Cross-Section Dope Vector Build routine.

TITLE: EXPRESSION PROCESSOR CONTROLLER ($NEXP)

Program Definition

Purpose and Usage

The Expression Processor Controller controls the generation of triads to evaluate all expressions, the assignment statement, and the CALL statement entry name and argument list.

Description

An expression is composed of elements which are alternately operands and operators. The expression, to be properly formed, must start and terminate with an operand. An operand is an identifier, an identifier followed by a parenthesized list, an operand preceded by a prefix operator, or a parenthesized operand. A list is a series of expressions or single * tokens separated by commas. A prefix operator is a + or - token appearing in the expression at an operand position. An operator is an infix +, infix -, *, /, ↑ or **, comparison operator, &, |, or assignment symbol. Infix + and - are defined as a + or - token appearing in an operator position.

The tokens of the token table are scanned starting at the token set by the calling routine. Control remains in the Expression Processor Controller until an unexpected token appears at parenthesis level zero or an array assignment statement is recognized.

Each left parenthesis token encountered in this scan causes the expression stack to be pushed down. Processing is continued assuming the start of a new expression. The type of expression (that is, argument list, subscript list, array cross section, expression, assignment statement, or array assignment statement) and other information pertinent to its processing are retained in the stack.

All tokens recognized as identifiers are processed to determine the dictionary attribute list entry which applies to the scope of the reference. The pointer to this attribute entry and the pointer to the token in the token table are entered at the head of the operand stack. The + and - tokens identified as prefix operators are processed. The resultant effect is added to the operand stack entry to which it applies.

The comma, left and right parentheses, semicolon, and all operator tokens are processed to determine their effect on the operator stack. Each of these tokens, except the left parenthesis, contains a priority number which is equal to the priority of the operator or separator plus 256 times the parenthesis level of the token. The operator priority assignments are:

** or ↑	10
* and /	9
+ and -	8
compare	7
&	6
|	5
,	4
(	3
)	2
assignment symbol (=)	1

The left parenthesis token contains a pointer to its associated right parenthesis. The left parenthesis priority is obtained by adding 1 to the priority of its associated right parenthesis.

The priority of the operator under process is compared with the priority of the operator entry at the top of the operator stack. If less, triads required for the operator at the top of the operator stack are generated. Triads are also generated when the priorities are equal unless the operator under process is a ↑ , **, or , (comma) within an argument list. The two entries at the top of the operand stack are the operands of the operator. The two operands are removed from the operand stack. The operands are processed and triads generated to convert them, if necessary, to the type required to complete the operation. The operand types are determined by the operator and the data characteristics of the operands. Triads are then generated to perform the operation. A token describing the result of the operation is added to the top of the operand stack. The operator is removed from the top of the operator stack and the process repeated for the new top entry.

When the left parenthesis is removed from the operator stack as the result of processing the right parenthesis, the expression stack is popped up and the right parenthesis token discarded. The sign resulting from processing the prefix operators preceding the left parenthesis is added to the sign of the expression result. Processing continues assuming an operator is required next in the expression at the next outer parenthesis level.

An array assignment is detected when an assignment is under process and an array or a cross-section of an array is encountered at parenthesis level zero or is encountered as the first argument of a built-in function or pseudo-variable; control passes to the alternate exit of the Expression Processor Controller to signal this condition to the calling routine.

Errors Detected

'___' WHERE OPERAND EXPECTED. (86)
'___' WHERE OPERATOR EXPECTED. (87)
INCORRECT NO. OF SUBSCRIPTS FOR ARRAY. (96)
COMPILER ERROR. (100)
'___' ILLEGAL OPERATOR. (111)
IMPROPER RELATIONAL EXPRESSION. (112)

Local Variables

NEPLEV	Current parenthesis level. Set to zero on initial entrance to the routine. Increased by one for each left parenthesis encountered and decreased by one when the corresponding right parenthesis is encountered.
NESTK	Equated to the global variable $SNESTK which contains the address of the current expression stack entry.
NATT	Contains the dictionary attribute list entry offset of the last identifier encountered in the token stream.
NPREI	Prefix Operator Indicator. Three bits of this variable are flags:

Bit 0:	If 1, the operand is a parenthesized expression.
Bit 4:	If 1, the operand has had at least one prefix + or - operator.
Bit 5:	If 1, the operand has a prefix minus operator.

NEQP	Contains the pointer to the subscript substitution table for the processing of arrays in an array expression containing array cross-section references. This variable points to the token to be substituted for the current * token.

Program Interface

Entry Points

$NEXP. The variable $PTR contains the pointer to the token table entry of the first token of the expression to be processed. The top entry of the expression stack is set to the type of expression to be processed. The variable $NEXPT contains the attribute type mask describing the conversion to be applied to the expression value. If all bits of the mask are set, no conversion is performed.

Exit Conditions

Normal exit. Control returns to the caller at six bytes following the invoking call. $PTR is set to the token following the last token of the expression. If the result of the expression is a constant, the value is returned in register G0.

Array expression exit. Control returns to the caller immediately following the invoking call. The subscript number of the array at which the array expression was detected and the pointer to the attribute entry of the array are returned in $EXPNS. The variable $PTR contains the pointer to the token table entry at which the array expression was detected.

Routines Called

$FVAR	Locate Variable
$NCONS	Constant Processor
$NCSDV	Cross-Section Dope Vector Build
$NOPCV	Convert Operand
$NOPRT	Operator Stack Processor
$NPRE	Operand Set-Up
$TRIAD	Generate Triad
$WBACK	Segment Management
$WSTEP	Segment Management
$WEXP	Segment Management
$WCTCT	Segment Management
$XERR	Error Message Editor

Global Variables

$ERROR	Parameter List for Error Message Editor
$NPVF	Complex Pseudo-Variable Flag
$EXA	Used to Dump Fixed Table Area on Token Processed by $NEXP
$NROPN	Right Operand Area of Expression Processing
$EXPNS	Expression Processor to Array Expansion Routine Information
$NRPI	Field of $NRSLT
$GBQF	Table of Switches
$NRPR	Field of $NRSLT
$NRTKN	Expression Processing Results
$NEXRT	Statement Processor Expression Result Type
$NRTM	Field of $NRSLT
$NFLAG	Fixed-Point Scale Flag
$NXFLG	Communication Flag
$PRIOR	Operator Priority
$NLOPI	Expression Processing Results

$NLOPN	Left Operand Area of Expression Processing
$TAILS	Table of Pointers to End-of-Segment Control Word for Expandable Tables
$NLOTM	Left Operand Type Mask
$NLTKN	Expression Processing Results
$NIDSI	Identifier Search Indicator
$PTR	Token Table Pointer
A List	Dictionary Attribute List
C Table	Constant Table
V Table	Expression Stack
Q Table	Subscript Substitution Table
T Table	Token Table
Y Table	Operand Stack
X Table	Operator Stack

Logic Diagram

Chart 46 shows the detailed logic diagram for the Expression Processor Controller routine.

TITLE: DIMENSION MULTIPLIER GENERATOR ($NMULT)

Program Definition

Purpose and Usage

The Dimension Multiplier Generator routine generates the triad required to multiply a subscript value by the dimension multiplier associated with the current array dimension.

Description

The array name and current dimension position are obtained from the top entry of the expression stack. If the array is a parameter, the right operand of the multiply operator is the address of the location of the array dope vector and the number of the dimension position. If the array is not a parameter, this operand is the actual address of the halfword dimension multiplier in the array dope vector. The left operand is the value of the computed subscript.

Errors Detected

None

Local Variables

None

Program Interface

Entry Points

$NMULT. The left operand area contains the description of the subscript value. The top entry of the expression stack contains the entry for the subscript list under process.

Routines Called

$STRD	Generate Triad

Global Variables

$NESTK	Address of Top of Expression Stack
$NXOP	Stack Triad-Building Operand Area First Word
$NXTKN	Stack Triad-Building Operand Area Second Word
$NXPTR	Stack Triad-Building Operand Area Third Word
$NLOPN	Left Operand Area of Expression Processing
V Table	Expression Stack
A List	Dictionary Attribute List

Logic Diagram

Chart 47 shows the detailed logic diagram for the Dimension Multiplier Generator routine.

TITLE: CONVERT OPERAND ($NOPCV)

Program Definition

Purpose and Usage

The Convert Operand routine is used to convert an operand to any type required. The operand will also be scaled to any scale, if desired. If the operand is a source constant and the conversion is arithmetic to arithmetic, the attribute entry of this source constant is changed to indicate the required type and scale. If the operand is not such a source constant, any triads required to achieve the desired type and scale are generated. If the operand cannot be converted to the required type, an error message is generated.

Description

The operand to be converted is described in an operand entry whose format is:

<table>
<tr><th></th><th>1</th><th>2</th><th>3</th><th>4</th><th>Byte</th></tr>
<tr><td rowspan="2">Word 1</td><td rowspan="2"></td><td rowspan="2">$NOTM</td><td colspan="2">$NOL</td></tr>
<tr><td>$NOPR</td><td>$NOSF</td></tr>
<tr><td>2</td><td colspan="4">$NOTKN</td></tr>
<tr><td>3</td><td>$NOPI</td><td colspan="3"></td></tr>
</table>

$NOTM = operand type flags
$NOL = operand length if a string
$NOPR and $NOSF = operand precision & scale if arithmetic
$NOTKN = operand token
$NOPI = sign and other flags

The address of the first word of this entry is passed to this routine. The desired type, precision, and scale of the converted operand are set before entrance to this routine in $NRTM, $NRPR, and $NRSF, respectively. A result type of X'FF' indicates no conversion is to be performed on the operand. If the operand is to be scaled, $NFLAG must be nonzero; no scaling of it is performed if $NFLAG is zero.

The type of the operand is compared with the desired type. If they are alike or if the desired type is X'FF', no conversion is performed. If a conversion to the desired type is not possible, an error message is generated. If an arithmetic to arithmetic conversion of a source constant operand is required, its attribute entry is changed to indicate the desired attributes. The constant will be converted directly to this type by the Constant Processor routine ($NCONS). In all other cases where a conversion is required, a convert triad or series of triads to call the string to arithmetic or arithmetic to string library conversion routine (IHEDCN or IHEDNC) is generated.

When the scaling flag is nonzero and the operand and desired types are both fixed, a multiply, divide, scale positive complex, or scale negative complex triad is generated to change the scale of a non-source constant operand to the desired scale; if the operand is a source constant, the desired scale is set in its attribute entry and the constant will be scaled to this value when converted by $NCONS.

Errors Detected

OPERAND INCOMPATIBLE WITH REQUIRED ATTRIBUTES. (105)

Local Variables

NOTM	The operand and result type masks EXCLUSIVE OR's together.
NOSGN	Contains the operation required to scale the operand (that is, multiply, divide, scale positive complex, or scale negative complex).

Program Interface

Entry Points

$NOPCV. Register P3 is the address of the first word of the operand area describing the operand to be converted. Registers G2, G3, and G4 contain pointers to operand stack, token table, and operator stack respectively.

Exit Conditions

Control returns to the calling routine immediately following the invoking call. No specific output values are returned.

Routines Called

$NCONS	Constant Processor
$STRD	Generate Triad
$WEXP	Segment Management
$WCTCT	Segment Management
$NCALL	Call Generator
$XERR	Error Message Editor

Global Variables

$NLFLAG	Fixed-Point Scale Flag
$NLOPN	Left Operand Area of Expression Processing
$NOPI	Field of $NO
$NOPR	Field of $NO
$NOPTR	Field of $NO
$NOSF	Field of $NO
$NOTKN	Field of $NO
$NOTM	Field of $NO
$NRL	Field of $NRSLT
$NROPN	Right Operand Area of Expression Processing
$NRPR	Field of $NRSLT
$NRSF	Field of $NRSLT
$NRSLT	Operation Result Attributes and Description Area of Expression Processing
$NRTM	Field of $NRSLT
$NXFLG	Communication Flag
$NXOP	Stack Triad-Building Operand Area First Word
$NXTKN	Stack Triad-Building Operand Area Second Word
$NYOP	Stack Triad-Building Operand Area First Word
A List	Dictionary Attribute List
N List	Dictionary Name List
Z Table	Triad Table

Logic Diagram

Chart 48 shows the detailed logic diagram for the Convert Operand routine.

TITLE: OPERATOR STACK PROCESSOR ($NOPRT)

Program Definition

Purpose and Usage

The Operator Stack Processor is used to process, in order, all operators in the operator stack whose priority is greater than or equal to the operator pending addition to the stack. The triads required to encode the operator and its operands are produced.

Note: The comma in an argument list and the exponentiation operators are exceptions. The comma is processed when the right parenthesis token is encountered; the exponent operator is processed when an operator of lower priority is encountered.

Description

The operator type at the top of the operator stack is used to branch to the section of code which controls the processing of that operator. These sections are: infix +, infix -, multiply, divide, exponentiation, compare operations, assignment symbol, comma, and parenthesis.

The tokens defining the operands of the operator are located at the top of the operand stack. The resultant type of all prefix operators influencing an operand is contained in the stack entry. Scale and conversion triads are formed, when required, to adjust the operands such that they are compatible with each other and their associated operator. The triad is then generated for the operator and its two operands. The two operands are removed from the operand stack and replaced with a token describing the result (that is, triad number, type, scale, and precision). The operator stack is popped-up and the process repeated for the operator at the head of the stack, if its priority is greater than or equal to the priority of the operator pending posting to the stack.

Errors Detected

ARRAY EXPRESSION ILLEGAL. (23)
ILLEGAL USE OF '___'. (29)
ILLEGAL SCALE FACTOR FOR '___'--IGNORED. (64)
ILLEGAL OPERAND FOR OPERATOR '___'. (89)
OPERAND OF '___' MUST BE BIT STRING. (91)
ILLEGAL ASSIGNMENT. (93)
INCORRECT NUMBER OF ARGUMENTS IN SUBPROGRAM. (94)
ARGUMENT '___' INCOMPATIBLE WITH PARAMETER IN SUBPROGRAM CALL. (95)
INCORRECT NO. OF SUBSCRIPTS FOR ARRAY. (96)
COMPILER ERROR. (100)
OPERAND INCOMPATIBLE WITH REQUIRED ATTRIBUTES. (105)

Local Variables

NXPARM	Used in processing argument list. Contains pointer to current parameter attribute entry if parameter attributes are declared.
NXBINF	If nonzero, a built-in function is under process.
NXARGC	Argument counter. Used to count arguments when processing an argument list.
NXARGN	Number of arguments in function reference
NXHTYP	Highest type of all arguments in an argument list
NXPQ	Largest precision minus scale of any argument of the list
NXRS	Precision minus scale of last argument of the list
NXNAME	Entry point number of library routine

Program Interface

Entry Points

$NOPRT. The priority of the operator pending posting to the operator stack is in $PRIOR. Register G2 points to the top of the operand stack; register G4 to the top of the operator stack.

Exit Conditions

Normal exit. Control returns to the caller four bytes following the invoking call.

Parenthesis return. When parentheses are closed off (that is, a left parenthesis operator removed from the stack), control returns immediately following the invoking call.

Routines Called

$NATTP	Argument Operand Processor
$NCALL	Call Generator
$NCONS	Constant Processor
$NLSIB	Library Search
$NMULT	Dimension Multiplier Generator
$NOPCV	Convert Operand
$NPRE	Operand Set-up
$TRIAD, $STRD	Generate Triad
$WBACK	Segment Management
$WCTCT	Segment Management
$WEXP	Segment Management
$WSTEP	Segment Management
$XERR	Error Message Editor

Global Variables

$DISPL	Displacement from Variable Tables Address to Fixed Tables
$ERROR	Parameter List for Error Message Editor
$NAARG	Count of Number of Array Arguments
$NCPXP	Save Area for Arguments and Right-Hand Side of Complex Pseudo-Variable
$NESTK	Address of Current Expression Stack Entry
$NFLAG	Fixed-Point Scale Flag
$NIDSI	Locate Variable Type
$NLOL	Left Operand String Length
$NLOPI	Expression Processing Result
$NLOPN	Left Operand Area of Expression Processing
$NLOPR	Left Operand Arithmetic Precision
$NLOSF	Left Operand Arithmetic Scale Factor
$NLOTM	Left Operand Type Mask
$NLTKN	Expression Processing Result
$NPVF	Complex Pseudo-Variable Flag
$NRL	Field of $NRSLT
$NROL	Right Operand String Length
$NROPI	Expression Processing Result
$NROPN	Right Operand Area of Expression Processing
$NROPR	Right Operand Arithmetic Precision
$NROSF	Right Operand Arithmetic Scale Factor
$NROTM	Right Operand Type Mask
$NRPI	Field of $NRSLT
$NRPR	Field of $NRSLT
$NRSF	Field of $NRSLT
$NRSLT	Operation Result Attributes and Description Area of Expression Processing
$NRTKN	Expression Processing Result

$NRTM	Field of $NRSLT
$NTCUR	Number of Last Triad Generated
$NRKN	Field of $NRSLT
$NXFLG	Communication Flag
$NXPTR	Stack Triad-Building Operand Area Third Word
$NXTKN	Stack Triad-Building Operand Area Second Word
$NYTKN	Stack Triad-Building Operand Area Second Word
$NYTM	Stack Triad-Building Operand Area Type Mask
$PRIOR	Operator Priority
$PTR	Token Table Pointer
$TAILS	Table of Pointers to End-of-Segment Control Word for Expandable Tables
A List	Dictionary Attribute List
C Table	Constant Table
N List	Dictionary Name List
Z Table	Triad Table
V Table	Expression Stack
Y Table	Operand Stack
X Table	Operator Stack

Logic Diagram

Chart 49 shows the detailed logic diagram for the Operator Stack Processor routine.

TITLE: OPERAND SET-UP ($NPRE)

Program Definition

Purpose and Usage

This routine is used by the Expression Processor Controller to move the top entry of the operand stack to an operand area and to extract its type, sign, scale, precision, and length attributes.

Description

The top entry of the operand stack is moved to the indicated operand area. If the operand is a dictionary attribute pointer, the attributes are obtained from this entry. In all other cases the attributes are obtained from the stack entry itself. If the operand is a parameterless function, triads are generated for this call. This result replaces the operand. If the operand is signed and of CHARACTER type, triads are generated to convert it to a FIXED integer. When $NXFLG is nonzero and the operand is a string, the attributes FIXED REAL (9,0) are substituted for the operand's attributes; the operand is flagged for conversion by the Convert Operand routine ($NOPCV).

Errors Detected

USE OF '___' HERE CONFLICTS WITH PREVIOUS USAGE. (57)
INCORRECT NO. OF ARGUMENTS IN SUBPROGRAM. (94)

Local Variables

NPSGN	Used to save sign of operand when conversion from string to arithmetic type required.

Program Interface

Entry Points

$NPRE. Register P4 contains the address of the operand area to be set up, G2 the operand stack pointer, G3 the token table pointer, and G4 the operator stack pointer.

Exit Conditions: Return is to the calling program following point of invocation.

Routines Called

$WCTCT	Segment Management
$NOPCV	Convert Operand
$NCALL	Call Generator
$XERR	Error Message Editor

Global Variables

$ERROR	Parameter List for Error Message Editor
$NFLAG	Fixed-Point Scale Flag
$NOPTR	Field of $NO
$NOL	Field of $NO
$NOPI	Field of $NO
$NOTKN	Field of $NO
$NOTM	Field of $NO
A List	Dictionary Attribute List
Y Table	Operand Stack

Logic Diagram: Chart 50 shows the detailed logic diagram for the Operand Set-Up routine.

TITLE: GENERATE TRIAD ($TRIAD, $STRD)

Program Definition

Purpose and Usage

The Generate Triad routine is invoked to generate a single entry in the triad table (Z table). Space for the entry is obtained and all fields of the triad entry completed.

Description

The operator and location of the two operands to be placed into the triad entry may be specified on entrance. If not specified, the operator from the top of the operator stack, the operand descriptions in the left and right operand areas, and the result type field are used.

If an operand is a constant, the value of the converted constant is searched in the constant table (C table) and the operand replaced with a token pointing to this constant table entry. If an operand is a triad reference, the last-use field of the triad it references is set to the number of the triad currently being generated.

Errors Detected

None

Local Variables

TRSGN	Sign to be placed on the generated triad
TRZPTR	Address of the current triad table entry
TRZP	Pointer to the current triad table entry (relative to the origin of the user's area)

Program Interface

Entry Points

$STRD. At entrance $STRD, register G0 contains the triad operator positioned in the leftmost byte and the type of the triad result in the rightmost byte. Registers P3 and P4 contain the addresses of the operand areas containing the descriptions of the left and right operands, respectively. The sign of the triad result is set plus.

$TRIAD. At entrance $TRIAD, the operator is assumed to be sitting at the top of the operator stack which is pointed to by register G4. The left and right operand descriptions are assumed set in $NLOPN and $NROPN. The result type and sign are contained in $NRTM and $NRPI.

Exit Conditions

Return is to the calling routine immediately following the invoking call. Register G7 contains an operand token which points to the generated triad.

Routines Called

$WEXP	Segment Management
$NCONS	Constant Processor

Global Variables

$NLOPN	Left Operand Area of Expression Processing
$NOPI	Field of $NO
$NOTKN	Field of $NO
$NROPN	Right Operand Area of Expression Processing
$NRTM	Field of $NRSLT
$NTCUR	Number of Last Triad Generated
$NXFLG	Communication Flag
A List	Dictionary Attribute List
Y Table	Operator Stack
Z Table	Triad Table

Logic Diagram

Chart 51 shows the detailed logic diagram for the Generate Triad routine.

PART 6 LOGIC DIAGRAMS

The detailed logic diagrams for the routines concerned primarily with expression processing follow.

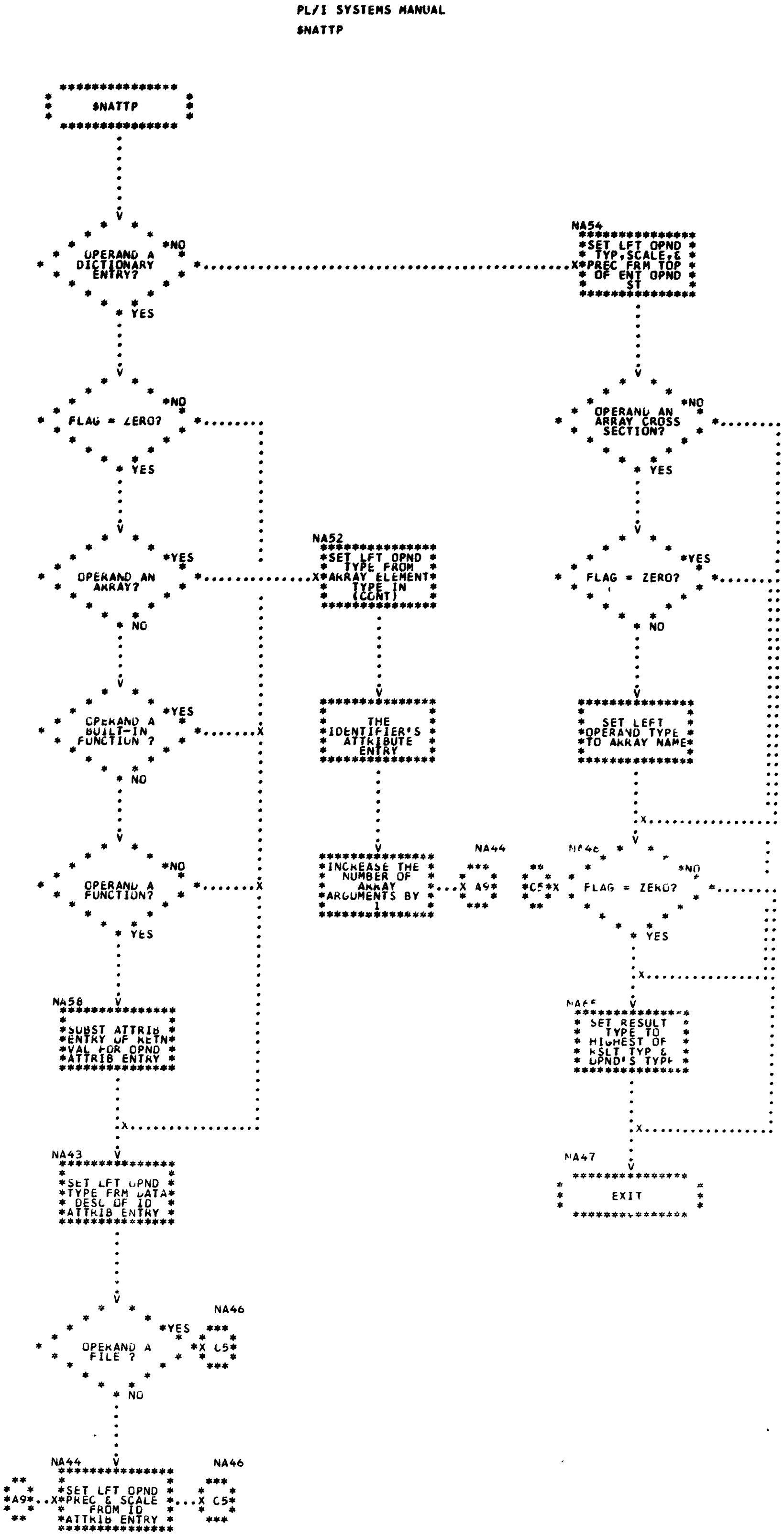


Chart 43. Argument Operand Processor

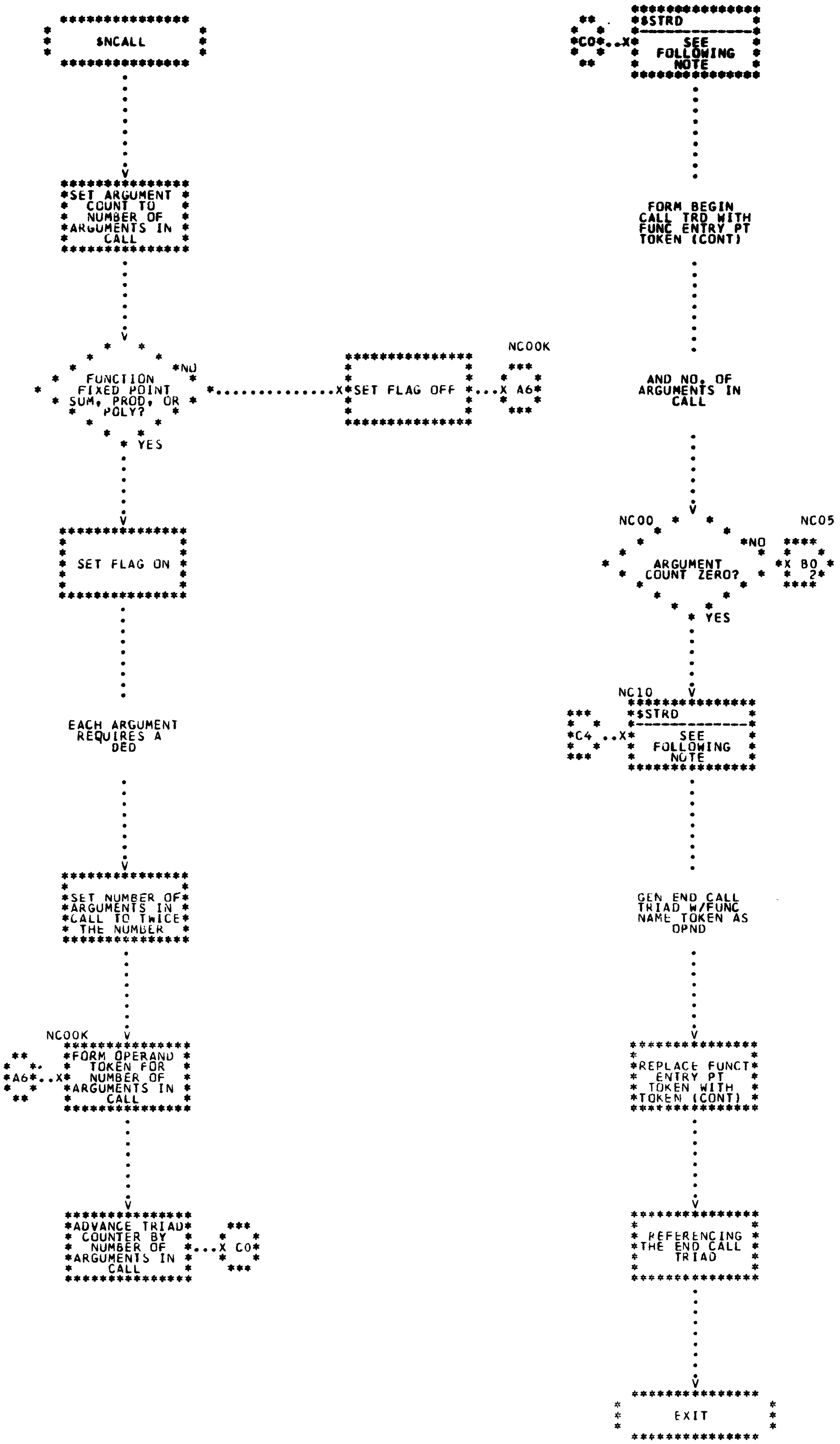


Chart 44. Call Generator (Page 1 of 2)

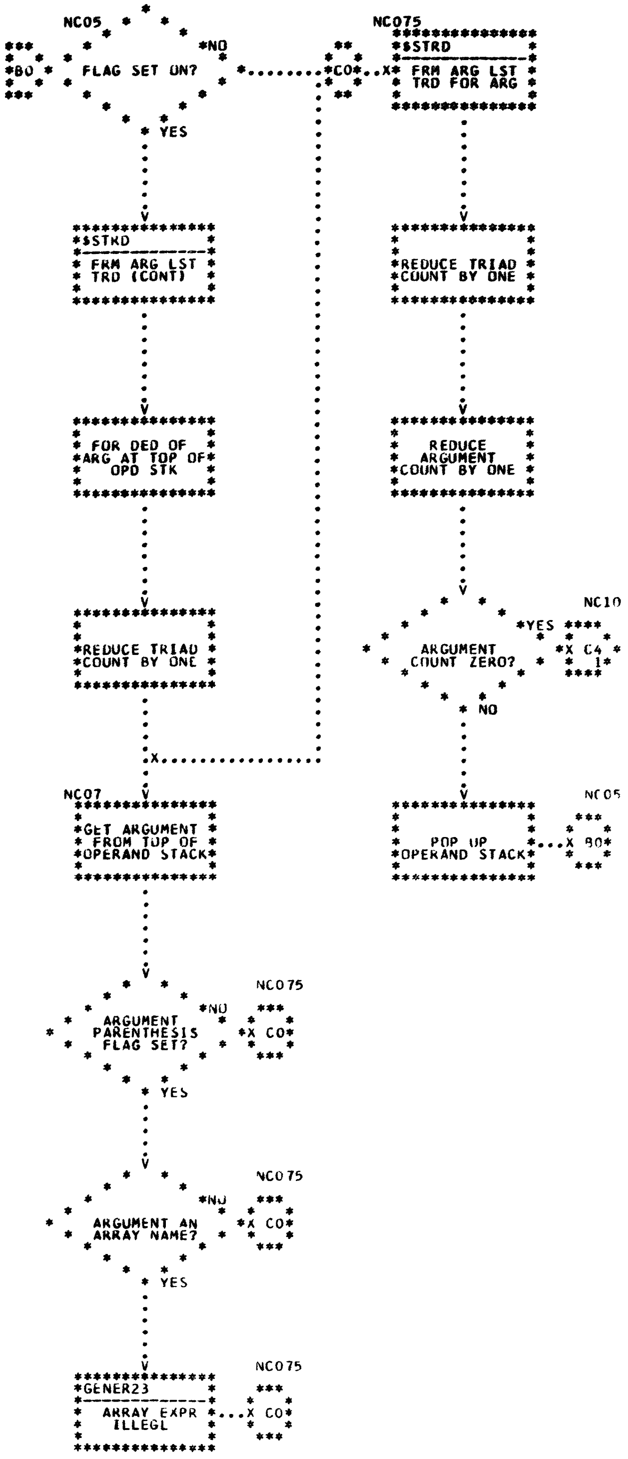


Chart 44. Call Generator (Page 2 of 2)

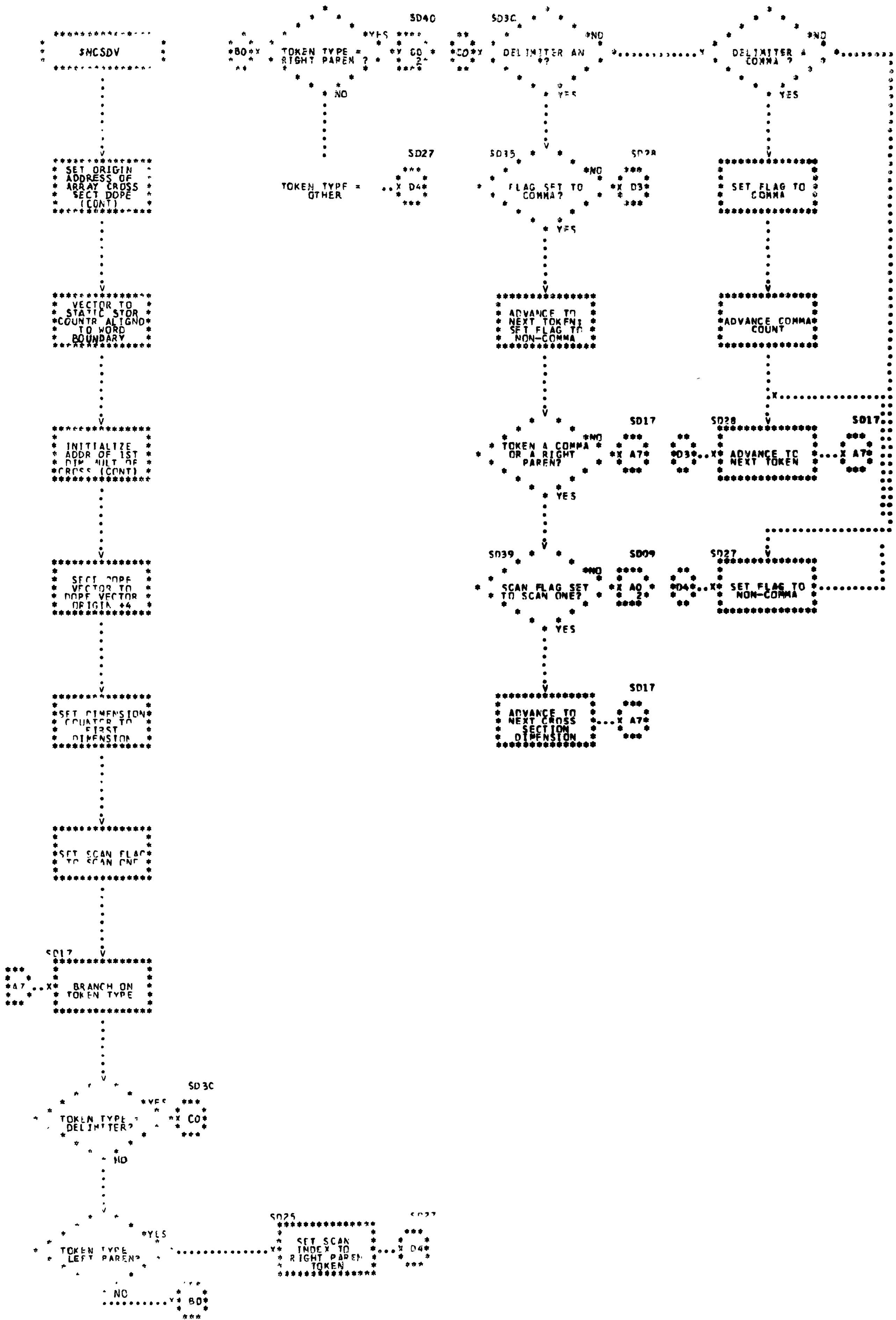


Chart 45. Cross Section Dope Vector Build (Page 1 of 2)

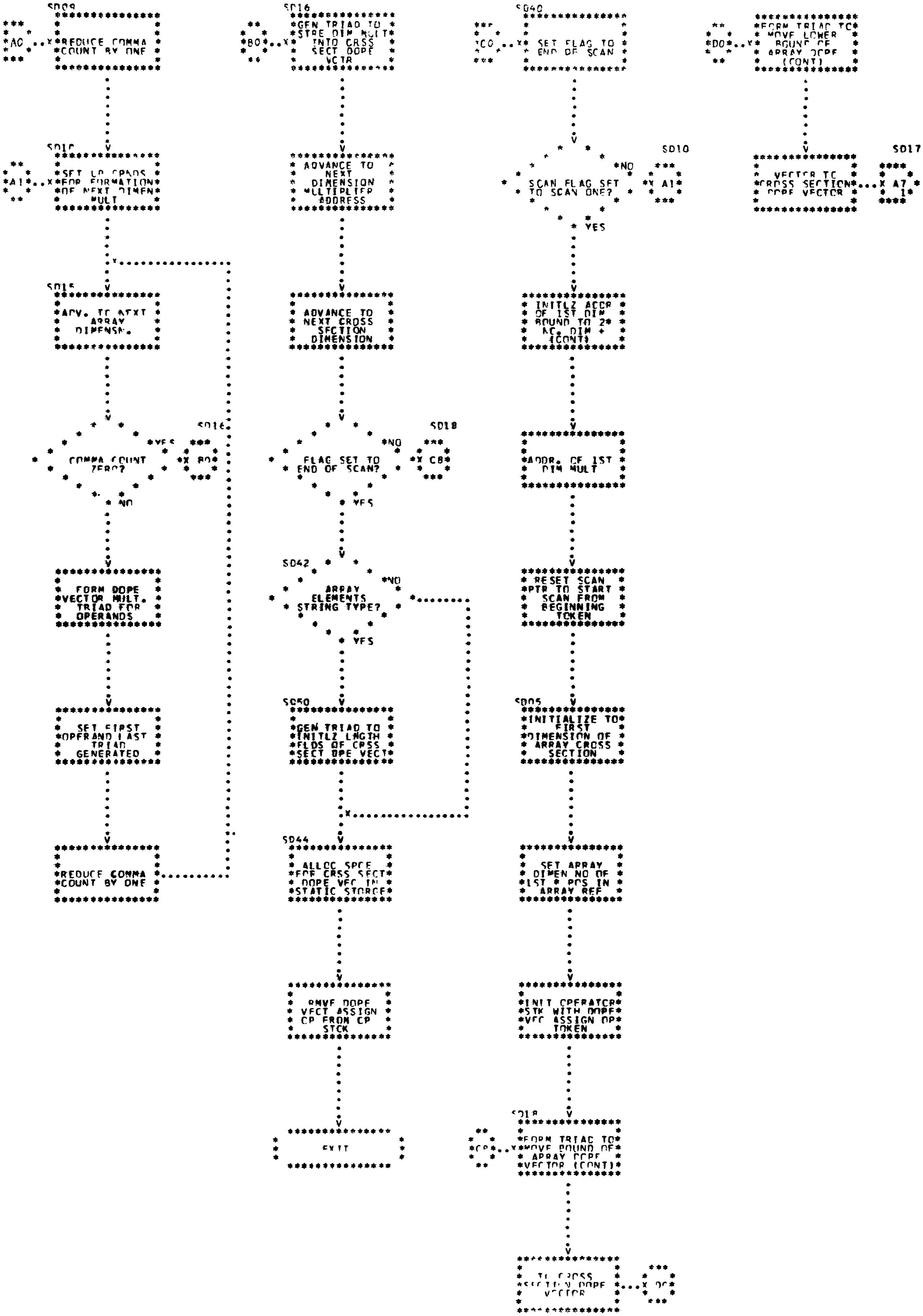


Chart 45. Cross Section Dope Vector Build (Page 2 of 2)

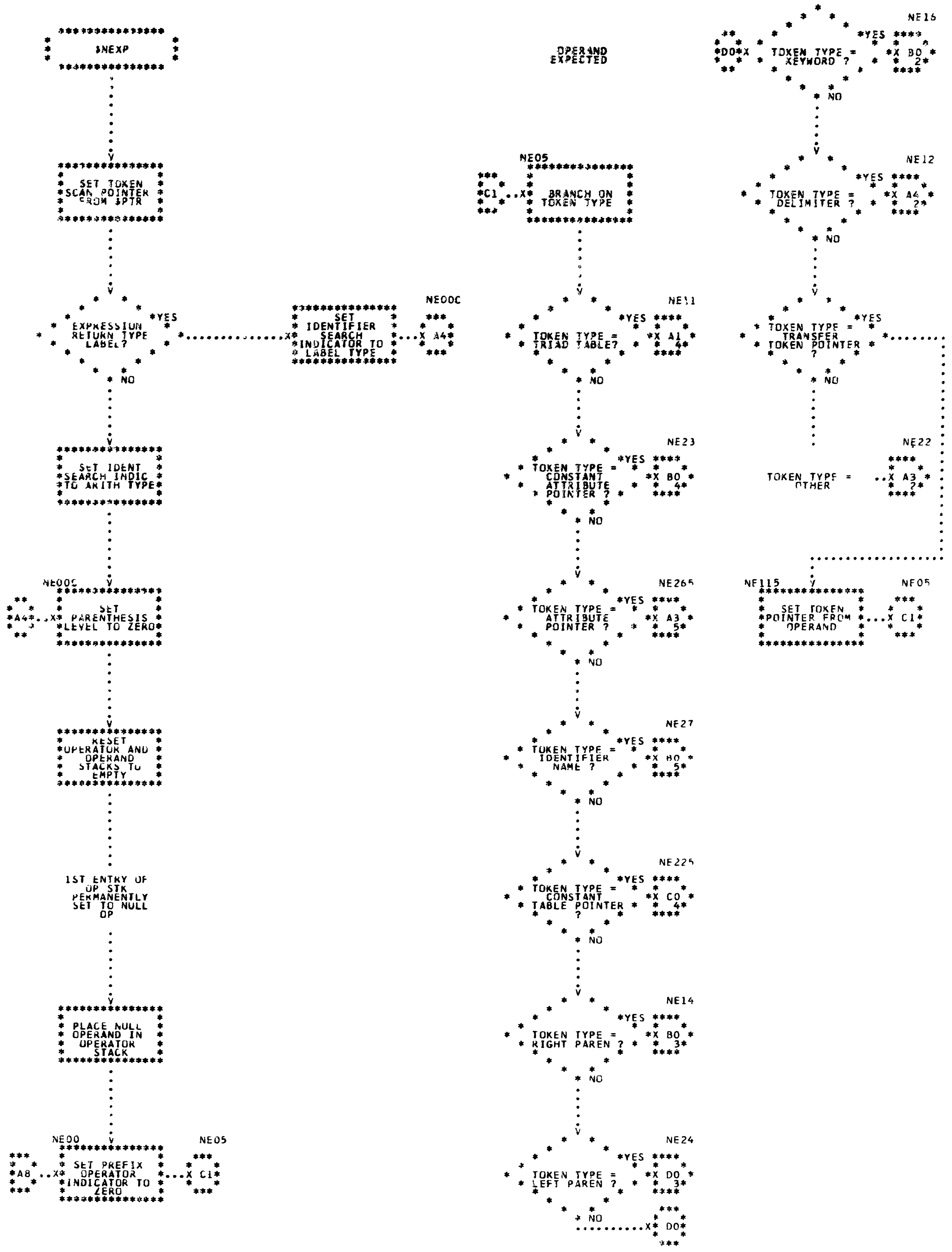


Chart 46. Expression Processor Controller (Page 1 of 11)

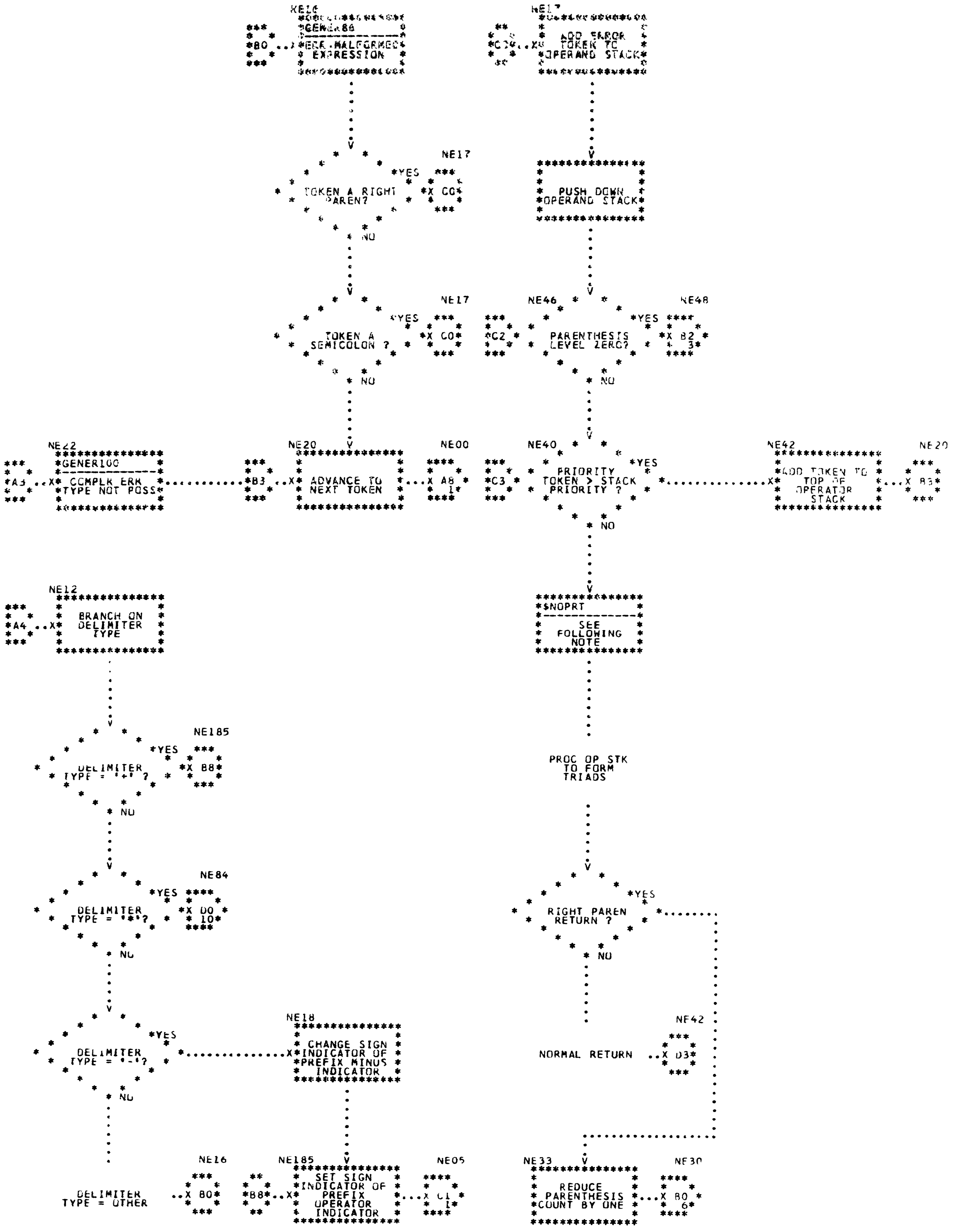


Chart 46. Expression Processor Controller (Page 2 of 11)

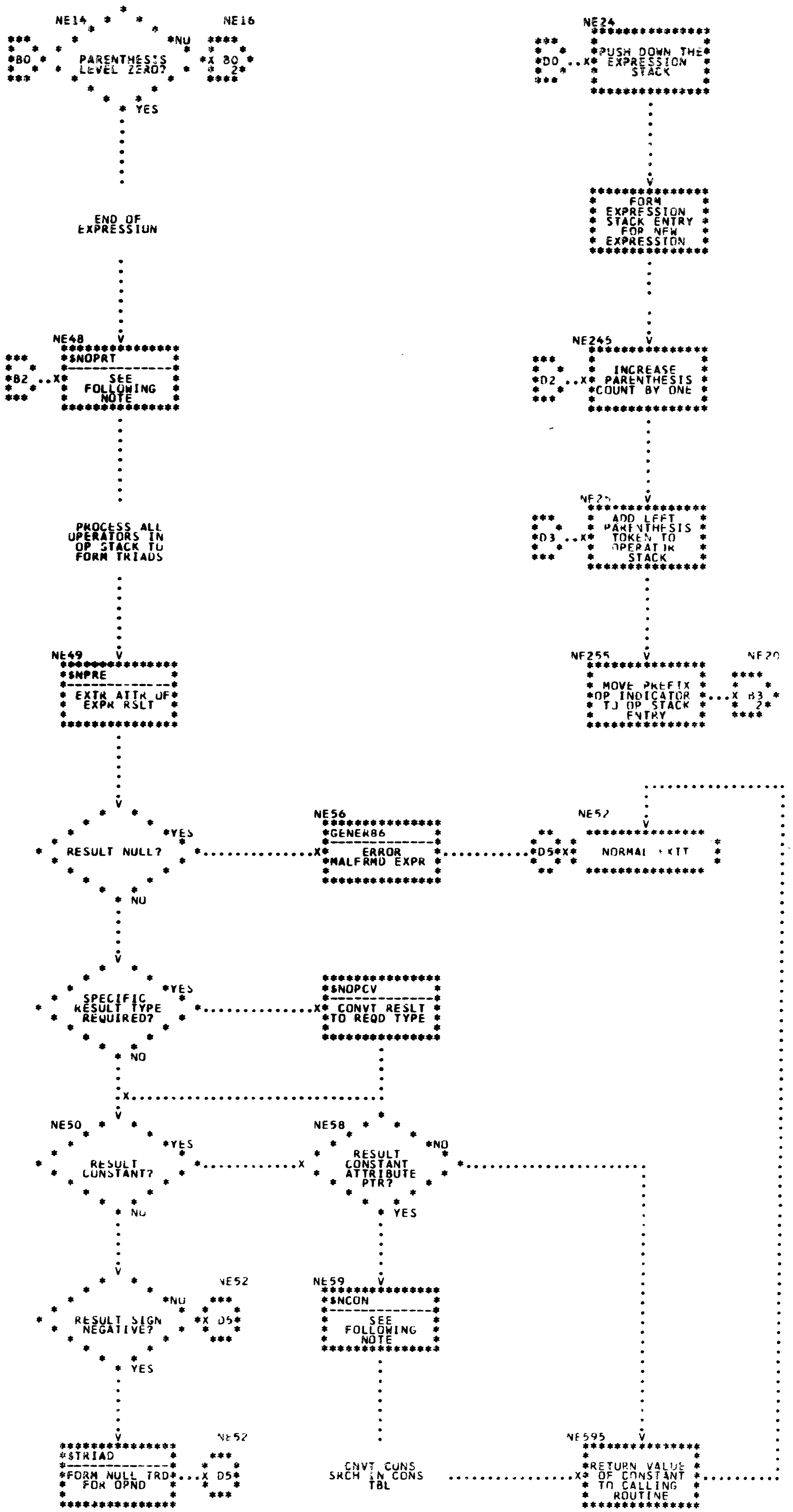


Chart 46. Expression Processor Controller (Page 3 of 11)

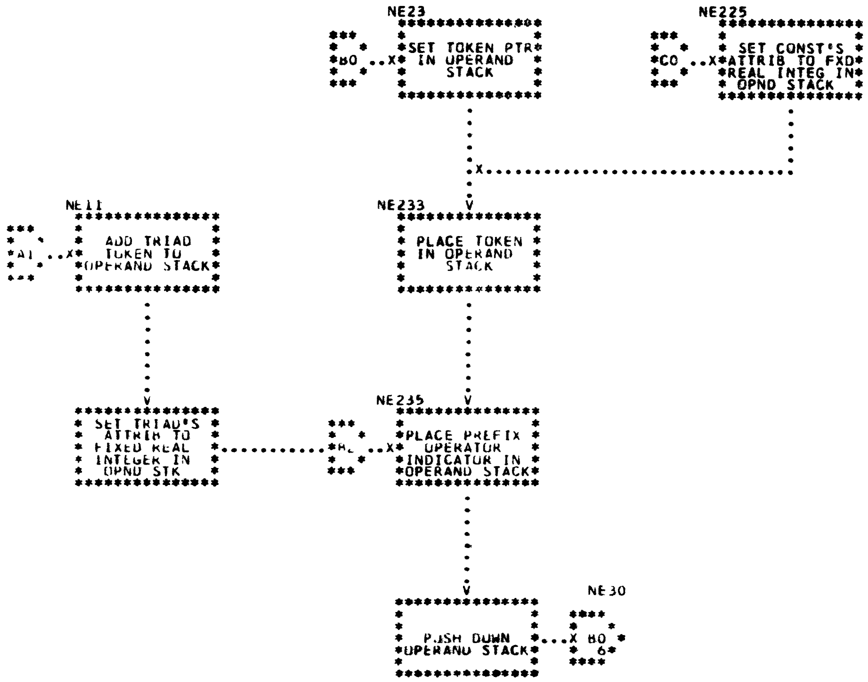


Chart 46. Expression Processor Controller (Page 4 of 11)

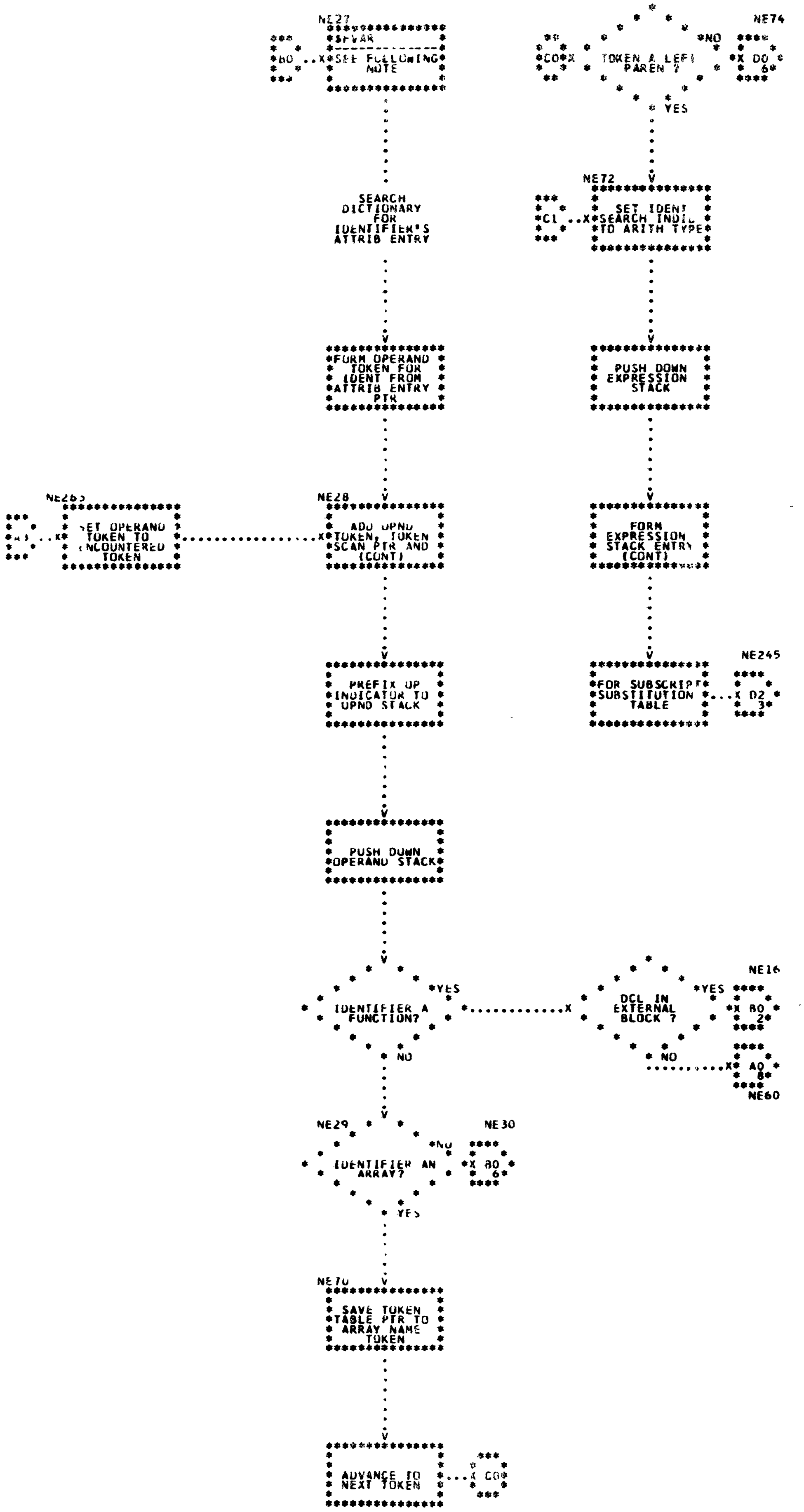
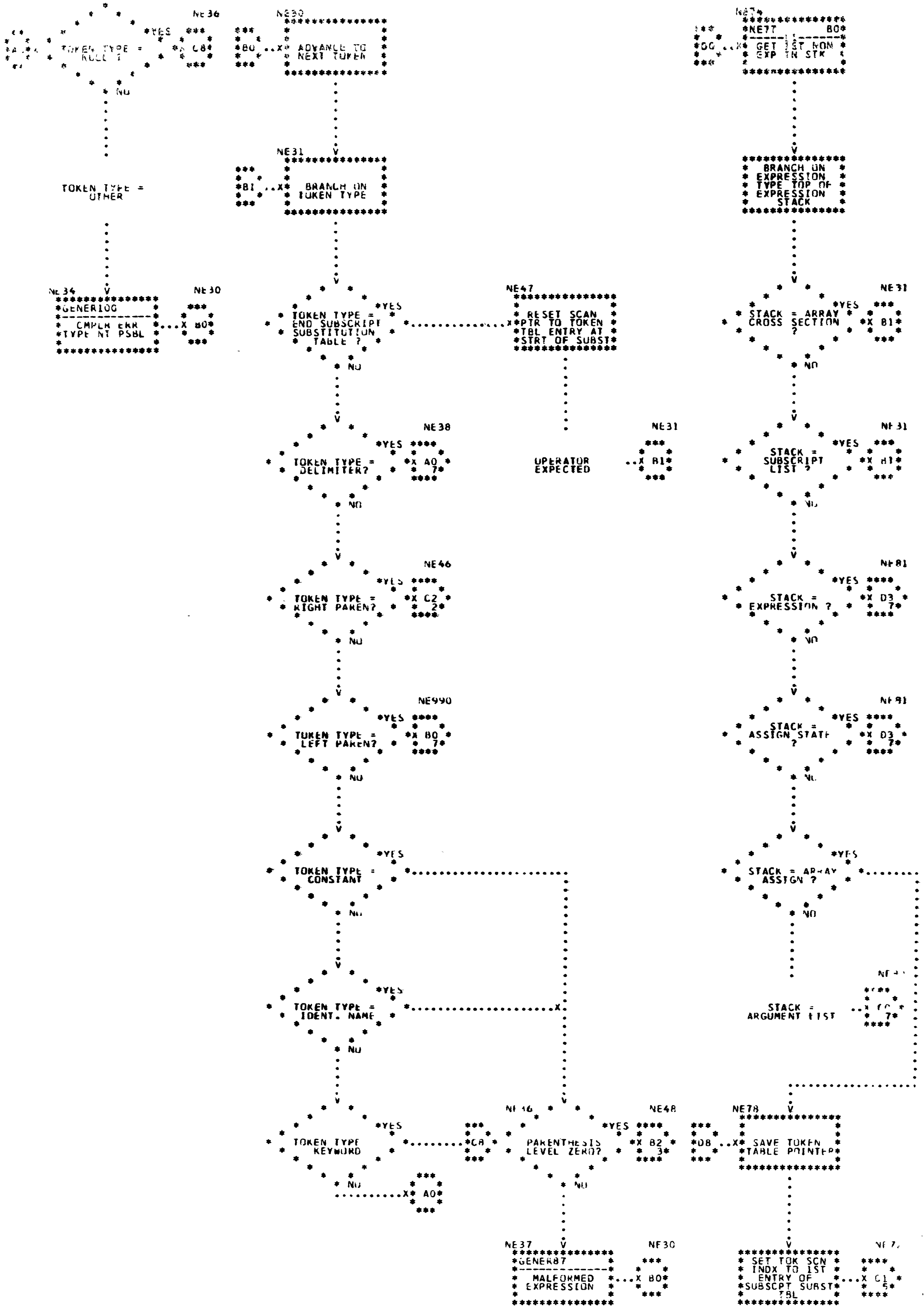


Chart 46. Expression Processor Controller (Page 5 of 11)



236 Chart 46. Expression Processor Controller (Page 6 of 11)

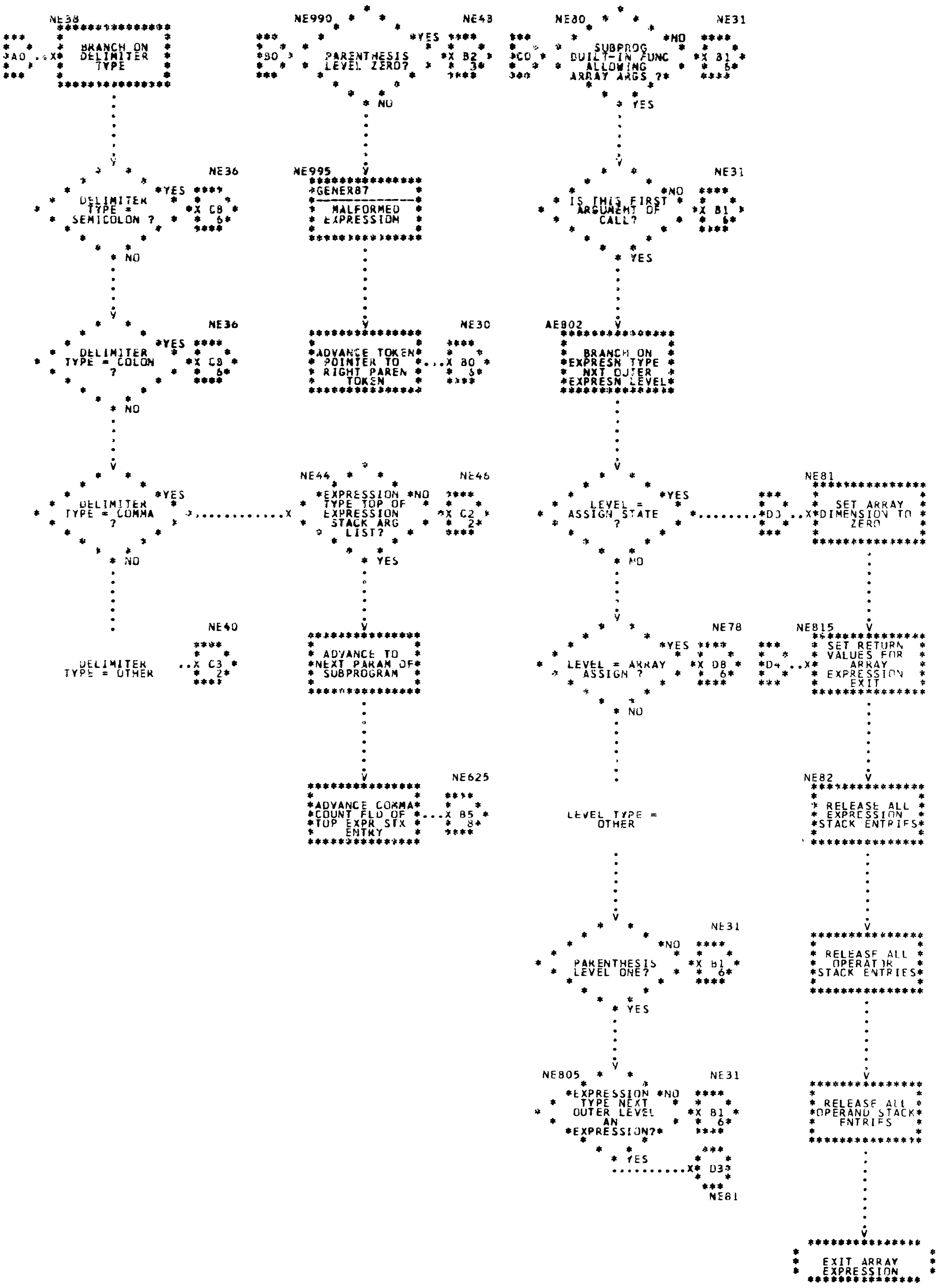


Chart 46. Expression Processor Controller (Page 7 of 11)

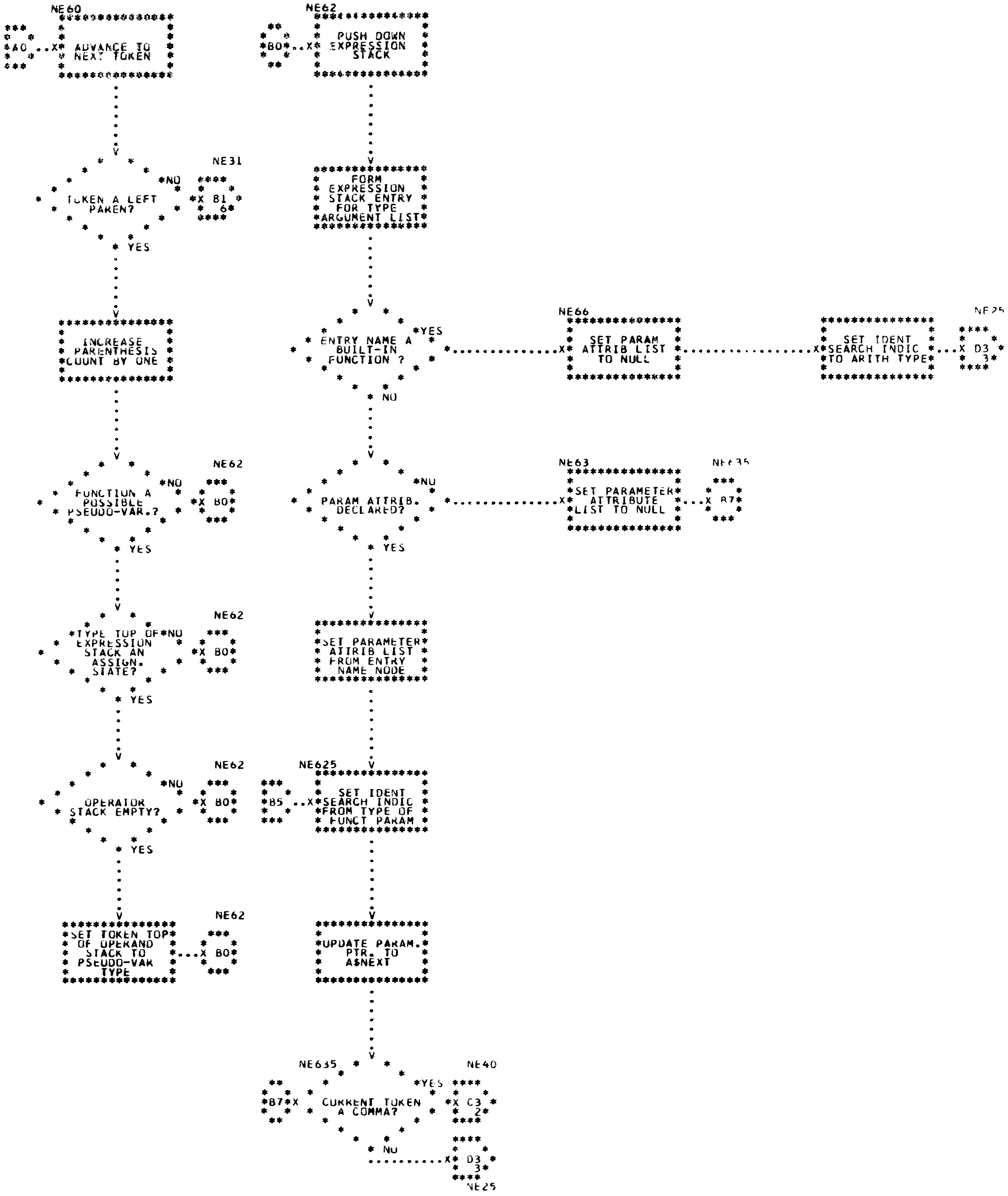


Chart 46. Expression Processor Controller (Page 8 of 11)

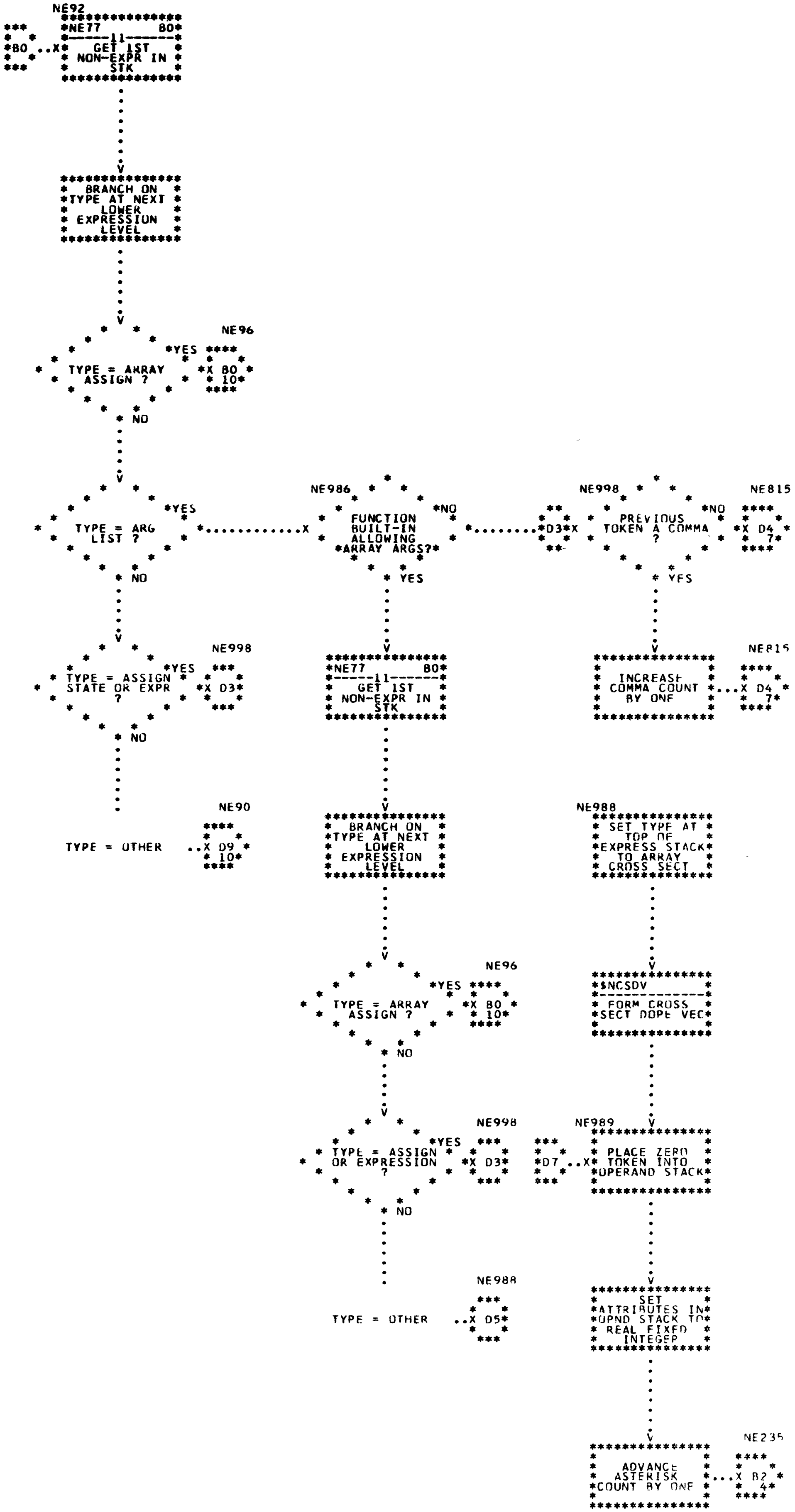


Chart 46. Expression Processor Controller (Page 9 of 11)

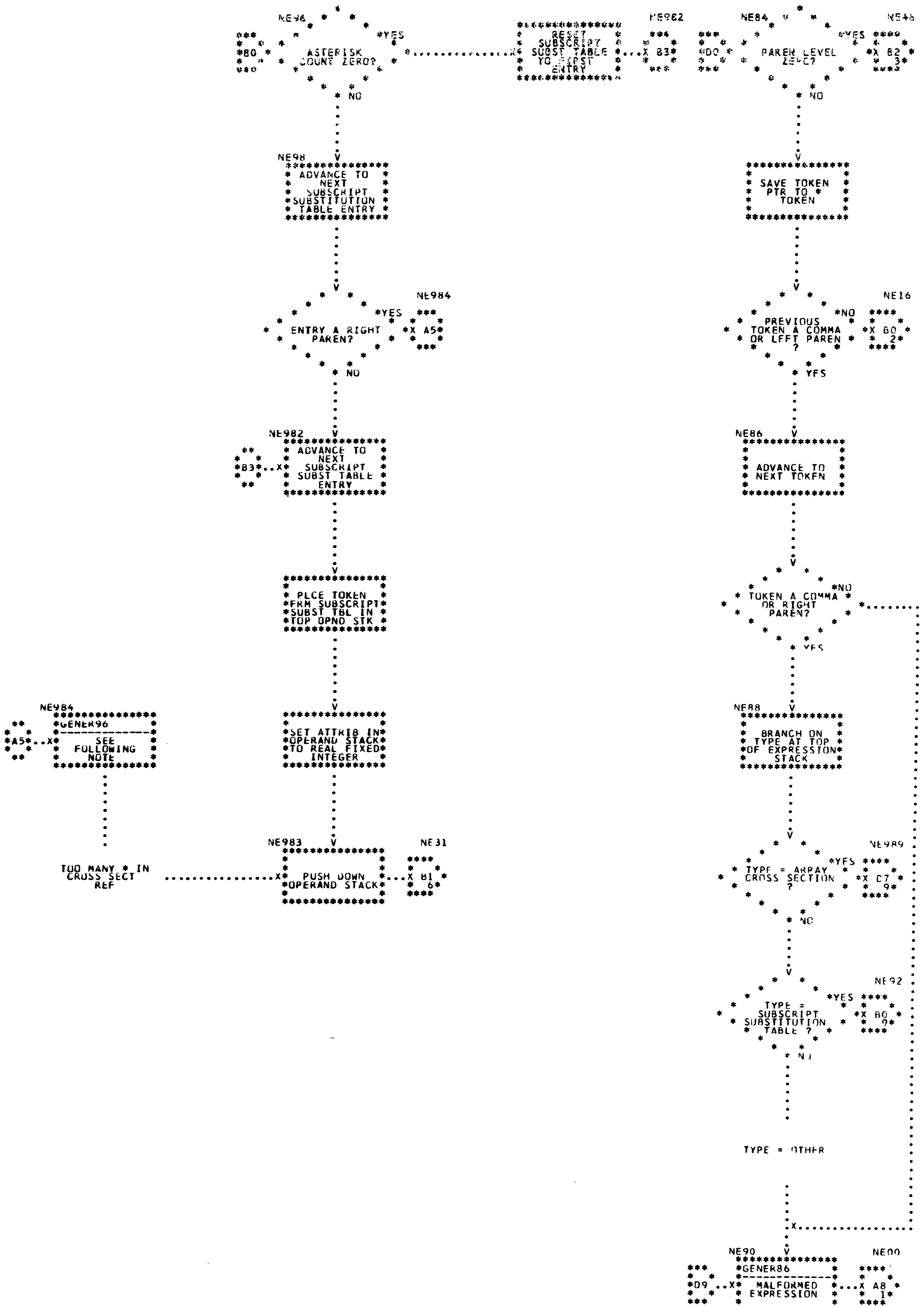


Chart 46. Expression Processor Controller (Page 10 of 11)

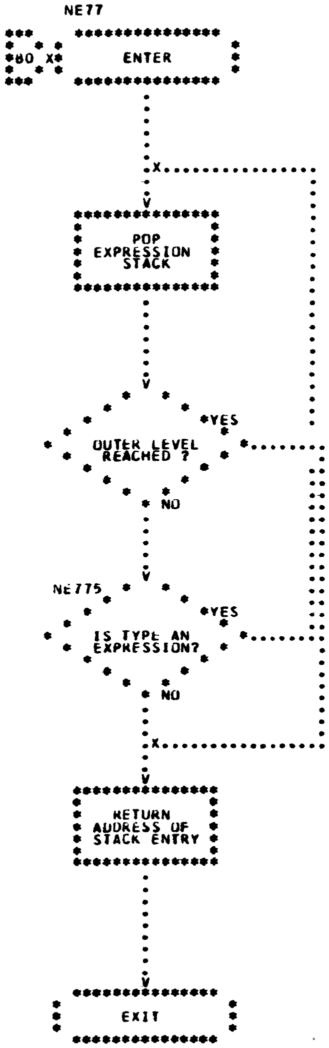


Chart 46. Expression Processor Controller (Page 11 of 11)

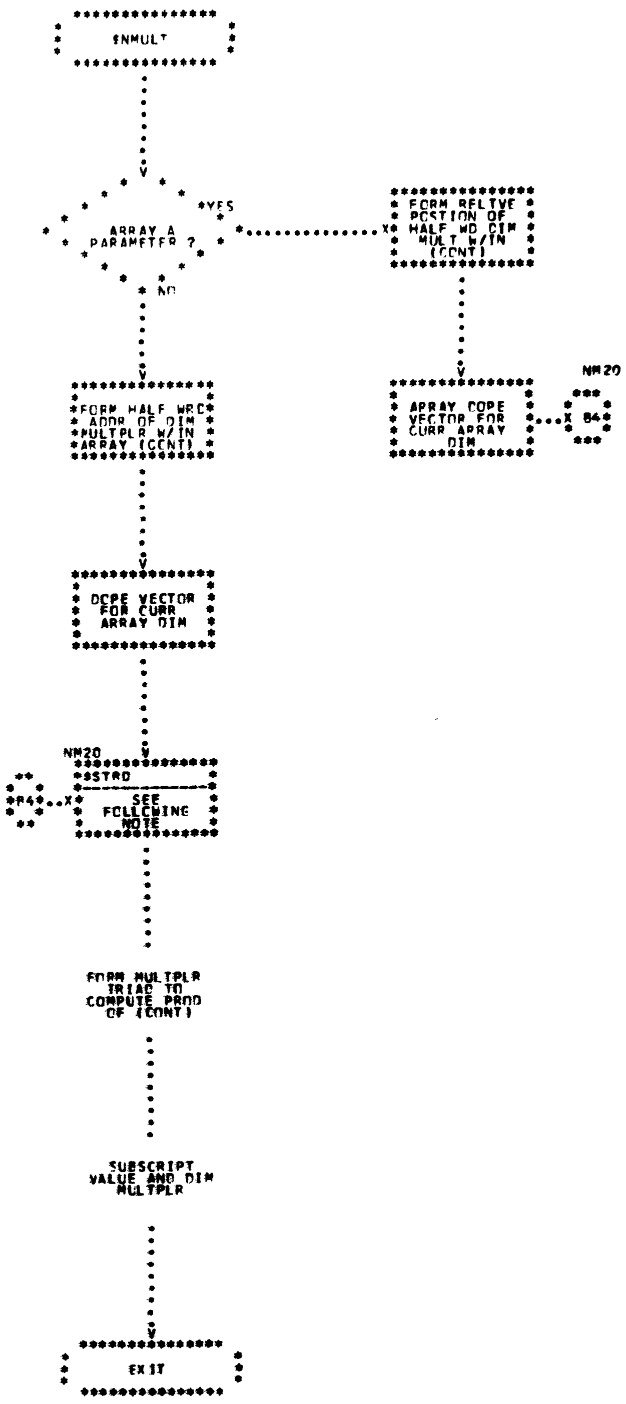


Chart 47. Dimension Multiplier Generator

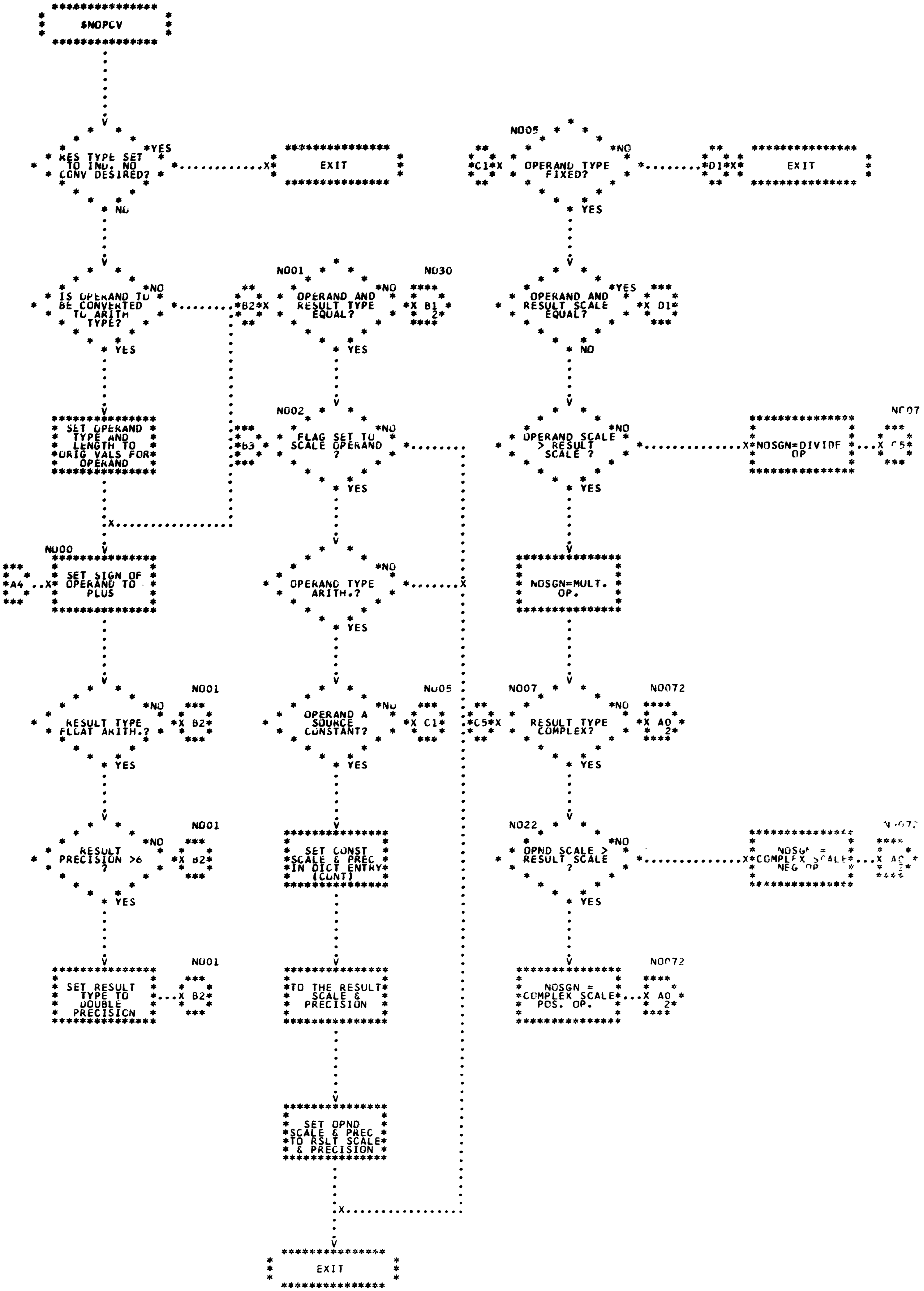


Chart 48. Convert Operand (Page 1 of 3)

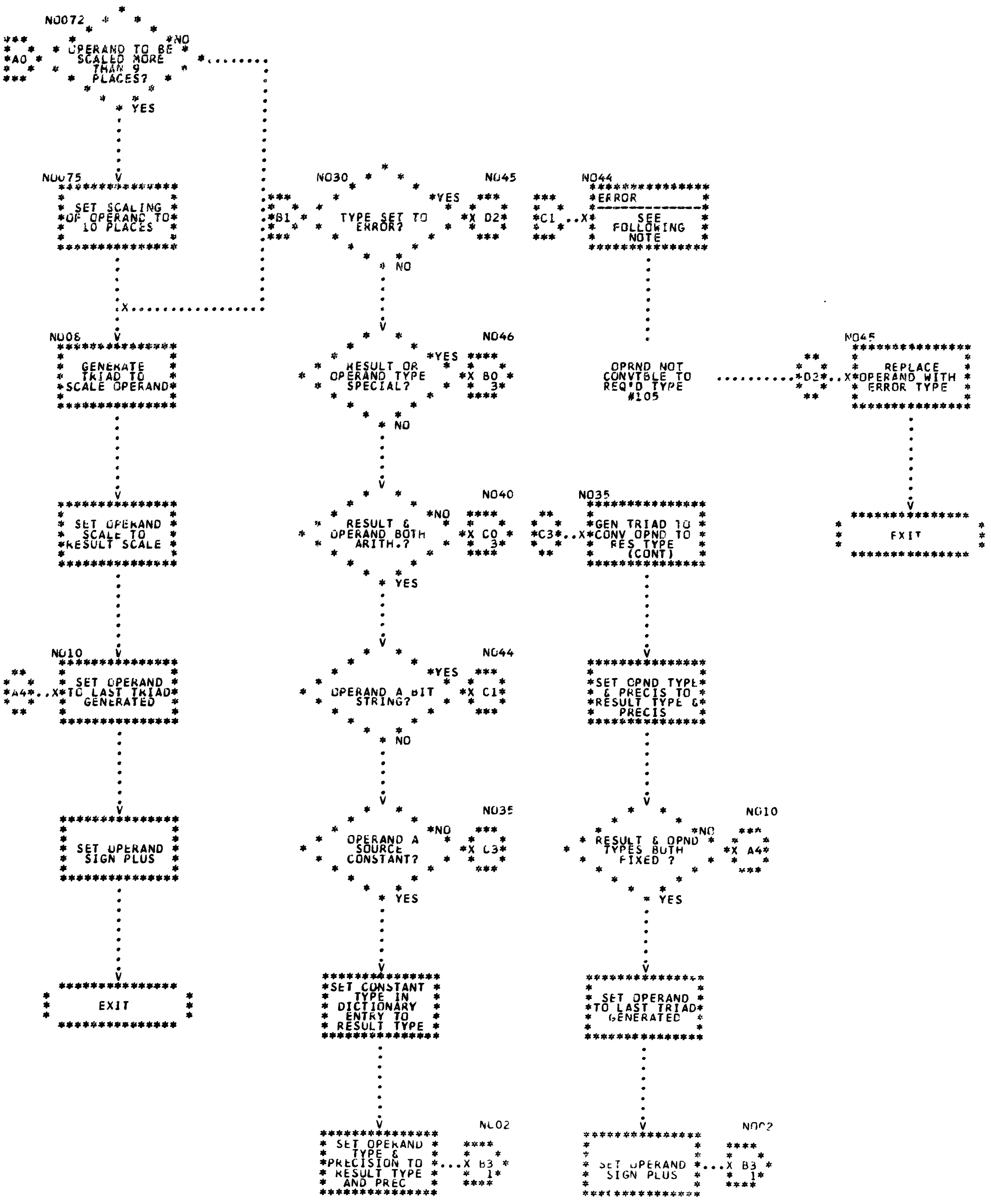


Chart 48. Convert Operand (Page 2 of 3)

