An asynchronous circuit design language system

Gregory Martin Bednar

AN ASYNCHRONOUS CIRCUIT DESIGN LANGUAGE SYSTEM

by

GREGORY MARTIN BEDNAR, 1944-

A DISSERTATION

Presented to the Faculty of the Graduate School of the

UNIVERSITY OF MISSOURI-ROLLA

In Partial Fulfillment of the Requirements for the Degree

DOCTOR OF PHILOSOPHY

in

ELECTRICAL ENGINEERING

1972

[illegible]
Advisor

[illegible]

[illegible]

[illegible]

[illegible]

ABSTRACT

This paper presents a system for specifying the behavior of asynchronous sequential circuits. The system consists of a special purpose Asynchronous Circuit Design Language (ACDL), a translator and a flow table generation algorithm. The language includes many special features which permit quick and precise specification of terminal behavior. It is best suited for problems originating from a word description of the circuit's operation. The translator is written with the XPL Translator Writing System and is a syntax-directed compilation method. From the translated ACDL specifications, the flow table algorithm generates a primitive flow table which is the required input for the conventional synthesis procedures of asynchronous sequential circuits. A thorough description of the translator and flow table programs is given in the Appendices. In addition a number of example problems illustrating the use of ACDL are provided.

ACKNOWLEDGEMENT

I wish to express my sincere appreciation to Dr. James H. Tracey, not only for his assistance and supervision given me throughout this project, but also for his personal concern shown me throughout the pursuit of my degrees. In addition, I extend my thanks to Dr. Tracey for providing me with the opportunity to work with him under a National Science Foundation Research Grant.

I would also like to acknowledge Mr. Wayne Omohundro for his helpful suggestions during this research.

Most of all, I wish to thank my wife, Carol, for her hard work, understanding, and encouragement rendered during the writing of this paper and pursuit of my degrees.

TABLE OF CONTENTS

LIST OF ILLUSTRATIONS

Figures Page

LIST OF TABLES

I. INTRODUCTION

A number of synthesis procedures for asynchronous sequential circuits now exist and many of these have been programmed for computer application. Although considerable work has been directed toward improving the synthesis procedures, little has been done in interfacing the user to these procedures.

This dissertation presents an Asynchronous Circuit Design Language (ACDL) system which interfaces the user to the conventional synthesis procedures of asynchronous sequential circuits as illustrated in Figure 1. ACDL is a special purpose language used to describe the terminal behavior of asynchronous sequential circuits. This description is then translated, and interpreted into a primitive flow table which is the initial input requirement of the conventional synthesis procedures.

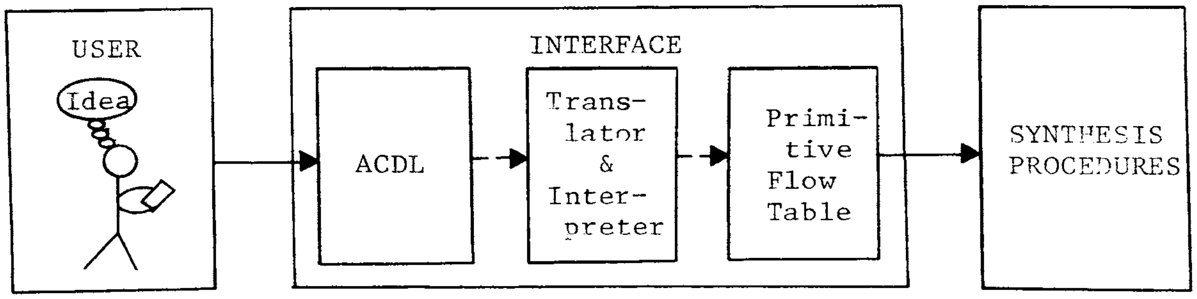


Figure 1. The Interfacing Characteristic of ACDL

Presently, a few procedures exist for specifying asynchronous sequential circuits when the terminal characteristics of the machine are easily expressed in input/output sequences [1,2]. However, only a small percentage of the designs are suitable for this type of terminal description. Hence typically, the designer will hand-construct a primitive flow table from a word statement or mental conception of the problem. This is not an easy or straightforward task since word statements and mental conceptions are informal descriptions of the problem.

After studying the specification problem for some time, it was decided that a language was needed that had the following three important characteristics:

1) it should permit ease of expression by coinciding with the designer's thinking process,
2) it should retain a formal meaning of the circuit description,
3) it should have a structure which would permit relatively easy automatic translation.

A review of all the well-known digital design languages was made to determine their applicability to the asynchronous circuit specification problem [3-9]. In general, it was found that these languages were intended for networks whose designs could best be described by functional operations and information transfers between basic hardware elements such as registers, switches, terminals, memory etc. None of the above languages were found to satisfy all of the desired characteristics mentioned for specifying the terminal (input/output) behavior of an asynchronous sequential circuit. Specific drawbacks of these languages included the inability to assign transition values to variables, the inability to make proper declarations such as "input constraints" and the inability to list multiple independent sequence paths without introducing additional control variables or cluttering the listing with many "go to" type statements.

ACDL was developed to meet the three important characteristics of the desired language and to overcome the drawbacks of the digital design languages noted above. This language provides a means to satisfy the specification problem by enabling the user to express his circuit characteristics formally, so the design can be carried out automatically.

II. ASYNCHRONOUS SEQUENTIAL CIRCUITS

Sequential circuits whose operation is not synchronized with clock pulses are called asynchronous sequential circuits. An important advantage of asynchronous sequential circuits is their ability to respond to input changes at basic device speed, rather than having to await the arrival of clocking signals. Also, many small circuits can be designed more easily and efficiently asychronously because it is not necessary to build a clock and synchronization circuitry. A further advantage of asynchronous design is seen in large circuits where signal lines are long and the skewing effect (difference in path propagation times) of the distributed clocking signals becomes a serious problem.

The operation of an asynchronous sequential circuit is often described by means of a flow table. As shown in Table I, it is a two-dimensional array consisting of next-state entries with its columns representing the input states and its rows representing the internal states of the circuit. The row in which the circuit is currently operating is often referred to as the present internal state or just the present state. For example, if the present state of the circuit described by Table I is 1 and then an input of I_2 is applied, the next state or state that the circuit will go to is 2.

TABLE I. FLOW TABLE

		Input states		
		I_1	I_2	I_3
	1	$\textcircled{1}$/0	2	3
Internal	2	1	$\textcircled{2}$/0	3
States	3	1	4	$\textcircled{3}$/0
	4	1	$\textcircled{4}$/1	3

If a next-state entry is found to be the same as the internal state representing that row, then the internal state is said to be stable with respect to that input column and is denoted by a circled next-state entry. Output states are usually only associated with stable next states as shown.

An asynchronous circuit is said to be operating in fundamental mode if the inputs are never changed unless the circuit is in a stable state. This paper only treats asynchronous circuits operating in fundamental mode. Further information on this class of circuits can be found in references [10] and [11].

A. The Conventional Design Process

The design process for asynchronous circuits can be divided into two major parts. The first part provides a formal description of the circuit's behavior such as a flow table. Based upon this formal description, the second part applies established synthesis techniques to generate the circuit design equations. These techniques include flow table reduction, internal-state assignment, hazard elimination and next-state and output equation generation [10,11].

Computer programs have made the synthesis techniques entirely automated. D. G. Raj-Karne [12] has recently programmed algorithms to provide either a Unicode Single Transition Time (USTT), Universal Totally Sequential (UTS) or combination USTT and UTS (Mixed Mode) state assignment. A flow table reduction algorithm and an algorithm to generate the static-hazard-free design equations have been programmed by R. J. Smith et al. [2,13].

B. Circuit Specification

1. Initial Descriptions

Presently, the most common initial description of a design is the English-word statement. This is an informal description that must be reworked into some type of formal description (usually a flow table). The word statement lacks total preciseness and often reflects uncertainty for many input/output conditions. An example of a typical word statement description is:

> A sequential circuit is to have two inputs A and B and one output Z. Z is to turn on only when B turns on, provided A is already on. Z is to turn off only when B turns off. Only one input can change state at a time.

In some designs a word statement may be accompanied by a timing chart [14,15] to express more explicitly particular input/output sequences required of the circuit. The timing chart usually does not show all possible input/output sequences of the circuit but rather shows important sequences which may help clarify the word statement. An example of the timing chart for the above word statement is shown in Figure 2.

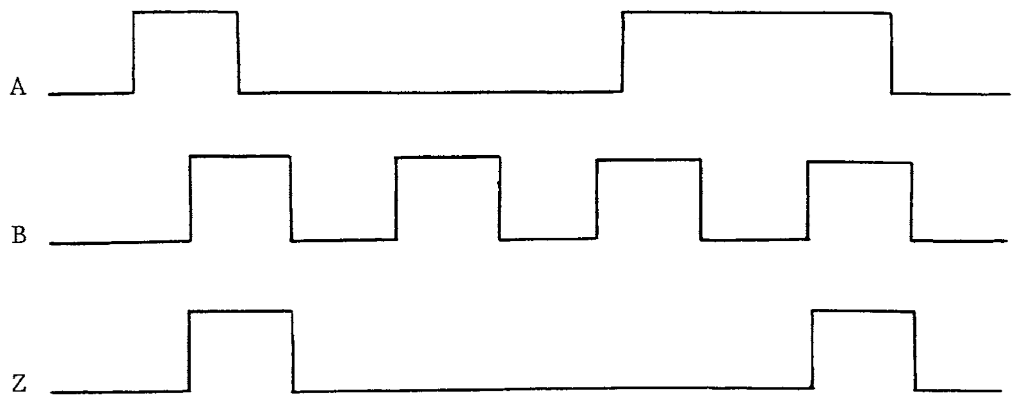


Figure 2. Timing Chart

2. Primitive Flow Table

In order to make the initial description more precise and appropriate for formal manipulation in the conventional synthesis procedures, the circuit specifications are made in the form of a flow table having exactly one stable state per row. This special table is called a primitive flow table [10,11,14,15] and is illustrated in Table II for the word statement description discussed earlier.

TABLE II. PRIMITIVE FLOW TABLE

Internal State	A B 00	01	11	10	Z
1	①	2	-	3	0
2	1	②	4	-	0
3	1	-	5	③	0
4	-	2	④	3	0
5	-	6	⑤	3	1
6	1	⑥	5	-	1

In constructing the primitive flow table, first a static situation corresponding to the initial state of the circuit is defined. This is usually (but not always) the state where all inputs and outputs are zero as indicated by stable state 1 in Table II. When operating in the initial stable state, the remaining next-state entries for row 1 are completed. Dash entries represent illegal input transitions which are later used in the synthesis as don't-cares. The dash in row 1 means the input transition 00 to 11 is illegal which agrees with the word description constraint of no inputs changing simultaneously. It isn't until state

5 is reached via state 3 that the output is set which concurs with the word description for setting Z.

For practice, the inexperienced reader should verify the remaining rows of the flow table. As seen from this example, considerable thought and time is required to construct the primitive flow table.

3. Other Methods

A method of specifying asynchronous circuits using input/output (I/O) sequences which could be translated into a flow table was developed by Altman [1]. However, the inefficiency of having to repeat long specification lists of I/O pairs at branch points and the inability to describe cyclic behavior of indeterminate duration greatly restricts the use of this method.

By developing a looping and branching technique and enabling the use of don't-care specifications, Smith [2] extended Altman's method to satisfy the above deficiencies. Smith's method is based on the philosophy that independent I/O sequences define submachines or modules; and when properly interconnected, these modules form the required sequential circuit. The method was primarily intended for those designs which originate from a circuit description having a set of I/O sequences and hence, is too restrictive to be a good, generalized design method. Since most designs originate from word descriptions or mental conceptions of an operative nature, listing the set of all independent I/O sequences for these designs becomes a difficult and confusing task.

A different approach to the synthesis of fundamental-mode asynchronous circuits was developed by Chuang [16] and is referred to as the transition logic synthesis method. In this method a binary level transition of 0 to 1 or 1 to 0 is considered as a pulse, and the realization of the circuit is similar to the standard pulse sequential circuit synthesis method [10]. Since no distinction is made between 0 to 1 and 1 to 0 transitions, the application of this method is limited to problems such as counters, where the toggling effect of the transition is of interest rather than the value of the transition's ending state. The specifications are made into an array called a transition flow table. Although the table may have fewer states than an equivalent primitive flow table, the unnatural specification method of flow table construction still exists.

III. A DESCRIPTION OF ACDL

This chapter contains a syntactic and semantic description of ACDL. The description begins with the lowest structural level of symbols and progresses to the higher structural levels of statements and programs.

A. The Definition of a Metalanguage

To prevent any ambiguities or paradoxes in the definition of ACDL, a metalanguage which is completely distinguishable from ACDL will be used in its description. To formalize the definitions in the metalanguage, each definition is given in the form of a statement or construct, which is analogous to a formula. The metalanguage employed is Backus Normal Form (BNF) [17] and consists of the following symbols:

<x>	to be read as "the object named x"
::=	to be read as "can be formed from"
|	to be read as "or" (exclusive or)

The metalanguage construct takes on the following meaning: "the object named in the corner braces may be formed from the objects named or specified on the right". Concatenation of names or objects is implied by the juxtaposition of names or objects in the construct. For clarity, those characters which are to form part of ACDL and the metalanguage symbols will be set in standard type while the names of objects enclosed in corner braces will be italicized.

B. Vocabulary of ACDL

1. Symbols

The set of symbols used in ACDL are defined as follows:

<*letter*>::= A|B|C|...|Z|#|_|@|$|'

<*binary digit*>::= 0|1

<nonbinary digit>::= 2|3|4|5|6|7|8|9

<digit>::= *<binary digit>*|*<nonbinary digit>*

<special character>::= "|/|?

<separator>::= ,|;|:|.|(|)

<relation symbol>::= =|->|=>

<replacement op>::= <-

<logical op>::= ¬|&|+

The relation symbol '->' is the transition symbol and is read as "makes a transition to", and '=>' is the implication symbol which is read as "implies that" or "causes". The special characters have special meanings in ACDL, and each will be explained later in the description at the place it is used. All other symbols take on their standard interpretations [3-9], [18].

2. Constants and Identifiers

The rules for constructing constants and identifiers from the symbols of ACDL are:

<constant>::= *<number>*|*<level>*|*<transition>*

<number>::= *<digit>*|*<number><digit>*

<level>::= *<binary digit>*

<transition>::= *<long transition>*|*<short transition>*

<long transition>::= 0->1|1->0

<short transition>::= ->1|->0|->?

<identifier>::= *<letter>*|*<identifier> <letter>*|*<identifier> <digit>*

Circuit input variables will take on transition values as well as level values. In many cases, both the beginning and ending states of an input transition are important in determining the resulting output level. The transition constant '0->1' indicates a transition from 0 to 1. The short transition is a shorthand notation for a long transition constant. For example, '->1' and '->0' are short for '0->1' and '1->0', respectively. The short transition '->?' represents a don't-care transition which essentially says that a transition is to occur and "you don't care" if it is a '0->1' or a '1->0' transition. This shorthand notation is a valuable asset to the user in making quick and easy specifications.

As in PL/I [18], the identifier may consist of both letters and digits with the restriction of beginning with a letter. Ideally, ACDL permits identifiers to be of an arbitrary length. However due to implementation restrictions of the current ACDL translator, the length of an identifier is limited to 256 characters.

3. Relations and Expressions

All expressions in ACDL are logical expressions. There are two types of logical expressions: the transition relation expressions and the Boolean variable expressions. The following rules for constructing these expressions will become more clear as the discussion progresses.

<level relation>::= *<identifier>* = *<level>*|*<level relation>*
& *<identifier>* = *<level>*|(*<level relation>*)

<transition relation>::= *<identifier>* = *<long transition>*
|*<identifier>* *<short transition>*
|*<transition relation>* & *<identifier>*
=*<long transition>*|*<transition relation>*
& *<identifier>* *<short transition>*|(*<transition relation>*)

<compound relation>::= *<transition relation>* WHILE *<level relation>*

<transition expression>::= *<transition relation>*|*<compound relation>*|*<transition expression>* + *<transition relation>*|*<transition expression>* + *<compound expression>*|*<dummy term>*|(*<transition expression>*)

<dummy term>::= LINKTEST | LK'T

<Boolean expression>::= *<logical factor>*|*<Boolean expression>* + *<logical factor>*

<logical factor>::= *<logical term>*|*<logical factor>* & *<logical term>*

<logical term>::= *<logical primary>* |¬*<logical primary>*

<logical primary>::= *<level>*|*<identifier>*|(*<Boolean expression>*)

Since all expressions are logical quantities, they evaluate to either of two values; true or false. The level and transition relations become true whenever the values of the identifiers equal the values of the constants. Similarly, the compound relation becomes true whenever the transition relation becomes true while the level relation is true. In circuit terminology this means some variables are to make a transition while others remain fixed. The dummy term, LINKTEST (LK'T) serves a special function which is described later in conjunction with the link statement.

The Boolean expression is the standard logical variable expression [3-10],[18]. Examples of this Boolean expression are given in Table III.

Free parenthetical form is permitted in Boolean expressions. However, if the order of logical operations is not specified by parenthesis, then the logical operators are applied in the standard hierarchical order of '¬', '&' and '+' [18].

TABLE III. EXAMPLES OF BOOLEAN EXPRESSIONS

Expression	Explanation
1	logic level 1
A	Boolean variable A
¬Z	Complement of Z
(A + B) & C	Logical OR of A and B ANDed with C
¬A + ¬B	Logical OR of the complements of A and B

Table IV further explains the use of the shorthand transition notation discussed earlier, and examples of the relation expression are given in Table V.

TABLE IV. SHORTHAND NOTATION FOR RELATION EXPRESSIONS

Expression	Shorthand Notation
X = 0 -> 1	X -> 1
X = 1 -> 0	X -> 0
(X = 0 -> 1) + (X = 1 -> 0)	X -> ?

4. Statement Labels

As in all programming languages, labels provide a means to select statements for execution that do not follow in the listed sequence [3-7, 18]. In ACDL there are two different types of labels: standard labels and output labels. These are further described as follows:

<*label*>::= <*standard label*>: |<*output label*>:

<*standard label*>::= <*letter* (*except* Z)>|<*standard label*>
<*letter*>|<*standard label*> <*digit*>

TABLE V. EXAMPLES OF THE RELATION EXPRESSION

Expression	Read As	Logical Meaning	Informal Meaning
X1->1	X1 makes a transition to 1	X1 goes from 0 to 1	X1 turns on X1 goes up etc.
X1->1 & X2->0	X1 makes a transition to 1 and X2 makes a transition to 0 simultaneously	X1 goes from 0 to 1 and X2 goes from 1 to 0 simultaneously	X1 turns on at the same time X2 turns off
X1->1 + X2->0	X1 makes a transition to 1 or X2 makes a transition to 0	X1 goes from 0 to 1 or X2 goes from 1 to 0	X1 is to turn on or X2 is to turn off
X1->?	X1 makes a transition	X1 goes from 0 to 1 or X1 goes from 1 to 0	X1 changes state i.e. it either turns on or turns off
X1->? & X2->?	X1 makes a transition and X2 makes a transition simultaneously	X1 and X2 go from 00 to 11 or 11 to 00 or 01 to 10 or 10 to 01	X1 and X2 change state simultaneously
X1->1 WHILE X2 = 1	X1 makes a transition to 1 while X2 equals 1	X1 goes from 0 to 1 while X2 stays at 1 i.e., inputs X1, X2 go from state 01 to 11	X1 turns on while X2 is held on

<output label>::= Z *<output code>*| Z (*<output state set>*)

<output state set>::= *<output code>*|*<output state set>*, *<output code>*

<output code>::= *<binary digit>*|*<output code> <binary digit>*|
<output code>/*<number>*

The letter Z is reserved for the beginning character of output labels, and it is followed by the current output state of the circuit. Hence, the output label serves two major purposes: 1) it provides the designer with the value of the present output state at a particular point in the design sequence; and 2) it indicates the next statement to be executed from that particular output state. The output state of the label is followed by a / *<number>* when it is necessary to distinguish a previous output label having the same output state. If the / *<number>* is not specified with the output label, the ACDL translator will assume a default value of /1 for the label. An output label may also be used to specify multiple output states in cases where the next statement to be executed is the same for each state.

A standard label is essentially a valid identifier with the restriction that the first character of the label cannot be the letter "Z". Examples of standard and output labels are given in Table VI.

TABLE VI. EXAMPLES OF LABELS

Label Type	Examples
Standard	FIRST:
Standard	$31:
Standard	BEGIN_HERE:
Output	Z00:
Output	Z(00, 01/2, 11):
Output	Z110/3:

5. Statements

There are nine types of statements in ACDL. As in PL/I [18], all statements must be separated by a semicolon; otherwise, they can be written in free format. Each is described below.

a. Design Statement

A description of the design statement is given below.

<design statement>::= DESIGN | DESIGN *<accounting information>*

<accounting information>::= *<design number>*|*<design number>*, *<designer's name>*|*<design number>*, *<designer's name>*, *<date>*

<design number>::= *<number>*

<designer's name>::= *<identifier>*|*<designer's name>* *<identifier>*

<date>::= *<identifier>* *<number>*, *<number>*

The design statement indicates the beginning of a new circuit design. The accounting information is optional and may indicate the design number, designer's name and date.

b. Declare Statement

The following is a description of the declare statement.

<declare statement>::= DECLARE*<declaration type>*|*<declare statement>* *<declaration type>*

<declaration type>::= *<input declaration>*|*<constraints declaration>* |*<output declaration>*|*<global declaration>*

<input declaration>::= INPUTS: *<variable definition>*|*<input declaration>*, *<variable definition>*

<variable definition>::= *<identifier>*|*<identifier>* *<initial condition>*

<initial condition>::= (*<level>*)

<constraints declaration>::= CONSTR: *<constraints>*

<constraints>::= NONE|AUS|SIC |*<transition expression>*
|*<level relation>*|*<constraints>*,
<transition expression>|*<constraints>*,
<level relation>

<output declaration>::= OUTPUTS: *<variable definition>*
|*<output declaration>*, *<variable definition>*

<global declaration>::= GLOBAL: *<list>*

<list>::= *<automatic link transition statement>*|*<list>*,
<automatic link transition statement>

The input and output declarations indicate the number and names of the input and output variables required in the design. The initial condition for each variable may or may not be given. If no initial value is explicitly shown, then an initial value of zero is assumed by default.

The constraints declaration indicates those input transitions that are not allowed. To assist the designer in problems which permit only single input changes, the constraints specification is given by the mneumonic SIC which stands for "Single Input Change." The mneumonic AUS means that "All Unspecified Sequences" of input transitions, i.e. those sequences which are not explicitly described in the design, are not permitted to occur. This constraint is extremely useful in problems where only a specific number of alternative sequences can occur. Level relation constraints restrict transitions to those input states which agree with the value of the relation. If there are no input transition constraints the word "NONE" must be written.

The global declaration is used when there are certain transition conditions which arise frequently throughout the design and are independent of any particular I/O sequence. Instead of repeating the transition statement many times in the design specifications, the statements are listed once in a global declaration. The global list consists of automatic link transition statements which are discussed later.

Some examples of the design and declare statements are given in Table VII.

TABLE VII. EXAMPLES OF THE DESIGN AND DECLARE STATEMENTS

Statement Type	Example
Design	DESIGN;
Design	DESIGN 103;
Design	DESIGN 103, JOHN DOE, APR 3, 1972;

Declare	DECLARE INPUTS: A,B
	CONSTR: SIC,
	A=1 and B=1
	OUTPUTS: Z;

Declare	DECLARE INPUTS: X1(1), X2(0) CONSTR: X1->1 WHILE X2=1, X2->1 WHILE X1=1 OUTPUTS: Z1, Z2 GLOBAL: X2->0=>Z1<-0/;

The initial conditions for the input variables are explicitly given in the example of the second declare statement, while they are left to default to zero in the first declare statement.

c. Start Statement

The start statement is defined as:

<start statement>::= START

This is always the first statement in the circuit's behavior description. Therefore, it acts as the starting point for the design sequences by invoking the initial I/O conditions and establishing the initial state of the machine.

d. Transition Statement

This statement is used to show input transitions and may relate an input transition to an output change. A description of this statement is shown below.

<transition statement>::= *<basic transition statement>*
| *<automatic link transition statement>*

<basic transition statement>::= *<transition expression>*
|*<transition expression>*
=> *<output change>*

<output change>::= *<identifier>*<-*<Boolean expression>*
|*<output change>*, *<identifier>* <-
<Boolean expression>

<automatic link transition statement>::= *<transition expression>*
=> *<output change>*
<autolink>

<auto link>::= / | / *<number>*

Since the inputs to the circuit change at random, the input transitions specified by the transition expression are essentially test conditions for particular changes. When the test conditions are satisfied the specified output change occurs.

The slash following the output change is an optional character used when automatic linking is desired. This concept is explained in a subsequent section on the list statement. Examples of basic transition statements are shown in Table VIII.

TABLE VIII. EXAMPLES OF THE TRANSITION STATEMENT

Statement	Explanation
A->1;	Input A is to make a transition to 1
A->0=>Z<-1;	Input A making a transition to 0 implies that output Z changes to 1 if not already 1
(C->1)+(B->1) =>G<-1,R<-0;	Input C making a transition to 1 or input B making a transition to 1 causes output G to be replaced to 1 and R to be replaced to 0

e. Link Statement

Generally, there will be many sequences of transition statements in a design specification and some subsequences of transition statements will be common to more than one sequence. Whenever a subsequence leads to two or more sequence paths, as in the case of alternate behaviors at a branch point, it is necessary to make the proper connection to each of these paths. These connections can be made with the link statement defined below.

<link statement>::= *<link conditional>*|*<link unconditional>*

<link conditional>::= *<tests>* *<branch points>*

<tests>::=(*<test condition>*|*<tests>*, *<test condition>*

<test condition>::= *<transition expression>*|*<level relation>*
|ELSE

<branch points>::=)*<single label>*|*<branch points>*, *<single label>*

<single label>::= *<standard label>*|Z*<output code>*

<link unconditional>::= LINK *<single label>*

The tests listed in the conditional link statement are the test conditions of the next transition statement for each new path, respectively. The next statement following the test condition for each path is given by the label corresponding by position to the test condition. Multiple output labels are not allowed in the link statement. This restriction, however, causes no branching limitations. Any statement having a multiple output label can be located by any one of its output states.

Whenever an output change follows a test condition of the LINK, it is shown as the first statement of the new path. In this case the dummy term LINKTEST (abbrev. LK'T) is inserted as the transition expression for this transition statement. This implies that the same test condition causing the link also causes the output change.

The link unconditional is the same as a "go to" statement and is primarily used for branching back to a previously specified statement.

f. Statement Block

Closely associated with the link statement are statement blocks. A description of the statement block is given below.

<statement block>::= *<beginning>* *<statement list>* *<ending>*

<beginning>::= BEGIN; | *<label>* BEGIN;

<ending>::= END|*<label>* END

Actually the BEGIN and END statements act as separators which serve to segregate a block of statements from other statements. This block of statements between a BEGIN and END is called a statement block and can only be entered from a link or list statement. Statement blocks may be nested within other statement blocks. After a statement block has been completed, control is transferred to the next statement in the listing which does not belong to another statement block of the same nested level. Examples of the link statement and statement blocks are given in Table IX.

TABLE IX. EXAMPLES OF LINK STATEMENTS

Statements	Explanation
LINK L3;	Link to the statement having the label L3
LINK (B->1,A->0)L1,L2; L1 : BEGIN; LK'T=>Z<-1; . . . END; L2 : BEGIN; C->1; . . . END; B->0; . . .	The 1st test condition transfers control to statement block L1 where the 1st statement says that the link test B->1 causes Z<-1. After the 1st statement block is completed, control is transferred to the transition statement B->0. If none of the Link statement test conditions are true for the current input transition, the sequence will not advance, but rather will remain at the Link statement until a test condition becomes true for some later transition.
LINK(A=1,ELSE)RESET, CONTINUE; RESET: A->0; CONTINUE: B->1; . . .	If input variable A is currently equal to 1, then branch to RESET else branch to CONTINUE.

g. Automatic Linking and the List Statement

Earlier it was noted that a slash "/" could follow the output change of a transition statement, and this slash meant automatic linking. This linking is accomplished by branching to the statement identified by an output label having the current output state. The current output state is the state entered after the output change of the transition statement has taken place. Therefore, if automatic linking is designated with the slash, the designer must ensure that a unique and correct output label has been assigned to some statement. To distinguish between output states having the same value, but occur at different points in the sequence, the designer follows the slash with a digit which must agree with the trailing digit of the correct output state label. Again if no digit is specified after the slash, a default value of 1 is assumed.

Automatic linking saves the designer having to explicitly specify a link statement and hence, improves the clarity of the specification listing. It was the automatic linking feature which led to the development of the list statement defined as:

<list statement>::= LIST *<list>*

<list>::= *<automatic link transition statement>*|*<list>*, *<automatic link transition statement>*

The list statement is a special purpose link statement in which all test conditions lead directly to an output change. It does not specify an executable sequence of transition statements, but rather, it is a set of statements from which only one is selected and executed. The test conditions of all automatic link transition statements in the list are scanned concurrently, and only one test condition may be true at a time. When a test condition becomes true, its corresponding output change indicates the next statement in the sequence via automatic linking. The transition statements within a LIST statement are separated by commas, while the end of the list is indicated by the LIST statement's semicolon. Some examples of the list statement are given in Table X.

TABLE X. EXAMPLES OF THE LIST STATEMENT

Example	Explanation
Z00: LIST X1->1 => Z1<-1 /, X2->1 => Z2<-1 /; . . . Z10: X2->1; . . . Z01: X1->1; . . .	In the list statement either of the two listed input transitions can occur. If X1->1 then an automatic link is made to the statement having the output label Z10. Similarly, if X2->1 is true, then a branch will be made to Z01 upon completion of the output change Z2<-1.
Z10: LIST A->0 => Z1<-0, Z2<-1 /2, B->1 => Z2<-1 /; . . . Z(01,11): B->0; . . . Z01/2: B->1 WHILE A=1; . . .	If the first test condition A->0 occurs, a link will automatically be made to Z01/2. If the second test condition B->1 occurs, a link will be made to the transition statement B->0 since Z11 is contained in the multiple output label Z(00,11).

6. Structure of an ACDL Program

Now that the statements have been defined, it is worth examining the overall structure of a program.

<program>::= <program head> <statement list> <ending>.

<program head>::= <design statement>;<declare statement>;
<start statement>;

<statement list>::= <statement>|<statement list> <statement>

<statement>::= <basic statement>|<statement block>;

<basic statement>::= <transition statement>;|<link statement>;
|<list statement>;|<label> <basic statement>

It is seen that the program ending also uses the word END. In this case END is followed by a period rather than a semicolon. The period signifies the end of the design as opposed to the end of a statement block. All links to this ending will indicate the end of certain sequences within the specifications.

The statements which make up the program head must be listed in the order shown. These statements are not part of the input/output behavior of the design, but rather provide basic information about the design. For this reason labels are not assigned to statements in the program head.

7. Comments and Translator Control Toggles

Comments are defined as follows:

<comment>::= "<almost anything>"

<almost anything>::= <any string of valid system 360 characters
that does not contain a " >

Comments help explain the program to persons reading it and are normally ignored by the translator. They do not result in the production of any translated text and they may be inserted any place a blank is allowed.

There is one case in which comments are not ignored. They may serve the special function of specifying control options which designate how the program is to be treated. For instance, a control option to perform and output a logic trace during the translation and/or execution of a program can be specified. A $ within a comment specifies that the next character is a control character. The valid control characters in ACDL are given in Table XI. Each control character acts as a toggle which can have the value of true or false. When $<*character*> is encountered by the translator, the value of the corresponding toggle is complemented. Therefore at the point where "$W" is first specified, the logic trace will be turned on, and will remain on until another "$W" is encountered which causes the trace to turn off. A more detailed description and use of the logic trace is given in Appendix B.

TABLE XI. COMMENT CONTROL OPTIONS

Character	Control Option
D	Print translation statistics, sequence tables and symbol tables at end of translation (Initially disabled)
L	List the source program (Initially enabled)*
T	Begin a machine code trace of the ACDL translator and interpreter (Initially disabled)*
U	Terminate the machine code trace of the ACDL translator and interpreter (Initially disabled)*
W	Begin high level trace of translation and execution (Initially disabled)
|	Set Margin. The portion of succeeding cards starting from the column containing the | will be ignored.*

*These options were already a part of the XPL system (See reference [17]).

C. Sequences

Statements in ACDL are executed in the sequential order in which they are listed, except when the physical sequence is interrupted by branching which results from explicit or automatic linking. The rules for interpreting sequences written in ACDL are:

1) In a test condition of an ACDL statement, any undesignated input variables are considered as don't-cares in the specified input state transition.

2) The sequence will advance to the next statement for any input state transition which agrees with a test condition of the current statement.

3) The sequence remains quiescent (i.e. does not move) for any input state transition which does not agree with a test condition of the current statement.

Before the designer can efficiently use ACDL, some idea as to what information is necessary for correctly specifying the operation of the sequential circuit is required.

Definition: A set of minimum length sequences of input states which cause the next output change and starts from the I/O state resulting from the previous output change is called a <u>critical</u> <u>event</u>.

A critical event may be an incompletely specified sequence i.e., a sequence which contains don't-care variables in some states. In this case the critical event will actually represent more than one possible sequence resulting from the random changing of the don't-care variables. However, any intermediate states that are introduced by the don't-care variables will not affect the integrity of the critical event, i.e., these states neither cause an output change nor destroy any past history of the critical event.

The designer must specify in ACDL all critical events of the circuit. This is done by starting from the initial state of the circuit and listing the critical events which cause the first output changes. Continuing from these points in the sequences all subsequent critical events which cause further output changes are listed. The tree process is continued until the critical events for all possible output changes have been listed.

Definition: A proposition of the design is a word statement (or mental conception) which implies one or more critical events.

From the design propositions, the designer should be able to formally specify the critical events in ACDL. Likewise from the ACDL specifications, the propositions should be easily determined. Examples of various types of circuit designs in ACDL are given in Chapter V.

D. Summary

Table XII summarizes the statements available in ACDL.

TABLE XII. SUMMARY OF ACDL

Statement	General Format Description		Use
Design	DESIGN	optional accounting information of design #, designer's name, and date;	Begins ACDL program
Declare	DECLARE		Defines all input variables, output variables, constraint transitions and global transitions of the design.
	INPUTS:	input names with or without initial conditions	
	CONSTR:	keywords, level expressions or transition expression constraints	
	OUTPUT:	output names with or without initial conditions	
	GLOBAL:	list of automatic link transition statements;	
Start	START;		The entry point of the sequence specifications.
Transition	input transition; (e.g. X->1;) or input transition=>output change with or without automatic linking specified; (e.g. X->1=>Z<-1;)		Expresses the I/O relationships of the critical events of the design.

TABLE XII. (Continued)

Link	LINK (test condition list) label list;	For branching in ACDL, where the test condition is an input level or transition test, causing a branch to the corresponding label.

Statement block	BEGIN ;
	statement list
	END ;
		For listing independent sequence paths resulting from a link or list statement.

List	LIST followed by a list of automatic link transition statements;	For branching when all test conditions lead directly to an output change.
Program end	END.	Designates the end of the sequence specifications and end of the design.
Comment	"any valid character string"	For clarification purposes and specifying control toggles.

IV. TRANSLATION AND INTERPRETATION

After the design has been specified in ACDL, a translation process is needed to convert the specifications into appropriate intermediate text. The intermediate text is then interpreted to produce a primitive flow table as the final output.

The translator and interpreter (flow table construction algorithm) is written in XPL, a special purpose translator writing language developed by McKeeman et al. [17]. A brief description of the XPL system follows.

A. The XPL Translator Writing System (TWS)

A diagram of the translator writing system provided with XPL is shown in Figure 3. The major components included in the system are:

1) a grammar analyzer (ANALYZER)
2) a proto-compiler (SKELETON) and
3) The XPL compiler (XCOM).

ANALYZER [17] is a program which reads the BNF grammar describing the syntax of the user's language, determines whether it is acceptable to the parsing algorithm and constructs parsing decision tables for that algorithm. SKELETON [17], which is written in XPL, provides the basic framework of the user's compiler such as, the routines for scanning, input and parse-stack maintenance. XCOM [17] is the compiler for the XPL language, and produces a System 360 machine language object program.

Depending on the amount of information supplied to TWS, the system will produce either a syntax checker, a translator or a combined translator/interpreter. If the user supplies only a syntax description (grammar) of his language in BNF, the resulting ANALYZER output deck

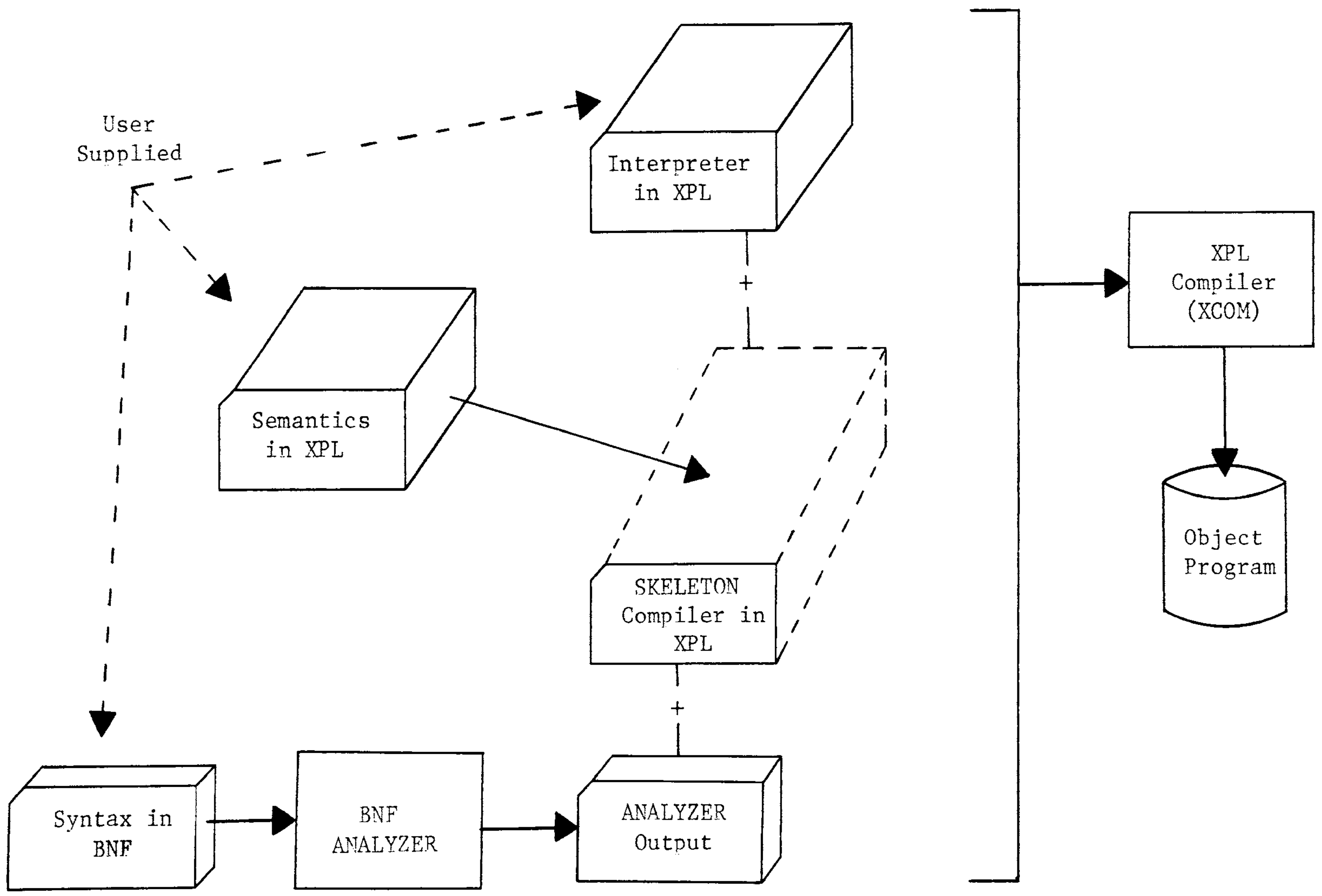


Figure 3. XPL Translator Writing System

and the SKELETON deck produce a syntax checker for the language. If the semantic description of his language (written in XPL) is also inserted into the SKELETON deck, the system will produce a translator. If in addition to the syntax and semantic descriptions, an interpreter is written in XPL and is placed behind the SKELETON deck, the system produces a combined translator/interpreter. It is this case which is illustrated in Figure 3.

An important advantage of TWS is the ease in which changes to the user's language can be made. Only the syntactic definition and semantic description corresponding to the language change are updated in the translator. A detailed description of the current syntactic definition (BNF grammar) and semantic routines of ACDL has been provided in Appendix D and Appendices E and F, respectively. Appendix C lists the job setup requirements for making a computer run to update the ACDL translator and the job setup for running a standard ACDL program.

B. ACDL Translation

From the syntax of a language, the translator knows what type of symbols are expected at every point in a language statement. The semantic routines then determine what action is to be taken, if any, when these symbols are encountered. They also generate internal data-structures which hold the results of the translation. These data-structures are referred to as the internal form or intermediate text. A description of the internal form used in the ACDL translation follows.