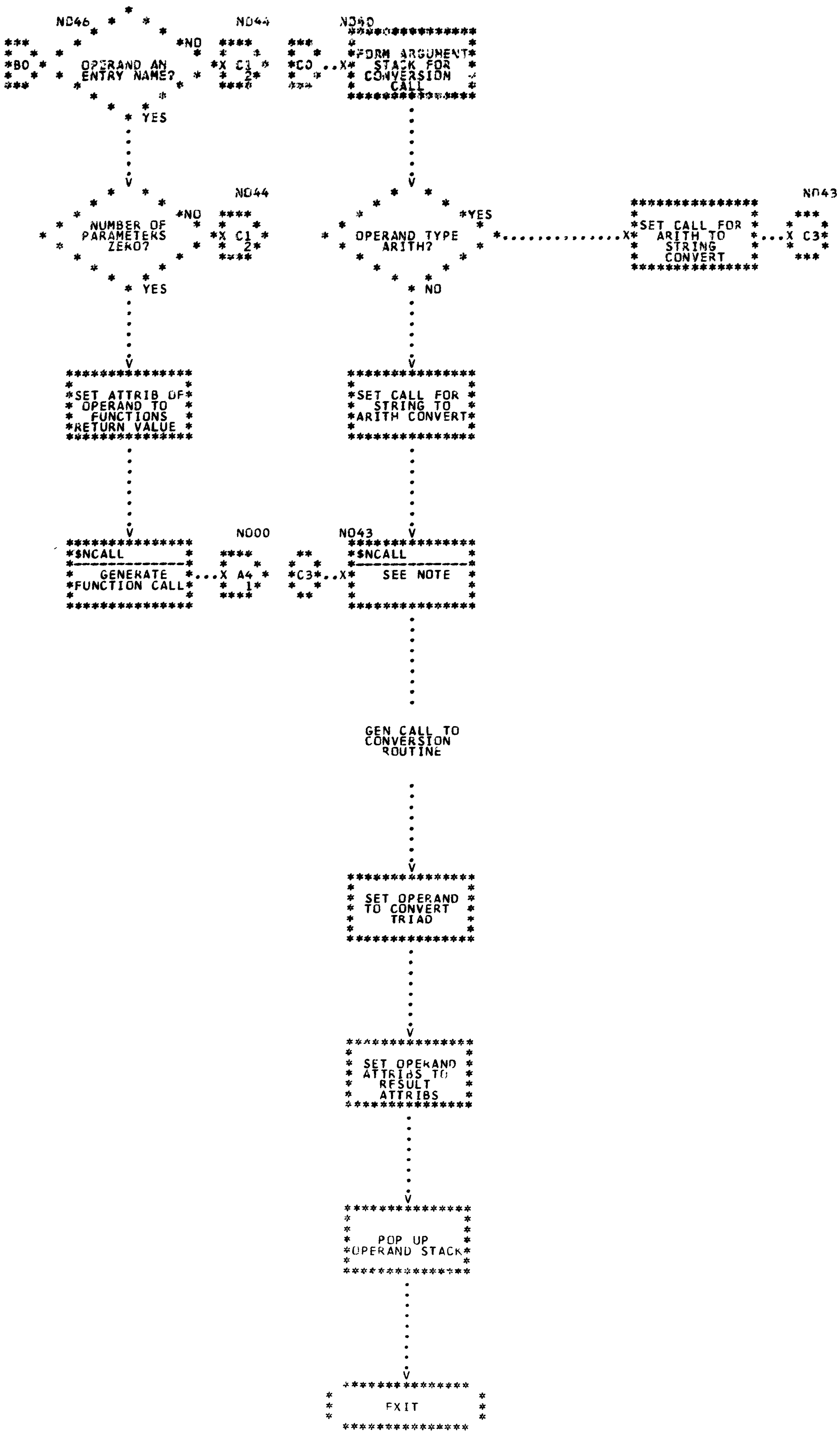


Chart 48. Convert Operand (Page 3 of 3)

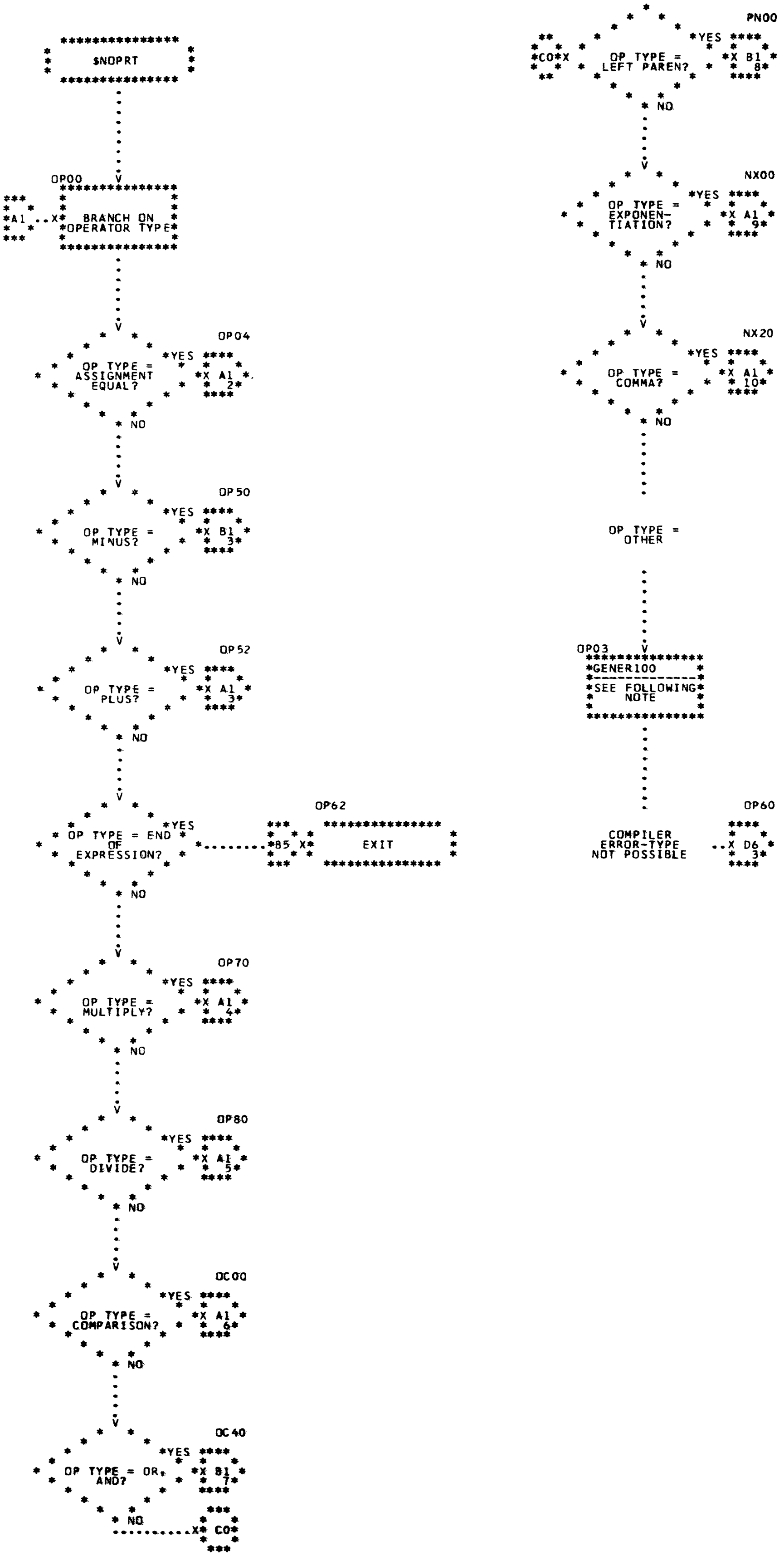


Chart 49. Operator Stack Processor (Page 1 of 17)

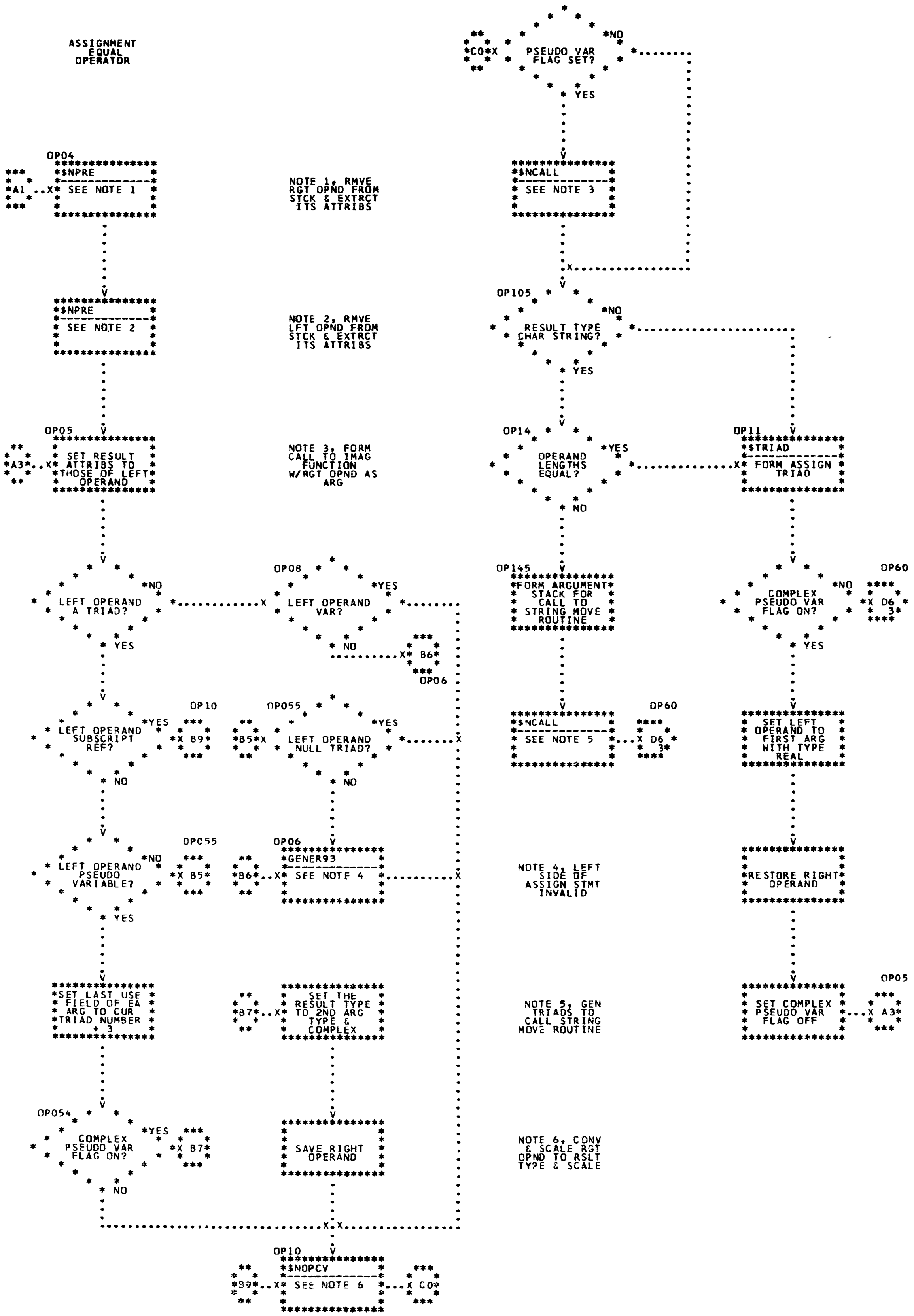


Chart 49. Operator Stack Processor (Page 2 of 17)

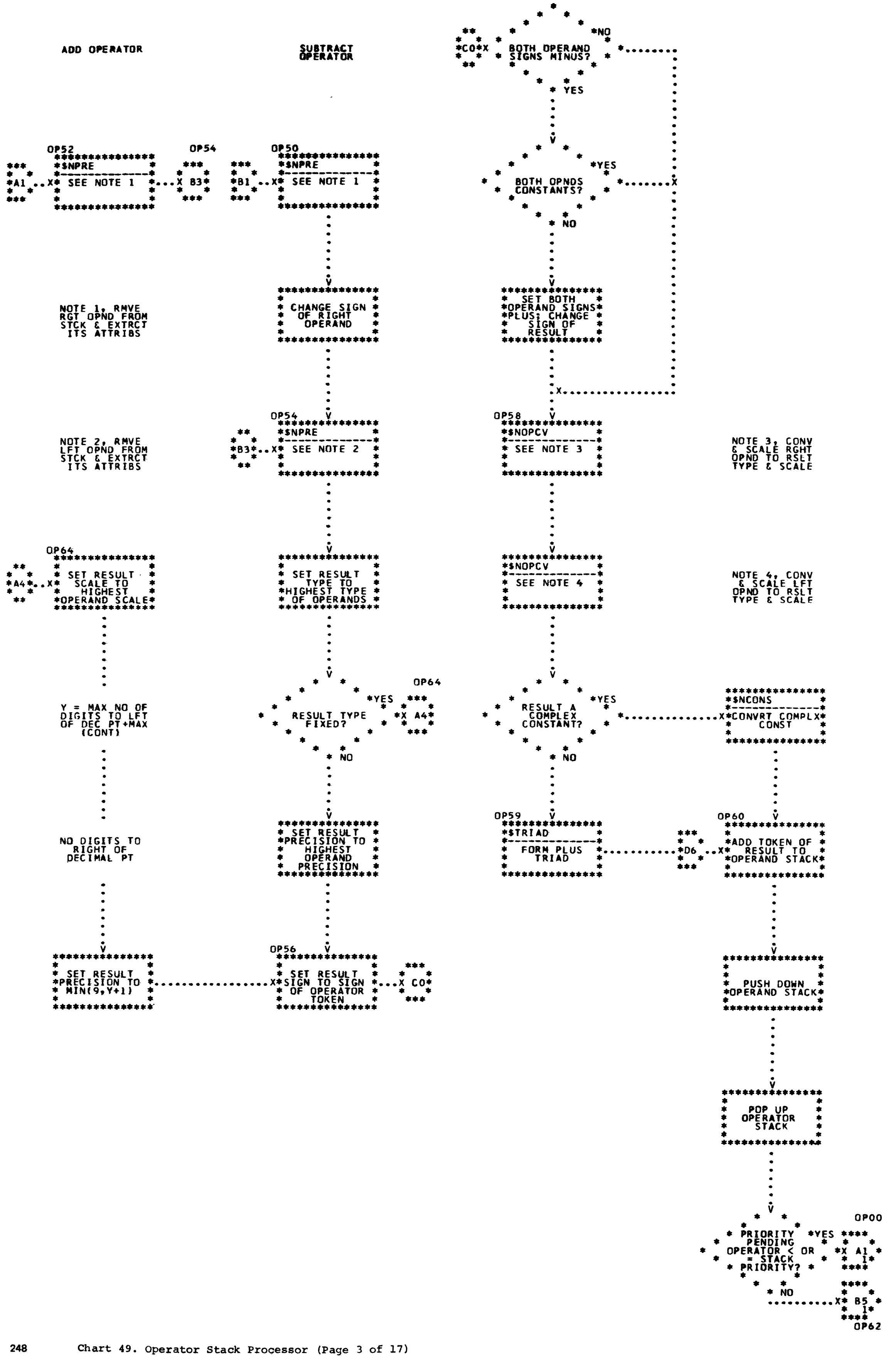


Chart 49. Operator Stack Processor (Page 3 of 17)

MULTIPLY
OPERATOR

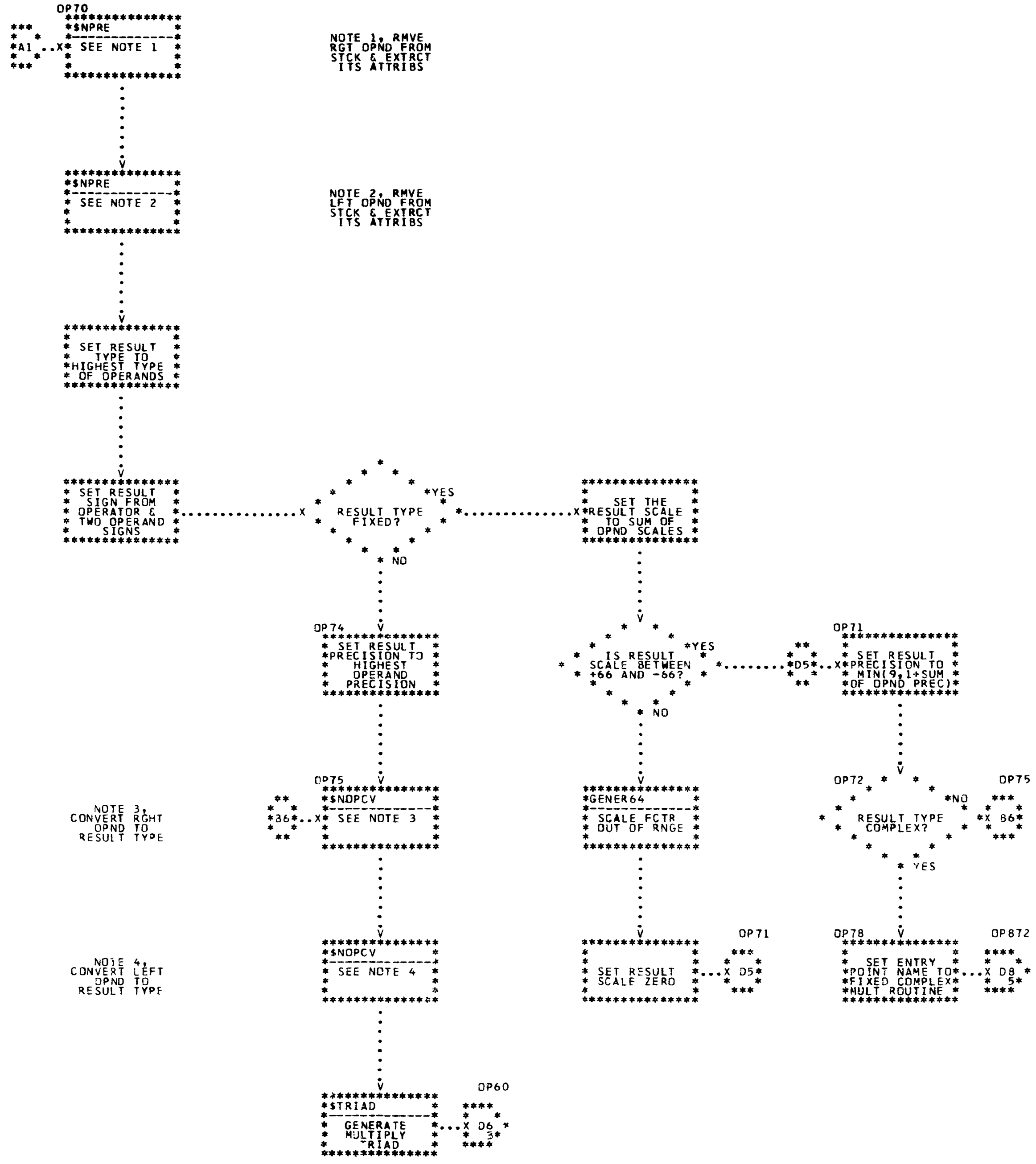
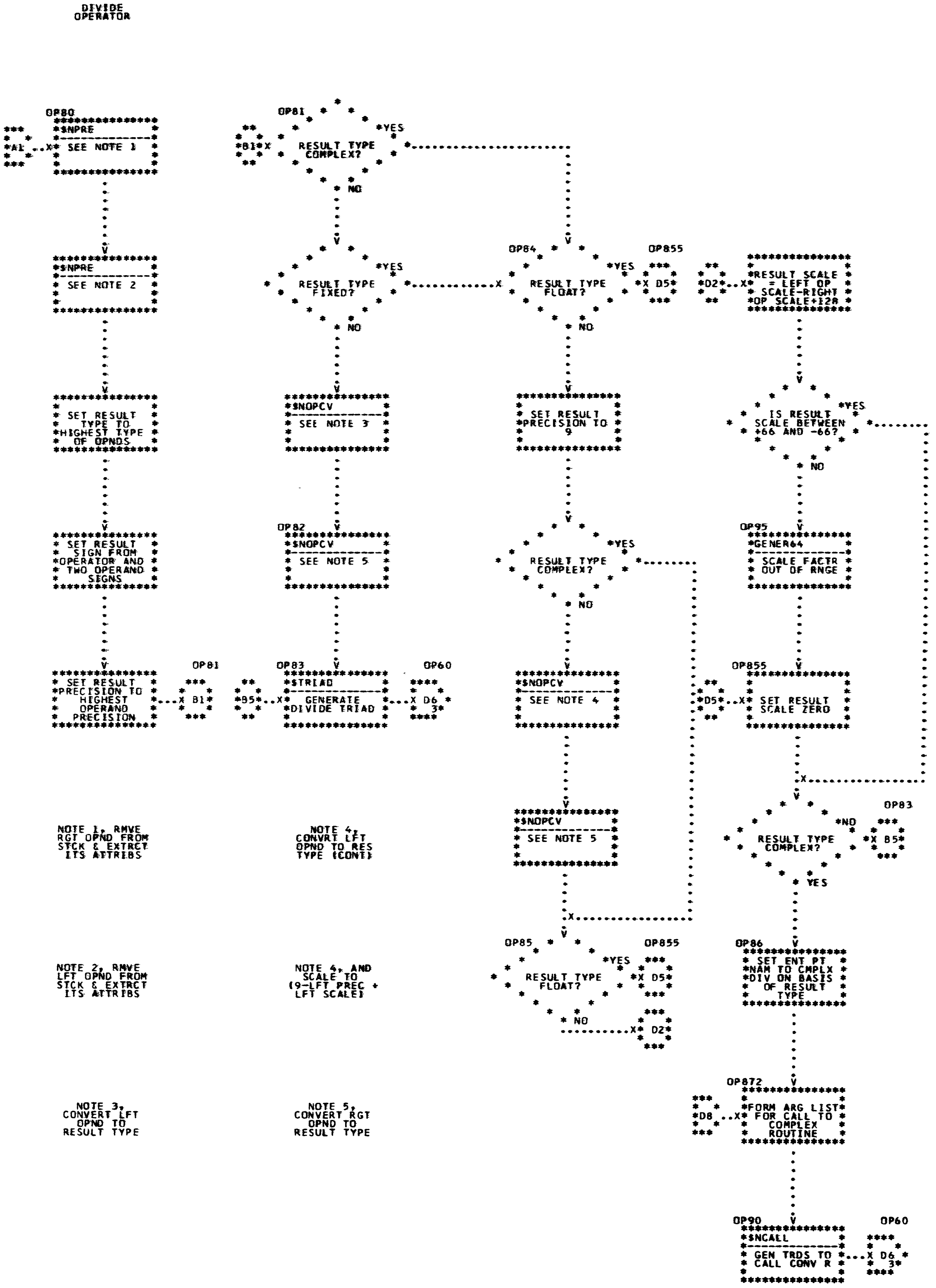


Chart 49. Operator Stack Processor (Page 4 of 17)



NOTE 1, RMVE RGT OPND FROM STCK & EXTRCT ITS ATTRIBS

NOTE 2, RMVE LFT OPND FROM STCK & EXTRCT ITS ATTRIBS

NOTE 3, CONVERT LFT OPND TO RESULT TYPE

NOTE 4, CONVRT LFT OPND TO RES TYPE (CONT)

NOTE 4, AND SCALE TO (9-LFT PREC + LFT SCALE)

NOTE 5, CONVERT RGT OPND TO RESULT TYPE

Chart 49. Operator Stack Processor (Page 5 of 17)

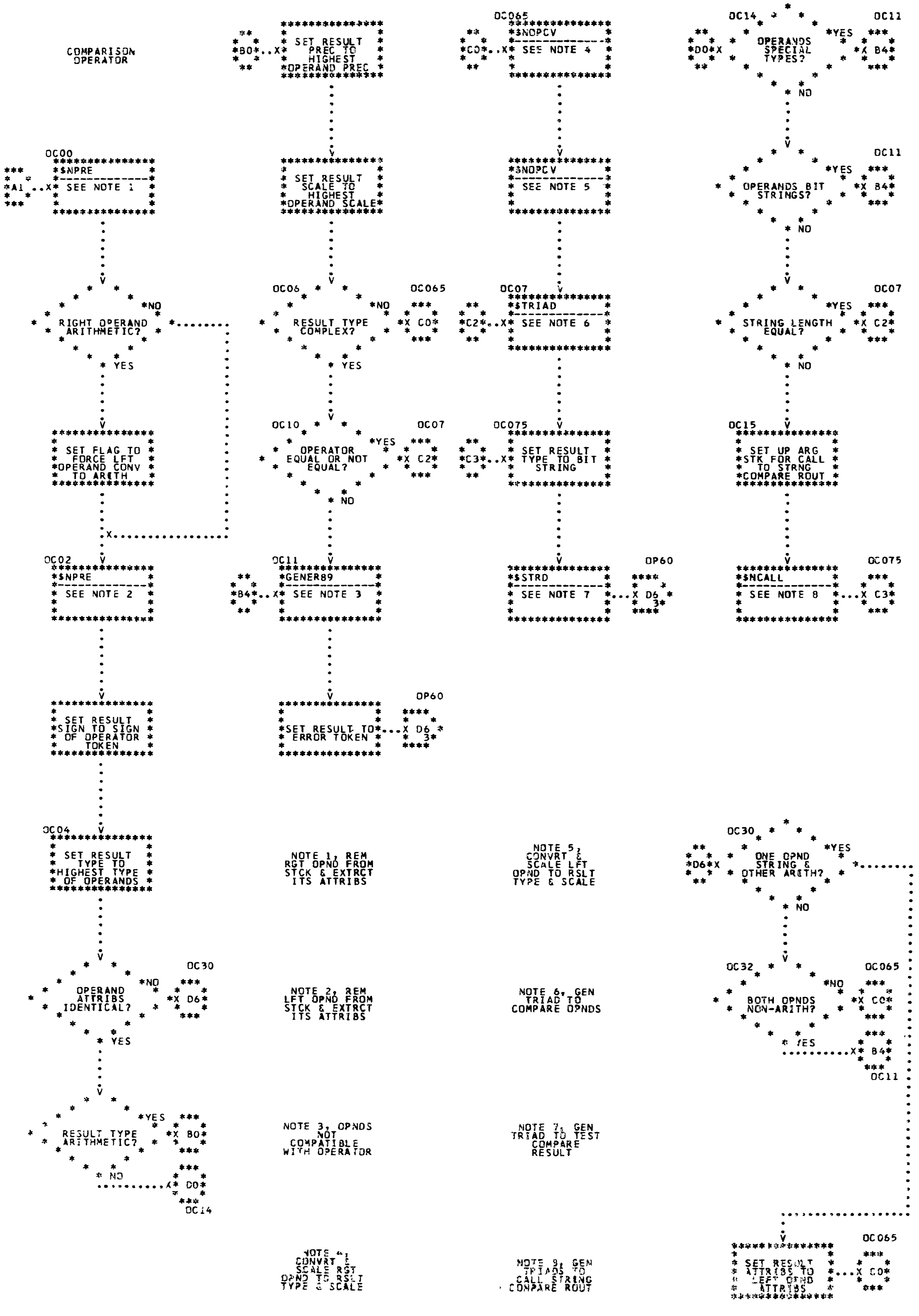
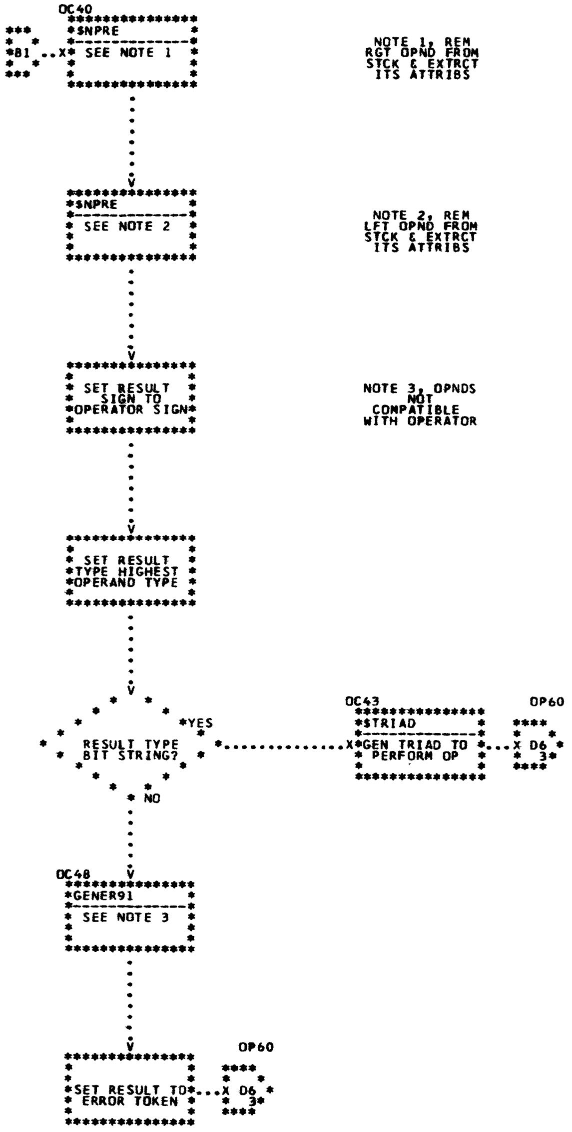
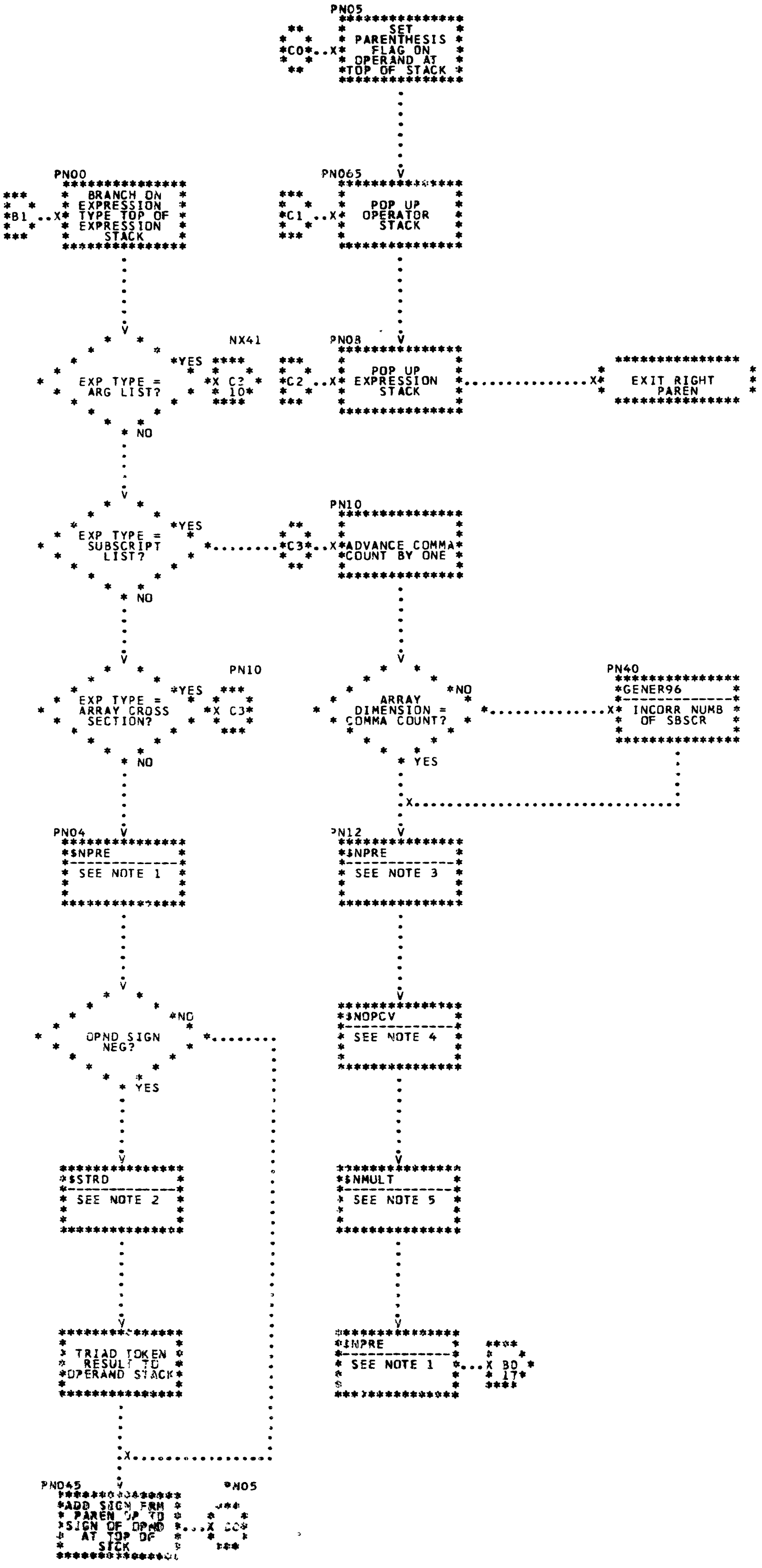


Chart 49. Operator Stack Processor (Page 6 of 17)

AND, OR
OPERATOR



LEFT PAREN OPERATOR



NOTE 1, REM LFT OPND FROM STCK & EXTRCT ITS ATTRIBS

NOTE 2, GEN NULL VALUE FOR NEG VAL

NOTE 3, REM RGT OPND FROM STCK & EXTRCT ITS ATTRIBS

NOTE 4, CONV & SCALE OPND TO REAL FIXED INT

NOTE 5, FORM TRIAD TO MULT SBSCRPT BY DIM MULTIPLIER

Chart 49. Operator Stack Processor (Page 3 of 17)

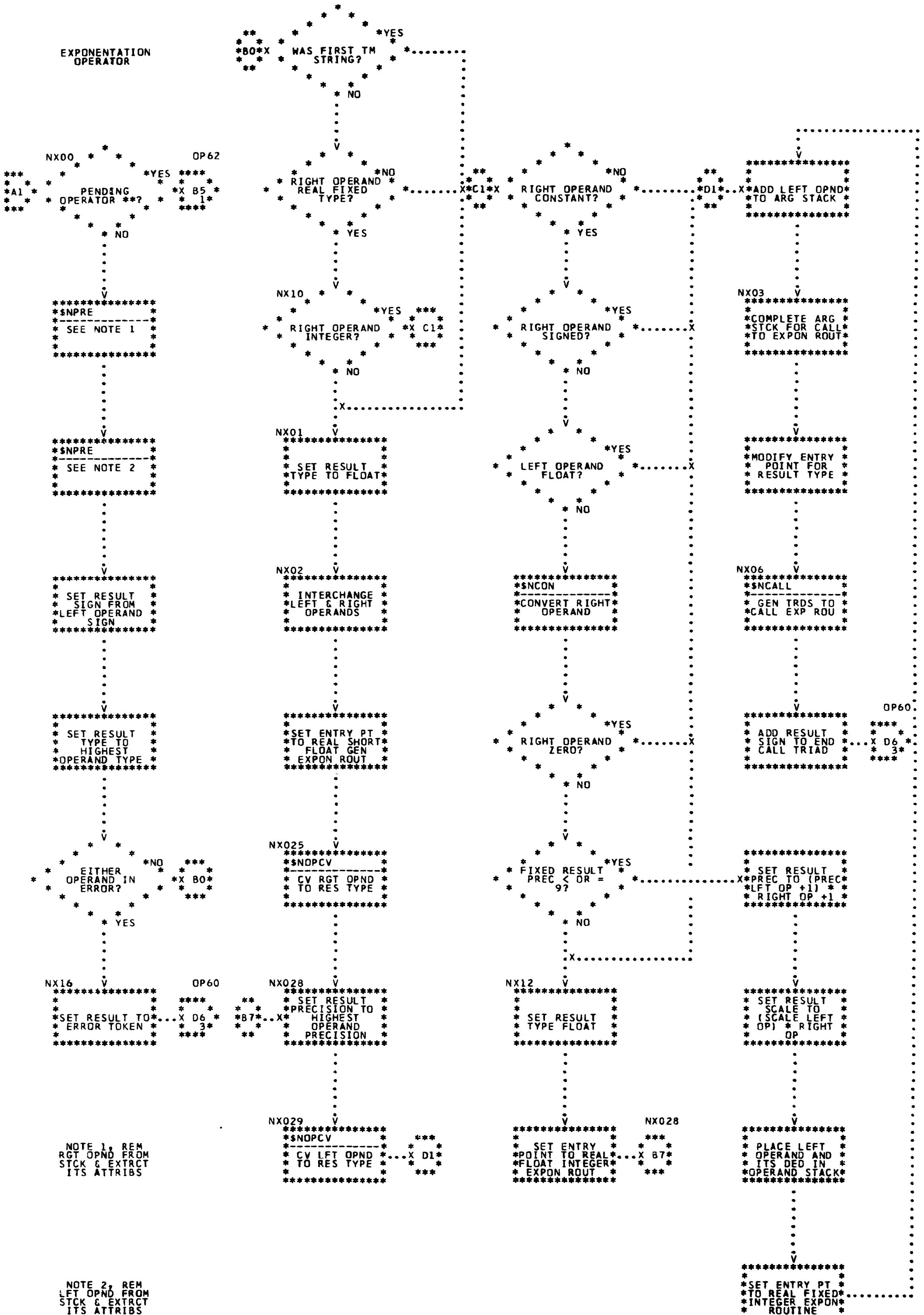


Chart 49. Operator Stack Processor (Page 9 of 17)

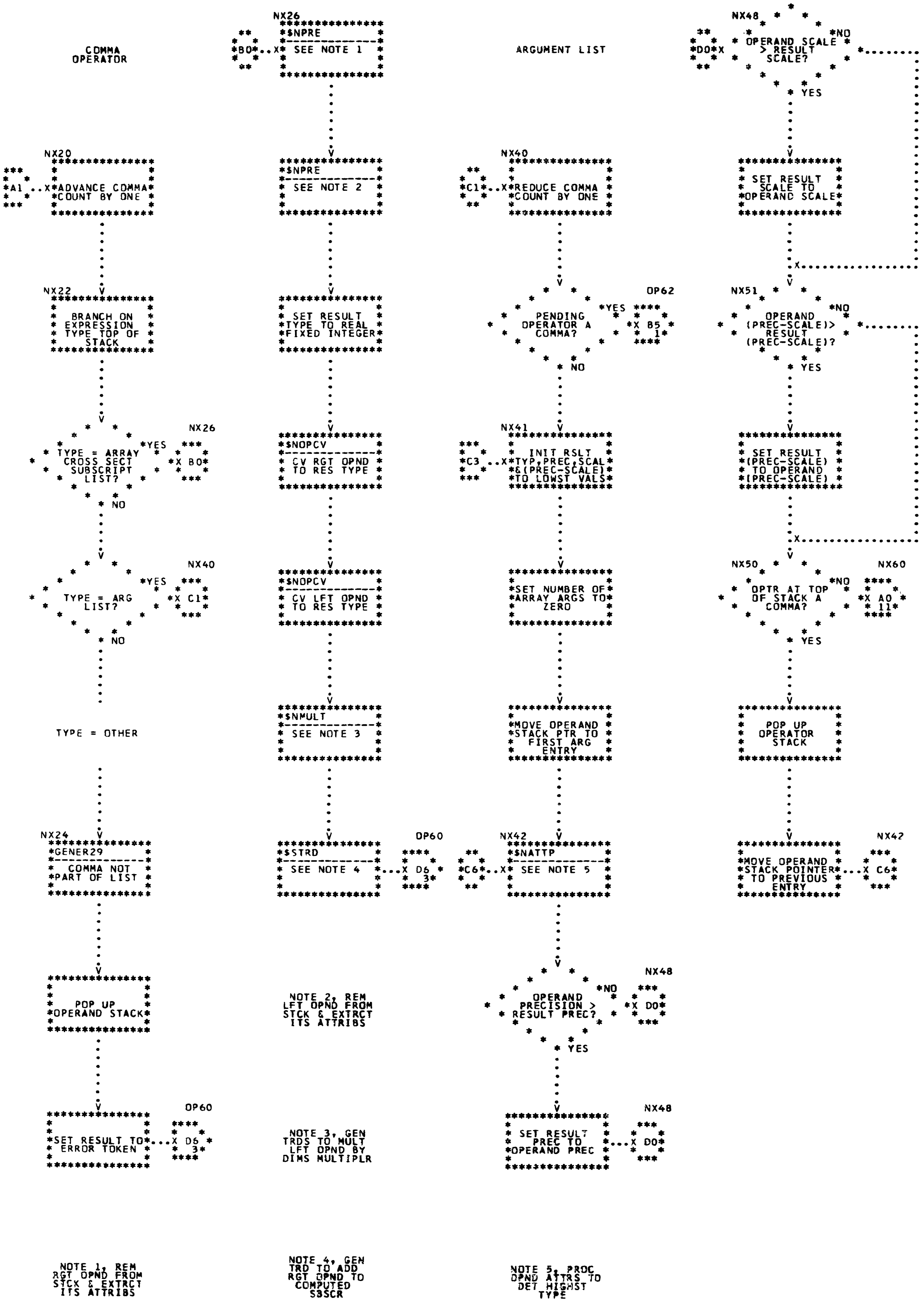


Chart 49. Operator Stack Processor (Page 10 of 17)

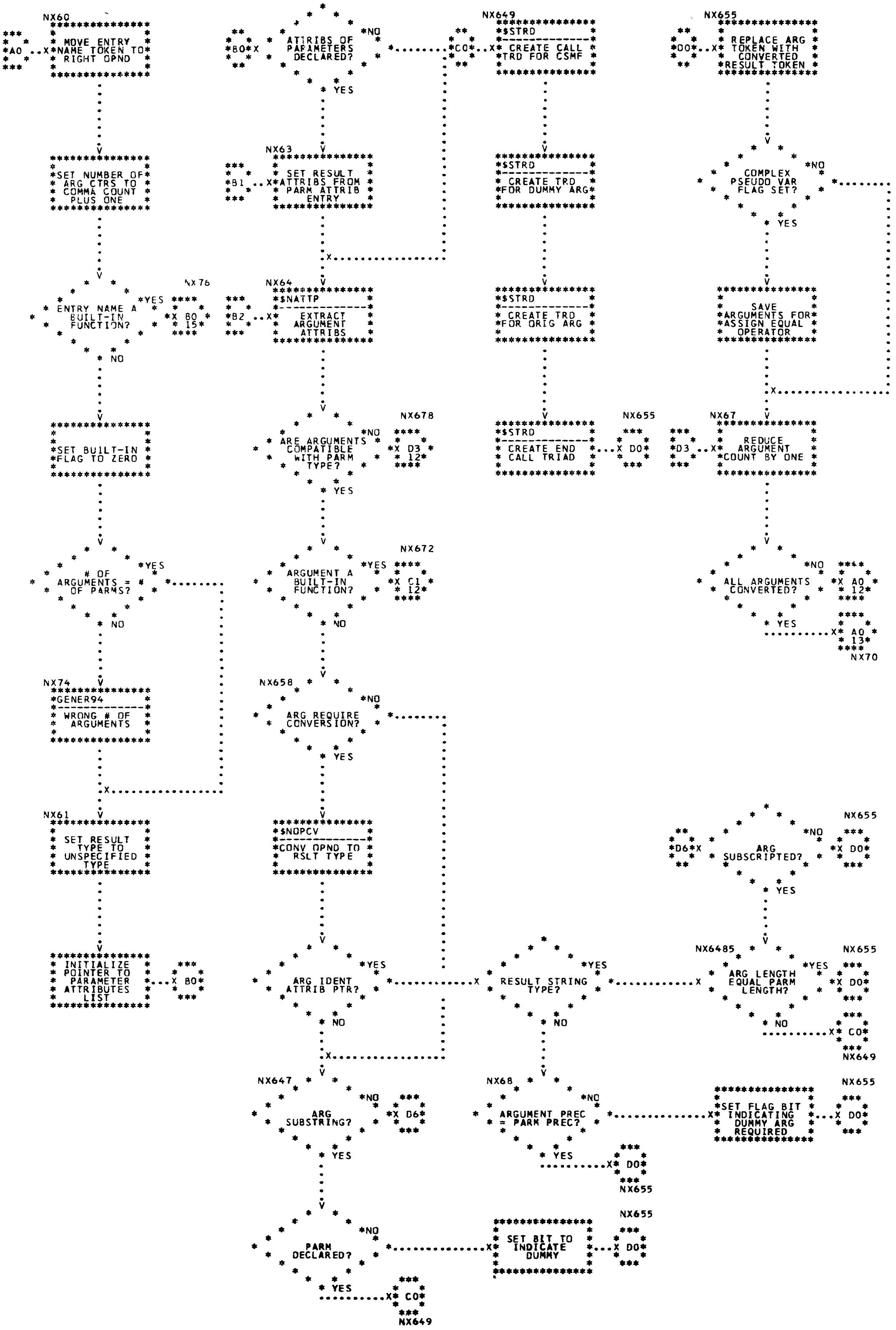
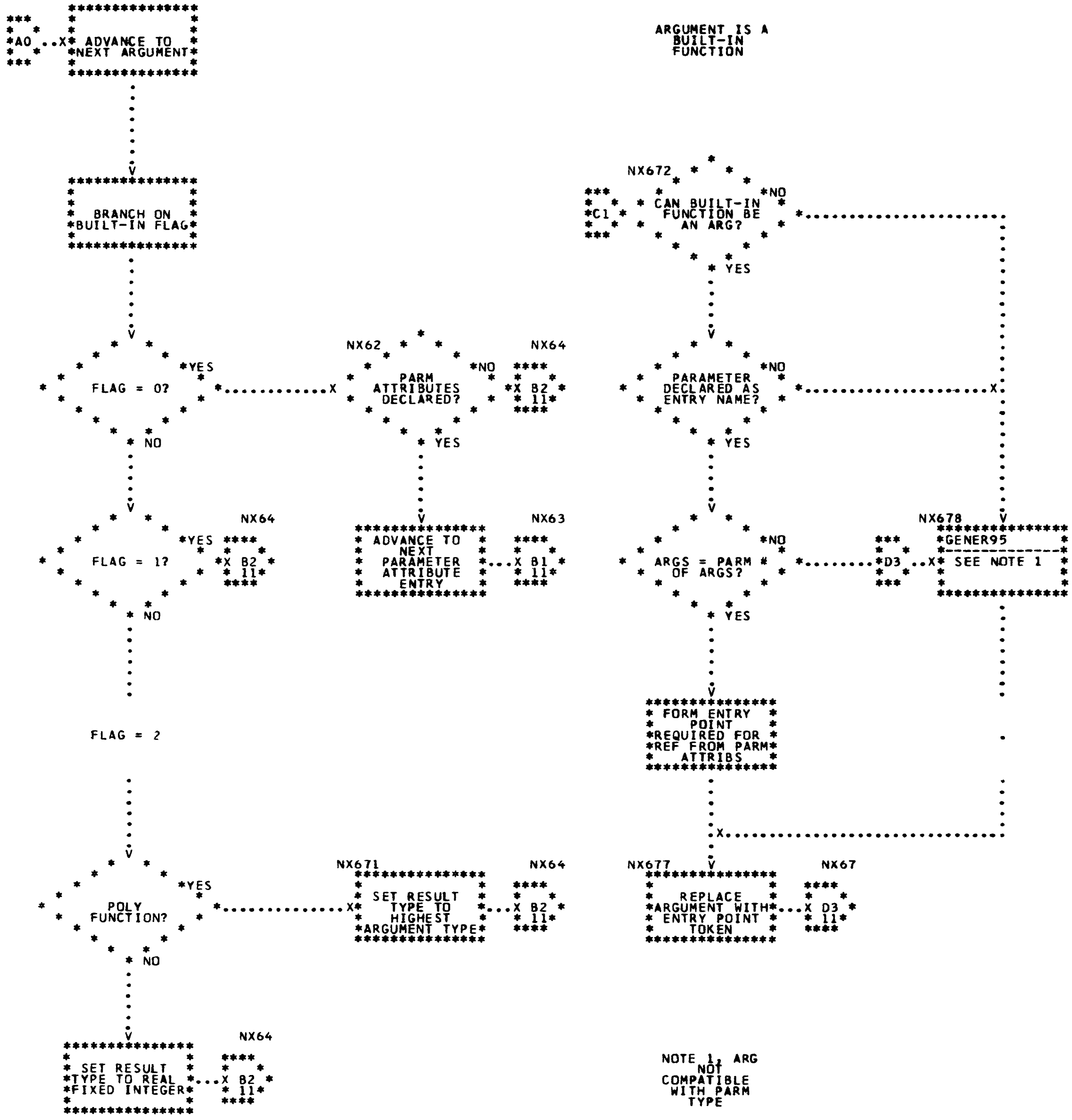


Chart 49. Operator Stack Processor (Page 11 of 17)


A0
ADVANCE TO NEXT ARGUMENT
BRANCH ON BUILT-IN FLAG
FLAG = 0?
YES
NO
NX62
PARM ATTRIBUTES DECLARED?
NO
NX64
B2 11
YES
NX63
ADVANCE TO NEXT PARAMETER ATTRIBUTE ENTRY
B1 11
FLAG = 1?
YES
NX64
B2 11
NO
FLAG = 2
POLY FUNCTION?
YES
NX671
SET RESULT TYPE TO HIGHEST ARGUMENT TYPE
NX64
B2 11
NO
NX64
SET RESULT TYPE TO REAL FIXED INTEGER
B2 11
ARGUMENT IS A BUILT-IN FUNCTION
NX672
C1
CAN BUILT-IN FUNCTION BE AN ARG?
NO
YES
PARAMETER DECLARED AS ENTRY NAME?
NO
YES
ARGS = PARM # OF ARGS?
NO
NX678
D3
GENER95
SEE NOTE 1
YES
FORM ENTRY POINT REQUIRED FOR REF FROM PARM ATTRIBS
NX677
REPLACE ARGUMENT WITH ENTRY POINT TOKEN
NX67
D3 11
NOTE 1, ARG NOT COMPATIBLE WITH PARM TYPE

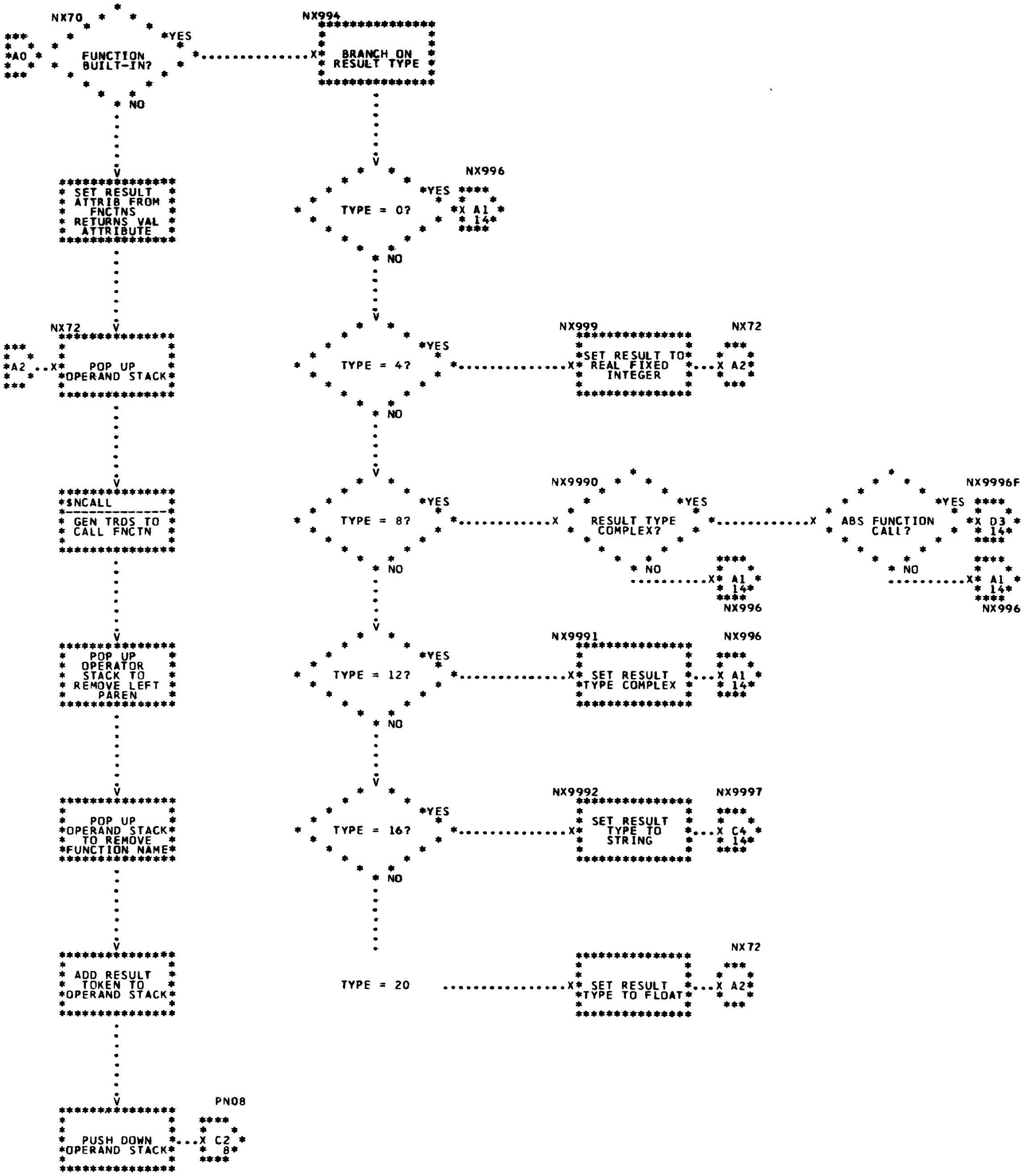


Chart 49. Operator Stack Processor (Page 13 of 17)

P = PRECISION
Q = SCALE
PQ =
PREC-SCALE

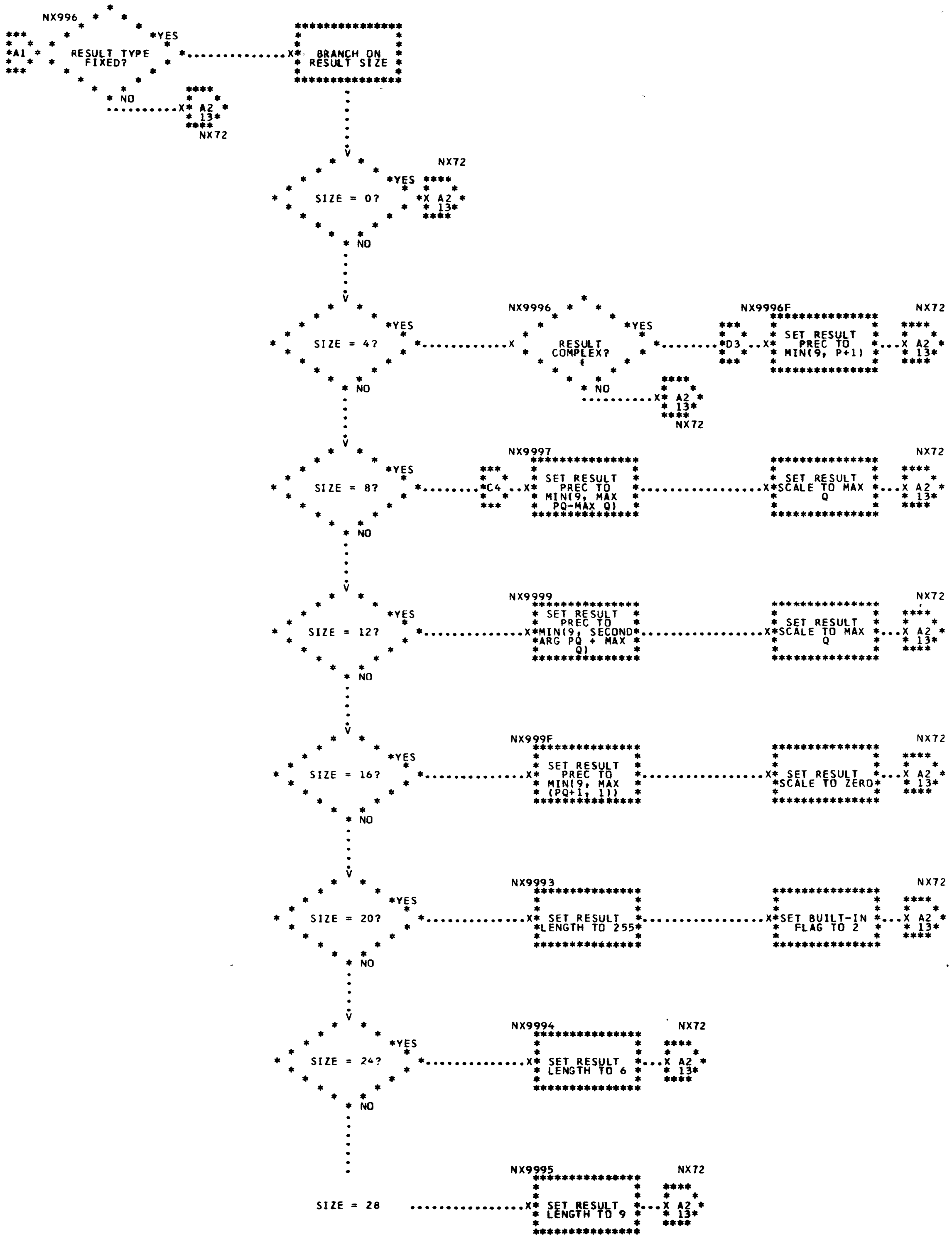


Chart 49. Operator Stack Processor (Page 14 of 17)

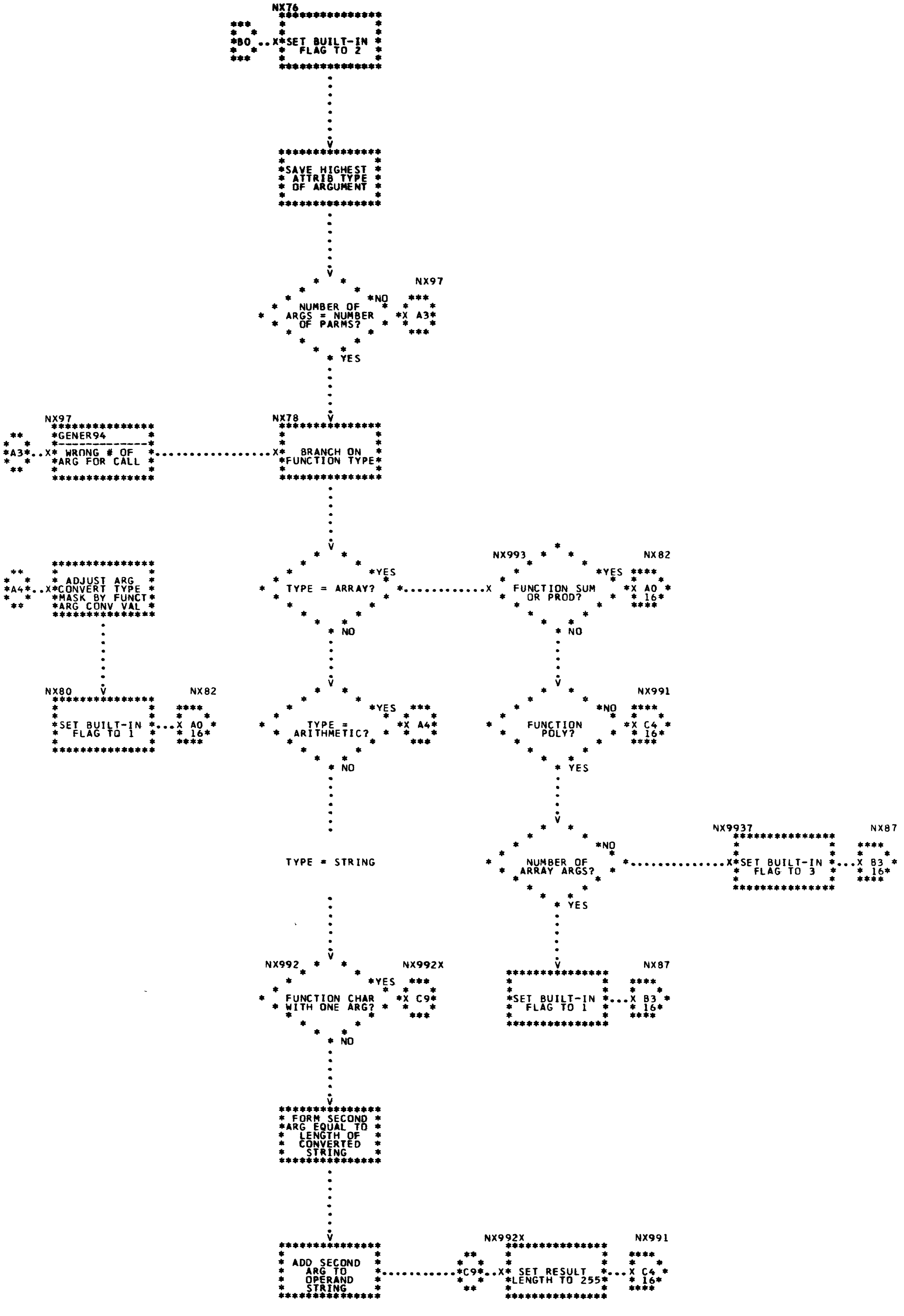


Chart 49. Operator Stack Processor (Page 15 of 17)

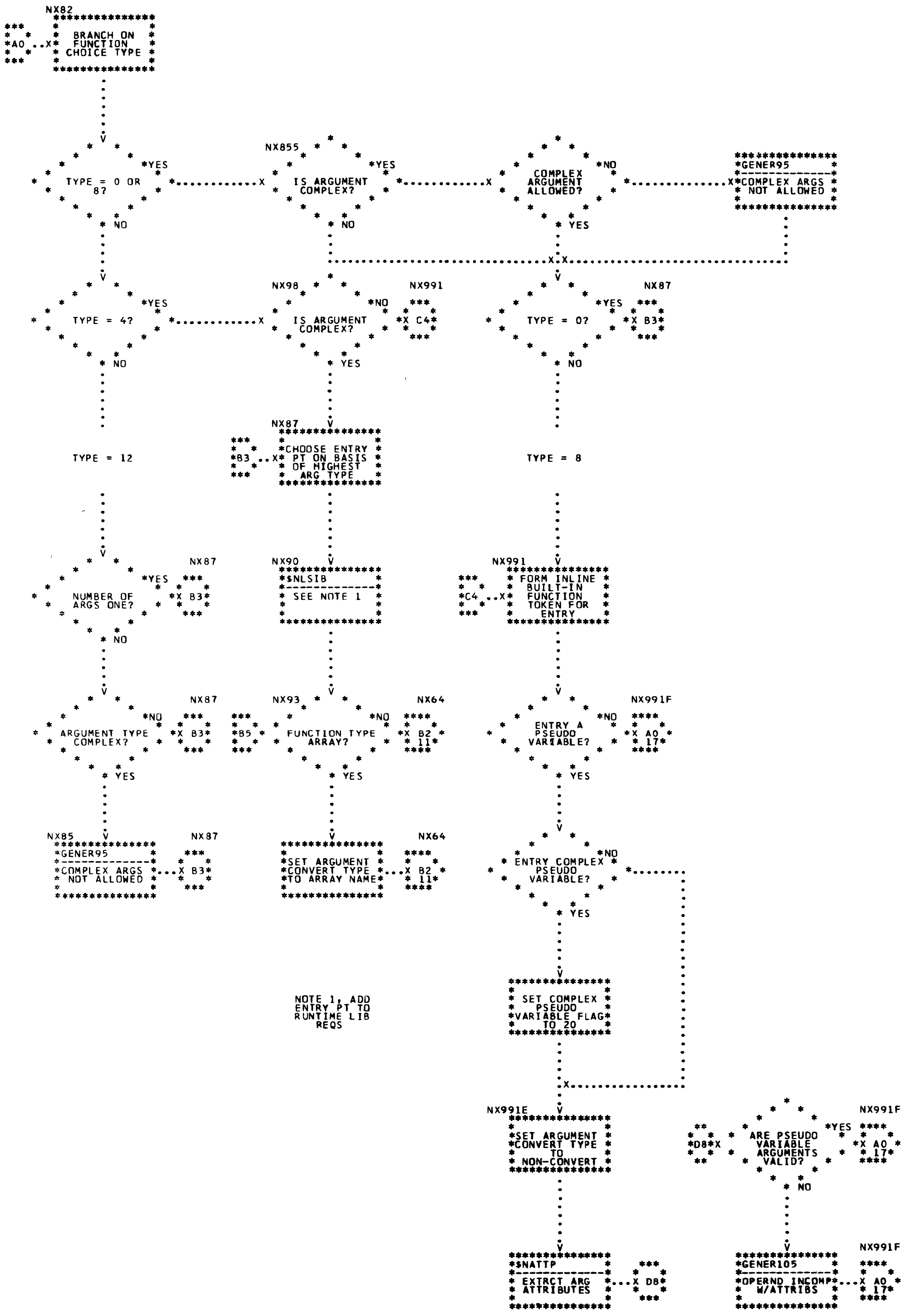


Chart 49. Operator Stack Processor (Page 16 of 17)

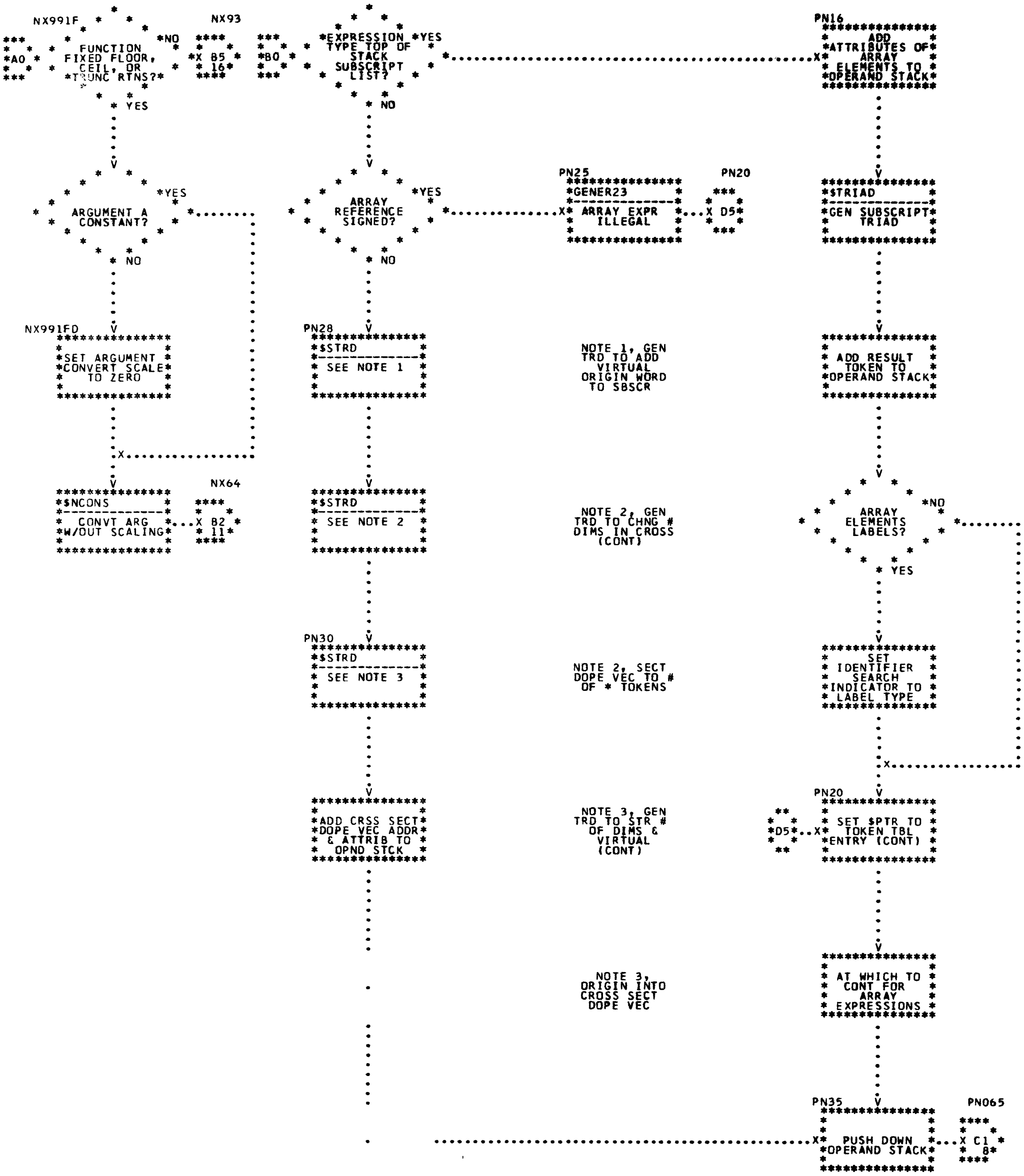


Chart 49. Operator Stack Processor (Page 17 of 17)

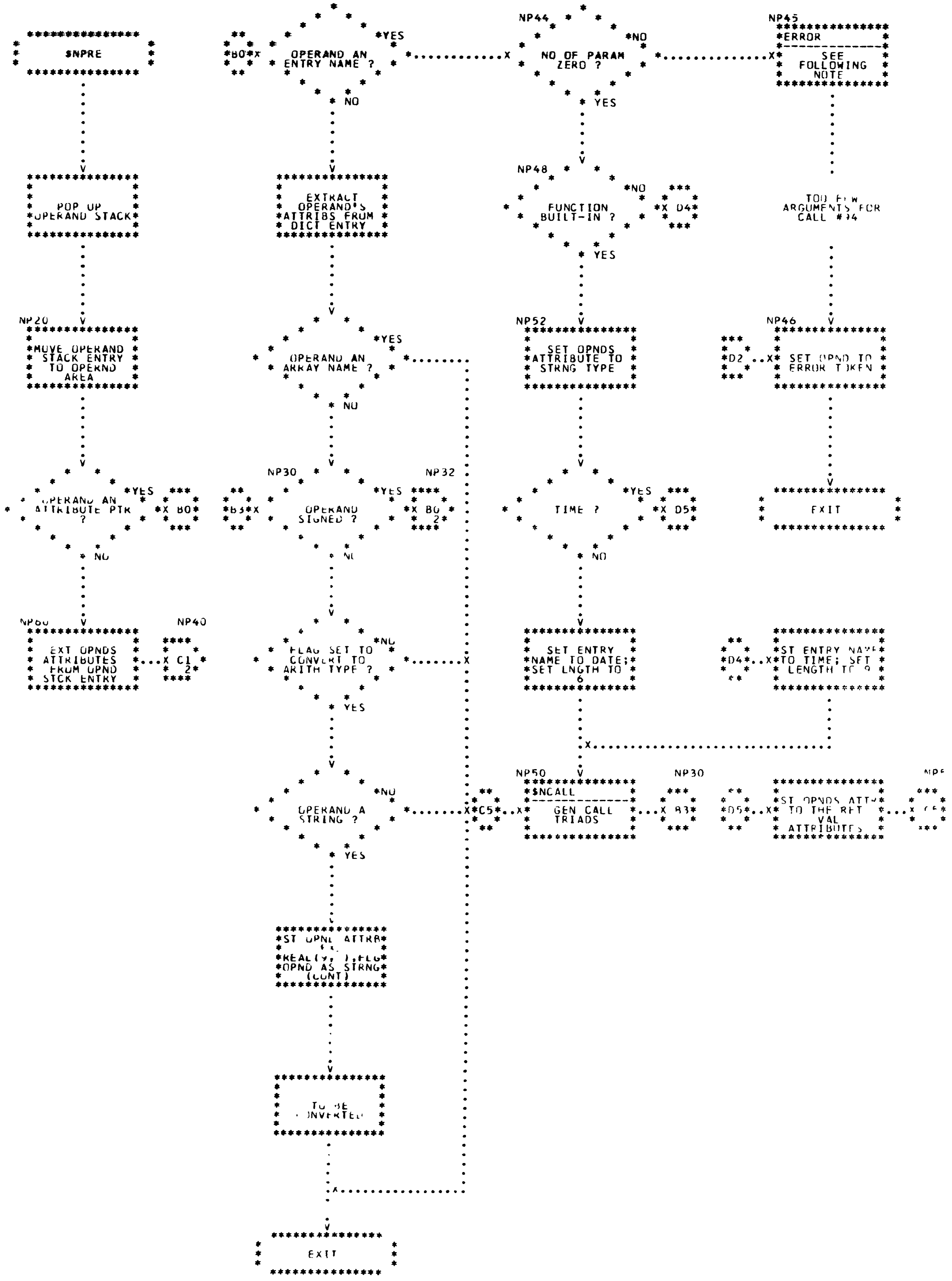


Chart 50. Operand Set-Up (Page 1 of 2)

NOTE 1, TYPE NOT CONVERTIBLE TO FIXED INTEGER #57

NOTE 2, CONVERT OPRND TO REQ'D TYPE

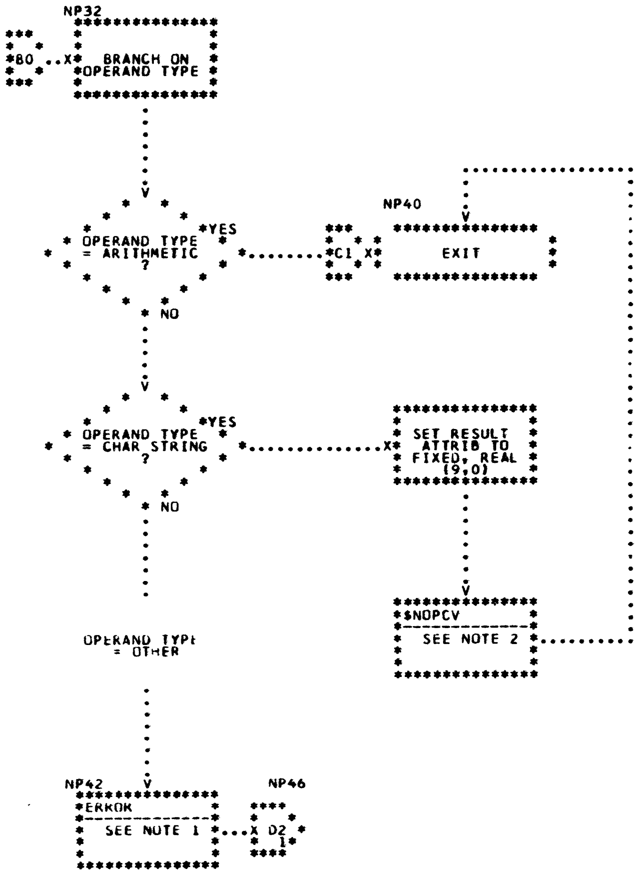


Chart 50. Operand Set-Up (Page 2 of 2)

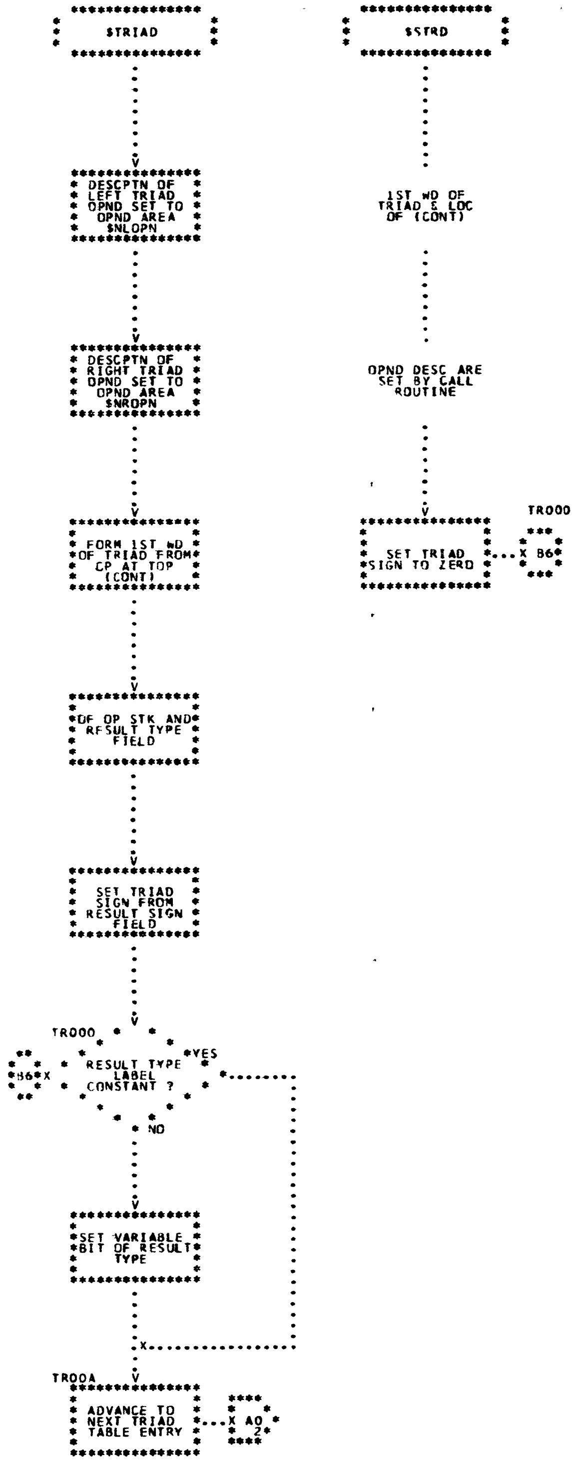


Chart 51. Generate Triad (Page 1 of 2)

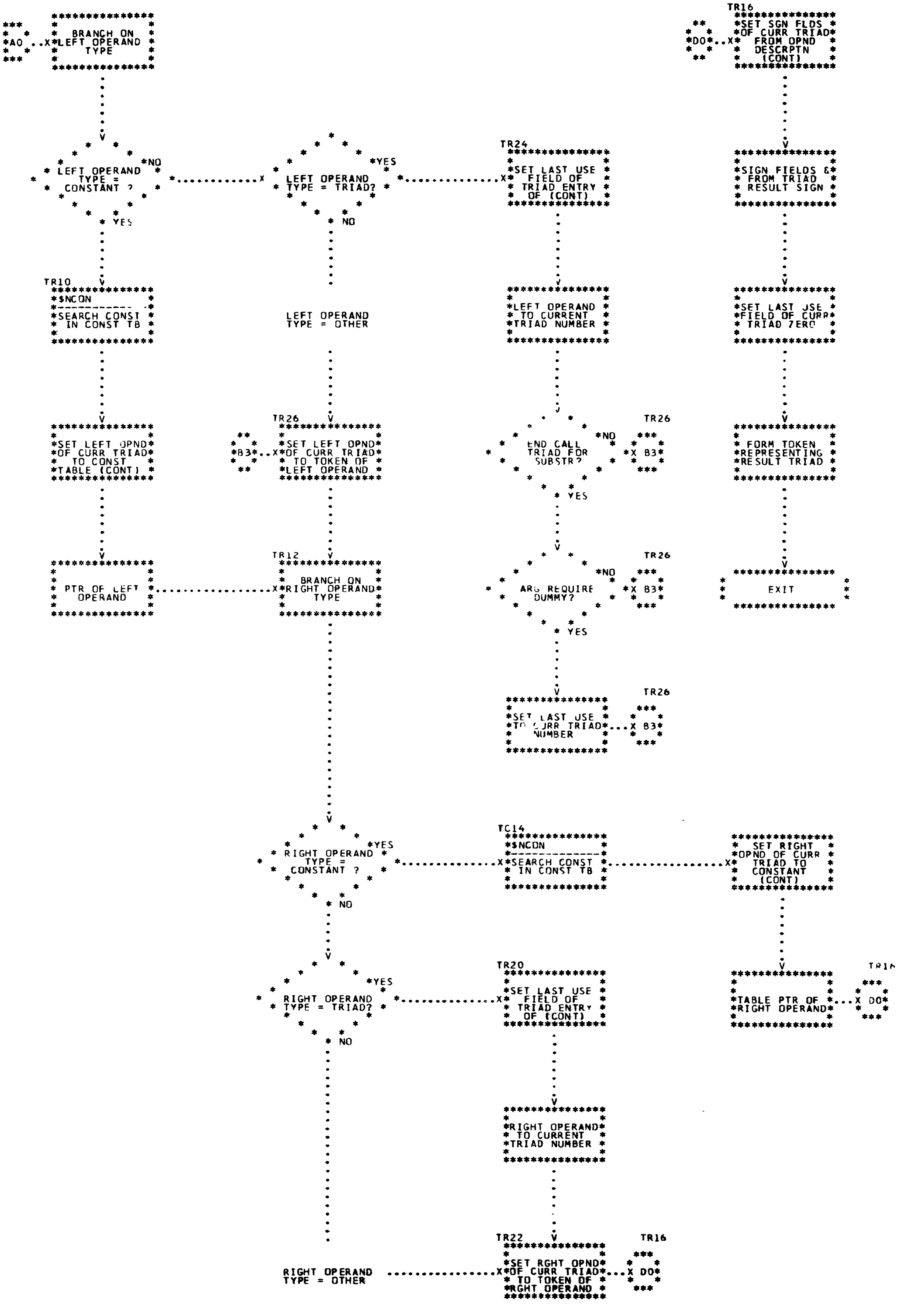


Chart 51. Generate Triad (Page 2 of 2)

PART 7 - CODE GENERATOR

The routines described in this subsection are concerned primarily with code generation. They are discussed in alphabetic order according to their mnemonics, as shown below. Detailed logic diagrams for the routines appear at the end of the subsection.

Optimize Operands ($OPMZO)
String Constant Dope Vector Initializer ($SCDV)
Triad Code Generator ($TCODE)
Triad Operand Processor ($TOPR)
Adcon Register Assignment ($VASGA)
Computational Register Assignment ($VASGC)
Register Table Initializer ($VCLR)
Storage Address Assembler ($VDSAC)
Temporary Storage Management ($VGTMP)
Instruction Assembler ($VINSA)

TITLE: OPTIMIZE OPERANDS ($OPMZO)

Program Definition

Purpose and Usage

The Optimize Operands routine is used to determine the effective sign of both triad operands and to arrange the order of the operands to optimize their referencing.

Description

This routine checks each operand to determine whether it is in a register. If so, it sets the effective sign for the operand to the operand's register sign. If the interchange flag is ON, the operands are ordered so that the left operand is in a register if either operand has a register address.

Errors Detected

None

Local Variables

None

Program Interface

Entry Point

$OPMZO. The operands are in the left and right operand areas. A flag is set to one if the operands may be interchanged to optimize their referencing; otherwise, it is zero.

Exit Conditions

Control returns to the caller immediately following the invoking call. Registers G5, G6, and G7 are destroyed. The flag is set to two if the operands were interchanged.

Routines Called

None

Global Variables

$NXFLG	Communication Flag
$PREG	Register Address of Operand
$PSGN	Operand Type Flag
$PTO	Origin of Table of Operands (Data Parameter Table)

Logic Diagram

Chart 52 shows the detailed logic diagram for the Optimize Operands routine.

TITLE: STRING CONSTANT DOPE VECTOR INITIALIZER ($SCDV)

Program Definition

Purpose and Usage

The String Constant Dope Vector Initializer routine forms the initialization table (I table) entry required to initialize the dope vector of a string constant.

Description

The address and length of the constant are obtained from its constant table (C table) entry. Eight bytes of word-aligned storage are obtained in the static and constants area for the dope vector. An initialization table entry is constructed for the initialization of the dope vector.

Errors Detected

None

Local Variables

None

Program Interface

Entry Points

$SCDV. Register G4 contains a pointer to the constant table (C table) for which the dope vector is formed.

Exit Conditions

Control passes to the instruction following the call. Register G4 contains an address word token representing the address associated with the string dope vector.

Routines Called

$WEXP	Segment Management

Global Variables

$ASC	Offset to Next Available Byte in Static and Constants Area
C Table	Constant Table
I Table	Initialization Table

Logic Diagram

Chart 53 shows the detailed logic diagram for the String Constant Dope Vector Initializer routine.

TITLE: TRIAD CODE GENERATOR ($TCODE)

Program Definition

Purpose and Usage

The Triad Code Generator controls the generation of code for the triads contained in the triad table. In general, this represents the code for a single statement.

Description

A triad with an end-of-table operator is placed in the triad table in the next available entry. Entries of the triad table are then processed in order starting with the first entry and continuing until the end-of-table entry is encountered. The triad operator is used to branch to the section of this routine which controls its processing. The types of triad operators and the numeric codes that identify them are listed below.

Null	0
End Triad Table	4
Begin Call	8
End Call	12
Convert	16
Argument LIst	28
Assignment Equal	32
Divide	36
Subtract	40
Compare	48,52,56 60,64,68
OR	72
AND	76
Multiply	80
Add	84
Subscript	88
Test Compare	92
Define Address	96
Branch	100
Combine or Resolve Statement Label	104
Backward Internal Branch	108
Forward Internal Branch	112
Resolve Forward Internal Branch	116
Store Address	120
Align	124
End Block	128
Prologue	132
Library Call	136
Symbol Table	140
Branch and Link	144
End Prologue	148
End DO	152
Begin DO	156
Return	160
Dope Vector Element Multiply	164
Dope Vector Store	168
Scale Complex Positive	172
Scale Complex Negative	176
Halfword Subscript	180
Store Code Address	184
Test Under Mask	188
Title	192
OR Immediate	196
Other (Error, because unused)	20,24,44

In general, processing of a triad consists of:

1. Determining the register and/or core address of the operands.
2. Determining the sign of the operands and the sign of the result of the execution of the triad code. Signs may be adjusted to optimize the required instruction sequence.
3. Arranging the operands, when possible, to insure optimum code generation.
4. Determining the entry of the symbolic instruction table (M table) to be used by the Instruction Assembler routine ($VINSA) in generating the code for the triad.

The triads with arithmetic operators, comparison operators, Boolean operators, and the assignment equal operator have two operands which agree in type, mode, and precision. Fixed-point operands are already scaled to the required scale. If the operands are strings, they may be treated as though they are both the length of the left operand. The type mask of the triad gives the attributes of the operands. The operand sign fields give the signs to be applied to the associated operands; the triad sign field gives the sign to be applied to the result of the triad.

The convert operator triad is used only for arithmetic to arithmetic conversions. The left operand indicates the source to be converted. The format of the right operand is shown below.

Byte	1	2	3	4
Word 1	0	Target Scale	Source Scale	Source Type

The triad type mask represents the target type. The operand sign fields are always zero. The triad sign field gives the sign to be applied to the result of the triad.

A subroutine call, either library or programmer-defined, is represented by a begin call operator triad, one argument list operator triad for each of its arguments and each of its required DED's, and one end call operator triad.

The begin call triad appears first in the triad table and has the indication of the entry point as the left operand and the number of argument list triads for the call in the right operand field. The begin call triad is immediately followed by the argument list triads for the call. The arguments and DED's appear in reverse order; that is, the last argument or DED appears first. The right operand of the last argument list triad points to the begin call triad; the right operands of all others point to the list element that immediately precedes it. The left operand is the argument or DED description. All arguments have been converted to the type required for the call. The sign of the argument is in the left operand sign field. This field also contains a flag to indicate whether the argument requires a dummy. The end call triad terminates the argument list. The right operand points to the last argument list triad (or to the begin call triad if there are no arguments). The left operand is the indication of the entry point. The triad sign field of this end call triad is the sign to be applied to the function result.