1. Internal Form

The internal form of the ACDL translator consists of tables which can be dumped at the end of the translation (see Table XI). The tables contain input/output sequence data as well as symbol information. As a means to increase program execution time, all input states are handled internally and recorded in the tables as decimal weights rather than as binary strings.

The main table used to hold the I/O sequence information is called the Primary Sequence Table. It stores the translated form of the statements making up the critical sequences of the design. The first row of the table is assigned to the START statement, with subsequent rows being assigned to each transition statement and each test condition of a link statement. Input and output secondary tables act as backup for the primary sequence table when statements have more than one input test or output change respectively. This technique of table organization saves storage since the secondary tables do not require all the fields that are needed in the primary table.

A special table called the Global Transition Table holds the information from the globally declared transition statements. The structure of this table is similar to the primary sequence table, and also uses the two secondary tables as backup tables. However, the Global Table functions differently in that all its input tests are queried continuously throughout the design.

The Constraint Transition Table records those input transitions that have been explicitly declared as input constraints. Restricted input transitions resulting from the SIC or AUS declaration are not recorded in this table because violations of these conditions are detected in an algorithmic manner. The constraint table does not use any backup tables. All restricted transitions are stored in sequential order in the table and are checked before examining any other tables.

The ACDL translation makes use of two separate symbol tables. One table is a standard symbol table used to record input, output and label names and their corresponding attributes. The other table is a special symbol table used to store output labels and statement pointers for automatic linking. Since only one table has to be searched for a particular operation, the two-table organization provides efficient symbol information retrieval and is especially advantageous in the automatic linking process.

A detailed description of the above tables and their corresponding fields is given in Appendix A. The appendix also includes an example program and corresponding table dump.

Another feature of the internal form is the handling of the Boolean output expressions. These expressions are converted from standard infix form to Reverse Polish form. The Reverse Polish form is stored in and executed from a single-dimensioned array. Special terminators are also stored in the array to indicate the beginning and end of an expression. For easy and rapid manipulation, the operators and operands are represented by their precedent values and symbol table indexes, respectively. For this reason, the Polish array was not included in the dump. However, the array is printed out whenever the logic trace is specified.

C. Primitive Flow Table Construction

Once the translation process has been completed, the interpretation process begins. Here the interpreter is an algorithm to construct a primitive flow table from the tables of the internal form. A flow chart describing the basic philosophy of the flow table algorithm is given in Figure 4.

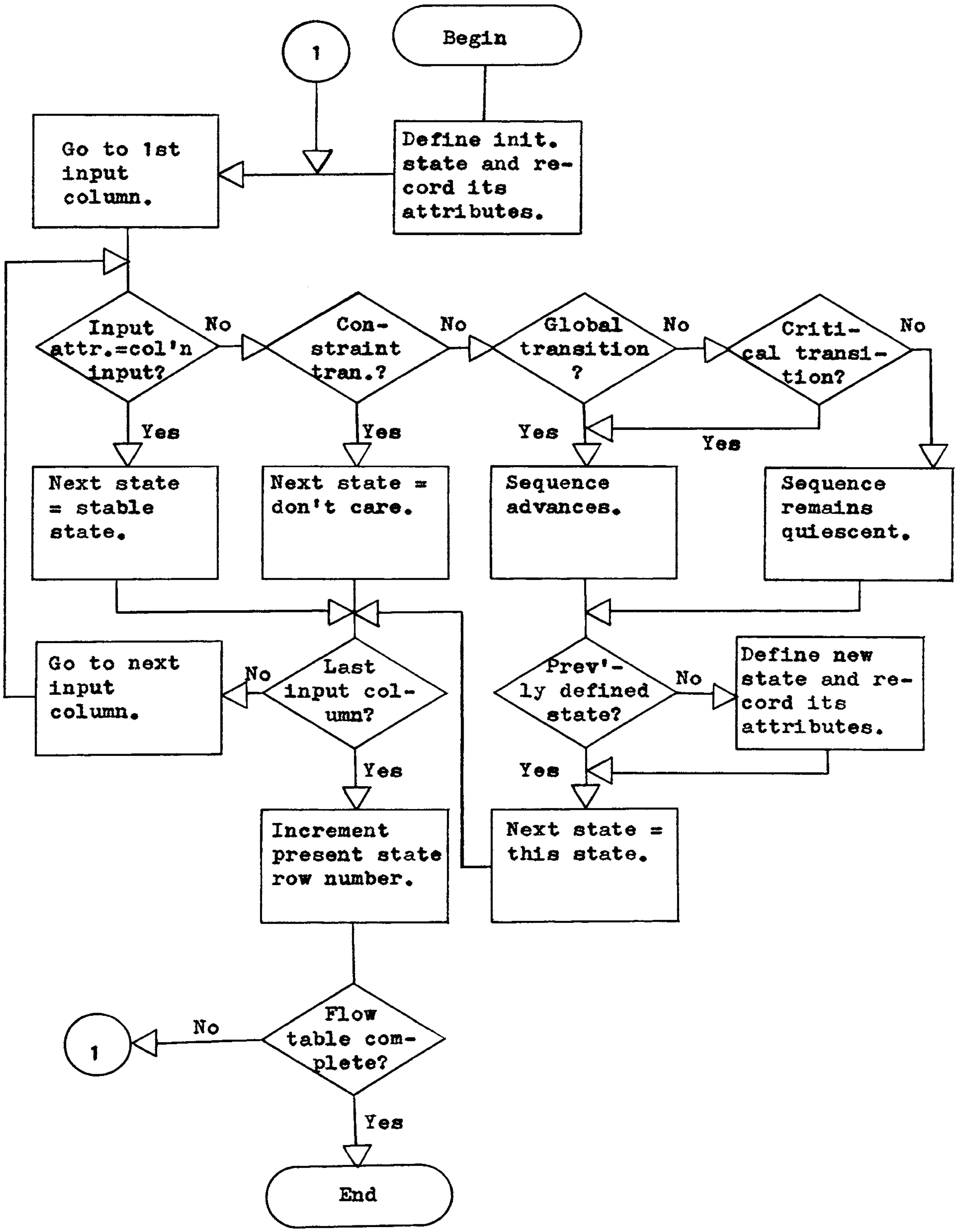


Figure 4. Flow Chart of the Primitive Flow Table Construction Algorithm

In addition to the output state, the flow table algorithm assigns input and sequence attributes to each row of the flow table. Each attribute has its own function in the flow table construction algorithm. The sequence attribute is an index which points to the statement in the Primary Sequence Table that is tested during the construction of the corresponding flow table row. It essentially keeps track of a flow table state's position in the design sequence. The input attribute contains the input state for which the corresponding flow table row is stable. Input transitions for a flow table row are simulated by using the input attribute as the beginning state and the column inputs as the ending states. This permits the computation of next-state entries to be conducted in an orderly manner with stable entries being recognized whenever the input attribute equals a column state.

To help understand the flow table algorithm, the following example problem [14] will be considered:

> Design a circuit which has two inputs, OSC and BTN, and one output, Z. The input OSC is the output of a square wave oscillator, and BTN is a button which, when depressed, gates one and only one full width oscillator pulse to the output. If the button is depressed for too short of time, a pulse will not occur at the output. An output pulse can occur only if the button depression overlaps the leading edge of an oscillator pulse. The inputs cannot change simultaneously.

The ACDL program for this design is:

```
          DESIGN 1, JOHN BROWN, SEP 27, 1972;
          DECLARE
               INPUTS:   OSC, BTN
               CONSTR:   SIC
               OUTPUTS:  Z;
Seq.#
  0       START;
  1  L2:  BTN->1;
  2       LINK (OSC->1,
  3            BTN->0) L1, L2;
  4  L1:  LK'T=>Z<-1;
  5       OSC->0=>Z<-0;
  6       END.
```

To simulate the internal form, the sequence numbers assigned to the statements in the above program correspond to the row indices of the Primary Sequence Table in which their translated form is stored. The start statement has the sequence number 0, because the Primary Sequence Table begins at index 0. The sequence numbers of the above program also relate to the sequence number attribute of each flow table row.

The first step in the algorithm is to define the initial state of the circuit. Since the start statement is responsible for setting up the initial conditions, this information is retrieved from row 0 of the Primary Sequence Table and assigned to the attributes of the first flow table row. At this point in the construction, the partial flow table resembles Table XIII (a). The sequence number attribute of flow table row 1 corresponds to the sequence number of the next statement to be executed in the ACDL program.

The next step is to compute next-state entries for row 1 of the flow table. The next state for the first input column is stable since the input attribute equals this input column. For the next input column, the first three input tests of the algorithm failed and the critical transition test is performed. For this test, the flow table transition, 00 to 01, is compared to the test condition, BTN->1, which is pointed to by the sequence attribute of 1. The test condition agrees with the flow table transition; therefore, the sequence advances. Since there is no previously defined state with correct attributes (i.e. Seq #=2, I=01, Z=0), the new state 2 is defined as shown in Table XIII (b).

The next input column implies the transition 00 to 10. Again the first three input tests failed, and since the flow table transition does not agree with the specified transition BTN->1, the critical transition test also fails. Therefore the sequence remains quiescent, and the next state retains the same sequence number and output state attributes as the present state. A new state 3 is defined because there is no previously defined state with the required attributes.

The last input column implies the transition 00 to 11, which will be detected by the constraint transition test as an illegal transition. A don't care will be entered in this column and row 1 is complete as shown in Table XIII (c). The algorithm moves to the next incomplete flow table row which is row 2 in this example and repeats the same procedure.

After all flow table rows are complete, the algorithm will terminate and produce the resulting flow table shown in Table XIV. From this table it is observed neither the sequence attribute 4 nor 6 is associated with any of the flow table rows. This is due to a property of the algorithm which advances the sequence through non-active test conditions such as LK'T, BEGIN, END and ELSE. The reader should now be able to verify the remaining flow table rows.

TABLE XIII. PARTIAL FLOW TABLE

Attributes		Present	OSC BTN				
Seq#	I	State	00	01	10	11	Z
1	00	1					0

(a) Definition of the Initial State

Attributes		Present	OSC BTN				
Seq#	I	State	00	01	10	11	Z
1	00	1	①	2			0
2	01	2					0

(b) Definition of State 2

Attributes		Present	OSC BTN				
Seq#	I	State	00	01	10	11	Z
1	00	1	①	2	3	-	0
2	01	2					0
1	10	3					0

(c) Completion of Row 1

An important feature of the algorithm is its ability to distinguish different states having the same input/output attributes. As seen from states 2 and 6 in Table XIV, it is the sequence attribute which provides this distinction.

To simplify the explanation of the flow table construction algorithm the previous example was performed without converting to internal form. Since the internal form is only a translated description of the ACDL statement, it should be clear that the same philosophy will apply. A more detailed description of the flow table procedures as they utilize the internal form is given in the flow charts of Appendix F.

TABLE XIV. RESULTING FLOW TABLE FOR EXAMPLE DESIGN

Attributes		Present	OSC BTN				
Seq#	I	State	00	01	10	11	Z
1	00	1	$\textcircled{1}$	2	3	-	0
2	01	2	1	$\textcircled{2}$	-	4	0
1	10	3	1	-	$\textcircled{3}$	5	0
5	11	4	-	6	7	$\textcircled{4}$	1
2	11	5	-	2	3	$\textcircled{5}$	0
1	01	6	1	$\textcircled{6}$	-	8	0
5	10	7	1	-	$\textcircled{7}$	4	1
1	11	8	-	6	3	$\textcircled{8}$	0

V. DESIGN EXAMPLES

This chapter is included to help the user become familiar with the ACDL specification process. It is felt this could best be accomplished by providing a set of example problems which illustrate the different features of the language. In each example a word description of the design is given. A reference number after the word description indicates the source from which the example was selected. The ACDL program and the primitive flow table follow each word description. All flow tables have been generated automatically from the ACDL descriptions. Due to its large size (78 rows by 8 columns), the flow table for example 8 is not shown.

A. Example 1

Design a Bounce Eliminator for a two position switch. The output state of the eliminator circuit is to indicate the desired position of the switch regardless of any contact bouncing which may occur when the switch is initially moved. It is assumed that the switch cannot bounce back far enough to contact the other position [19].

The ACDL Design is:

```
DESIGN 1  "BOUNCE ELIMINATOR";
DECLARE
     INPUTS:  A(1), B(0)
     CONSTR:  A=1 & B=1
     OUTPUTS: Z;
START;
B->1 => Z<-1;
A->1 => Z<-0;
END.
```

TABLE XV. PRIMITIVE FLOW TABLE FOR EXAMPLE 1

Present State	A B 00	01	10	11	Z
1	2	3	①	-	0
2	②	3	1	-	0
3	4	③	1	-	1
4	④	3	1	-	1

B. Example 2

Design a fundamental mode sequential circuit with two inputs X1 and X2. The single output Z is to be 1 only when X1,X2=01 provided that this is the fourth of a sequence of input combinations 00, 10, 11, 01. Otherwise, Z=0. Both inputs will not change simultaneously [19].

The ACDL Design is:

```
          DESIGN 2, JOHN BROWN, SEP 25, 1972;
          DECLARE
               INPUTS:  X1, X2
               CONSTR:  SIC
               OUTPUTS: Z;
          START;
     L1:  X1->1 WHILE X2=0;
          LINK(X2->1, X1->0) L3, L1;
     L3:  LINK(X1->0, X2->0) L4, L1;
     L4:  LK'T => Z<-1;
          (X1->?) + (X2->?) => Z<-0;
          END.
```

TABLE XVI. PRIMITIVE FLOW TABLE FOR EXAMPLE 2

Present State	X1 X2 00	01	10	11	Z
1	①	2	3	-	0
2	1	②	-	4	0
3	1	-	③	5	0
4	-	2	6	④	0
5	-	7	6	⑤	0
6	1	-	⑥	4	0
7	1	⑦	-	4	1

C. Example 3

A circuit is to be designed in which two push buttons A and B control the lighting of two lamps G and R. Whenever both push buttons are released, neither lamp is to be lit. Starting with both buttons released, the operation of either button causes lamp G to light. Operation of the other button, with the first button still held down, causes lamp R to light. Henceforth, as long as either button remains operated, the button which first caused lamp R to light controls lamp R--causing it to extinguish when the button is released and to light when the button is operated. The other button controls lamp G in the same fashion. It is not possible to operate or release both buttons simultaneously [10].

The ACDL Design is:

```
          DESIGN 3;
          DECLARE
               INPUTS:  A,B
               CONSTR:  SIC
               OUTPUTS: G,R
               GLOBAL:  (A->0 WHILE B=0) + (B->0 WHILE A=0)
                        => G<-0, R<-0 /;
          START;
    Z00:  LIST
            A->1 => G<-1 /,
            B->1 => G<-1 /2;
    Z10:  B->1 => R<-1 /;
    Z11:  LIST
            A->0 => G<-0 /,
            B->0 => R<-0 /;
    Z01:  A->1 => G<-1 /;
  Z10/2:  A->1 => R<-1 /2;
  Z11/2:  LIST
            A->0 => R<-0 /2,
            B->0 => G<-0 /2;
  Z01/2:  B->1 => G<-1 /2;
          END.
```

TABLE XVII. PRIMITIVE FLOW TABLE FOR EXAMPLE 3

Present	A B				
State	00	01	10	11	GR
1	$\textcircled{1}$	2	3	-	00
2	1	$\textcircled{2}$	-	4	10
3	1	-	$\textcircled{3}$	5	10
4	-	2	6	$\textcircled{4}$	11
5	-	7	3	$\textcircled{5}$	11
6	1	-	$\textcircled{6}$	4	01
7	1	$\textcircled{7}$	-	5	01

D. Example 4

Design an asynchronous 3-bit Gray Code counter which has one input X and 3 outputs, Z1, Z2, Z3. The counter is to count as indicated in Table 18 [20].

TABLE XVIII. THREE BIT GRAY CODE

Count	Z1	Z2	Z3
0	0	0	0
1	0	0	1
2	0	1	1
3	0	1	0
4	1	1	0
5	1	1	1
6	1	0	1
7	1	0	0

The ACDL Design is:

```
"GRAY CODE COUNTER"
DESIGN 4;
DECLARE
     INPUTS:  X
     CONSTR:  NONE
     OUTPUTS: Z1, Z2, Z3;
START;
     Z(000, 011, 110, 101): X->1=>Z3<- ¬Z3/;
     Z(001, 111): X->1=>Z2<- ¬Z2/;
     Z(010, 100): X->1=>Z1<- ¬Z1/;
END.
```

TABLE XIX. PRIMITIVE FLOW TABLE FOR EXAMPLE 4

Present	X				
State	0	1	Z1	Z2	Z3
1	$\textcircled{1}$	2	0	0	0
2	3	$\textcircled{2}$	0	0	1
3	$\textcircled{3}$	4	0	0	1
4	5	$\textcircled{4}$	0	1	1
5	$\textcircled{5}$	6	0	1	1
6	7	$\textcircled{6}$	0	1	0
7	$\textcircled{7}$	8	0	1	0
8	9	$\textcircled{8}$	1	1	0
9	$\textcircled{9}$	10	1	1	0
10	11	$\textcircled{10}$	1	1	1
11	$\textcircled{11}$	12	1	1	1
12	13	$\textcircled{12}$	1	0	1
13	$\textcircled{13}$	14	1	0	1
14	15	$\textcircled{14}$	1	0	0
15	$\textcircled{15}$	16	1	0	0
16	1	$\textcircled{16}$	0	0	0

E. Example 5

The timing signal X1 alternates between being off for 60 seconds (X1=0) and on for 30 seconds (X1=1). The only time Route 1 traffic can see a red light, a condition designated by Z=1, is during an interval in which X1=1. Only at the start of an X1=1 interval can Z go on, and once on, it must remain on for the full interval. If a car on Crumb Road actuates a switch, a condition designated by X2=1 (when no car is over such a switch X2=0), while Z=0, then Z should go on the next time X1 goes on [11].

The ACDL Design is:

```
     DESIGN 5;
     DECLARE
          INPUTS:   X1, X2
          CONSTR:   NONE
          OUTPUTS:  Z;
     START;
     X2->1;
L1:  X1->1=>Z<-1;
     X1->0=>Z<-0;
     LINK(X2=1, ELSE) L1, L2;
L2:  END.
```

TABLE XX. PRIMITIVE FLOW TABLE FOR EXAMPLE 5

Present State	X1 X2 00	01	10	11	Z
1	①	2	3	4	0
2	5	②	6	7	0
3	1	2	③	4	0
4	5	2	8	④	0
5	⑤	2	6	7	0
6	9	10	⑥	7	1
7	9	10	6	⑦	1
8	5	2	⑧	4	0
9	⑨	2	3	4	0
10	5	⑩	6	7	0

F. Example 6

Design an asynchronous sequential circuit for which only the four alternative sequences shown in the timing chart of Figure 5 can occur[15].

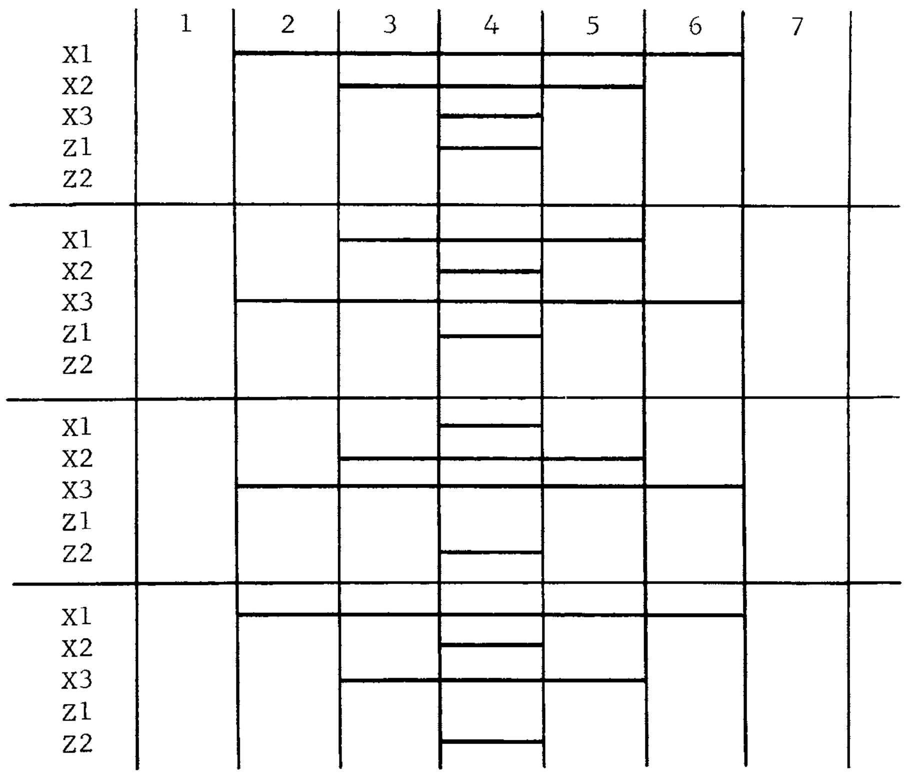


Figure 5. Timing Chart Indicating Allowable Sequences for Example 6

The ACDL Design is:

```
     DESIGN 6;
     DECLARE   INPUTS:  X1, X2, X3
               CONSTR:  AUS
               OUTPUTS: Z1, Z2;
     START;
     LINK(X1->1,X3->1)L1, L2;
L1:  BEGIN;
     LINK(X2->1, X3->1)L3, L4;
          L3:  BEGIN;
               X3->1⇒Z1<-1;
               X3->0⇒ Z1<-0;
               X2->0;
               END;
          L4:  BEGIN;
               X2->1=>Z2<-1;
               X2->0=>Z2<-0;
               X3->0;
               END;
     X1->0;
     END;
L2:  BEGIN;
     LINK(X1->1, X2->1)L5, L6;
          L5:  BEGIN;
               X2->1=>Z1<-1;
               X2->0=>Z1<-0;
               X1->0;
               END;
          L6:  BEGIN;
               X1->1=>Z2<-1;
               X1->0=> Z2<-0;
               X2->0;
               END;
     X3->0;
     END;
     END.
```

G. Example 7

Design an asynchronous version of a clamp-gate circuit. The circuit has two serial inputs X and Y and an output Z. The characteristics are such that Z is made equal to the present value of X if Y=1, or to the previous value of X if Y=0 [20].

The ACDL Design is:

```
     DESIGN 7 "CLAMP-GATE CIRCUIT";
     DECLARE  INPUTS:  X,Y
              CONSTR:  NONE
              OUTPUTS: Z;
     START;
     LINK(Y->?  WHILE X =0 + Y->?  WHILE X=1, X->?)L1, L2;
L1:  BEGIN;
     LK'T => Z<-X;
     END;
L2:  BEGIN;
     LK'T => Z<-(Y&X) + (¬Y&¬X);
     END;
     END.
```

TABLE XXI. PRIMITIVE FLOW TABLE FOR EXAMPLE 6

Present	X1 X2 X3									
State	000	001	010	011	100	101	110	111	Z1	Z2
1	①	2	-	-	3	-	-	-	0	0
2	-	②	-	4	-	5	-	-	0	0
3	-	-	-	-	③	6	7	-	0	0
4	-	-	-	④	-	-	-	8	0	0
5	-	-	-	-	-	⑤	-	9	0	0
6	-	-	-	-	-	⑥	-	10	0	0
7	-	-	-	-	-	-	⑦	11	0	0
8	-	-	-	12	-	-	-	⑧	0	1
9	-	-	-	-	-	13	-	⑨	1	0
10	-	-	-	-	-	14	-	⑩	0	1
11	-	-	-	-	-	-	15	⑪	1	0
12	-	16	-	⑫	-	-	-	-	0	0
13	-	16	-	-	-	⑬	-	-	0	0
14	-	-	-	-	17	⑭	-	-	0	0
15	-	-	-	-	17	-	⑮	-	0	0
16	1	⑯	-	-	-	-	-	-	0	0
17	1	-	-	-	⑰	-	-	-	0	0

TABLE XXII. PRIMITIVE FLOW TABLE FOR EXAMPLE 7

Present State	X Y 00	01	10	11	Z
1	①	2	3	4	0
2	1	②	3	4	0
3	5	2	③	4	0
4	5	2	6	④	1
5	⑤	2	3	4	1
6	5	2	⑥	4	1

H. Example 8

Design an asynchronous circuit which has 3 inputs A, B, and C and two outputs Z1 and Z2 and operates according to the following description. If C turns on Z1 goes on, or if A turns on Z2 goes on, provided B has turned on first in either case. However, B is not required to remain on until A or C turn on. Z1 and Z2 turn off whenever C and A turn off, respectively. Only single input changes are permitted.

The ACDL Design is:

```
          DESIGN 8;
          DECLARE INPUTS:  A, B, C
                  CONSTR:  SIC
                  OUTPUTS: Z1, Z2;
          START;
     Z00: B->1;
      S1: LIST
            C->1 => Z1<-1 /,
            A->1 => Z2<-1 /;
     Z10: LIST
            C->0 => Z1<-0 /,
            A->1 => Z2<-1 /;
   Z00/2: LINK (A->1, B->1)L1, S1;
      L1: LK'T => Z2<-1 /2;
```

```
Z01/2: LINK (A->0, B->1)L2, S1;
   L2: LK'T => Z2<-0 /;
  Z11: LIST
         C->0 => Z1<-0 /2,
         A->0 => Z2<-0 /2;
Z10/2: LINK(C->0, B->1)L3,S1;
   L3: LK'T => Z1<-0 /;
  Z01: LIST
         C->1 => Z1<-1 /,
         A->0 => Z2<-0 /3;
Z00/3: LINK(C->1, B->1)L4, S1;
   L4: LK'T => Z1<-1 /2;
       END.
```

As noted earlier, this example produces a 78 row primitive flow table which is not reproduced here.

VI. CONCLUSION

ACDL has proven to be a flexible system for specifying the terminal behavior of asynchronous circuits in terms of its problem versatility which includes designs originating from word descriptions or I/O sequences and other designs such as switches, flip-flops, counters etc. It is, however, best suited for problems originating from a word description of the circuit's operation since it was this type of problem which motivated the development of the language. Problems originating from I/O sequences are specified easily, but somewhat less naturally, then with the I/O sequence methods of Smith [2] and Altman [1]. The I/O sequences have to be converted to the ACDL transition statements as opposed to a direct listing.

In many problems, the critical event philosophy of listing only minimum sequences of input changes which cause output changes, greatly reduces the amount of information needed for specification. This feature may enable the designer to handle some large problems, but usually the exponential rate of increase of input states and therefore input sequences makes the problem too cumbersome. For this reason, the present implementation of the language is limited to six input variables, but this could easily be extended if necessary.

ACDL has also shown to be an efficient system in terms of computer and user specification times. As an indication of program execution time, the 78 row flow table which was generated for Example 8 of Chapter V, took only 6 seconds of CPU time. The byte storage capability of XPL permits efficient memory utilization of 120K bytes. User specification is easier and saves time compared to constructing the primitive flow table by hand. Also a problem can be designed in different ways by using the various features available in ACDL.

The ACDL system was tested with many examples including those given in Chapter V. The examples tested were an attempt to use every feature of the language to verify the correctness of the flow table generation algorithm. Correct flow tables were produced for every example tested.

Further research in the area of this dissertation may be directed toward:

1) the addition of pulse-mode design to the ACDL system,
2) the capability to connect previously designed networks to a current design by a library call technique and
3) the interconnection of the ACDL system with the available synthesis techniques to permit complete automated design.

BIBLIOGRAPHY

1. R. A. Altman, "The Computer Aided Generation of Flow Tables for Asynchronous Sequential Circuits," Masters Thesis, University of Missouri-Rolla, 1968.

2. R. J. Smith II, "Synthesis Heuristics For Large Asynchronous Sequential Circuits," Ph. D. Dissertation, University of Missouri-Rolla, 1970.

3. T. C. Bartee, I. L. Lebow, and I. S. Reed, Theory and Design of Digital Machines, New York: McGraw-Hill, P. 324, 1962.

4. H. Schorr, "Computer-Aided Digital System Design and Analysis Using a Register Transfer Language," IEEE Transactions on Electronic Computers, Vol. EC-13, pp. 730-737, December 1964.

5. Y. Chu, "An ALGOL-like Computer Design Language," Communications of the ACM, Vol. 8, pp. 607-615, October 1965.

6. K. E. Iverson, A Programming Language, New York: John Wiley and Sons, 1962.

7. J. R. Duley and D. L. Dietmeyer, "A Digital System Design Language (DDL)," IEEE Transactions on Electronic Computers, Vol. C-17, pp. 850-861, September 1968.

8. D. M. Rouse, "A Design Oriented Digital Design Language," Masters Thesis, University of Missouri-Rolla, 1969.

9. C. G. Bell and A. Newell, Computer Structures: Readings and Examples, New York: McGraw-Hill, Inc., 1971.

10. E. J. McCluskey, Introduction to the Theory of Switching Circuits, New York: McGraw-Hill, Inc., 1965.

11. S. H. Unger, Asynchronous Sequential Switching Circuits, New York: John Wiley and Sons, Inc., 1969.

12. D. G. Raj-Karne, "A Method for Generating a UTS Assignment with An Iterative State Transition Algorithm," Ph. D. Dissertation, University of Missouri-Rolla, Expected Completion Date, 1972.

13. R. J. Smith, J. H. Tracey, W. L. Schoeffel, and G. K. Maki, "Automation in the Design of Asynchronous Sequential Circuits," Proceedings 1968 Spring Joint Computer Conference, Vol. 32, pp. 55-60, 1968.

14. G. A. Maley and J. Earle, The Logic Design of Transistor Digital Computers, Englewood Cliffs, N. J.: Prentice-Hall, Inc. p. 261, 1963.

15. M. P. Marcus, Switching Circuits for Engineers, Second Edition, Englewood Cliffs, N. J.: Prentice-Hall, Inc., p. 190, 1967.

16. Y. H. Chuang, "Transition Logic Circuits and a Synthesis Method," IEEE Transactions on Computers, Vol. C-18, pp.154-168, February 1969.

17. W. M. McKeeman, J. J. Horning and D. B. Wortman, A Compiler Generator, Englewood Cliffs, N. J.: Prentice-Hall, Inc., 1970.

18. C. T. Fike, PL/I for Scientific Programmers, Englewood, N. J.: Prentice-Hall, Inc., 1970.

19. F. J. Hill and G. R. Peterson, Introduction to Switching Theory and Logical Design, New York: John Wiley and Sons, Inc., 1968.

20. D. Lewin, Logical Design of Switching Circuits, New York: American Elsevier, Inc., 1969.

VITA

Gregory Martin Bednar was born on June 26, 1944, in St. Louis, Missouri. He was graduated Valedictorian of his class from Cuba High School, Cuba, Missouri. He received a Bachelor of Science degree in Electrical Engineering from the University of Missouri-Rolla in August 1966. He was actively employed by IBM at Rochester, Minnesota, from August 1966 to September 1967, after which he began two years active duty as an officer in the United States Army. One year was spent in Vietnam where he was awarded the Bronze Star, May 1969, and the Bronze Star First Oak Leaf Cluster, July 1969, for Meritorious Achievement in Ground Operations Against Hostile Forces. Since September 1969 he has been enrolled in the Graduate School of the University of Missouri-Rolla and completed a Master of Science degree in Electrical Engineering in December 1970.

He is a member of IEEE, Eta Kappa Nu, Tau Beta Pi, and Phi Kappa Phi.

He is married to the former Carol Jeanne Diderrich and is the father of two children, Brian and Jeanne.

APPENDIX A

Description of the Internal Tables

The tables of the internal form are printed whenever the control toggle $D is specified in a comment statement (See Table XI). A detailed description of each table is given below. To help clarify the description, an example design program is illustrated below and its corresponding table dump is shown in Table XXIII.

```
      "$DUMP INTERNAL TABLES AT END OF COMPILATION"
      DESIGN 9 ;
      DECLARE INPUTS:  A,B
              CONSTR:  A->?  &  B->?
              OUTPUTS: Z1, Z2
              GLOBAL:  (A->0 WHILE B=0) + (B->0 WHILE A=0)
                        => Z1<-0, Z2<-0 / ;
      START ;
Z00:  A->1  =>  Z1<-1 / ;
Z10:  B->1  =>  Z2<-1 / ;
Z11:  LIST
          A->0  =>  Z1<-0 / ,
          B->0  =>  Z2<-0 / ;
Z01:  A->1  =>  Z1<-1 / ;
      END.
```

TABLE XXIII. A TABLE DUMP OF THE INTERNAL FORM FOR EXAMPLE DESIGN 9

PRIMARY SEQUENCE TABLE

LEVEL_BIT	LINK_BIT	B_INPUT	E_INPUT	TAB1	OCHANGE	TAB2	PTRAN	N_STMT
1	0	0	0	0	0	0	0	1
0	0	0	2	2	-5	0	0	-1
0	0	0	1	3	-7	0	0	-1
0	1	2	0	4	-9	0	0	-1
0	0	1	0	5	-11	0	0	-1
0	0	0	2	6	-13	0	0	-1
0	0	222	0	0	0	0	0	1

TABLE XXIII. (Continued)

SECONDARY TABLE 1

B_INPUT	E_INPUT	PTR1	MTRAN
1	0	0	0
1	3	0	0
2	3	0	0
3	1	0	0
3	2	0	0
1	3	0	0

SECONDARY TABLE 2

OCHANGE2	PTR2
-3	0

GLOBAL TRANSITION TABLE

GB_INPUT	GE_INPUT	GTAB1	GOCHANGE	GTAB2	GAL_DIGIT	GTRAN
2	0	1	-1	1	1	1

CONSTRAINT TRANSITION TABLE

CLEVEL_BIT	CB_INPUT	CE_INPUT	CTAB1
0	0	3	0
0	1	2	0
0	2	1	0
0	3	0	0

STANDARD SYMBOL TABLE

NAME	VALUE	TKN	POSITION
A	0	1	1
B	0	1	2
Z1	0	2	1
Z2	0	2	2

SPECIAL SYMBOL TABLE

OLABEL	AL_DIGIT	OINDEX
00	1	1
10	1	2
11	1	3
01	1	5

1. Primary Sequence Table

The rows of this table hold I/O specifications of the design. The table's indexing begins with row 0 which corresponds to the START statement. All input-state entries in the table are recorded as decimal weights. The fields of the table are:

1) LEVEL_BIT - is set to 1 when the test condition of the statement is a level test rather than a transition test.
2) LINK_BIT - is set to 1 whenever another test condition of a link or list statement is to be checked next if the current test condition is not true.
3) B_INPUT - holds either a level input test, the beginning input state of a transition test or a special code for LK'T, ELSE, BEGIN and END conditions. The ELSE, BEGIN and END conditions have the code 222 while LK'T (or LINKTEST) has the code 141.
4) E_INPUT - holds the ending input state of a transition test.
5) TAB1 - is a pointer to the next input test in secondary table 1 when don't cares or expressions cause more than one test per test condition. A value of 0 indicates no further tests are to be checked.
6) OCHANGE - contains information leading to the output change of a transition statement. A value of 0 indicates there is no output change. A positive integer is the address of a Boolean output expression. A negative integer is a pointer to the address of an output value in the standard symbol table.

7) TAB2 - is a pointer to the next output variable change in secondary <u>table</u> <u>2</u> when more than 1 output variable changes concurrently in a transition statement.

8) PTRAN - is set to 1 when there exists another input test to be checked as part of a <u>multiple</u> <u>transition</u> expression of the current statement.

9) N_STMT - indicates the <u>next</u> <u>statement</u> to be executed. A positive integer is a pointer to a statement in the primary sequence table. A negative integer indicates automatic linking and denotes the automatic link digit.

2. Secondary Table 1

Whenever a test condition of a transition statement or a link statement implies more than one possible input state transition, this table is used to store these extra input tests. For example in a two variable input design, these extra tests can be the result of unspecified variables in a test condition, "X1->1", a multiple transition expression, "X1->1 + X2->0", or a don't care transition, "X1->?". The TAB1 field of the primary sequence table is updated to the row number of the secondary table where the first extra test is stored. The four fields of secondary table 1 are:

1) B_INPUT1 - holds either another level test or the <u>beginning</u> <u>input</u> state of another input transition for the current statement.

2) E_INPUT1 - holds the <u>ending</u> <u>input</u> state of another input transition for the current statement.

3) PTR1 - is a <u>pointer</u> set to 1 when the next input test in this table is to follow the present test for the current statement.

4) MTRAN - is set to 1 whenever the following input test in the table is to be checked as part of a multiple transition expression of the current statement.

3. Secondary Table 2

When a transition statement contains more than one output change, this table holds the addresses of the additional output changes. The fields of this table are:

1) OCHANGE2 - is the same as the output change field (OCHANGE) of the primary sequence table for the additional output variables of a transition statement.

2) PTR2 - is a pointer set to 1 when the next output change in this table is to follow the present output change for the current statement.

4. Global Transition Table

This table is structurally similar to the primary sequence table, however, it only contains data from globally declared transition statements. It also uses the secondary tables as backup tables. The fields of this table are:

1) GB_INPUT - holds the beginning input state of a global transition test.

2) GE_INPUT - holds the ending input state of a global transition test.

3) GTAB1 - same as TAB1 in primary sequence table, except it is used for global statements.

4) GOCHANGE - same as OCHANGE in primary sequence table except it is used for global statements.

5) GTAB2 - same as TAB2 in primary sequence table except it is used for global statements.

6) GAL_DIGIT - denotes the automatic link digit of the corresponding global transition statement.

7) GTRAN - same as PTRAN in the primary sequence table except it is used for global transition statements.

5. Constraints Transition Table

This table stores all constrained input transitions except those declared by "SIC" and "AUS". The fields of this table are:

1) CLEVEL_BIT - is set to 1 when the input test of the constrained transition is a level test rather than a transition test. In this case it is the ending state of an input transition for which the level test is made.

2) CB_INPUT - holds the beginning input state of a constrained input transition.

3) CE_INPUT - holds the ending input state of a constrained input transition.

4) CTAB1 - is a pointer set to 1 when the next row in this table contains another constraint to be checked.

6. Standard Symbol Table

The standard symbol table contains the names of input variables, output variables and standard labels and their corresponding attributes. This table includes the following fields:

1) NAME - contains the names of inputs, outputs and standard labels.

2) VALUE - contains the value of a variable or the primary sequence table index of a label.

3) TKN - is a token field which is set to 0, 1 or 2 for distinguishing labels, input variables, and output variables, respectively.

4) POSITION - indicates the position of the input and output variables in an input and output state, respectively.

7. Special Symbol Table

This table is used to hold output labels and associated information. It plays an important role in the automatic linking process. The fields of this table are:

1) OLABEL - contains the binary string representation of an output state which is designated in an output label.

2) AL_DIGIT - contains the automatic link digit of the output label. If no digit is specified it defaults to 1.

3) OINDEX - contains the index of the corresponding statement in the primary sequence table.

APPENDIX B

Use of the Logic Trace Switch "$W"

The logic trace was extremely useful in debugging the translator/interpreter program. For this reason the trace was not removed. Instead, it is made readily available to the user in case changes to the language or program are desired in the future. The following will be a general description of the meaning and use of the information obtained when the logic trace is activated.

The logic trace is a built-in trace of the translator and flow table construction program. Within the XPL program listing, the location of trace output statements is easily recognized because they have the following format: "If T_SW > 0 then ...".

Whenever the logic trace is activated, the trace information follows the XPL program flow from procedure to procedure and within a procedure. Most of the trace information is just the current values of some variables. In these cases the variable names and their values are given. Comments defining all variables have been given in the program at their place of declaration. In a few cases duplicate information is seen, but two different procedures have output this information which allows the user to follow the procedure to procedure flow more easily.

If the logic trace, $W, is activated at the beginning of the design, a trace of the entire translation and flow table construction procedure is given. If "$W" is specified both at the beginning and end (i.e. ahead of END.) of the design, then the trace is only turned on for the translation process. Similarly if "$W" is only specified at the end of the design, the trace is just turned on for the flow table construction portion of the program.

APPENDIX C

Job Setup

This appendix provides the job setup or job control language (JCL) requirements for using the ACDL system on the IBM System 360 computer. The following listing shows the necessary cards to run an ACDL program when the ACDL translator/interpreter program is in object form and is residing on disk.

```
//OS  Job Card
//*  LIMITS=(R=130)
//S1  EXEC  XPLG
//G.PGM  DD  DSN=USER.S0150.TRACEY.ACDL.BEDNAR,
//       VOL=SER=USERVL,DISP=OLD,UNIT=DISK
//G.DATA  DD  *
          .
          .
          .
     ACDL Program
          .
          .
          .
/*
```

If a recompilation of the translator/interpreter program is required, the job setup for this is:

```
//OS  Job Card
//*  LIMITS=(R=(250),T=5,P=100)
//S1  EXEC  XPLC
//C.FILE1  DD  DSN=USER.S0150.TRACEY.ACDL.BEDNAR,
//       VOL=SER=USERVL,DISP=OLD,UNIT=DISK
//C.SOURCE  DD  *
          .
          .
          .
     Source deck of XPL program
          .
          .
          .
/*
```

The job setup for running an ACDL program with the translator/ interpreter program in source deck form is:

```
//OS  Job Card
//*  LIMITS=(R=(250),T=5,P=200)
//S1  EXEC  XPLCG
//C.SOURCE  DD  *
           .
           .
           .
    Source Deck of XPL Program
           .
           .
           .
/*
//G.DATA  DD  *
           .
           .
           .
    ACDL Program
           .
           .
           .
/*
```

The job setup for a BNF ANALYZER run is:

```
//OS  Job Card
//*  LIMITS=(R=150,P=40,T=5,C=100)
//S1  EXEC  XANALYZE
//XPL.SYNTAX  DD  *
         .
         .
         .
   BNF Description of ACDL
         .
         .
         .
$PUNCH
/*
```

APPENDIX D

BNF Description of ACDL for the Translator Writing System

In order to conserve computer storage and execution time the parsing algorithm of the ANALYZER program has the restriction of being context bounded. Specifically, it will parse only those grammars for which it can compute the stacking decision function by using no more than the top 2 symbols in the stack and the next symbol in the input text, and the production selection function, by using no more than 1 symbol below the production in the stack and the next symbol in the input text [17].