The subscript operator triad indicates a subscripted reference. The left operand indicates the array and the right operand the value of the subscript converted to a fixed integer. The sign of the subscript is in the right operand sign field. The sign to be applied to the subscripted reference is the resultant of the left operand sign and triad sign field.

The test compare operator triad immediately follows a triad containing a comparison operator. Often this triad is replaced with a triad with a branching operator. The right operand contains the operator appearing on the comparison. The left operand points to the comparison triad. The triad sign field, if negative, indicates that the greater than and less than tests must be reversed.

The branch operator triad is used to encode the GOTO statement. The left operand is a pointer to the attribute node of a statement-label constant or a statement-label variable or a pointer to a triad table entry of a subscripted label array reference. The right operand is a comparison operator or the indication of an unconditional branch. This operand is used to determine the condition code for the branch. The code to perform the branch (if no definition for the label precedes the reference in the current block) is provided by the Instruction Assembler.

The combine or resolve statement label operator triad is used for the definition of a statement label having branches which precede the definition and for labels having references within an inner block but no explicit declaration. In the first case this triad appears preceding the triads of the statement it prefixes. The code for the branches to the label is completed by the Instruction Assembler. In the second case this triad appears after the triads defining the body of the inner block. If the label is defined in the next outer block, this case is treated just as the first with the references resolved to the label of the other block; if not, the list of references to the label in the inner block is added to the list of references to the label in the outer block. The left operand points to the attribute node of a label in the dictionary attribute list. The right operand is an object code address or triad pointer to the chain of references to the label. The sign fields of the triad are ignored.

The backward internal branch operator triad is used to define a branch to a compiler-generated label which precedes the reference. The left operand is either a code address or a pointer to the triad associated with the code to be referenced. The right operand is a comparison operator or the indication of an unconditional branch. This operand is used to determine the condition code for the branch. The sign fields are ignored. This type of branch must be within the current block of code already generated; therefore, a branch on condition to the code is always generated immediately.

The forward internal branch operator triad is used to define a branch to a compiler-generated label which follows the reference. The left operand is a pointer to a word containing zero or a pointer to a triad which also references the same label. The right operand is either a comparison operator or an indication of an unconditional branch, or a pointer to a triad with a comparison operator. A negative right operand indicates that the greater and less than conditions of the comparison should be inverted. Since the code referenced by this branch has not been generated, maximum space for the branch on condition is reserved in the code area by the Instruction Assembler. The last word of this triad contains a pointer to either a triad or a word within a table. This field is used in forming the list of references to the same generated label.

The branch and link operator triad is used to establish linkage code to the code associated with an I/O format specification or to return from a subprogram. The left operand may be an absolute register specification, an attribute node pointer to a format label, a triad pointer to a subscripted label array reference, or a pointer to the triad associated with a format specification. The right operand and sign fields are ignored. Code is generated for the linkage with the return address placed in absolute register RP, the standard return point register.

The resolve forward internal branch operator triad is used to complete the code which branches to a generated label whose definition appears after references to the label. This triad appears at the point in the program flow at which the generated label is defined. The left operand contains an object code address to the first reference to the generated label to be completed. All other fields of the triad are ignored.

The define address operator triad is used to save the current code location counter in a table entry or in a triad. The left operand is either a table pointer or a triad pointer. All other fields are null.

The store address operator triad is used to load the address of an operand into a register or to store the address of an operand into an adcon. The left operand may be an absolute adcon register designation, an adcon area address, or an attribute node pointer. This operand indicates where the address is to be placed. If the triad sign is nonzero, its value is added to the address of the left operand. The right operand may be any type of operand. The right operand sign field applies to the operand.

The align operator triad is used to align the code location counter to a fullword. After the alignment is performed, the operator performs the same function as the define address operator triad. The operands are the same as for the define address.

The end block operator triad is used to complete the chain of code generated for the block prologue (that is, generates a branch from the last section of prologue code to the first source statement code) and to generate the epilogue code. The left operand is the pointer to the block information table (B table) entry for the block. All other fields are ignored.

The prologue operator triad is used to generate the call to the Initial Prologue, Expand DSA, End Prologue, Object Program Initiation routine (IHESADA). This routine performs the function of opening the block for execution. It saves all necessary registers, establishes the fixed-length part of the dynamic storage area, and sets up ON-condition action. The left operand is the pointer to the block information table entry associated with the block.

The library call operator triad is used to generate the linkage to a standard nonbuilt-in function. The left operand identifies the entry point to be called. The right operand, if not null, indicates the address of the parameter list. All other fields are ignored.

The symbol table operator triad is used to generate entries for the runtime symbol table required for elements of the DATA I/O lists. This triad causes one symbol entry or an end-of-table entry to be generated. The left operand is a pointer to the dictionary attribute node for the identifier to be added to the symbol table of the data list. The right operand is the dictionary name list pointer of the identifier; a null pointer indicates that an end-of-table entry is to be generated. (For the format of symbol table entries, see Appendix E.)

The end prologue operator triad is used to initialize tables for the generation of the chain of code required to evaluate variable lengths and bounds, to allocate variable-length data areas, and to initialize dope vectors for the variables of the block. Space is reserved in the code area sufficient to branch to the first element of this chain. The address of the first byte of code for the block is saved so that this prologue chain may be completed when the end of the block is reached. Code is generated which closes the block's prologue.

The left operand is the pointer to the block information table associated with the block. All other fields are null.

The begin DO and end DO operator triads are used to define the code bounds of an iterative DO-group. The Instruction Assembler processes the register table (R table) to generate code to store the expression values contained in registers which still have active usages. These values must be retained in their temporary storage until the associated end DO is reached. All operand fields of these triads are ignored.

The return operator triad is used to generate code to return from a subprogram. All operand fields of the triad are ignored.

The dope vector element multiply operator triad is used to form the dope vector for an array cross-section. Code is generated to form the product of the two operands. The right operand always refers to a multiplier from an array dope vector. The operand field is the dictionary attribute node pointer of the array. The right operand sign field indicates the dimension number of the multiplier required. If the left operand is an attribute node pointer, it too refers to a dimension multiplier from the dope vector and its sign field indicates the dimension number of that multiplier. Dimension multipliers are halfword quantities. In all other cases, the left operand is a fullword value as specified by the operand and its sign field has the usual meaning.

The dope vector store operator triad is used to form the dope vector for an array cross-section. Code is generated to store the right operand into the halfword left operand. If the right operand is a dictionary attribute node pointer, the operand refers to a halfword field of an array dope vector. The operand field is the attribute node pointer of the array. The right operand sign field indicates the dimension number of the dope vector element required. The sign field of the triad indicates whether a multiplier (4), upper bound (0), or lower bound (2) is required. The left operand is always an address in static storage of a halfword dope vector element.

The scale complex positive and scale complex negative operator triads are used to scale fixed-point complex operands. The left operand defines the complex operand; the right operand, the constant table (C table) pointer to the power of ten by which the left operand is to be multiplied or divided. The sign fields and the type mask have the usual meanings for arithmetic operations.

The halfword subscript operator triad is used to generate DO-loops for the expansion of array expressions. This triad is used to refer to the upper and lower bound fields in an array dope vector. The left operand is the dictionary attribute node pointer to the array. The right operand is the offset to the dope vector element within the dope vector. All sign fields are always plus.

The store code address operator triad is used to generate code to store the current value of the code location counter into an adcon for use by the Error Routine (IHEERR) at runtime. All operand fields of the triad are ignored.

Errors Detected

CONVERSION ERROR--SCALE FACTOR TOO LARGE. (84)
COMPILER ERROR. (100)

Local Variables

None

Program Interface

Entry Point

$TCODE. The pointer to the current triad table entry is pointing to the last triad for which code is to be generated.

Exit Conditions

Control returns to the caller immediately following the invoking call except when the size of the user area is too small to continue. In this case an SVC 6 is given and control does not return. Registers at exit are the same as at entry; no specific output values are returned.

Routines Called

$NCONS	Constant Processor
$NLSIB	Library Search
$OPMZO	Optimize Operands
$SCDV	String Constant Dope Vector Initializer
$SVC	SVC Director
$TOPR	Triad Operand Processor
$VGTMP	Temporary Storage Management
$VINSA	Instruction Assembler
$WBACK	Segment Management
$WCTCT	Segment Management
$WEXP	Segment Management
$WSTEP	Segment Management
$NCVT	Constant Conversion
$XERR	Error Message Editor

Global Variables

$ACODE	Pointer to Next Available Byte in Object Code Area
$ASC	Offset to Next Available Byte in Static and Constants Area
$COMAD	Address of Communications Area
$ERROR	Parameter List for Error Message Editor
$GBQF	Table of Switches
$NBIF	Built-In Function Flag
$NCCUR	N List Current-Entry Pointer
$NLINE	New Line Flag
$NPVF	Complex Pseudo-Variable Flag
$NTCUR	Number of Last Triad Generated
$NXFLG	Communication Flag
$PADD	Core Address of Operand
$PARAM	Address of Library Parameter List
$PLNG	String Length of Operand
$PREG	Register Address of Operand
$PSGN	Operand Type Flag
$PTKN	Operand Token
$PTR	Token Table Pointer
$SEVCT	Total Number of Error Messages Produced

$TAILS	Table of Pointers to End-of-Segment Control Word for Expandable Tables
$VLPAK	Doubleword-Aligned Work Area
FREPTR	Pointer to First Available Word in Compiler's Variable Data Area
NOERMSG	Number of Error Messages (Communications Area)
A List	Dictionary Attribute List
B Table	Block Information Table
C Table	Constant Table
N List	Dictionary Name List
M Table	Symbolic Instruction Table
O Table	Operation Code Table
R Table	Register Table
Z Table	Triad Table

Logic Diagram

Chart 54 shows the detailed logic diagram for the Triad Code Generator routine.

TITLE: TRIAD OPERAND PROCESSOR ($TOPR)

Program Definition

Purpose and Usage

The Triad Operand Processor routine processes the two operands of a triad to determine the core address, core sign, register address, register sign, and length of each. This information is placed in the adjacent operand areas indicated by register P4.

Description

Both operands of the current triad are processed. The type identification of an operand is used to determine where its addressing, sign, and length information can be obtained. This information and the operand token are stored in a data parameter table. The left operand is placed in the first entry of the table; the right operand in the second. (For a description of this table, see $PTO under "Compiler Variables" in Appendix A.)

Errors Detected

None

Local Variables

TOFLG	If zero, the left operand is being processed; otherwise, the right.

Program Interface

Entry Points

$TOPR. The address of the triad is in register P5. The address of the operand area in which to store the left operand is in register P4.

Exit Conditions

Control returns to the caller immediately following the invoking call. All G and P registers other than P5 may be destroyed.

Routines Called

$NLSIB	Library Search
$NCONS	Constant Processor

Global Variables

$CBKNO	Current Block Number
A List	Dictionary Attribute List
C Table	Constant Table
Z Table	Triad Table

The follow fields of the data parameter table are set.

$PADD	Core Address of Operand
$PLNG	String Length of Operand
$PREG	Register Address of Operand
$PSGN	Operand Type Flag
$PTKN	Operand Token

Logic Diagram: See Chart 55.

TITLE: ADCON REGISTER ASSIGNMENT ($VASGA)

Program Definition

Purpose and Usage

The Adcon Register Assignment routine searches the adcon register portion of the register table (R table) and selects a register for assignment.

Description

The adcon register portion of the register table contains an entry for each adcon register used in the object program. Some of the registers are permanently assigned and cannot be selected by $VASGA. A volatile register may be specifically inhibited or have a symbolic assignment, in which case it will not be selected. A simple reference count is associated with each register and effectively causes the available registers to be assigned in rotation.

Errors Detected

None

Local Variables

R$ARRC	Reference count for adcon register assignment. This count is increased by one and associated with the selected register each time a register is assigned. In choosing a register, the one with the lowest previous count is selected.

Program Interface

Entry Points

$VASGA. No formal parameters.

Exit Conditions

Control returns to the caller immediately following the invoking call. Register G2 contains a pointer to the register table entry for the selected register.

Routines Called

None

Global Variables

R$AD	Pointer to Adcon Register Portion of Register Table
R Table	Register Table

Logic Diagram

Chart 56 shows the detailed logic diagram for the Adcon Register Assignment routine.

TITLE: COMPUTATIONAL REGISTER ASSIGNMENT ($VASGC)

Program Definition

Purpose and Usage

The Computational Register Assignment routine searches the computational register portion of the register table (R table) and selects the best register or pair of registers for assignment.

Description

The computational register portion of the register table is divided into two parts, one for fixed-point registers and the other for floating-point registers. The type of register to be obtained is determined by a flag set by the calling routine. Registers are arranged in pairs of even-odd numbered addresses. Either a single or a double register may be assigned; the option is indicated by a flag set by the calling routine.

The register table entries consist of a principal assignment and a list of synonyms. (See the description of the Instruction Assembler ($VINSA) for a discussion of synonyms.) The principal assignment may be a constant, variable, or intermediate result (indicated by a triad pointer). In selecting registers, every effort is made to avoid choosing one that contains an intermediate result which has yet to be used. To aid in this determination, every triad contains a "last-use" count -- the number of the triad after which the intermediate result will no longer be needed. If the last-use count associated with a register is not greater than the number of the current triad for which code is being generated, the register contains active intermediate results and receives a low priority in the selection algorithm. Of course, selection of an active register may be unavoidable. When this condition occurs, $VASGC generates object coding to store the contents of the register into temporary storage and adjusts the associated triad to point to the storage assigned for it.

In selecting a register, each entry in the appropriate part of the table is examined (every second entry, if a double register is wanted) and assigned a priority code. A register meeting the requirements for priority 7 is selected at once; otherwise the entire table is searched and the register with the highest priority is selected. If two registers have the same priority, that with the higher last-use count is selected. (For a double register pair, the last-use count used for comparison with other registers is the lower count of the pair.)

The priorities are assigned as follows:

1. A single register is needed, and the examined register is an active double register.
2. A double register is needed, and the examined register is an active double register.
3. A double register is needed, and, of the examined pair, one register is active, the other is free, and either or both registers have synonyms.
4. A double register is needed, and, of the examined pair, one register is active, the other is free, and neither has synonyms.
5. A double register is needed, and the examined pair is free, but both registers have synonyms.

6. A double register is needed, the examined pair is free, and one of the two registers has synonyms.

7. The examined register is a free register of the required size (single or double) and has no synonyms.

$VASGC has three subordinate entry points which perform limited services for the calling routine:

1. $VRSYN removes the synonyms from a designated register table entry.

2. $VSAVE stores the contents of a designated register into temporary storage.

3. $VFREE examines a designated register to determine whether it can be made free.

Errors Detected

None

Local Variables

VSFLG	Increment to second address when storing double registers into temporary.
VSOP	Operation code to be used when storing registers into temporary.
VSSAVE	Storage for register C1 on internal subroutine calls.
VSTEMP	Miscellaneous work area.
VSX	Currently assigned selection priority.
VSY	Last-use count of currently selected register.

<u>Program</u> <u>Interface</u> - $VASGC

Entry Points

$VASGC. High-order bit of $VRTYP set to zero for floating-point register selection, to one for fixed-point; $VRAMT set to zero for single register selection, to nonzero for double.

Exit Conditions

Control returns to the caller immediately following the invoking call. Register G2 contains a pointer to the register table entry for the selected register.

<u>Program</u> <u>Interface</u> - $VRSYN

Entry Points

$VRSYN. Register G2 contains a pointer to the register table entry from which synonyms are to be removed.

Exit Conditions

Control returns to the caller immediately following the invoking call. Registers are the same as at entry.

<u>Program</u> <u>Interface</u> - $VSAVE

Entry Points

$VSAVE. Register G2 contains a pointer to the register table entry whose contents are to be stored into temporary storage.

Exit Conditions

Control returns to the caller immediately following the invoking call. Registers are the same as at entry.

Program Interface - $VFREE

Entry Points

$VFREE. Register G2 contains a pointer to the register table entry which is to be examined for possible freeing.

Exit Conditions

Control returns to the caller immediately following the invoking call. Registers are the same as at entry. The condition code is set to zero if the register could not be freed; to nonzero if the register was freed.

Routines Called

$VDSAC	Storage Address Assembler
$VGTMP	Temporary Storage Management

Global Variables

$ACODE	Pointer to Next Available Byte in Object Code Area
$VRAMT	Flag for Register Assignment
$VRTYP	Flag for Register Type Assignment
R$FL	Pointer to Floating-Point Register Portion of Register Table
R$FX	Pointer to Fixed-Point Register Portion of Register Table
R$SY	Pointer to Head of Register Table Synonym List
$NCCUR	N List Current-Entry Pointer
Z Table	Triad Table
A List	Dictionary Attribute List
C Table	Constant Table
R Table	Register Table
O Table	Operation Code Table

Logic Diagram

Chart 57 shows the detailed logic diagram for the Computational Register Assignment routine.

TITLE: REGISTER TABLE INITIALIZER ($VCLR)

Program Definition

Purpose and Usage

The Register Table Initializer routine allocates space for the register table and establishes its initial values.

Description

Space for the register table is deducted from the free area (FREPTR). Initial values for the fixed, floating, and adcon register portions of the table are moved into the proper entries. The synonym list is linked together.

Errors Detected

None

Local Variables

None

Program Interface

Entry Points

$VCLR. Negative value in register G0 to cause initialization. A positive value will cause resetting of the table to its initial condition (without space procurement). This feature is not presently used by the compiler.

Exit Conditions

Control returns to the caller immediately following the invoking call.

Routines Called

None

Global Variables

R$TBL	Pointer to Base of Register Table
R$FX	Pointer to Fixed-Point Register Portion of Register Table
R$FL	Pointer to Floating-Point Register Portion of Register Table
R$AD	Pointer to Adcon Register Portion of Register Table
R$ND	Pointer to End of the Linear Register Portion of Register Table
R$SY	Pointer to Head of Register Table Synonym List
FREPTR	Pointer to First Available Word in Compiler's Variable Data Area
R Table	Register Table

Logic Diagram

Chart 58 shows the detailed logic diagram for the Register Table Initializer routine.

TITLE: STORAGE ADDRESS ASSEMBLER ($VDSAC)

Program Definition

Purpose and Usage

The Storage Address Assembler routine converts the compiler's internal representation of a storage address (base code and displacement) into a machine address (base register number and twelve-bit displacement). It generates any instructions needed to load adcon registers to accomplish the final addressing.

Description

The storage address to be converted may be one of four types:

1. Directly addressed data
2. Parameter data
3. Subscripted address
4. Literal displacement

Directly addressed data may be allocated to one of five classes:

1. Static and constants storage.
2. Address constant storage.
3. Object code storage.
4. Automatic data storage for the block currently being compiled.
5. Automatic data storage for a block encompassing the one currently being compiled.

At object program execution, one register is permanently allocated to address the first page of static and constants storage; another, the first page of address constant storage; a third, the first page of automatic storage for the current block; and three more to address the first three pages of object code. Data allocated to these areas can be addressed with the appropriate permanent register; no supplementary instructions are needed. (For a description of object code storage layout, see Appendix E.)

Data allocated to an address beyond the range of the permanent registers is addressed by generating an instruction which loads one of the volatile adcon registers with a displacement of 4095 from the permanent register. If this base is still insufficient, additional instructions are generated to increment the base by 4095 until the displacement of the data relative to the base is less than 4096. The assigned adcon register and the calculated displacement are then used as the assembled machine address.

Data allocated to automatic storage in an encompassing block has no register permanently assigned for it. However, every block has a section reserved for its own use in the adcon storage area. If a block is active, one word of this area will contain the address of the base of the block's dynamic storage area. $VDSAC will generate instructions to load one of the volatile adcon registers from this word, thus obtaining a starting base in the encompassing block's DSA. From this point, calculations proceed as described above to determine an assembled machine address with a displacement less than 4095. Note that in addressing the block information in the adcon area, extra instructions may be required to determine a suitable displacement for the "load" instruction.

Parameter data is first processed as if it were directly addressed data in the adcon storage area. Having determined an assembled address for the data, $VDSAC then generates an instruction which loads one of the volatile adcon registers with the contents of the computed address. This technique introduces the extra level of indirectness required to address parameter data. The assembled machine address which is returned to the caller is then set to a displacement of zero relative to the assigned adcon register. A special flag is set by the caller to indicate whether $VDSAC is to interpret a storage address as parameter data. $VDSAC enters a special flag into the adcon register table entry for the volatile register, indicating that the register contains a parameter. This enables subsequent references to the same parameter to be optimized by using the register already loaded.

Subscripted addresses are presented in two ways: the computed subscript is currently in a register, or the computed subscript has been stored in temporary storage. In addition, the subscripted address can be used in one of two ways: in an RX instruction, where two indexes are available; or in an SS instruction, where only one register can be used.

The subscript value itself represents an increment to the value assigned to the base of the object code area. This base value is permanently loaded at object time in code-cover register 1 (register 6). Accordingly, $VDSAC must always combine the subscript value with the value of register 6 to assemble a valid machine address. If the subscript is in a register and the instruction is an RX format, the machine address is computed as a displacement of zero, base register of 6, and index register (X1 field) of the assigned subscript register. If the subscript is in a register and the instruction is SS format, $VDSAC generates a "load address" into one of the volatile adcon registers, which combines register 6 and the subscript register. The machine address is then assembled as a displacement of zero relative to the adcon register.

If the subscript is in temporary storage, regardless of instruction format, $VDSAC first computes the assembled address of the temporary storage as if it were directly addressed data. Then it assigns an adcon register, r, and generates the following code:

```
    LR     r,6
    A      r,temporary-storage
```

The assembled machine address is then a displacement of zero relative to the adcon register.

String data are indirectly addressed through a dope vector. The addressing problem is therefore almost identical to that of using a subscript which has been placed in temporary storage, except that the dope vector is used instead of temporary storage, and the assembled address computed for the dope vector may be a parameter and require the extra level of indirectness. However, logic used for processing string addresses is substantially the same as that used for stored subscripts.

Occasionally, code generation computes in advance the literal value of the displacement needed in a machine address. This condition is indicated by a base code of zero on the storage address. In this case, $VDSAC merely returns the literal displacement with a base register of zero as the assembled machine address.

Errors Detected

None

Local Variables

VDWORK	Miscellaneous Temporary Storage

Program Interface

Entry Points

$VDSAC. Parameter flag (from $PTO table) in symbolic bit #PRAM in the low-order byte of register G3. Storage address (base code and displacement) in register G2.

Base code 0 indicates literal displacement; base code X'FF' indicates subscript in storage; base code X'FE' indicates subscript in register (register address in low-order byte of G2). For base codes X'FF' and X'FE', local variable VNOPC (in $VINSA) must contain the operation code for the current instruction (used to determine SS format). For SS format instructions, local variable VNDVA (in $VINSA) must contain zero or the true base code assigned to the dope vector, since the incoming base code was set to 'FF' to trigger subscript addressing logic.

Exit Conditions

Assembled machine address in register G0 in the form 000XBDDD, where X is the index register, B the base register, and DDD the displacement. Register G2 points to the register table entry for the base register, B.

Routines Called

$VASGA	Adcon Register Assignment
$WBACK	Segment Management

Global Variables

$ACODE	Pointer to Next Available Byte in Object Code Area
B$ACTV	B Table Active-Entry Pointer
B$CURR	B Table Current-Entry Pointer
R$AD	Pointer to Adcon Register Portion of Register Table
B Table	Block Information Table
R Table	Register Table
O Table	Operation Code Table

Logic Diagram

Chart 59 shows the detailed logic diagram for the Storage Address Assembler routine.

TITLE: TEMPORARY STORAGE MANAGEMENT ($VGTMP)

Program Definition

Purpose and Usage

The Temporary Storage Management routine allocates an area for temporary storage at the proper level within the dynamic storage area of the current block. (For a description of DSA's, see Appendix E.)

Description

The caller supplies the length of the area desired, which may be 4, 8, 16, 32, or 255 bytes, and the number of the triad after which the temporary storage may be reused (last-use count). Temporary storage is assigned separately for each block, in the dynamic storage area for that block; in the external procedure, temporary storage is assigned in the static and constants storage area. Within a block, temporary storage is further assigned a level whenever it is used to save values around DO-loops. In addition, temporary storage areas which are no longer required are reusable so that total storage requirements can be minimized.

A temporary storage table (S table) is organized and maintained by block. For each block, entries in the table indicate the length, level, allocated address, and last-use count for all available temporary storage areas. When $VGTMP is called, this table is searched backwards from the most recently added entry. The search is conducted for a previously allocated temporary which is the same size as (or larger than) the requested area, at the requested level within the current block, and whose last-use count indicates that the temporary storage is free for reuse. The search is organized so that if multiple candidates meet the selection criteria, the one which is closest to the requested size (but not smaller) is used. The last-use count of the selected area is updated to the new value and the allocated address is returned to the caller.

If no entries meet the selection criteria, a new temporary storage must be created. The DSA address for the required block is aligned to a word boundary if the requested length is 4 or to a doubleword boundary for any other length. It is incremented by the requested length. A new entry is added to the temporary storage table, and the length, allocated address, level, and requested last-use count are established. The allocated address is then returned to the caller.

Errors Detected

None

Local Variables

VTCAND	Used to hold table pointer during determination of best matching length.

Program Interface

Entry Points

$VGTMP. Length of desired area in register G0. May be 4, 8, 16, 32, or 255. When the length given is 255, 256 bytes are actually reserved. The last-use count, which controls when the temporary area can be reused, is supplied in register G7. The requested level number is supplied in $TEMPN and is normally 0 except for register saving around DO-loops.

Exit Conditions

Control returns to the caller immediately following the invoking call. Register G2 contains the base code and displacement of the allocated temporary storage.

Routines Called

$WBACK	Segment Management
$WEXP	Segment Management

Global Variables

$TEMPN	Temporary Storage Level Number
$NCCUR	N List Current-Entry Pointer
B$ACTV	B Table Active-Entry Pointer
S$ACTV	S Table Active-Entry Pointer
S$CURR	S Table Current-Entry Pointer
S$TAIL	S Table End-of-Table Pointer
B Table	Block Information Table
S Table	Temporary Storage Table

Logic Diagram

Chart 60 shows the detailed logic diagram for the Temporary Storage Management routine.

TITLE: INSTRUCTION ASSEMBLER ($VINSA)

Program Definition

Purpose and Usage

The Instruction Assembler routine produces machine-language (object) code from a series of symbolic instructions which have been selected by the Triad Code Generator ($TCODE). Each set of symbolic instructions represents the coding necessary for the processing of a triad.

Description

The symbolic instructions selected by the Triad Code Generator represent in a general way the coding needed to process a triad. It is the function of the Instruction Assembler to interpret, modify, and in some cases optimize the generalized symbolic coding into the specific machine-language instructions required for each case. By maintaining the symbolic instructions at a generalized level, the Triad Code Generator minimizes both the space required for the symbolic instruction table (M table) and the number of processing paths needed for code generation.

Symbolic Instructions

A symbolic instruction consists of a symbolic operation code and zero, two, or four symbolic operands. At entry, the Instruction Assembler receives a pointer to the first symbolic instruction to be processed; the Instruction Assembler proceeds through the table, generating machine-language coding for each instruction encountered until it finds a special operation which signals the end of the instruction sequence.

Symbolic Operation Codes

The first field of a symbolic instruction is a symbolic operation code, which is a simple numerical index into a table of symbolic operations. Each symbolic operation contains a flag field, an effect code, and an operation code. The symbolic operations are divisible into three types: unmodifiable machine codes, modifiable machine codes, and pseudo-operations.

Unmodifiable Machine Codes: If the symbolic operation is an unmodifiable machine code, the Instruction Assembler uses the hexadecimal operation code found in the table as the operation code of the machine instruction being generated.

Modifiable Machine Codes: A modifiable operation may be modified in either or both of two ways, as indicated by settings of bits in the flag field of the operation code table (O table) entry. First, the operation may be modified for fixed-point or floating-point mode (either long or short). Secondly, it may be modified from the RX (storage-register) format to the RR (register-register) format. Modifiable machine codes are kept in the table in double-precision floating-point form. The symmetrical structure of System/360 machine codes makes it possible to convert the operation to the required form by the addition or subtraction of simple constants.

When modifying for type, the Instruction Assembler determines whether the mode shall be long-float, short-float, or fixed by testing the attribute codes in the triad for which code is being generated. (The location of the triad is supplied by the Triad Code Generator at entry to the Instruction Assembler.)

When modifying for format, the Instruction Assembler assumes that the code generation process keeps track of what the contents of machine registers will be at execution time. Each constant, variable, and intermediate result has associated with it a field which indicates whether it is (at that point in the compilation) in a register and which register it is in. Whenever the machine code is modifiable for RR format and the second (storage) operand of the instruction has a register associated with it, the Instruction Assembler modifies the operation code to RR format. If the resulting instruction is one which loads a register with itself without change of sign, generation of the instruction is suppressed.

Pseudo-Operations: Certain instruction sequences are too specifically machine-oriented or too complex to represent conveniently in the symbolic instruction tables. For these cases, pseudo-operations have been defined by convention between the Triad Code Generator and the Instruction Assembler. The symbolic operands of the pseudo-operations are defined to fit the needs of each case and cause the Instruction Assembler to generate specially tailored coding. Detailed descriptions of pseudo-operations are given later in this discussion under "Pseudo-Operation Logic".

Register Effects

It is the responsibility of the Instruction Assembler to maintain, during compilation, a table which indicates the contents of the general machine registers as they will be during execution of the object program. The availability of this information enables both the Triad Code Generator and the Instruction Assembler to select optimum sequences of instructions when generating code for a given triad. To assist the Instruction Assembler in the maintenance of the register table, each symbolic operation contains a field which indicates the effect of the operation upon the register designated in the R1 field of the instruction. The effects are:

Load
Destroy
Multiply
Divide
Store
Destroy all register synonyms
Change sign
No effect

A register synonym results when a variable, intermediate result, or constant is assigned to a variable. Thus, for the statement:

A = B + C

a given register will contain the intermediate result B + C. When this result is stored into A, the same register will then also represent the value of A. In subsequent instructions, the register may be used in lieu of a storage reference to A, with resulting optimization of the code. The variable A is considered a synonym of the register until such time as the register contents are changed.

The same register may have multiple synonyms. In the sequence of statements:

A = B + C
D = A

both A and D are synonyms of the register containing the intermediate result B + C.

It is the function of the Instruction Assembler to add and remove synonyms from the register table, as indicated by the "effect" code of the instructions being generated.

When a synonym is being added, a pointer to the synonymous constant, identifier, or triad is chained to the principal register table entry. The address of the register is placed in a field in the triad, constant table, or identifier dictionary attribute node. When a synonym is being removed, the pointer is detached from the register table chain. The register address field in the entry pointed to is set to zero. Similar techniques are used to maintain the principal contents of the register. In general, triads which represent intermediate results are used as the principal register entry, and identifiers and constants are maintained as synonyms of the principal entry.

The various effects are processed as follows:

Load: The previous principal content of the register and all its synonyms are removed. The triad, constant, or identifier which forms the second operand of the current instruction is attached to the register as a synonym. (The principal content of the register is not set until a result pseudo-operation occurs; see below.)

Destroy: The principal contents and all synonyms are removed from the register. No new information is added.

Multiply: A fixed-point multiply operation requires a pair of registers. The useful portion of the resulting product is in the odd-numbered register. However, any overflow resulting from the operation must be detectable. Accordingly, a multiply effect causes the Instruction Assembler to generate a double-register left shift of 32, and to remove all synonyms from the even-numbered register, since the multiplication destroys the previous contents.

Divide: A fixed-point divide operation requires a pair of registers and leaves the quotient in the odd-numbered register. A divide result accordingly causes synonyms to be removed from both registers; the double-register pair is flagged as now consisting of two single, independent registers, and pertinent information is transferred from the even to the odd register table entry. Any symbolic assignment (see "Symbolic Operands," below) is associated with the odd register. Floating-point multiply and divide operations are handled as if they had a simple destructive effect, since the double register considerations which affect fixed-point operations do not apply.

Store: A store effect causes the removal of any synonyms previously associated with the register; the triad, constant, or identifier which forms the second operand of the current instruction is attached to the register as a new synonym.

Destroy All Register Synonyms: At certain points in the object code, the contents of registers during execution become unpredictable to the compiler. This may happen where a label (either programmer- or compiler-generated) is defined, indicating a point where object-time control may be transferred, with resulting uncertainty of register contents; around calls to the object-time library, where certain registers are volatile; and at the end of program blocks, where the subsequent instructions belong to a different block with different register usages. Synonyms are removed from all registers at such points to insure correct code generation. A special effect code is used to indicate this condition.

Change Sign: A register may be complemented in order to effect an assignment. In this case, any synonyms of the register also change sign. For example:

```
A = B + C
D = -A
E = A
```

The register synonym for A may be used for the third assignment statement, but a record must be made of the fact that its sign was changed at the second statement. Triads and identifier attribute nodes have a field which indicates the sign of their synonym registers. A change-sign effect causes the setting of this flag to be reversed. Signs are not associated with constants; therefore a change-sign effect applied to a constant synonym causes the synonym to be removed.

No Effect: To increase the efficiency of the compiler, many intermediate operations in an instruction sequence are flagged as having no effect, since the intermediate effects are of no consequence in optimizing the generated code. By indicating no effect, unnecessary manipulation of the register table is avoided.

Indexable Operations

A flag bit in the operation code table (O table) indicates whether the operation is to be indexed. Symbolic instructions which use unindexed operation codes have two operands; symbolic instructions which use indexable operation codes have four operands. The third operand is a symbolic representation of the value to be placed in the X1 field of the generated instruction. The fourth operand represents the value to be placed in the B1 field of the generated instruction.

Symbolic Operands

A symbolic operand used in a symbolic instruction may be one of the following types:

Symbolic Computational Register
Symbolic Adcon Register
Absolute Register
Data Parameter
Imaginary Address of Data Parameter
Scratch Storage
Constant
Literal Value
Local Label Reference
Null

Symbolic Computational Register: To preserve generality, symbolic instruction sequences which perform computational arithmetic refer to registers symbolically. It is the responsibility of the Instruction Assembler to assign absolute registers to correspond to the symbolic ones. To accomplish this, the Instruction Assembler maintains a symbolic computational register table. This table is reset upon initial entry to the Instruction Assembler. The first reference to a symbolic computational register causes the Instruction Assembler to look up the corresponding entry in the table. Finding that no assignment has been made, the Instruction Assembler calls the Computational Register Assignment routine ($VASGC) to obtain an absolute register (or pair of registers, if required) and associates the absolute assignment with the symbolic register number. Thereafter, any reference to a given symbolic register always obtains the associated absolute register. The assignment is maintained until the end of the current symbolic instruction sequence. Upon reentry to the Instruction Assembler, new assignments will be made. The algorithm for determining which of the six computational registers to use is part of the Computational Register Assignment routine ($VASGC).

Symbolic Adcon Register: Symbolic instruction sequences which compute subscripts, develop arguments for subroutine calls, etc., require the use of adcon registers. For generality, these registers are referred to symbolically. The Instruction Assembler maintains a symbolic adcon register table similar in use and function to the one described above for computational registers. Because the object program must be instantly relocatable, computational and adcon registers cannot be interchanged. They require separate tables and separate symbolic notation. The Adcon Register Assignment routine ($VASGA) is used to obtain the required adcon register number.

Absolute Register: A symbolic operand may specify directly which register to use by presenting the Instruction Assembler with an absolute register address. This address is actually the displacement from the base of the register table (maintained in R$TBL) to the desired register entry. For communication between routines, register (table entry) addresses are used rather than the actual numbers of the registers. This allows a degree of flexibility in the assignment of the actual register numbers within the table and permits the convention that a register address of zero means that no register is assigned or available.

Data Parameter: A symbolic operand which represents an identifier, source-program constant, or intermediate result (triad) is presented to the Instruction Assembler as a data parameter operand. The Triad Code Generator ($TCODE) maintains a small data parameter table ($PTO) into which it places pertinent information regarding the data to be processed. The symbolic operand indicates which table entry the Instruction Assembler is to use in forming the D2 or R2 field of the current instruction. (The format and use of specific entries in this table are explained under "Compiler Variables" in Appendix A.)

Imaginary Address of Data Parameter: A symbolic operand may indicate the imaginary address of a data parameter table entry. In this case, the Instruction Assembler uses the data parameter table, but for RX instructions the storage address given in the table is incremented by an appropriate amount to obtain the imaginary portion of the data, and for RR instructions the register adjacent to the one specified in the table is used.

Scratch Storage: A symbolic operand may specify scratch storage. For each call to the Instruction Assembler, the first reference to scratch storage causes the Instruction Assembler to call the Temporary Storage Management routine ($VGTMP) to obtain a 32-byte allocation of temporary storage. This area is logically subdivided into two areas of 16 bytes each. The symbolic operand indicates by codes of 0, 2, 4, and 6 that the Instruction Assembler is to address the real or imaginary part of the first or second areas, respectively. In general, scratch storage is used to hold intermediate results of in-line data conversion. The same scratch storage area is maintained until the end of the current instruction sequence, at which point it is available for other use.

Constant: Code generation, especially in-line data conversion, may require constants which were not written as part of the original source program. When a symbolic operand specifies such a constant, the Instruction Assembler locates the hexadecimal text of the constant in the operation code table (O table) and calls the Constant Processor routine ($NCON). This routine assigns an address to the constant or returns its allocated address if the constant is already part of the object program. The Instruction Assembler then uses the address in constructing the current instruction.

Literal: A symbolic operand may specify a literal value (less than 256) to be placed directly into the generated instruction. Literal values may be given for either the register or storage operands of the instruction. They are used for condition codes, object-time displacements, and other cases where the Triad Code Generator knows in advance the absolute value required in the generated instruction.

Local Label Reference: Instruction sequences which perform comparisons and tests need the facility to branch to selected instructions within the sequence. Although these branches are quite local, the precise location of the branch address cannot be determined in advance, since the Instruction Assembler may introduce an unpredictable number of addressing instructions into the generated code. (See the discussion of addressing, below.) Hence, each instruction sequence may define up to ten local symbolic labels and may make one (and only one) reference to each.

The point at which a label is to be defined is specified with a DEF pseudo-operation; a reference to the label is indicated by a special symbolic operand. The Instruction Assembler maintains a symbolic label table, which is reset at each entry to the Instruction Assembler. If the label definition occurs before its reference, the code address of the definition-point is entered into the table. When the subsequent reference to the label is encountered, the address can be assembled and placed in the generated instruction at once. If the label is referred to before it is defined (which is the usual case), the location of the reference is placed in the table, and the address portion of the generated instruction is set to zero. Then when the definition occurs, the correct address is assembled and inserted in the generated code in the open space reserved for it at the point of reference.

Null: Symbolic instructions must contain either zero, two, or four operands so that the Instruction Assembler can scan the table easily. Since some pseudo-operations and indexable operations use an odd number of operands, a null operand code is employed to fill the unused space in the symbolic instruction table.

General Processing Logic

At initial entry to the Instruction Assembler, the local symbolic label and symbolic register assignment tables are cleared. An operation-code modifier is determined by inspecting the attribute codes in the incoming triad. Register assignments and the modifier value are set once for the entire instruction sequence.

The components of the symbolic instruction are moved into fixed work areas to simplify addressing them, and the operation code, flags, and register effect are established. If the operation indicates an indexable code, the third and fourth operands are moved from the symbolic instruction. A pointer is updated to the next symbolic instruction. If indicated, modification of the operation code is performed. On the basis of the results, a basic register-type flag is set for fixed or floating.

The first and second symbolic operands are inspected. If they are data parameter table entries, the information from the data parameter table is moved into fixed areas and amplified as necessary to simplify further processing.

At this point, if the instruction is a pseudo-operation, control passes to the appropriate pseudo-operation generator; when all instructions have been generated for the particular pseudo-operation, control returns to the beginning of the processing cycle. The next symbolic instruction is procured. If the instruction is a machine operation, modification for register format is performed, and the resulting code is tested. Control then passes to one of four paths, depending on whether the instruction format is RR, RX, RS(SI), or SS. These paths use two major subroutines, one of which prepares a machine register operand (for the R1 and R2 fields of the instruction), using either the first or second symbolic operand as input. The other routine prepares a storage operand (B2 and D2 fields) using the second symbolic operand and calls the register assignment routines ($VASGC and $VASGA) as required to establish symbolic assignments. The storage operand routine calls the Storage Address Assembler routine ($VDSAC) to obtain assembled addresses of source program constants and variables and the Temporary Storage Management routine ($VGTMP) to prepare compiler-generated constant addresses. It also prepares literal operands and local label references.

When all components of the machine instruction have been prepared, the object code is generated directly in the user's area of memory. After the code is generated, control converges into two paths; one path determines the register effects for register-to-register instructions; the other, for instructions involving main storage locations. After the appropriate action is taken to maintain the register tables, control returns to the beginning of the cycle. The next symbolic instruction is then procured, and the cycle iterates until terminated by a result pseudo-operation.

Addressing

Storage addresses must be assembled as a base register and a displacement value not exceeding 4095. Internally in the compiler, the storage addresses of constants, variables, temporary storage, etc., are maintained in the form of a base code which represents the class of storage (address, static, dynamic, etc.) and the displacement relative to the base allocated to that data.

In the object code, certain registers are assigned permanent base values to facilitate the addressing problem. Three registers address the first three pages of object code; one register addresses the base of the static and constants storage area; another, the address constant area; and another, the dynamic storage area for the block currently being executed. Four other address registers are volatile.

Since data may be allocated displacements which exceed that addressable by the permanent base register for that storage class, supplementary instructions may be needed to load one of the volatile address registers with a base value sufficient to enable the desired data to be addressed. This function is performed by the Storage Address Assembler routine ($VDSAC). In general, the Instruction Assembler presents the base code and displacement representing the storage address of an operand to $VDSAC, which returns an assembled halfword containing a base register number and appropriate displacement that the Instruction Assembler can then insert as required into the machine instruction being generated. Any supplementary instructions necessary to set the base register contents are generated by $VDSAC. The total number of generated instructions depends upon the allocation and addressability of the data involved.

In addition, certain classes of data present special addressing problems. Parameters used by CALL/360-OS PL/I subroutines are passed as addresses. Hence, references to parameter data require an extra level of indirectness in the addressing. A special flag associated with the data indicates whether it is a parameter. If it is, $VDSAC generates the coding necessary to accomplish the indirect addressing.

Subscripted data require an extra index to achieve the correct addressing. For RX format instructions, the extra index may be specified directly in the X1 field of the instruction. For SS instructions, $VDSAC generates coding to combine the base value and index value into a single register, since these instructions can only be singly indexed. However, SS instructions are generally generated to manipulate string-type data, which is addressed indirectly through a dope vector. Accordingly, when processing SS instructions, the Instruction Assembler tests the operand type. If it is a constant or already subscripted, normal direct addressing is used. If it is string-type data, the Instruction Assembler sets flags so $VDSAC will generate the index-combining instructions used for subscripts.

Pseudo-Operation Logic

Load Multiplier: If the operation is fixed-point, this pseudo-operation assigns a double register pair for the register operand and loads the storage operand into the odd-numbered register. If the second operand is already in a register, an attempt is made to use one of the adjacent registers, if possible, to form the register-pair.

Load Dividend: If the operation is fixed-point, this pseudo-operation assigns a double register pair for the register operand, loads the storage operand into the even register to establish the sign, and shifts the even register into the odd register to establish the proper magnitude. If the second operand is already in a register, an attempt is made to use one of the adjacent registers to form the register-pair.

Load Complex: Two registers (either fixed or floating, depending on type) are assigned and loaded from the real and imaginary parts of the storage operand, respectively. If the storage operand is already in registers, the load instruction is suppressed.

Store Complex: The two registers assigned to the register operand are stored into the real and imaginary parts of the storage operand, respectively.

Transfer: To save space in the symbolic instruction table, common instructions are shared between instruction sequences. The transfer pseudo-operation directs the Instruction Assembler to continue its processing at the symbolic instruction pointed to by the operand of the transfer.

Save: The register operand is saved in temporary storage, and the data parameter table entry given as the second operand is modified to contain the temporary storage address for the remainder of the current instruction sequence.

Define Label: The current object code location counter is entered in the local label table against the symbolic number of the label. If a reference was previously made to this label, the assembled value of the label is placed in the storage operand of the referencing instruction.

Load Multiple: A pair of registers is assigned to the register operand, and the storage operand is loaded into them with an LM instruction.

Store Multiple: The pair of registers assigned to the register operand is stored into the storage operand with an STM instruction.

Compute Subscript Address: The pseudo-operation is given a subscript displacement, which may be either in a register or in storage and may be either positive or negative. A subscript displacement is the computed displacement of an array element from the virtual origin of an array. (See "Dope Vectors" in Appendix E for a description of the computation of an array element address.) It is also given an array virtual origin, which may be either in a register or in storage. The pseudo-operation selects the optimum instruction sequence to load the subscript displacement positively into a fixed register other than general register zero (which cannot be used to index) and add the contents of the array virtual origin to it. In the usual case, the subscript displacement is positively in a register (which is sometimes register zero) and the array address is in storage. The instructions then generated are either:

```
    A  r,array
```

or

```
    LR r,0
    A  r,array
```

In processing the result of this operation, special flags are set back into the associated triad and register table entry to indicate that the result is a subscript address. The flags cause future storage references to the subscripted data to be double-indexed in order to achieve the desired subscripting.

Assign Double Register: This pseudo-operation causes a double register pair to be assigned to the symbolic computational register specified in the first operand.

BEGIN or END Block: The table entry for the adcon register which is assigned to address the block's dynamic storage area is updated to indicate that the register now addresses the base of the particular block being processed.

Begin DO: The level of temporary storage assignment is increased by one. Code is generated to store all registers containing active intermediate results into temporary storage at the new level.

End DO: The level of temporary storage assignment is reduced by one.

Save Volatile Registers: This pseudo-operation is used prior to the generation of calls to object-program library routines. It causes the generation of code to store fixed registers 0 and 1 and floating registers 0 and 2 (registers which are volatile across library calls) into temporary storage if they contain active intermediate results.

Forward Compiler-Generated Branch: Nested procedures, THEN clauses, DO-loops, etc., require the compiler to generate a forward branch. Since the address of the branch point cannot be computed in advance, this pseudo-operation reserves eight bytes of srequire the compiler torage for the branch instruction to be inserted later. The branch point may not be addressable by any of the three registers permanently assigned to cover the object code. In this case, the branch must be resolved by loading an adcon from a table in the adcon area and then branching. Hence, two instructions may be needed when the branch is resolved, and an adcon register will be required. Accordingly, when reserving the space, an adcon register is assigned and its number is stored in the reserved area along with the condition code to be used in the final branch. The reserved area thus has the format:

```
    DC     AL4(0)
    BC     condition,0(0,adconreg)
```

When multiple branches are made to the same point, a chain of pointers is linked through the first word of each reserved area. When the definition of the branch point occurs, all references to it can be resolved by following the chain.

Resolve Compiler-Generated Branch: This pseudo-operation assembles the address of the current code location counter. This operation may involve generating a load instruction if the code location is not addressable by any of the three permanently assigned cover registers. The assembled address (and load instruction, if needed) is then placed in all the forward compiler-generated branches which refer to the current location. The references are found by following a chain of pointers established in the reserved areas (see "Forward Compiler-Generated Branch" above). The layout of the reserved areas after resolution (if no additional cover is needed) is as follows:

```
    BC      condition,branch-point-displ(0,codecover)
    BC      condition,branch-point-displ(0,codecover)
```

or, if the branch point is not directly addressable,

```
    L       adconreg,code-cover-table(0,adconcover)
    BC      condition,branch-point-displ(0,adconreg)
```

Remainder: This pseudo-operation is given two symbolic registers as its operands. The even-numbered register adjacent to the one which is currently assigned to the second operand is assigned to the first operand. This operation enables the register containing the remainder of a previous fixed-point division to be recovered.

Load Index Register: A flag is set to prevent the Computational Register Assignment routine ($VASGC) from assigning absolute register zero; the operation code is set for a fixed-point load, and processing continues as for a machine operation. This insures that the storage operand will be loaded into a register capable of indexing.

Source Label Pseudo-Operations: The Instruction Assembler has the responsibility for the final resolution of references to source program labels. These references are of three kinds: assignment statements, branches (GO TO statements), and linkages to remote FORMAT statements. Further, in the first two cases, the references may be either to a label within the current block (intrablock branch) or to one in an encompassing block (interblock branch). Remote FORMAT statements must be within the current block.

For assignments and interblock branches, information about the label is conveyed to the object-program library routines by a special two-word constant which has the following format:

```
 0           2            4            6            8  Byte
┌──────────────────────────────────────────────────────┐
│      Displacement of Branch Point       │ Displacement │
│      Within Code Area                   │ Block Adcon  │
│                                         │ Area Within  │
│                                         │ Adcon Area   │
└──────────────────────────────────────────────────────┘
```

The Instruction Assembler builds such constants as required and causes them to be added to the object program by calling the Constant Processor ($NCON). The processing performed by the source label pseudo-operations must take into account the fact that the location allocated for this constant may not be directly addressable by the register which covers the object-program constant area, and that an extra load instruction may be required whenever a reference is made to the constant. An additional consideration arises in the following case:

```
A: PROC;
   DCL M LABEL;
   B:  PROC;
        .
        .
        .
   C:  GO TO M;
        .
        .
        .
   M:   .
        .
        .
       END B;
   END A;
```

At statement C above, it is impossible for the compiler to know whether the identifier M to be used in resolving the GO TO statement is the statement-label variable declared in block A or the label of some statement internal to block B. To insure correct resolution, the Instruction Assembler must determine whether the operand of a GO TO is potentially a statement-label variable in another block. If so, the Instruction Assembler must generate the code necessary to address it, since the allocated address may not be directly coverable. In the final resolution, the addressing instructions may not be needed if the identifier is a label within the current block. However, this case is not determinable in advance.

The processing performed for the various conditions may be summarized as follows:

Backward Assignment: The statement-label constant is formed, allocated, and addressed. A doubleword load and store is then generated:

```
LD          reg,constant
STD         reg,label-variable
```

Backward Intrablock Branch: To a statement-label variable: The location allocated to the statement-label constant is addressed and assembled. A conditional branch using the complement of the desired condition is generated to branch around coding which is then generated to call the GO TO Interpreter (IHESAF), using the address of the statement-label constant as a parameter.

To a statement-label constant: The address allocated to the label is obtained and assembled; a conditional branch instruction is generated using the assembled address.

Backward Interblock Branch: The statement-label constant is formed and allocated. Processing then proceeds as for a backward intrablock branch to a statement-label variable.

Forward Assignment: If the rightmost operand is a potential statement-label variable, its address is assembled, and any necessary cover instructions are generated. The address of the leftmost operand is assembled, and any necessary cover instructions for it are generated. A twelve-byte area is then reserved in the object code. A floating-point register is assigned, and operation codes for LD and STD are generated for it. A flag of zero is set as the first byte of the area (to distinguish assignment from GO TO), and a chained pointer is established in the remainder of the first word to link together all references to a given label. The completed area has the format:

```
[addressing instructions for right-operand]
[addressing instructions for left-operand]
DC          AL1 (block number)
DC          AL3 (pointer list)
LD          reg,right-operand or LD reg,0
STD         reg,left-operand
```

Resolve Forward Assignment: At the end of a block, those assignments for which definitions are available are completed. For the case where a tentative definition of the right operand was the correct one, the resolved instructions are:

```
NOP         0
LD          reg,right-operand
STD         reg,left-operand
```

For the case where no tentative definition was made, or where the tentative definition was overridden by a later declaration, the necessary statement-label constant is constructed, allocated, and addressed. Depending on whether or not the constant can be directly addressed, the resolved area has the format:

```
   NOP         0
   LD          reg,label-constant
   STD         reg,left-operand
or
   L           13,cover-constant(0,adconarea)
   LD          reg,label-constant
   STD         reg,left-operand
```

Register 13 is inhibited from assignment during the construction of other storage addresses in the sequence so that it can be used, if needed, to obtain cover for the constant when the sequence is resolved.

Forward Branch: If the operand is a potential statement-label variable, its address is assembled, and any necessary cover instructions are generated. Next, a conditional branch using the complement of the desired condition code is generated to branch around a 14-byte area which is then reserved. Into the reserved area are placed the tentatively assembled operand address, if any, a flag byte of 1 (to distinguish GO TO from assignment), the current block number, and a linked pointer (to chain together all references to a given label). The area has the format:

```
[addressing instructions for operand, if needed]
[conditional branch to *+14, if needed]
DC          AL1(1)
DC          AL3(pointer link)
DC          AL1(current block number)
DC          AL1(reserve block number if potential statement-label
               variable)
DC          AL2(assembled operand address)
DS          6C
```

Resolve Forward Intrablock Branch: At the end of a block, forward branches to those labels which have been defined are resolved. An intrablock branch is indicated when the block number that was stored in the reserved area matches that of the block being terminated. Depending upon whether or not the branch point is directly addressable, the resolved area has the format:

```
BC          15,branch-point-displ(0,codecover)
NOP         0
NOP         0
NOPR        0
```

or

```
L           13,code-cover-adcon-displ(0,adconarea)
BC          15,branch-point-displ(0,13)
NOP         0
NOPR        0
```

Resolve Forward Interblock Branch: An interblock branch is indicated when the block number that was stored in the reserved area does not match the number of the block being terminated. In this case, the GO TO Interpreter must be called. If the operand was an actual or potential statement-label variable, its assembled address is already positioned in the reserved area. If it was not, a statement-label constant is constructed and allocated, and its address is assembled. Depending upon whether or not the constant is directly addressable, the resolved area has the format:

```
NOP         0
L           13,label-constant-displ(0,constant-area)
L           15,=A(go-to-interpreter)
BALR        14,15
```

or

```
L           13,constant-cover-displ(0,adconarea)
L           13,label-constant-displ(0,13)
L           15,=A(go-to-interpreter)
BALR        14,15
```

Backward Format List Linkage: To a statement-label constant: Processing is the same as for a backward intrablock branch, except that the operation generated is a BAL instruction, using register 14 as the return address register.

To a statement-label variable: The following coding is generated:

```
LR      15,6                Place code base in R15
A       15,label variable   Compute branch address
BALR    14,15               Link to remote format
```

Forward Format List Linkage: An area is reserved as for a forward branch, with a special flag byte of 2 to indicate a format list reference. Since format list linkages are always unconditional, no conditional branch coding is generated for either forward or backward references.

Resolve Forward Format List Linkage: The resolution is the same as that for an intrablock branch, except that the branch instruction is a BAL using register 14 as the return address register.

Result Pseudo-Operations: Result pseudo-operations terminate the sequence of symbolic instructions and cause relevant information to be posted in the register table and in the triad for which code was generated. They are principally used to indicate the disposition and type of intermediate arithmetic computations.

The various result types are:

Real: A pointer to the current triad is placed in the register table entry for the register assigned to symbolic computational register zero, and the address of this register is placed in the register-address field of the triad. The complex result bit in the register table entry is reset. The long/short, fixed/float result bits are left as set by the effects of the various individual generated instructions. The storage-sign bit is transferred to the register-sign bit position in the triad, and the last-use count field of the triad is transferred to the register table, to be used in determining when the register will next be free for reassignment.

Real, Short: The effect is the same as real, except that the short precision result bit is set in the register result field.

Real, Long: The effect is the same as real, except that the long precision result bit is set in the register result field.

Complex: The effect is the same as real, except that the complex result bit is set in the register result field.

Complex, Short: The effect is the same as complex, except that the short precision result bit is set in the register result field.

Complex, Long: The effect is the same as complex, except that the long precision result bit is set in the register result field.

Mask Type: The effect is the same as real, except that the result bits in the register result field for long/short, fixed/float, and real/complex are set from the corresponding positions in the triad type mask.

Real Part: Symbolic computational register zero represents a double register pair containing complex results, of which only the real half is to be retained. The odd-numbered register is freed, both registers are marked as single usage, and synonyms are removed from the even-numbered register. The remainder of the effect is then as described above for real.

Imaginary Part: Symbolic computational register zero represents a double register pair containing complex results, of which only the imaginary half is to be retained. The even-numbered register is freed and its synonyms are removed. Both registers are marked as single usage, and the odd-numbered register is used as the result register for the remainder of the processing, which is the same as that described above for real.

Null: Non-arithmetic instruction sequences generally terminate with a null result, which stops the generation of instructions without changing either the triad or the register table.

Errors Detected

None

Local Variables

W$CVN	C register save area
W$GVN	G register save area
W$PVN	P register save area
VNEFF	Register effect code multiplied by 4 (halfword)
VNINST	Symbolic operation code from symbolic instruction (1 byte)

VNMOD	Modifier for operation mode (halfword); 0 = long float, X'0010' = short float, X'00F0' = fixed
VNOPC	Machine operation code of pseudo-operation number (fullword, value in third byte)
VNOPSV	Save area for operand information, used during construction of SS instructions (4 words)
VNOP1	Code for symbolic operand 1 (byte)
VNOP1C	Storage address for operand 1 (word)
VNOP1G	Register table displacement for operand 1 (byte)
VNOP1K	Encoded syntactic token for operand 1 (word)
VNOP1L	Length of operand 1 (halfword)
VNOP1R	Pointer to register assigned to operand 1 (word)
VNOP1S	Parameter and sign bits for operand 1 (byte)
VNOP2	Code for symbolic operand 2 (byte)
VNOP2C	Storage address for operand 2 (word)
VNOP2G	Register table displacement for operand 2 (byte)
VNOP2K	Encoded syntactic token for operand 2 (word)
VNOP2L	Length of operand 2 (halfword)
VNOP2R	Pointer to register assigned to operand 2 (word)
VNOP2S	Parameter and sign bits for operand 2 (byte)
VNOP3	Symbolic operand 3 (index register) (byte)
VNOP4	Symbolic operand 4 (base register) (byte)
VNSAVE	Save area for register C1 on internal subroutine linkages (word)
VNSINS	Address of next symbolic instruction to be processed (word)
VNSSTG	Temporary storage address allocated for scratch usage (word)
VNTOP1	Symbolic code for operand 1 type (byte)
VNTOP2	Symbolic code for operand 2 type (byte)
VNTOP3	Symbolic code for operand 3 type (byte)
VNTOP4	Symbolic code for operand 4 type (byte)
VNWORK	Miscellaneous work area (2 words)
$VLBLT	Symbolic label table (64 bytes)
$VLS	Flag for enabling listings (halfword)
$VSART	Symbolic adcon register table (9 words)
$VSCRT	Symbolic computational register table (9 words)

Program Interface

Entry Points

$VINSA. Register P5 addresses the triad for which code is to be generated; register G7 points to the first symbolic instruction in the selected sequence.

Exit Conditions

Control returns to the caller immediately following the invoking call. Registers at exit are the same as at entry; no specific output values are returned.

Routines Called

$GPUT	Output Director
$NCONS	Constant Processor
$NLSIB	Library Search
$VASGA	Adcon Register Assignment
$VASGC, $VFREE, $VRSYN, $VSAVE	Computational Register Assignment
$VDSAC	Storage Address Assembler
$VGTMP	Temporary Storage Management
$WBACK	Segment Management
$SVC	SVC Director

Global Variables

$ACODE	Pointer to Next Available Byte in Object Code Area
$CBKNO	Current Block Number
$NCCUR	Number of Last Triad with Generated Code
$VRTYP	Flag for Register Type Assignment
M$	Address of Symbolic Instruction Table
O$	Address of Operation Code Table
$NXFLG	Communication Flag
$PADD	Core Address of Operand
$PTKN	Operand Token
B$ACTV	B Table Active-Entry Pointer
B$CURR	B Table Current-Entry Pointer
B$TAIL	B Table End-of-Table Pointer
$PTO	Origin of Data Parameter Table
$TEMPL	Temporary Storage Level Count
$TEMPN	Temporary Storage Level Number
R$AD	Pointer to Adcon Register Portion of Register Table
R$FL	Pointer to Floating-Point Register Portion of Register Table
R$FX	Pointer to Fixed-Point Register Portion of Register Table
$VLINE	Print-Line Work Area
$VLPAK	Doubleword-Aligned Work Area
$VRAMT	Flag for Register Assignments
R$SY	Pointer to Head of Register Table Synonym List
R$TBL	Pointer to Base of Register Table
S$ACTV	S Table Active-Entry Pointer
$CTON	Count of Current Nesting of On-Units
$SEVCT	Number of Error Messages Produced
S$CURR	S Table Current-Entry Pointer
S$TAIL	S Table End-of-Table Pointer
Z Table	Triad Table
M Table	Symbolic Instruction Table
O Table	Operation Code Table
A List	Dictionary Attribute List
C Table	Constant Table
R Table	Register Table
B Table	Block Information Table
S Table	Temporary Storage Table

Logic Diagram

Chart 61 shows the detailed logic diagram for the Instruction Assembler routine.

PART 7 LOGIC DIAGRAMS

The detailed logic diagrams for the routines concerned primarily with code generation follow.

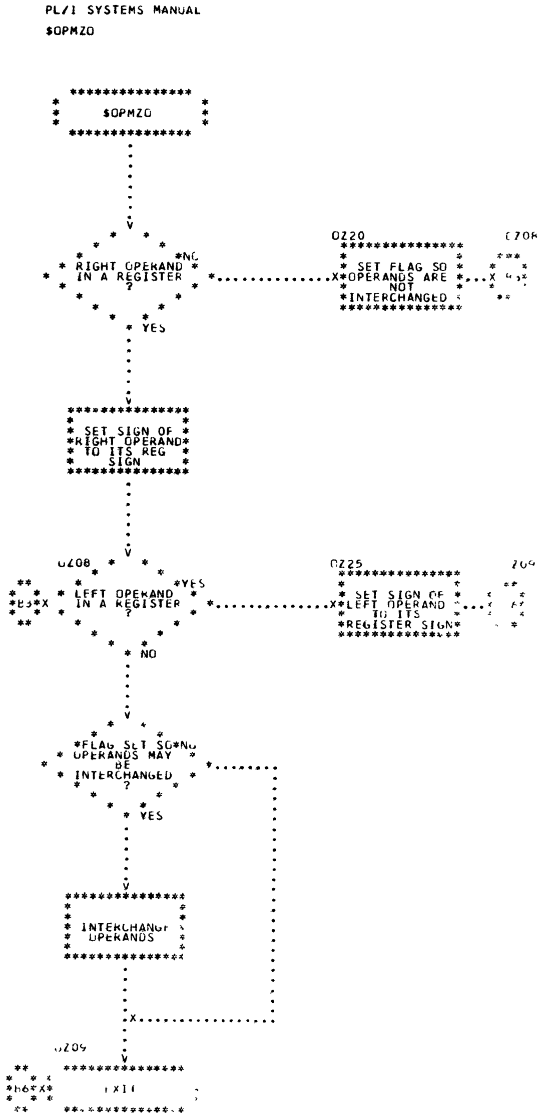


Chart 52. Optimize Operands

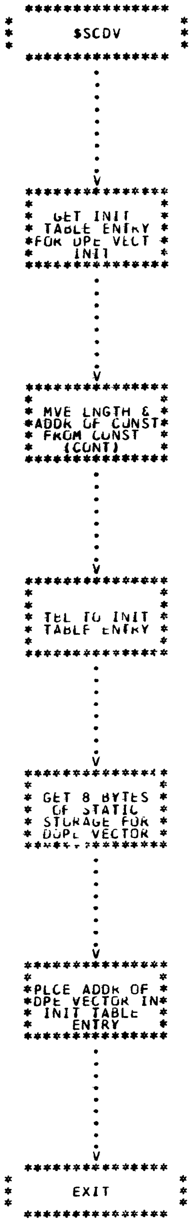


Chart 53. String Constant Dope Vector Initializer

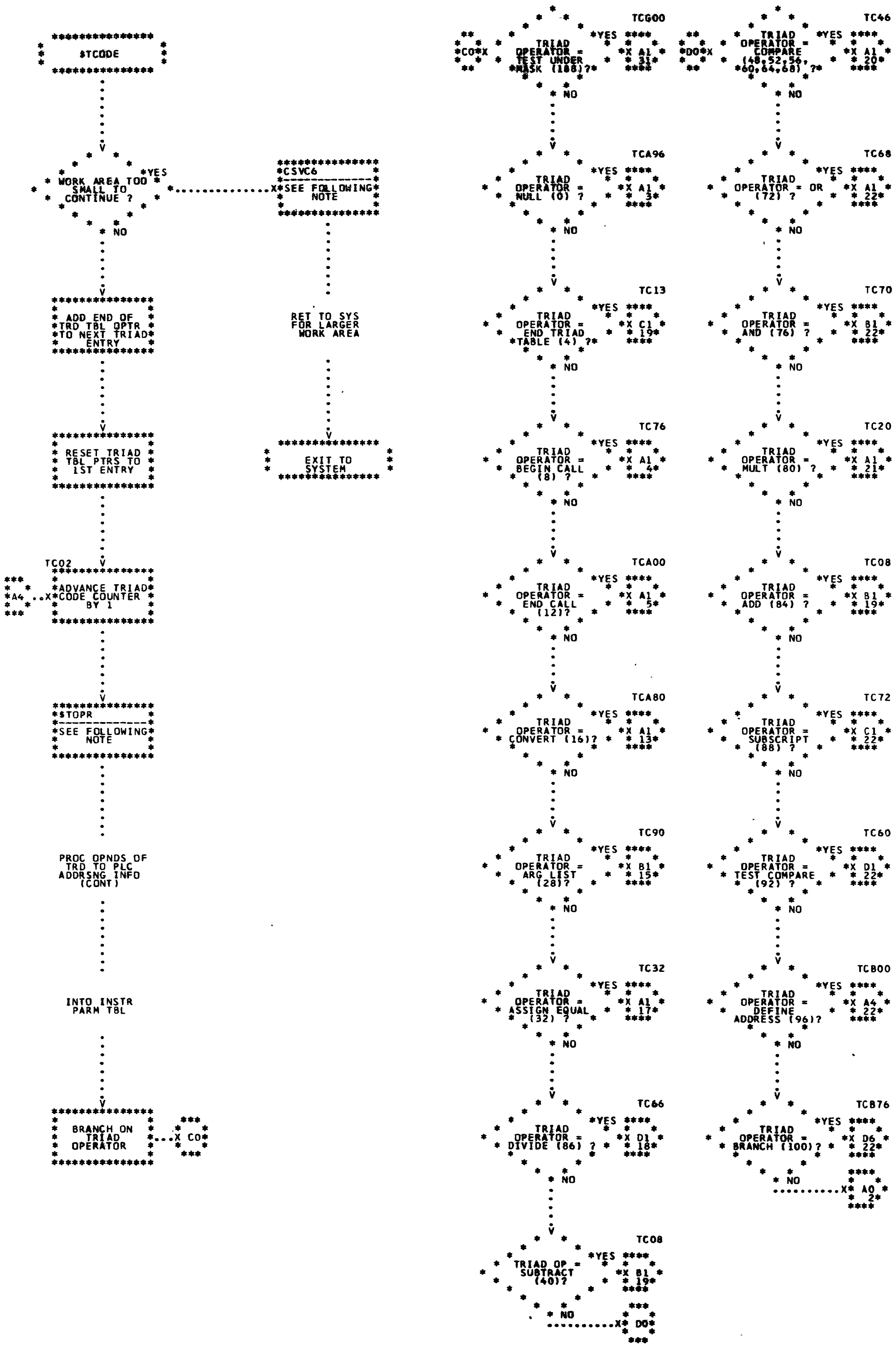


Chart 54. Triad Code Generator (Page 1 of 31)

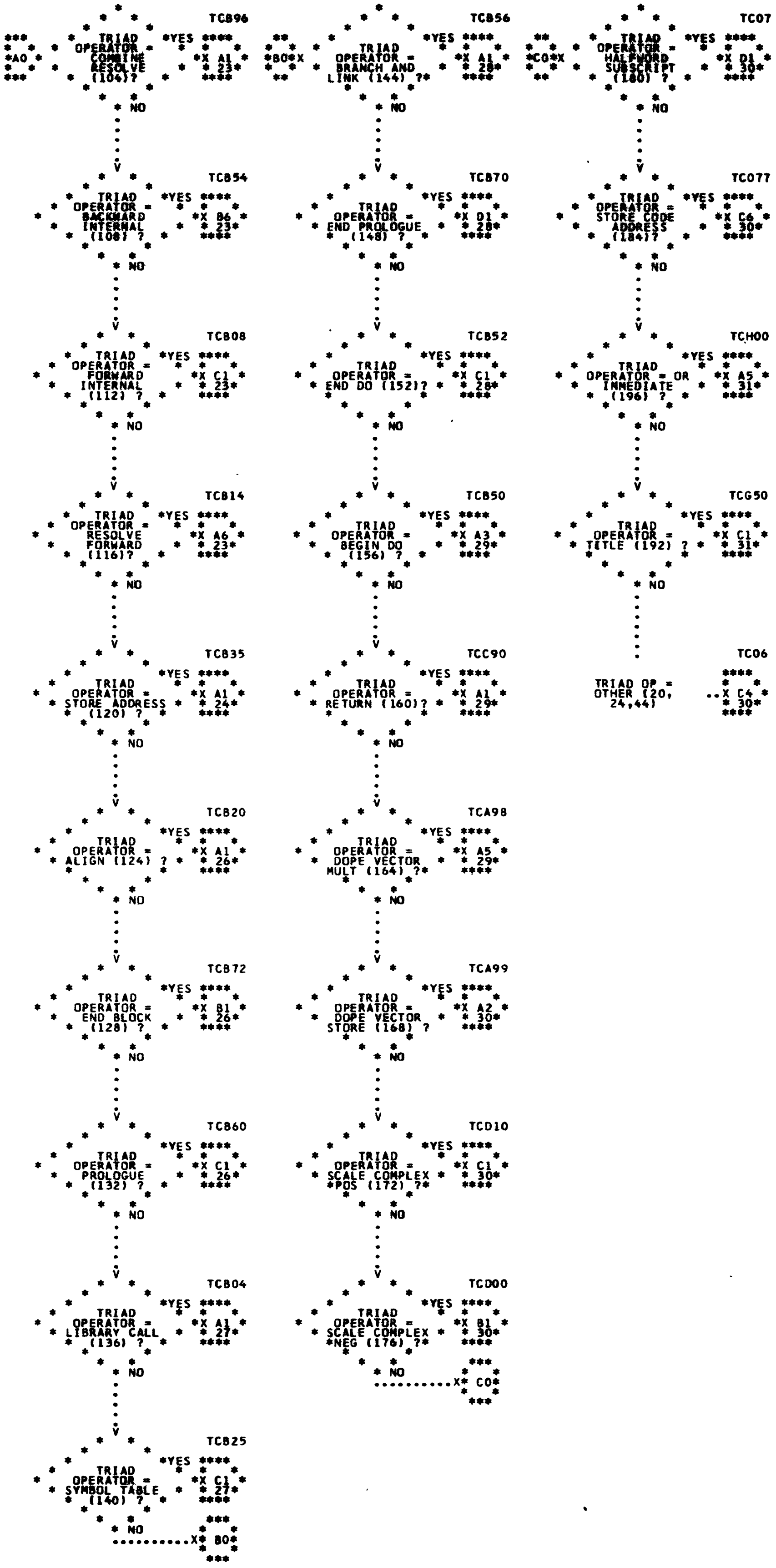


Chart 54. Triad Code Generator (Page 2 of 31)

NULL OPERATOR

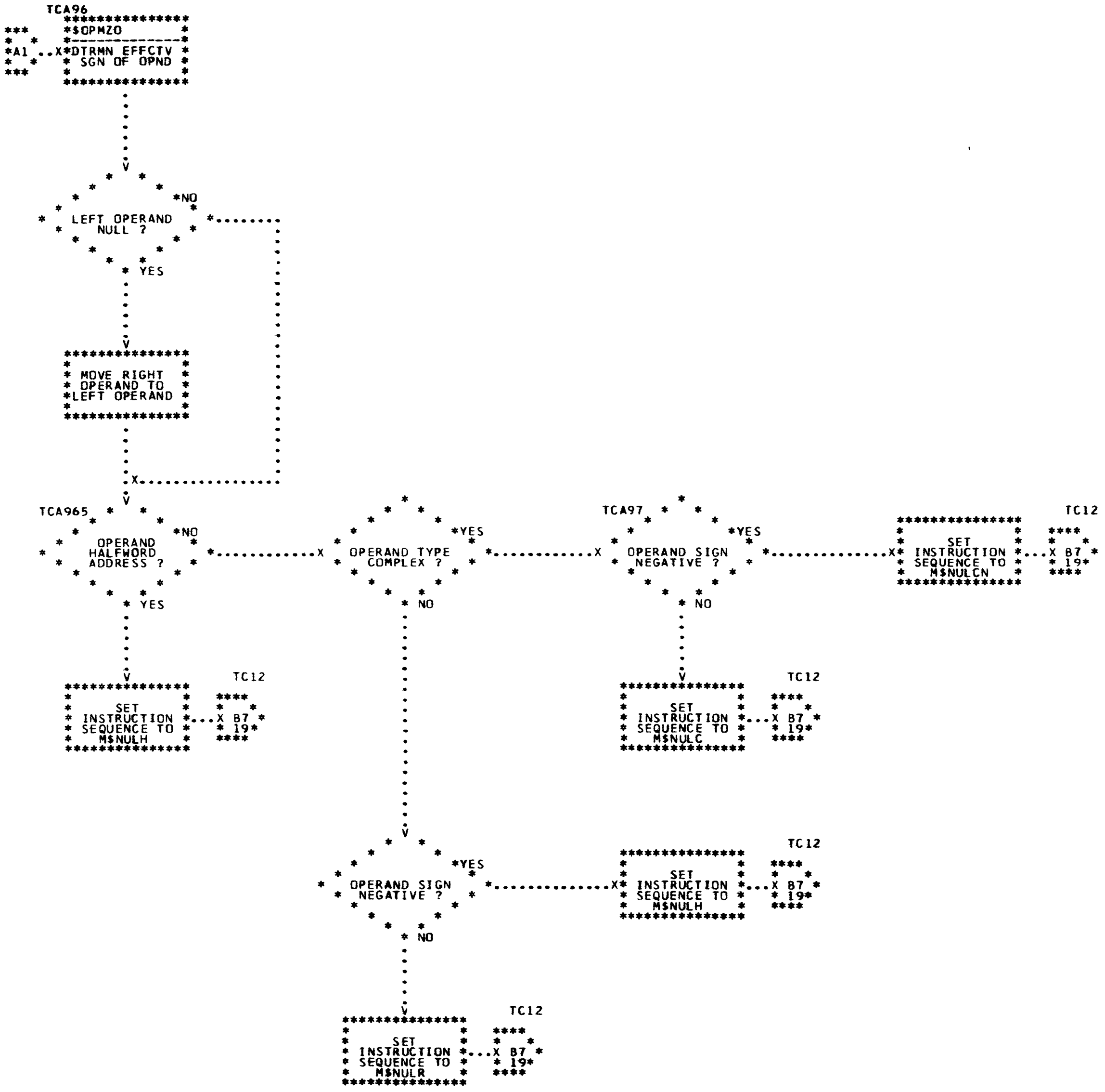
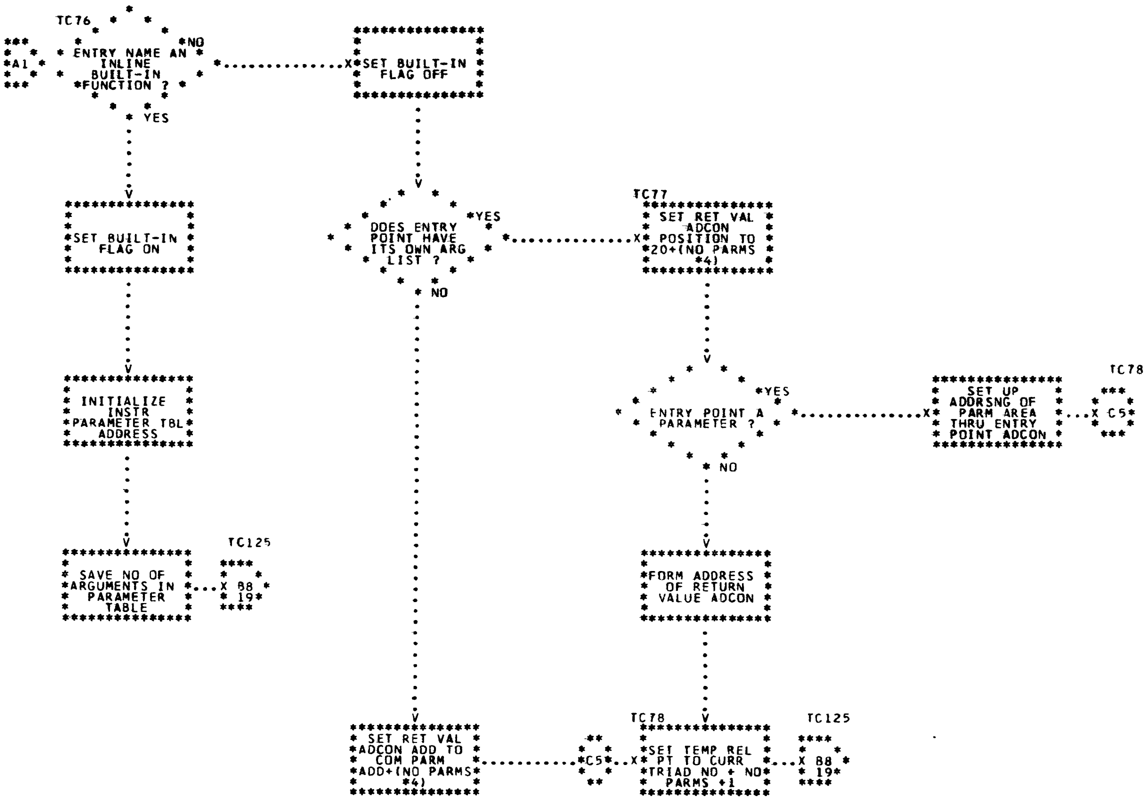
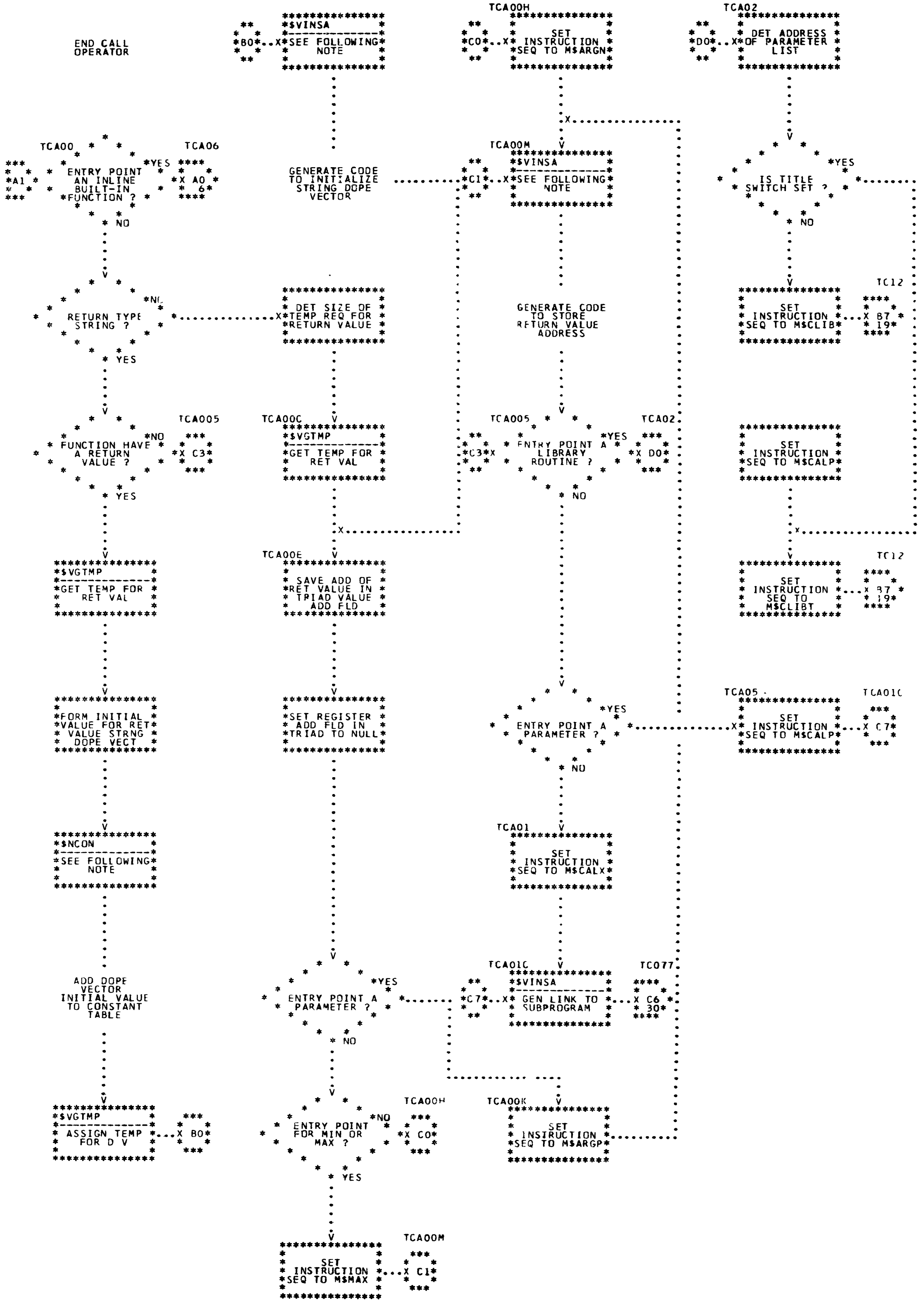


Chart 54. Triad Code Generator (Page 3 of 31.