In order to make the BNF grammar of ACDL compatible to the context restrictions of ANALYZER, some changes to the description of Chapter III are required. The modified version of the grammar which was used as input to ANALYZER is shown below. In some cases, commas and parentheses required special definitions since these symbols were inadequate contexts for decision making.

To incorporate changes or extensions to the current structure of ACDL into the translator program, deletions, additions or modifications are made as necessary to the productions shown in the BNF description below. The job setup for an ANALYZER run is given in APPENDIX C.

The following is the present ANALYZER version of the BNF grammar for ACDL as it appears in the program listing.

```
1   <PROGRAM>  ::=  <PROGRAM HEAD>  <STATEMENT LIST>  <ENDING>

2   <STATEMENT LIST>  ::=  <STATEMENT>
3                        |  <STATEMENT LIST>  <STATEMENT>

4   <PROGRAM HEAD>  ::=  <DESIGN STMT>  ;  <DECLARE STMT>  ;  <START STMT>

5   <STATEMENT>  ::=  <BASIC STMT>
6                  |  <STMT BLOCK>  ;

7   <BASIC STMT>  ::=  <TRANSITION STMT>  ;
8                   |  <LINK STMT>  ;
9                   |  <LIST STMT>  ;
10                  |  <LABEL>  <BASIC STMT>

11  <ENDING>  ::=  END
12              |  <LABEL>  END

13  <BEGINNING>  ::=  BEGIN  ;
14                 |  <LABEL>  BEGIN  ;

15  <LABEL>  ::=  <SINGLE LABEL>  :
16             |  <LETTER  Z>  <(2>  <OUTPUT STATE SET>  )  :

17  <SINGLE LABEL>  ::=  <IDENTIFIER>
18                    |  <IDENTIFIER>  /  <NUMBER>

19  <LETTER Z>  ::=  <IDENTIFIER>

20  <OUTPUT STATE SET>  ::=  <OUTPUT CODE>
21                        |  <OUTPUT STATE SET>  ,  <OUTPUT CODE>

22  <OUTPUT CODE>  ::=  <NUMBER>
23                   |  <NUMBER>  /  <NUMBER>
```

```
24  <DESIGN STMT>  ::=  DESIGN
25                  |  DESIGN  <ACCOUNTING INFO>

26  <ACCOUNTING INFO>  ::=  <DESIGN NUMBER>
27                      |  <DESIGN NUMBER>  <,3>  <DESIGNERS NAME>
28                      |  <DESIGN NUMBER>  <,3>  <DESIGNERS NAME>  <,3>  <DATE>

29  <,3>  ::=  ,

30  <DESIGN NUMBER>  ::=  <NUMBER>

31  <DESIGNERS NAME>  ::=  <IDENTIFIER>
32                     |  <DESIGNERS NAME>  <IDENTIFIER>

33  <DATE>  ::=  <IDENTIFIER>  <NUMBER>  ,  <NUMBER>

34  <DECLARE STMT>  ::=  DECLARE  <DECLARATION TYPE>
35                   |  <DECLARE STMT>  <DECLARATION TYPE>

36  <DECLARATION TYPE>  ::=  <INPUT DCL>
37                       |  <CONSTRAINTS DCL>
38                       |  <OUTPUT DCL>
39                       |  <GLOBAL DCL>

40  <INPUT DCL>  ::=  INPUTS  :  <VARIABLE DEFN>
41                |  <INPUT DCL>  <,2>  <VARIABLE DEFN>

42  <VARIABLE DEFN>  ::=  <IDENTIFIER>
43                    |  <IDENTIFIER 1>  <INITIAL CONDITION>

44  <IDENTIFIER 1>  ::=  <IDENTIFIER>

45  <INITIAL CONDITION>  ::=  <(1>  <LEVEL>  )
```

```
46  <(1>  ::=  (

47  <CONSTRAINTS DCL>  ::=  CONSTR  :  <CONSTRAINTS>

48  <CONSTRAINTS>  ::=  NONE
49                    |  AUS
50                    |  SIC
51                    |  <TRANSITION TERM>
52                    |  <LEVEL FACTOR>
53                    |  <CONSTRAINTS>  <,2>  <TRANSITION TERM>
54                    |  <CONSTRAINTS>  <,2>  <LEVEL FACTOR>

55  <TRANSITION EXPRESSION>  ::=  <TRANSITION TERM>
56                             |  <TRANSITION EXPRESSION>  +  <TRANSITION TERM>
57                             |  <DUMMY TERM>

58  <DUMMY TERM>  ::=  LINKTEST
59                  |  LK'T

60  <COMPOUND RELATION>  ::=  <TRANSITION PART>  WHILE  <LEVEL PART>

61  <TRANSITION PART>  ::=  <TRANSITION FACTOR>
62                       |  <(2>  <TRANSITION  FACTOR>  )

63  <LEVEL PART>  ::=  <LEVEL FACTOR>
64                  |  <(2>  <LEVEL FACTOR>  )

65  <TRANSITION TERM>  ::=  <TRANSITION PART>
66                       |  <COMPOUND RELATION>
67                       |  <(2>  <COMPOUND RELATION>  )

68  <(2>  ::=  (
69  <TRANSITION FACTOR>  ::=  <TRANSITION RELATION>
70                           |  <TRANSITION FACTOR>  &  <TRANSITION RELATION>

71  <LEVEL FACTOR>  ::=  <LEVEL RELATION>
72                    |  <LEVEL FACTOR>  &  <LEVEL RELATION>

73  <TRANSITION RELATION>  ::=  <IDENTIFIER>  =  <TRANSITION>
74                           |  <IDENTIFIER>  <SHORT TRAN>

75  <LEVEL RELATION>  ::=  <IDENTIFIER>  =  <LEVEL>

76  <LEVEL>  ::=  <NUMBER>

77  <TRANSITION>  ::=  <NUMBER> - >  <NUMBER>

78  <SHORT TRAN>  ::=  - >  <NUMBER>
79                  |  - >  ?

80  <OUTPUT DCL>  ::=  OUTPUTS  :  <VARIABLE DEFN>
81                  |  <OUTPUT DCL>  <,2>  <VARIABLE DEFN>

82  <,2>  ::=  ,

83  <GLOBAL DCL>  ::=  GLOBAL  :  <LIST>

84  <START STMT>  ::=  START  ;

85  <STMT BLOCK>  ::=  <BEGINNING>  <STATEMENT LIST>  <ENDING>

86  <TRANSITION STMT>  ::=  <BASIC TRAN STMT>
87                       |  <AUTO LINK TRAN STMT>

88  <BASIC TRAN STMT>  ::=  <TRANSITION EXPRESSION>
89                       |  <TRANSITION EXPRESSION>  = >  <OUTPUT CHANGE>
```

```
 90  <AUTO LINK TRAN STMT>  ::=  <BASIC TRAN STMT>  <AUTO LINK>

 91  <AUTO LINK>  ::=  /
 92                 |  /  <NUMBER>

 93  <OUTPUT CHANGE>  ::=  <IDENTIFIER>  <REPLACEMENT OP>  <OUTPUT EXPRESSION>
 94                     |  <OUTPUT CHANGE>  <,1>  <IDENTIFIER>  <REPLACEMENT OP>  <OUTPUT EXPRESSION>

 95  <REPLACEMENT OP>  ::=  < -

 96  <OUTPUT EXPRESSION>  ::=  <LEVEL>
 97                         |  <BOOL EXPR>

 98  <BOOL EXPR>  ::=  <LOG FACTOR>
 99                 |  <BOOL EXPR>  +  <LOG FACTOR>

100  <LOG FACTOR>  ::=  <LOG TERM>
101                  |  <LOG FACTOR>  &  <LOG TERM>

102  <LOG TERM>  ::=  <LOG PRIMARY>
103                |  ¬  <LOG PRIMARY>

104  <LOG PRIMARY>  ::=  <IDENTIFIER>
105                   |  <( 2>  <BOOL EXPR>  )

106  <LINK STMT>  ::=  LINK  <PARAMETER LIST>

107  <PARAMETER LIST>  ::=  <NO TESTS>
108                      |  <1 TEST>  <1 LABEL>
109                      |  <2 TESTS>  <2 LABELS>
110                      |  <3 TESTS>  <3 LABELS>
111                      |  <4 TESTS>  <4 LABELS>
112                      |  <5 TESTS>  <5 LABELS>
113                      |  <6 TESTS>  <6 LABELS>
```

```
114  <NO TESTS>  ::=  <SINGLE LABEL>

115  <1 TEST>  ::=  <(1>  <TEST CONDITION>

116  <1 LABEL>  ::=  )  <SINGLE LABEL>

117  <2 TESTS>  ::=  <1 TEST>  <,1>  <TEST CONDITION>

118  <2 LABELS>  ::=  <1 LABEL>  <,1>  <SINGLE LABEL>

119  <3 TESTS>  ::=  <2 TESTS>  <,1>  <TEST CONDITION>

120  <3 LABELS>  ::=  <2 LABELS>  <,1>  <SINGLE LABEL>

121  <4 TESTS>  ::=  <3 TESTS>  <,1>  <TEST CONDITION>

122  <4 LABELS>  ::=  <3 LABELS>  <,1>  <SINGLE LABEL>

123  <5 TESTS>  ::=  <4 TESTS>  <,1>  <TEST CONDITION>

124  <5 LABELS>  ::=  <4 LABELS>  <,1>  <SINGLE LABEL>

125  <6 TESTS>  ::=  <5 TESTS>  <,1>  <TEST CONDITION>

126  <6 LABELS>  ::=  <5 LABELS>  <,1>  <SINGLE LABEL>

127  <,1>  ::=  ,

128  <TEST CONDITION>  ::=  <TRANSITION EXPRESSION>
129                       |  <LEVEL FACTOR>
130                       |  ELSE
```

```
    131  <LIST STMT>  ::=  LIST  <LIST>

    132  <LIST>  ::=  <AUTO LINK TRAN STMT>
    133              |  <LIST>  <,1>  <AUTO LINK TRAN STMT>
$PUNCH
```

APPENDIX E

Program Structure

The ACDL translation and flow table construction program consists of 44 procedures. Figure 6 indicates the overall structure of the program with respect to these procedures.

```
MAIN_
PROCEDURE  [ invokes INITIALIZATION
             invokes COMPILATION_LOOP
             invokes PRINT_SUMMARY

             INITIALIZATION [ invokes PRINT_DATE_AND_TIME
                              invokes SCAN

                              PRINT_
                              DATE_
                              AND_TIME [ invokes PRINT_TIME

                                         PRINT_TIME [

                                  SCAN [ invokes ERROR
                                         invokes GET_CARD
                                         invokes CHAR

                                                ERROR [ invokes I_FORMAT

                                                        I_FORMAT [

                                             GET_CARD [ invokes ERROR
                                                        invokes I_FORMAT

                                                 CHAR [ invokes GET_CARD

              COMPILATION_
              LOOP          [ invokes STACKING
                              invokes ERROR
                              invokes SCAN
                              invokes REDUCE
```

Figure 6. Structure of the ACDL Translator/Interpreter Program

```
|           |   STACKING [ invokes ERROR
|           |            | invokes STACK_DUMP
|           |            | invokes RECOVER
|           |            |
|           |            | STACK_DUMP [
|           |            |
|           |            |    RECOVER [ invokes SCAN
|           |            |            | invokes RIGHT_CONFLICT
|           |            |            |
|           |            |            |  RIGHT_
|           |            |            |  CONFLICT [
|           |            [            [
|           |     REDUCE [ invokes PR_OK
|           |            | invokes SYNTHESIZE
|           |            | invokes ERROR
|           |            | invokes STACK_DUMP
|           |            | invokes RECOVER
|           |            |
|           |            |      PR_OK [ invokes RIGHT_CONFLICT
|           |            |
|           |            | SYNTHESIZE [ invokes ERROR
|           |            |            | invokes STACK_DUMP
|           |            |            | invokes SPEC_LOOKUP
|           |            |            | invokes STD_LOOKUP
|           |            |            | invokes ENDING
|           |            |            | invokes BEGINNING
|           |            |            | invokes PAD_ZEROS
|           |            |            | invokes VARSTORE
|           |            |            | invokes CONSTR_TRAN
|           |            |            | invokes DEC_XFM
```

Figure 6. (Continued)

```
|          |       |          | invokes DC_TRAN
|          |       |          | invokes MULTI_TRAN
|          |       |          | invokes OVAR_CHK
|          |       |          | invokes IVAR_CHK
|          |       |          | invokes POLISHX
|          |       |          |
|          |       |          |     SPEC_   ┌
|          |       |          |     LOOKUP  │ invokes ERROR
|          |       |          |             └
|          |       |          |     STD_    ┌
|          |       |          |     LOOKUP  │ invokes ERROR
|          |       |          |             └
|          |       |          |     ENDING  [
|          |       |          |
|          |       |          | BEGINNING   [
|          |       |          |
|          |       |          | PAD_ZEROS   [
|          |       |          |
|          |       |          |  VARSTORE   ┌
|          |       |          |             │ invokes ERROR
|          |       |          |             │ invokes STD_LOOKUP
|          |       |          |             └
|          |       |          |    CONSTR_  ┌
|          |       |          |    TRAN     │ invokes CTRAN_STORE
|          |       |          |             │ invokes DC_TRAN
|          |       |          |             │
|          |       |          |             │ CTRAN_  ┌
|          |       |          |             │ STORE   │ invokes DEC_XFM
|          |       |          |             └         └
|          |       |          |       DEC_  ┌
|          |       |          |       XFM   │ invokes EXPN
|          |       |          |             │
|          |       |          |             │   EXPN  [
|          |       |          |             └
```

Figure 6. (Continued)

```
|            |        |          |   DC_TRAN  [ invokes EXPN
|            |        |          |              invokes MULTI_TRAN
|            |        |          |
|            |        |          |   MULTI_   [ invokes DEC_XFM
|            |        |          |   TRAN       invokes ERROR
|            |        |          |
|            |        |          |   OVAR_CHK [ invokes ERROR
|            |        |          |              invokes STD_LOOKUP
|            |        |          |
|            |        |          |   IVAR_CHK [ invokes ERROR
|            |        |          |              invokes STD_LOOKUP
|            |        |          |
|            |        |          |   POLISHX  [
|            |_       |_         |_
|   PRINT_SUMMARY [ invokes PRINT_DATE_AND_TIME
|                   invokes DUMPIT
|                   invokes PRINT_TIME
|
|                    DUMPIT [
|_
FLOW_  [
TABLE    invokes LINKAGE
         invokes EXPN
         invokes PRIMARY_TABLE
         invokes GLOBAL_TABLE
         invokes CONSTR_TABLE
         invokes ERROR
         invokes OUTPUT_FLOW_TABLE

               LINKAGE [
```

Figure 6. (Continued)

```
PRIMARY_TABLE
    invokes TABLE1_SEARCH
    invokes LINKAGE
    invokes POLISH_EXEC
    invokes TABLE2_SEARCH
    invokes SPEC_LOOKUP

    TABLE1_
    SEARCH

    POLISH_
    EXEC

    TABLE2_
    SEARCH
        invokes POLISH_EXEC
GLOBAL_TABLE
    invokes TABLE1_SEARCH
    invokes POLISH_EXEC
    invokes TABLE2_SEARCH
    invokes SPEC_LOOKUP
    invokes LINKAGE
CONSTR_TABLE
OUTPUT_FLOW_
TABLE
    invokes PAD
    invokes I_FORMAT

        PAD
```

Figure 6. (Continued)

APPENDIX F

Procedure Descriptions

This appendix presents a description of the procedures composing the translator and flow table synthesis algorithm. A brief description of the procedures' main functions is given. Functional flow charts accompany those procedure descriptions which require a more detailed explanation; however, their emphasis is directed toward the synthesis procedures rather than the analysis (parsing) procedures, since it is these procedures that are ACDL dependent.

1. MAIN_PROCEDURE Procedure*

This procedure is the main entry point of the program. It is the master control for the translation process and collects timing information for the different phases of the translation.

2. INITIALIZATION Procedure*

This procedure prints the headings for the compilation listing. It initializes the character classes for the scanner and various other global variables. No initializations of 0 have to be made since all variables are automatically initialized to 0 (or null in the case of character string variables) by the XPL compiler, XCOM, unless otherwise indicated.

3. PRINT_DATE_AND_TIME Procedure*

This procedure decodes the date into year, month and day, and then calls PRINT_TIME to print it.

*This procedure is part of the original proto-compiler SKELETON of TWS [17].

4. PRINT_TIME Procedure*

This procedure decodes time from hundreds of seconds into hours, minutes and seconds, and prints it together with its message parameter.

5. SCAN Procedure*

An ACDL program consists of a sequence of symbols interspersed with blanks and comments. A call to SCAN either produces the next symbol, removes the blanks and comments, or is responsible for the setting and resetting of control toggles.

6. GET_CARD Procedure*

This procedure reads a source card and stores it in the global character variable TEXT. It also prints the card image unless control toggle $L has been specified, which inhibits the source program listing.

7. CHAR Procedure*

This procedure is used to advance the scan pointer by one character and to get a new card if necessary.

8. I_FORMAT Procedure*

This procedure right justifies an integer in the field width specified.

9. ERROR Procedure*

This procedure prints error messages, counts total errors and severe errors and terminates compilation in case of excessive errors.

10. COMPILATION_LOOP Procedure*

This procedure coordinates the stacking of symbols and their

*This procedure is part of the original proto-compiler SKELETON of TWS [17].

reduction according to the BNF constructs. A flow chart for this procedure is shown in Figure 7.

11. STACKING Procedure*

This procedure is the basic decision function of the parsing algorithm. When the function is true, a symbol is stacked; when it is false, a reduction is made. If an error is detected, a recovery is initiated and a new value is computed.

12. RECOVER Procedure*

This procedure removes enough of the parse stack and input text to ensure that translation can proceed at least one more step without further errors. In many cases this procedure prevents single errors from causing multiple messages.

13. RIGHT_CONFLICT Procedure*

The most recently scanned symbol is stored in the global variable TOKEN. This procedure decides if a string in the parse stack is reducible on the basis that the result of the reduction must yield an allowed pair between the top of the stack and TOKEN. Similarly, when an error is encountered, parsing is not resumed until an allowed pair is in TOKEN and on top of the parse stack.

14. STACK_DUMP Procedure*

When syntactic errors are discovered by the analysis algorithm, the state of the parse stack is printed by this procedure as a diagnostic aid.

15. REDUCE Procedure*

This procedure looks up the proper reduction, calls SYNTHESIZE

*This procedure is part of the original proto-compiler SKELETON of TWS [17].

to produce the associated semantic action and then makes the reduction.

16. PR_OK Procedure*

When there is more than one reducible string on the parse stack, this procedure uses the syntactic analysis tables to choose the proper reduction.

17. SYNTHESIZE Procedure

This procedure and the procedures it calls compose the semantic routines that are inserted into the skeleton deck. Corresponding to each production recognized by REDUCE, this procedure takes appropriate action to produce code and data images of the program being translated. Flow charts describing the action to be taken at the important productions are given in Figure 8. The case number corresponds to the position of the production in the BNF description of Appendix D. No action is taken in the cases not shown.

18. SPEC_LOOKUP Procedure

This procedure is a function procedure with two arguments, the output state symbol being searched for and the corresponding automatic link digit of the output state. The procedure searches the special symbol table in a sequential manner for the output symbol and automatic link digit passed to it. If the output symbol and correct automatic link digit are not found, the procedure will add them to the table. The row number of the symbol table in which the arguments were found (or added) is returned as the value of the function.

*This procedure is part of the original proto-compiler SKELETON of TWS [17].

19. STD_LOOKUP Procedure

This procedure is a function procedure with the symbol being searched for as its argument. The procedure sequentially searches the standard symbol table for the symbol passed to it. If the symbol is not found, the procedure will add the symbol to the table. The row number of the table in which the symbol was found (or added) is returned as the value of the function.

20. ENDING Procedure

This procedure is used to handle the ends of statement blocks and the end of the design (see Figure 9).

21. BEGINNING Procedure

This procedure is a special procedure used to handle the beginning of statement blocks. A flow chart for this procedure is shown in Figure 10.

22. PAD_ZEROS Procedure

This procedure adds zeros on the left of an integer until the specified field width is reached.

23. VARSTORE Procedure

This procedure stores input and output variables and their corresponding attributes in the standard symbol table via the function procedure STD_LOOKUP.

24. CONSTR_TRAN Procedure

This procedure controls the construction of the constraint transition table. A flow chart describing this procedure is illustrated in Figure 11.

25. DEC_XFM Procedure

This procedure transforms a binary representation of an input transition or level to a decimal representation as indicated in Figure 13.

26. DC_TRAN Procedure

This procedure is used to handle the don't-care shorthand transition. The flow chart describing this procedure is illustrated in Figure 13.

27. MULTI_TRAN Procedure

This procedure stores the additional input transitions resulting from don't-cares and multiple transition expressions in secondary table 1. Figure 14 illustrates the flow chart for this procedure.

28. OVAR_CHK Procedure

This procedure checks the standard symbol table to see if output variables in transition statements have been properly defined. If not, an error message is printed.

29. IVAR_CHK Procedure

This procedure checks the standard symbol table to see if input variables in transition statements have been properly defined. If not, an error message is printed.

30. POLISHX Procedure

This procedure translates a Boolean infix expression to Reverse Polish format. A flow chart describing this procedure is given in Figure 15.

31. CTRAN_STORE Procedure

This procedure stores the input constraints in the constraint transition table. The flow chart for this procedure is illustrated in Figure 16.

32. EXPN Procedure

This is a function procedure which performs the exponentiation operation. The result is returned as the value of the function.

33. PRINT_SUMMARY Procedure*

This procedure prints the statistics of the translation which includes error statistics and translation times and rates. It also calls DUMPIT procedure to dump the internal tables if $D was given in an ACDL comment.

34. DUMPIT Procedure

This procedure dumps the internal form tables whenever the control toggle $D is specified in an ACDL comment.

35. FLOW_TABLE Procedure

This procedure controls the construction of the primitive flow table. Its flow chart is shown in Figure 17.

36. LINKAGE Procedure

This is a function procedure which is a housekeeping routine for the state linkage table. The state linkage table records the internal states (i.e. flow table row numbers) that have been defined. Therefore, this procedure determines whether or not a correct next-state entry has been previously defined. If not, it defines a new state and records it in the state linkage table. The procedure has 3 arguments; the current input column state, the sequence # attribute of the next state, and the output attribute of the next state. The procedure returns the proper next-state entry as the value of its function. The flow chart for this procedure is given in Figure 18.

*This procedure is part of the original proto-compiler SKELETON of TWS [17].

37. PRIMARY_TABLE Procedure

This procedure checks the primary sequence table to see if the current input transition was specified in the design. A flow chart describing this procedure is shown in Figure 19.

38. GLOBAL_TABLE Procedure

This is a function procedure which checks the global transition table to see if the current input transition is a global transition. If so, the procedure returns a value of true as the value of its function. A flow chart describing this procedure is given in Figure 20.

39. CONSTR_TABLE Procedure

This is a function procedure which checks the constraint transition table to see if the current input transition is a constraint transition. If so, the procedure returns a value of 1, otherwise 0. The flow chart for this procedure is given in Figure 21.

40. TABLE1_SEARCH Procedure

This is a function procedure which searches secondary table 1 for additional input test conditions resulting from don't cares or multiple transition expressions. The value returned by the procedure indicates the point where the calling procedure is to continue. Figure 22 illustrates the flow chart for this procedure.

41. TABLE2_SEARCH Procedure

This procedure searches secondary table 2 for additional output changes whenever more than one output variable changes concurrently in a transition statement. The flow chart for the procedure is shown in Figure 23.

42. POLISH_EXEC Procedure

This procedure interpretively executes a Reverse Polish Boolean expression. A flow chart of the procedure is given in Figure 24.

43. OUTPUT_FLOW_TABLE Procedure

This procedure is used to print the resulting primitive flow table in its standard format. Stable states are indicated by attaching a minus sign to the next-state entry.

44. PAD Procedure*

This is a function procedure with two arguments; a character string variable and a format field width. The procedure adds blanks to the right of the character string variable to give it the field width specified. The padded string is then returned by the procedure.

* This procedure is part of the original proto-compiler SKELETON of TWS [17].

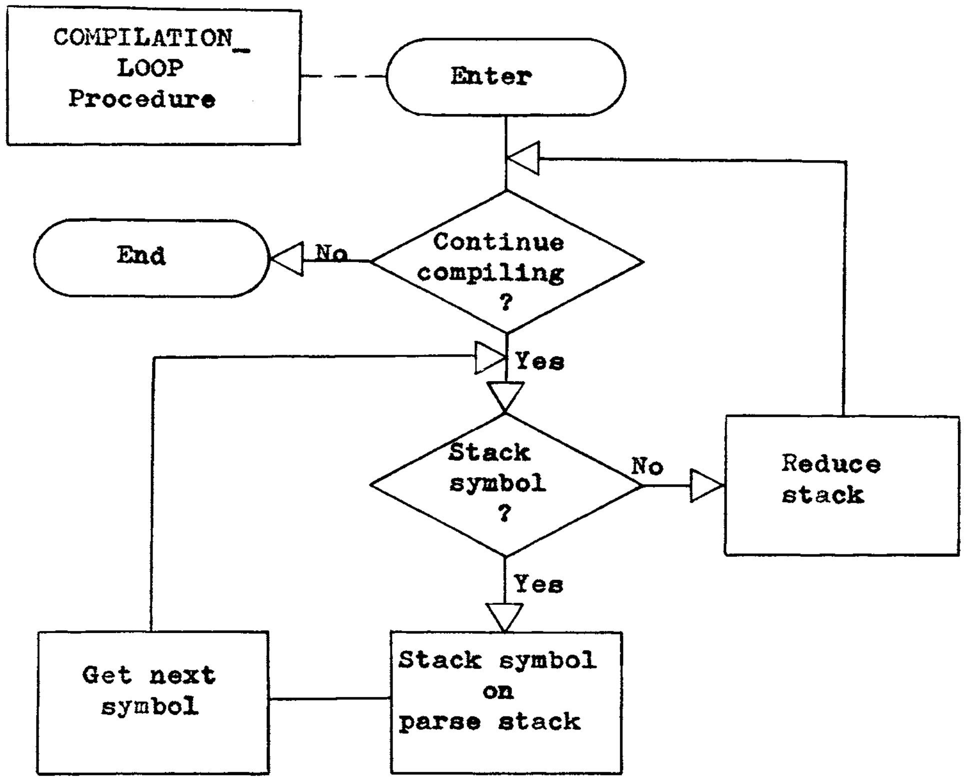


Figure 7. Flow Chart of COMPILATION_LOOP Procedure

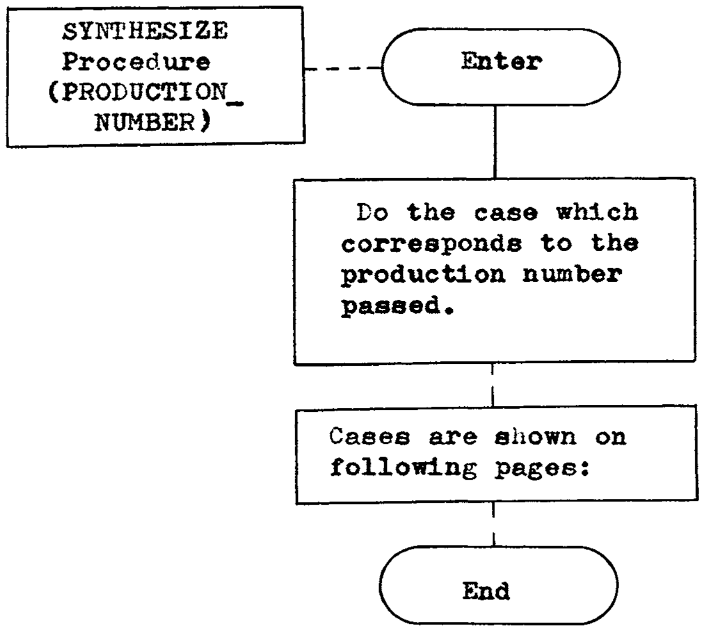

SYNTHESIZE Procedure (PRODUCTION_ NUMBER)
Enter
Do the case which corresponds to the production number passed.
Cases are shown on following pages:
End

Figure 8. Flow Chart of SYNTHESIZE Procedure

Case 1. <PROGRAM> ::= <PROGRAM HEAD> <STATEMENT LIST> <ENDING>.

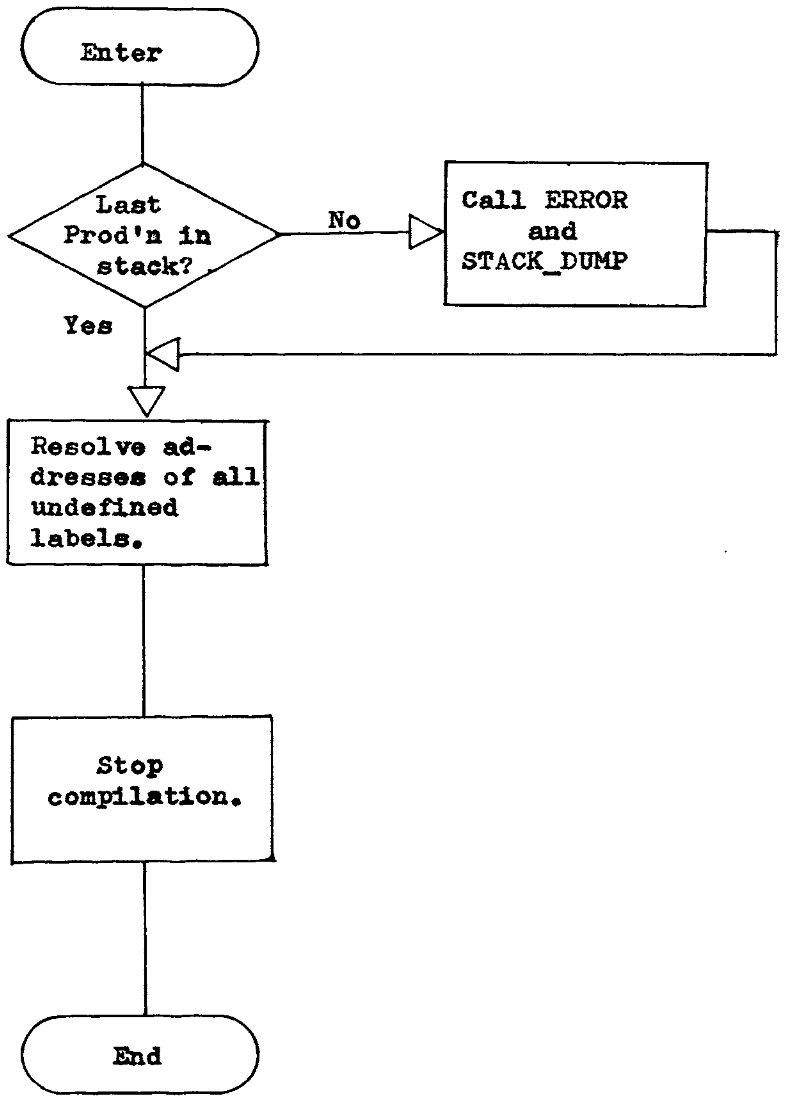
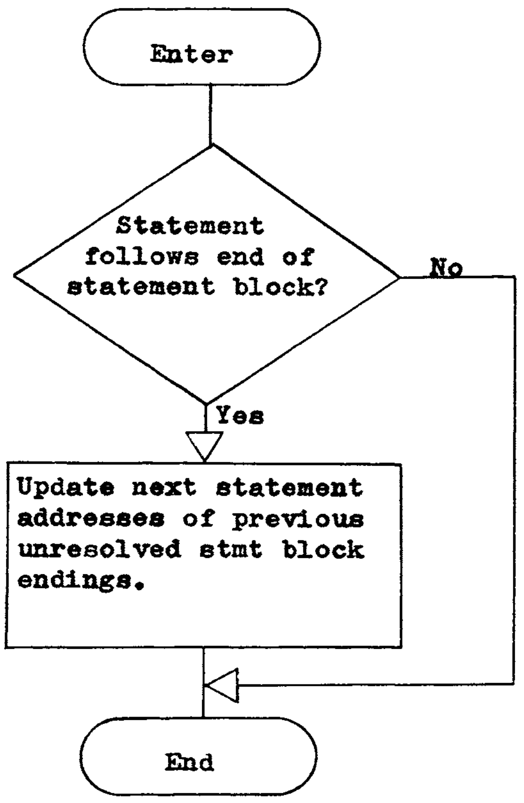

Enter
Last Prod'n in stack?
No
Call ERROR and STACK_DUMP
Yes
Resolve addresses of all undefined labels.
Stop compilation.
End

Figure 8. (Continued)

Case 5. <STATEMENT> ::= <BASIC STMT>


Enter
Statement follows end of statement block?
No
Yes
Update next statement addresses of previous unresolved stmt block endings.
End

Case 7. <BASIC STMT> ::= <TRANSITION STMT>

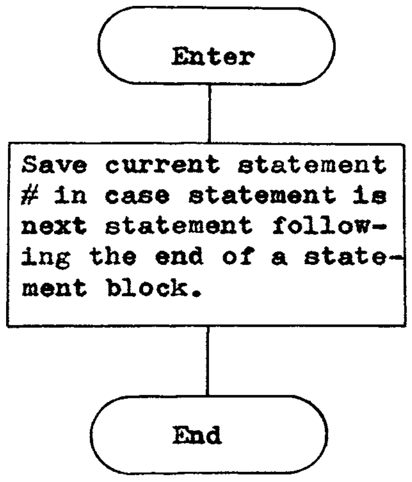
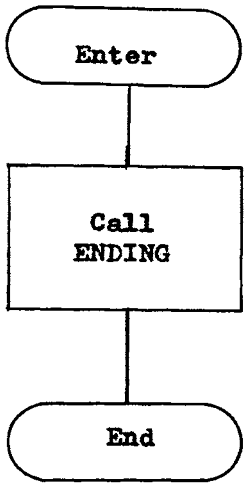

Enter
Save current statement # in case statement is next statement following the end of a statement block.
End

Figure 8. (Continued)

```
Case 11.  <ENDING> ::= END
Case 12.                | <LABEL> END
```



```
Case 13.  <BEGINNING> ::= BEGIN;
Case 14.                   | <LABEL> BEGIN;
```

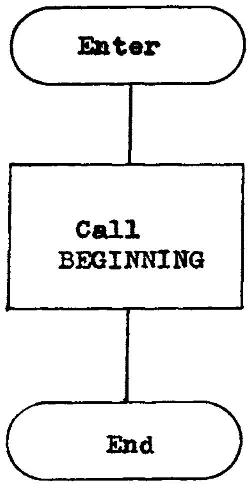


Figure 8. (Continued)

Case 17. <SINGLE LABEL> ::= <IDENTIFIER>

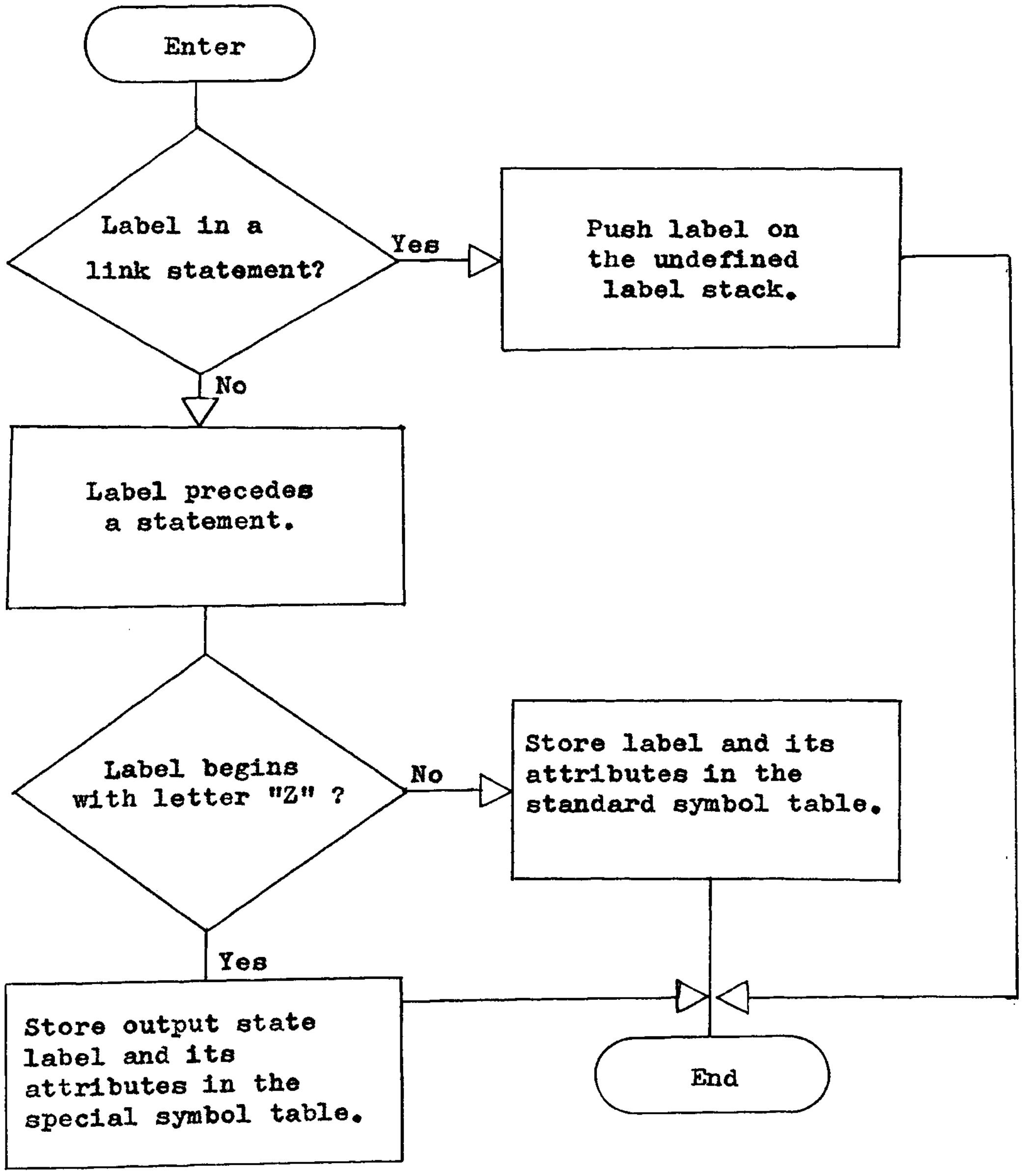

Enter
Label in a link statement?
Yes
Push label on the undefined label stack.
No
Label precedes a statement.
Label begins with letter "Z" ?
No
Store label and its attributes in the standard symbol table.
Yes
Store output state label and its attributes in the special symbol table.
End

Figure 8. (Continued)