BEGIN CALL
OPERATOR




END CALL OPERATOR
B0
$VINSA
SEE FOLLOWING NOTE
TCA00H
C0
SET INSTRUCTION SEQ TO M$ARGN
TCA02
D0
DET ADDRESS OF PARAMETER LIST
TCA00
A1
ENTRY POINT AN INLINE BUILT-IN FUNCTION ?
YES
NO
TCA06
A0 6
GENERATE CODE TO INITIALIZE STRING DOPE VECTOR
TCA00M
C1
$VINSA
SEE FOLLOWING NOTE
IS TITLE SWITCH SET ?
YES
NO
RETURN TYPE STRING ?
NO
YES
DET SIZE OF TEMP REQ FOR RETURN VALUE
GENERATE CODE TO STORE RETURN VALUE ADDRESS
TC12
SET INSTRUCTION SEQ TO M$CLIB
B7 19
FUNCTION HAVE A RETURN VALUE ?
NO
YES
TCA005
C3
TCA00C
$VGTMP
GET TEMP FOR RET VAL
TCA005
C3
ENTRY POINT A LIBRARY ROUTINE ?
YES
NO
TCA02
D0
SET INSTRUCTION SEQ TO M$CALP
$VGTMP
GET TEMP FOR RET VAL
TCA00E
SAVE ADD OF RET VALUE IN TRIAD VALUE ADD FLD
TC12
SET INSTRUCTION SEQ TO M$CLIBT
B7 19
FORM INITIAL VALUE FOR RET VALUE STRNG DOPE VECT
SET REGISTER ADD FLD IN TRIAD TO NULL
ENTRY POINT A PARAMETER ?
YES
NO
TCA05
TCA01C
SET INSTRUCTION SEQ TO M$CALP
C7
$NCON
SEE FOLLOWING NOTE
TCA01
SET INSTRUCTION SEQ TO M$CALX
ADD DOPE VECTOR INITIAL VALUE TO CONSTANT TABLE
ENTRY POINT A PARAMETER ?
YES
NO
TCA01C
C7
$VINSA
GEN LINK TO SUBPROGRAM
TC077
C6 30
$VGTMP
ASSIGN TEMP FOR D V
B0
ENTRY POINT FOR MIN OR MAX ?
NO
YES
TCA00H
C0
TCA00K
SET INSTRUCTION SEQ TO M$ARGP
SET INSTRUCTION SEQ TO M$MAX
TCA00M
C1

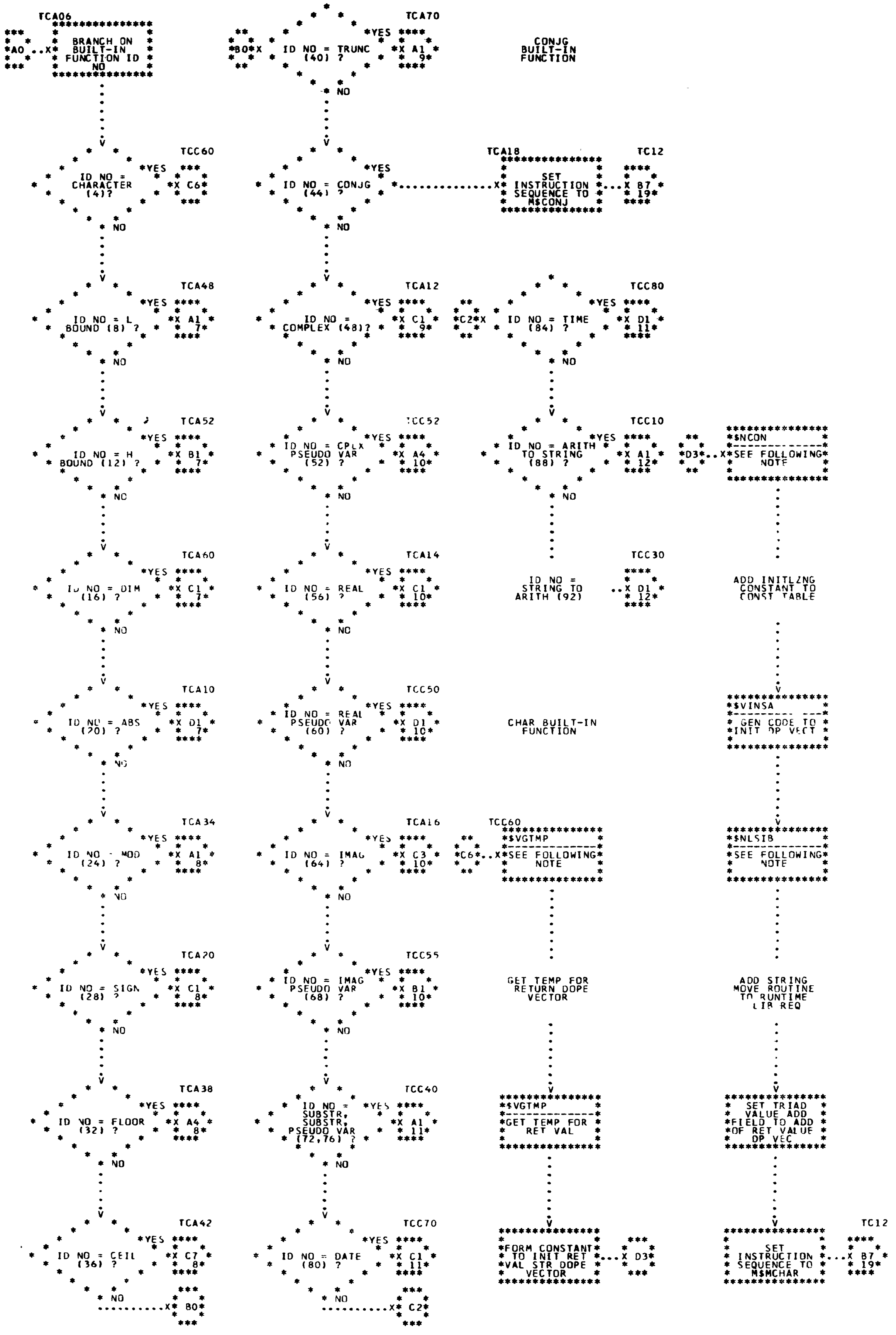


Chart 54. Triad Code Generator (Page 6 of 31)

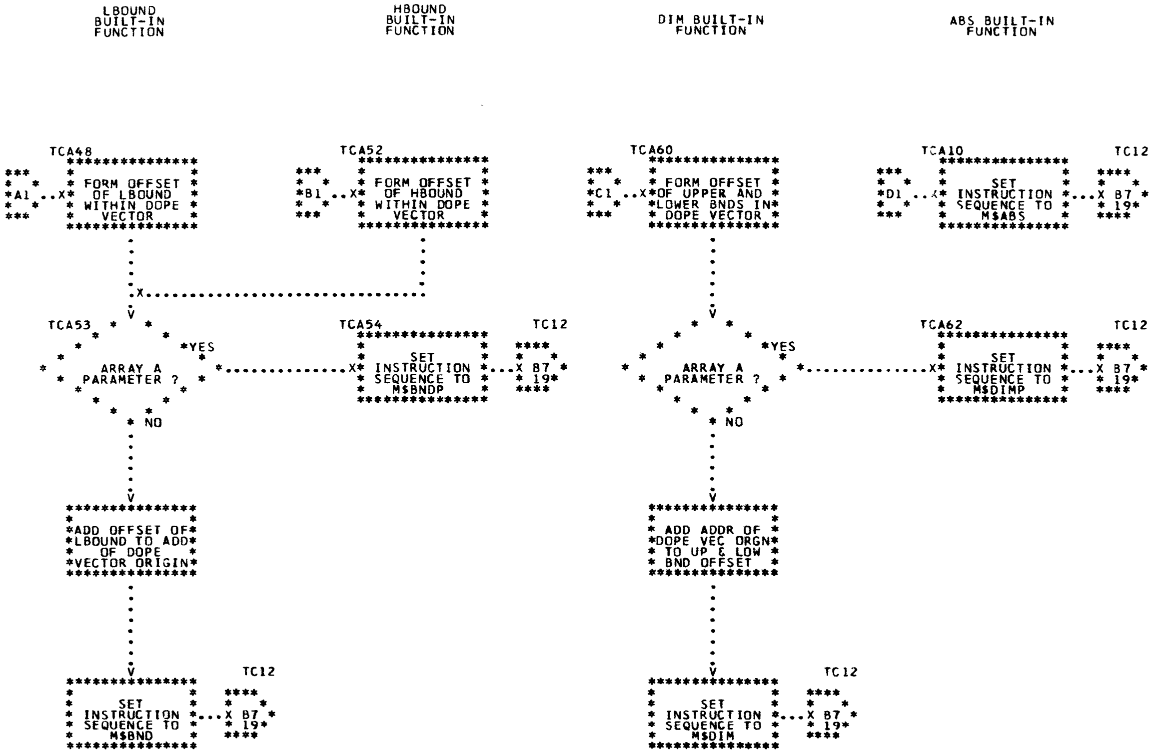


Chart 54. Triad Code Generator (Page 7 of 31)

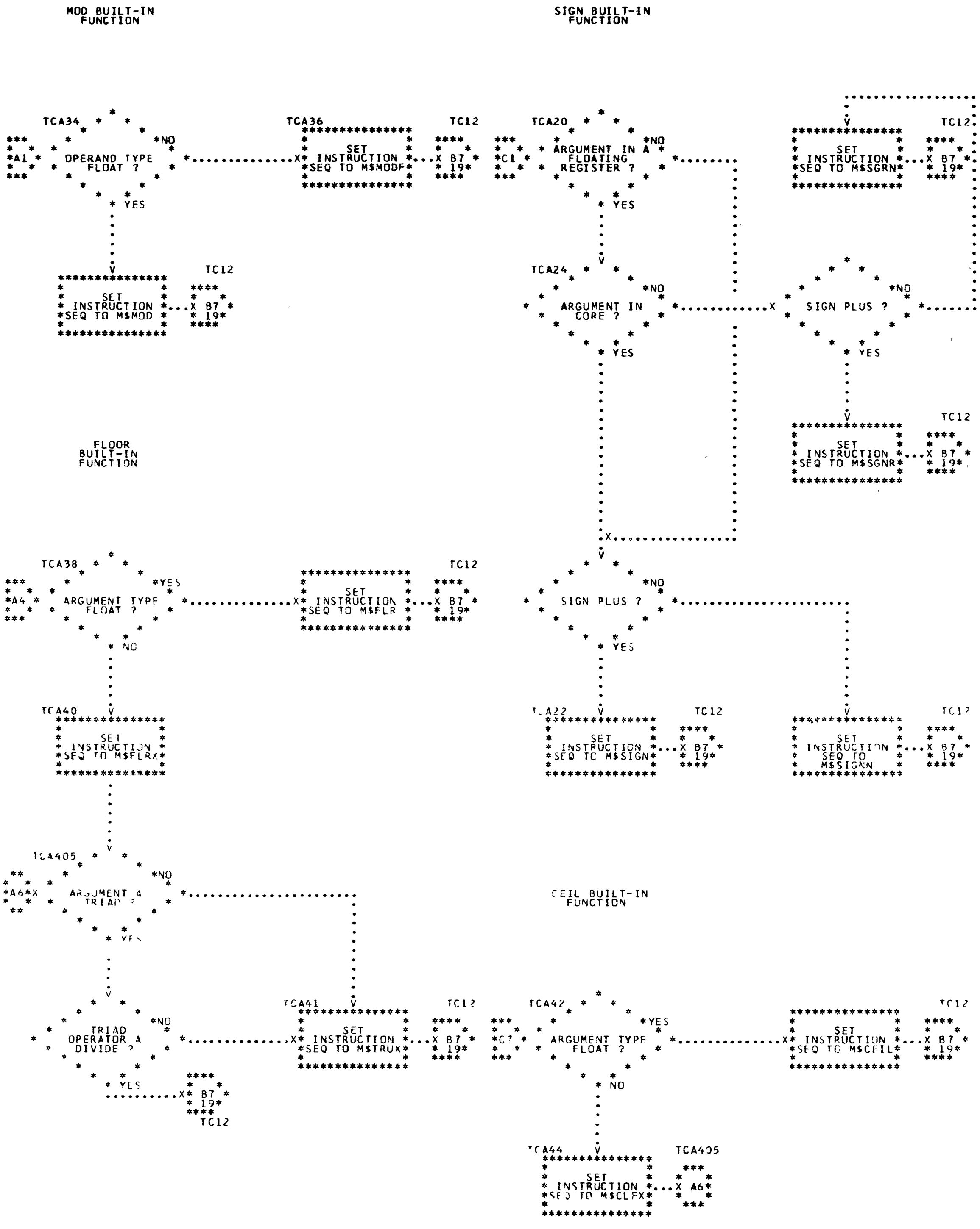
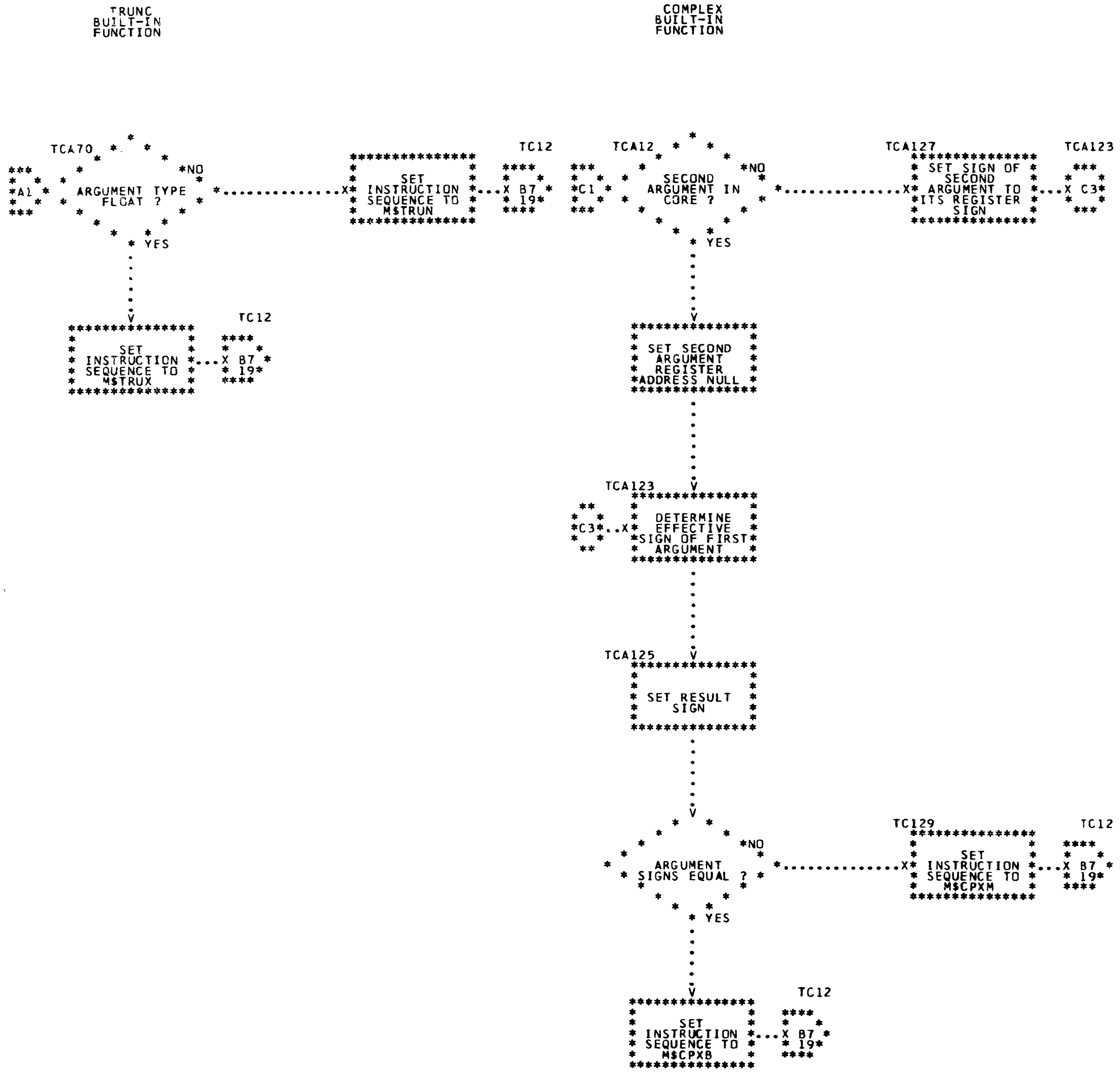


Chart 54. Triad Code Generator (Page 8 of 31)


TRUNC BUILT-IN FUNCTION
COMPLEX BUILT-IN FUNCTION
TCA70
A1
ARGUMENT TYPE FLOAT ?
NO
YES
SET INSTRUCTION SEQUENCE TO M$TRUN
TC12
B7 19
SET INSTRUCTION SEQUENCE TO M$TRUX
TC12
B7 19
TCA12
C1
SECOND ARGUMENT IN CORE ?
NO
YES
TCA127
SET SIGN OF SECOND ARGUMENT TO ITS REGISTER SIGN
TCA123
C3
SET SECOND ARGUMENT REGISTER ADDRESS NULL
TCA123
C3
DETERMINE EFFECTIVE SIGN OF FIRST ARGUMENT
TCA125
SET RESULT SIGN
ARGUMENT SIGNS EQUAL ?
NO
YES
TC129
SET INSTRUCTION SEQUENCE TO M$CPXM
TC12
B7 19
SET INSTRUCTION SEQUENCE TO M$CPXB
TC12
B7 19

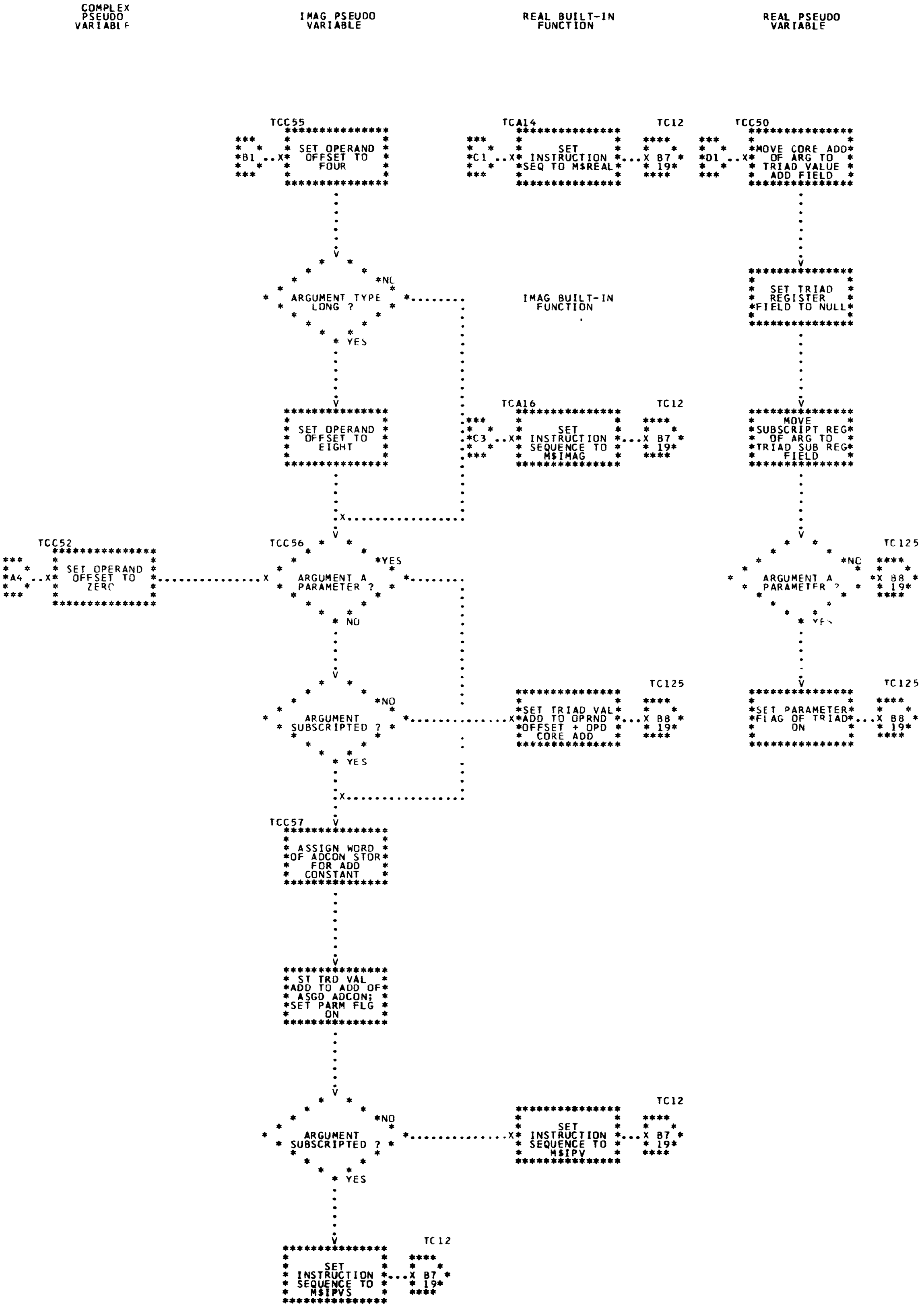
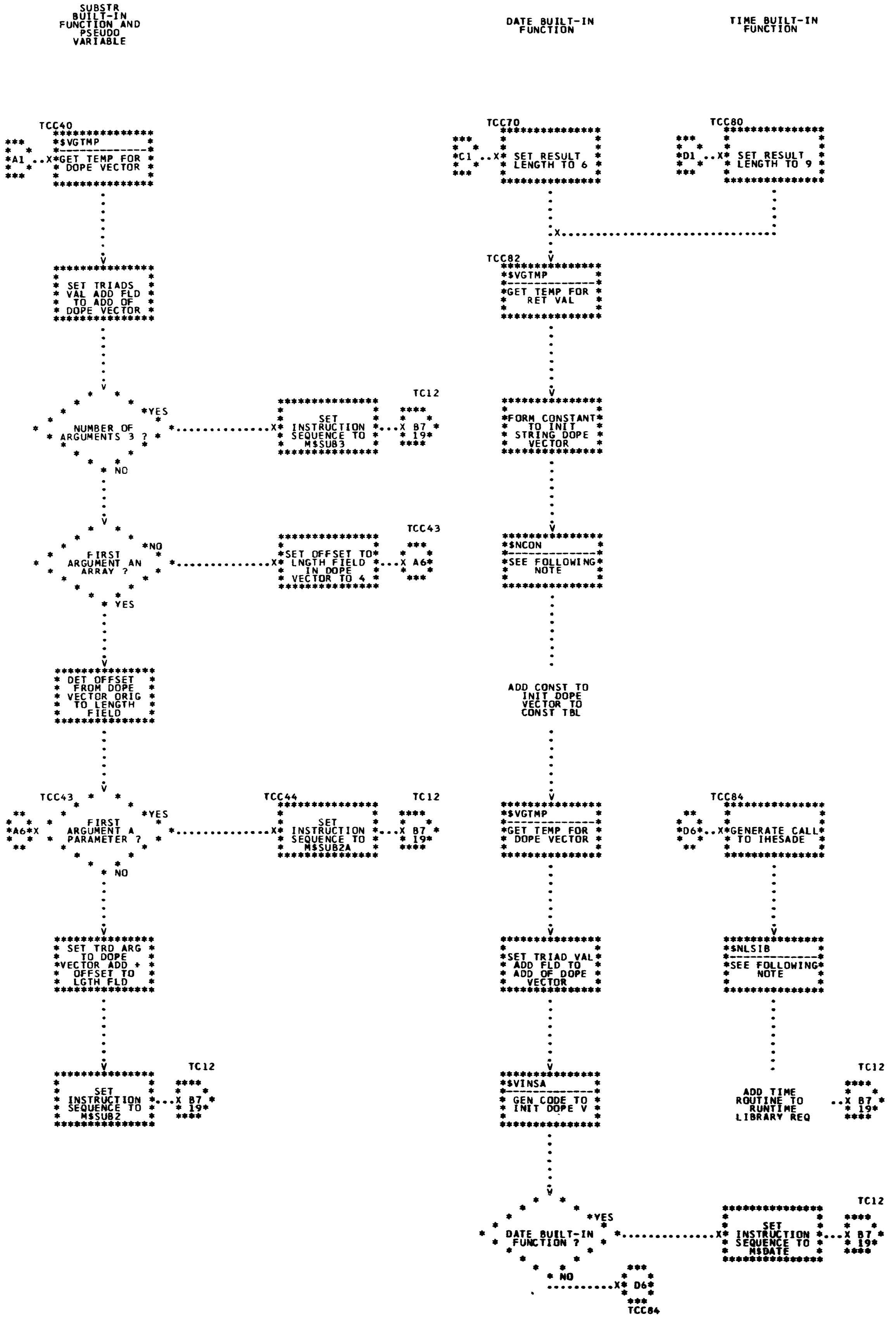


Chart 54. Triad Code Generator (Page 10 of 31)


SUBSTR BUILT-IN FUNCTION AND PSEUDO VARIABLE
DATE BUILT-IN FUNCTION
TIME BUILT-IN FUNCTION
TCC40
A1
$VGTMP
GET TEMP FOR DOPE VECTOR
SET TRIADS VAL ADD FLD TO ADD OF DOPE VECTOR
NUMBER OF ARGUMENTS 3 ?
YES
NO
SET INSTRUCTION SEQUENCE TO M$SUB3
TC12
B7
19
FIRST ARGUMENT AN ARRAY ?
NO
YES
SET OFFSET TO LNGTH FIELD IN DOPE VECTOR TO 4
TCC43
A6
DET OFFSET FROM DOPE VECTOR ORIG TO LENGTH FIELD
TCC43
A6
FIRST ARGUMENT A PARAMETER ?
YES
NO
TCC44
SET INSTRUCTION SEQUENCE TO M$SUB2A
TC12
B7
19
SET TRD ARG TO DOPE VECTOR ADD + OFFSET TO LGTH FLD
SET INSTRUCTION SEQUENCE TO M$SUB2
TC12
B7
19
TCC70
C1
SET RESULT LENGTH TO 6
TCC80
D1
SET RESULT LENGTH TO 9
TCC82
$VGTMP
GET TEMP FOR RET VAL
FORM CONSTANT TO INIT STRING DOPE VECTOR
$NCON
SEE FOLLOWING NOTE
ADD CONST TO INIT DOPE VECTOR TO CONST TBL
$VGTMP
GET TEMP FOR DOPE VECTOR
SET TRIAD VAL ADD FLD TO ADD OF DOPE VECTOR
$VINSA
GEN CODE TO INIT DOPE V
DATE BUILT-IN FUNCTION ?
YES
NO
SET INSTRUCTION SEQUENCE TO M$DATE
TC12
B7
19
D6
TCC84
TCC84
D6
GENERATE CALL TO IHESADE
$NLSIB
SEE FOLLOWING NOTE
ADD TIME ROUTINE TO RUNTIME LIBRARY REQ
TC12
B7
19

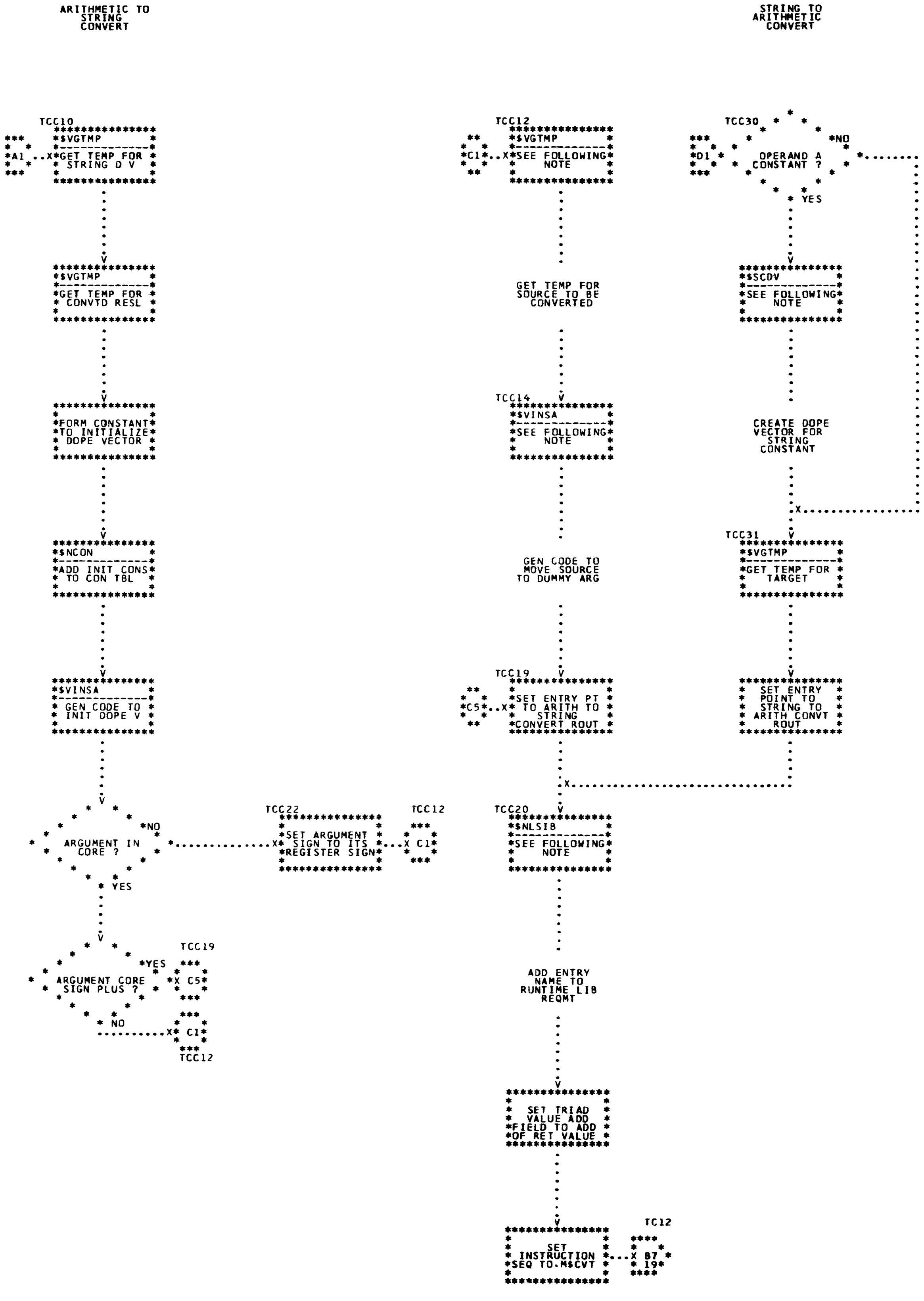


Chart 54. Triad Code Generator (Page 12 of 31)

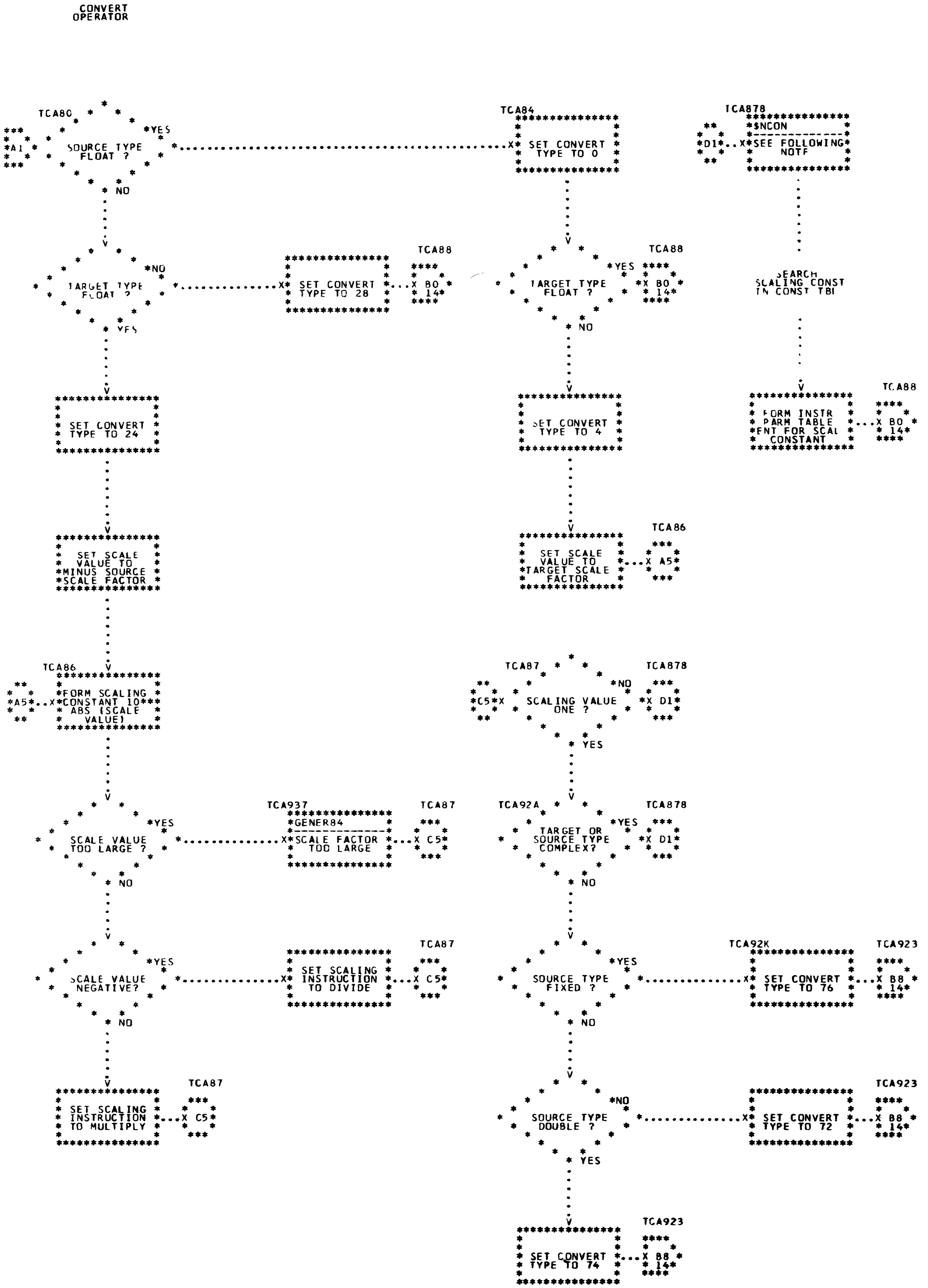


Chart 54. Triad Code Generator (Page 13 of 31)

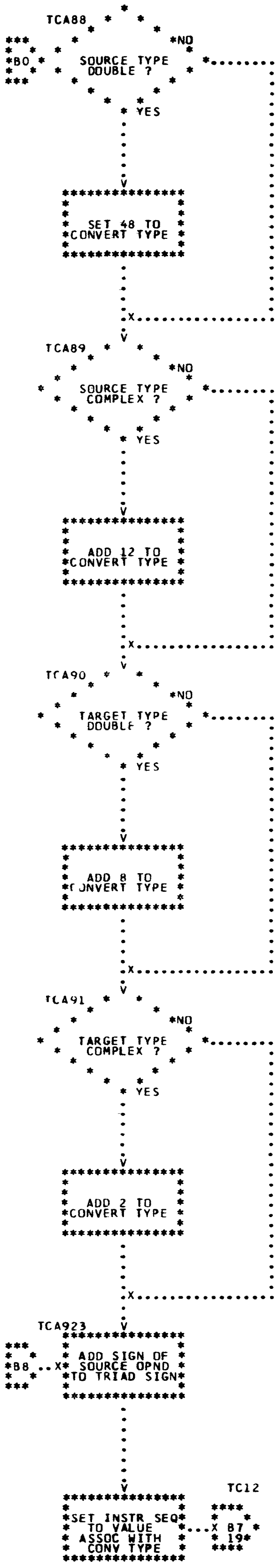
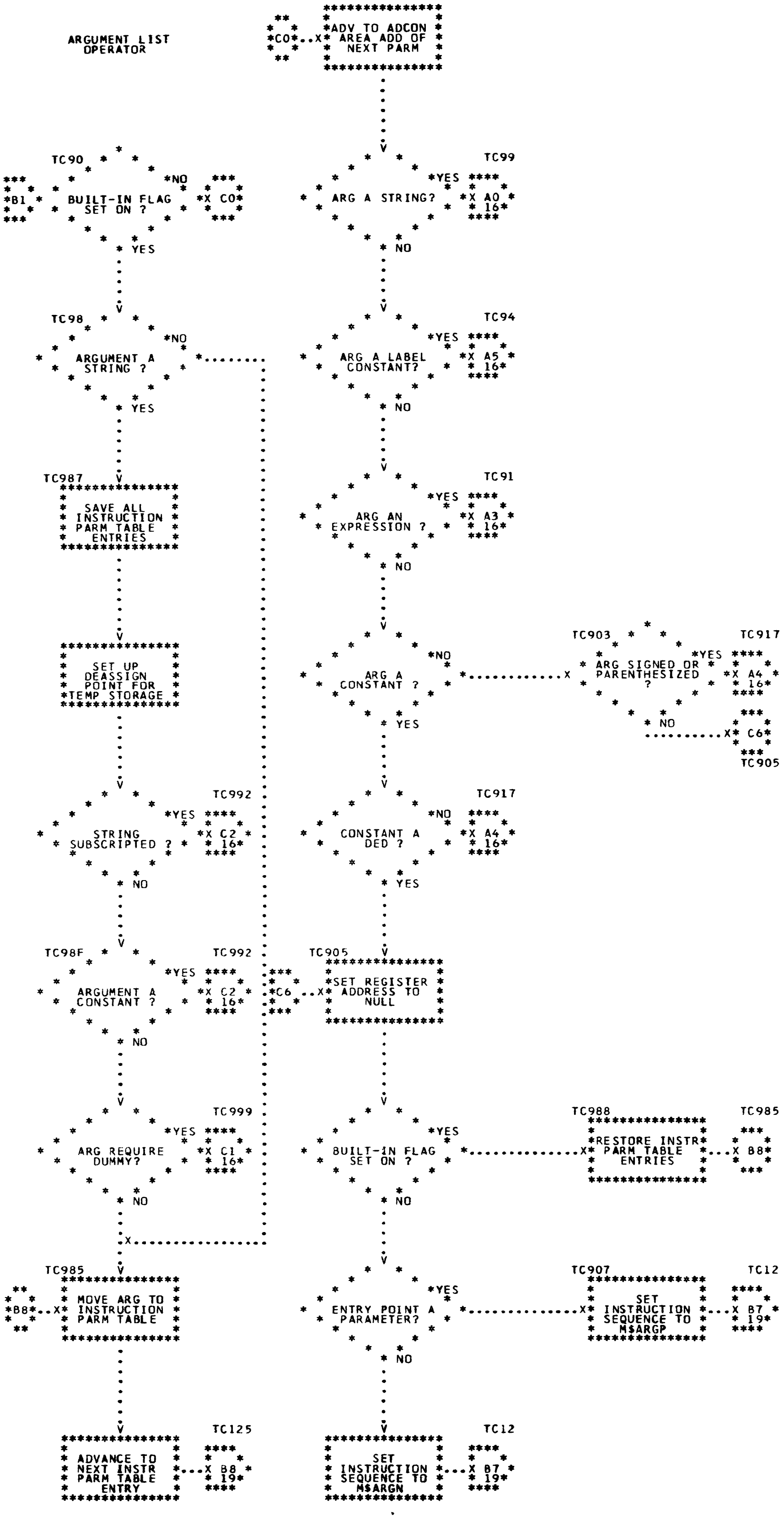
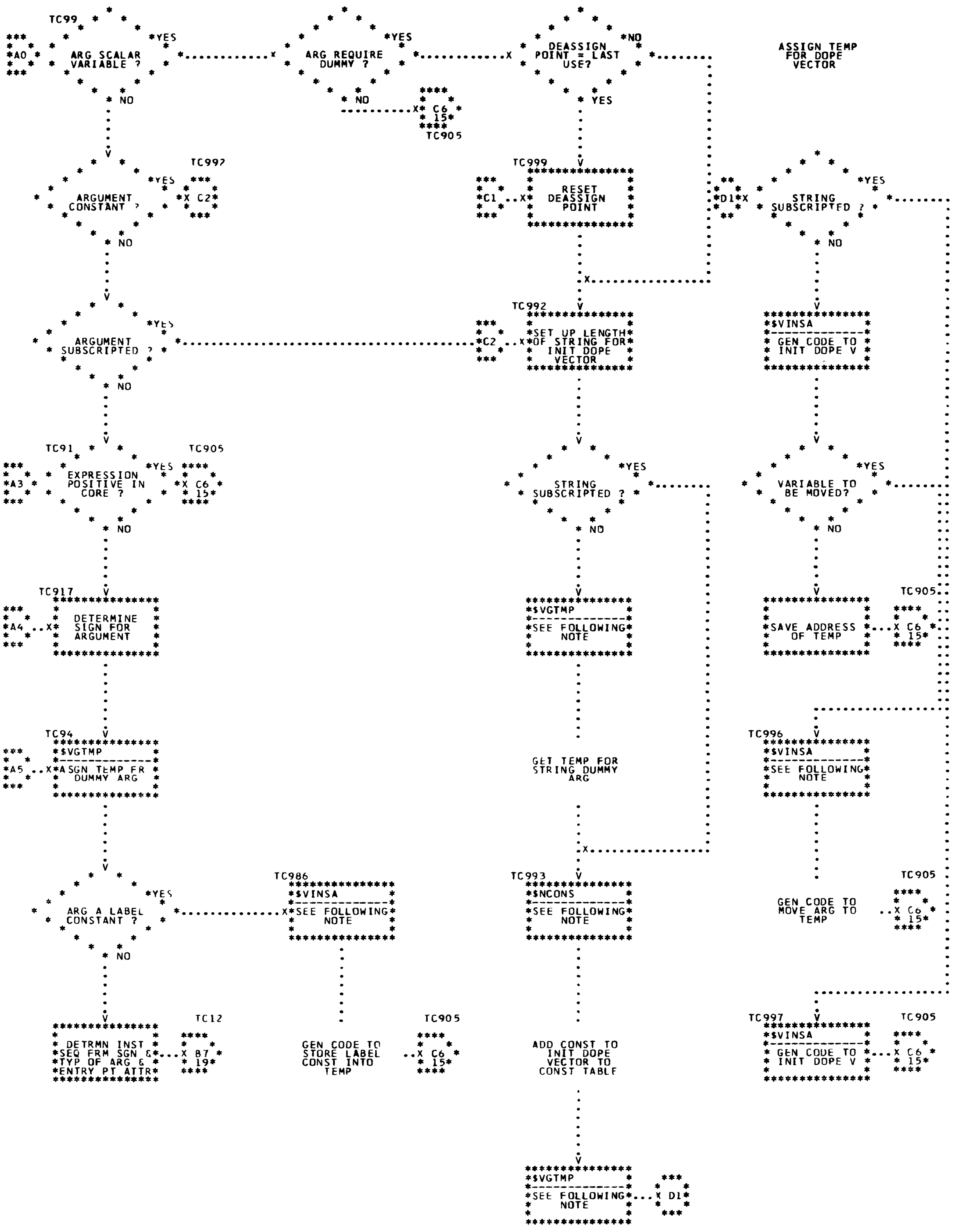
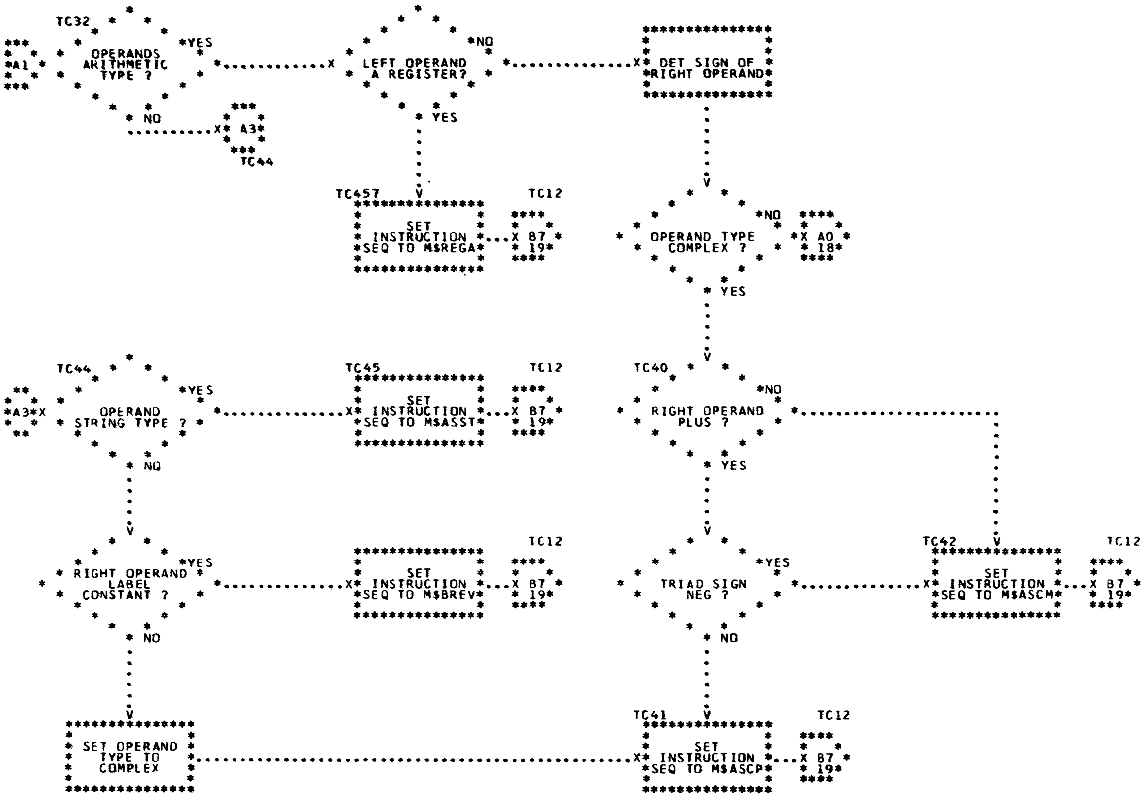


Chart 54. Triad Code Generator (Page 14 of 31)


ARGUMENT LIST OPERATOR
CO
ADV TO ADCON AREA ADD OF NEXT PARM
TC90
B1
BUILT-IN FLAG SET ON ?
NO
X CO
YES
TC99
ARG A STRING?
YES
X A0 16
NO
TC98
ARGUMENT A STRING ?
NO
YES
TC94
ARG A LABEL CONSTANT?
YES
X A5 16
NO
TC987
SAVE ALL INSTRUCTION PARM TABLE ENTRIES
TC91
ARG AN EXPRESSION ?
YES
X A3 16
NO
SET UP DEASSIGN POINT FOR TEMP STORAGE
ARG A CONSTANT ?
NO
YES
TC903
ARG SIGNED OR PARENTHESIZED ?
TC917
YES
X A4 16
NO
X C6
TC905
TC992
STRING SUBSCRIPTED ?
YES
X C2 16
NO
TC917
CONSTANT A DED ?
NO
X A4 16
YES
TC98F
ARGUMENT A CONSTANT ?
TC992
YES
X C2 16
NO
TC905
C6
SET REGISTER ADDRESS TO NULL
TC999
ARG REQUIRE DUMMY?
YES
X C1 16
NO
BUILT-IN FLAG SET ON ?
YES
NO
TC988
RESTORE INSTR PARM TABLE ENTRIES
TC985
X B8
TC985
B8
MOVE ARG TO INSTRUCTION PARM TABLE
ENTRY POINT A PARAMETER?
YES
NO
TC907
SET INSTRUCTION SEQUENCE TO M$ARGP
TC12
X B7 19
TC125
ADVANCE TO NEXT INSTR PARM TABLE ENTRY
X B8 19
TC12
SET INSTRUCTION SEQUENCE TO M$ARGN
X B7 19


TC99
A0
ARG SCALAR VARIABLE ?
YES
NO
ARG REQUIRE DUMMY ?
YES
NO
C6 15
TC905
DEASSIGN POINT = LAST USE?
NO
YES
ASSIGN TEMP FOR DOPE VECTOR
TC992
ARGUMENT CONSTANT ?
YES
X C2
NO
TC999
C1
RESET DEASSIGN POINT
D1
STRING SUBSCRIPTED ?
YES
NO
ARGUMENT SUBSCRIPTED ?
YES
NO
TC992
C2
SET UP LENGTH OF STRING FOR INIT DOPE VECTOR
$VINSA
GEN CODE TO INIT DOPE V
TC91
A3
EXPRESSION POSITIVE IN CORE ?
YES
TC905
X C6 15
NO
STRING SUBSCRIPTED ?
YES
NO
VARIABLE TO BE MOVED?
YES
NO
TC917
A4
DETERMINE SIGN FOR ARGUMENT
$VGTMP
SEE FOLLOWING NOTE
SAVE ADDRESS OF TEMP
TC905
X C6 15
TC94
A5
$VGTMP
ASGN TEMP FR DUMMY ARG
GET TEMP FOR STRING DUMMY ARG
TC996
$VINSA
SEE FOLLOWING NOTE
ARG A LABEL CONSTANT ?
YES
NO
TC986
$VINSA
SEE FOLLOWING NOTE
TC993
$NCONS
SEE FOLLOWING NOTE
GEN CODE TO MOVE ARG TO TEMP
TC905
X C6 15
DETRMN INST SEQ FRM SGN & TYP OF ARG & ENTRY PT ATTR
TC12
X B7 19
GEN CODE TO STORE LABEL CONST INTO TEMP
TC905
X C6 15
ADD CONST TO INIT DOPE VECTOR TO CONST TABLE
TC997
$VINSA
GEN CODE TO INIT DOPE V
TC905
X C6 15
$VGTMP
SEE FOLLOWING NOTE
X D1

ASSIGNMENT
EQUAL
OPERATOR



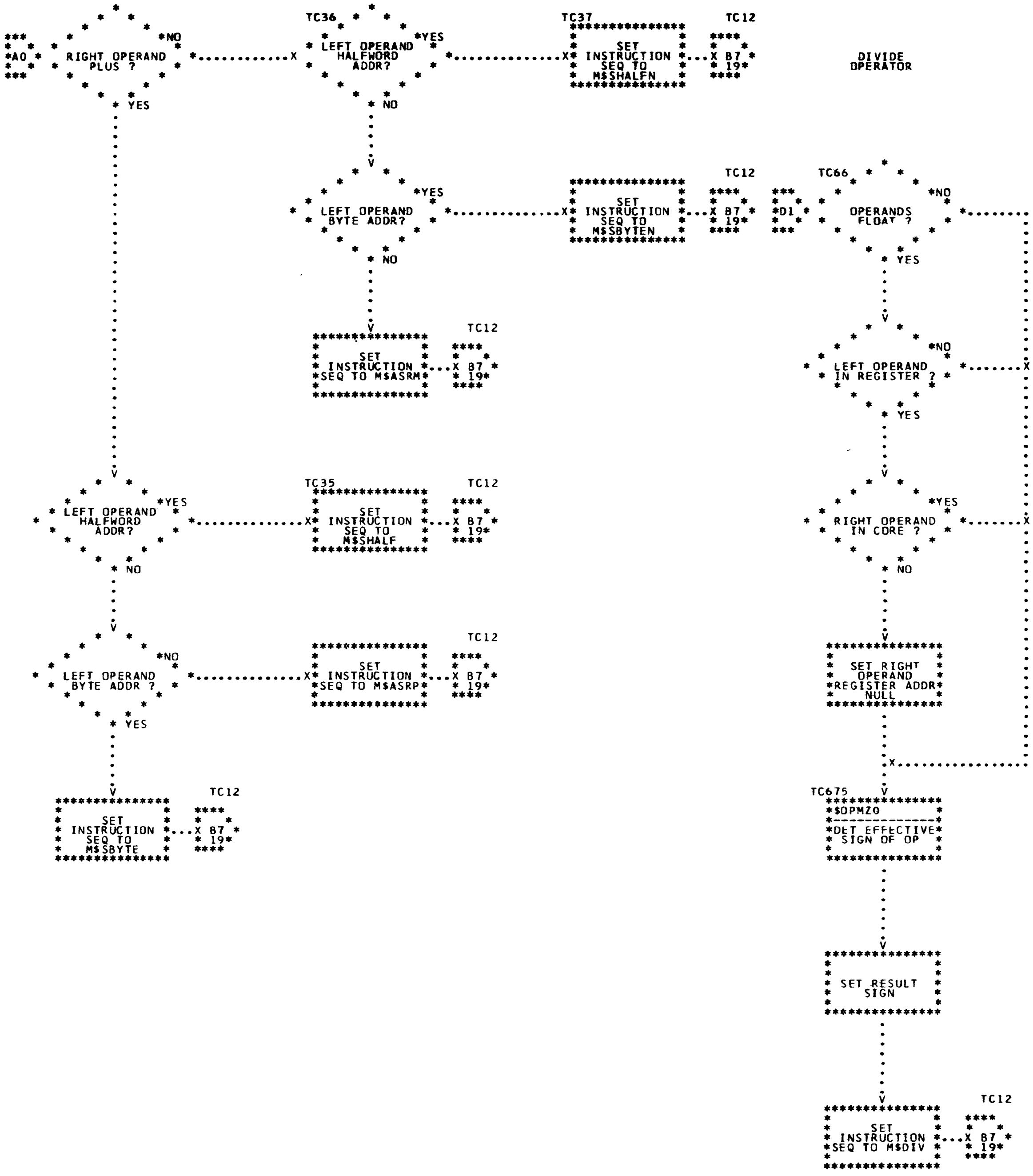
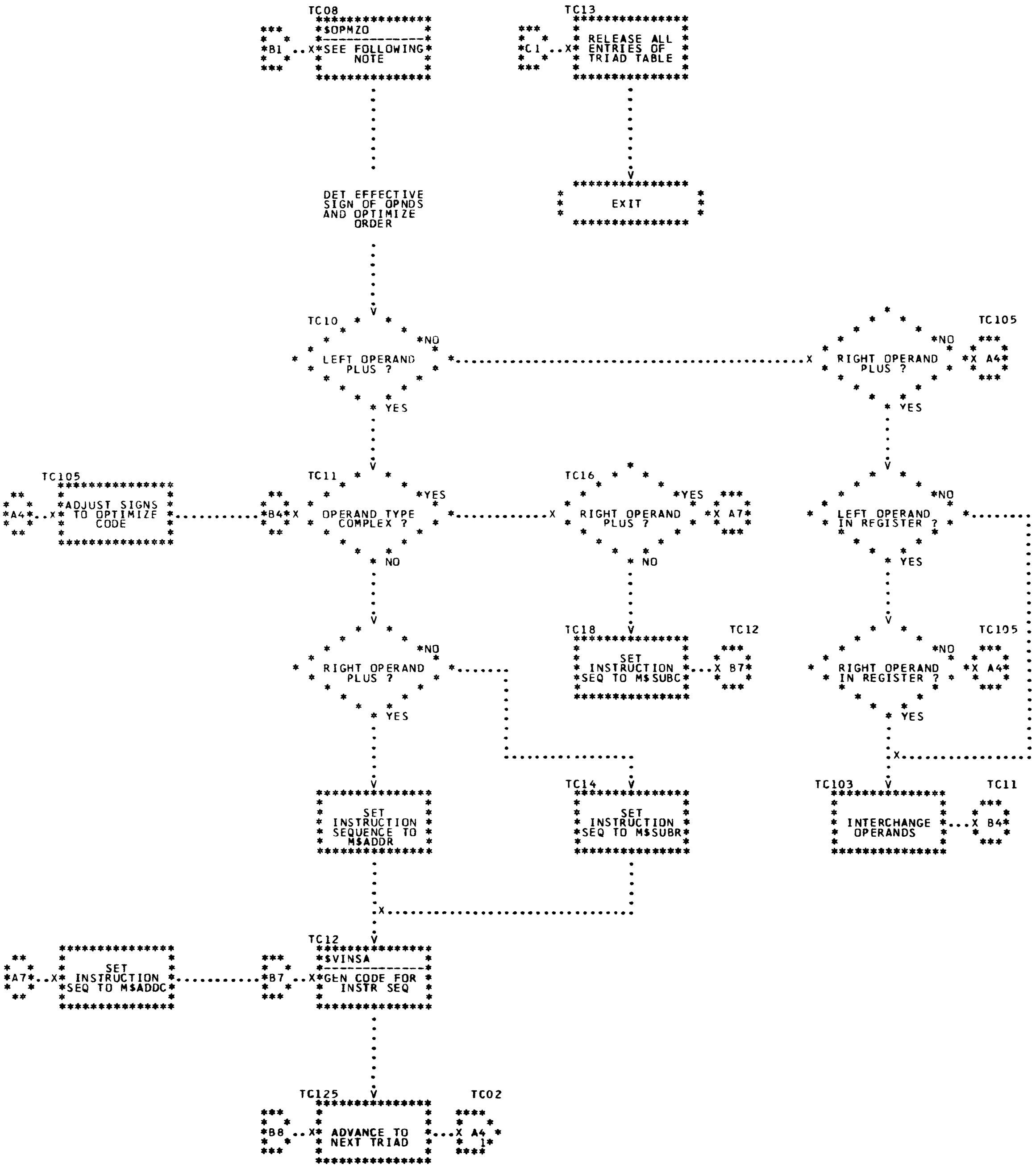


Chart 54. Triad Code Generator (Page 18 of 31)

ADD AND SUBTRACT OPERATIONS



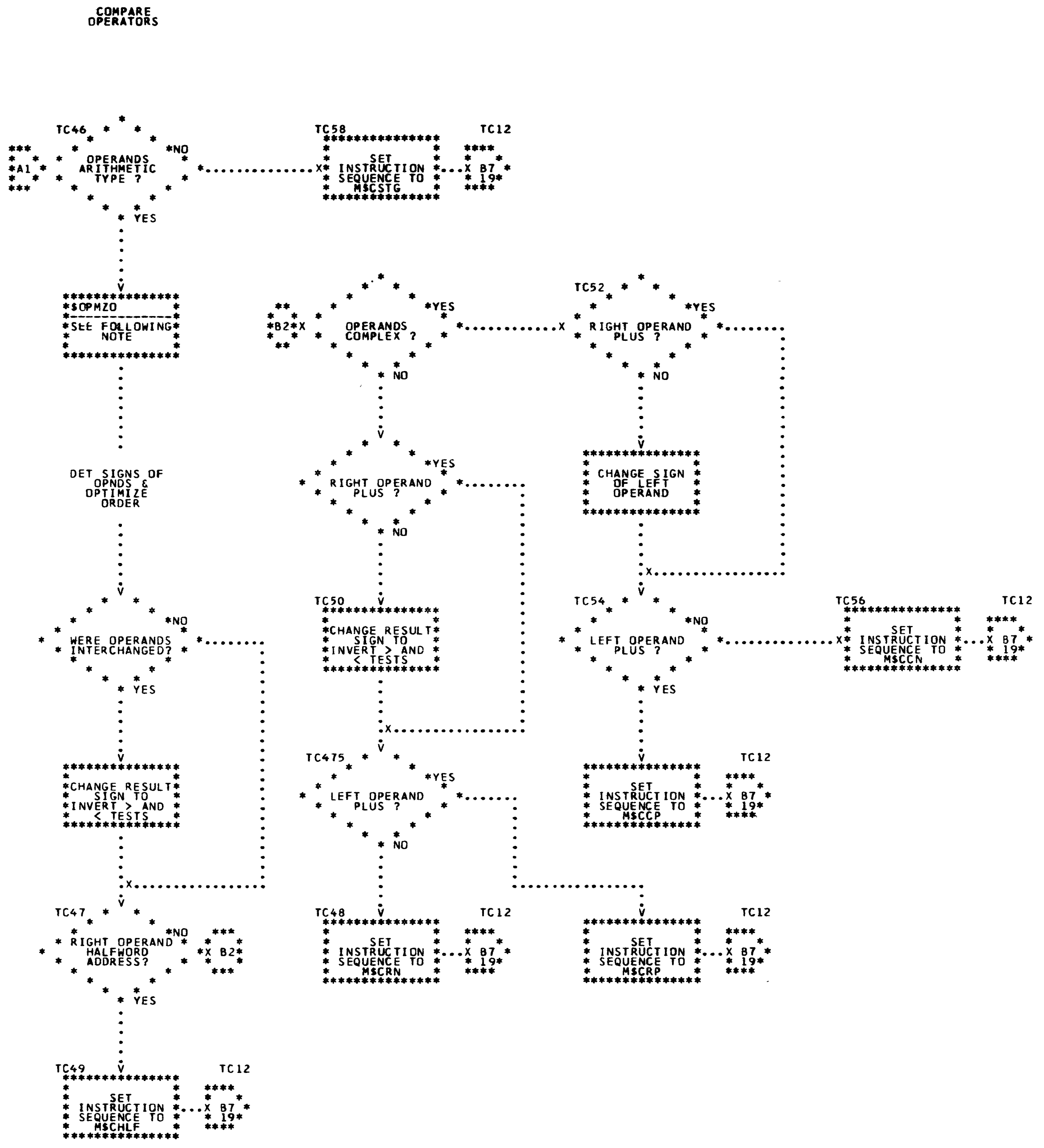


Chart 54. Triad Code Generator (Page 20 of 31)

MULTIPLY
OPERATOR

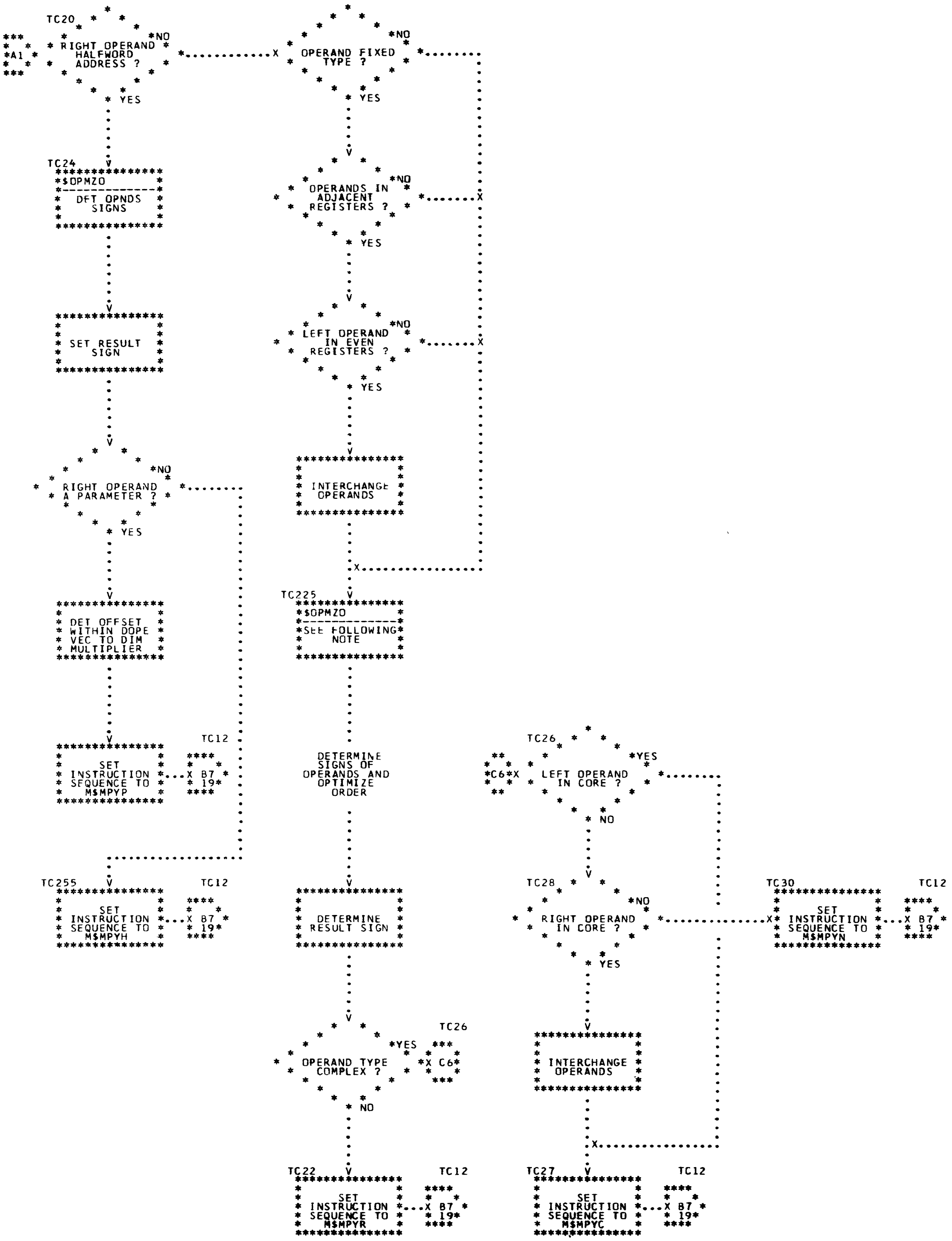


Chart 54. Triad Code Generator (Page 21 of 31)

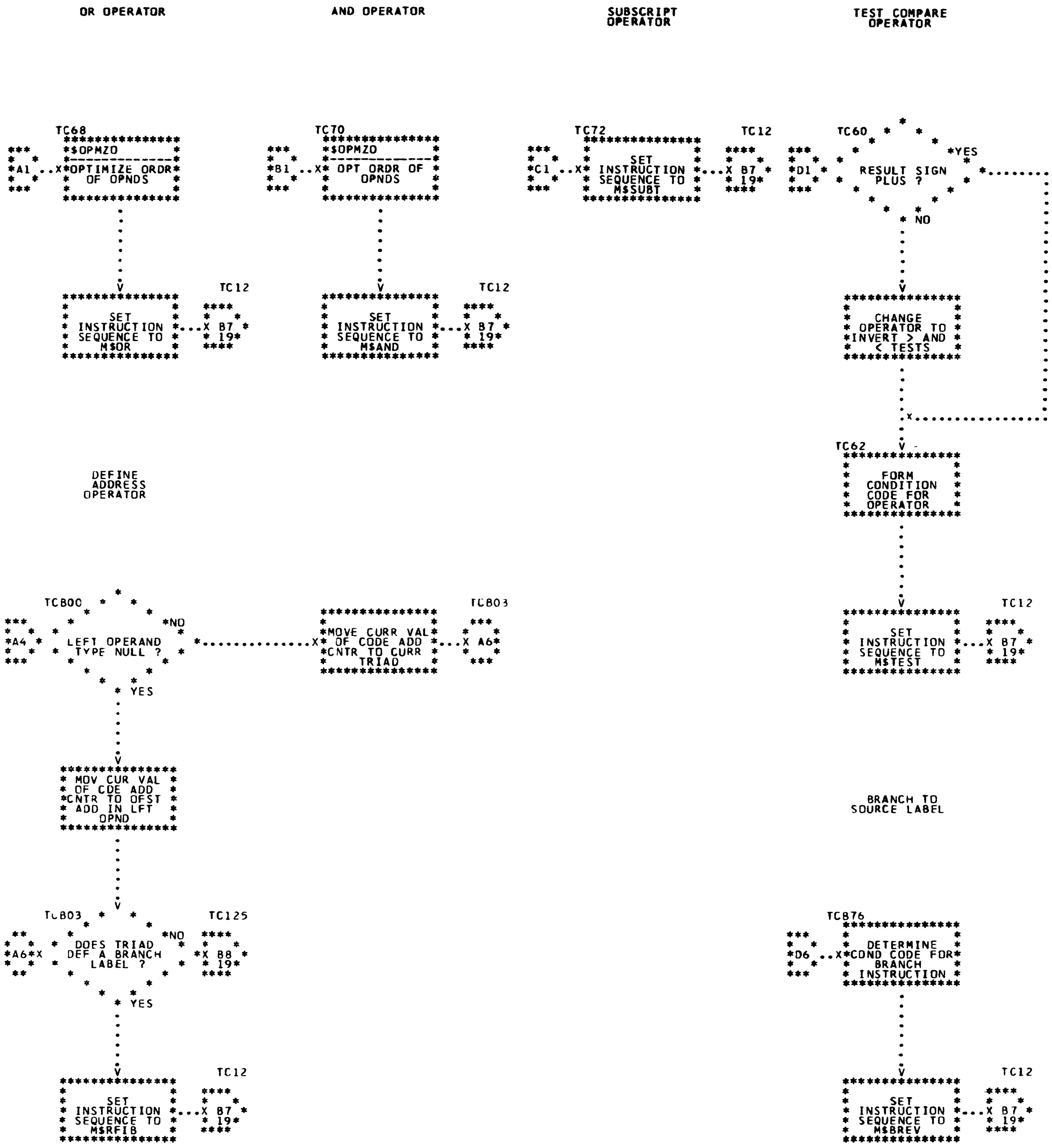
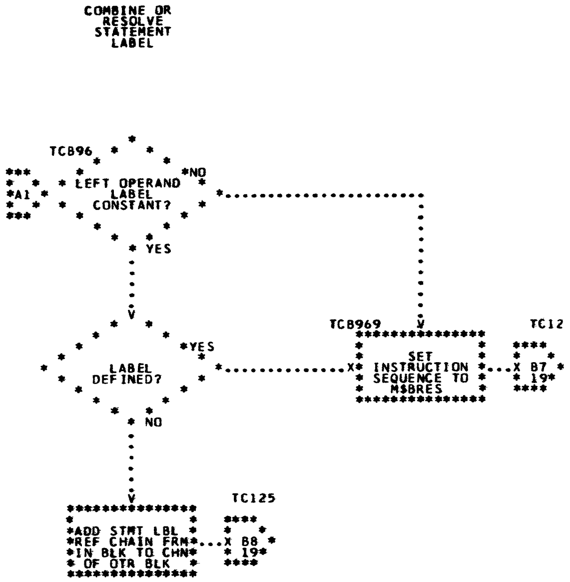
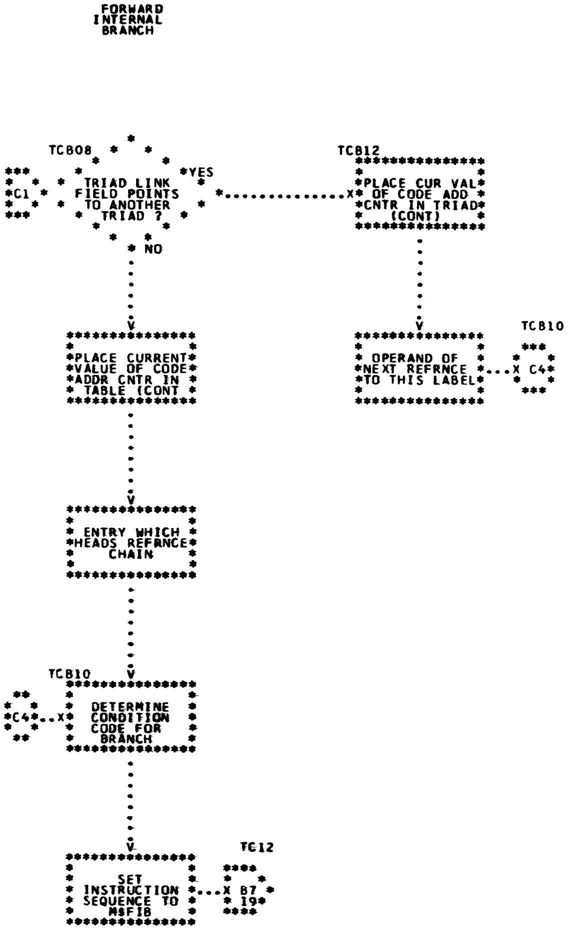
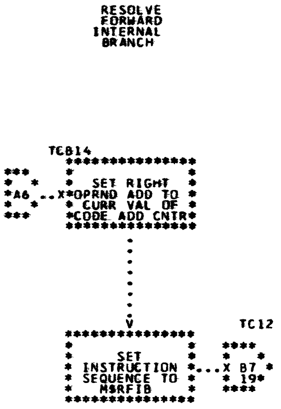


Chart 54. Triad Code Generator (Page 22 of 31)







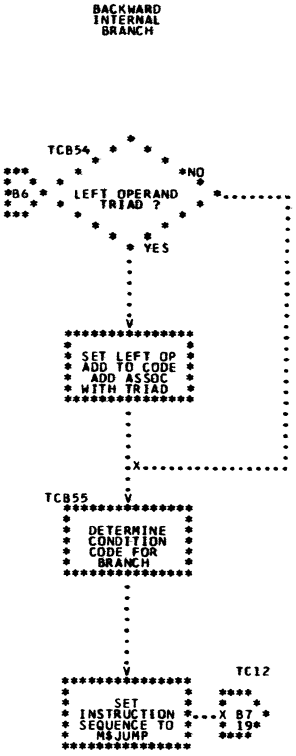


Chart 54. Triad Code Generator (Page 23 of 31)

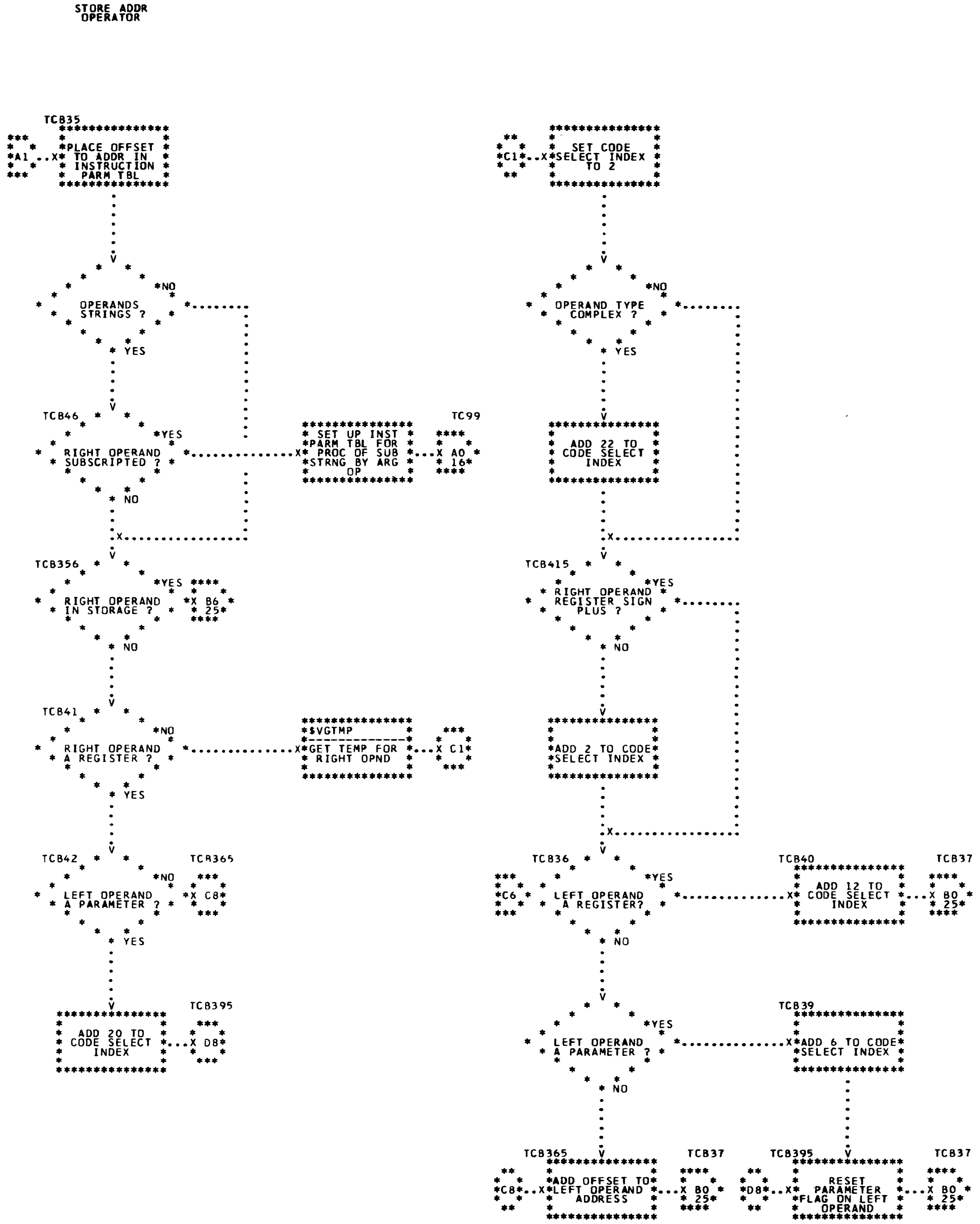


Chart 54. Triad Code Generator (Page 24 of 31)

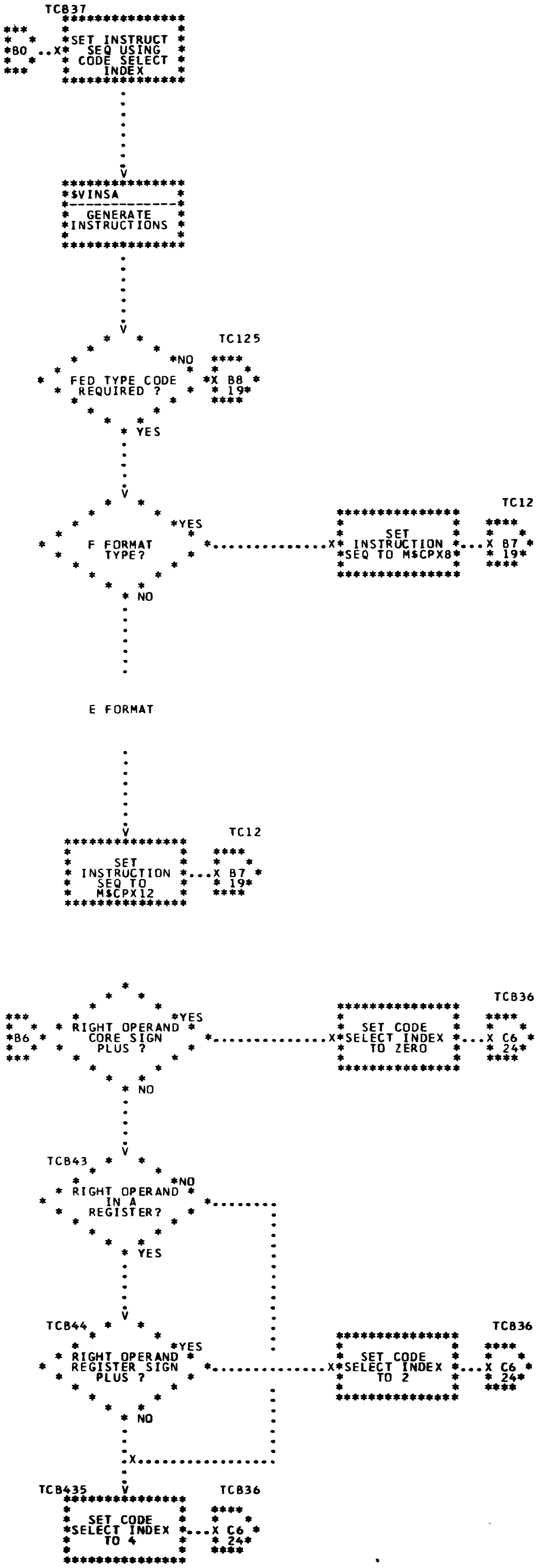


Chart 54. Triad Code Generator (Page 25 of 31)

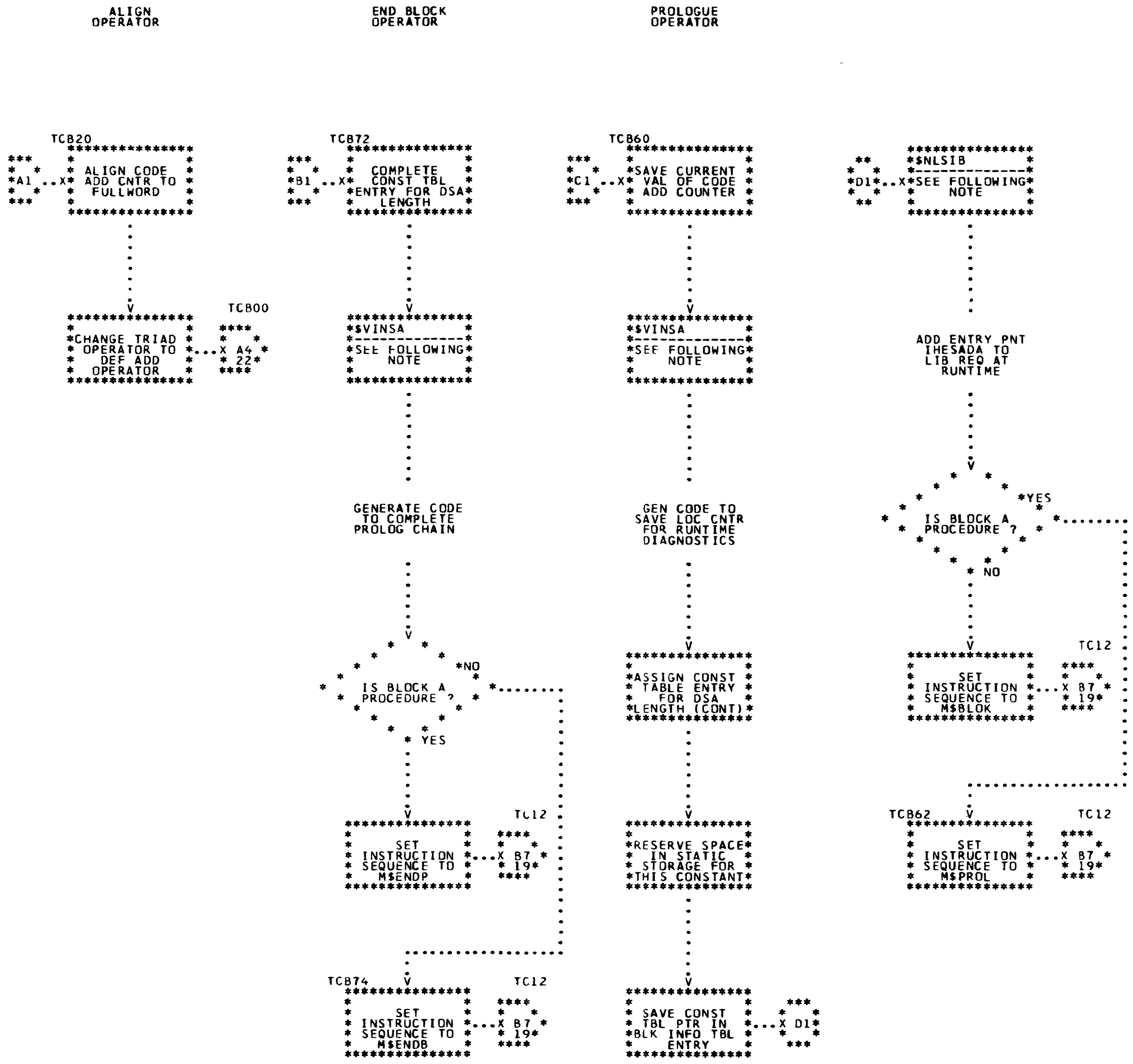


Chart 54. Triad Code Generator (Page 26 of 31)

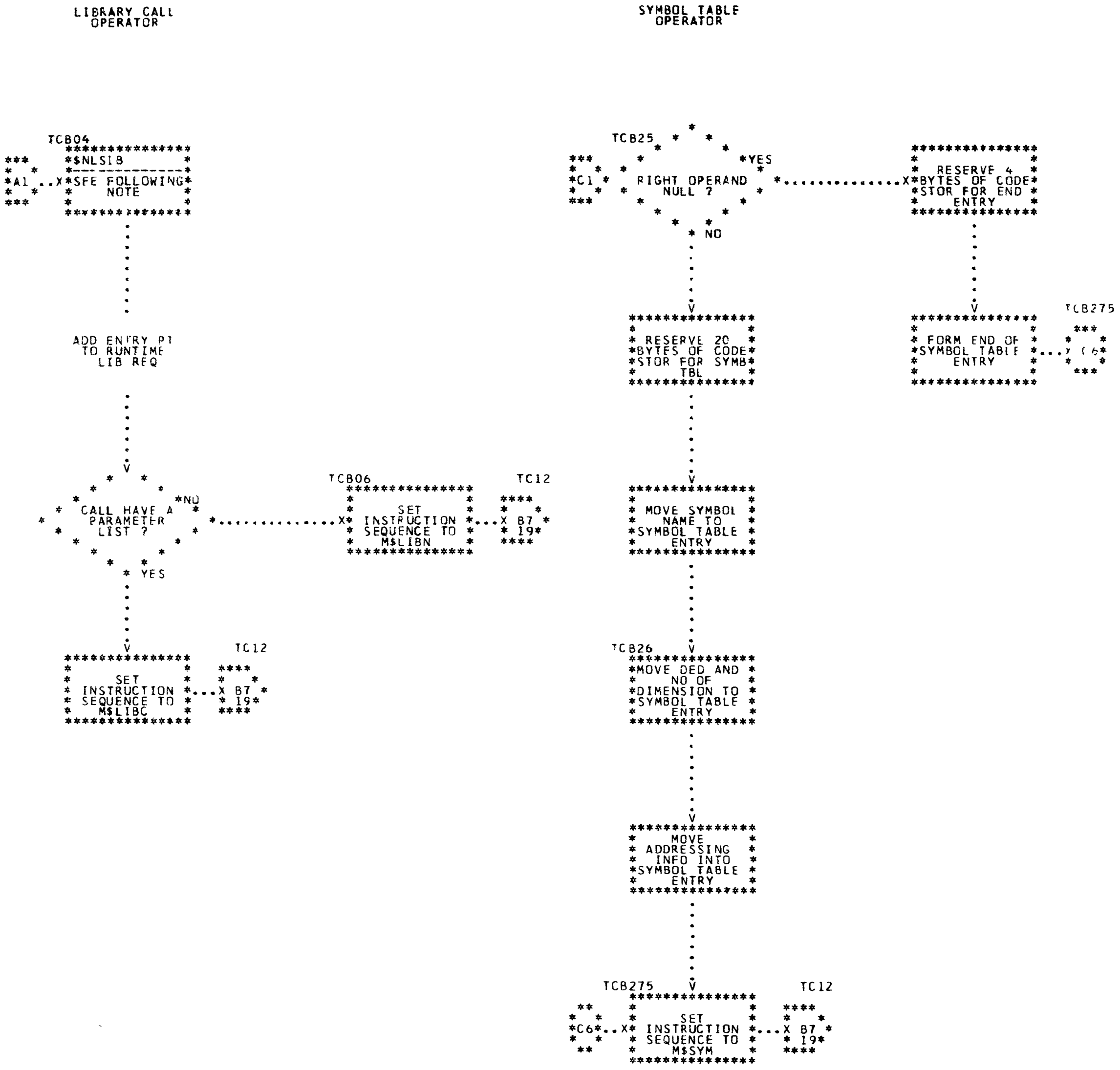


Chart 54. Triad Code Generator (Page 27 of 31)

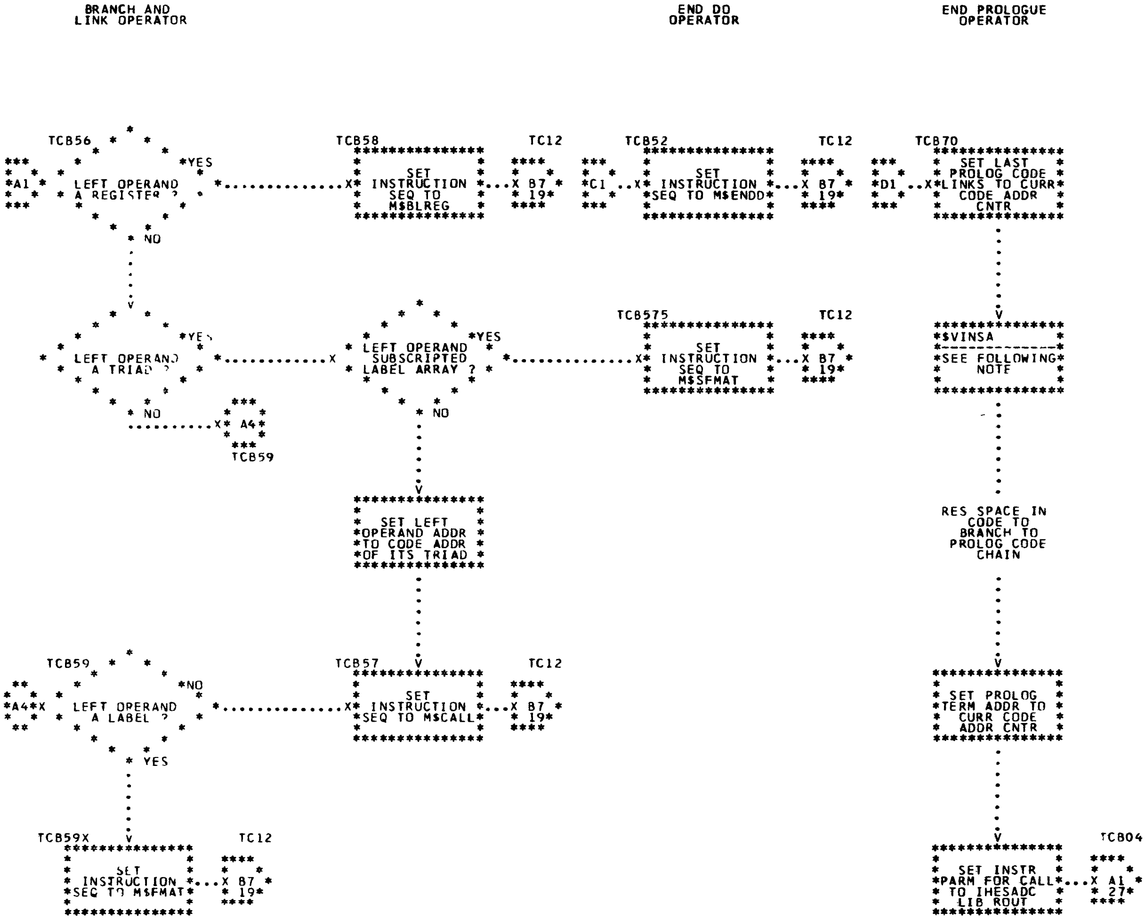
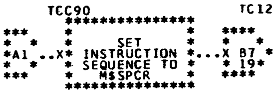
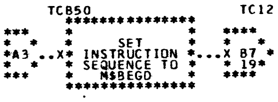


Chart 54. Triad Code Generator (Page 28 of 31)

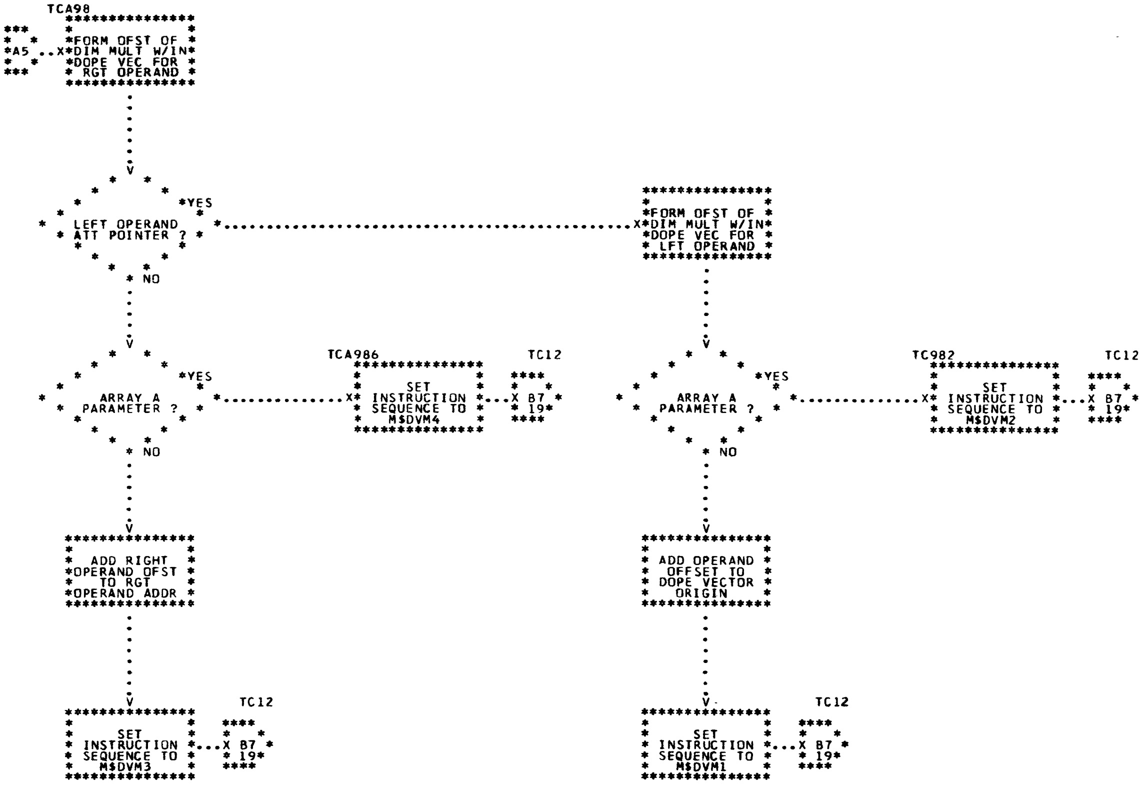
RETURN
OPERATOR

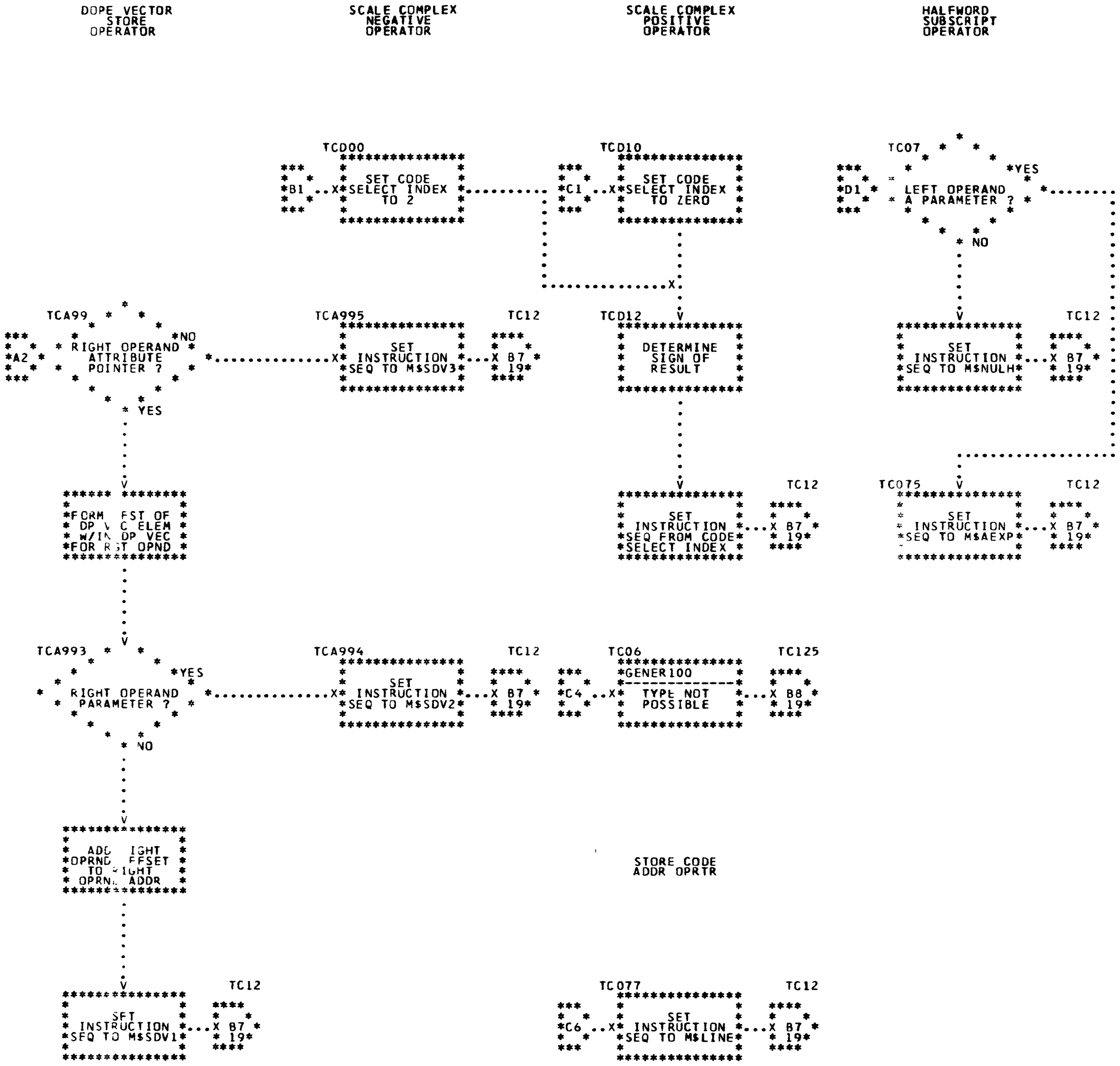


BEGIN DO
OPERATOR

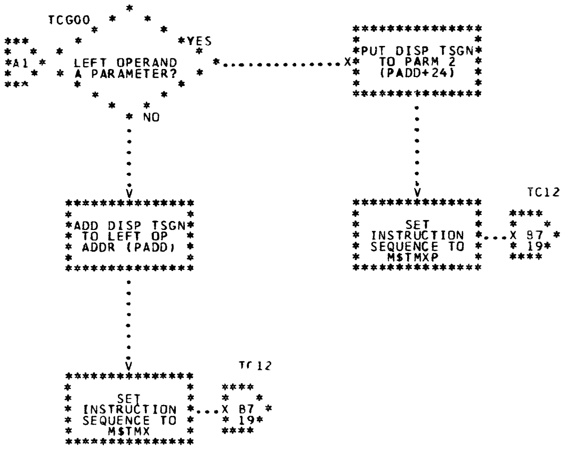


DOPE VECTOR
ELEMENT
MULTIPLY
OPERATOR

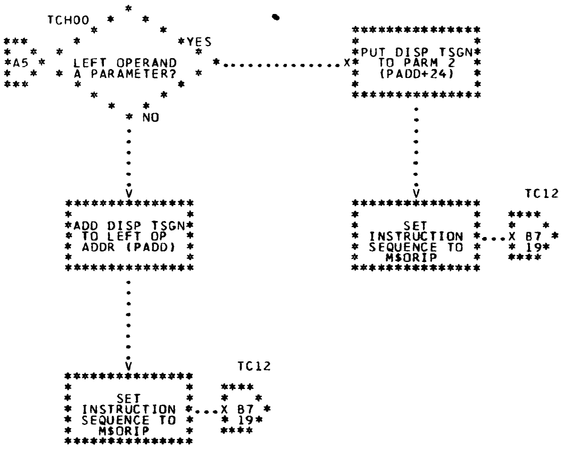



DOPE VECTOR STORE OPERATOR
SCALE COMPLEX NEGATIVE OPERATOR
SCALE COMPLEX POSITIVE OPERATOR
HALFWORD SUBSCRIPT OPERATOR
TCD00
B1
SET CODE SELECT INDEX TO 2
TCD10
C1
SET CODE SELECT INDEX TO ZERO
TC07
D1
LEFT OPERAND A PARAMETER ?
YES
NO
TCA99
A2
RIGHT OPERAND ATTRIBUTE POINTER ?
NO
YES
TCA995
SET INSTRUCTION SEQ TO M$SDV3
TC12
B7 19
TCD12
DETERMINE SIGN OF RESULT
SET INSTRUCTION SEQ TO M$NULH
TC12
B7 19
FORM 1ST OF DP VEC ELEM W/IN DP VEC FOR RGT OPND
SET INSTRUCTION SEQ FROM CODE SELECT INDEX
TC12
B7 19
TC075
SET INSTRUCTION SEQ TO M$AEXP
TC12
B7 19
TCA993
RIGHT OPERAND PARAMETER ?
YES
NO
TCA994
SET INSTRUCTION SEQ TO M$SDV2
TC12
B7 19
TC06
C4
GENER100
TYPE NOT POSSIBLE
TC125
B8 19
ADD RIGHT OPRND OFFSET TO RIGHT OPRND ADDR
STORE CODE ADDR OPRTR
SET INSTRUCTION SEQ TO M$SDV1
TC12
B7 19
TC077
C6
SET INSTRUCTION SEQ TO M$LINE
TC12
B7 19

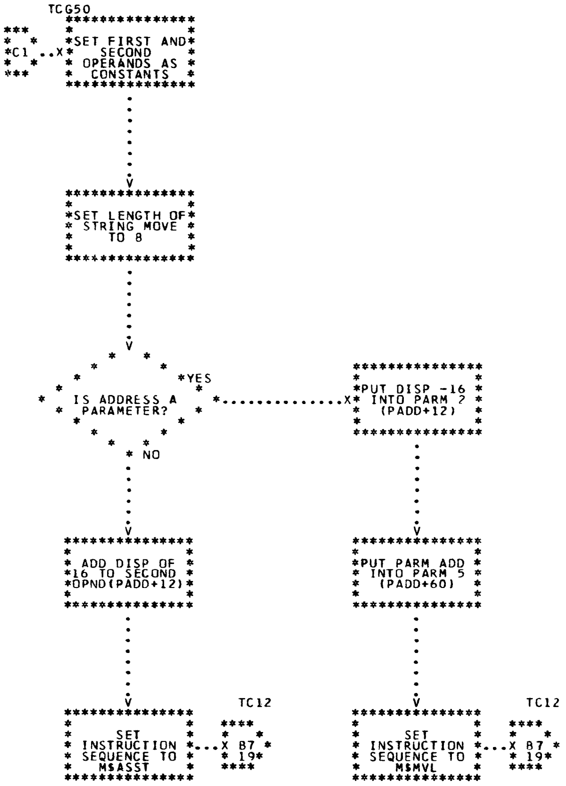
TEST UNDER
MASK



OR IMMEDIATE



TITLE
OPERATOR



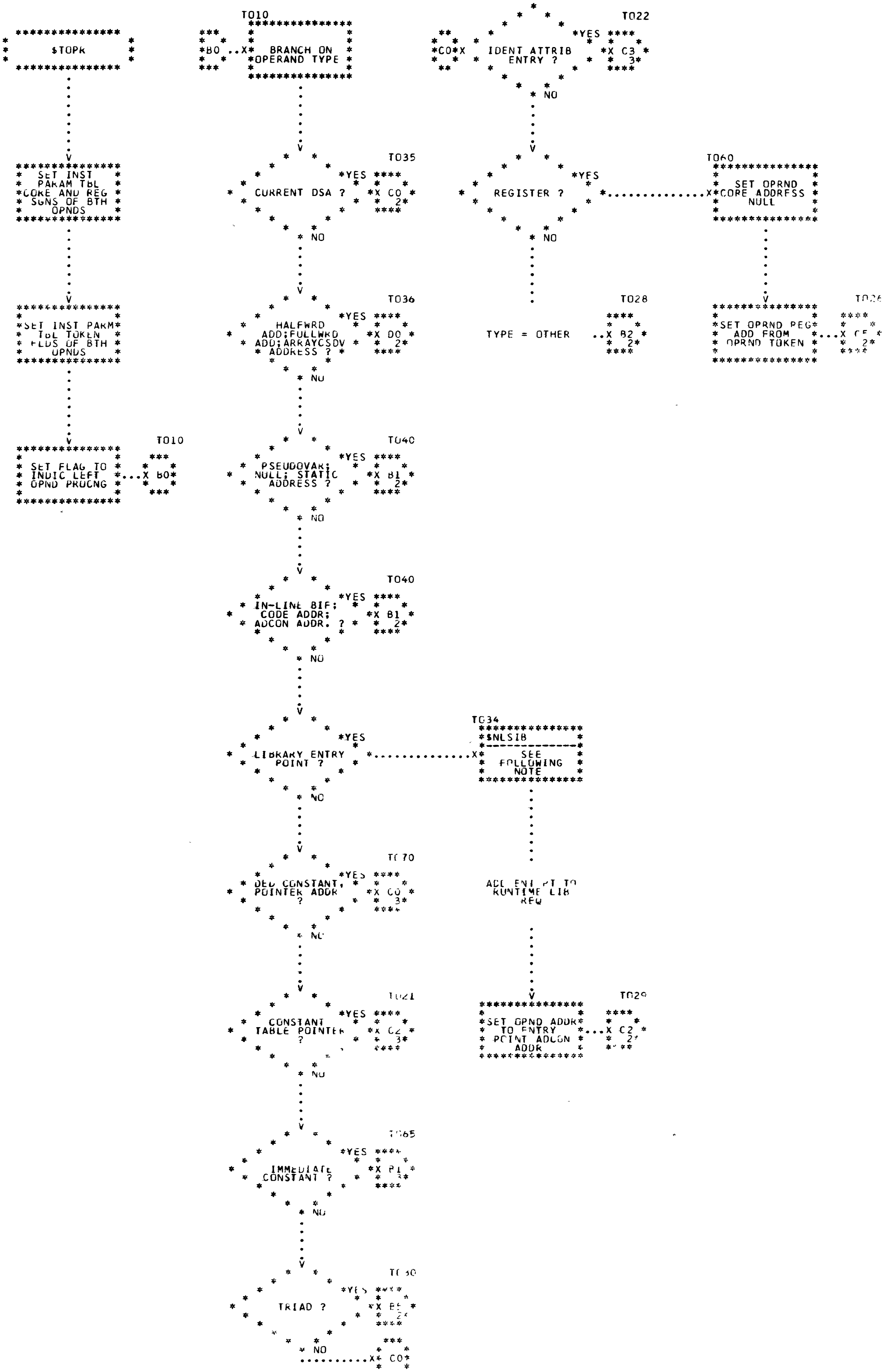


Chart 55. Triad Operand Processor (Page 1 of 3)

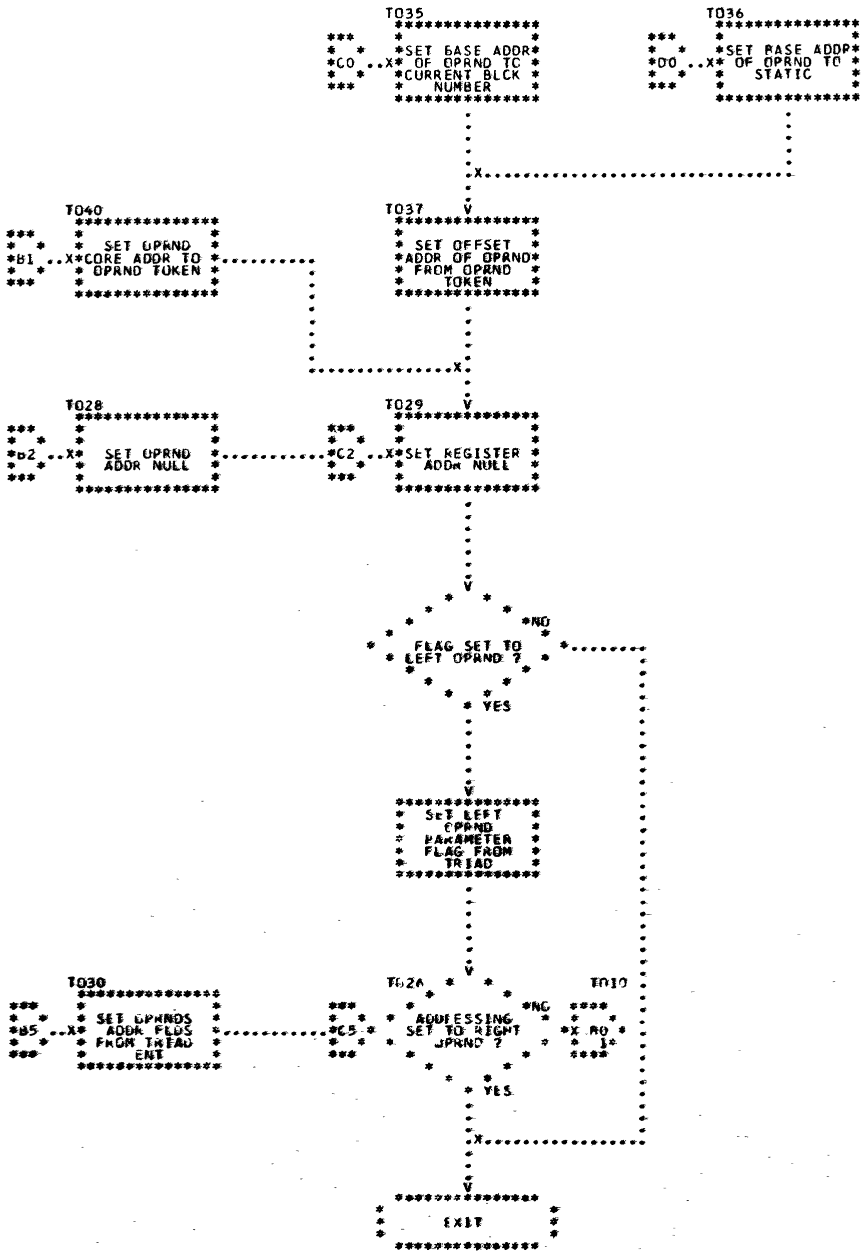


Chart 55. Triad Operand Processor (Page 2 of 3)

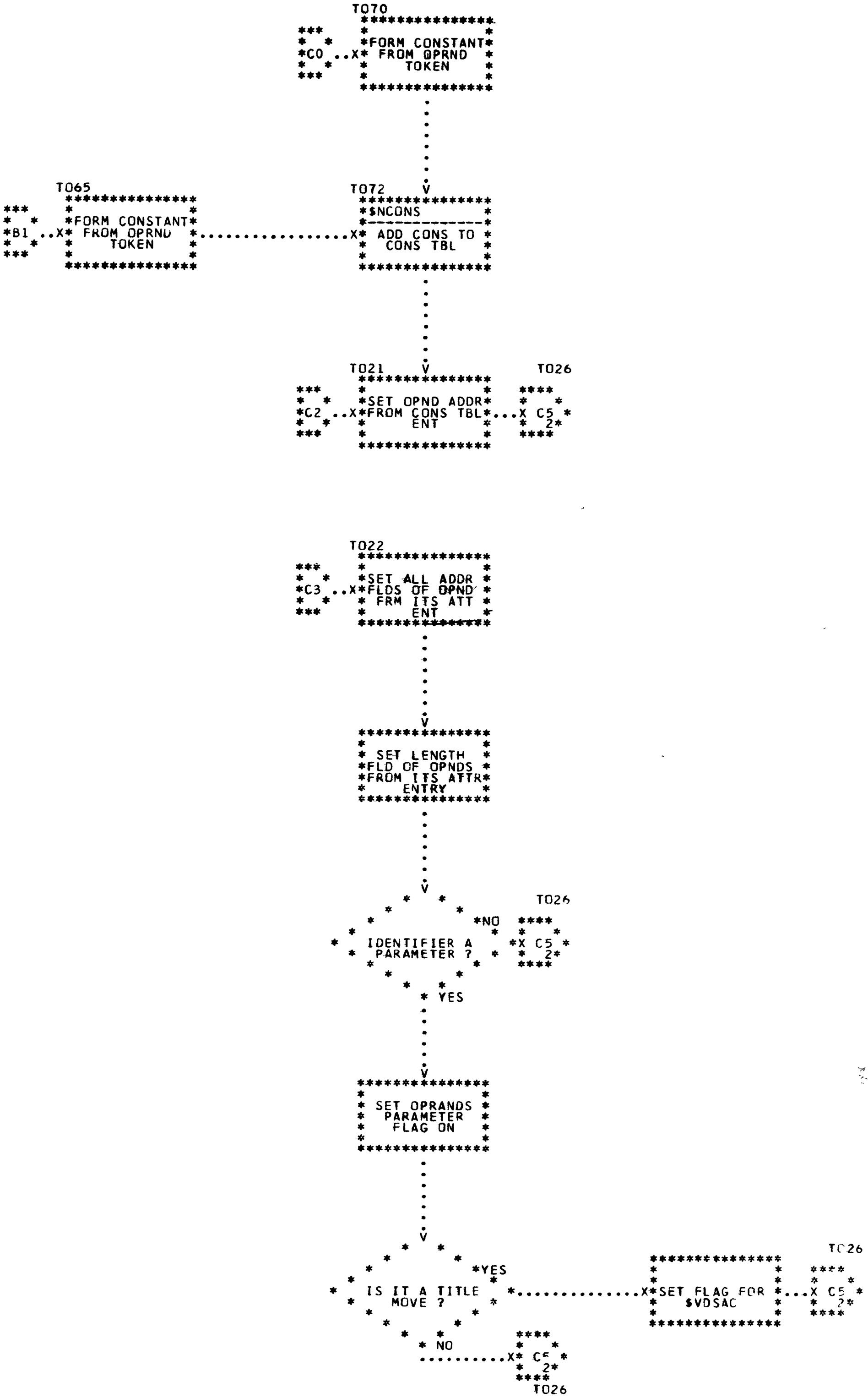


Chart 55. Triad Operand Processor (Page 3 of 3)

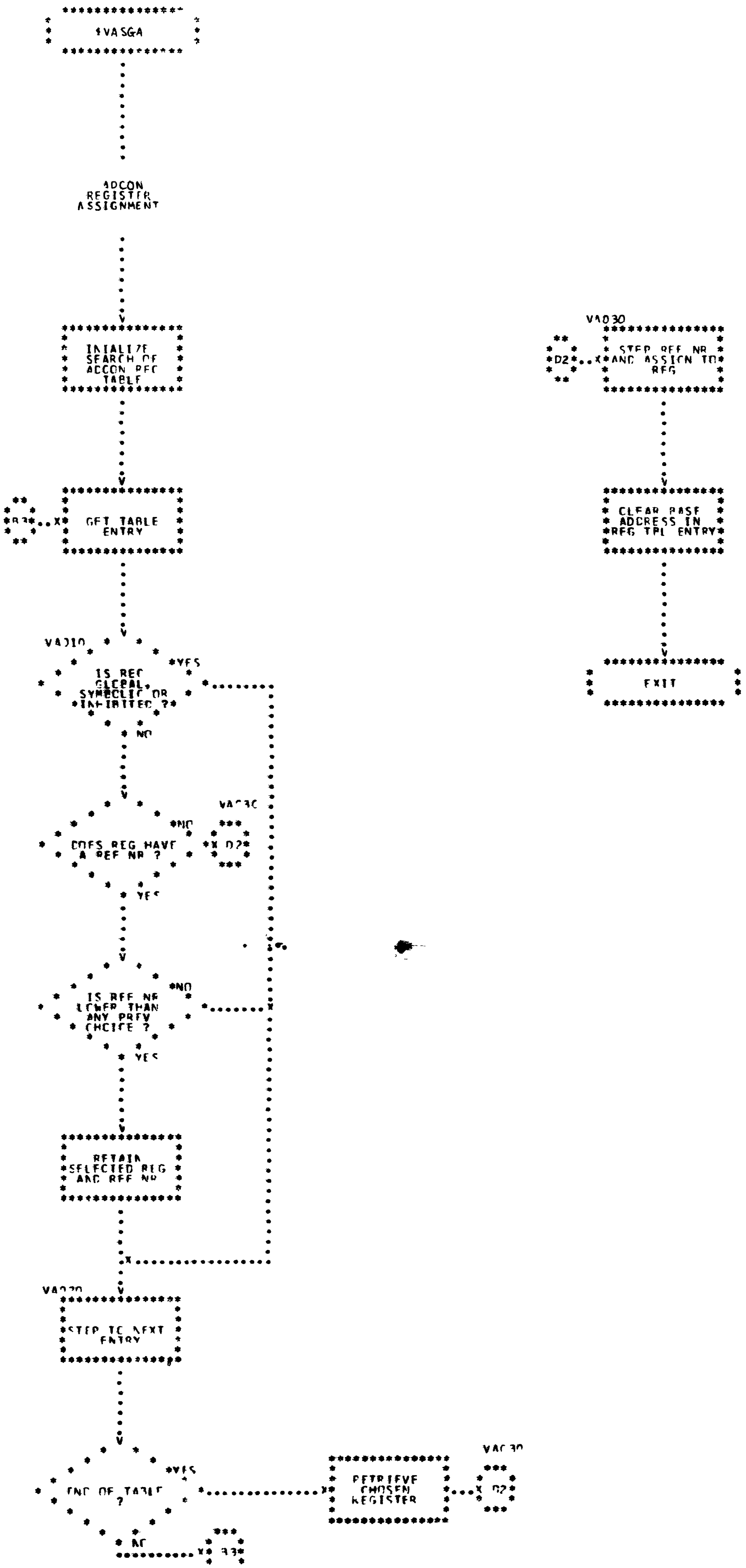


Chart 56. Adcon Register Assignment

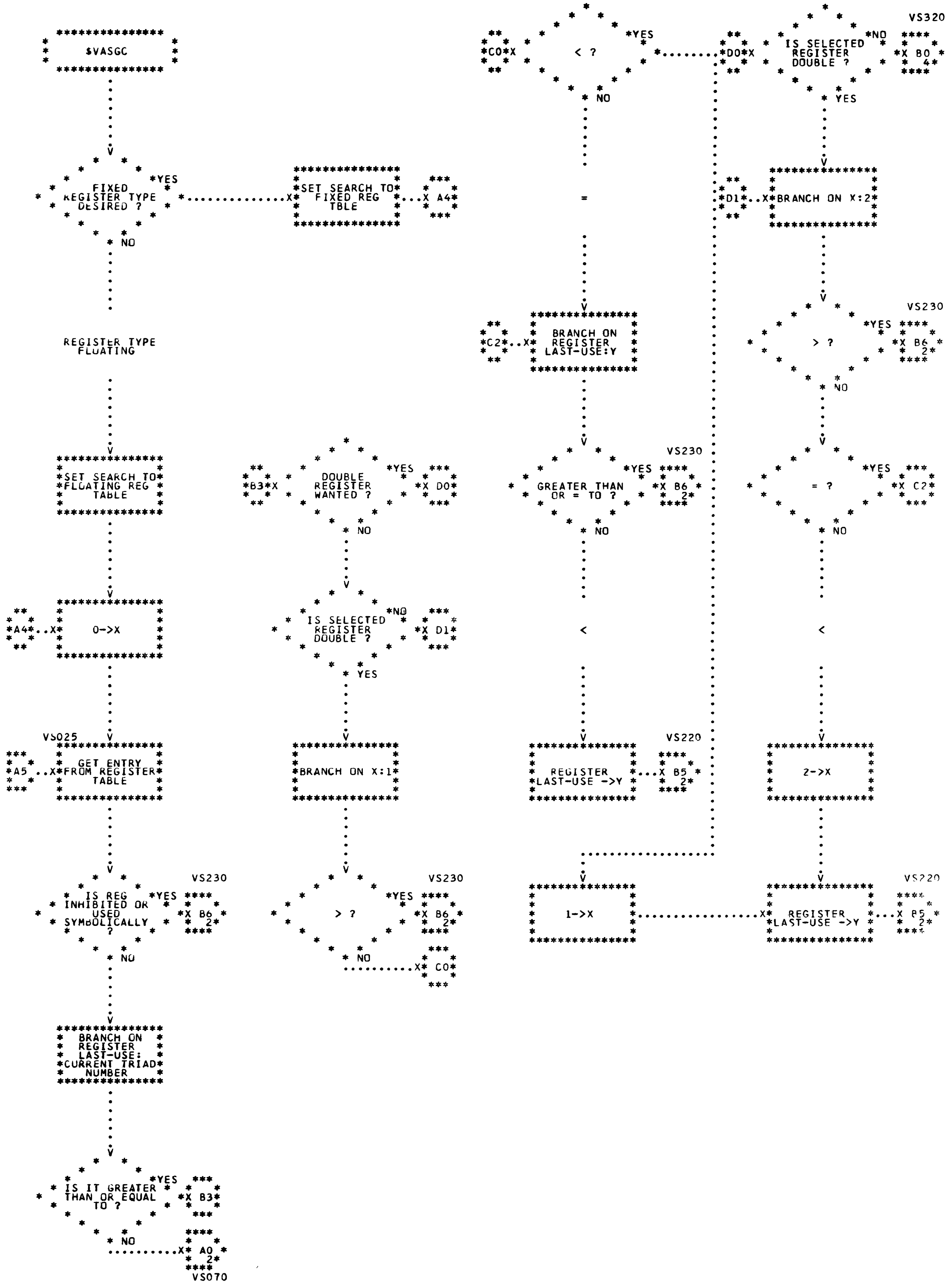


Chart 57. Computational Register Assignment (Page 1 of 8)

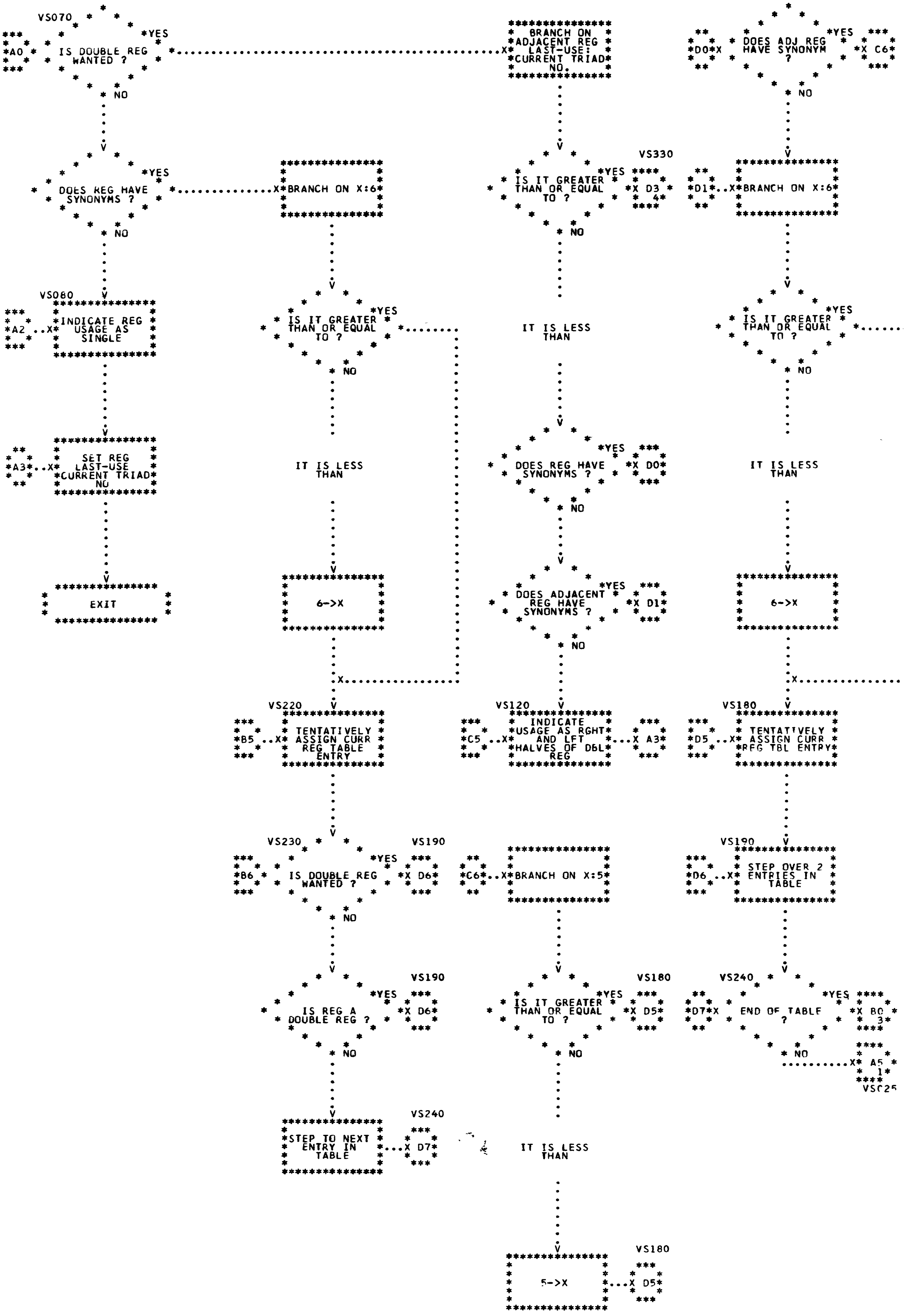


Chart 57. Computational Register Assignment (Page 2 of 8)

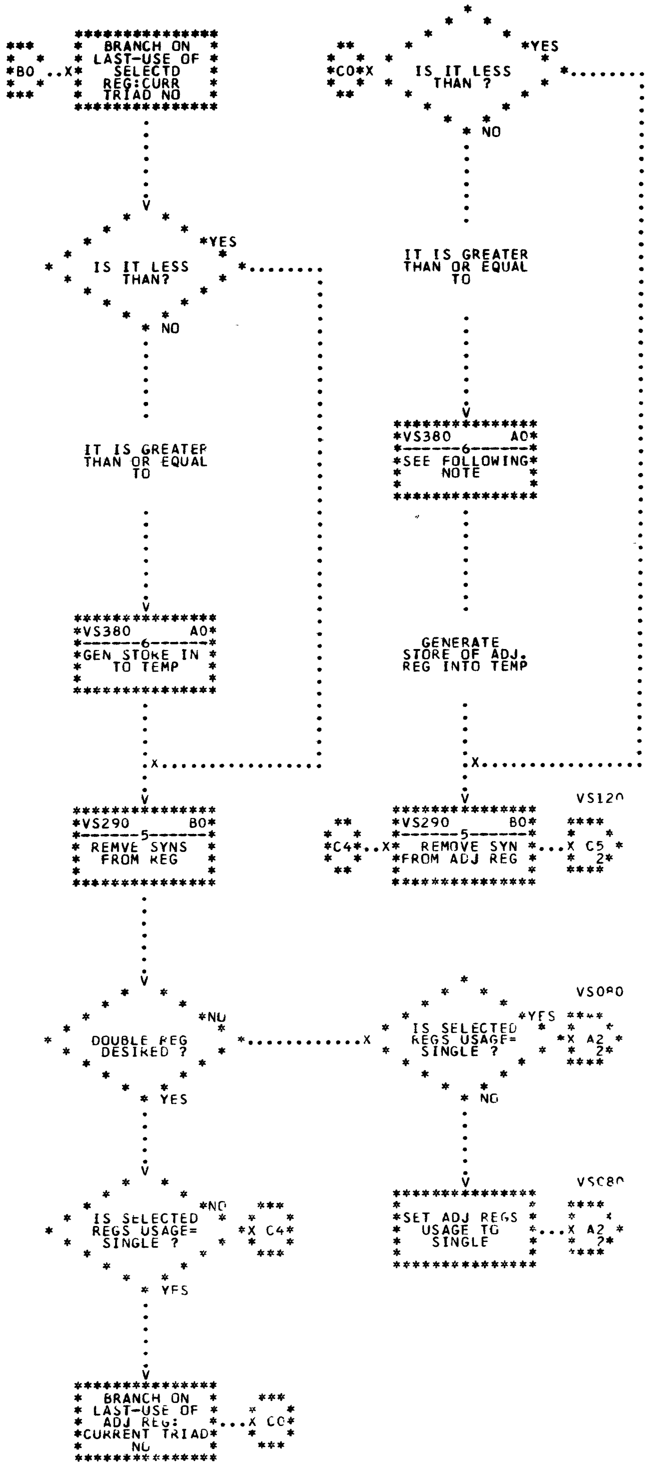


Chart 57. Computational Register Assignment (Page 3 of 8)

]

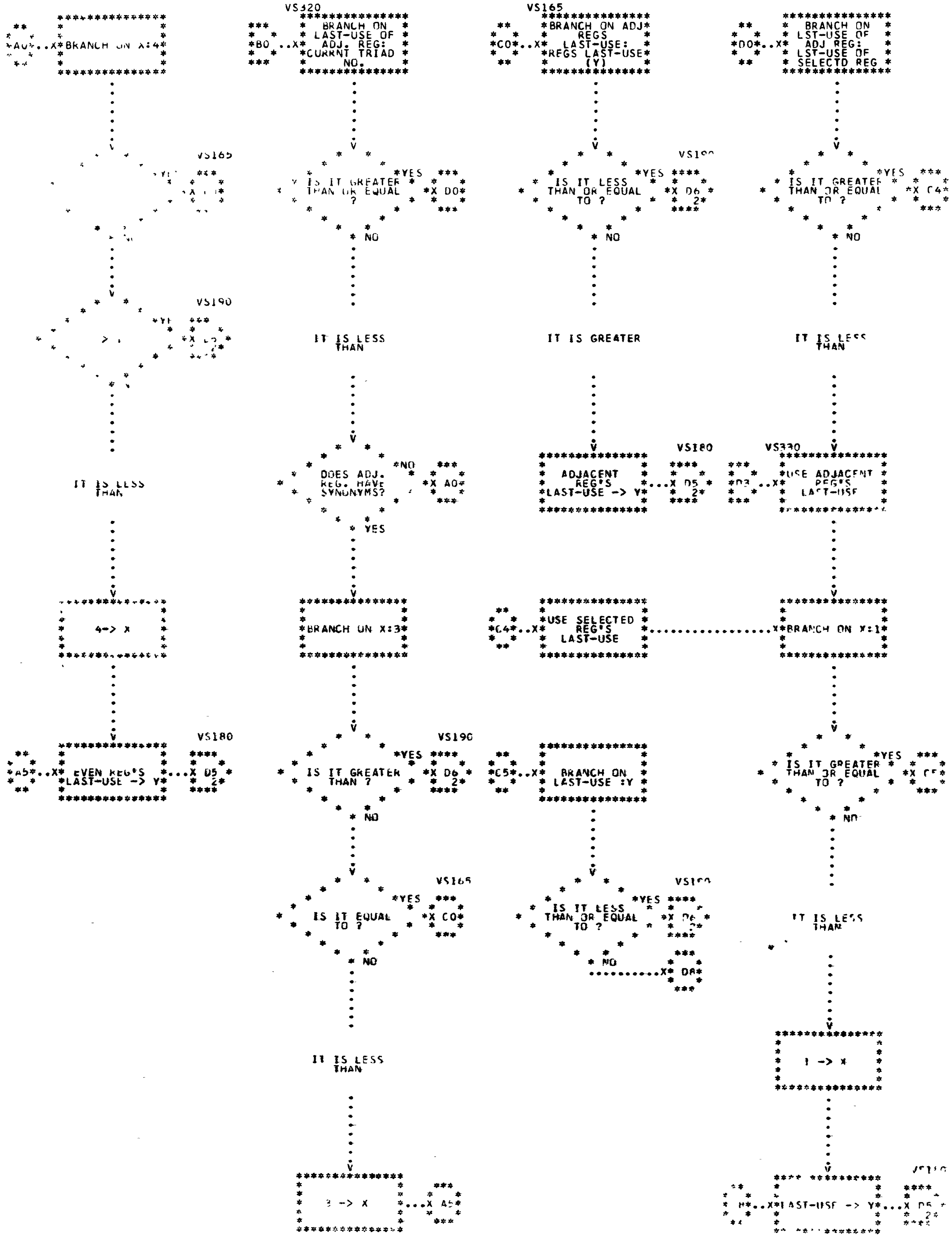


Chart 57. Computational Register Assignment (Page 4 of 8)

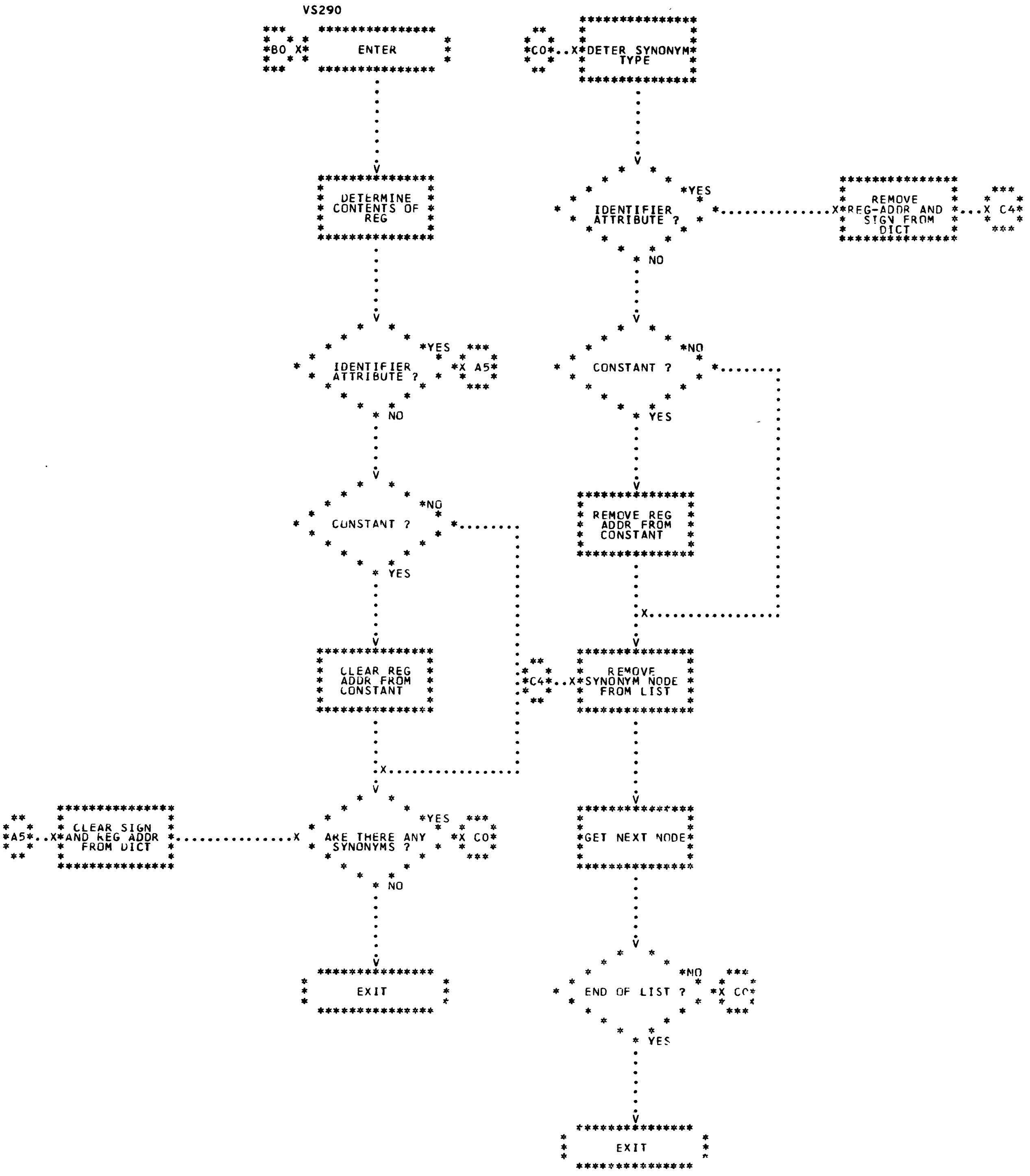


Chart 57. Computational Register Assignment (Page 5 of 8)

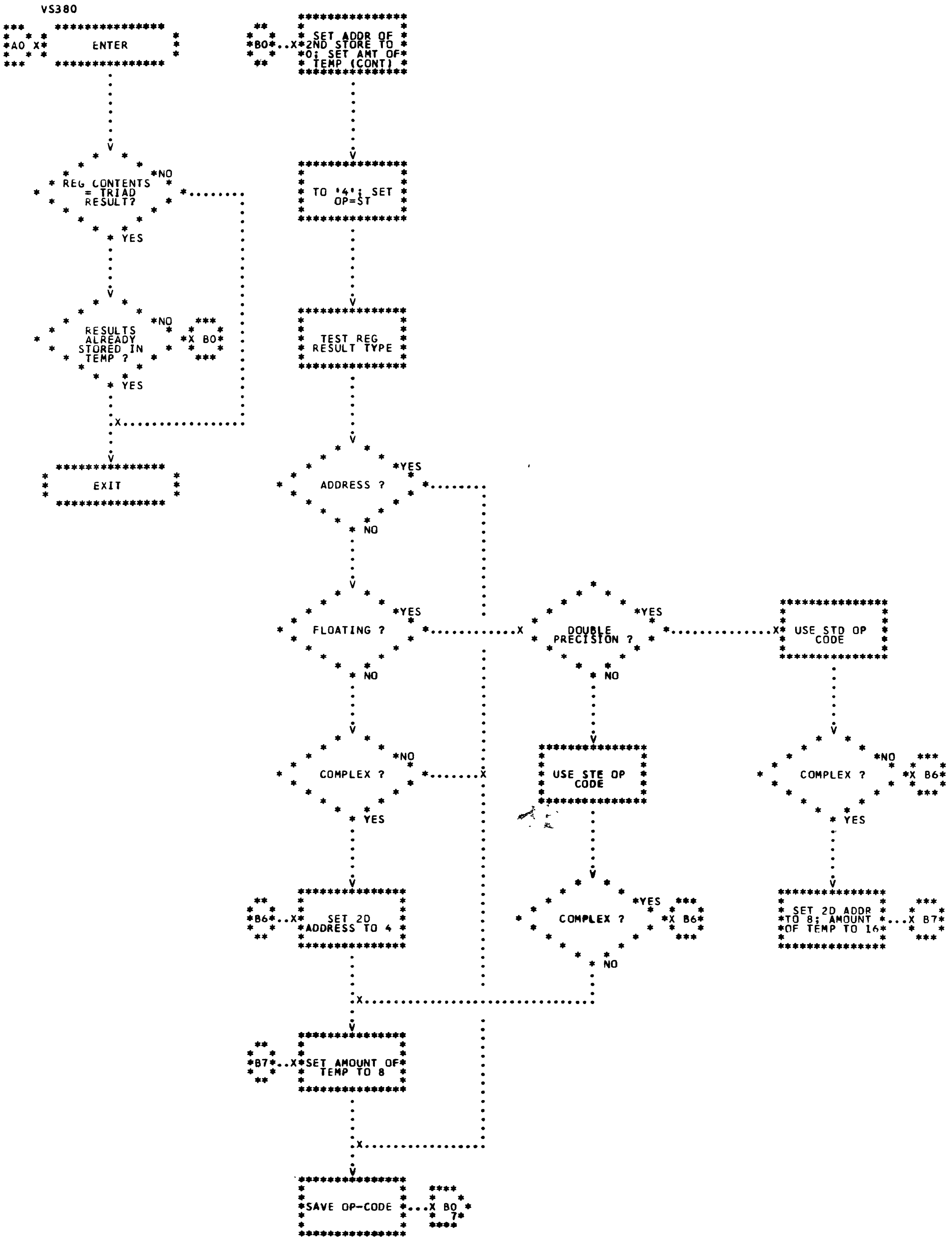


Chart 57. Computational Register Assignment (Page 6 of 8)

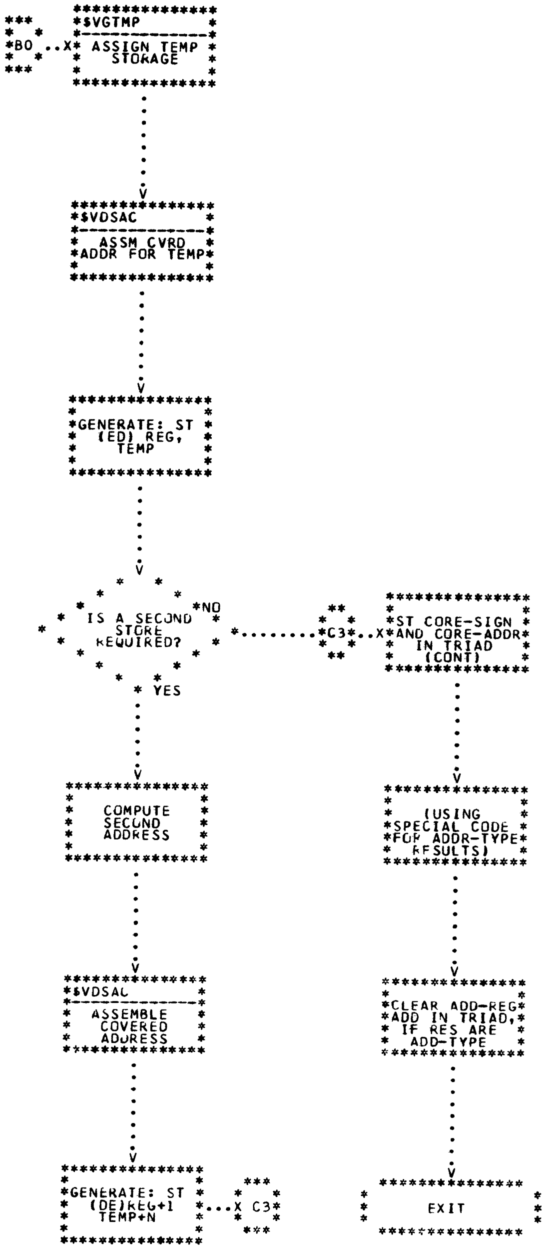
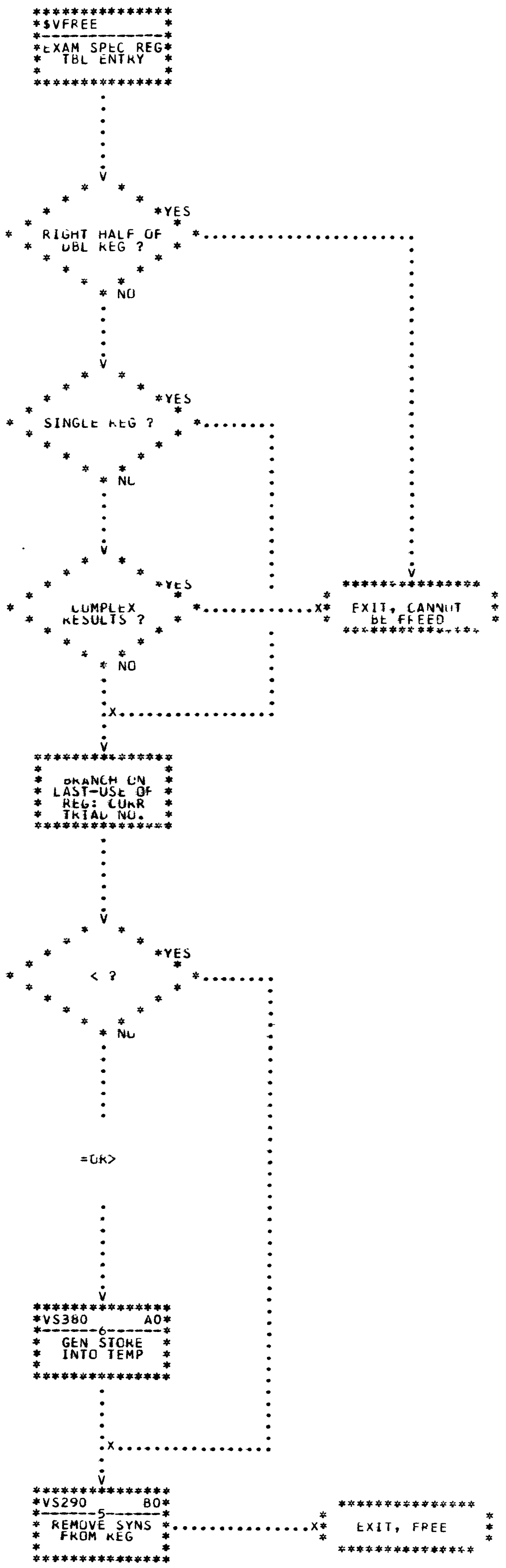
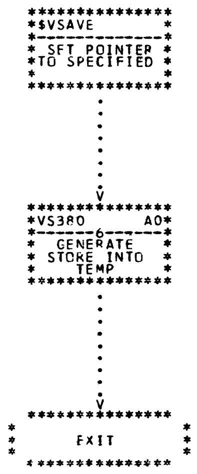


Chart 57. Computational Register Assignment (Page 7 of 8)





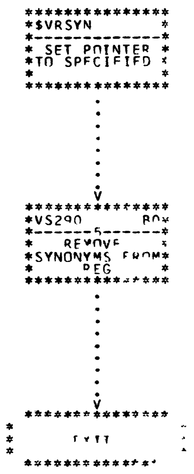


Chart 57. Computational Register Assignment (Page 8 of 8)

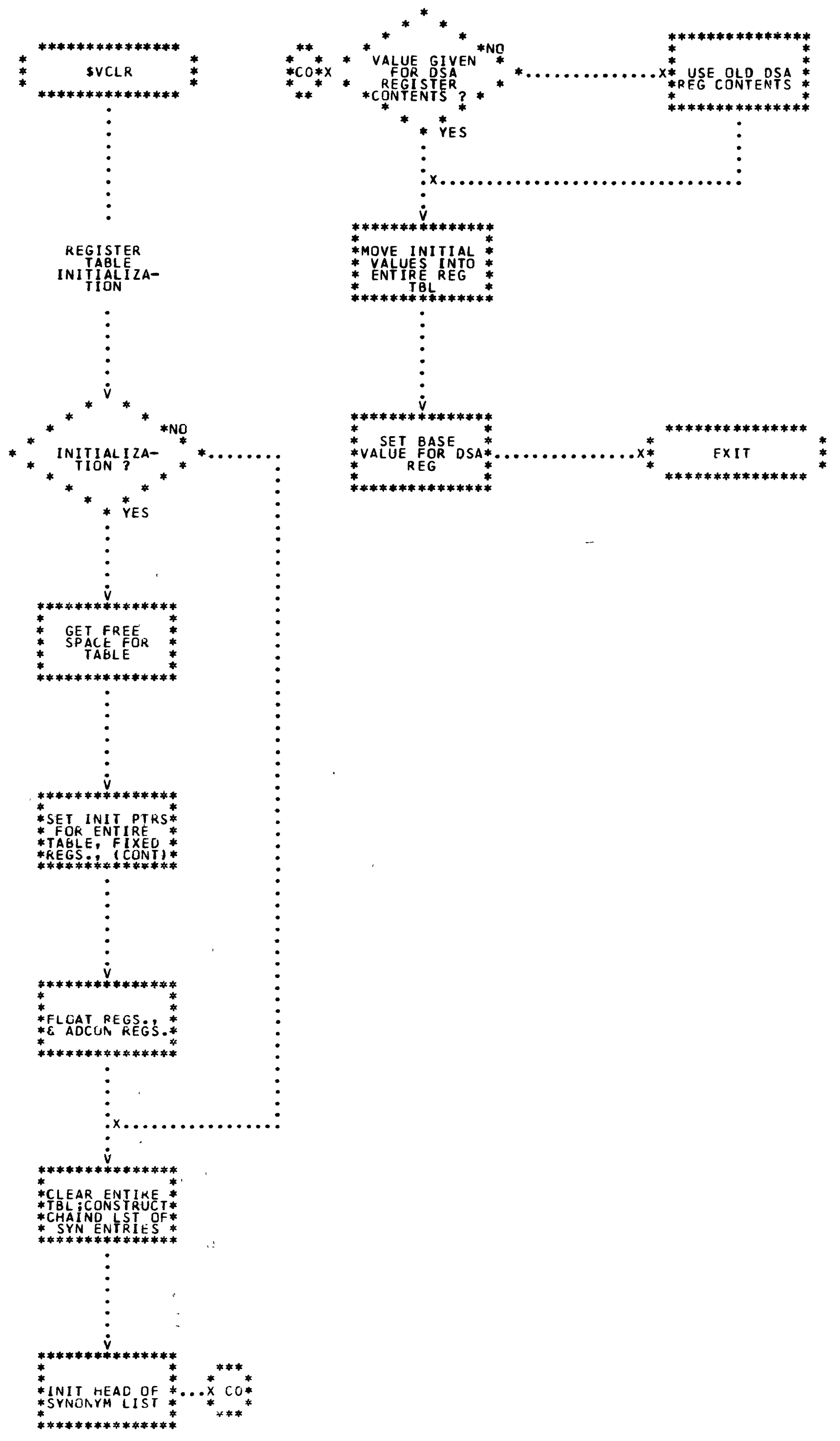


Chart 58. Register Table Initializer

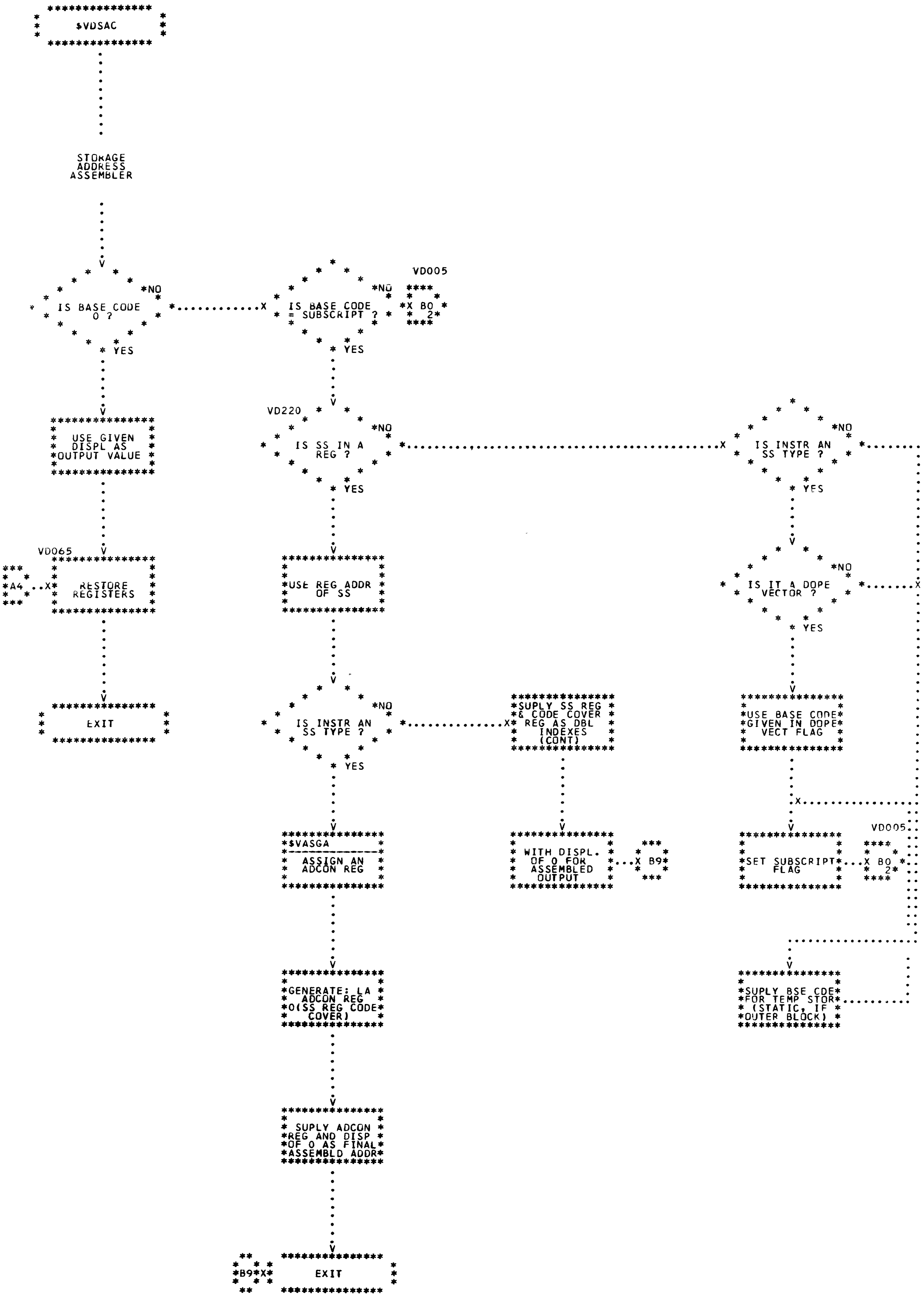


Chart 59. Storage Address Assembler (Page 1 of 5)

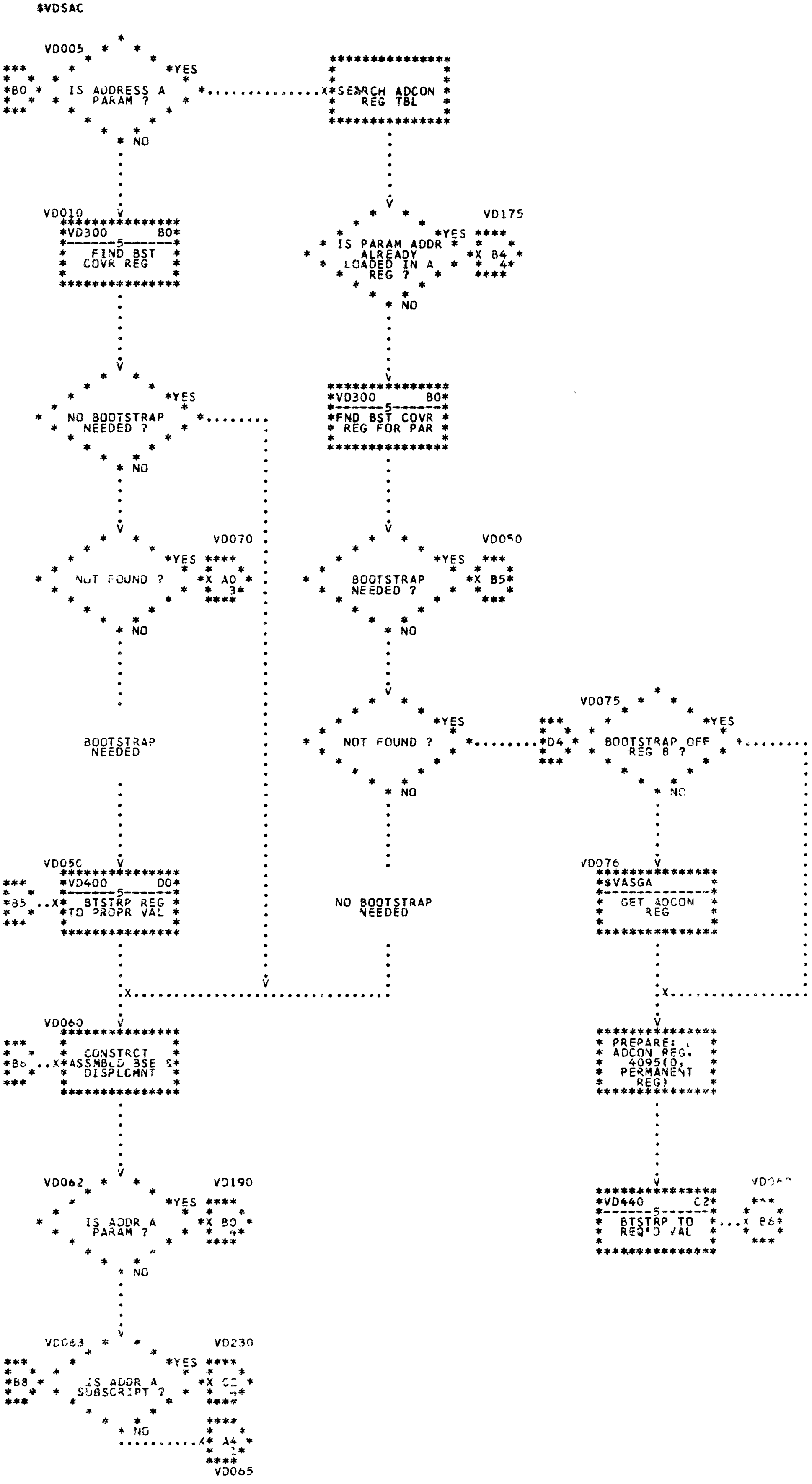


Chart 59. Storage Address Assembler (Page 2 of 5)

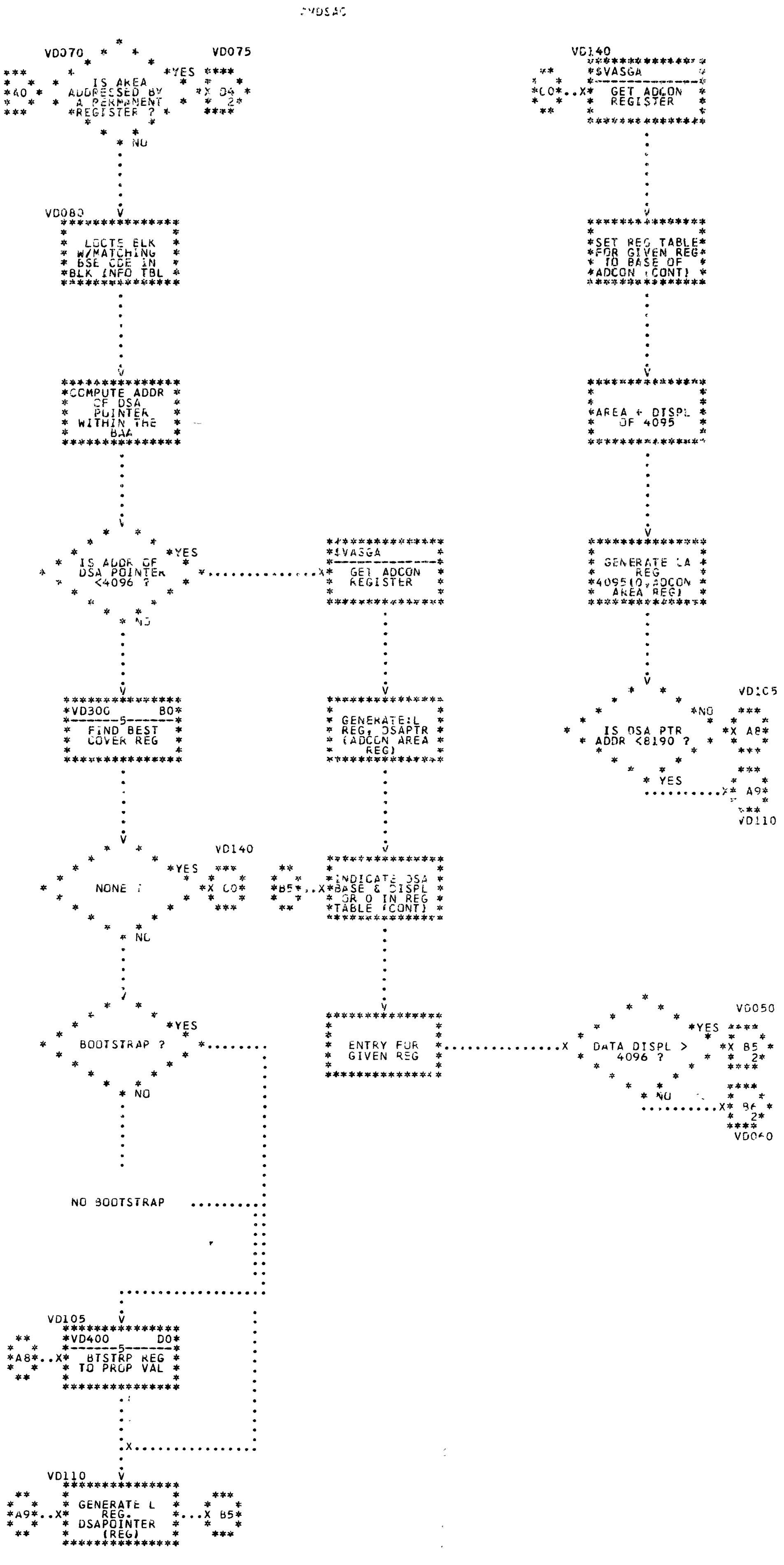


Chart 59. Storage Address Assembler (Page 3 of 5)

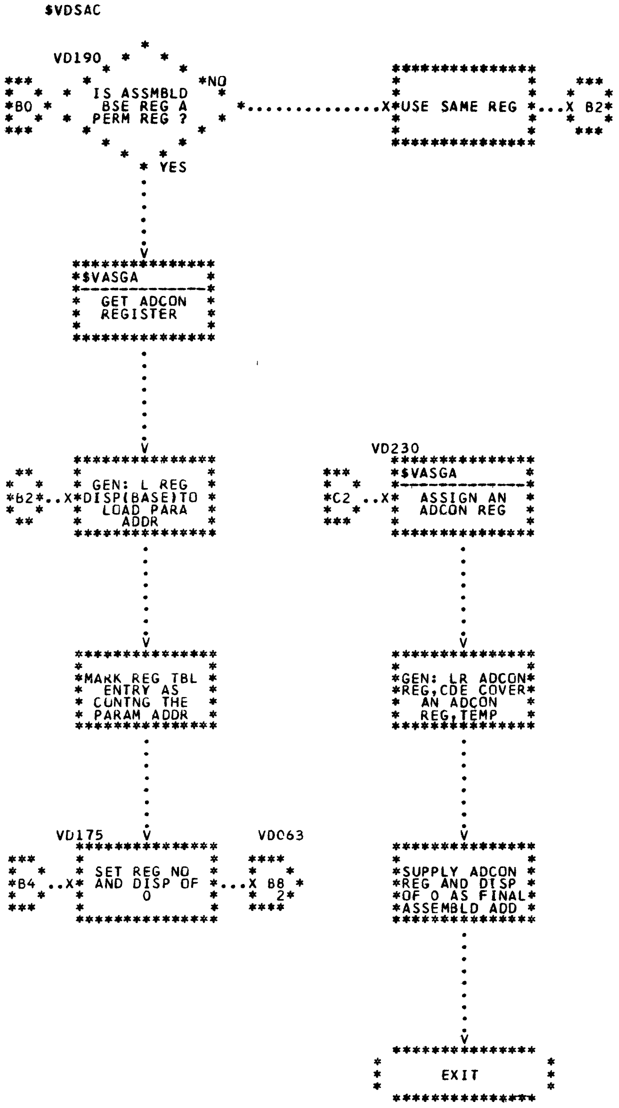


Chart 59. Storage Address Assembler (Page 4 of 5)

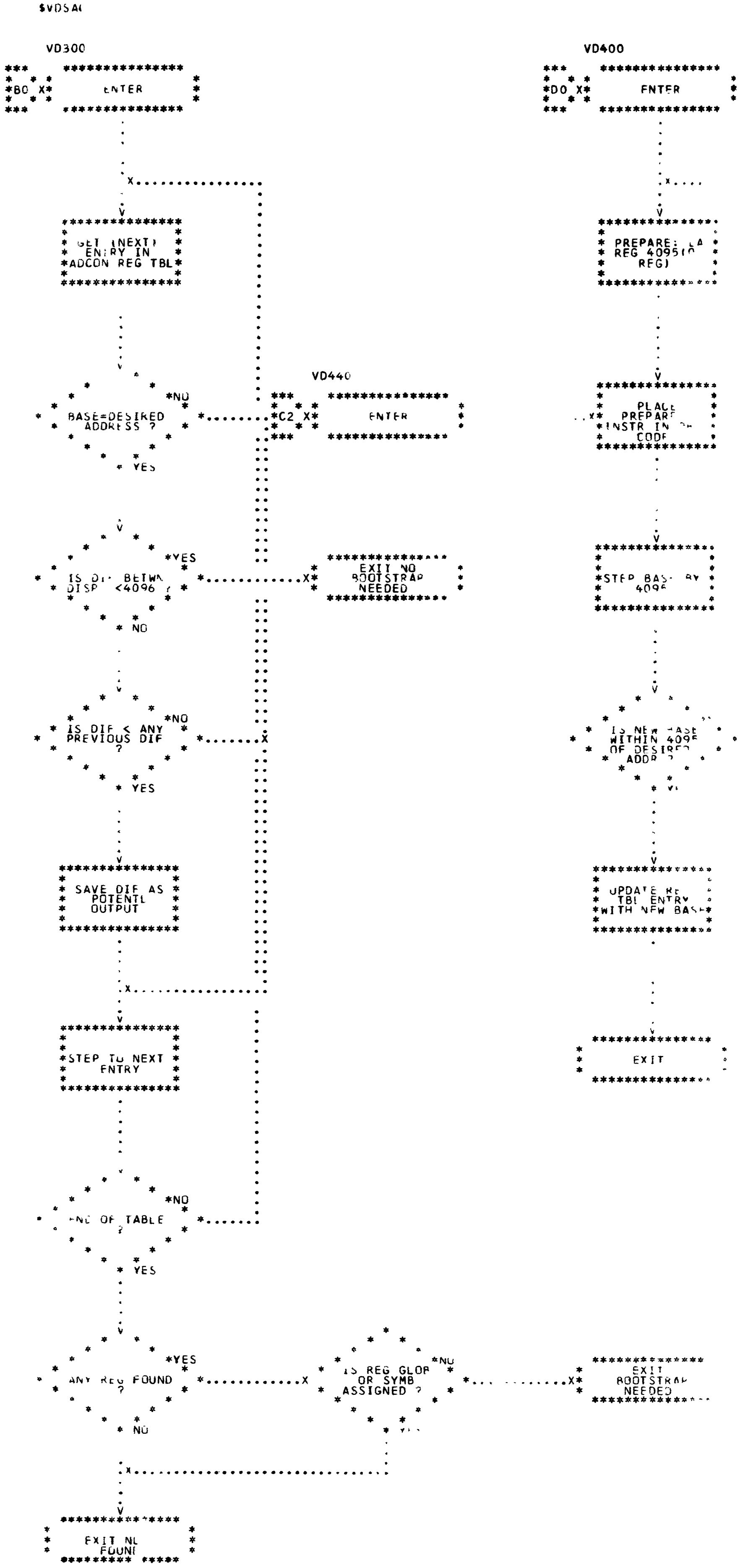


Chart 59. Storage Address Assembler (Page 5 of 5)

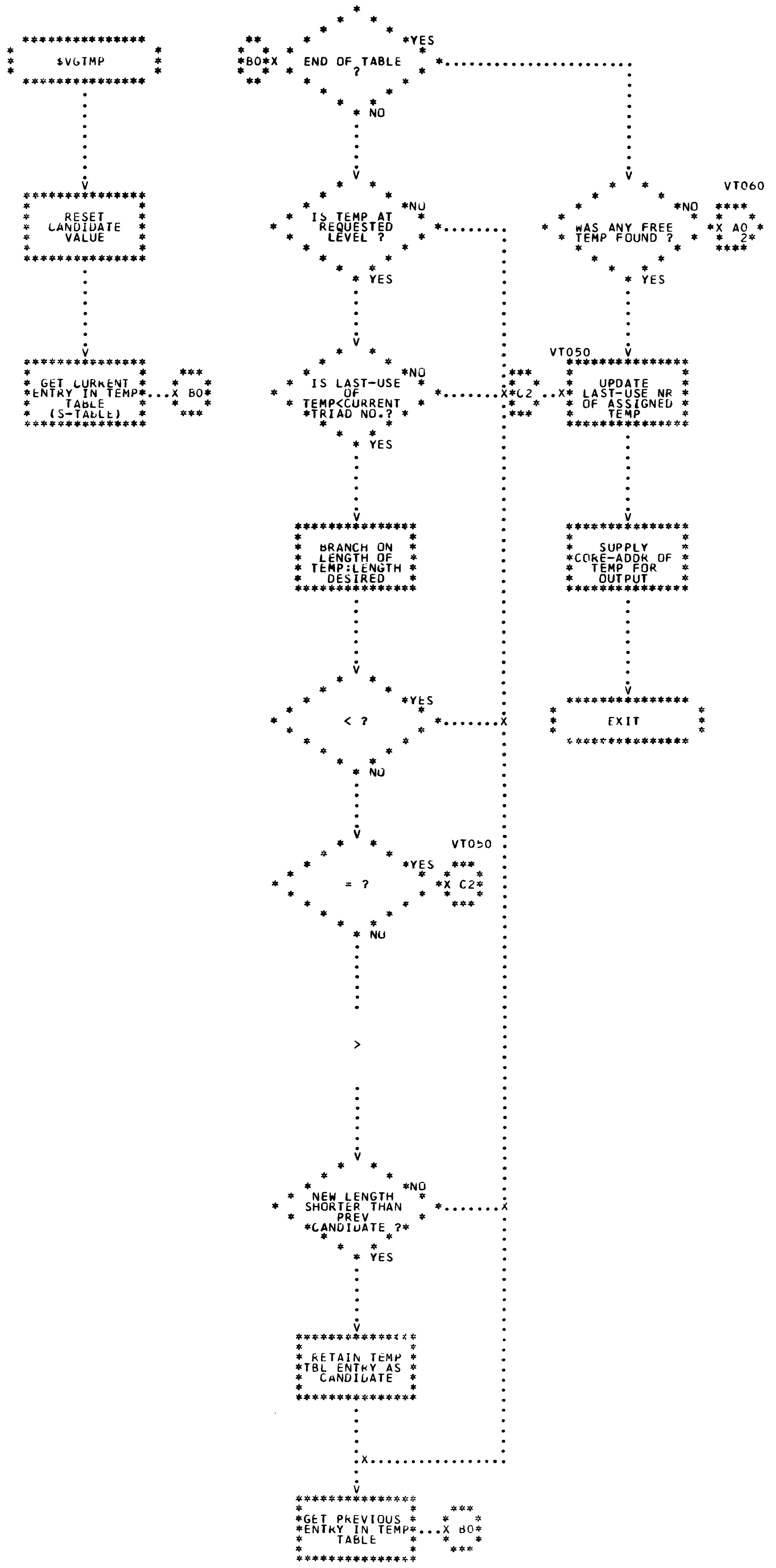


Chart 60. Temporary Storage Management (Page 1 of 2)

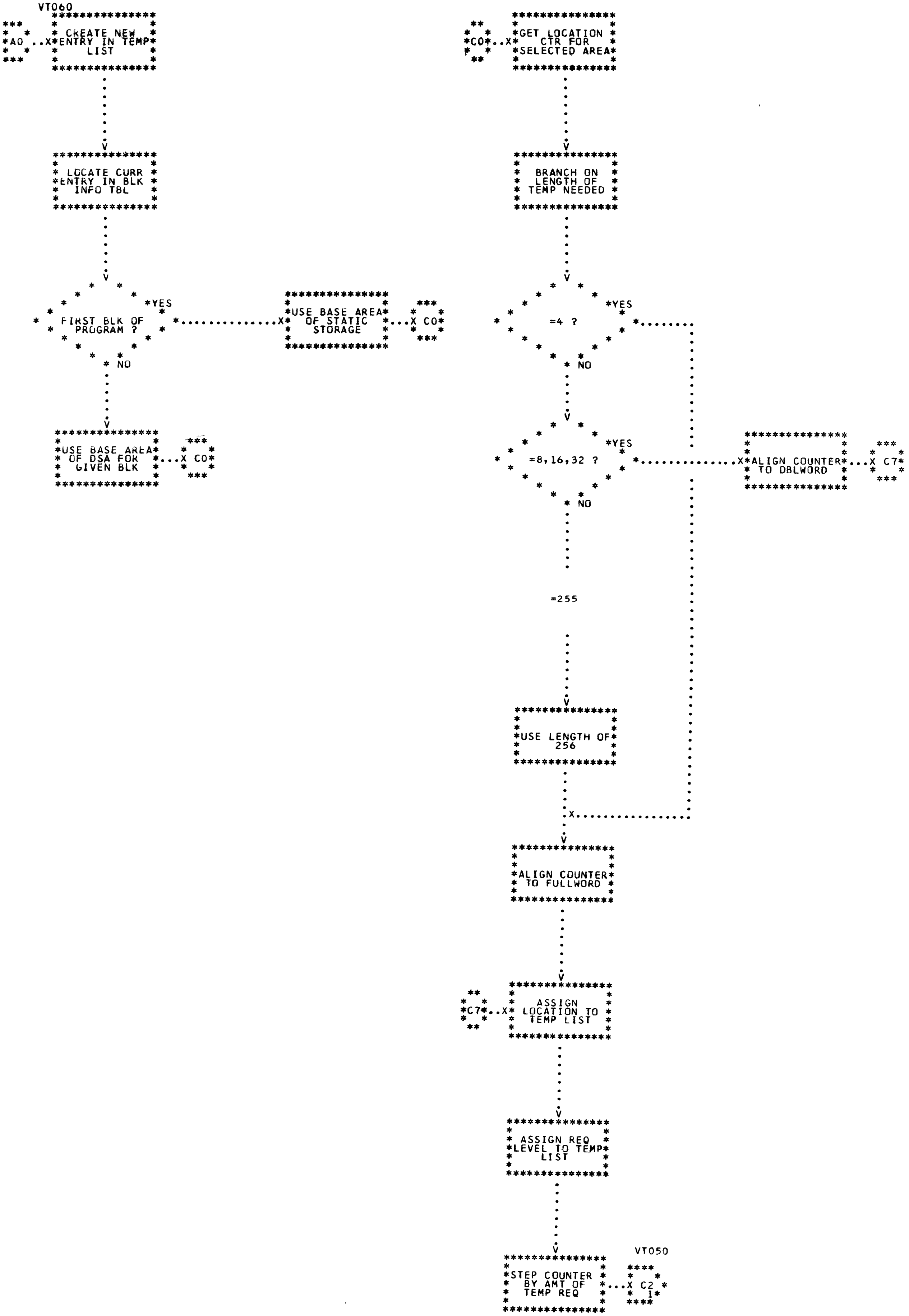


Chart 60. Temporary Storage Management (Page 2 of 2)

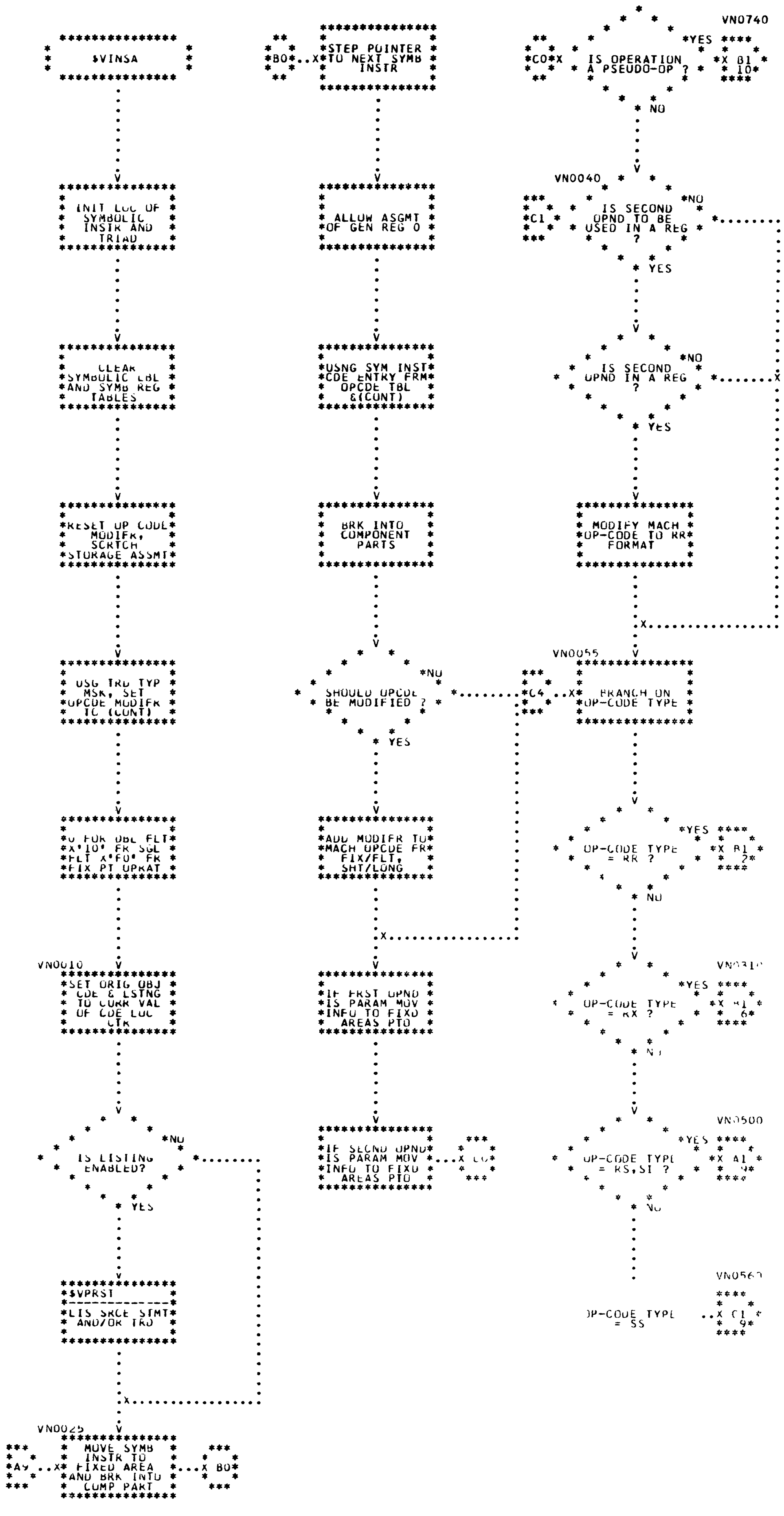
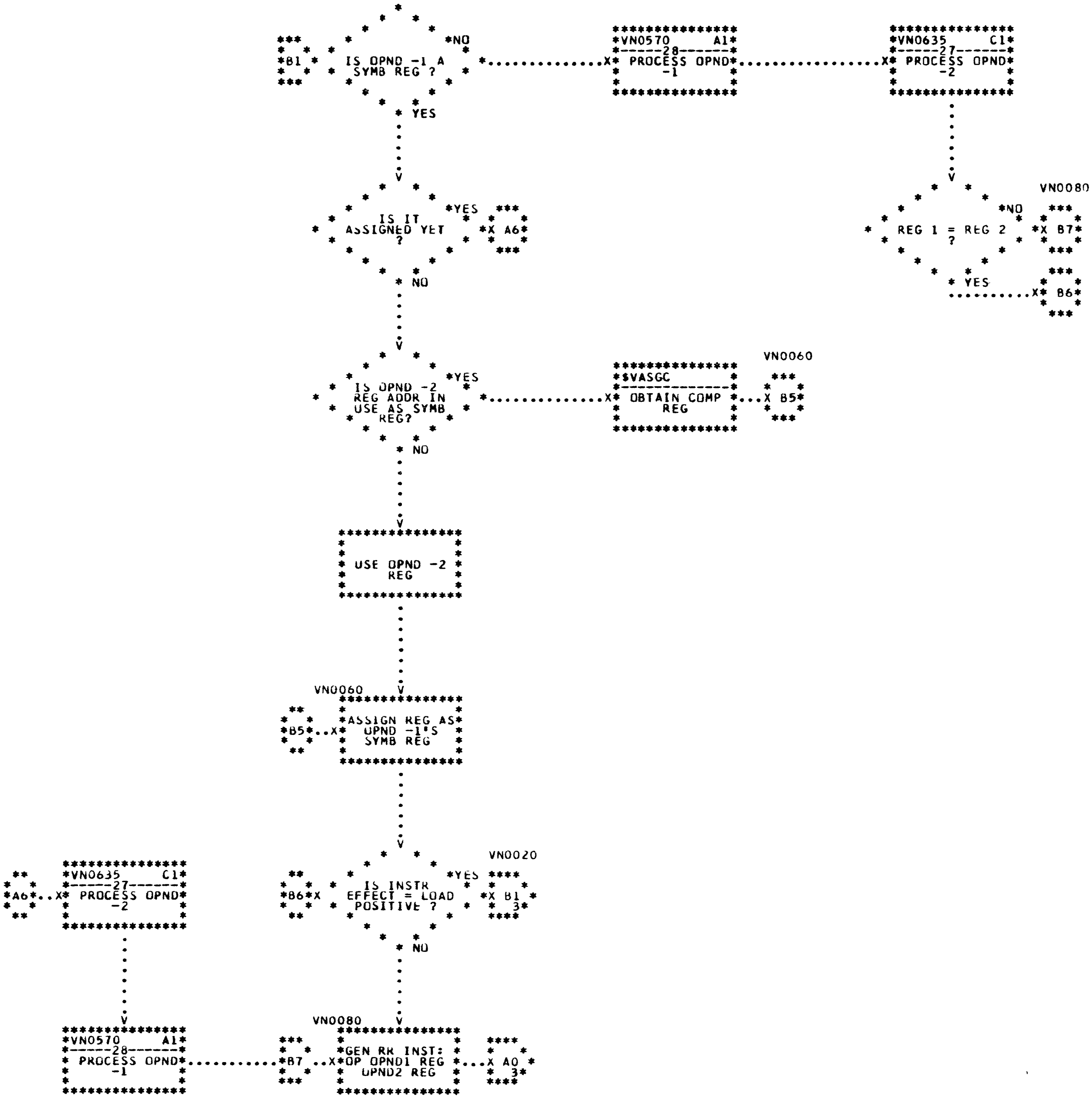