Case 18. <SINGLE LABEL> ::= <IDENTIFIER> / <NUMBER>

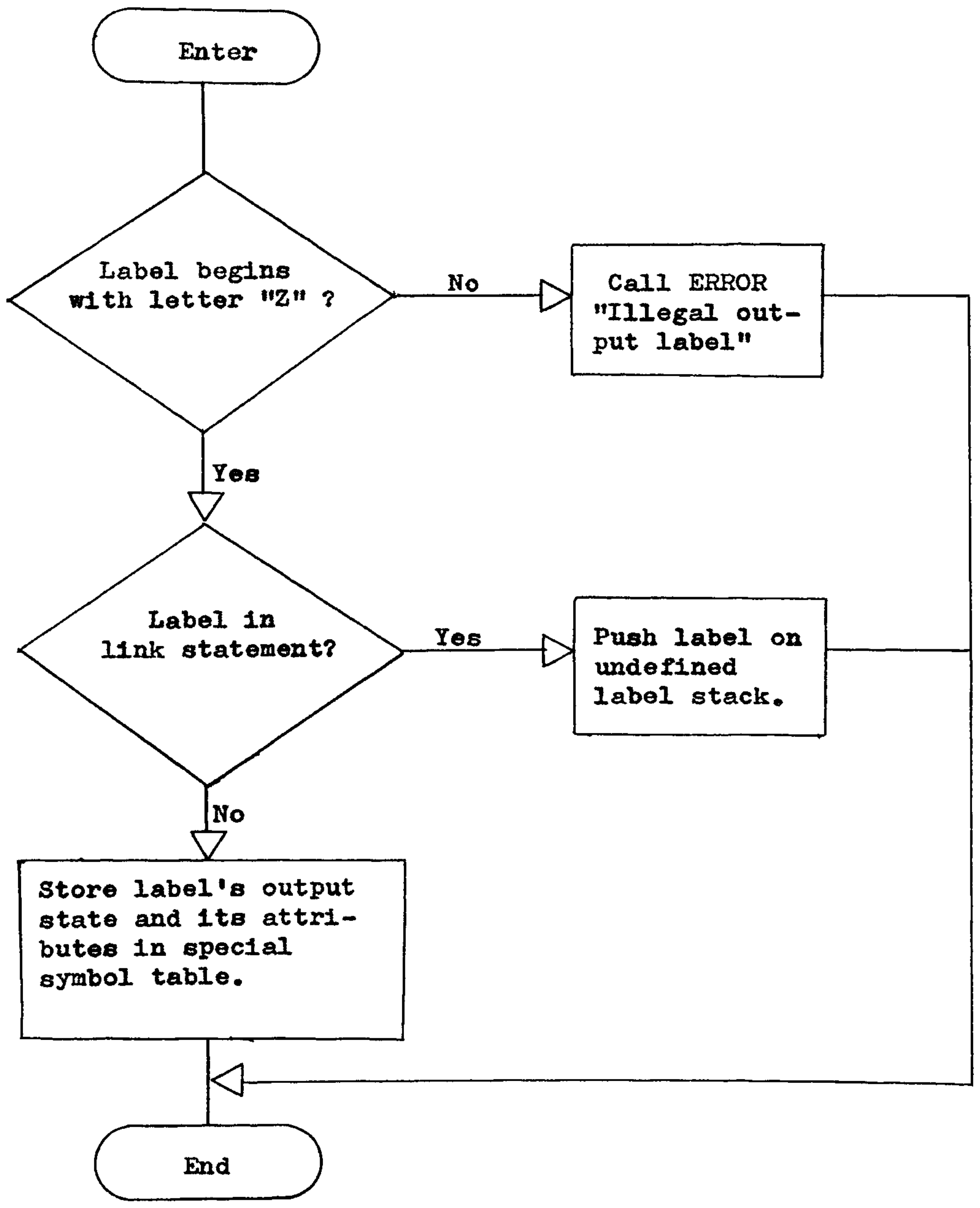

Enter
Label begins with letter "Z" ?
No
Call ERROR "Illegal output label"
Yes
Label in link statement?
Yes
Push label on undefined label stack.
No
Store label's output state and its attributes in special symbol table.
End

Figure 8. (Continued)

Case 19. <LETTER Z> ::= <IDENTIFIER>

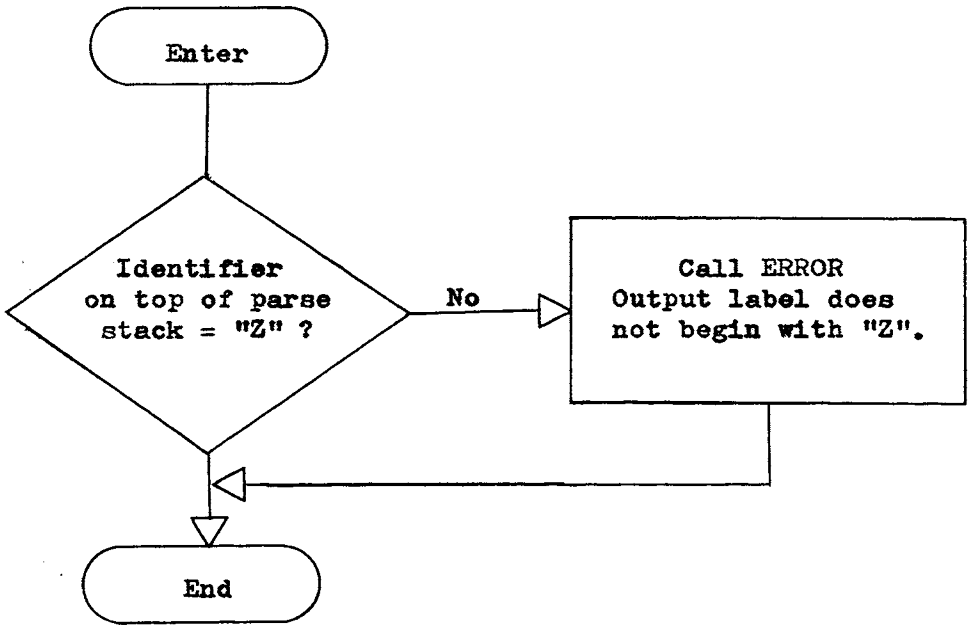

Enter
Identifier on top of parse stack = "Z" ?
No
Call ERROR Output label does not begin with "Z".
End

Figure 8. (Continued)

237261

Case 22. <OUTPUT CODE> ::= <NUMBER>

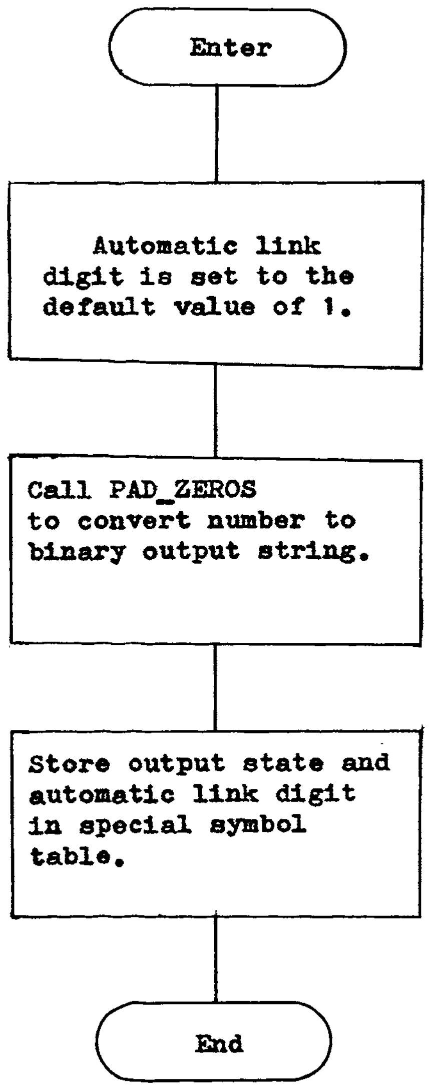

Enter
Automatic link digit is set to the default value of 1.
Call PAD_ZEROS to convert number to binary output string.
Store output state and automatic link digit in special symbol table.
End

Figure 8. (Continued)

Case 23. <OUTPUT CODE> ::= <NUMBER> / <NUMBER>

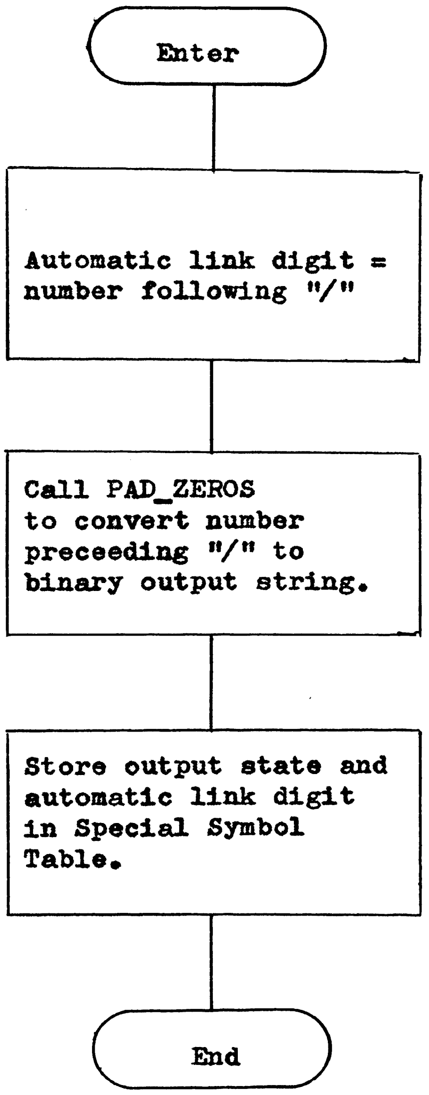
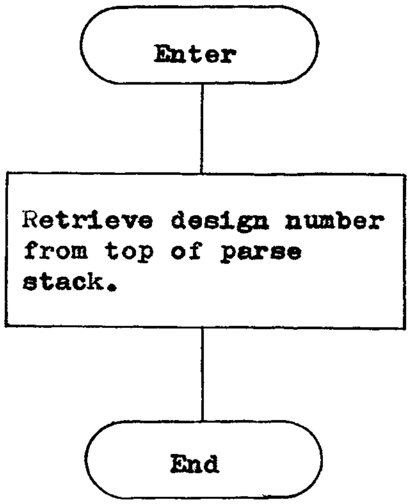

Enter
Automatic link digit = number following "/"
Call PAD_ZEROS to convert number preceeding "/" to binary output string.
Store output state and automatic link digit in Special Symbol Table.
End

Figure 8. (Continued)

Case 30. <DESIGN NUMBER> ::= <NUMBER>



Case 31. <DESIGNERS NAME> ::= <IDENTIFIER>
Case 32. | <DESIGNERS NAME> <IDENTIFIER>

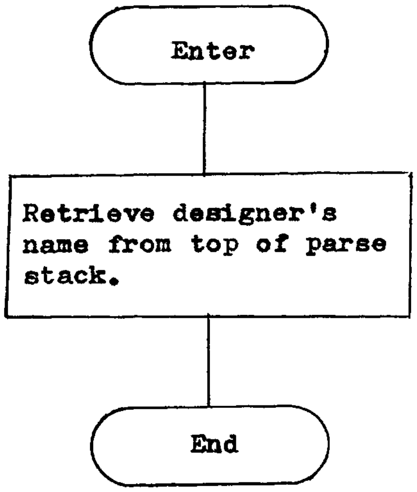
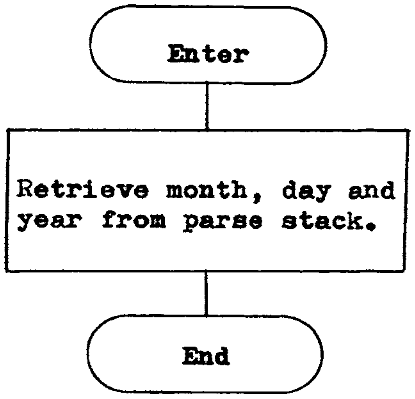
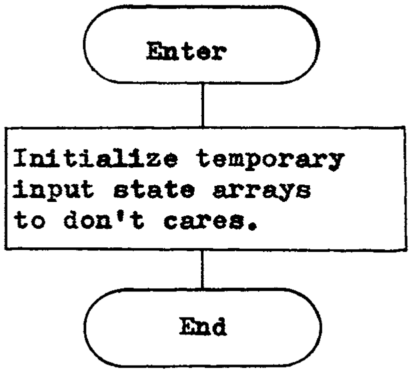


Figure 8. (Continued)

Case 33. <DATE> ::= <IDENTIFIER> <NUMBER> , <NUMBER>



Case 36. <DECLARATION TYPE> ::= <INPUT DECLARATION>



Case 42. <VARIABLE DEFN> ::= <IDENTIFIER>
Case 44. <IDENTIFIER1> ::= <IDENTIFIER>

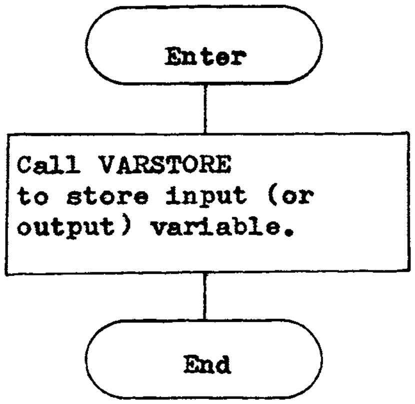
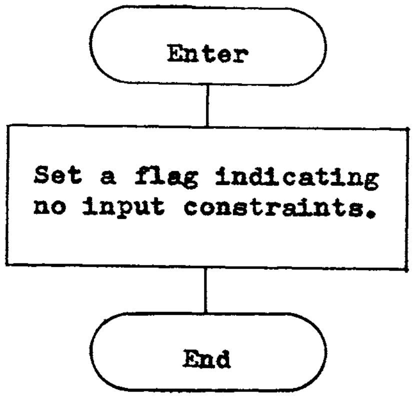
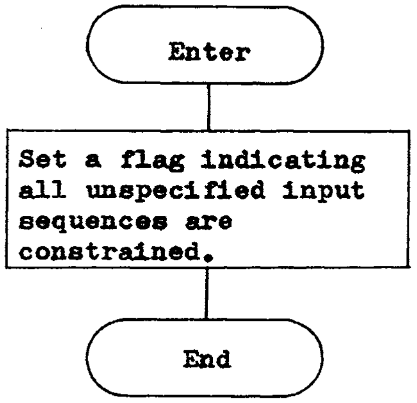


Figure 8. (Continued)

Case 48. <CONSTRAINTS> ::= NONE



Case 49. <CONSTRAINTS> ::= AUS


Enter
Set a flag indicating all unspecified input sequences are constrained.
End

Case 50. <CONSTRAINTS> ::= SIC

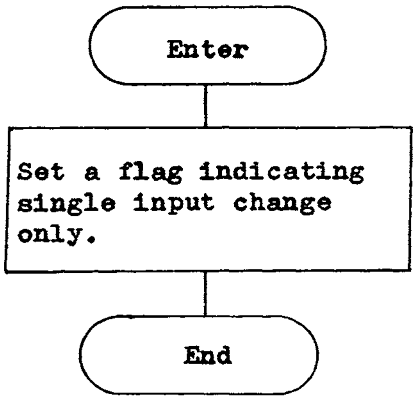
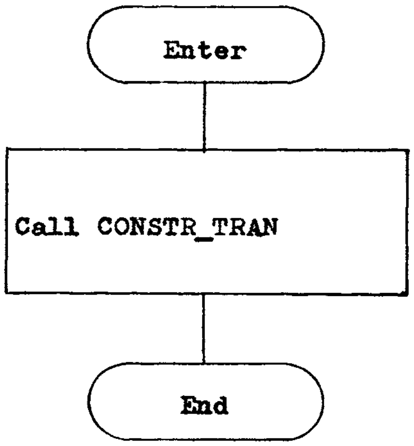


Figure 8. (Continued)

```
Case 51.  <CONSTRAINTS> ::= <TRANSITION TERM>
Case 53.                  | <CONSTRAINTS> <,2>
                            <TRANSITION TERM>
```



```
Case 52.  <CONSTRAINTS> ::= <LEVEL FACTOR>
Case 54.                  | <CONSTRAINTS> <,2>
                            <LEVEL FACTOR>
```

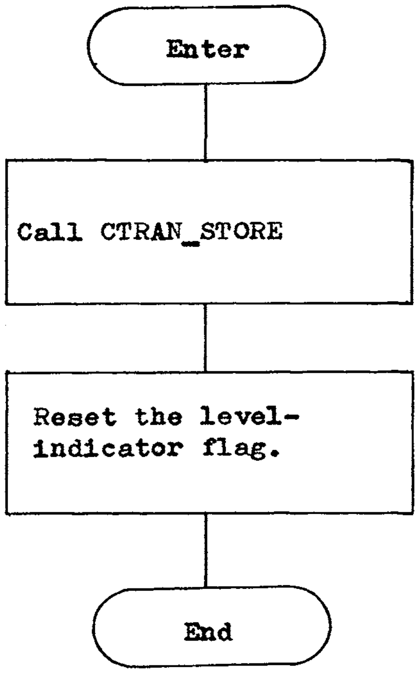


Figure 8. (Continued)

Case 55. <TRANSITION EXPRESSION> ::= <TRANSITION TERM>

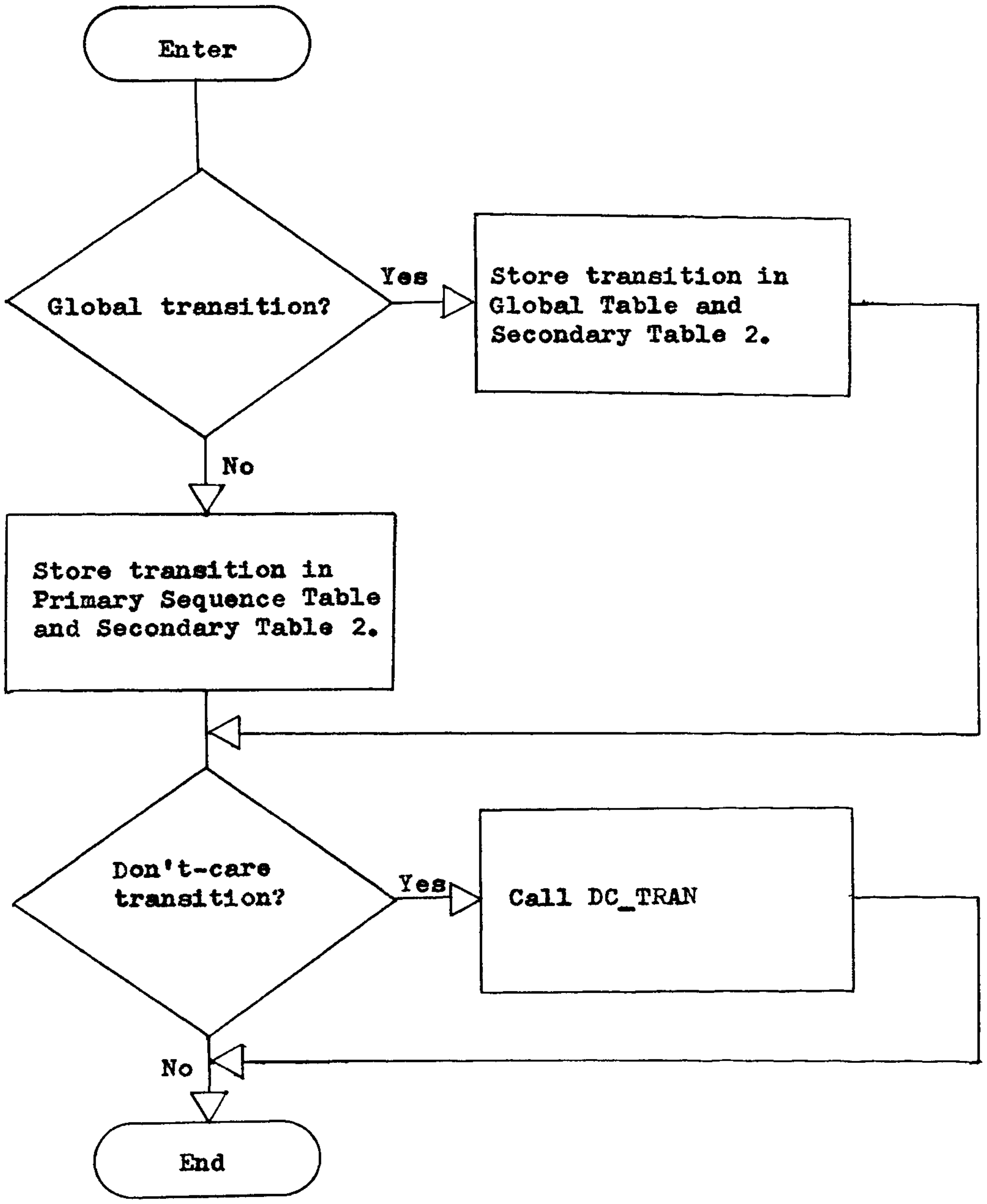

Enter
Global transition?
Yes
Store transition in Global Table and Secondary Table 2.
No
Store transition in Primary Sequence Table and Secondary Table 2.
Don't-care transition?
Yes
Call DC_TRAN
No
End

Figure 8. (Continued)