Chart 61. Instruction Assembler (Page 1 of 32)

RR INSTRS


B1
IS OPND -1 A SYMB REG ?
NO
YES
VN0570 A1
28
PROCESS OPND -1
VN0635 C1
27
PROCESS OPND -2
IS IT ASSIGNED YET ?
YES
A6
NO
VN0080
REG 1 = REG 2 ?
NO
B7
YES
B6
IS OPND -2 REG ADDR IN USE AS SYMB REG?
YES
NO
VN0060
$VASGC
OBTAIN COMP REG
B5
USE OPND -2 REG
VN0060
B5
ASSIGN REG AS OPND -1'S SYMB REG
VN0020
B6
IS INSTR EFFECT = LOAD POSITIVE ?
YES
B1 3
NO
A6
VN0635 C1
27
PROCESS OPND -2
VN0570 A1
28
PROCESS OPND -1
VN0080
B7
GEN RR INST: OP OPND1 REG OPND2 REG
A0 3

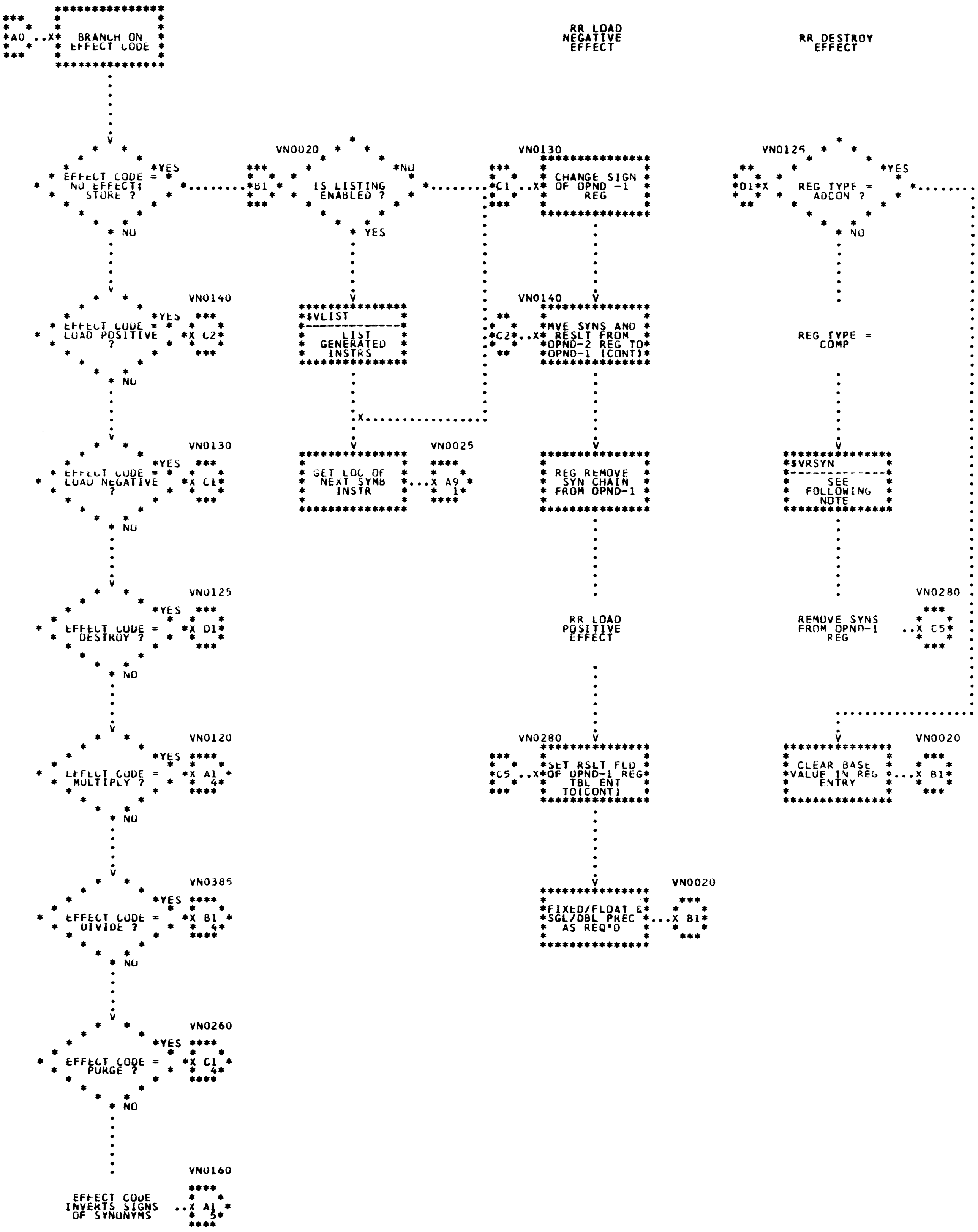


Chart 61. Instruction Assembler (Page 3 of 32)

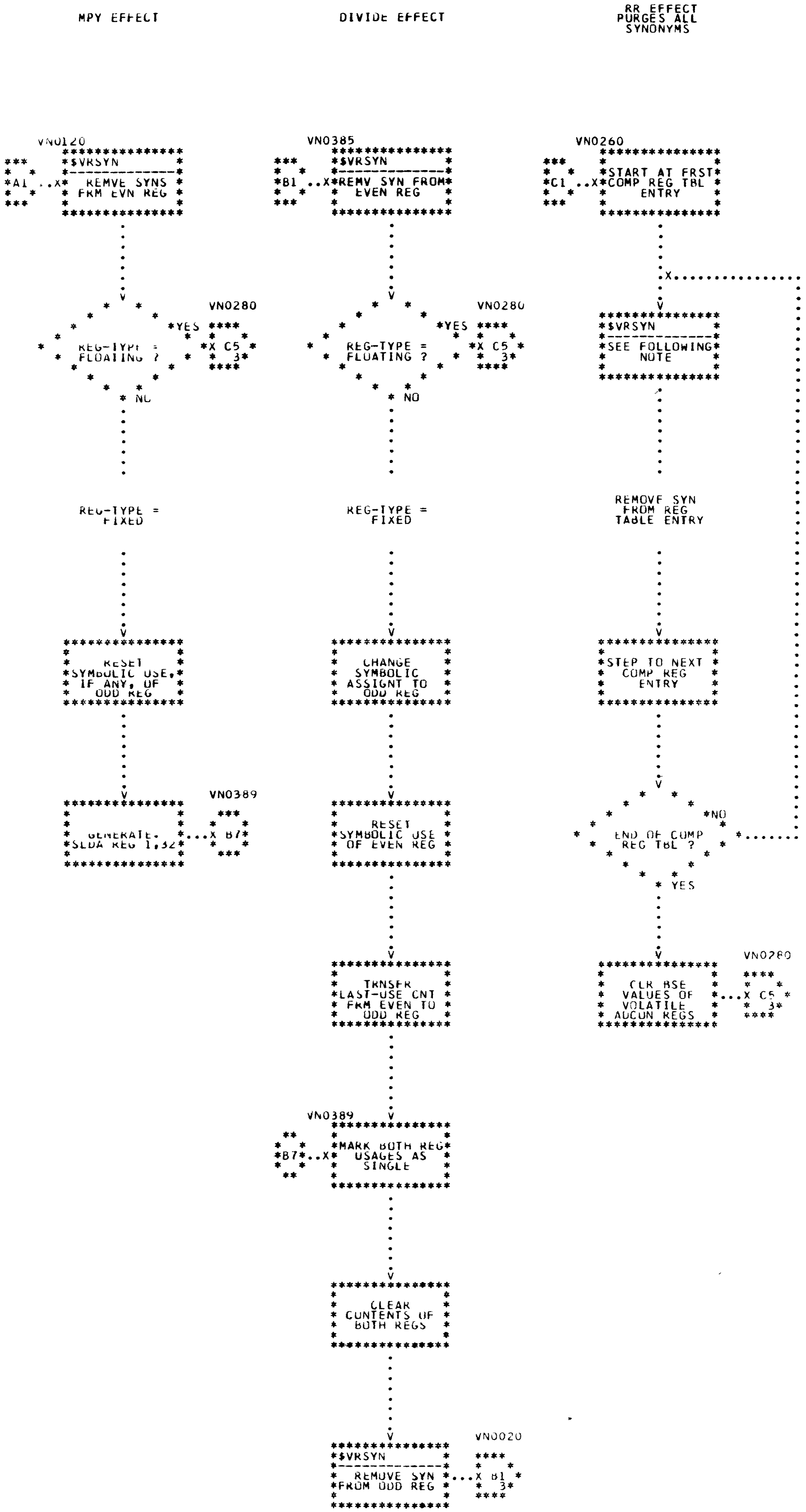


Chart 61. Instruction Assembler (Page 4 of 32)

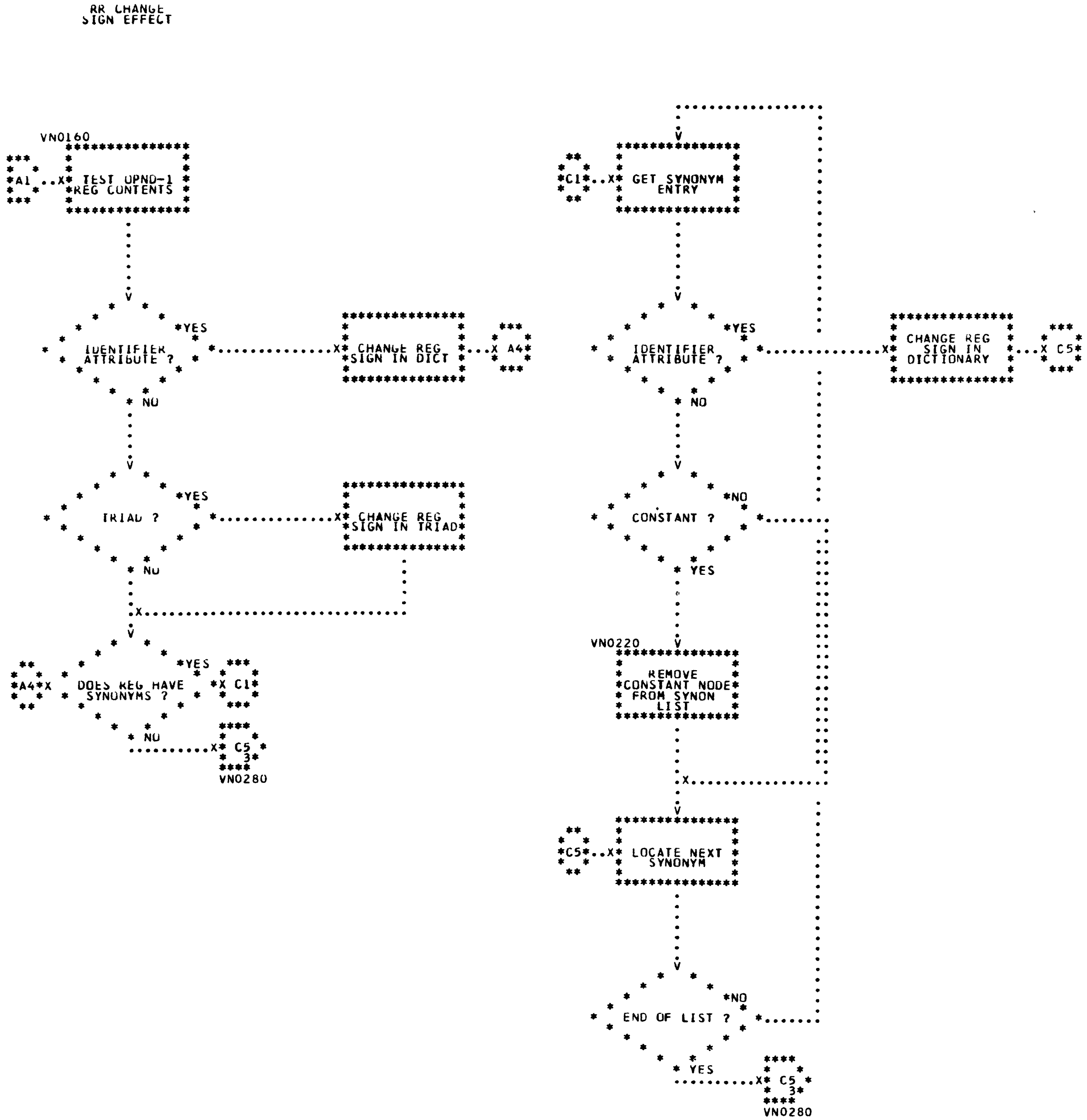


Chart 61. Instruction Assembler (Page 5 of 32)

RX INSTRS

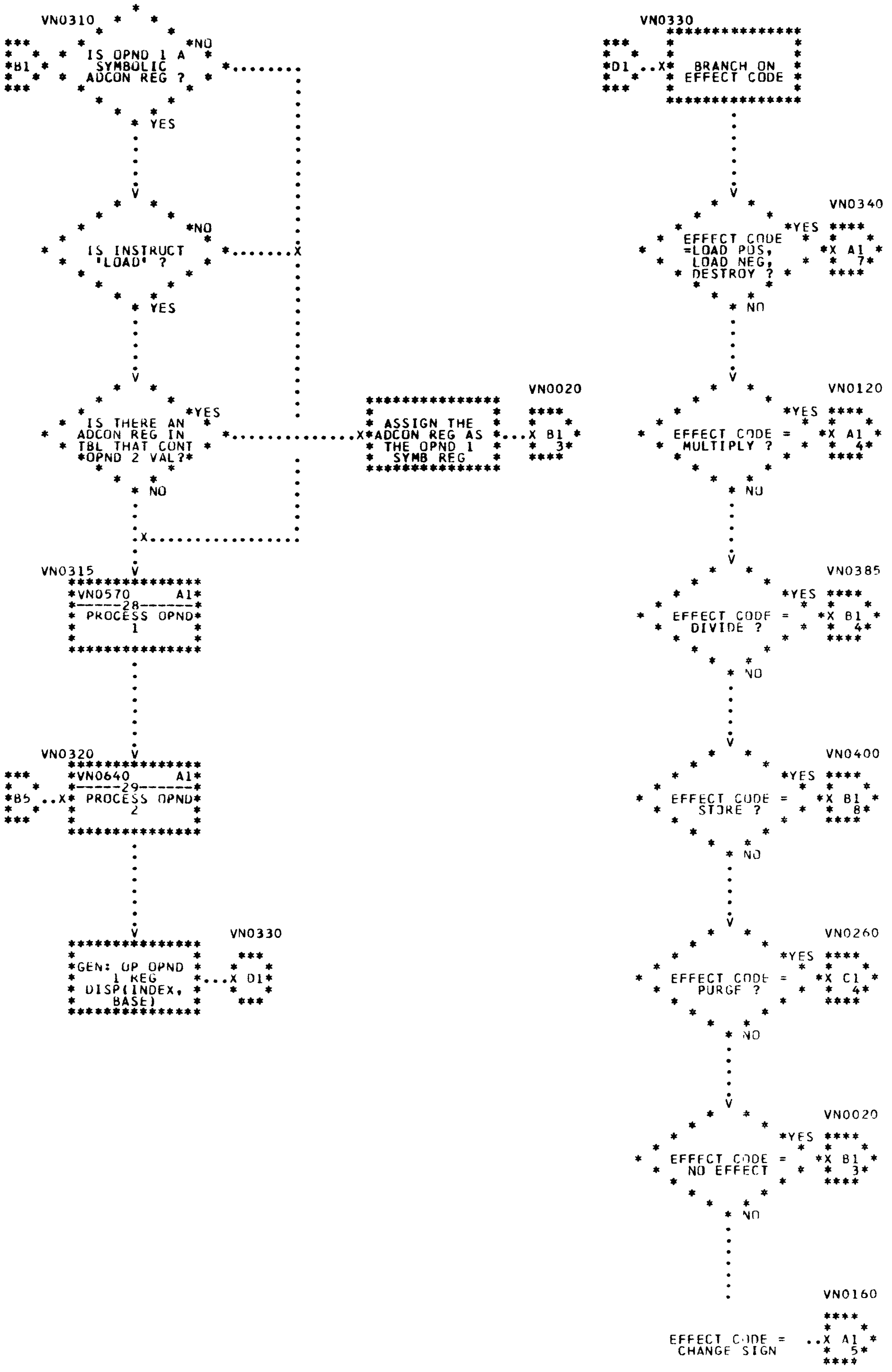
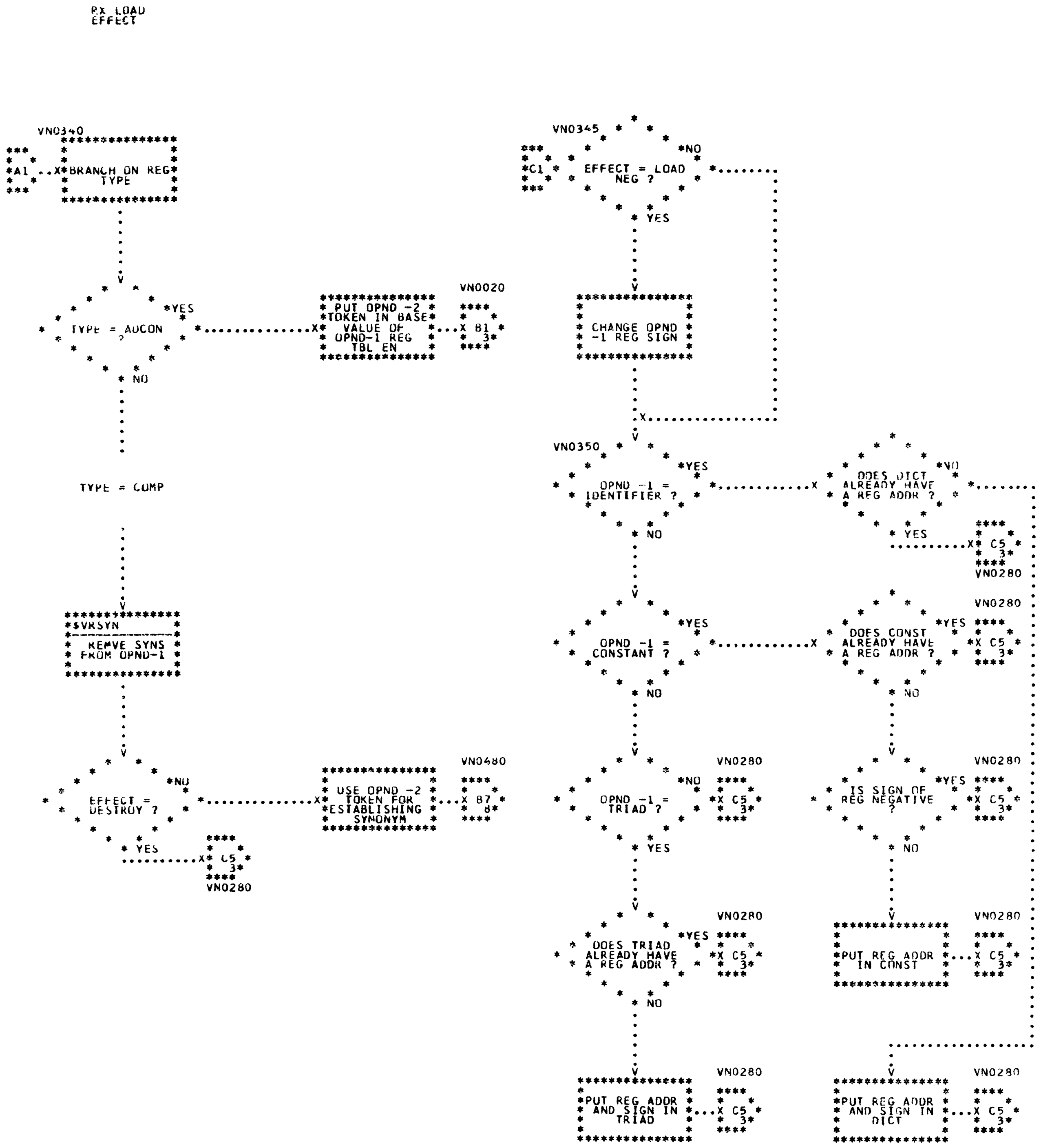


Chart 61. Instruction Assembler (Page 6 of 32)


RX LOAD EFFECT
VN0340
A1
BRANCH ON REG TYPE
TYPE = ADCON ?
YES
NO
PUT OPND -2 TOKEN IN BASE VALUE OF OPND-1 REG TBL EN
VN0020
B1 3
TYPE = COMP
$VRSYN
REMVE SYNS FROM OPND-1
EFFECT = DESTROY ?
NO
YES
C5 3
VN0280
USE OPND -2 TOKEN FOR ESTABLISHING SYNONYM
VN0480
B7 8
VN0345
C1
EFFECT = LOAD NEG ?
NO
YES
CHANGE OPND -1 REG SIGN
VN0350
OPND -1 = IDENTIFIER ?
YES
NO
DOES DICT ALREADY HAVE A REG ADDR ?
NO
YES
C5 3
VN0280
OPND -1 = CONSTANT ?
YES
NO
DOES CONST ALREADY HAVE A REG ADDR ?
YES
NO
C5 3
VN0280
OPND -1 = TRIAD ?
NO
YES
C5 3
VN0280
IS SIGN OF REG NEGATIVE ?
YES
NO
C5 3
VN0280
DOES TRIAD ALREADY HAVE A REG ADDR ?
YES
NO
C5 3
VN0280
PUT REG ADDR IN CONST
C5 3
VN0280
PUT REG ADDR AND SIGN IN TRIAD
C5 3
VN0280
PUT REG ADDR AND SIGN IN DICT
C5 3

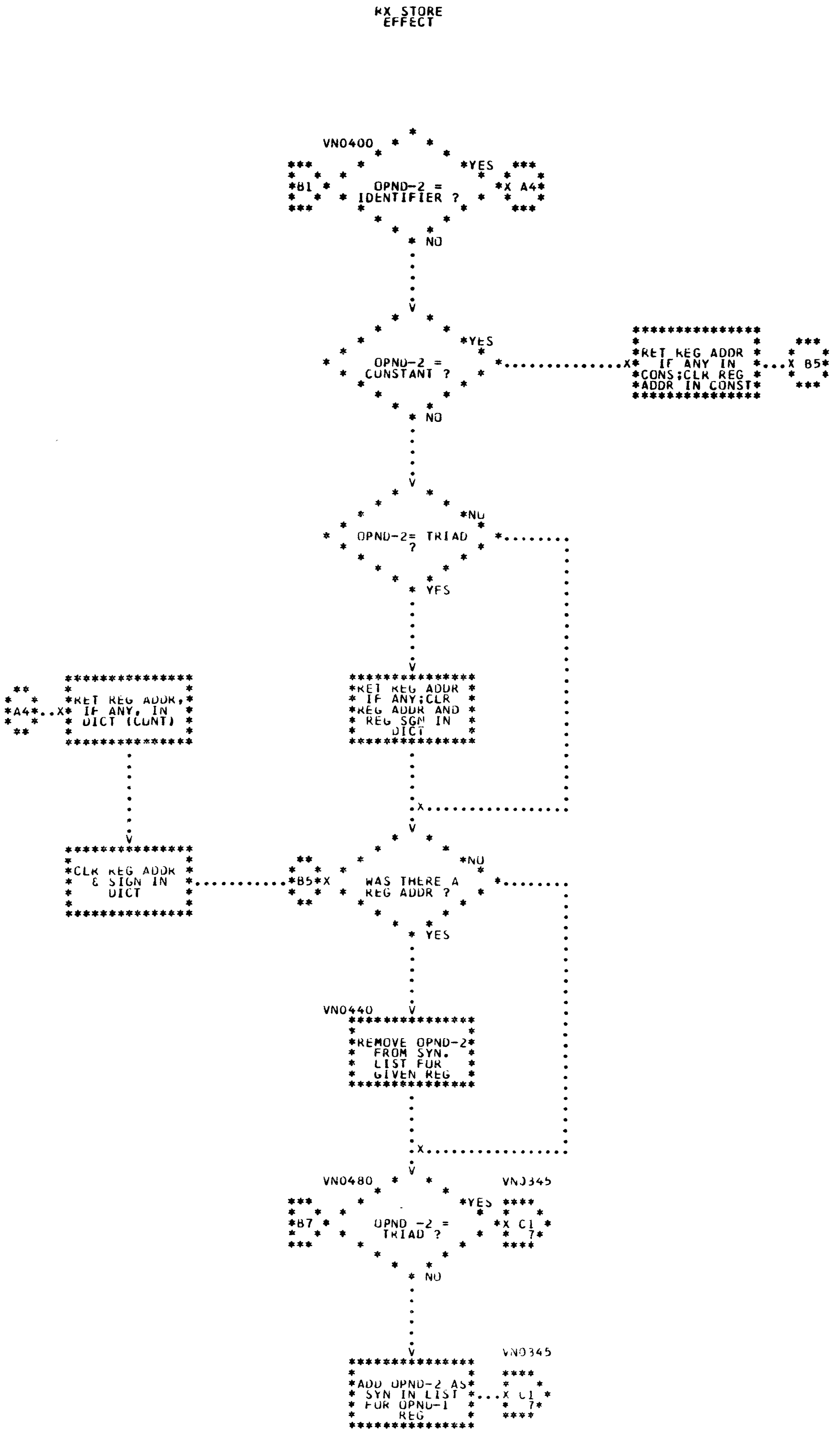


Chart 61. Instruction Assembler (Page 8 of 32)

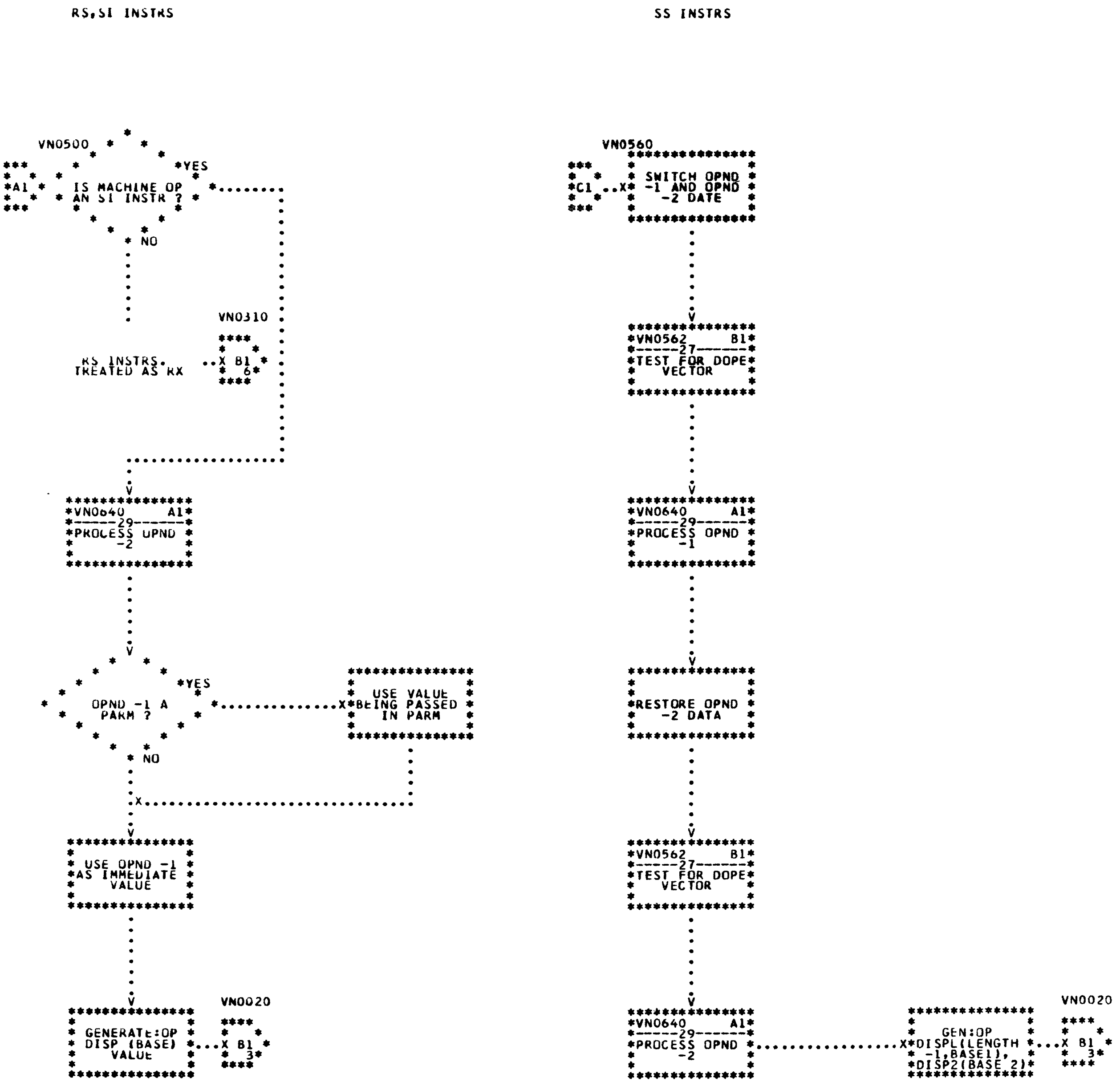


Chart 61. Instruction Assembler (Page 9 of 32)

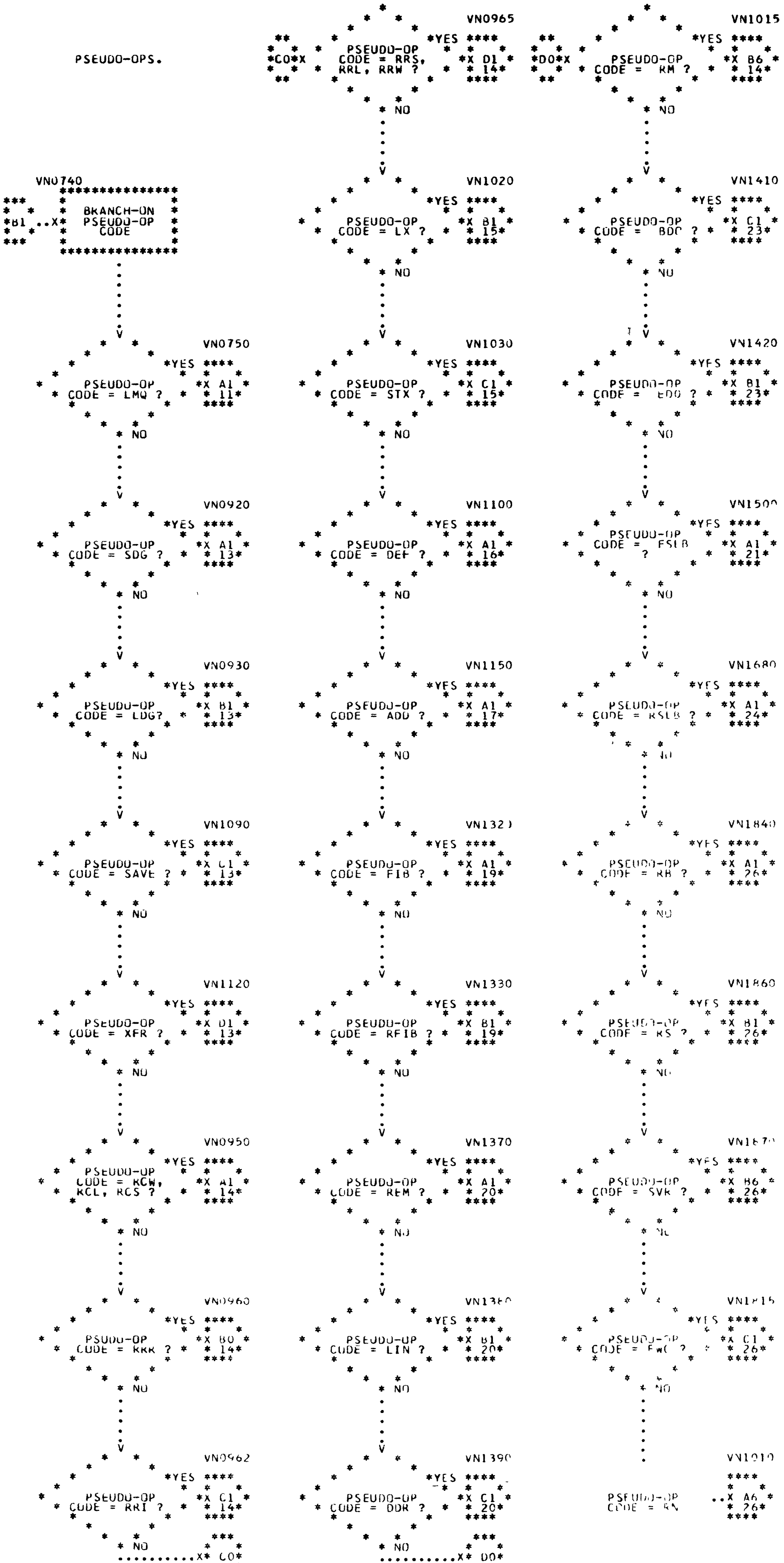


Chart 61. Instruction Assembler (Page 10 of 32)

LMQ LDIV

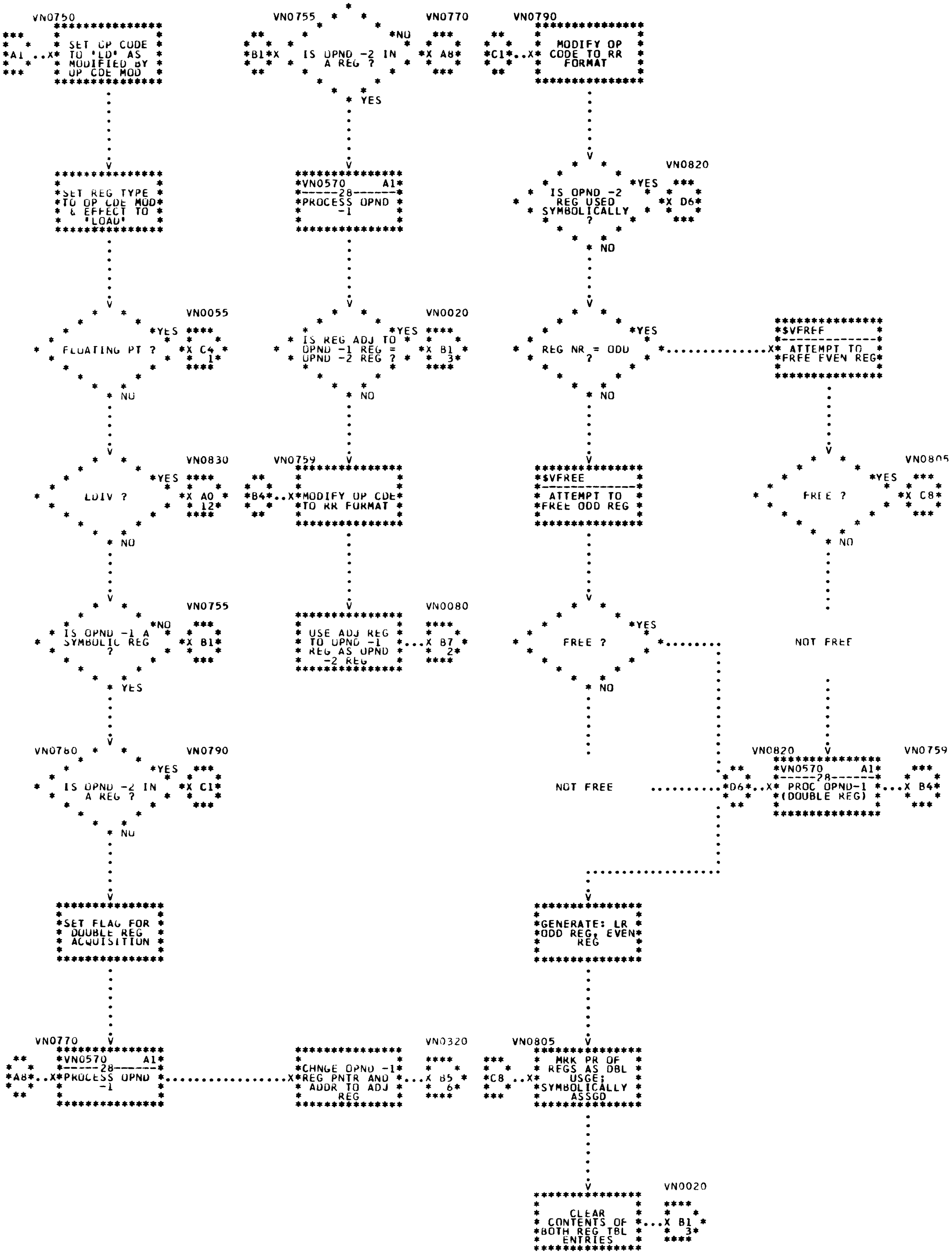


Chart 61. Instruction Assembler (Page 11 of 32)

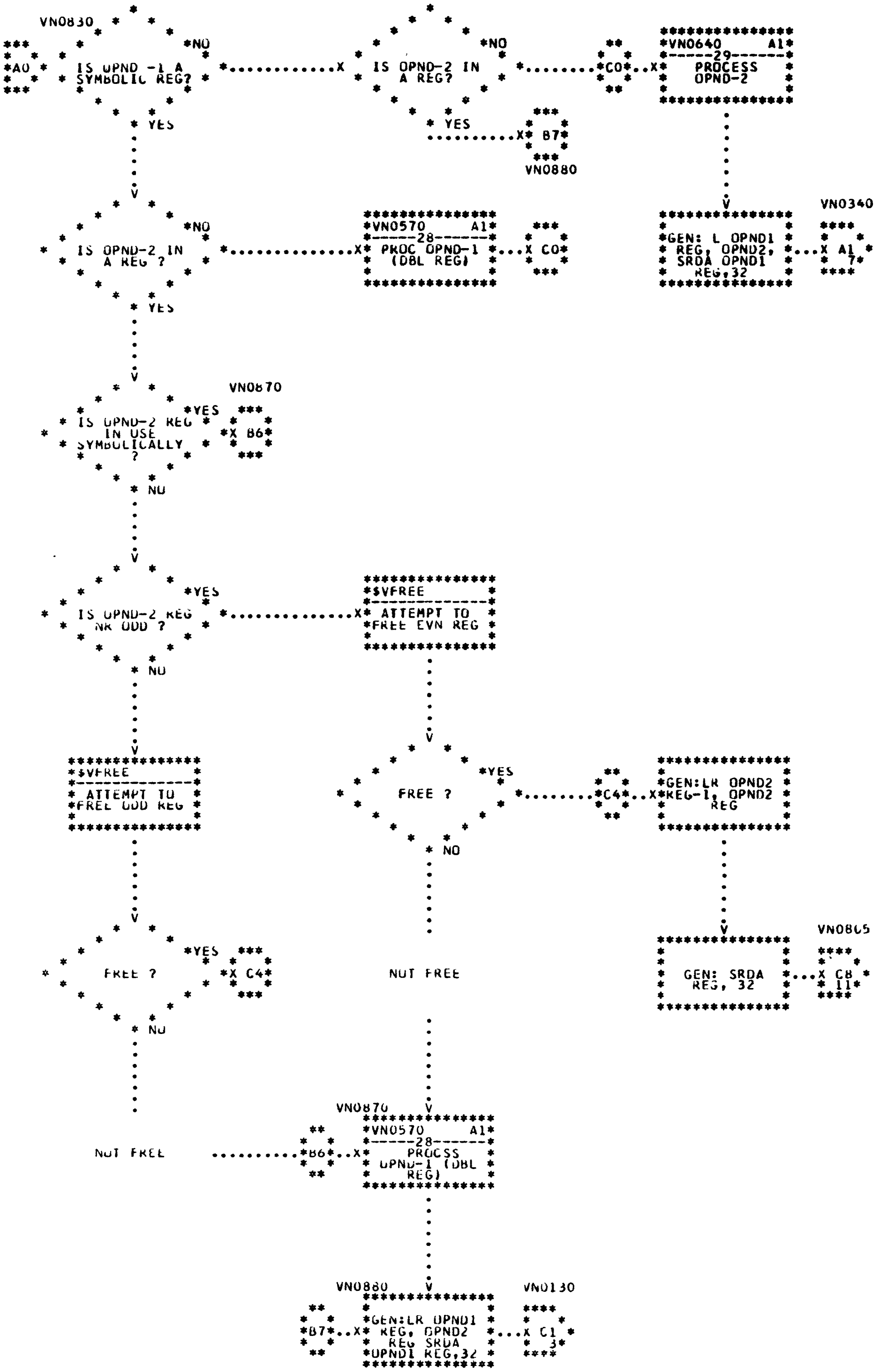


Chart 61. Instruction Assembler (Page 12 of 32)

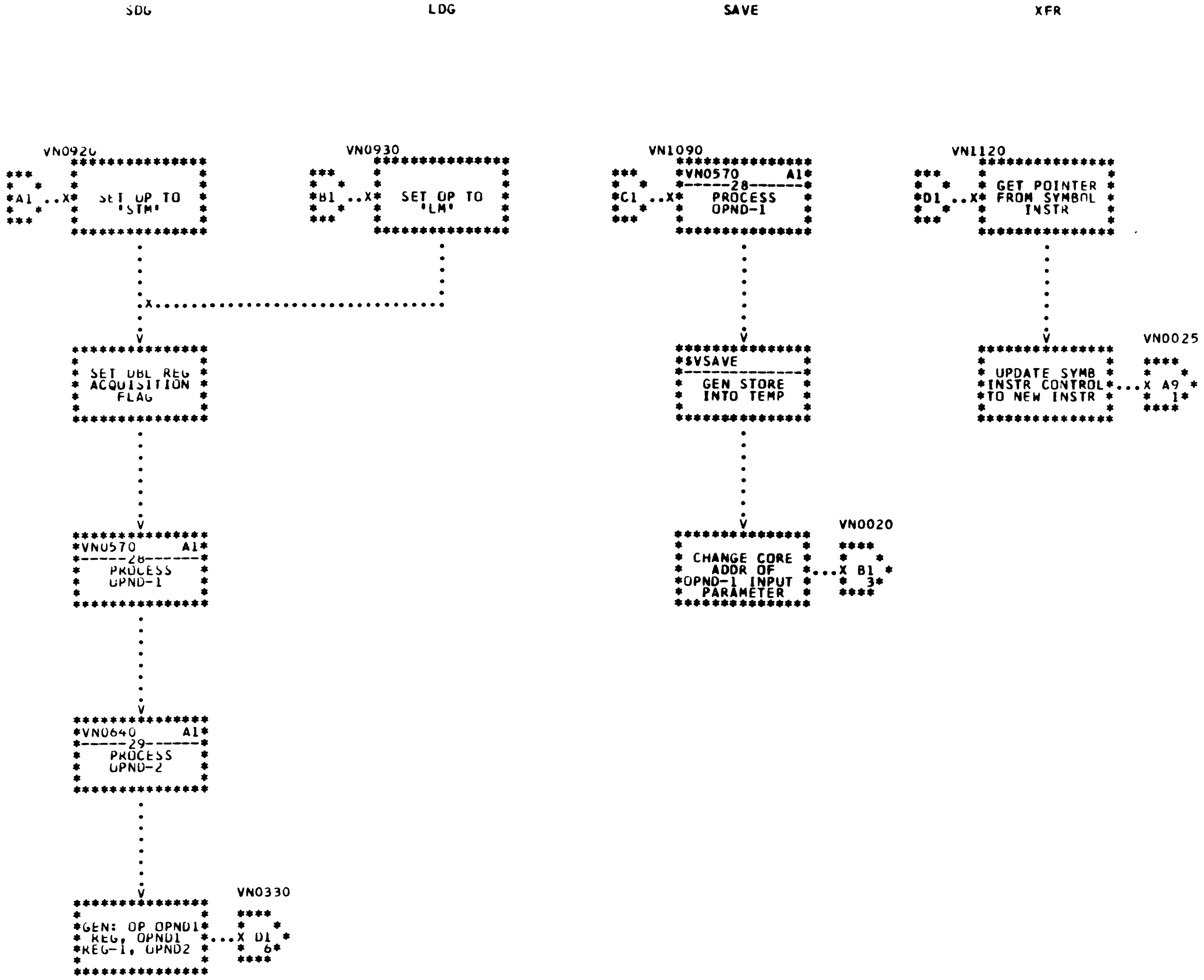
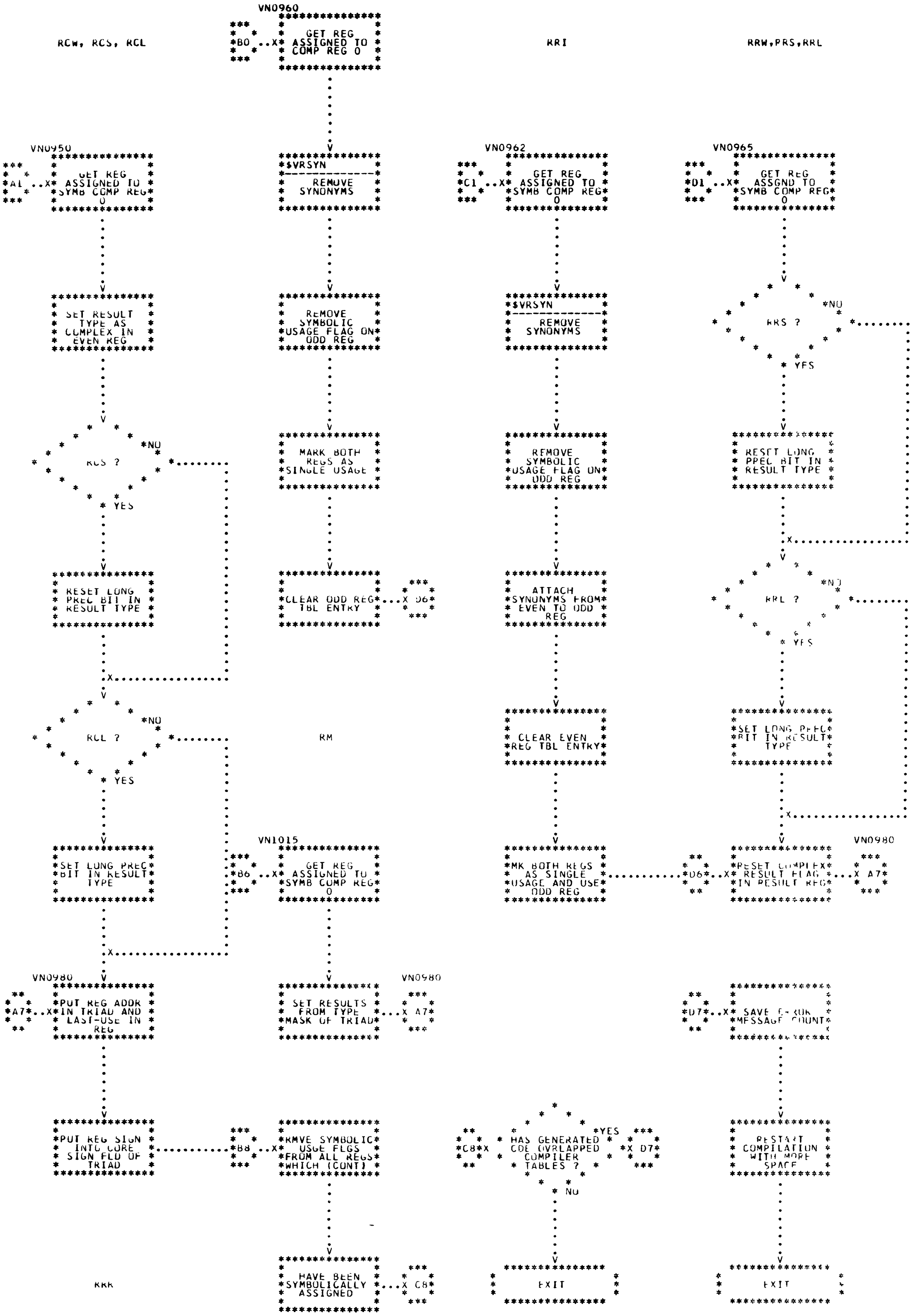


Chart 61. Instruction Assembler (Page 13 of 32)


RCW, RCS, RCL
RRI
RRW,PRS,RRL
VN0960
B0
GET REG ASSIGNED TO COMP REG 0
VN0950
A1
GET REG ASSIGNED TO SYMB COMP REG 0
$VRSYN
REMOVE SYNONYMS
VN0962
C1
GET REG ASSIGNED TO SYMB COMP REG 0
VN0965
D1
GET REG ASSGND TO SYMB COMP REG 0
SET RESULT TYPE AS COMPLEX IN EVEN REG
REMOVE SYMBOLIC USAGE FLAG ON ODD REG
$VRSYN
REMOVE SYNONYMS
RRS ?
NO
YES
RCS ?
NO
YES
MARK BOTH REGS AS SINGLE USAGE
REMOVE SYMBOLIC USAGE FLAG ON ODD REG
RESET LONG PREC BIT IN RESULT TYPE
RESET LONG PREC BIT IN RESULT TYPE
CLEAR ODD REG TBL ENTRY
D6
ATTACH SYNONYMS FROM EVEN TO ODD REG
RRL ?
NO
YES
RCL ?
NO
YES
RM
CLEAR EVEN REG TBL ENTRY
SET LONG PREC BIT IN RESULT TYPE
SET LONG PREC BIT IN RESULT TYPE
VN1015
B6
GET REG ASSIGNED TO SYMB COMP REG 0
MK BOTH REGS AS SINGLE USAGE AND USE ODD REG
D6
RESET COMPLEX RESULT FLAG IN RESULT REG
VN0980
A7
VN0980
A7
PUT REG ADDR IN TRIAD AND LAST-USE IN REG
SET RESULTS FROM TYPE MASK OF TRIAD
VN0980
A7
D7
SAVE ERROR MESSAGE COUNT
PUT REG SIGN INTO CORE SIGN FLD OF TRIAD
B8
RMVE SYMBOLIC USGE FLGS FROM ALL REGS WHICH (CONT)
C8
HAS GENERATED CDE OVRLAPPED COMPILER TABLES ?
YES
D7
NO
RESTART COMPILATION WITH MORE SPACE
RRR
HAVE BEEN SYMBOLICALLY ASSIGNED
C8
EXIT
EXIT

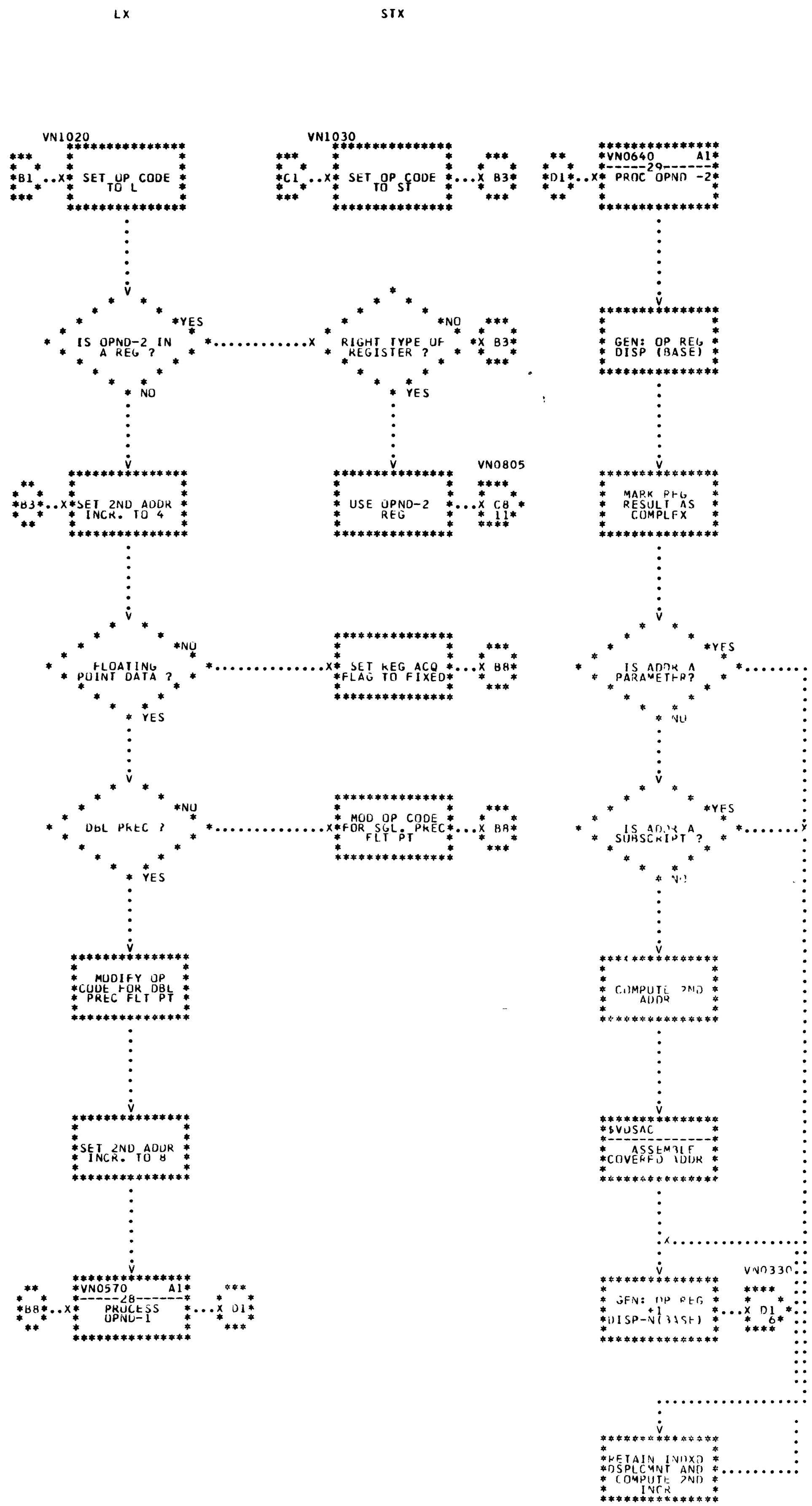


Chart 61. Instruction Assembler (Page 15 of 32)

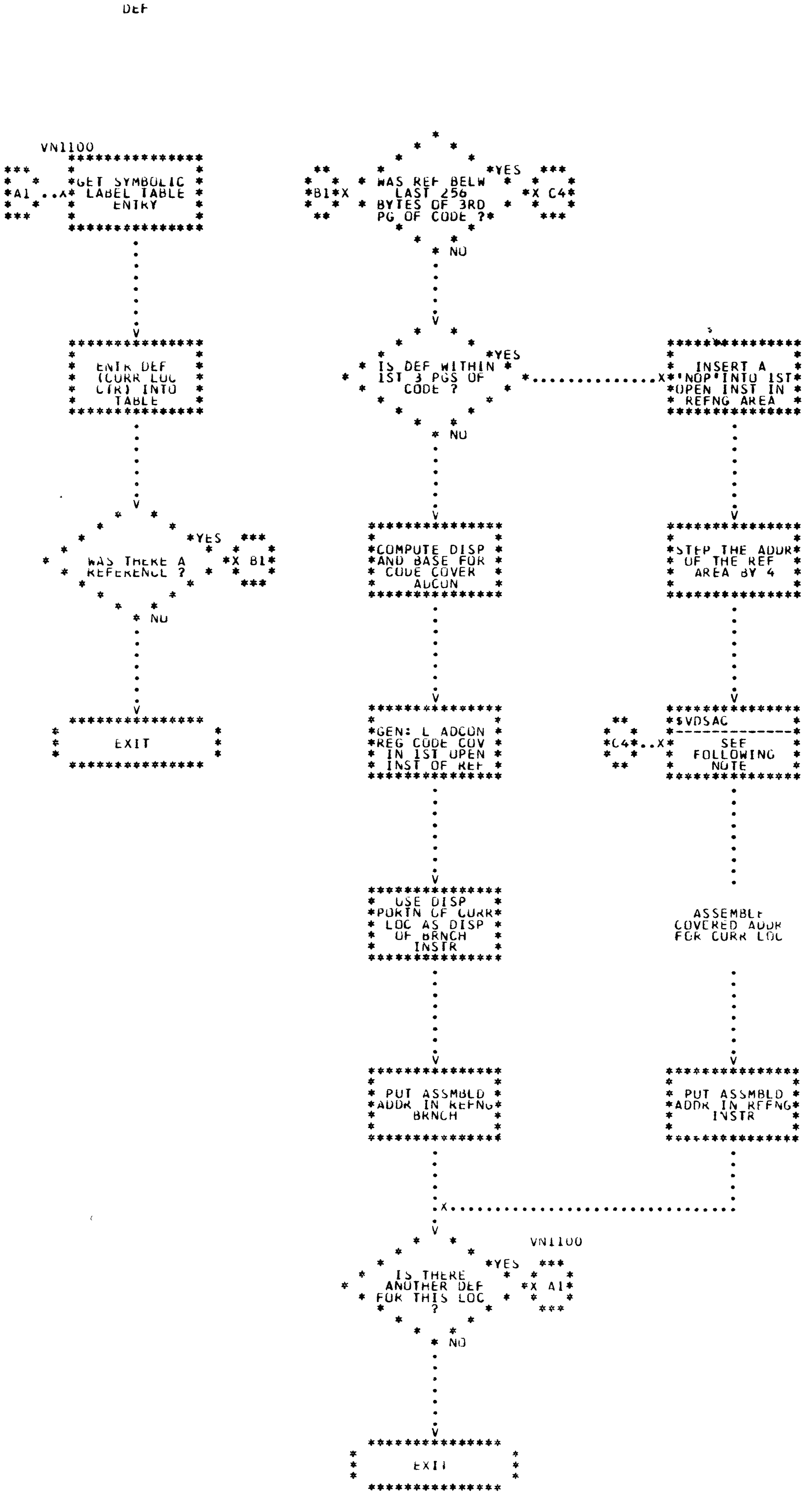


Chart 61. Instruction Assembler (Page 16 of 32)

ADD

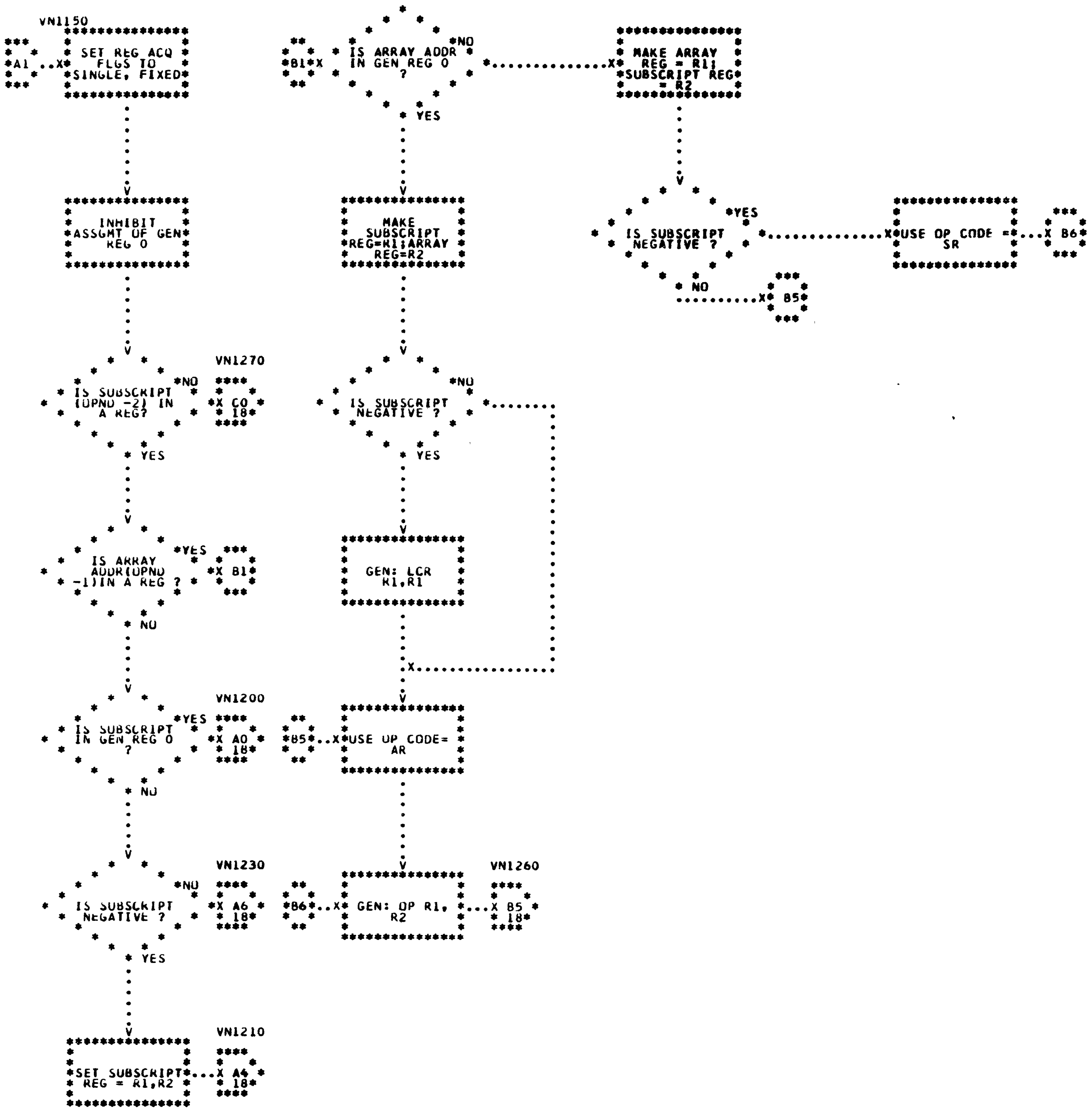


Chart 61. Instruction Assembler (Page 17 of 32)

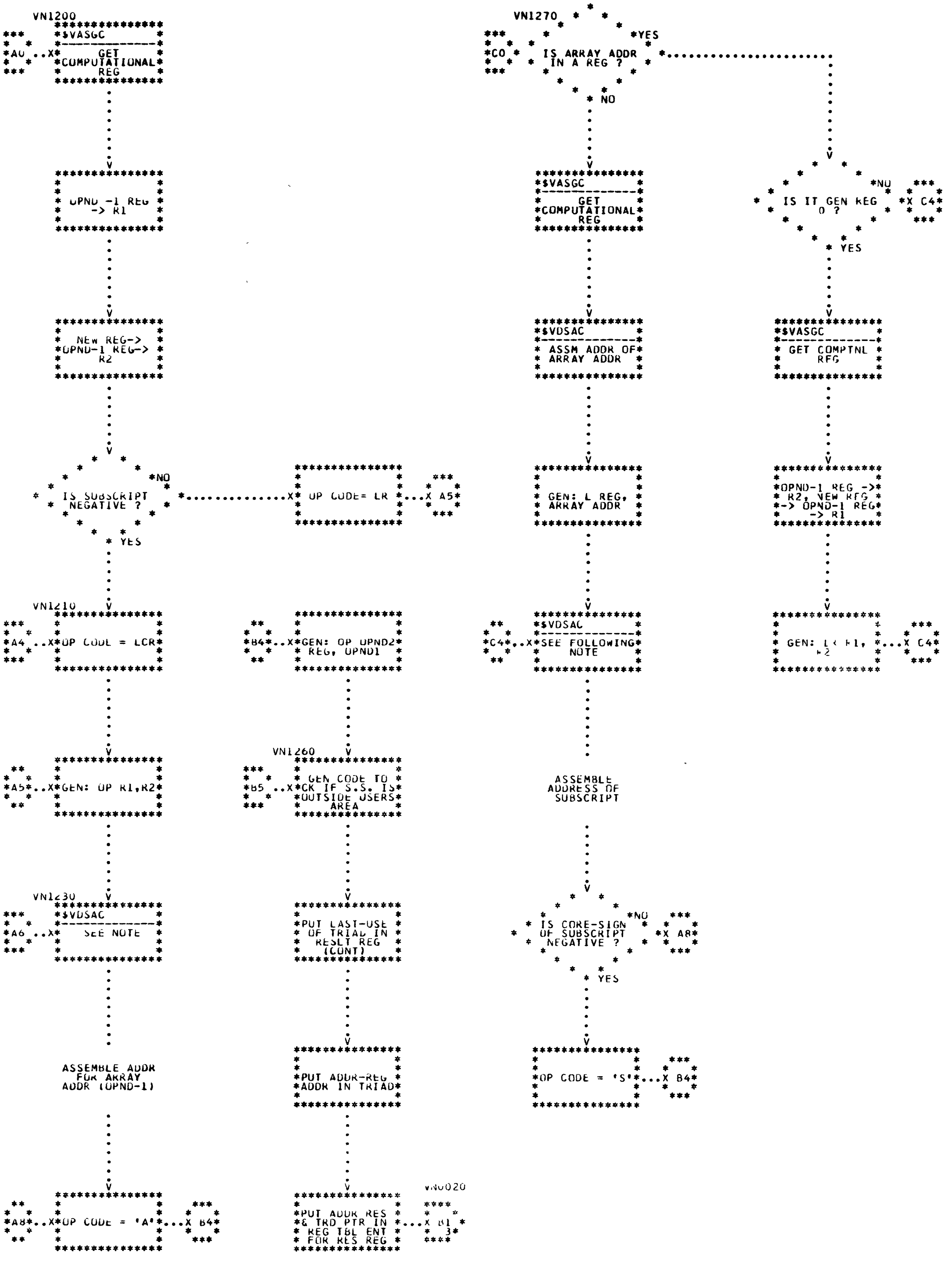


Chart 61. Instruction Assembler (Page 18 of 32)

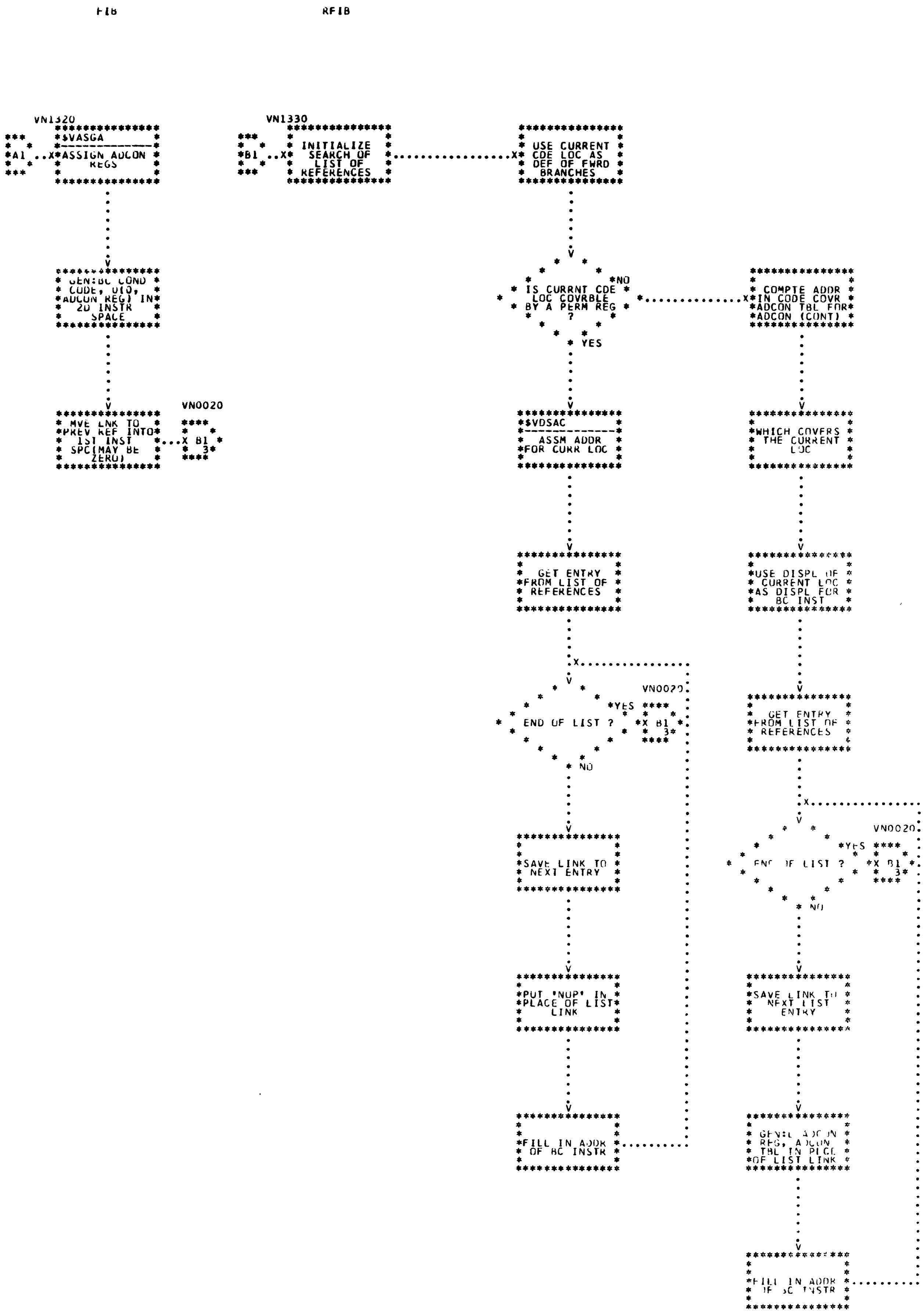
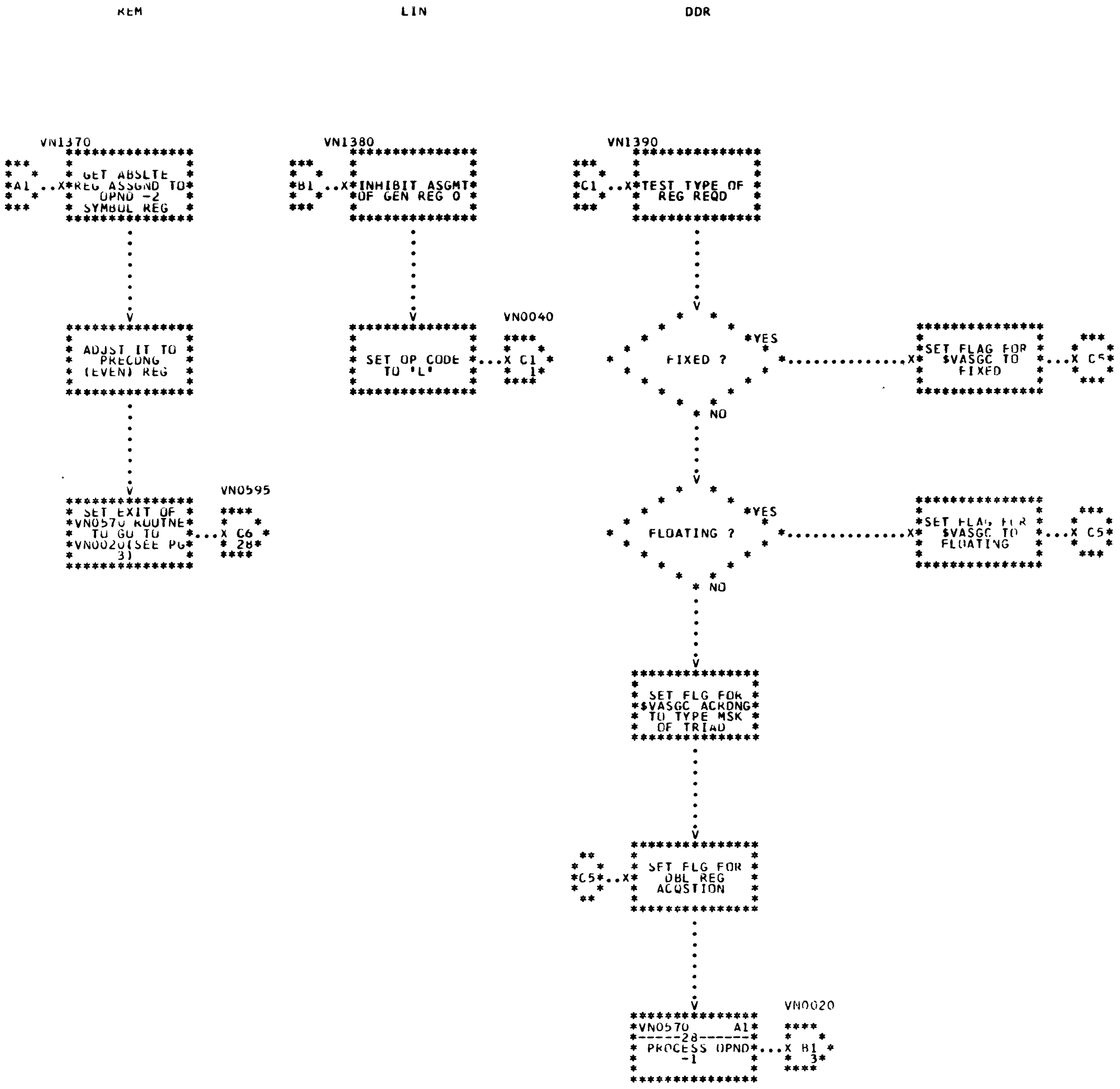


Chart 61. Instruction Assembler (Page 19 of 32)


REM
LIN
DDR
VN1370
A1
GET ABSLTE REG ASSGND TO OPND -2 SYMBOL REG
ADJUST IT TO PRECDNG (EVEN) REG
SET EXIT OF VN0570 ROUTNE TO GO TO VN0020(SEE PG 3)
VN0595
C6
28
VN1380
B1
INHIBIT ASGMT OF GEN REG 0
SET OP CODE TO 'L'
VN0040
C1
1
VN1390
C1
TEST TYPE OF REG REQD
FIXED ?
YES
NO
SET FLAG FOR $VASGC TO FIXED
C5
FLOATING ?
YES
NO
SET FLAG FOR $VASGC TO FLOATING
C5
SET FLG FOR $VASGC ACRDNG TO TYPE MSK OF TRIAD
C5
SET FLG FOR DBL REG ACQSTION
VN0570 A1
28
PROCESS OPND -1
VN0020
B1
3

FSLB

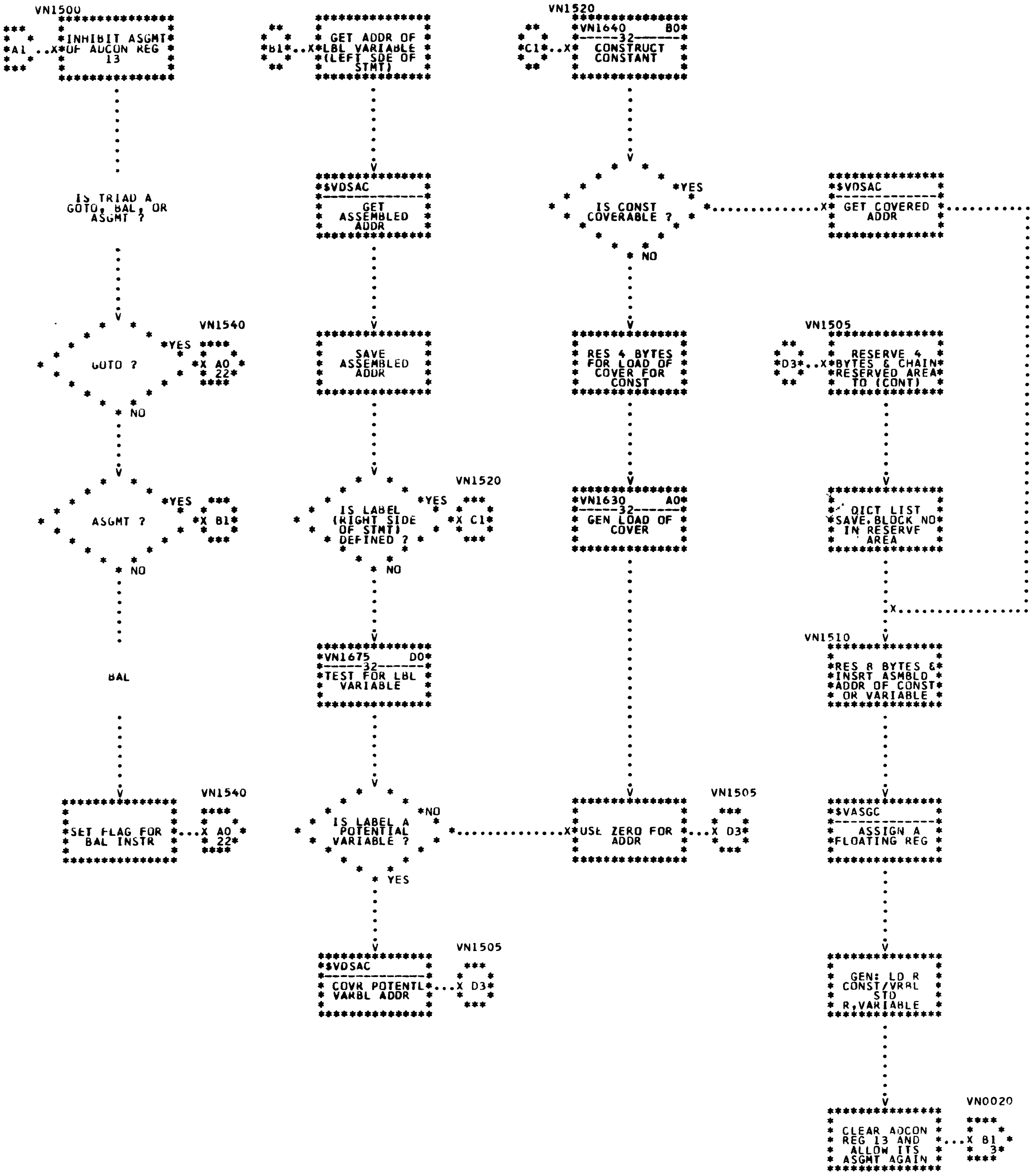


Chart 61. Instruction Assembler (Page 21 of 32)

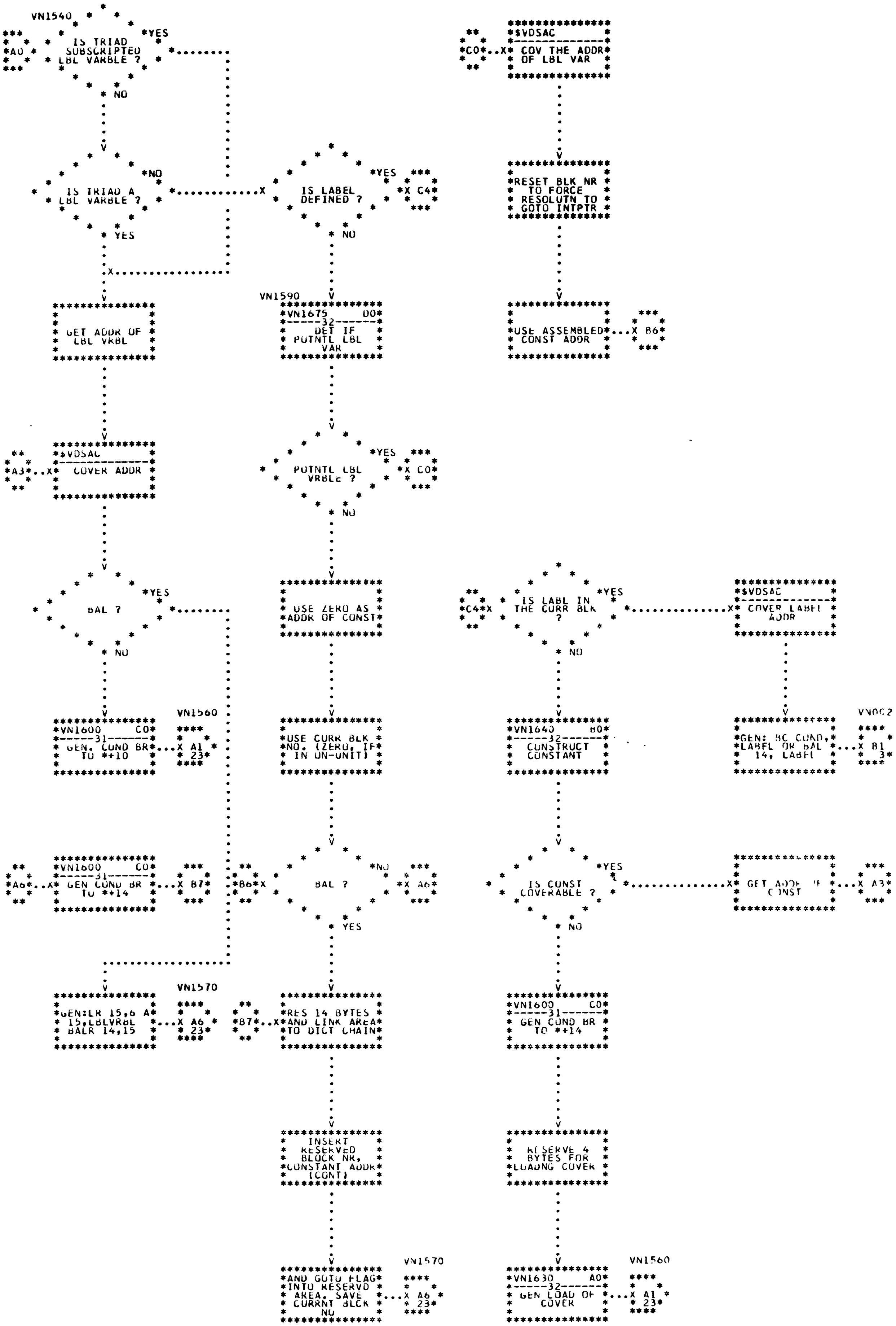


Chart 61. Instruction Assembler (Page 22 of 32)

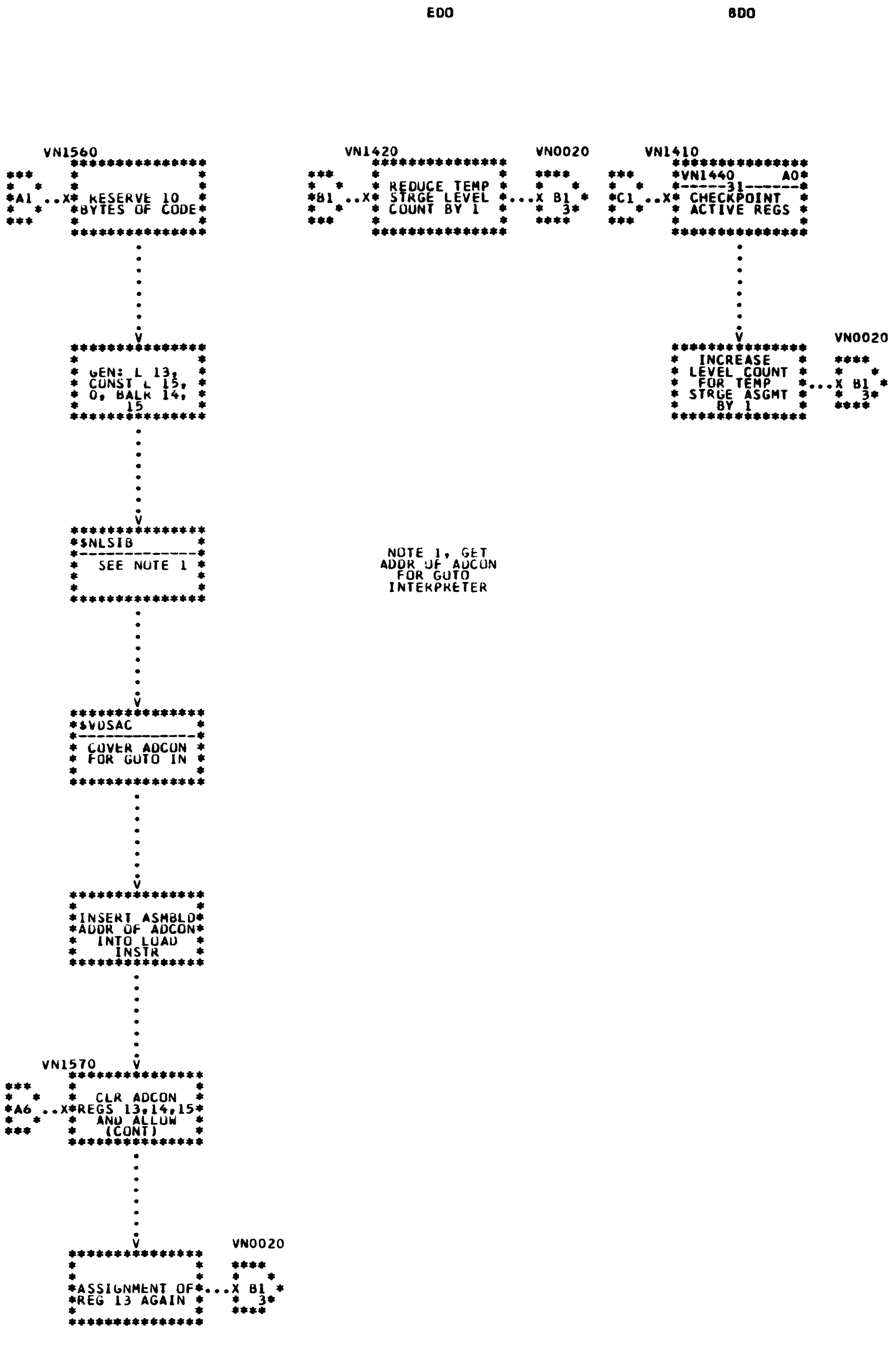


Chart 61. Instruction Assembler (Page 23 of 32)

RSLB

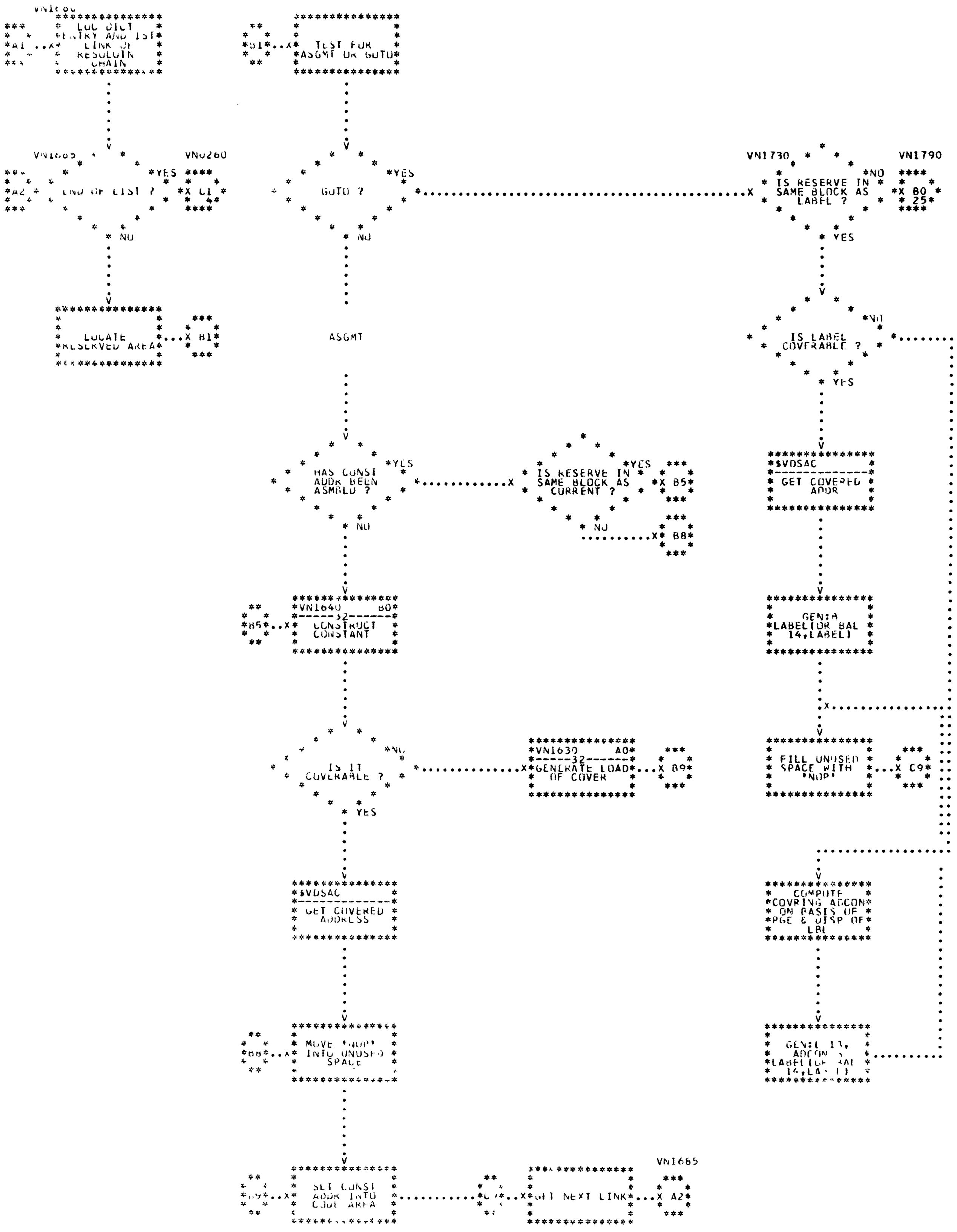


Chart 61. Instruction Assembler (Page 24 of 32)

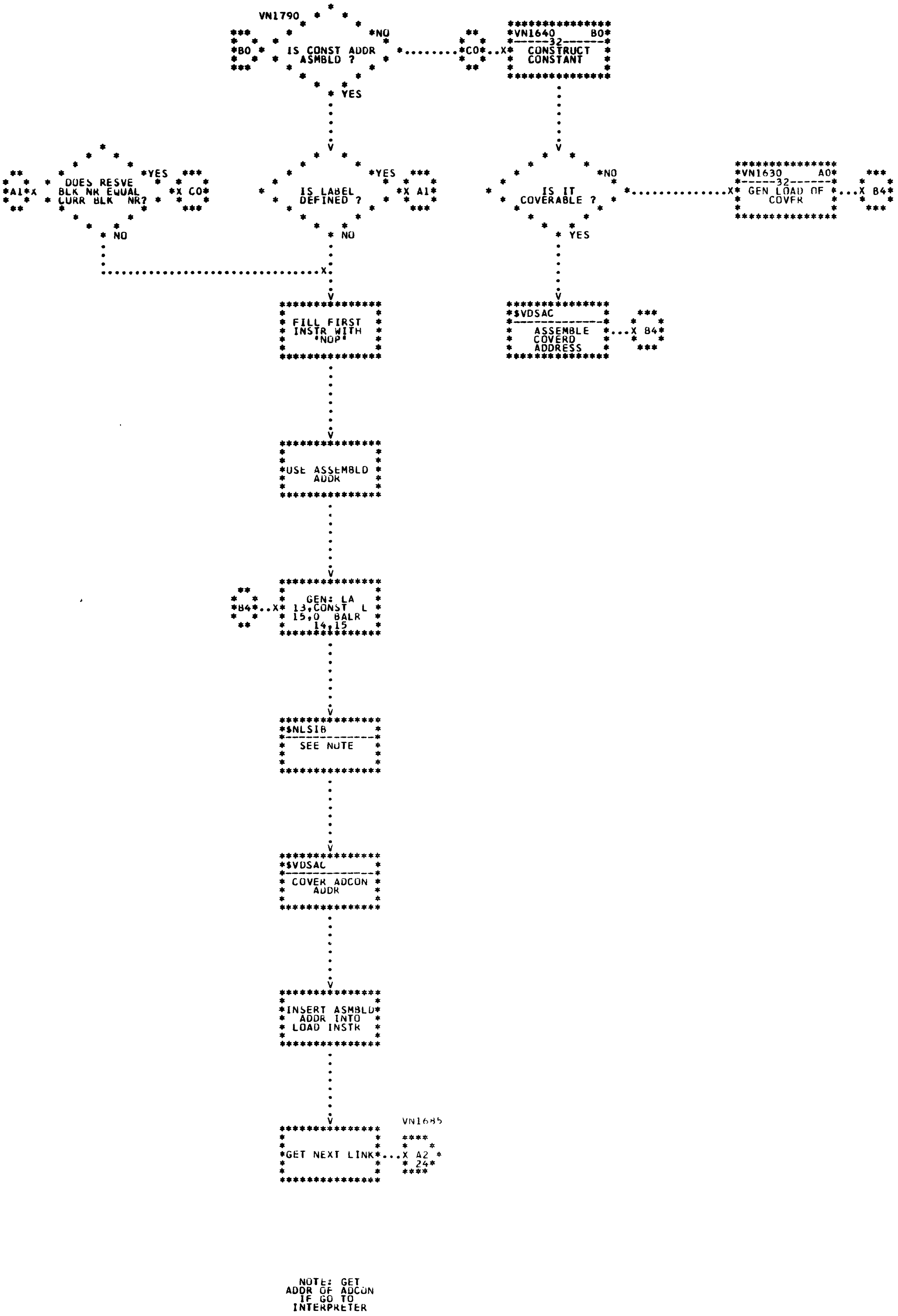


Chart 61. Instruction Assembler (Page 25 of 32)

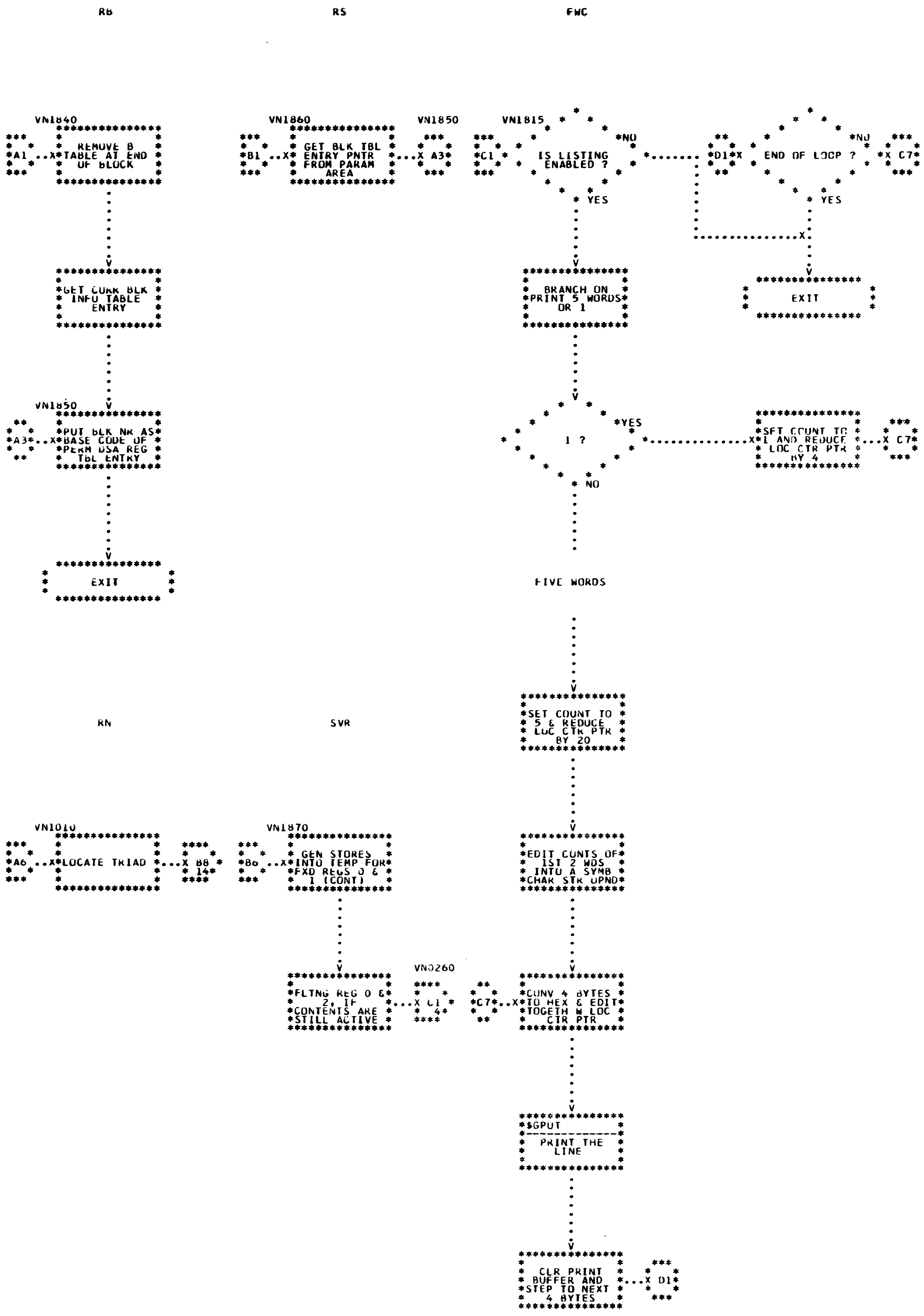


Chart 61. Instruction Assembler (Page 26 of 32)

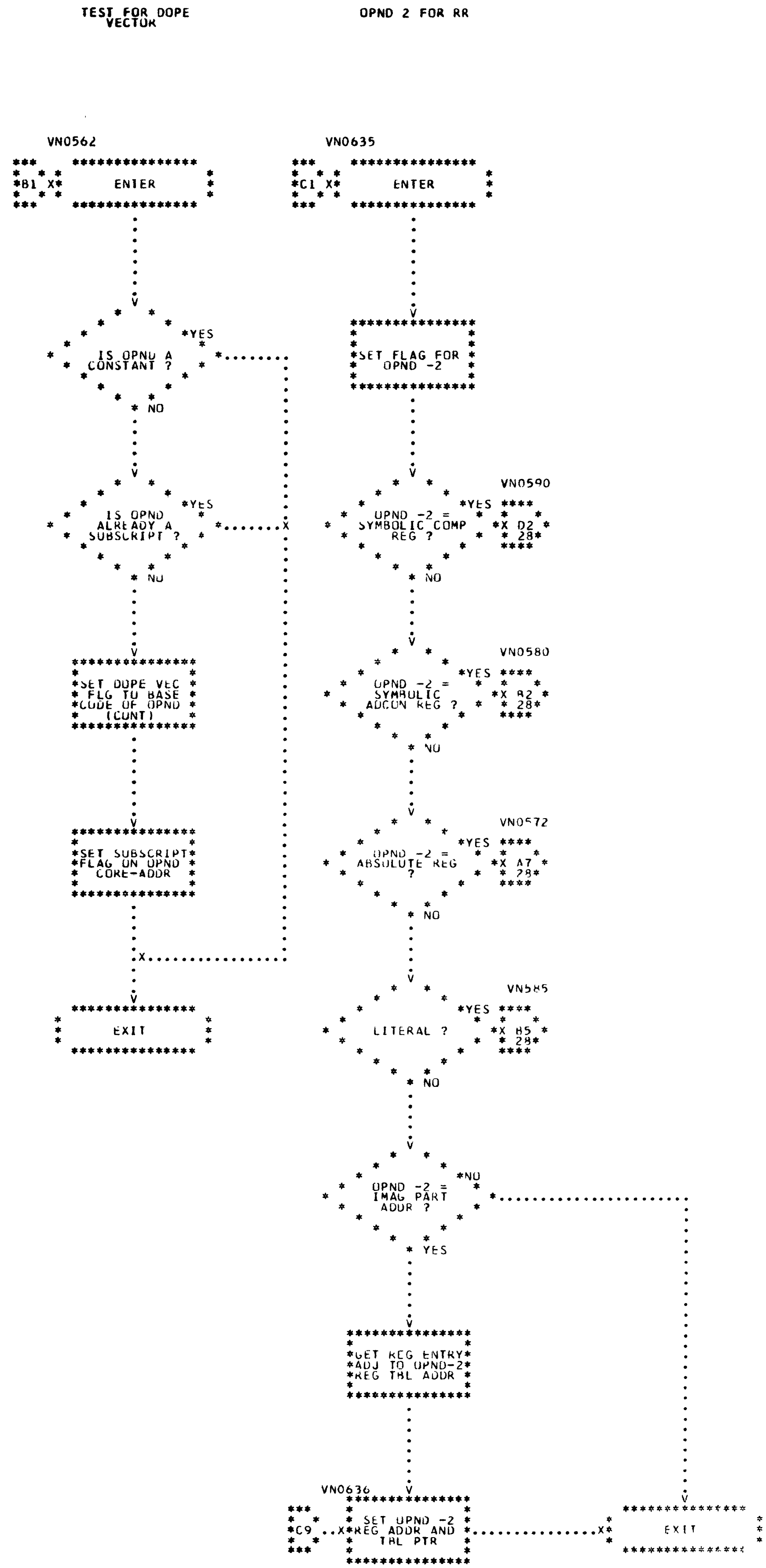


Chart 61. Instruction Assembler (Page 27 of 32)

PROCESS OPND 1

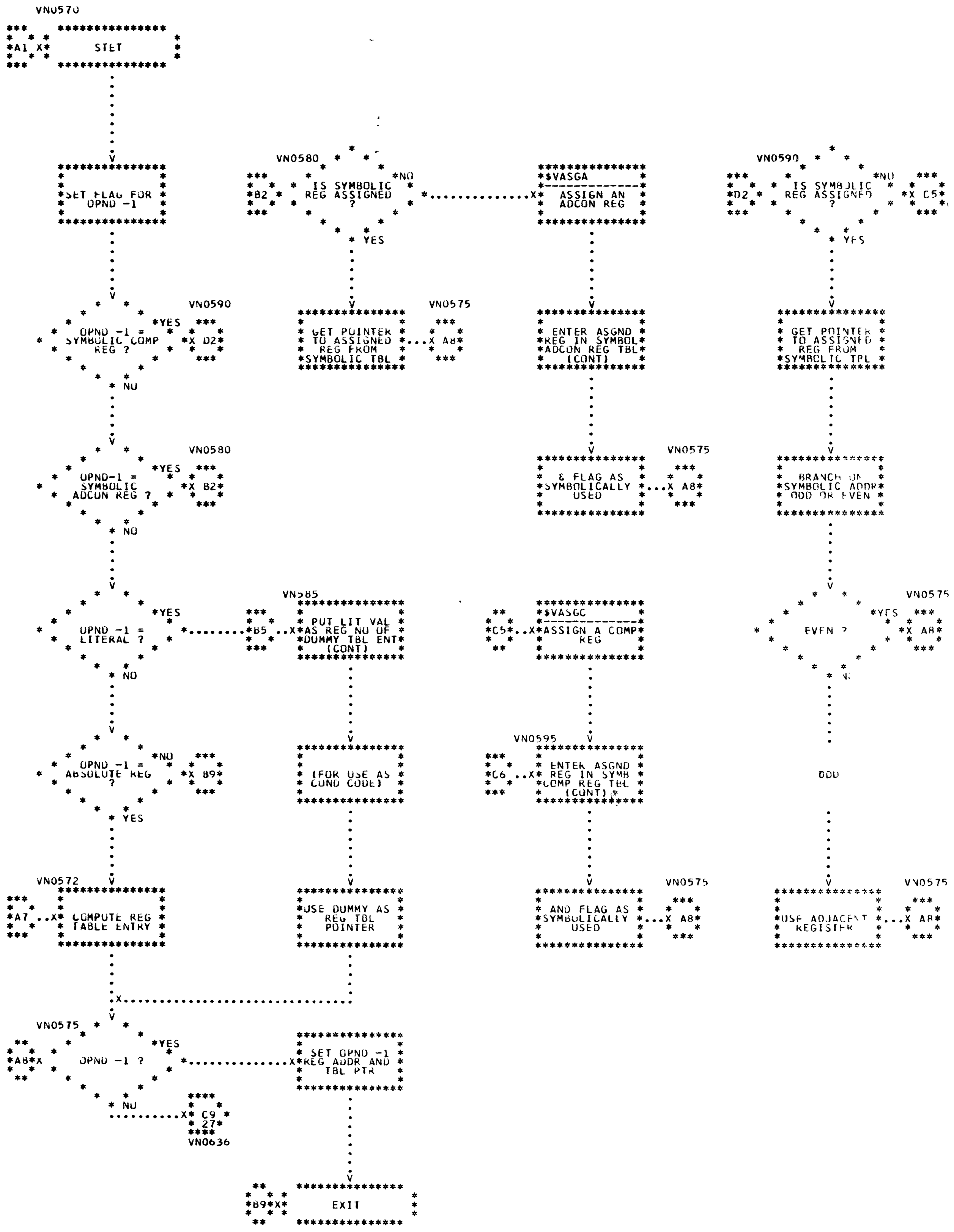


Chart 61. Instruction Assembler (Page 28 of 32)

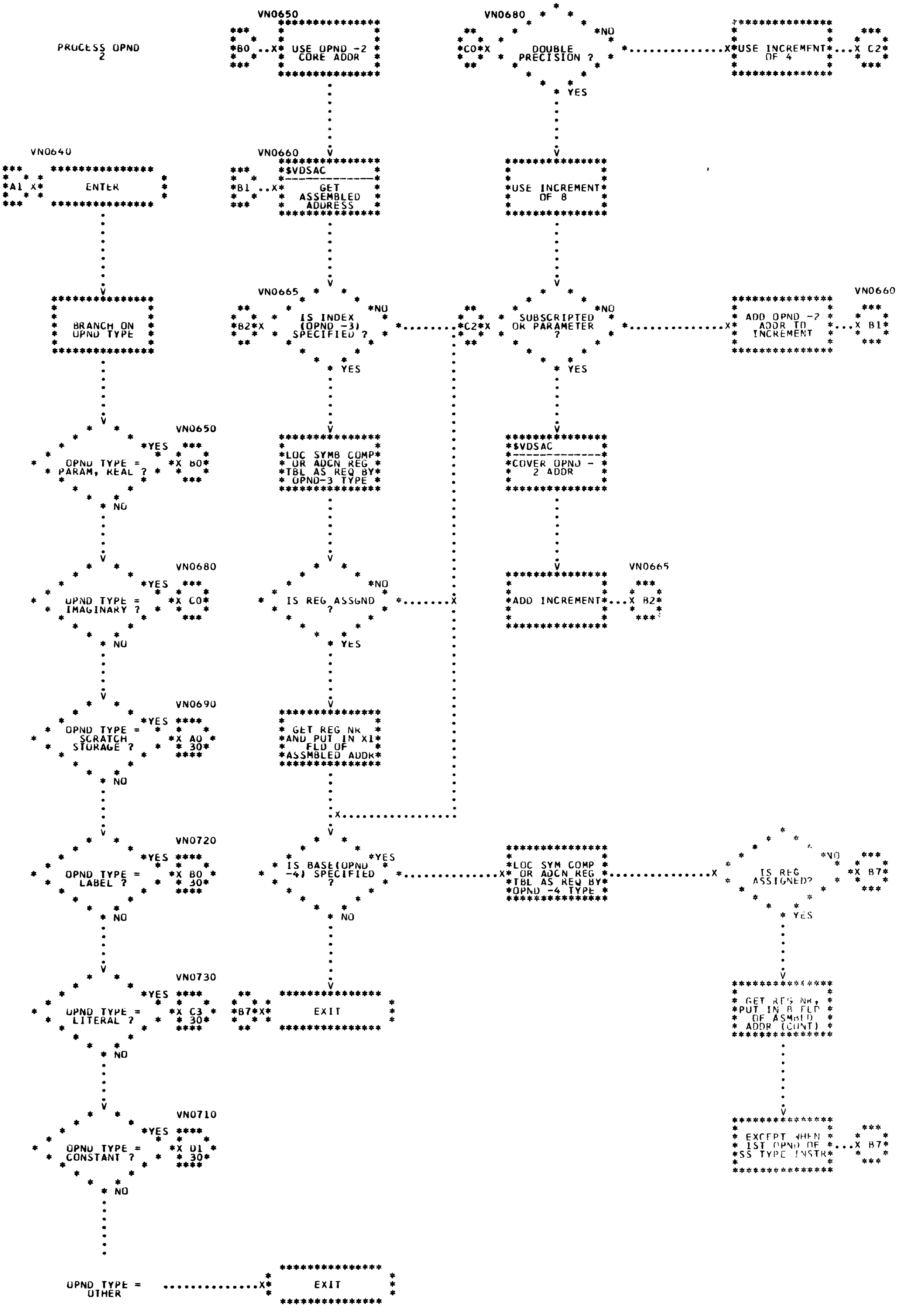
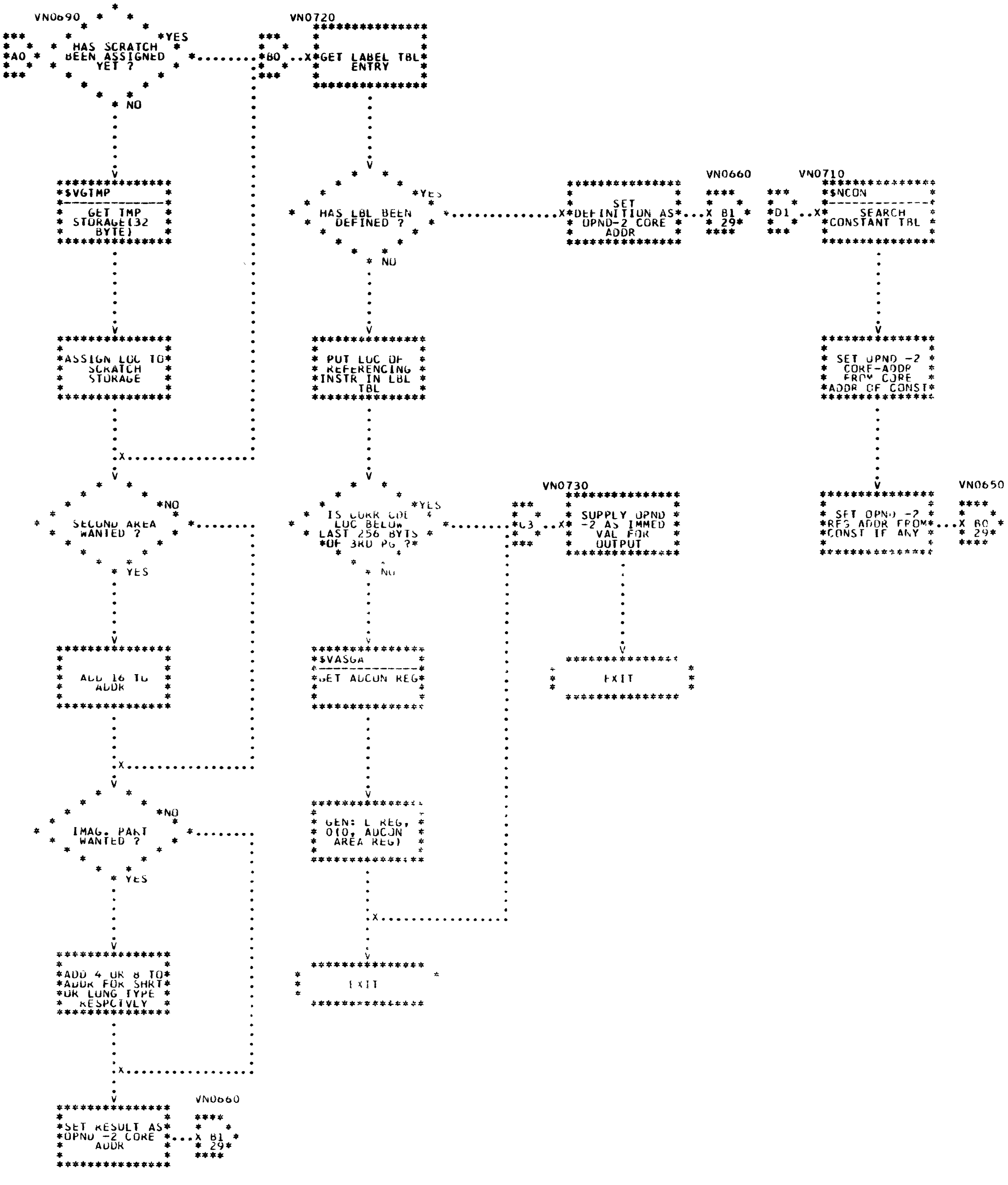
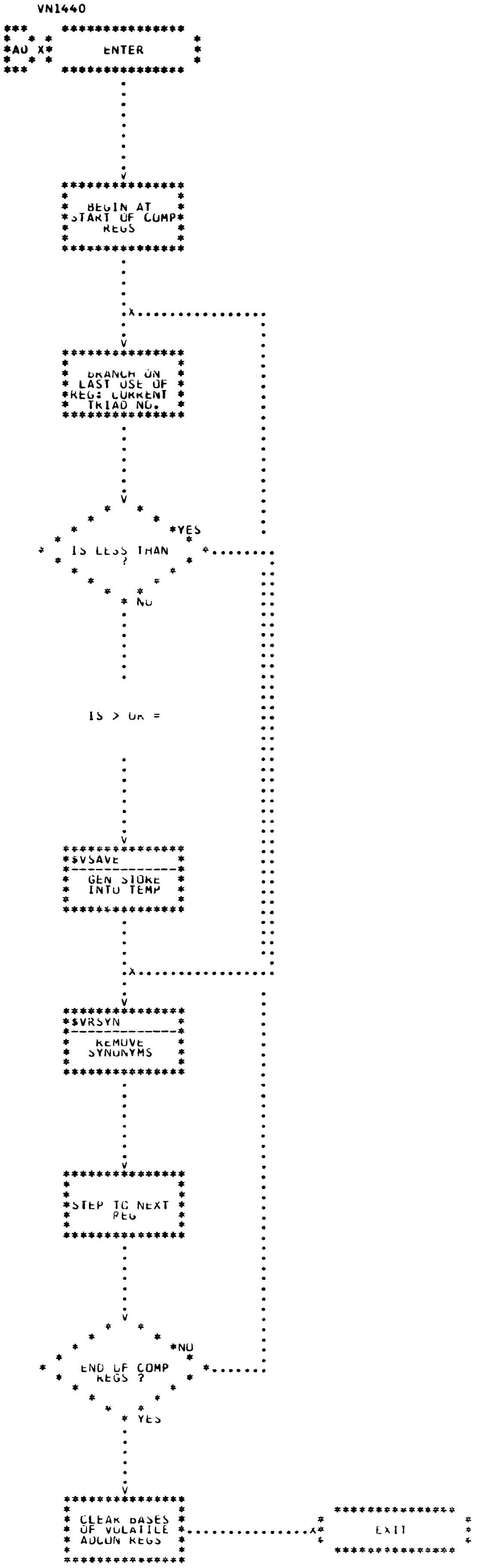


Chart 61. Instruction Assembler (Page 29 of 32)


VN0690
A0
HAS SCRATCH BEEN ASSIGNED YET ?
YES
NO
$VGTMP
GET TMP STORAGE(32 BYTE)
ASSIGN LOC TO SCRATCH STORAGE
SECOND AREA WANTED ?
NO
YES
ADD 16 TO ADDR
IMAG. PART WANTED ?
NO
YES
ADD 4 OR 8 TO ADDR FOR SHRT OR LONG TYPE RESPCTIVLY
SET RESULT AS OPND -2 CORE ADDR
VN0660
X B1 29
VN0720
B0
GET LABEL TBL ENTRY
HAS LBL BEEN DEFINED ?
YES
NO
PUT LOC OF REFERENCING INSTR IN LBL TBL
IS CORR LOC LOC BELOW LAST 256 BYTS OF 3RD PG ?
YES
NO
$VASGA
GET ADCON REG
GEN: L REG, 0(0, ADCON AREA REG)
EXIT
SET DEFINITION AS OPND-2 CORE ADDR
VN0660
X B1 29
VN0730
C3
SUPPLY OPND -2 AS IMMED VAL FOR OUTPUT
EXIT
VN0710
D1
$NCON
SEARCH CONSTANT TBL
SET OPND -2 CORE-ADDR FROM CORE ADDR OF CONST
SET OPND -2 REG ADDR FROM CONST IF ANY
VN0650
X B0 29



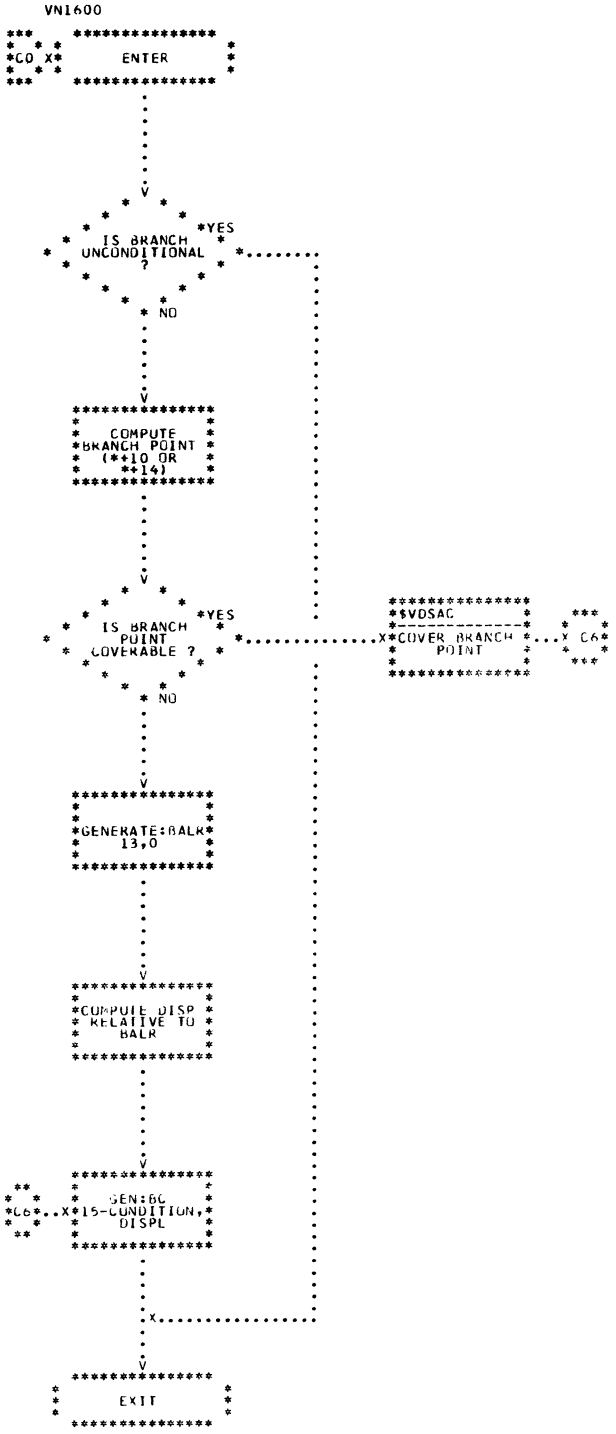


Chart 61. Instruction Assembler (Page 31 of 32)

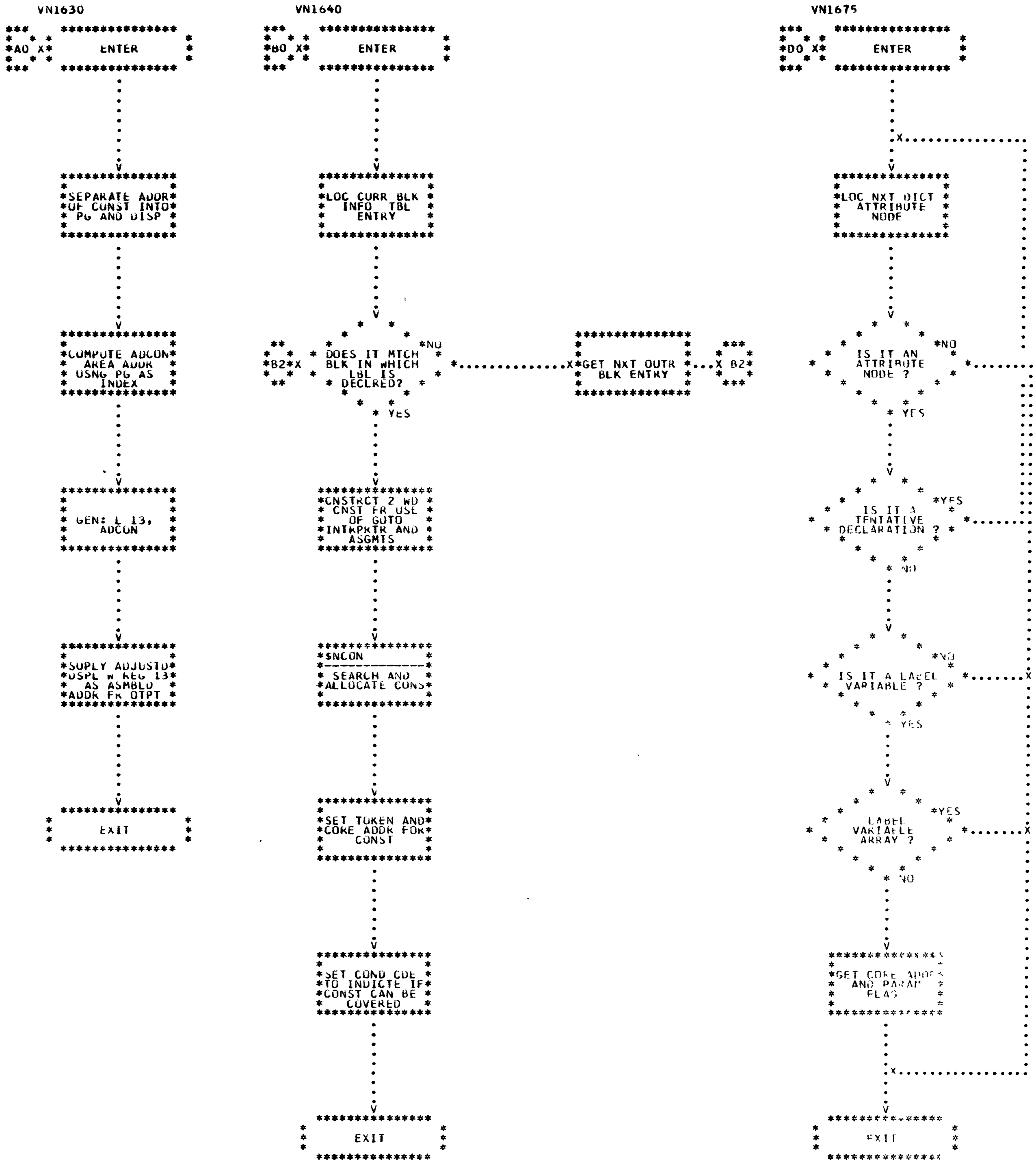


Chart 61. Instruction Assembler (Page 32 of 32)

PART 8 - WRAP-UP

The routines which perform compilation wrap-up are described in this subsection. The routines are discussed in alphabetic order according to their mnemonics, as shown below. Detailed logic diagrams for the routines appear at the end of the subsection.

Adcon Initialization ($HAINI)
Constant Table Processor ($HCTP)
Dope Vector List Processor ($HDVTP)
Line Number Table Processor ($HLNTP)
Runtime Library Loader ($HRTLL)
Static Constants-Adcon Loader ($HSCAL)
Table Collapse ($HTCR)
Compilation Wrap-Up Driver ($MCWU)

TITLE: ADCON INITIALIZATION ($HAINI)

Program Definition

Purpose and Usage

The Adcon Initialization routine converts all adcons from their base code and displacement form to true addresses and resets the user area-relocation constants in the communications area.

Description

PSPTR is set to point to the start of the library adcon area, and all adcons in this area are initialized before the swap flag (SWPFLG) is set to inhibit swapping. Each adcon in this area is initialized by forming the true address in a P-register, incrementing PSLTH to cover the adcon, and storing the true address back into the adcon.

Next the swap flag is set, and the compiler relocation pointers (CSPTR, CSLTH, and CSREG) are cleared in the communications area. The remainder of the adcons are then initialized in sequence, starting at the front of the adcon area. This initialization includes initialization of the file control blocks and allocation of space for the disk files when necessary, and initialization of the adcons covering the generated object code, the static and constants area, and the library work spaces.

PSPTR and PSLTH are then reset to cover the entire adcon area and PSREG is set to the execution-time relocatable register configuration (registers 6 through 15).

Errors Detected

SPACE FOR COMPILED CODE EXCEEDED. (113)

The adcon area is set up to include a total of 15 adcons to cover the generated object code and the static and constants area. During adcon initialization, a check is made to insure that the total number of adcons needed for this purpose does not exceed 15; if it does, the message is printed and compilation terminated.

Local Variables

None

Program Interface

Entry Points

$HAINI. No formal parameters.

Exit Conditions

Normal exit. Return to caller. Final exit to CALL/360-OS Executive if the source program is too large.

Routines Called

$GPUT	Output Director
$SVC	SVC Director

Global Variables

CSPTR	Processor Swap Compiler Save Pointer (Communications Area)
CSLTH	Processor Swap Compiler Save Length (Communications

	Area)
CSREG	Processor Swap Compiler Registers (Communications Area)
PSPTR	User Program Swap Compiler Save Pointer (Communications Area)
PSLTH	User Program Swap Compiler Save Length (Communications Area)
PSREG	User Program Swap Compiler Save Registers (Communications Area)
SWPFLG	Swap Flag (Communications Area)
L Table	Library Load Table
$LNTA	Displacement from Code to Line Number Table
$CAA	Displacement from Code to Constants Area
$AAA	Displacement from Code to Adcon Area
$RTLA	Displacement from Code to Library Area
$LWSA	Displacement from Code to LWS Area
$IOBA	Displacement from Code to I/O Buffer Area
$SASA	Displacement from Code to Static Array Area
$DSAA	Displacement from Code to DSA Area
$DBUF	Number of Disk File Buffers
TMBUF	Pointer to Terminal I/O Buffer
$COMAD	Address of Communications Area
$GABU	Print Buffer
$MFCB	Mask for FCB Control Bytes
$MLWS	Displacement from Code to Address-Modifiable LWS

Logic Diagram

Chart 62 shows the detailed logic diagram for the Adcon Initialization routine.

TITLE: CONSTANT TABLE PROCESSOR ($HCTP)

Program Definition

Purpose and Usage

The Constant Table Processor processes the constant table (C table) by moving constants to the static and constants area (adjacent to the line number table area) and, when necessary, converting string constants.

Description

The first and eleventh through seventeenth words of the constants area are cleared to zeros. (This area will later be used as the dynamic storage area for the external block.)

FILEPTR in communications area is initialized to the displacement from the start of user area to the FCIB offset table. This table is located at displacement X'1E0' from the start of the static and constants area.

The C table is then processed one entry at a time, first checking to determine whether the entry contains the constant or a pointer to a character-string constant in the source program. If the entry contains the constant, it is moved to the location in the constants area indicated by a pointer in the constant table entry. If, on the other hand, the entry contains a pointer to a character string in the source program, this character string is converted (that is, duplicate quotes removed) and placed in the constants area.

Errors Detected

None

Local Variables

None

Program Interface

Entry Points

$HCTP

Exit Conditions

Normal exit. Return to caller.

Routines Called

$WSTEP	Segment Management

Global Variables

C Table	Constant Table
$CAA	Displacement from Code to Constants Area
$TAREA	Translate Area

Logic Diagram

Chart 63 shows the detailed logic diagram for the Constant Table Processor routine.

TITLE: DOPE VECTOR LIST PROCESSOR ($HDVTP)

Program Definition

Purpose and Usage

The Dope Vector List Processor sets the pointers of the dope vectors of all static arrays and strings to point to the virtual origins or beginnings of the strings, respectively.

Description

The storage pointer of each dope vector pointed to by an entry in the dope vector list is set as follows:

- If the dope vector is a string, the pointer is set to point (from P1) to the start of the string in static storage.
- If the dope vector is an array/string-array, the pointer is set to the virtual origin of the array.

Errors Detected

None

Local Variables

None

Program Interface

Entry Points

$HDVTP

Exit Conditions

Normal exit. Return to caller.

Routines Called

None

Global Variables

$TSA	Address of Dope Vector List
$SASA	Displacement from Code to Static Array Area
$CAA	Displacement from Code to Constants Area
J List	Dope Vector List

Comments

The dope vector pointer cannot be set when the dope vector is moved to the constants area because static storage has not been allocated at that time.

Logic Diagram

Chart 64 shows the detailed logic diagram for the Dope Vector List Processor routine.

TITLE: LINE NUMBER TABLE PROCESSOR ($HLNTP)

Program Definition

Purpose and Usage

The Line Number Table Processor processes and moves the line number table (D table) adjacent to the generated object code.

Description

This program receives the line number table (D table) as input. The table is processed one entry at a time by locating the line number in the source program, converting it to packed decimal, and combining it with the object code offset from the new table entry. The table is formed following the generated object code.

Errors Detected

None

Local Variables

None

Program Interface

Entry Points

$HLNTP

Exit Conditions

Normal exit. Return to caller.

Routines Called

$WSTEP	Segment Management

Global Variables

$LNTA	Displacement from Code to (Procured) Line Number Table
D Table	Line Number Table

Comments

The program assumes integer line numbers equal to or less than seven characters in length.

Logic Diagram

Chart 65 shows the detailed logic diagram for the Line Number Table Processor routine.

TITLE: RUNTIME LIBRARY LOADER ($HRTLL)

Program Definition

Purpose and Usage

The Runtime Library Loader determines all library routines that must be loaded and then loads the routines and sets (to a base code and displacement) the adcons covering entry names invoked. This routine also allocates both fixed and address-modifiable library work space.

Description

This routine receives as input the library load table (L table) which contains, at this time, a pointer to the adcon for each library entry name invoked directly by the generated object code.

Using the library load table as the director, the jump table of each library routine containing an invoked entry is processed, setting the library load table entries (and, if necessary, assigning adcons) for lower-level entry names. This process is repeated at each level in the calling structure until the end of the structure has been reached, at which time the next sequential nonzero library load table entry is processed in the same manner. As entries are set in the library load table, a flag is set in the routine entry name processed table to prevent subsequent search of a calling structure already processed. (For descriptions of the library load table and routine entry name processed table, see Appendix B. For description of the jump table for a routine, see "Runtime Routine Structure" in Appendix A.)

Following the library load table search, all additional adcons assigned are cleared to zeros and the address-modifiable library work space is allocated. A second pass is made over the library load table to load the required runtime library routines. Before each routine is loaded, a check is made (against FREPTR) to determine if there is sufficient space available to load the routine without destroying any of the compiler's fixed tables area. If there is not, compilation is terminated and control is returned to the CALL/360-OS Executive.

If sufficient space is available, the routine is moved to the runtime library area, those adcons corresponding to entries to the routine that are invoked are set (to a base code and displacement), and the routine's jump table is processed. Jump table entries (corresponding to library entry names invoked by this routine) of those entry names whose associated adcon is not in a fixed place in the adcon area are replaced with a pointer to the associated adcon and moved with the routine.

As each routine is loaded, a flag is set in the high-order byte of each library load table entry corresponding to an entry to the routine. These entries may be ignored when encountered subsequently in processing the table.

After the library has been loaded, FREPTR is set to point to the end of the user's area and the fixed library work space is allocated.

Errors Detected

None

Local Variables

HRCSTK	Jump table address push-down stack
HRGSTK	Invoked entry count push-down stack
HRTLTH	Approximate length of the library

HRDISP	Displacement from code to where each routine is loaded
HRLTX	Load table index name area
HRDIR	Table, each entry of which is the displacement from the start of the table to a library routine
HRDIX	Table, each entry of which corresponds to an entry to a library routine. The content of each entry, when multiplied by four, becomes the displacement from the front of the HRDIR to the HRDIR entry for the routine which contains the entry.

Program Interface

Entry Points

$HRTLL

Exit Conditions

Normal exit. Return to caller.

Exit to CALL/360-OS Executive to get more space and recompile the program. No return.

Routines Called

$GPUT	Output Director
$SVC	SVC Director

Global Variables

L Table	Library Load Table Routine Entry Name Processed Table
UTT	User Terminal Table
$ACODE	Pointer to Next Available Byte in Object Code Area
$LIBBC	Library Base Code
$LTEND	Length of the Library Load Table
$MLWS	Displacement from Code to Address-Modifiable LWS
FREPTR	Pointer to First Available Word in Compiler's Variable Data Area
L$LIBX	Adcon Displacement
$AAA	Displacement from Code to Adcon Area
$LWSA	Displacement from Code to LWS Area
$RTLA	Displacement from Code to Library Area
ZCNT	Length of the Fixed Library Work Space (LWS)
ZCNTP	Length of the Address-Modifiable Library Work Space (LWS)

Logic Diagram

Chart 66 shows the detailed logic diagram for the Runtime Library Loader routine.

TITLE: STATIC CONSTANTS-ADCON LOADER ($HSCAL)

Program Definition

Purpose and Usage

The Static Constants-Adcon Loader processes the initialization table (I table) and dope vector list (J list), initializing the static and constants adcon area.

Description

The initialization table is processed one entry at a time, performing those operations dictated by the entry type, then moving the entry to either the static and constants or adcon area, again as dictated by the entry type.

The following is a list of the initialization table entries, a description of the processing required before the entry is moved, and an indication of the area to which it is moved:

1. Data/Format Element Descriptor: Moved directly to the location in the static and constants area pointed to by the pointer in the I table entry.
2. Adcon: Moved directly to the location in the adcon area pointed to by the pointer in the I table entry.
3. String Dope Vector, Array/String Array Dope Vector: If the storage class is STATIC, a dope vector list entry pointing to the dope vector is built and then the dope vector is moved to the static and constants area.
4. Special SDV: The string pointer in the dope vector is set to point to the static and constants area and then the dope vector is moved to this area.
5. Special Block Adcon Area: A special base code is put in the first byte of the BAA (for use by $HAINI) and then the BAA is moved to the adcon area.
6. Discarded Entry: Ignored.

The J list is processed after the I table and contains only two types of entries: ADV/SADV entries too large to fit into the I table and discarded entries. The action required for both types is the same as described above for their equivalent I table entries.

Errors Detected

None

Local Variables

HSTBL	Branch table to process the various types of entries

Program Interface

Entry Points

$HSCAL

Exit Conditions

Normal exit. Return to caller.

Routines Called

$WSTEP	Segment Management

Global Variables

I Table	Initialization Table
J List	Dope Vector List
$AAA	Displacement from Code to Adcon Area
$CAA	Displacement from Code to Constants Area
$TSA	Address of First Word Boundary

Logic Diagram

Chart 67 shows the detailed logic diagram for the Static Constants-Adcon Loader routine.

TITLE: TABLE COLLAPSE ($HTCR)

Program Description

Purpose and Usage

The Table Collapse routine collapses the J list, C table, D table, and I table to obtain working space during compilation wrap-up.

Description

If, at any time during the compilation wrap-up phase prior to the loading of the runtime library, there is insufficient free space in the variable table area (that is, line number table, static and constants, or adcon areas), this routine is called to collapse the C table, D table, I table, and J list entries that remain to be processed at that time.

The tables are collapsed into an area bounded by the start of the register table area (R$TBL) in lower core and the start of the first I table entry in higher core. Each table in turn is moved an entry at a time and the linkage adjusted until all have been moved or until this area has been filled. In the latter case, an exit is made to the CALL/360-OS Executive to get more memory and to recompile the program. If the tables are collapsed successfully, a flag ($MCOL) is set to prevent a subsequent attempt to collapse them and control is returned to the calling program.

Errors Detected

None

Local Variables

HTHEAD	Pointers to the table heads
HTACTV	Pointers to the table active entry
HTTAIL	Pointers to the table tails

Program Interface

Entry Points

$HTCR

Note: Upon entry, register G0 contains the count of the number of tables active (remaining to be processed) at the time.

Exit Conditions

Return to caller if tables are collapsed successfully.

Exit to CALL/360-OS Executive if collapse is not successful, or if tables have already been collapsed, and recompile the program. No return.

Routines Called

$SVC	SVC Director

Global Variables

C Table	Constant Table
D Table	Line Number Table
I Table	Initialization Table
J List	Dope Vector List

$MCOL	Tables Collapsed Flag
C$ACTV	C Table Active-Entry Pointer
C$HEAD	C Table Head-of-Table Pointer
C$TAIL	C Table End-of-Table Pointer
I$ACTV	I Table Active-Entry Pointer
I$HEAD	I Table Head-of-Table Pointer
I$TAIL	I Table End-of-Table Pointer
D$ACTV	D Table Active-Entry Pointer
D$HEAD	D Table Head-of-Table Pointer
D$TAIL	D Table End-of-Table Pointer
J$HEAD	J List Head-of-Table Pointer
R$TBL	Pointer to Base of Register Table
FREPTR	Pointer to First Available Word in Compiler's Variable Data Area

Logic Diagram

Chart 68 shows the detailed logic diagram for the Table Collapse routine.

TITLE: COMPILATION WRAP-UP DRIVER ($MCWU)

Program Definition

Purpose and Usage

Following generation of the object code, the Compilation Wrap-Up Driver is called to perform the housekeeping necessary to prepare the code for execution and start execution.

Description

This program is the driver for the second phase of the compiler. As such, its primary functions are:

1. Call the various routines necessary to process the tables generated in phase 1 and initialize the user's area in preparation for execution of the generated object program.
2. Keep track of the utilization of space within the user's area and, if necessary, go to the CALL/360-OS Executive to:
    a. Get more memory.
    b. Through the Table Collapse routine ($HTCR) or the Runtime Library Loader routine ($HRTLL), terminate compilation when there is insufficient space to continue processing.
3. Initiate execution of the object program.

Errors Detected

None

Local Variables

MCPID	Displacement from the communications area to the start of the object code area.
MCINTR	Instructions moved into ARINTRP of the communications area for CALL/360-OS PL/I. When a program check occurs, the Executive transfers to ARINTRP. The instructions moved in will: • Save registers 6-15 in the adcon area • Set the program mask from the second word of PSW2SV of the communications area • Save the second word of PSW2SV in the adcon area • Transfer to IHEERRA
MCPRGBN	Code which initializes registers and sets the interrupt mask. This is moved into PRGBN of the communications area at the start of execution.

Program Interface

Entry Points

$MCWU

Exit Conditions

This routine contains eleven possible exits, all of which are normal exits and from which a return is expected except as noted below:

1. After the call to the CALL/360-OS Executive to initiate execution (SVC11), control is returned to location PRGBN in the communications area.

2. During any of the calls to either the Table Collapse routine ($HTCR) or the Runtime Library Loader routine ($HRTLL), control may be returned to the CALL/360-OS Executive and compilation terminated if there is insufficient space to continue processing in the user's area.

Routines Called

$HTCR	Table Collapse
$HLNTP	Line Number Table Processor
$HCTP	Constant Table Processor
$HSCAL	Static Constants-Adcon Loader
$HRTLL	Runtime Library Loader
$HDVTP	Dope Vector List Processor
$HAINI	Adcon Initialization

Global Variables

UTT	User Terminal Table
L Table	Library Load Table
$DBUF	Number of Disk File Buffers Needed (4 maximum @3712 bytes each)

Logic Diagram

Chart 69 shows the detailed logic diagram for the Compilation Wrap-Up Driver routine.

PART 8 LOGIC DIAGRAMS

The detailed logic diagrams for the routines which perform compilation wrap-up follow.

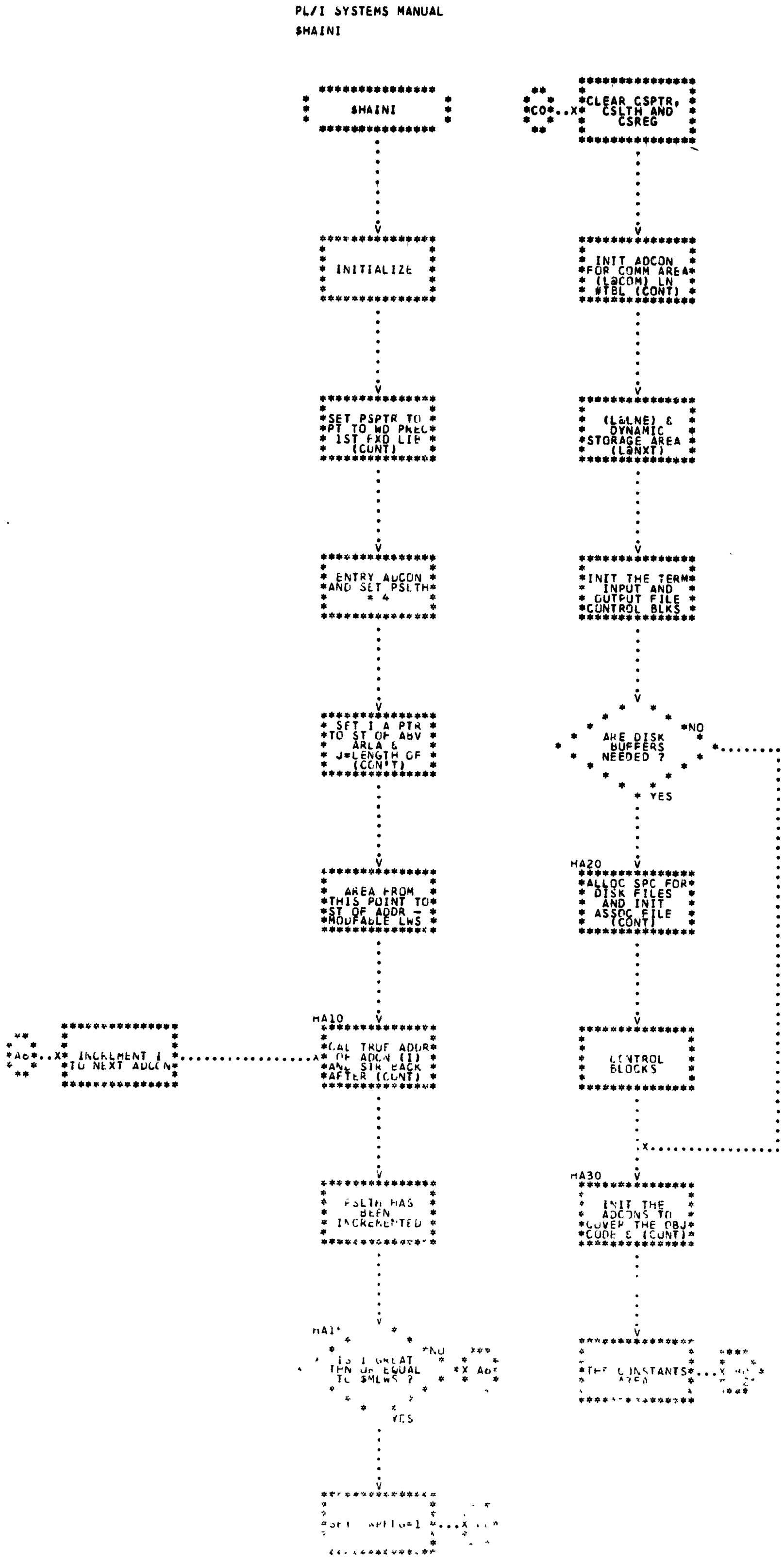


Chart 62. Adcon Initialization (Page 1 of 2)

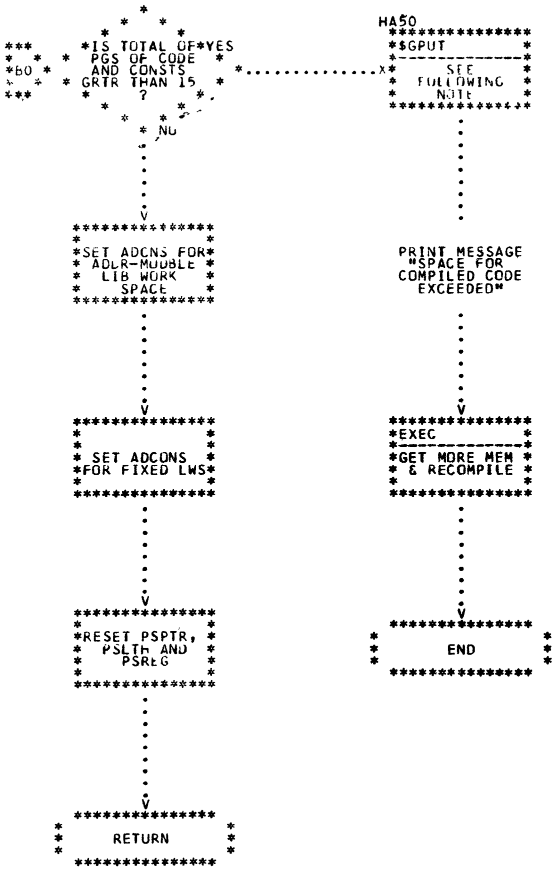


Chart 62. Adcon Initialization (Page 2 of 2)

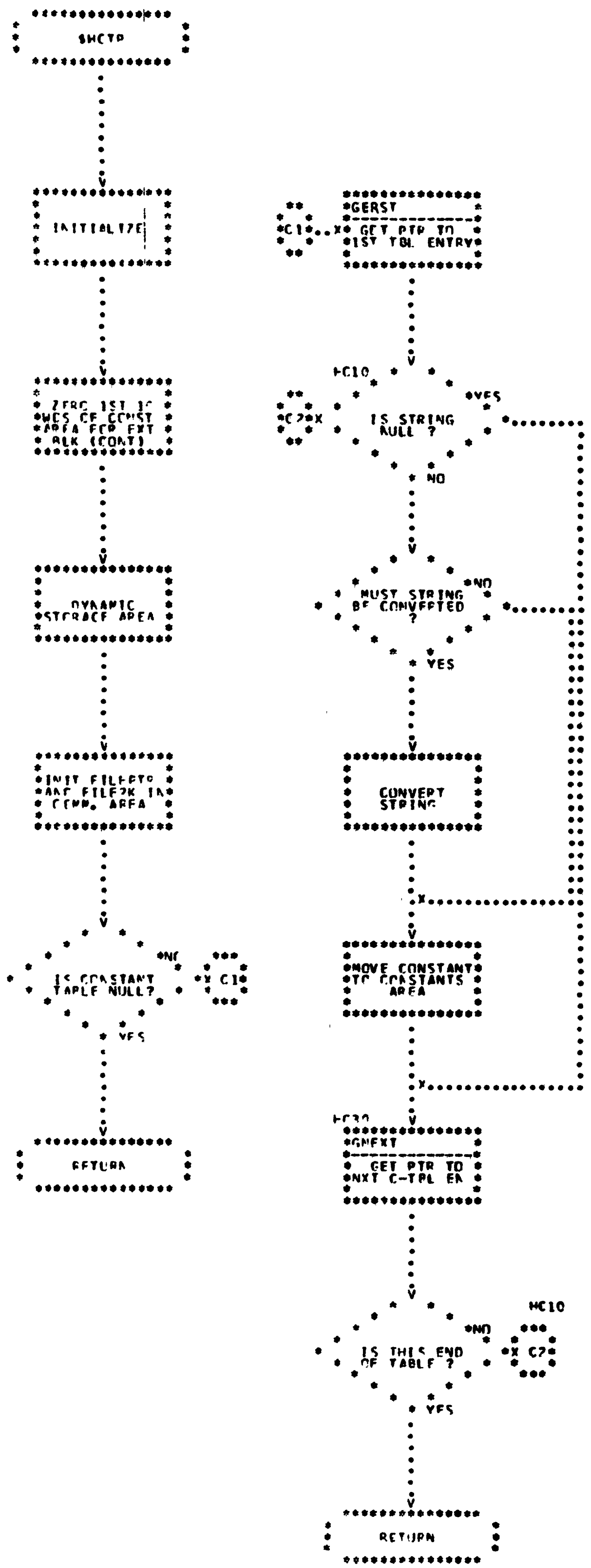


Chart 63. Constant Table Processor

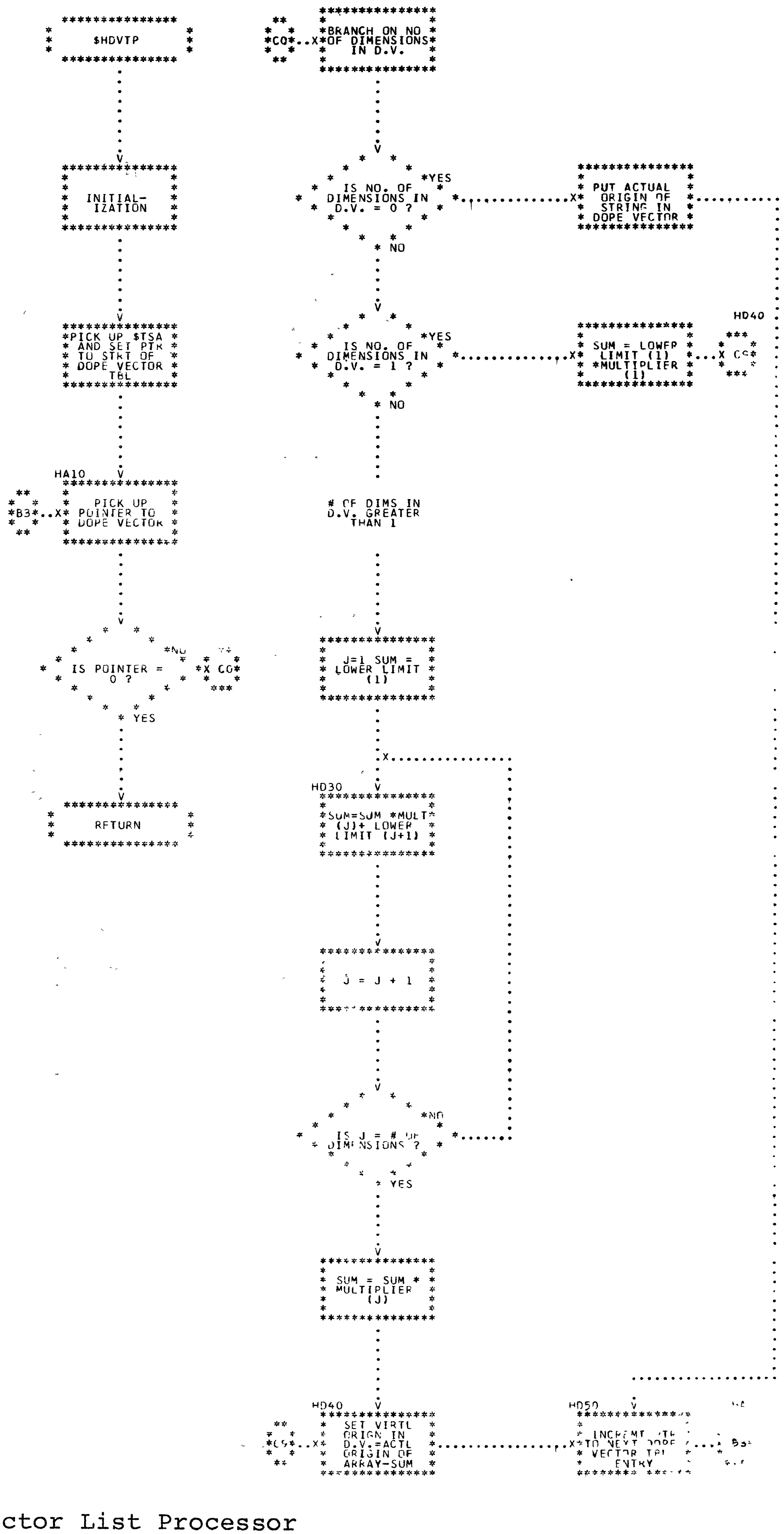


Chart 64. Dope Vector List Processor

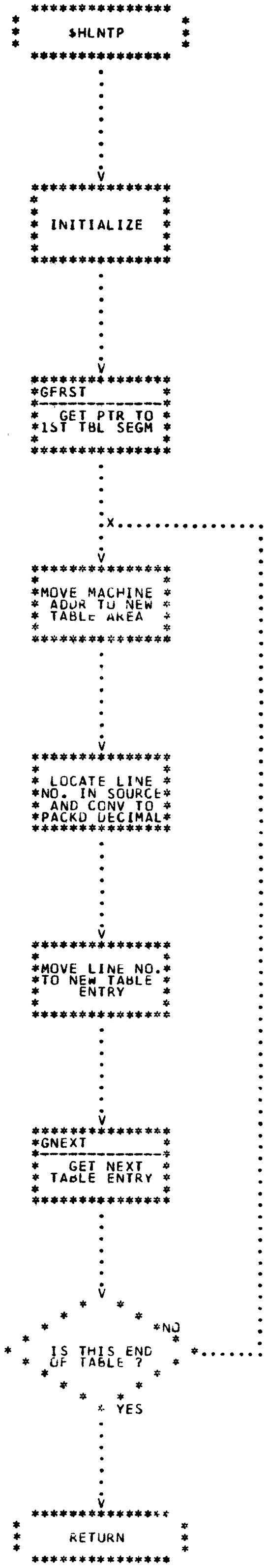


Chart 65. Line Number Table Processor

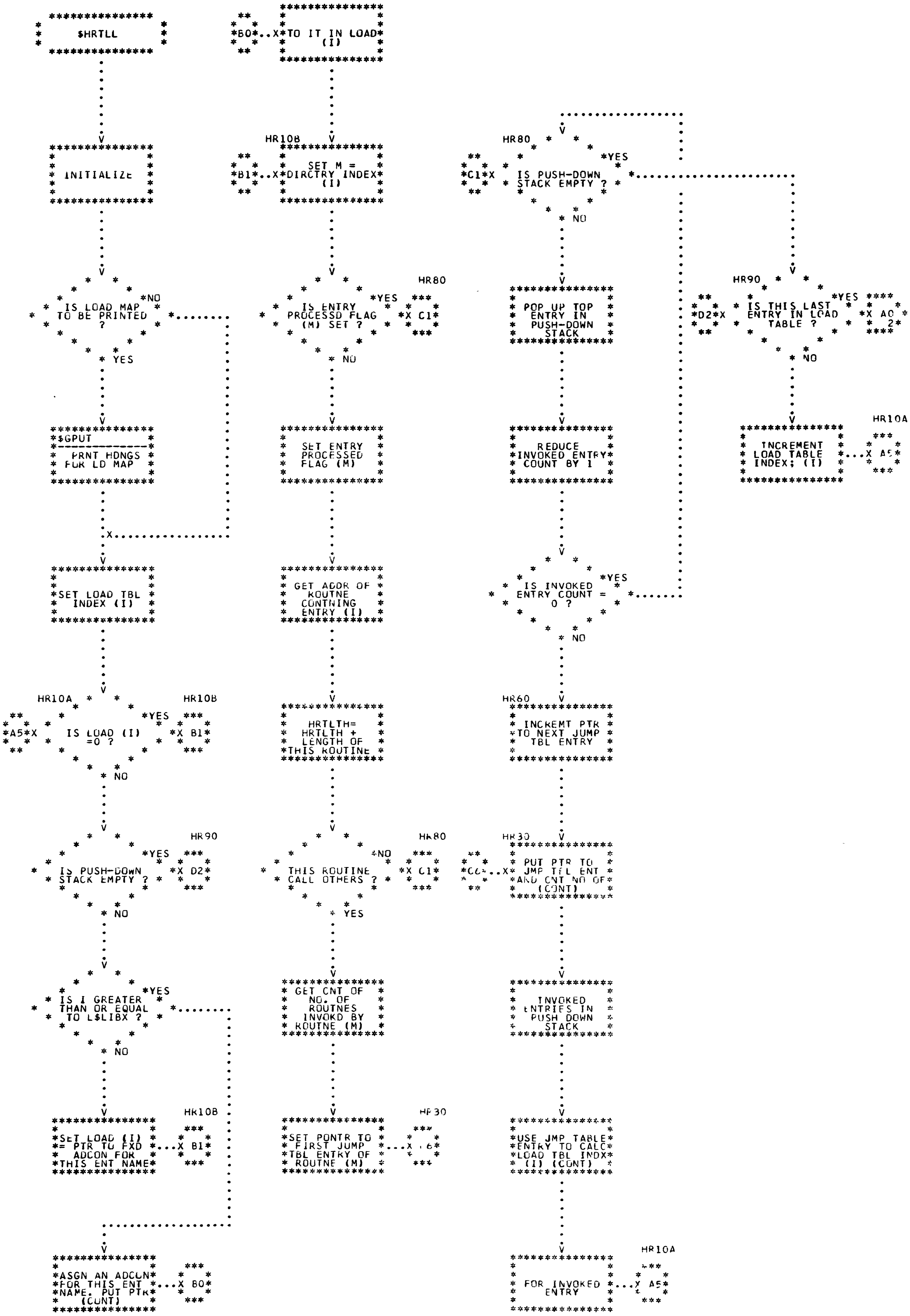


Chart 66. Runtime Library Loader (Page 1 of 2)

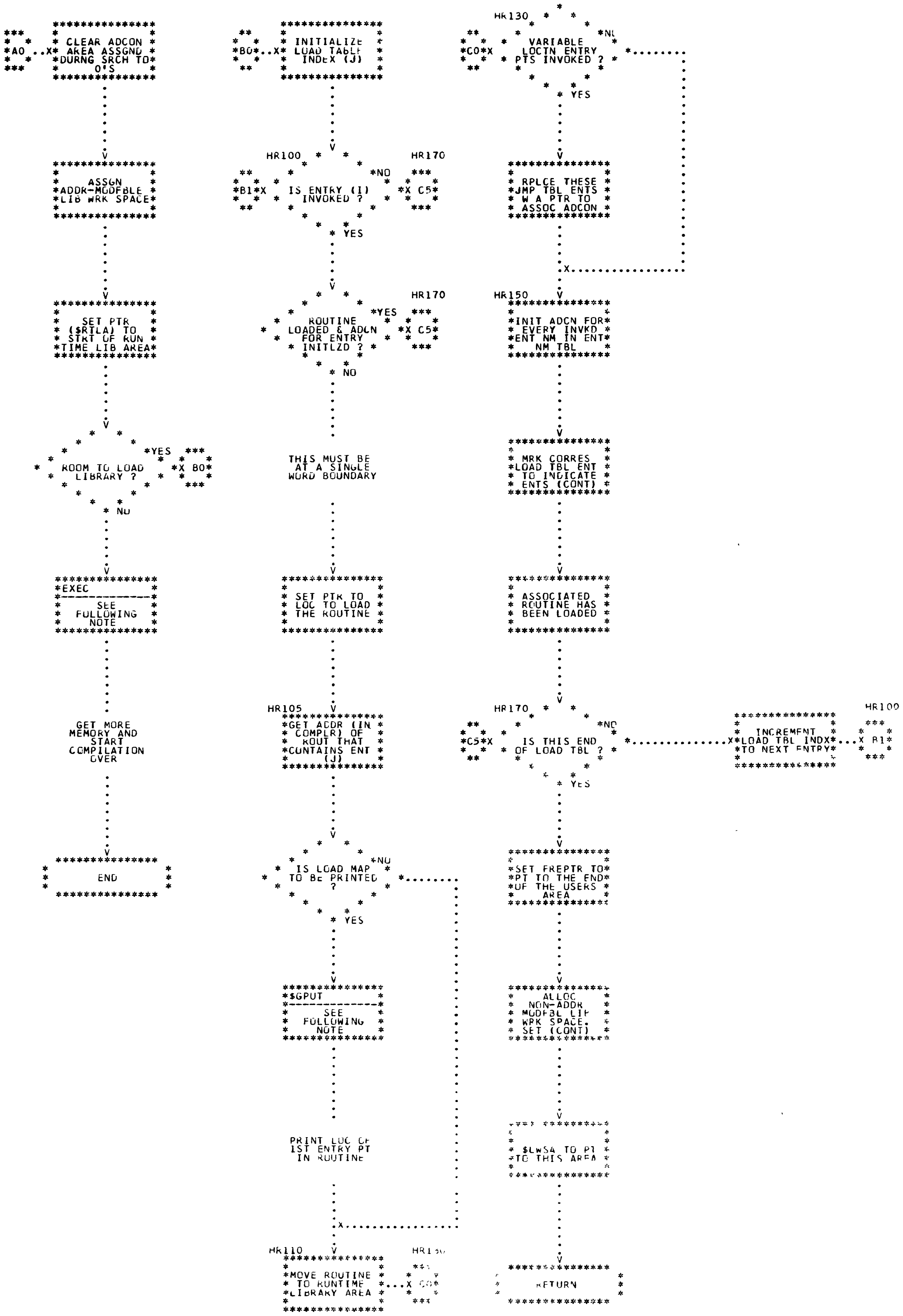


Chart 66. Runtime Library Loader (Page 2 of 2)

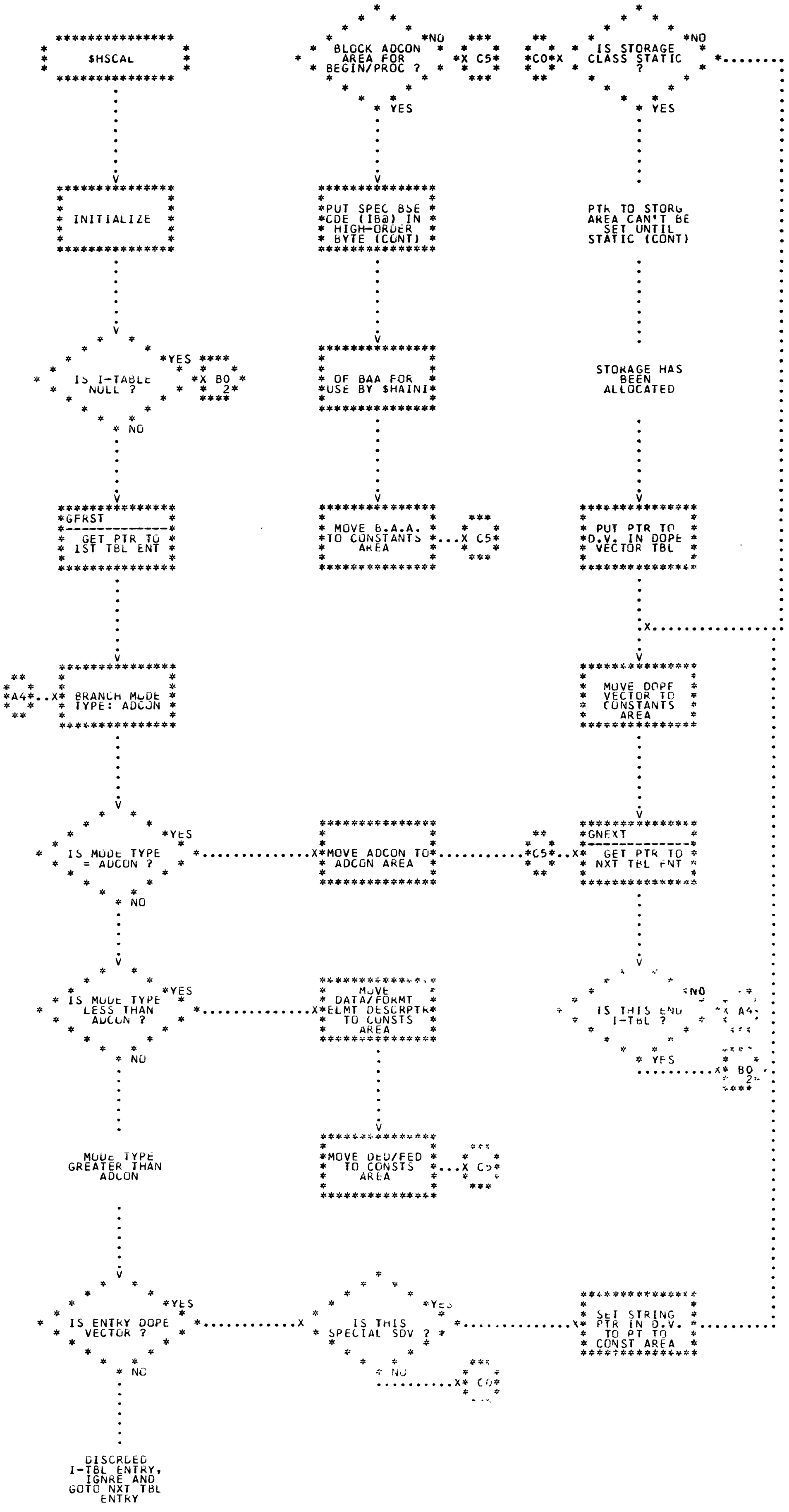


Chart 67. Static Constants-Adcon Loader (Page 1 of 2)

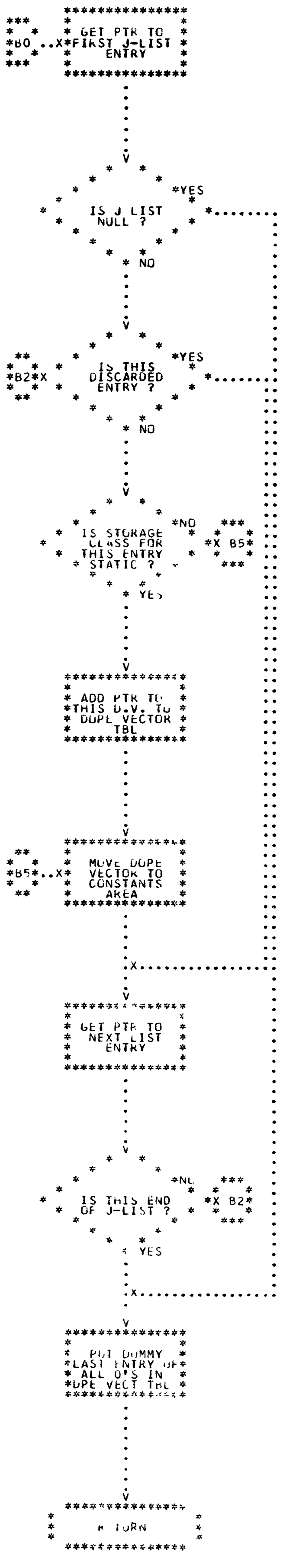


Chart 67. Static Constants-Adcon Loader (Page 2 of 2)

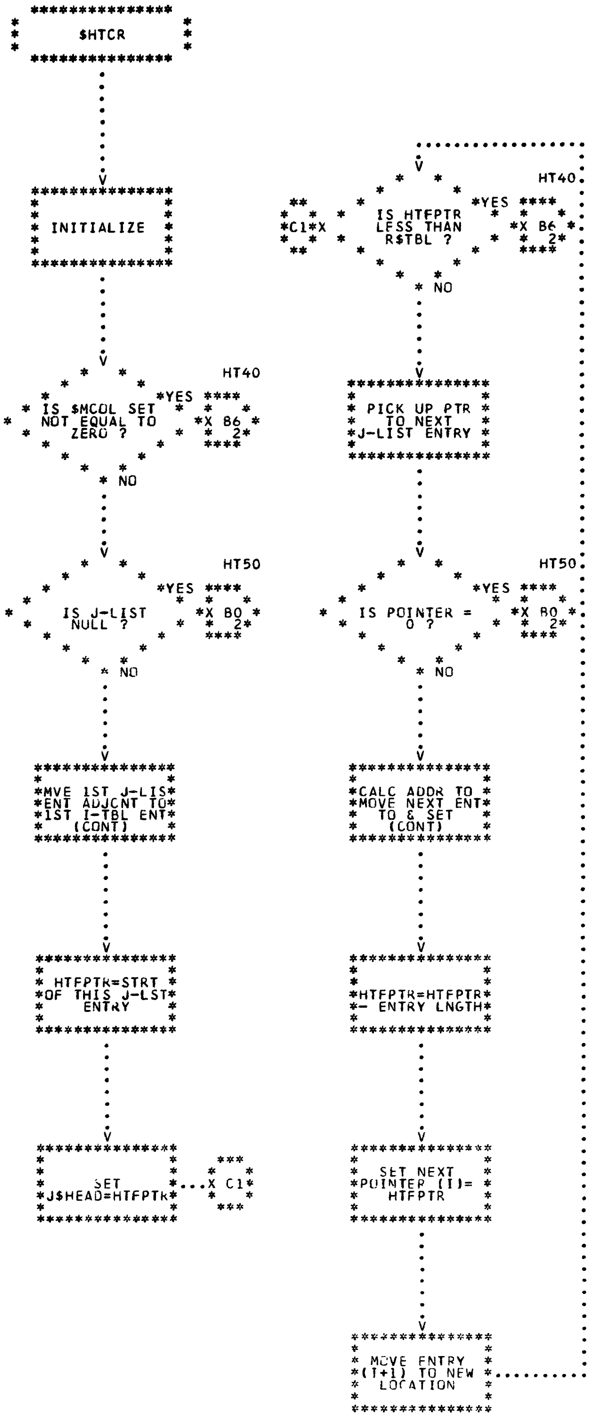


Chart 68. Table Collapse (Page 1 of 2)

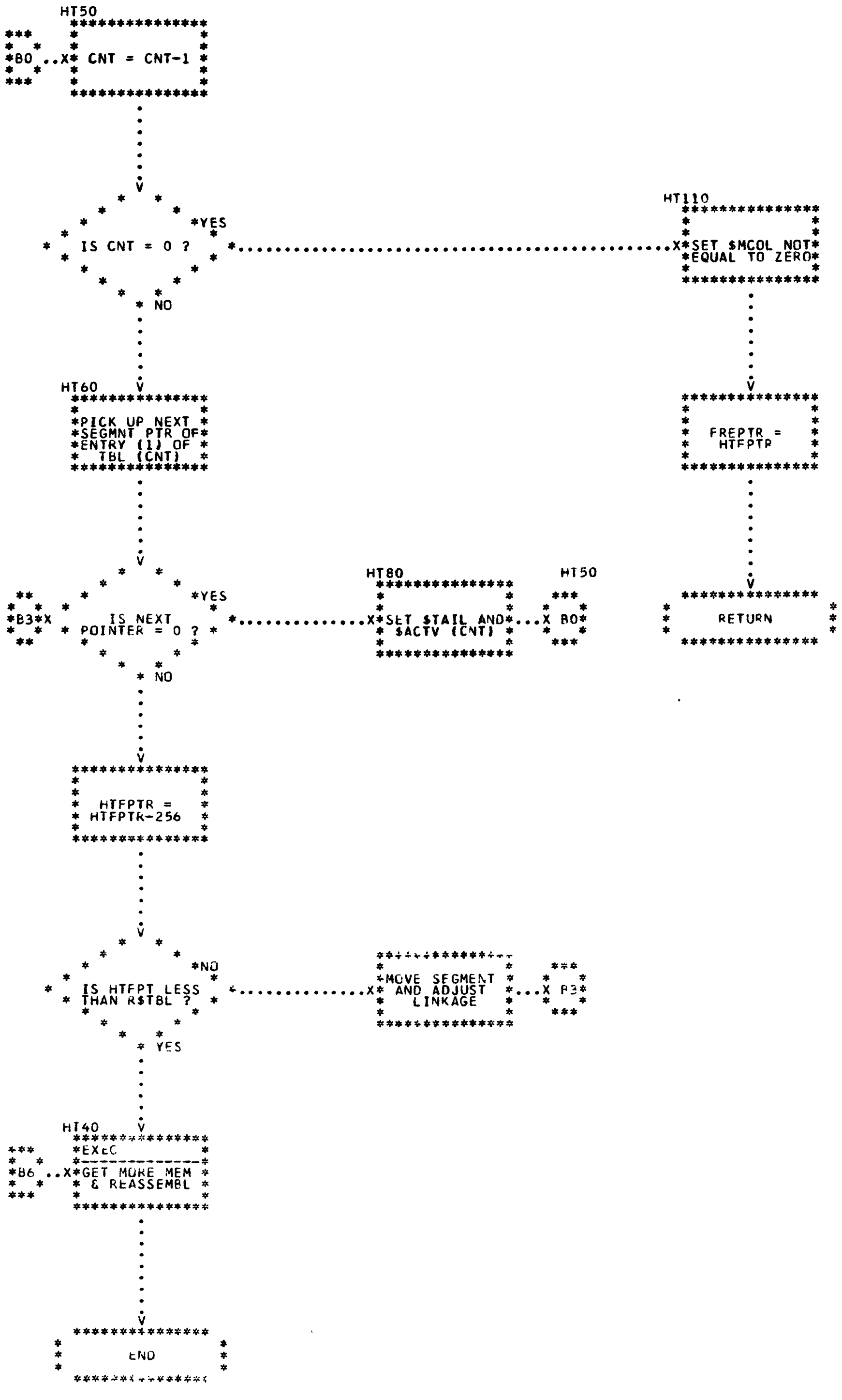


Chart 68. Table Collapse (Page 2 of 2)

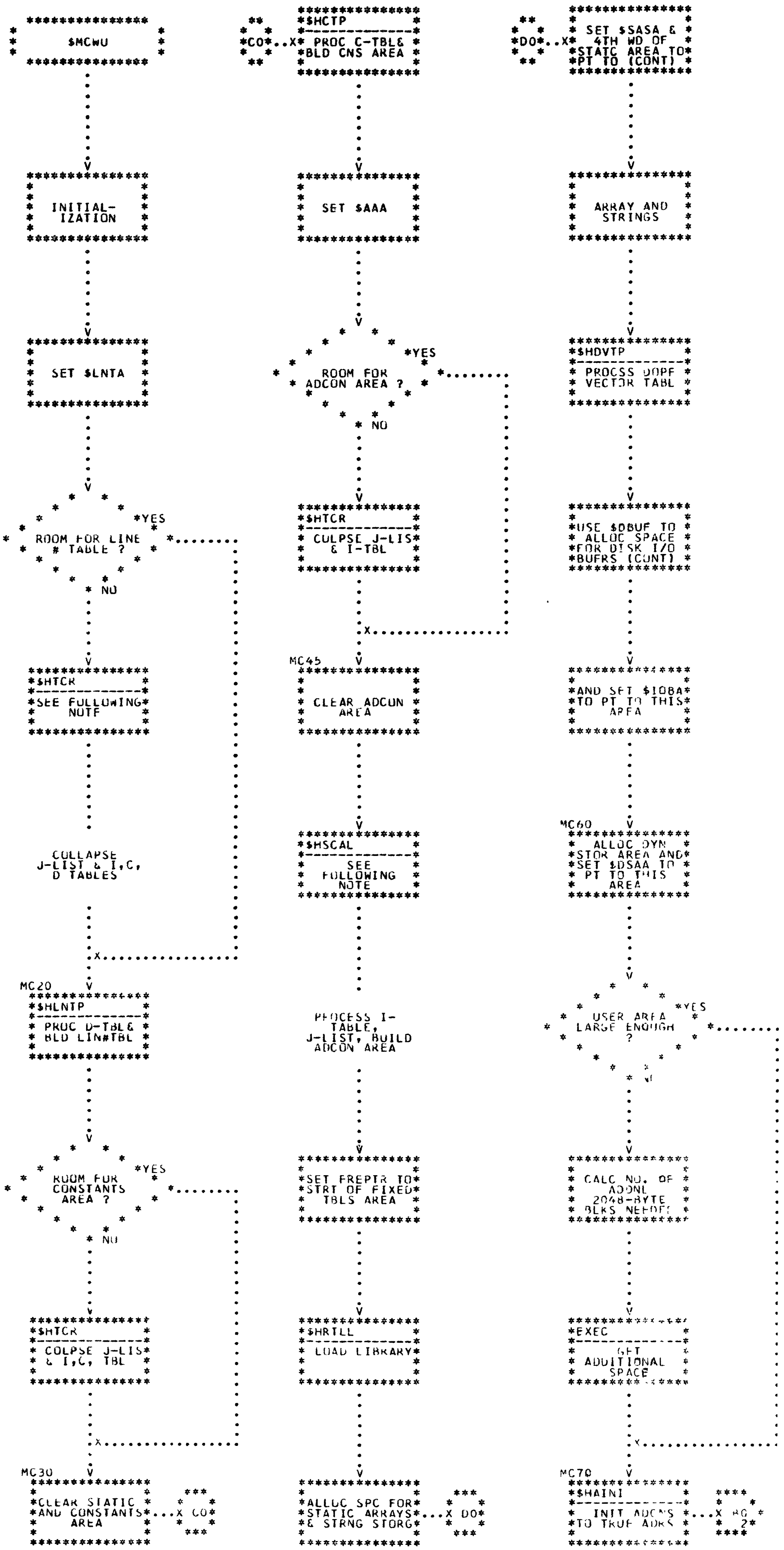


Chart 69. Compilation Wrap-Up Driver (Page 1 of 2)

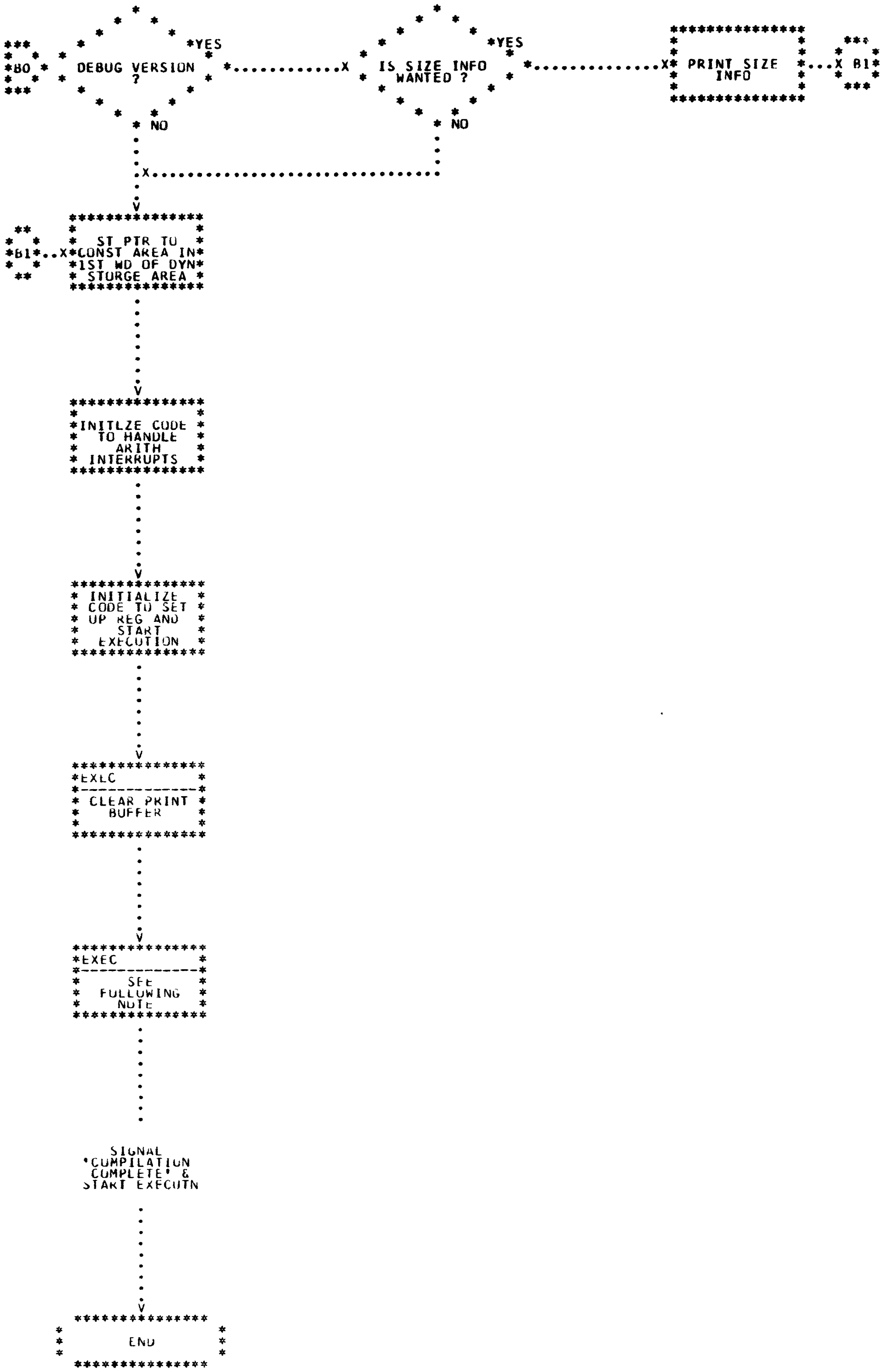


Chart 69. Compilation Wrap-Up Driver (Page 2 of 2)

PART 9 - SUPPORT

CALL/360-OS PL/I routines which perform support functions for other routines are described in this subsection. The routines are discussed in alphabetic order according to their mnemonics, as shown below. Detailed logic diagrams for the routines appear at the end of the subsection.

Array Expression Error ($AREXP)
Compiler Error ($CERR)
Output Director ($GPUT)
Get Next Triad Entry ($GTRIAD)
Constant Processor ($NCONS, $NCON)
Constant Conversion ($NCVT)
Library Search ($NLSIB)
Segment Management ($WBACK, $WSTEP, $WEXP, $WCTCT)
Error Message Editor ($XERR)

TITLE: ARRAY EXPRESSION ERROR ($AREXP)

Program Definition

Purpose and Usage

The Array Expression Error routine is used to generate the error message for an array expression in an illegal position. A value of zero is substituted for the illegal expression.

Description

An error message is output. The result fields from expression processing are set to a fixed-point zero.

Errors Detected

ARRAY EXPRESSION ILLEGAL. (23)

Local Variables

None

Program Interface

Entry Points

$AREXP. No parameters.

Exit Conditions

Normal exit.

Routines Called

$XERR	Error Message Editor

Global Variables

$NLTKN	Expression Processing Results
$NLOPN	Left Operand Area of Expression Processing

Logic Diagram

Chart 70 shows the detailed logic diagram for the Array Expression Error routine.

TITLE: COMPILER ERROR ($CERR)

Program Definition

Purpose and Usage

The Compiler Error routine fields all hardware interrupts caused by the compiler. It also accepts calls from points in the compiler where an unrecoverable error is detected.

Description

A message indicating the offset inside the compiler phase where the error was detected or generated is printed on the terminal. Compilation is then terminated.

The format of the issued message is shown below.

```
PLIiixxxxxx:  INVALID SOURCE PROGRAM--COMPILE TERMINATED
```

If ii = 00, $CERR was called by a compiler module. xxxxxx will be the hexadecimal offset from the beginning of the compiler to the beginning of the module that called $CERR.

If ii ≠ 00, $CERR was invoked because of a program interrupt, and ii is the program interrupt code. (See IBM System/360 Principles of Operation (GA22-6821) for the meaning of various program interrupt codes.) xxxxxx will be the offset from the beginning of the compiler to the point of interruption.

Errors Detected

```
(iixxxxxx) PROGRAM ERROR - COMPILE TERMINATED. (114)
```

Local Variables: None

Program Interface

Entry Points

$CERR. This entry is called by compiler modules which detect an error warranting cessation of compilation. Register C3 contains the address of the origin of the module that detected the error. The high-order byte of C3 is zero.

$CERR+4. When a program interrupt occurs, the Executive transfers to this entry via instructions beginning at ARINTRP in the communications area. The low-order three bytes of C3 contain the machine interrupt address. The high-order byte contains the program interrupt code.

Exit Conditions: SVC 0 to CALL/360-OS System.

Routines Called

$GPUT	Output Director
$SVC	SVC Director

Global Variables

$BASE	Base Address of Compiler

Logic Diagram

Chart 71 shows the detailed logic diagram for the Compiler Error routine.

TITLE: OUTPUT DIRECTOR ($GPUT)

Program Definition

Purpose and Usage

The Output Director places a 120-character line in the terminal I/O buffer. While doing this, it removes trailing blanks and checks for line width.

Description

A 120-character line is processed and placed in the terminal I/O buffer. If there is insufficient space in the buffer, it is emptied first. Trailing blanks are removed from the line. If the line exceeds the line width, it is broken into segments of the maximum length. Before return, the line is cleared to blanks.

Errors Detected

None

Local Variables

None

Program Interface

Entry Points

Normal linkage to $GPUT. Register P5 has address of line.

Exit Conditions

Normal exit. All registers restored. Line cleared.

Routines Called

$SVC	SVC Director

Global Variables

None

Logic Diagram

Chart 72 shows the detailed logic diagram for the Output Director routine.

TITLE: GET NEXT TRIAD ENTRY ($GTRIAD)

Program Definition

Purpose and Usage

The Get Next Triad Entry routine gets the next available entry in the triad table (Z table).

Description

The Generate Triad macro (GTRD) calls this routine to get the next available entry in the triad table. When the triad table exceeds a segment (256 bytes), a call is made to $WEXP to link another segment to the triad table.

Errors Detected

None

Local Variables

$XSAVE+4	Save area for register C2
W$CGTR	Save area for register C1

Program Interface

Entry Points

$GTRIAD

Calling Sequence

```
    L       C1, @GTRIAD
    BALR    C1,C1
```

Exit Conditions

P5 contains address of triad entry.

Routines Called

$WEXP	Segment Management

Global Variables

None

Logic Diagram

Chart 73 shows the detailed logic diagram for the Get Next Triad Entry routine.

TITLE: CONSTANT PROCESSOR ($NCONS, $NCON)

Program Definition

Purpose and Usage

The Constant Processor controls the conversion of source constants, immediate constants, DED constants, and other compiler-generated constants.

Description

The constant is converted to its binary representation if it is an arithmetic source constant. Alphameric constants longer than 16 bytes or containing primes or split over source lines are not converted until compilation wrap-up. Each such constant is given its own constant table (C table) entry. After any required conversion, the constant is searched in the constant table to determine if the constant has been previously entered. If found, that constant table entry is returned for the constant. If not found, a new entry is added to the constant table. Storage is allocated for the constant in static and constants storage. Alignment is dependent on length. A length of four is word-aligned, lengths of eight and 16 are doubleword-aligned, and all others are byte-aligned. The constant table is divided into four lists for searching: those of length four, those of length eight, those of length 16, and all others.

Errors Detected

None

Local Variables

None

Program Interface

Entry Points

$NCONS. When entered through $NCONS, register G0 contains the fullword value of the constant to be searched.

$NCON. When entered through $NCON, the value of $NXFLG indicates the type of call. If zero, register G7 contains the length of the constant (in bytes) and registers F1 and F2 contain the converted value. If one, register P5 contains the address of the operand area of a source constant. If two, the left and right operand areas contain the real and imaginary parts of a complex source constant.

Exit Conditions

Control returns to the calling routine immediately following the invoking call. Register G0 contains a constant table pointer token. The floating-point registers are destroyed if the constant required conversion.

Routines Called

$NCVT	Constant Conversion
$WEXP	Segment Management

Global Variables

$ASC	Offset to Next Available Byte in Static and Constants Area
$ERROR	Parameter List for Error Message Editor
$NC1W	Head of Chain of Constant Table 4-Byte Entries
$NC2W	Head of Chain of Constant Table 8-Byte Entries
$NC4W	Head of Chain of Constant Table 16-Byte Entries
$NDIG	Floating-Point Value of Digit of Source Constant
$NOTKN	Expression Processing Results
$NROPN	Right Operand Area for Expression Processing
$NXFLG	Communication Flag
$VLPAK	Doubleword-Aligned Work Area
A List	Dictionary Attribute List
C Table	Constant Table

Logic Diagram

Chart 74 shows the detailed logic diagram for the Constant Processor routine.

TITLE: CONSTANT CONVERSION ($NCVT)

Program Definition

Purpose and Usage

The Constant Conversion routine converts a source constant of any arithmetic type to any arithmetic target type.

Description

The dictionary attribute list entry of a source constant contains the type, scale, and precision of the constant as it appeared in the source program. This entry also contains the attributes required for the constant in the object program. The source constant is scanned and converted to an intermediate double-precision floating-point form. This value is then converted to the largest type with real mode.

Errors Detected

CONVERSION OF CONSTANT PRODUCES EXPONENT OUT OF RANGE. (98)
CONSTANT VALUE OR PRECISION TOO LARGE. (99)

Local Variables

None

Program Interface

Entry Points

$NCVT. The address of the dictionary attribute node of the constant to be converted is in register P4. Register P5 contains the address of the operand area which contains the sign to be applied to the constant.

Exit Conditions

Control returns to the calling routine immediately after the invoking call. Register G2 contains the length of the converted constant. If the length is four bytes, the converted constant is returned in register G0. If the length is eight bytes, the most significant half of the converted constant is returned in register G0 and the least significant half in register G1.

Routines Called

$XERR	Error Message Editor
$WEXP	Segment Management

Global Variables

$ERROR	Parameter List for Error Message Editor
$NDIG	Floating-Point Value of Digit of Source Constant
$NOPI	Field of $NO
$VLPAK	Doubleword-Aligned Work Area
A List	Dictionary Attribute List
Z Table	Triad Table

Logic Diagram

Chart 75 shows the detailed logic diagram for the Constant Conversion routine.

TITLE: LIBRARY SEARCH ($NLSIB)

Program Definition

Purpose and Usage

Given the number corresponding to a library entry name and the number (minus 1) of parameters used in the call to the entry, the Library Search routine returns a pointer to the adcon that is to provide cover for the entry.

Description

This routine returns (to the calling routine) a pointer to the adcon that will contain the entry point of the routine at runtime. For loading, library routines are of two types: fixed adcon location routines and variable adcon location routines.

For any fixed adcon routine, the adcon that will contain the location of the routine will always be at the same displacement in adcon storage. The adcon displacement for a variable adcon routine may vary for different programs.

Furthermore, all fixed adcon routines will store the same parameter table. Calls to library routines may require the passing of arguments. In such cases, the actual arguments or the locations of the arguments will be put into entries in the parameter table, and the location of the parameter table will be passed to the particular library routine.

For a variable adcon routine, $NLSIB allocates storage sufficient for a block adcon area (BAA) from the available adcon region. After the routine is loaded, this BAA will contain the location of the routine and an area reserved for passing arguments. Since this is a special use of a BAA, remaining fields of the BAA will not be used.

Note: Recognition of two types of library routines saves adcon area. The fixed adcon routines are most frequently used. A word of the adcon area is permanently reserved for each of these (regardless of whether a particular routine is loaded). A variable adcon routine requires much more adcon area. However, an area is reserved for a particular variable adcon routine only if the routine is actually loaded.

A check is made to determine if the adcon for the entry is in a fixed place in the adcon area. If it is not, the library load table entry corresponding to the entry name is checked to see if the required BAA has been assigned for the entry name. If the BAA has already been assigned, a pointer to the start of this BAA (with the library base code in the high-order byte) is returned to the calling routine. If the BAA has not been assigned, it is assigned; the pointer is placed in the library load table entry corresponding to the entry name, and the pointer (as described above) is returned to the calling routine.

If the adcon for the entry name is in a fixed place in the adcon area, a check is made to see if the present parameter table (pointed to by $PARAM) is large enough. If so, the library load table entry is set to point to the adcon and the adcon pointer (with the library base code in the high-order byte) is returned to the calling routine. If, on the other hand, the parameter table is not large enough, a new table is allocated before setting the library load table entry, etc., as described above.

Errors Detected

None

Local Variables

None

Entry Points

$NLSIB

Input Parameters

G0. One less than the number of words needed in the parameter table.
G2. Library load number of entry name.

Exit Conditions

Normal exit. Return to caller.

G7. Pointer to adcon for the entry name with the library base code in the high-order byte.

Routines Called

None

Global Variables

L Table	Library Load Table
$HADCN	Length of Adcon Area
$PSIZE	Length of Current Library Parameter Table
$PARAM	Address of Library Parameter Table
L$LIBX	Adcon Displacement

Logic Diagram

Chart 76 shows the detailed logic diagram for the Library Search routine.

TITLE: SEGMENT MANAGEMENT ($WBACK, $WCTCT, $WEXP, $WSTEP)

Program Definition

Purpose and Usage

The Segment Management routines provide the necessary bookkeeping services in support of compiler macros which manipulate expandable table segments.

Description

The inline code generated by the compiler macros which process expandable tables must, under certain conditions, step from one table segment to another. Since the amount of coding required for these operations is common to all tables, and is somewhat lengthy to generate inline, the necessary functions are isolated as independent subroutines as follows:

$WBACK	Proceeds from one segment of a table to the preceding segment; the old segment is retained as part of the table and is not released to the free pool.
$WCTCT	Releases a segment from the table to the free pool and adjusts pointers to the preceding segment.
$WEXP	Adds a new segment to the table and adjusts pointers to it.
$WSTEP	Proceeds from segment of a table to the succeeding segment, where the succeeding segment was already part of the table.

Errors Detected

None

Local Variables

$XSAVE	Save area for register C2

Program Interface - $WBACK

Entry Points

$WBACK

Calling Sequence

```
CALL   WBACK
DC     AL2((SEGSZE-8)/t@L*t@L+4)
DC     AL2(t$TAIL-$TAILS)
```

where t is the table-prefix letter for the table to be used. The first constant indicates the number of unused bytes at the end of a segment; segment size is not usually evenly divisible by item size. The second constant gives the relative distance of the pointers for the called table into the tables of segment control pointers.

Exit Conditions

Segment control pointers t$CURR and t$CURR+4 are updated to point to the beginning and end of the segment now active. Register G6 points to the beginning-of-segment control word of the new segment. The condition code is set to zero if an attempt was made to back off the first segment of the table; otherwise, it is nonzero.

Program Interface - $WCTCT

Entry Points

```
    $WCTCT
```

Calling Sequence

```
    L      P5,t$TAIL
    CALL   WCTCT
```

Exit Conditions

The released segment is chained to the free segment list (SEGLST). t$TAIL is updated to point to the current segment end. The condition code is set to zero if an attempt was made to release the first segment of a table; otherwise, it is nonzero.

Program Interface - $WEXP

Entry Points

```
    $WEXP
```

Calling Sequence

```
    L      P5,t$TAIL
    CALL   WEXP
```

Exit Conditions

Register G5 points to the top-of-segment control word of the newly added segment. The macro code will store this into t$CURR. Register G6 points to the first available data space in the new segment. The macro code will store this into t$ACTV. Register G7 and t$TAIL point to the end-of-segment control word of the newly added segment. FREPTR or SEGLST is updated, depending upon whether new space had to be acquired for the new segment or a previously freed segment could be used. If insufficient space remains in the user's area for the creation of a new segment, an SVC 6 is issued to restart the compilation with more space.

Program Interface - $WSTEP

Entry Points

```
    $WSTEP
```

Calling Sequence

```
    CALL   WSTEP
    DC     AL2(expression)
    DC     AL2(t$TAIL-$TAILS)
```

Expression has the value X'FF00' for the I table and C table. Items in these two tables are variable length, and computation of values for t$CURR must be handled separately for these cases. For all other tables, *expression* has the value (SEGSZE-8)/t@L*t@L and represents the number of unused bytes at the end of a segment. The second constant gives the relative distance of the pointers for the called table into the tables of segment control pointers.

Exit Conditions

Segment control pointers t$CURR and t$CURR+4 are updated to point to the beginning and end of the segment now active. Register G6 points to the beginning-of-segment control word of the new segment. The condition code is set to zero if an attempt was made to step beyond the end of the table; otherwise, it is nonzero.

Global Variables

SEGLST	Pointer to Beginning-of-Segment Control Word
FREPTR	Pointer to First Available Word in Compiler's Variable Data Area
$SEVCT	Total Number of Error Messages Produced
$ACODE	Pointer to Next Available Byte in Object Code Area

Routines Called

None

Logic Diagrams

Charts 77 through 80 show the detailed logic diagrams for the Segment Management routines.

TITLE: ERROR MESSAGE EDITOR ($XERR)

Program Definition

Purpose and Usage

The Error Message Editor constructs, parameterizes, and prints a diagnostic error message in response to encoded calls from other compiler routines.

Description

Upon detection of error conditions, compiler routines use the Error Interface macro (GENER) to call the Error Message Editor. (See Appendix C.) The calling sequence supplies information which allows the editor to obtain the source line and column number at which the error was detected and to insert variables into the main text of the message.

The GENER macro sets pointers and strings into a communications area, $ERROR, as follows:

$ERROR	Pointer to Error Token
+4	Parameter-1 Pointer
+8	Parameter-2 Pointer
+12	Parameter-3 Pointer
+16	Character String (8 bytes)
+20	

The pointer to the error token is always present. By analyzing the segment of the token table (T table) which contains the token pointed to, the Error Message Editor is able to obtain the number of the column in the source statement which contains the first character of the syntactic unit at which the error was found. The Error Message Editor then scans backwards through the token table until a new-line token is found; this token enables the source-statement line number to be obtained.

The three parameter pointers are optional. If no character string is given, the first parameter with a zero pointer ends the parameter list. If a character string is present, it becomes the value associated with the first parameter with a zero pointer. Parameter pointers point either to tokens or to dictionary name list (N list) nodes. The tokens represent delimiters, identifiers, or constants. In each case, the Error Message Editor ultimately determines the length and location of an appropriate character string for each parameter; these strings are then inserted into the main text of the message when called for. Each message is identified by a message number obtained from the calling sequence.

To conserve as much space as possible, the texts of the error messages are encoded. The first byte of a message text contains the severity level code for that message. Thereafter, the message format is determined by control bytes which the Error Message Editor scans and interprets as follows:

0001nnnnnnnnnnnn

Immediate Data: Insert the following n+1 bytes of text into the message.

0011pppppppppppp

Remote Data: Insert into the message the data found at displacement p in the text table. (Location p must be a control byte for immediate data.)

0100pppppppppppp

Transfer: Continue the scan of the text at the control byte found at displacement p in the text table.

0111nnnn

Parameter: Insert the character string associated with parameter n into the message.

00000000

End of Message: Stop scanning the text and print the completed message.

To further conserve space, the messages are constructed from a standardized set of words and phrases, which minimizes the amount of immediate data required in the encoded texts.

The Error Message Editor also maintains a variable, $SEVCO, which is the highest severity level encountered during compilation. This code is used to suspend execution when fatal errors have been detected. If insufficient space was allocated to the compiler, preventing completion of compilation, the compiler requests additional space from the CALL/360-OS Executive and begins compilation of the source program again. Under these circumstances, any error messages printed during the first attempt will be printed again on the retry. To avoid user confusion that would result from this situation, the Error Message Editor maintains a message count, $SEVCT. When a retry is attempted, this count is moved to NOERMSG. The Error Message Editor will not print a message until $SEVCT (the current compilation count) exceeds NOERMSG (previous compilation count). This insures that an error message appears only once, regardless of the number of recompilations needed.

Errors Detected

None

Local Variables

XDP1L	Table of parameter pointers and lengths
XDWORK	Miscellaneous work area

Program Interface

Entry Points

$XERR

Calling Sequence

```
L       C1,@HERR
BALR    C1,C1
DC      AL2(message-number)
```

$ERROR area must be preset with pointers and strings by use of the GENER macro.

Exit Conditions

Control returns to the caller immediately following the invoking call (plus 2 bytes).

Routines Called

$GPUT	Output Director

Global Variables

$SEVCO	Highest Severity Code
$SEVCT	Total Number of Error Messages Produced
NOERMSG	Number of Error Messages (Communications Area)
$VLINE	Print-Line Work Area
$VLPAK	Doubleword-Aligned Work Area
$ERROR	Parameter List for Error Message Editor
$LLINE	Last-Line-Number Pointer
FREPTR	Pointer to First Available Word in Compiler's Variable Data Area
T Table	Token Table
A List	Dictionary Attribute List
N List	Dictionary Name List

Logic Diagram

Chart 81 shows the detailed logic diagram for the Error Message Editor routine.

PART 9 LOGIC DIAGRAMS

The detailed logic diagrams for the support routines follow.

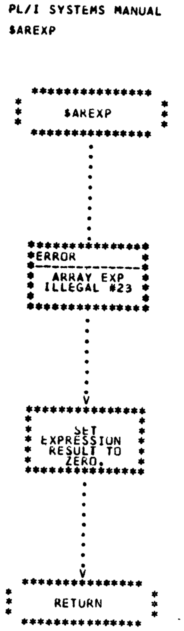


Chart 70. Array Expression Error

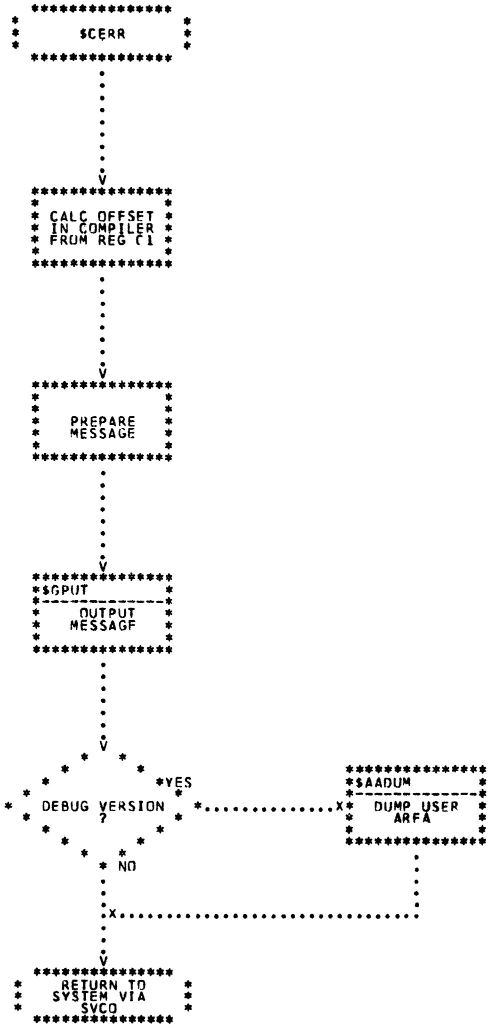


Chart 71. Compiler Error

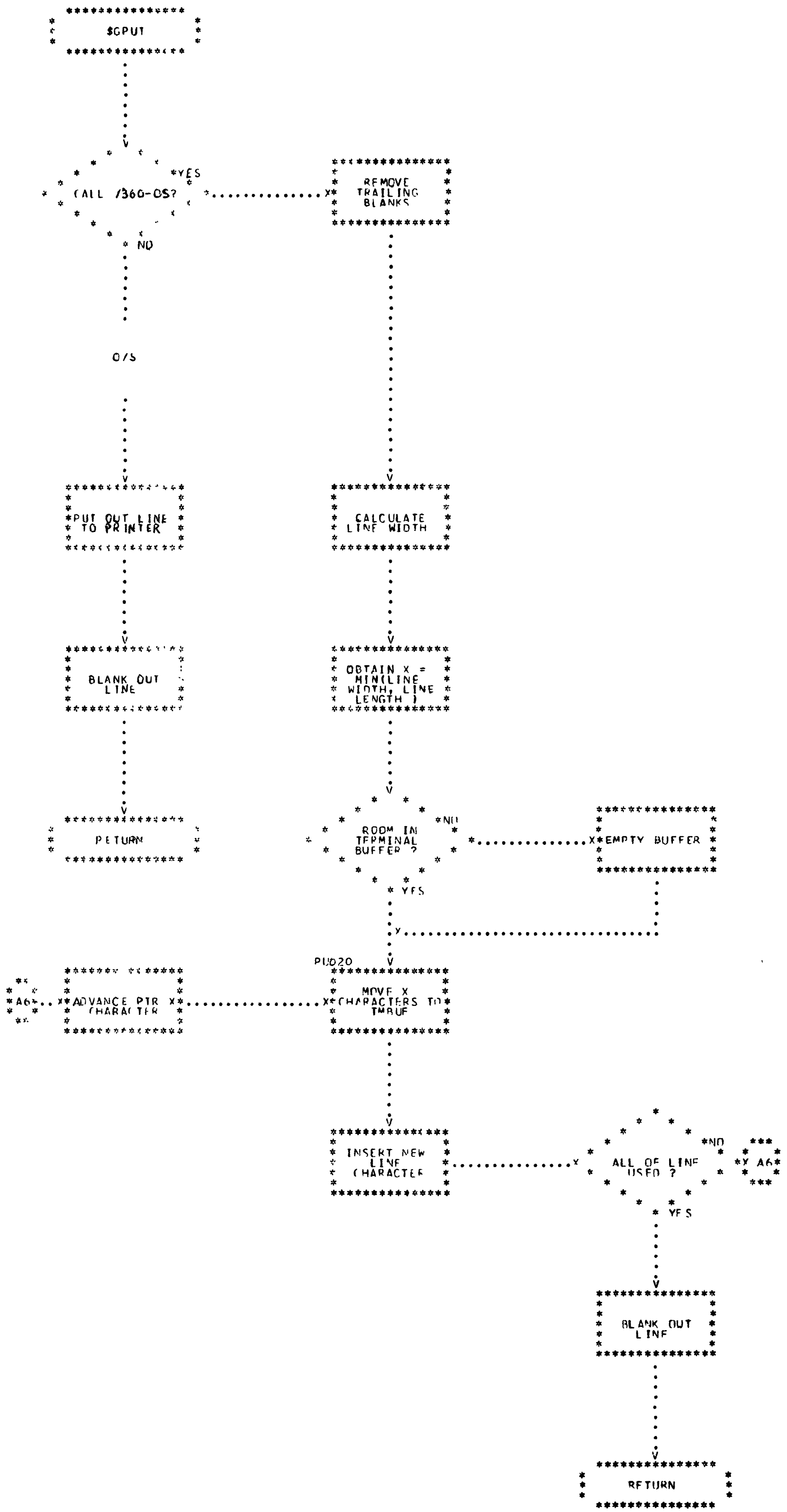


Chart 72. Output Director

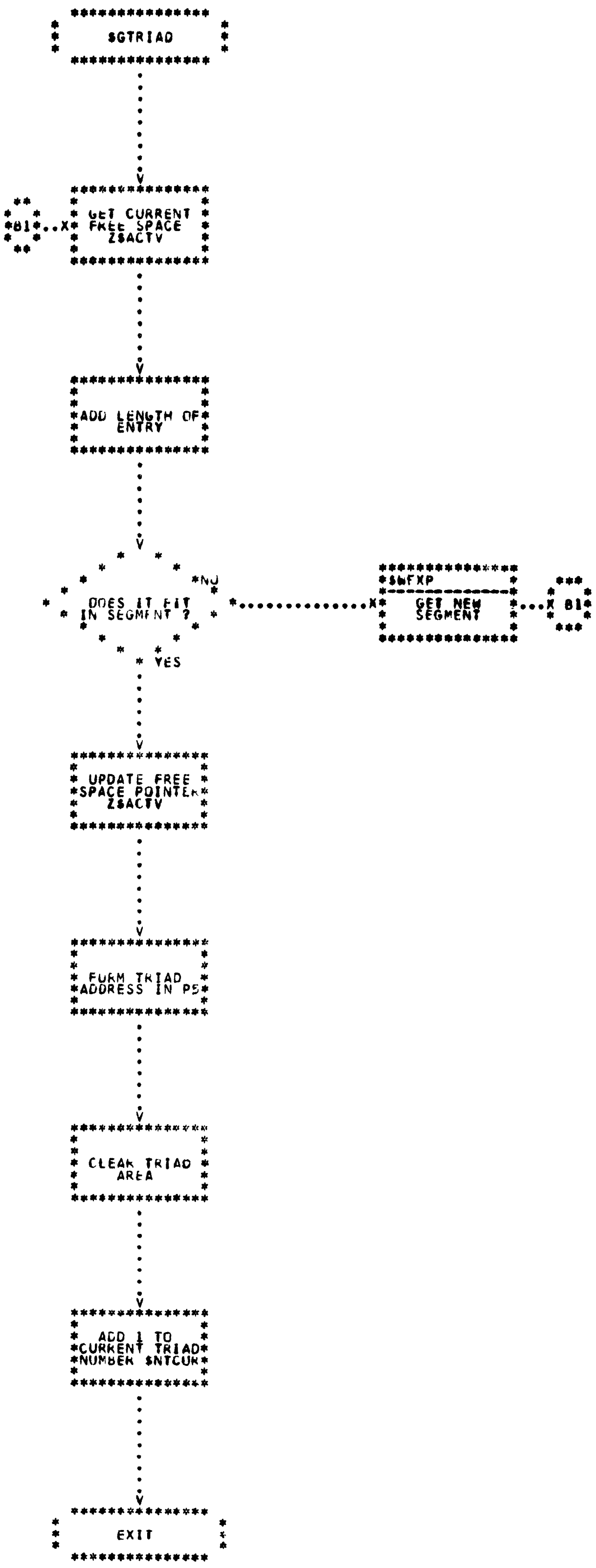


Chart 73. Get Next Triad Entry

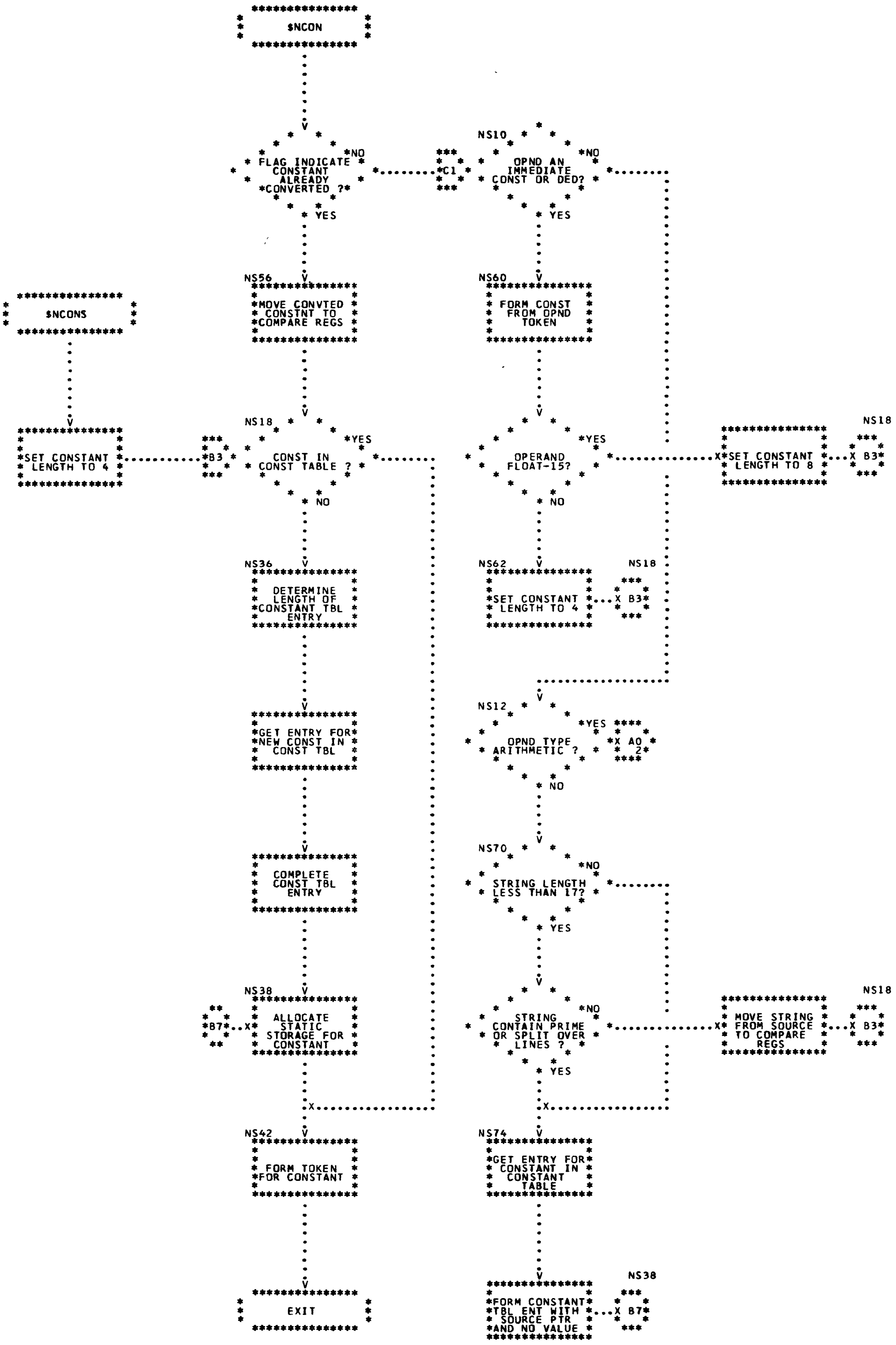


Chart 74. Constant Processor (Page 1 of 2)

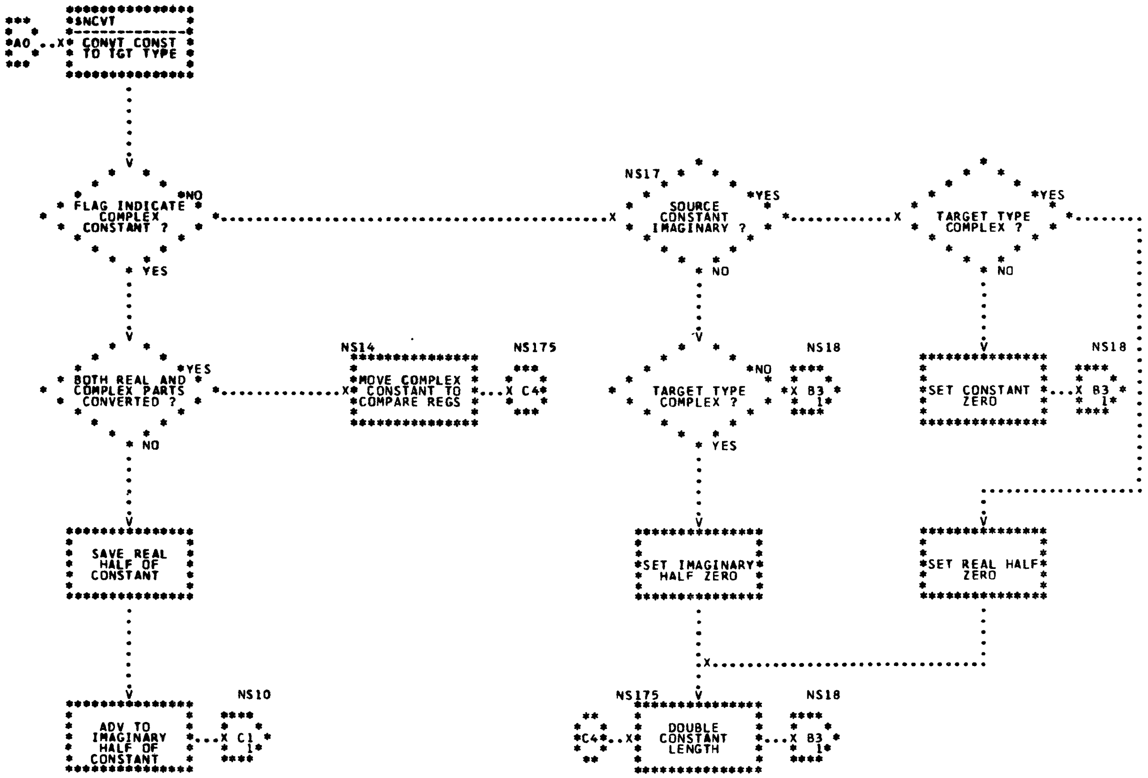


Chart 74. Constant Processor (Page 2 of 2)

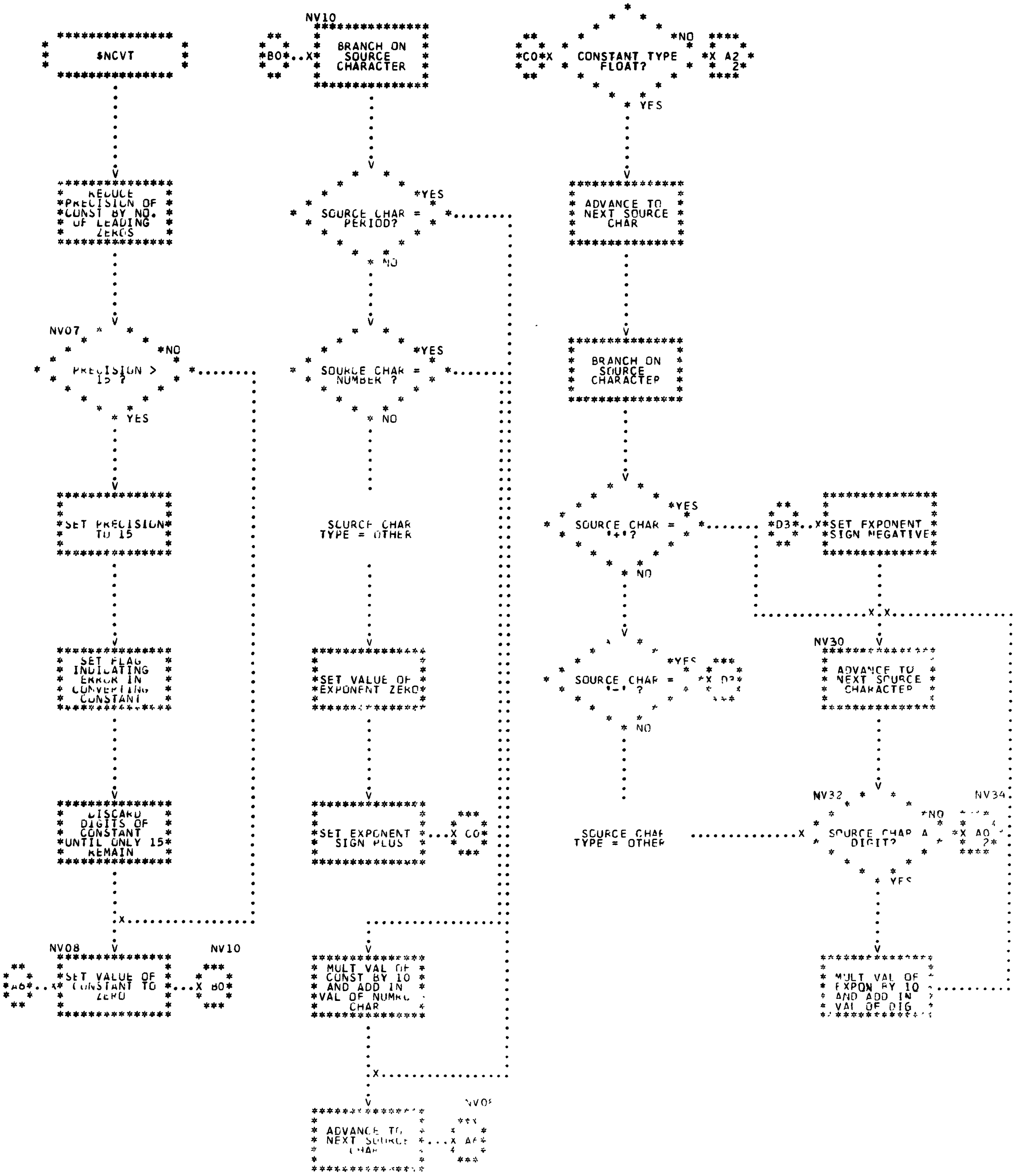


Chart 75. Constant Conversion (Page 1 of 2)

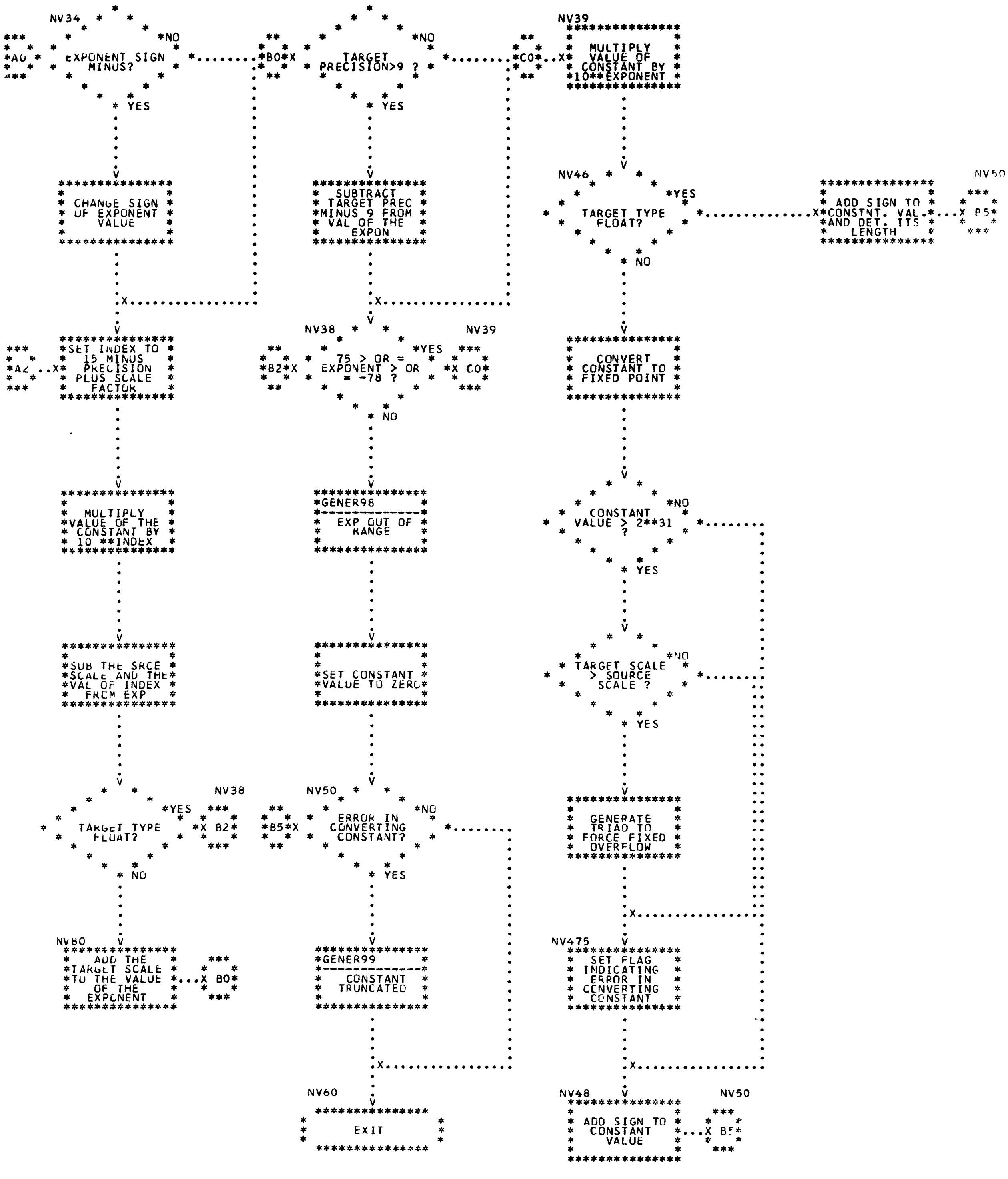


Chart 75. Constant Conversion (Page 2 of 2)

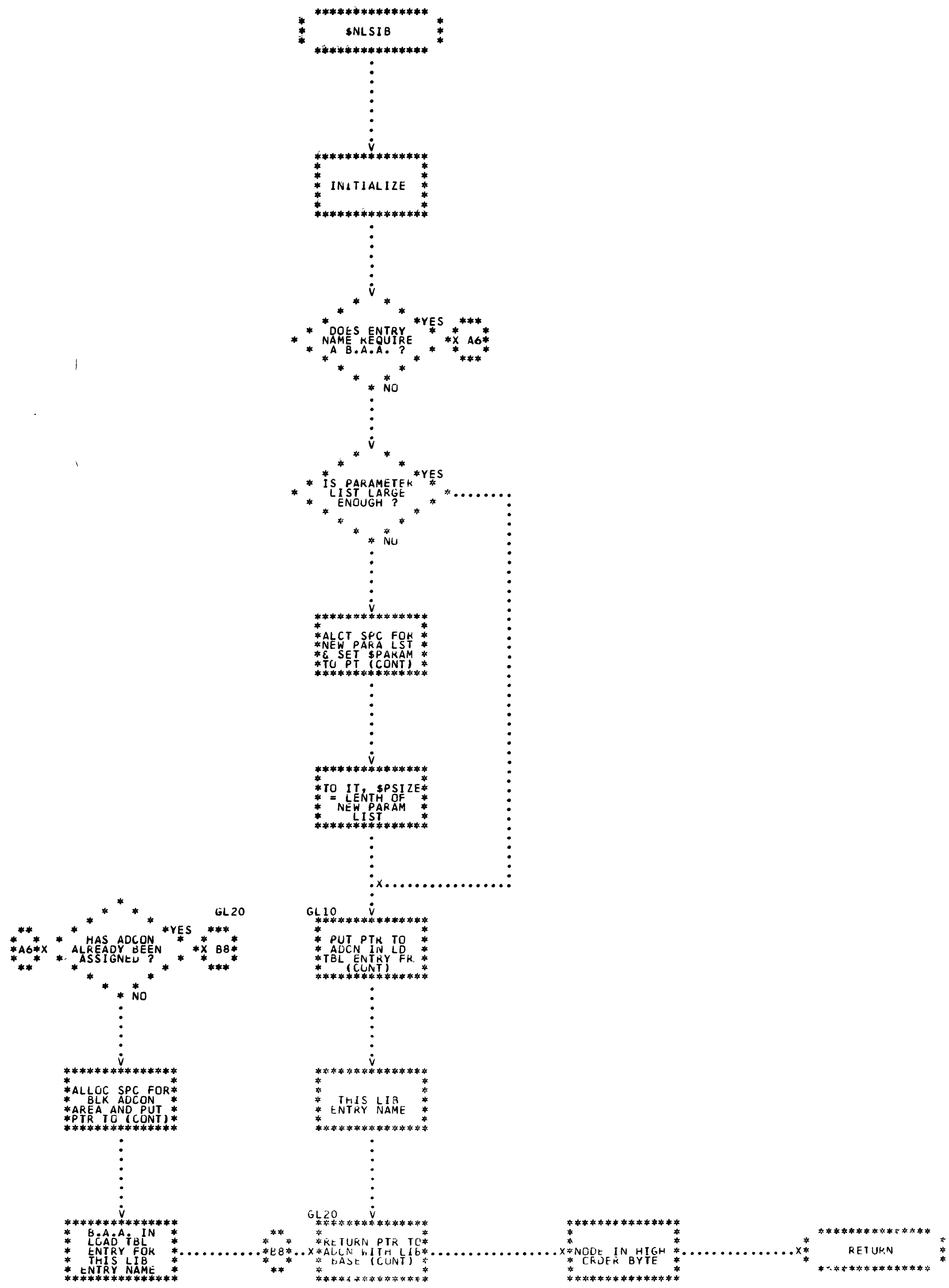


Chart 76. Library Search

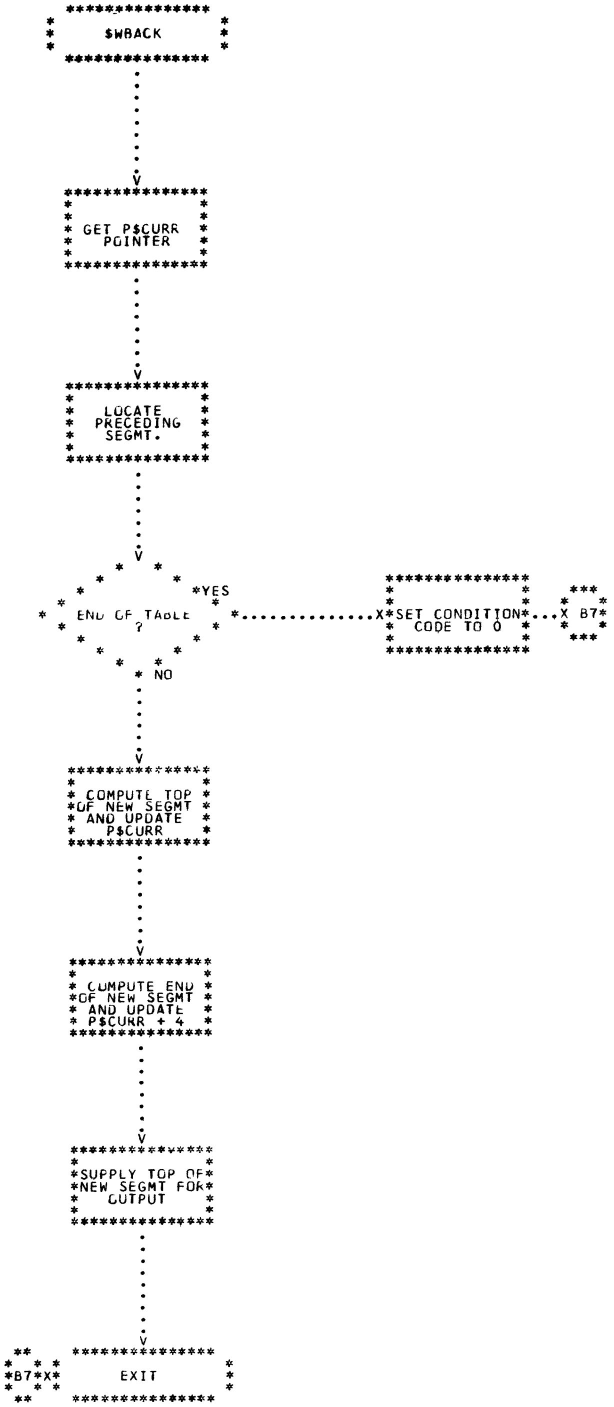


Chart 77. Segment Management ($WBACK)

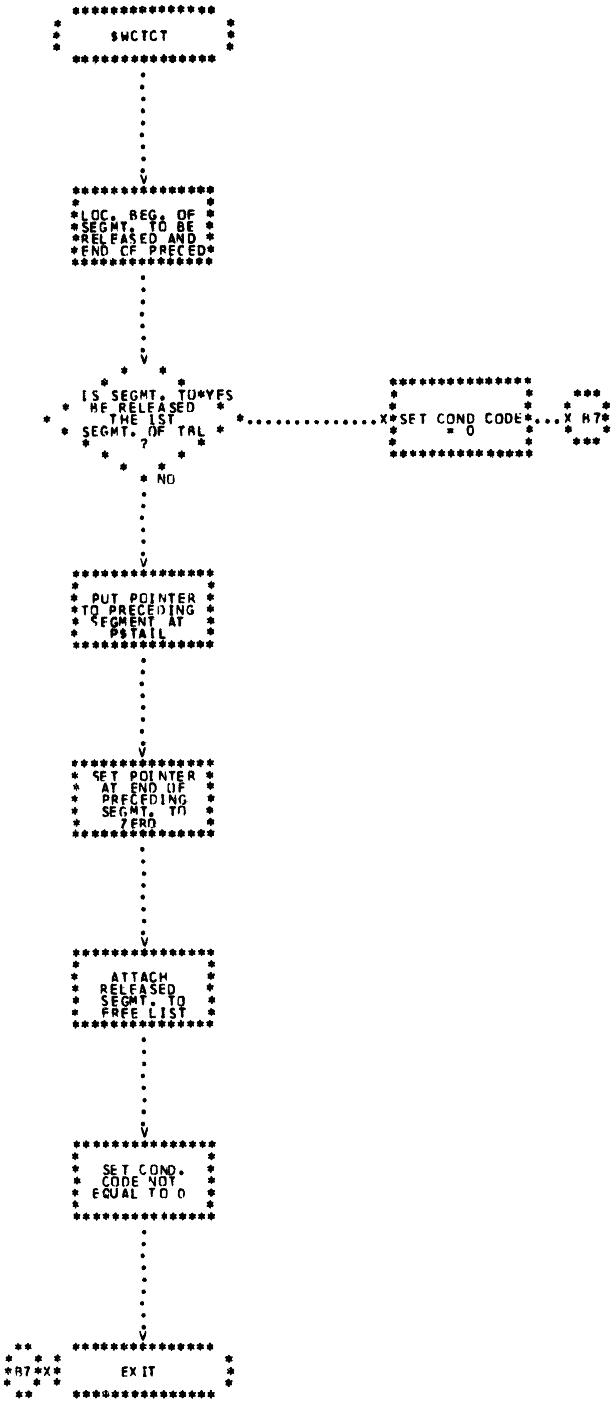


Chart 78. Segment Management ($WCTCT)

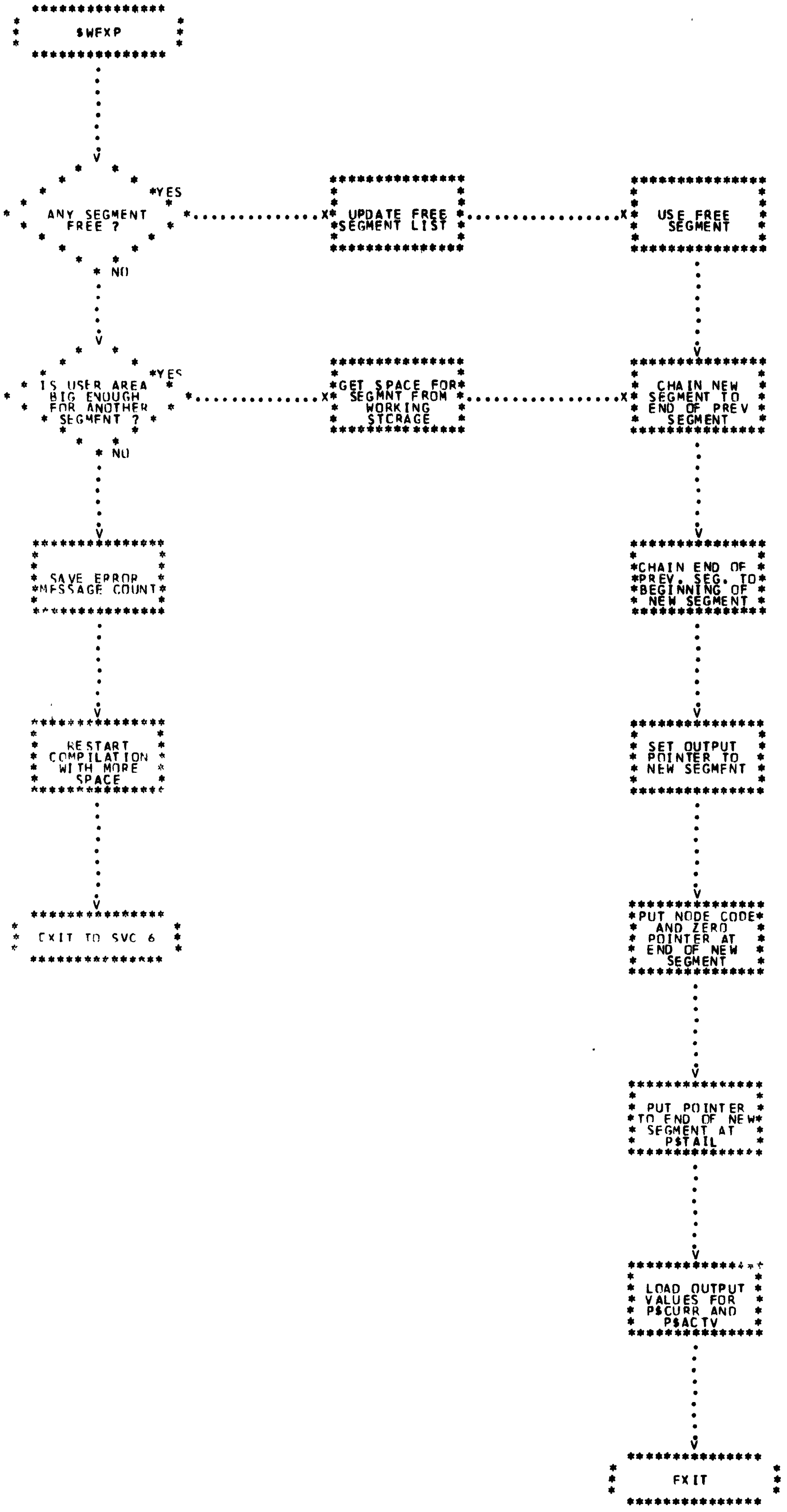


Chart 79. Segment Management ($WEXP)

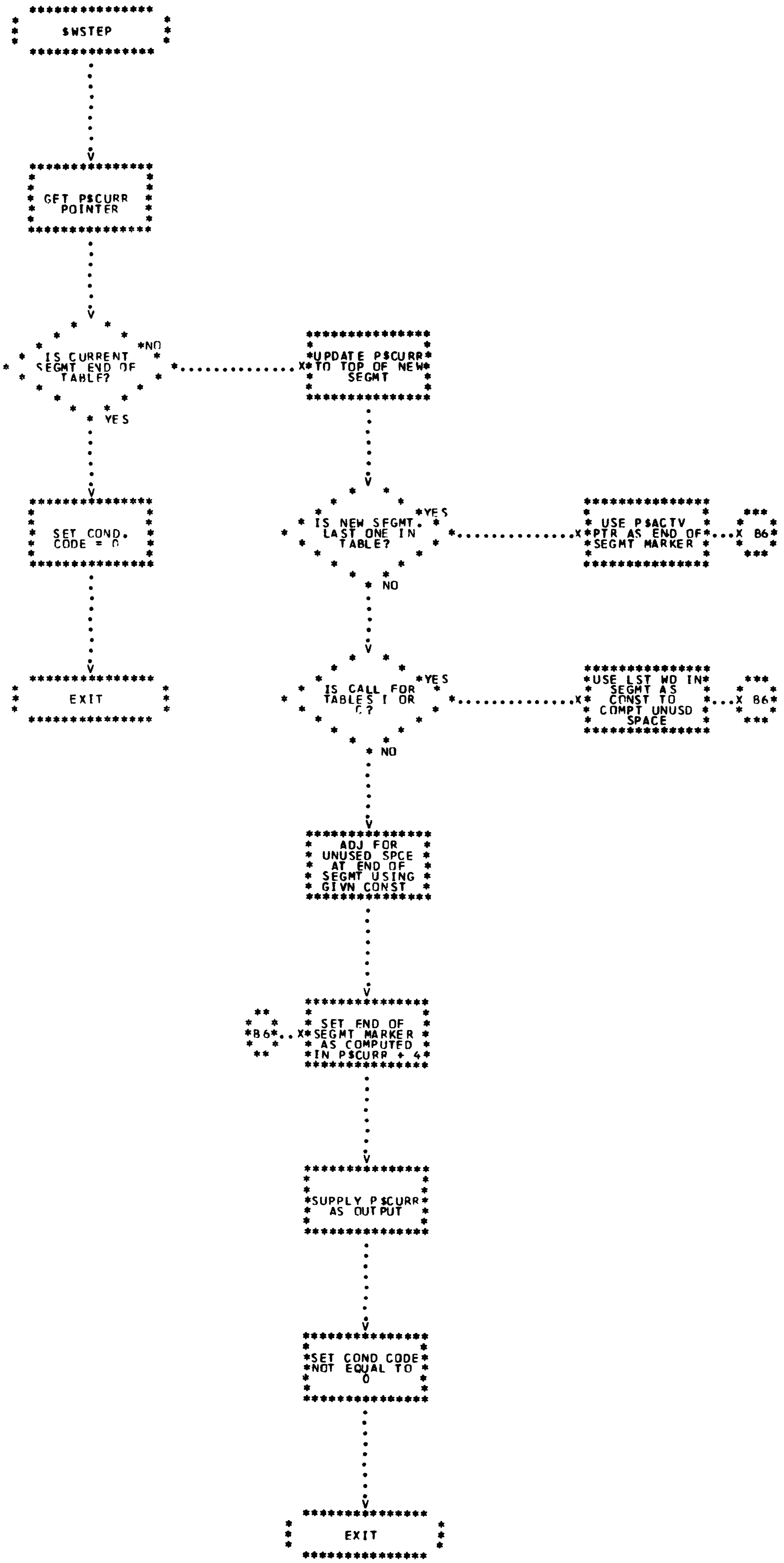


Chart 80. Segment Management ($WSTEP)

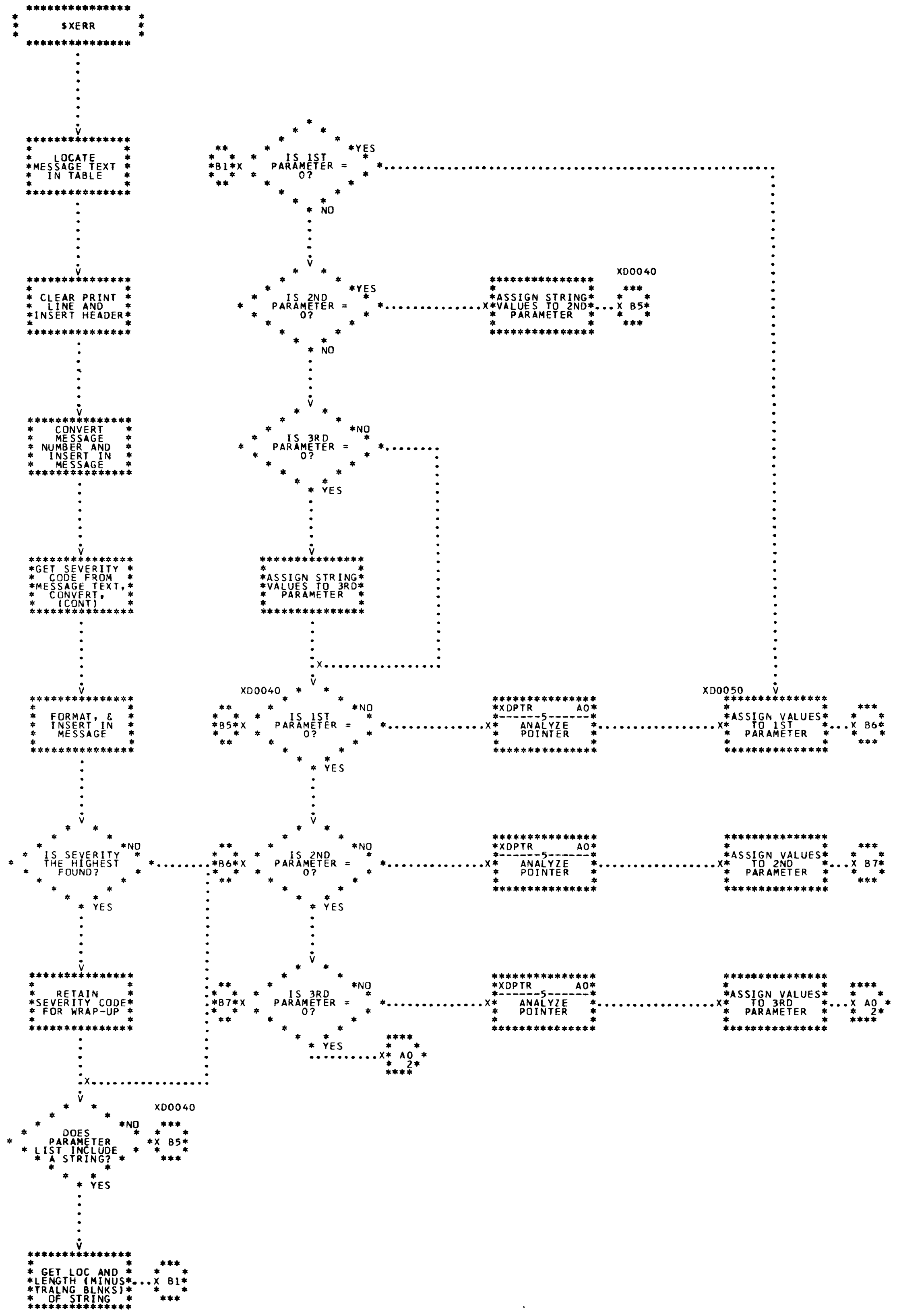
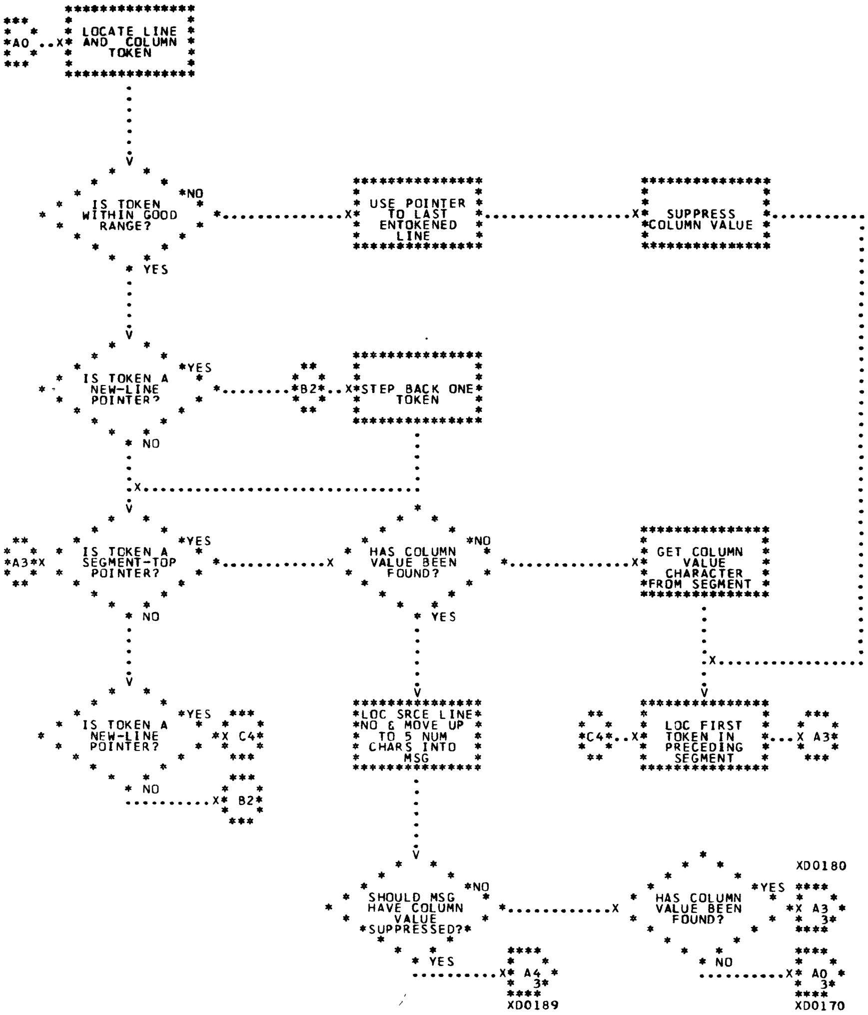


Chart 81. Error Message Editor (Page 1 of 5)


A0
LOCATE LINE AND COLUMN TOKEN
IS TOKEN WITHIN GOOD RANGE?
NO
YES
USE POINTER TO LAST ENTOKENED LINE
SUPPRESS COLUMN VALUE
IS TOKEN A NEW-LINE POINTER?
YES
NO
B2
STEP BACK ONE TOKEN
A3
IS TOKEN A SEGMENT-TOP POINTER?
YES
NO
HAS COLUMN VALUE BEEN FOUND?
NO
YES
GET COLUMN VALUE CHARACTER FROM SEGMENT
IS TOKEN A NEW-LINE POINTER?
YES
NO
C4
B2
LOC SRCE LINE NO & MOVE UP TO 5 NUM CHARS INTO MSG
C4
LOC FIRST TOKEN IN PRECEDING SEGMENT
A3
SHOULD MSG HAVE COLUMN VALUE SUPPRESSED?
NO
YES
A4 3
XD0189
HAS COLUMN VALUE BEEN FOUND?
YES
NO
XD0180
A3 3
A0 3
XD0170

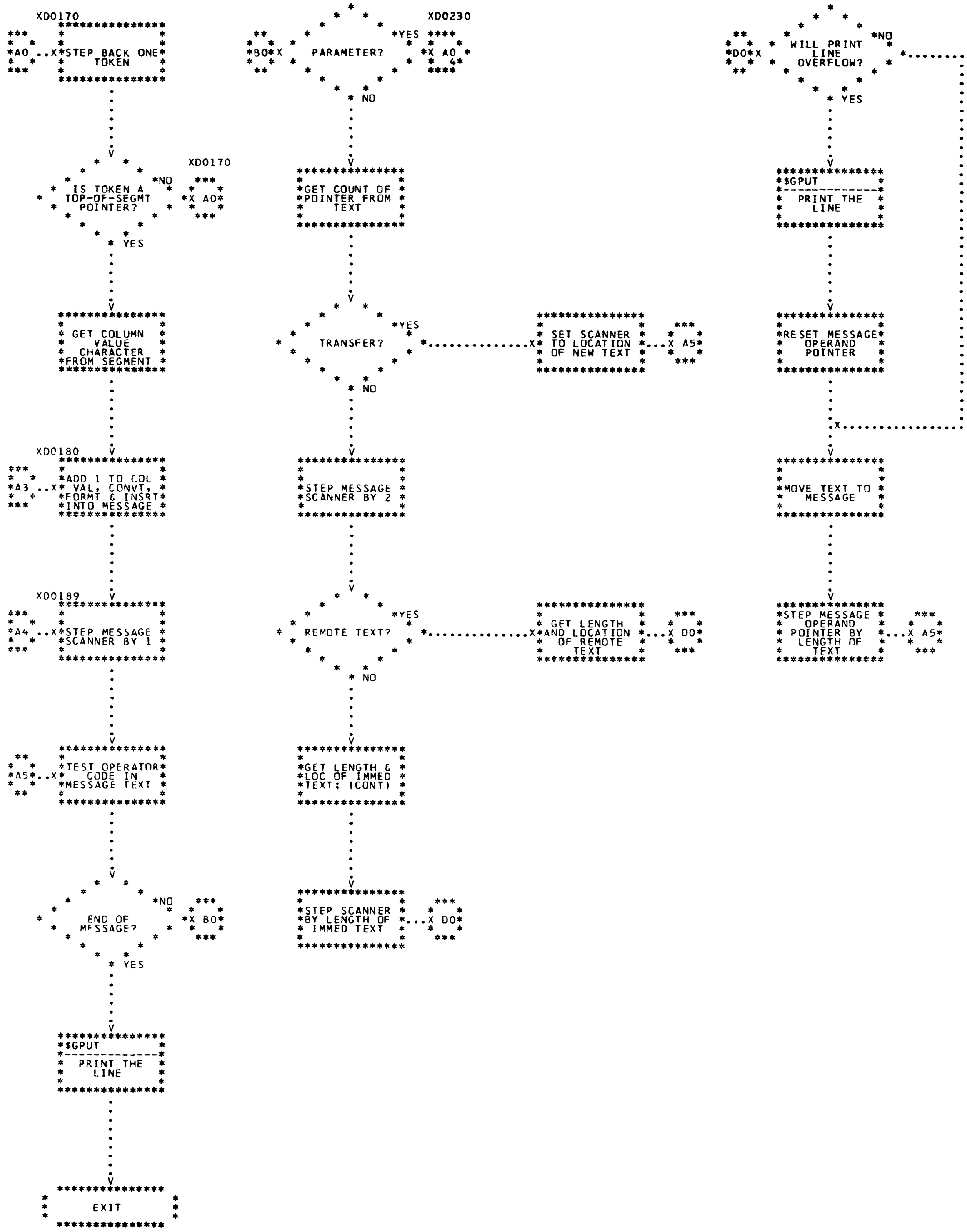


Chart 81. Error Message Editor (Page 3 of 5)

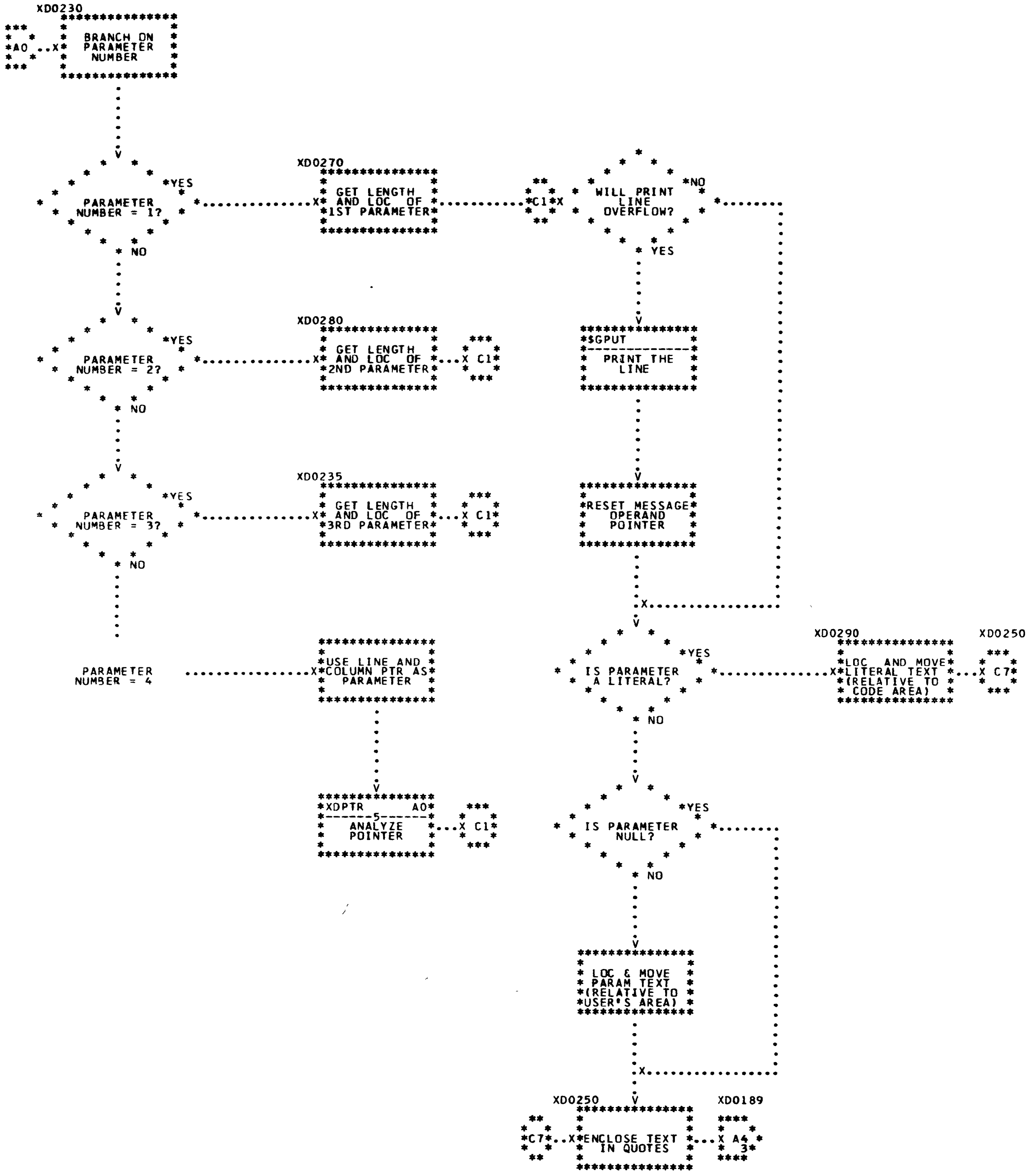


Chart 81. Error Message Editor (Page 4 of 5)

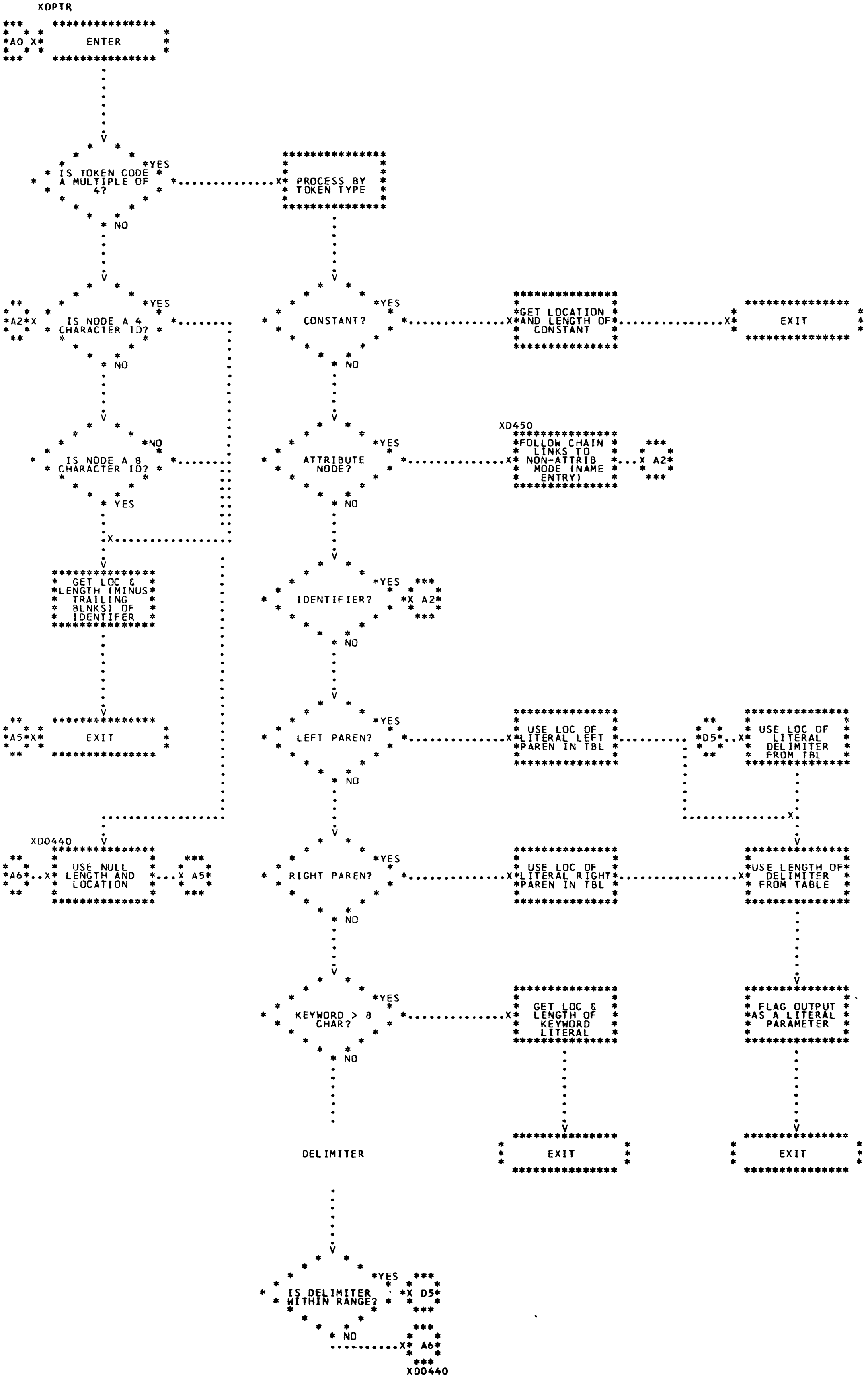


Chart 81. Error Message Editor (Page 5 of 5)

PART 10 - EXECUTIVE INTERFACE

TITLE: SVC DIRECTOR ($SVC)

Program Definition

Purpose and Usage

The SVC Director handles all SVC interfaces with the CALL/360-OS system for the CALL/360-OS PL/I compiler.

Description

The SVC code is picked up from the halfword following the BALR to this routine. The proper SVC is executed. Return is to the location immediately following the halfword SVC code.

Errors Detected

None

Local Variables

None

Program Interface

Entry Points

$SVC. Halfword SVC code immediately after BALR to this routine.

Calling Sequence

```
    L       C2,@SVC
    BALR    C1,C2
    DC      AL2(svc-code-number)
```

Exit Conditions

Return is to two bytes after BALR.

Routines Called

None

Global Variables

SVCINST	SVC Instruction to Transfer to Executive (Communications Area)

Logic Diagram

Chart 82 shows the detailed logic diagram for the SVC Director routine.

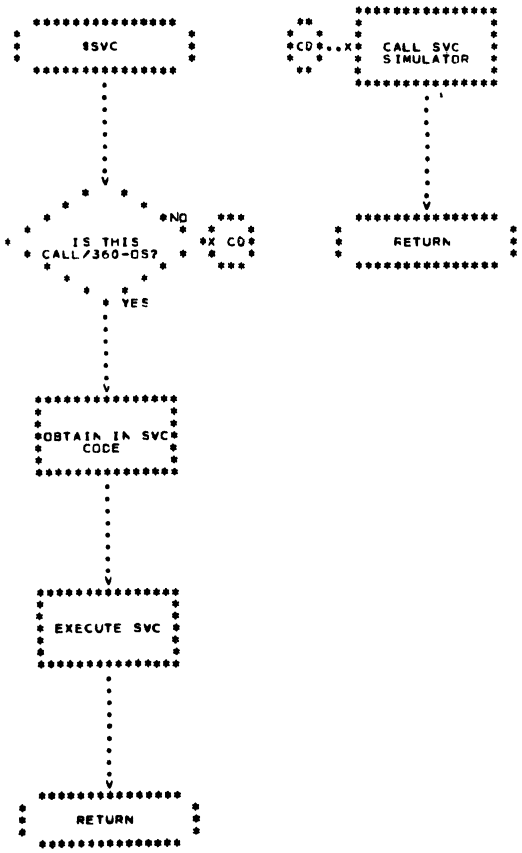


Chart 82. SVC Director

READER'S COMMENT FORM

CALL/360-OS PL/I
Systems Manual

GY20-0567-1

Please comment on the usefulness and readability of this publication, suggest additions and deletions, and list specific errors and omissions (give page numbers). All comments and suggestions become the property of IBM. If you wish a reply, be sure to include your name and address.

COMMENTS

fold

fold

fold

fold

- Thank you for your cooperation. No postage necessary if mailed in the U.S.A. FOLD ON TWO LINES, STAPLE AND MAIL.

YOUR COMMENTS PLEASE...

Your comments on the other side of this form will help us improve future editions of this publication. Each reply will be carefully reviewed by the persons responsible for writing and publishing this material.

Please note that requests for copies of publications and for assistance in utilizing your IBM system should be directed to your IBM representative or the IBM branch office serving your locality.

fold fold

FIRST CLASS
PERMIT NO. 1359
WHITE PLAINS, N. Y.

BUSINESS REPLY MAIL

NO POSTAGE NECESSARY IF MAILED IN THE UNITED STATES

POSTAGE WILL BE PAID BY...

IBM Corporation
112 East Post Road
White Plains, N. Y. 10601

Attention: Technical Publications

fold fold

IBM®

International Business Machines Corporation
Data Processing Division
112 East Post Road, White Plains, N.Y. 10601
[USA Only]

IBM World Trade Corporation
821 United Nations Plaza, New York, New York 10017
[International]

GY20-0567-1

CALL/360-OS PL/I SM Vol 1 Printed in U.S.A. GY20-0567-1

International Business Machines Corporation
Data Processing Division
112 East Post Road, White Plains, New York 10601
(USA only)

IBM World Trade Corporation
821 United Nations Plaza, New York, New York 10017
(International)