Case 56. <TRANSITION EXPRESSION> ::= <TRANSITION EXPRESSION>
+ <TRANSITION TERM>

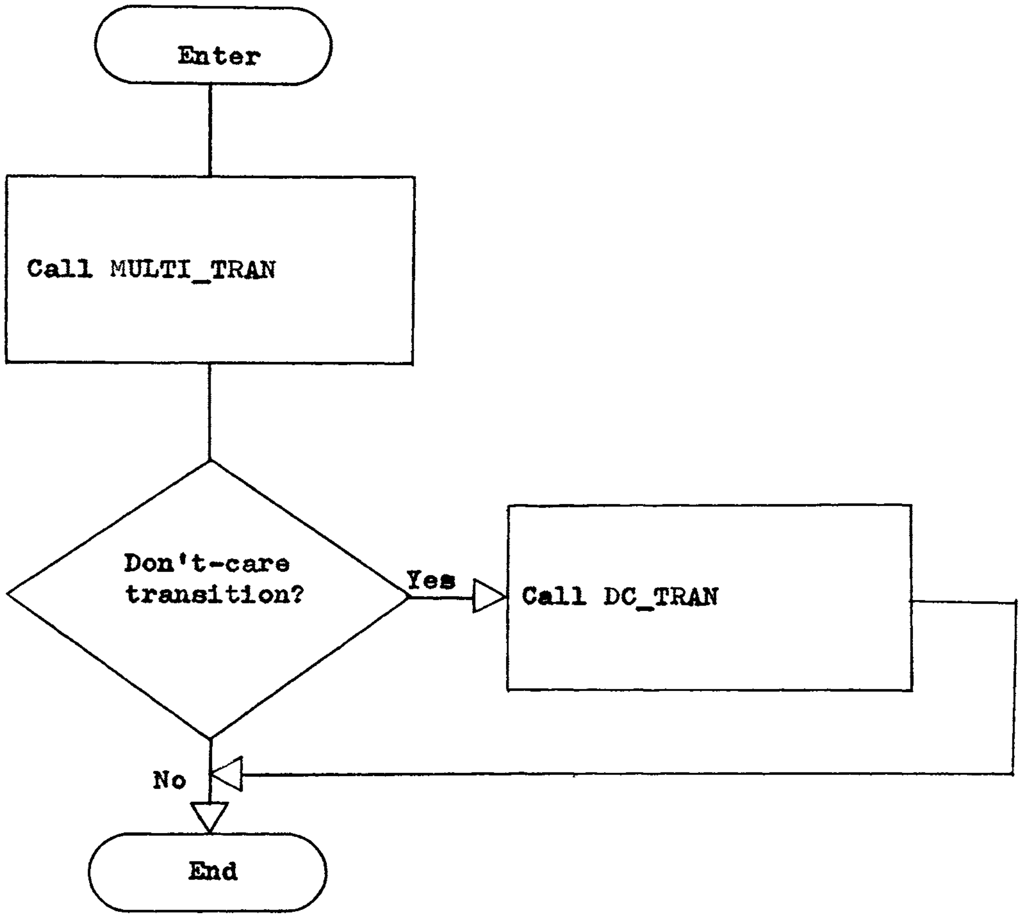


Figure 8. (Continued)

Case 58. <DUMMY TERM> ::= LINKTEST
Case 59. | LK'T

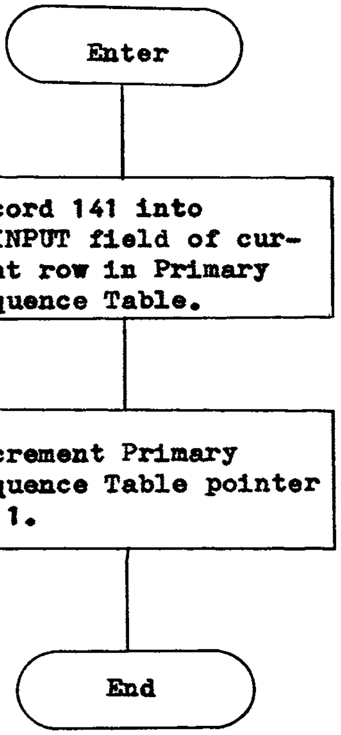

Enter
Record 141 into B_INPUT field of current row in Primary Sequence Table.
Increment Primary Sequence Table pointer by 1.
End

Figure 8. (Continued)

Case 76. <LEVEL> ::= <NUMBER>

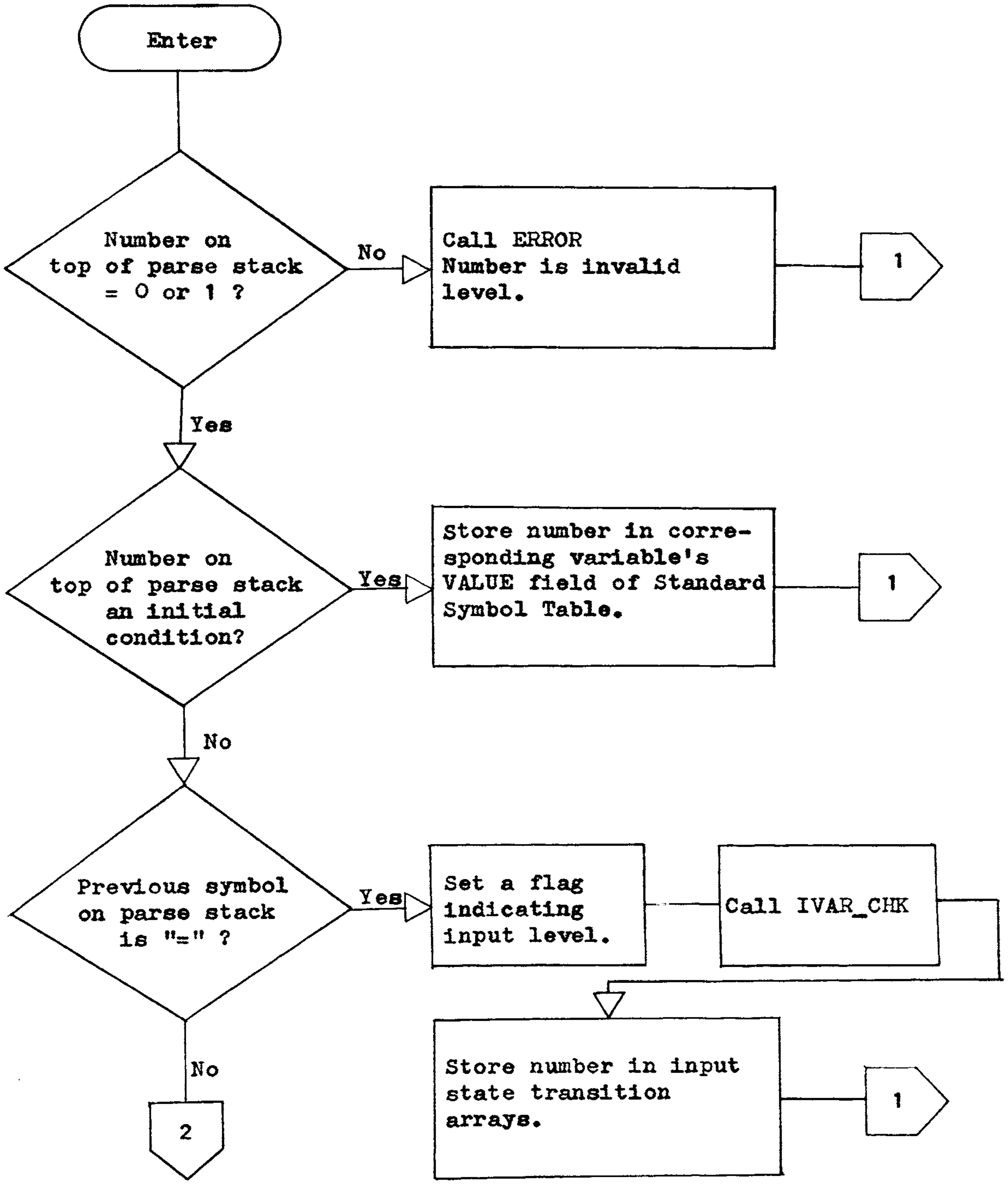

Enter
Number on top of parse stack = 0 or 1 ?
No
Call ERROR Number is invalid level.
1
Yes
Number on top of parse stack an initial condition?
Yes
Store number in corresponding variable's VALUE field of Standard Symbol Table.
1
No
Previous symbol on parse stack is "=" ?
Yes
Set a flag indicating input level.
Call IVAR_CHK
Store number in input state transition arrays.
1
No
2

Figure 8. (Continued)

Case 76. (Continued)

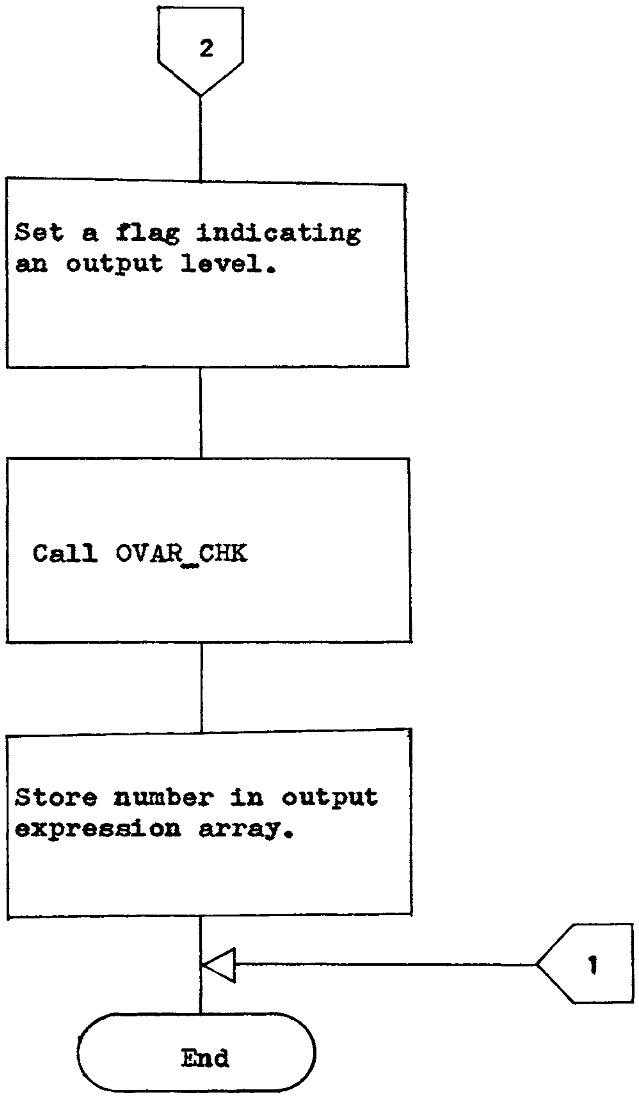

2
Set a flag indicating an output level.
Call OVAR_CHK
Store number in output expression array.
1
End

Figure 8. (Continued)

Case 77. <TRANSITION> ::= <NUMBER> - > <NUMBER>

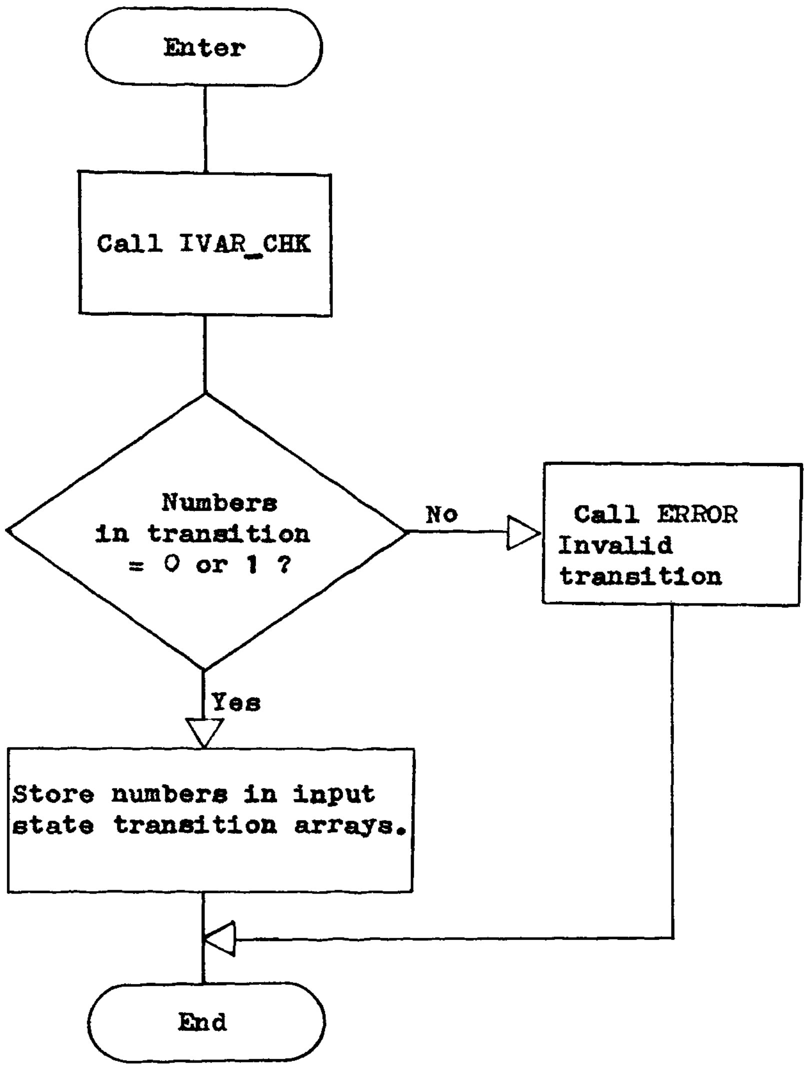

Enter
Call IVAR_CHK
Numbers in transition = 0 or 1 ?
No
Call ERROR Invalid transition
Yes
Store numbers in input state transition arrays.
End

Figure 8. (Continued)

Case 78. <SHORT TRAN> ::= - > <NUMBER>

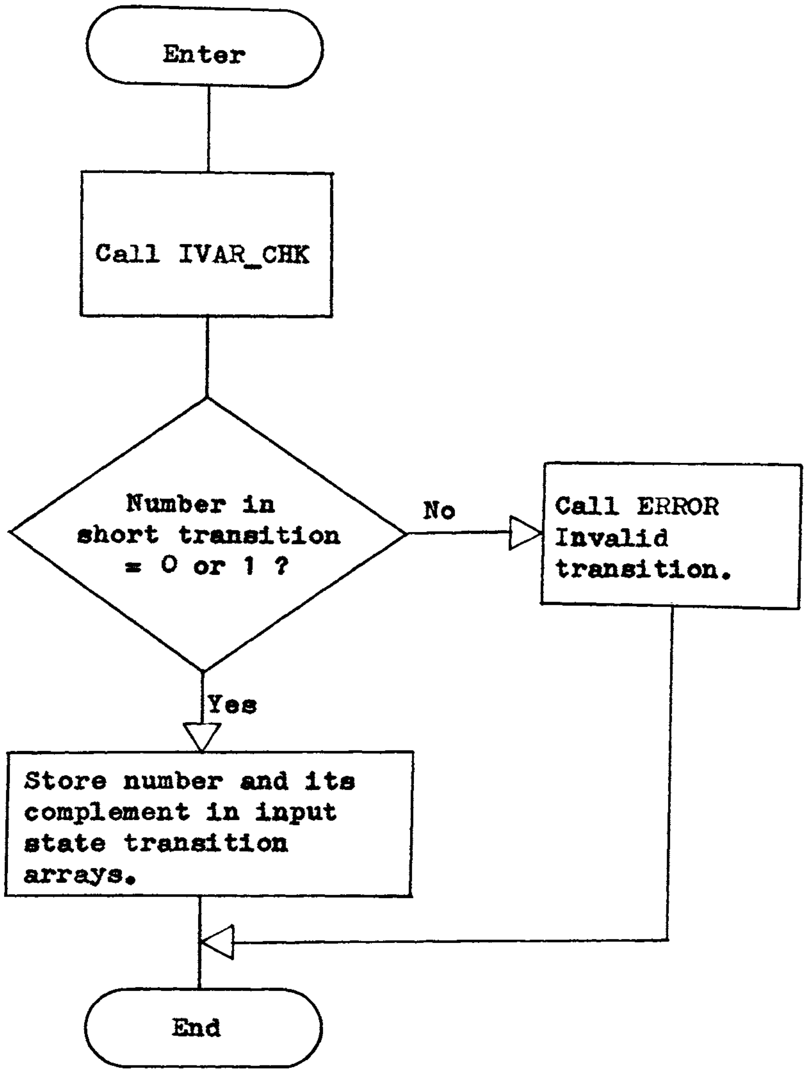

Enter
Call IVAR_CHK
Number in short transition = 0 or 1 ?
No
Call ERROR Invalid transition.
Yes
Store number and its complement in input state transition arrays.
End

Figure 8. (Continued)

Case 79. <SHORT TRAN> ::= - > ?

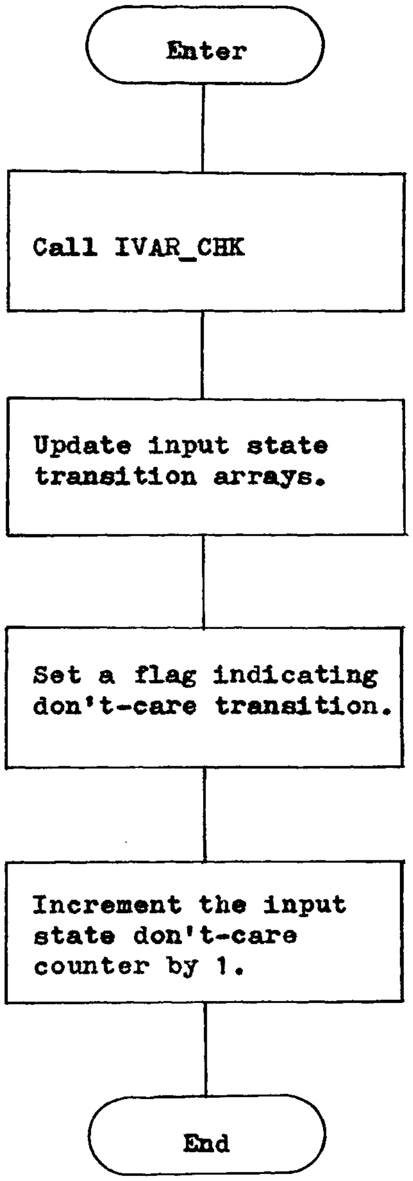
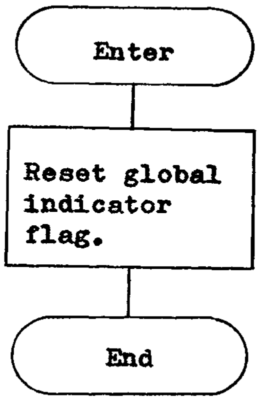

Enter
Call IVAR_CHK
Update input state transition arrays.
Set a flag indicating don't-care transition.
Increment the input state don't-care counter by 1.
End

Figure 8. (Continued)

Case 83. <GLOBAL DCL> ::= GLOBAL : <LIST>



Case 84. <START STMT> ::= START ;

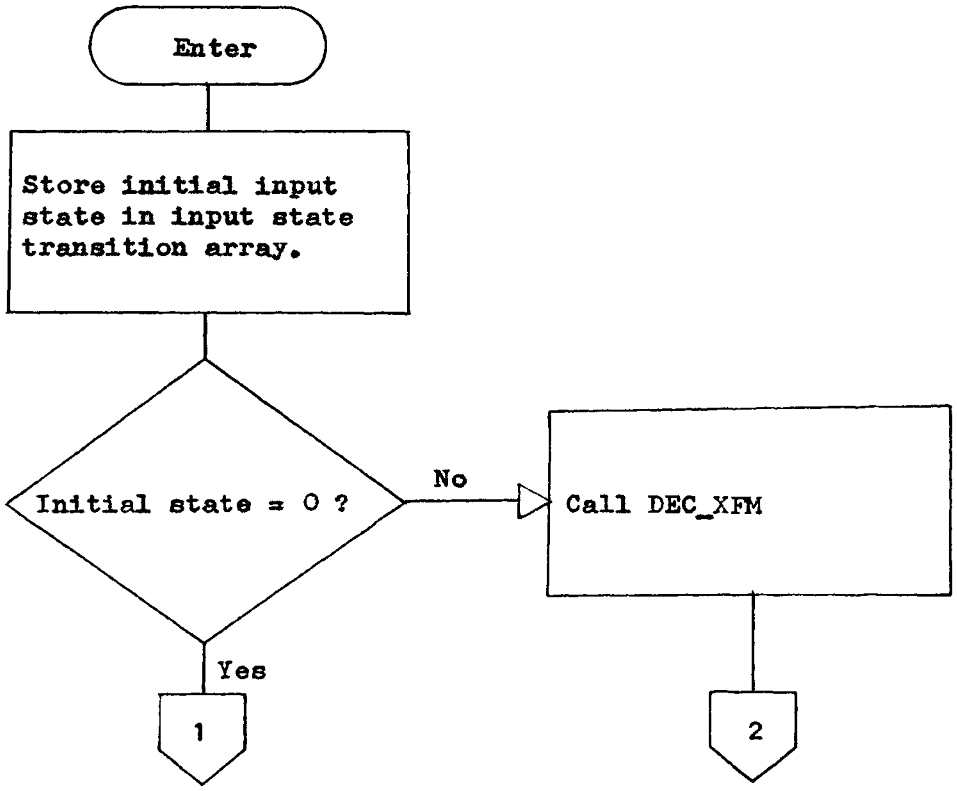


Figure 8. (Continued)

Case 84. (Continued)

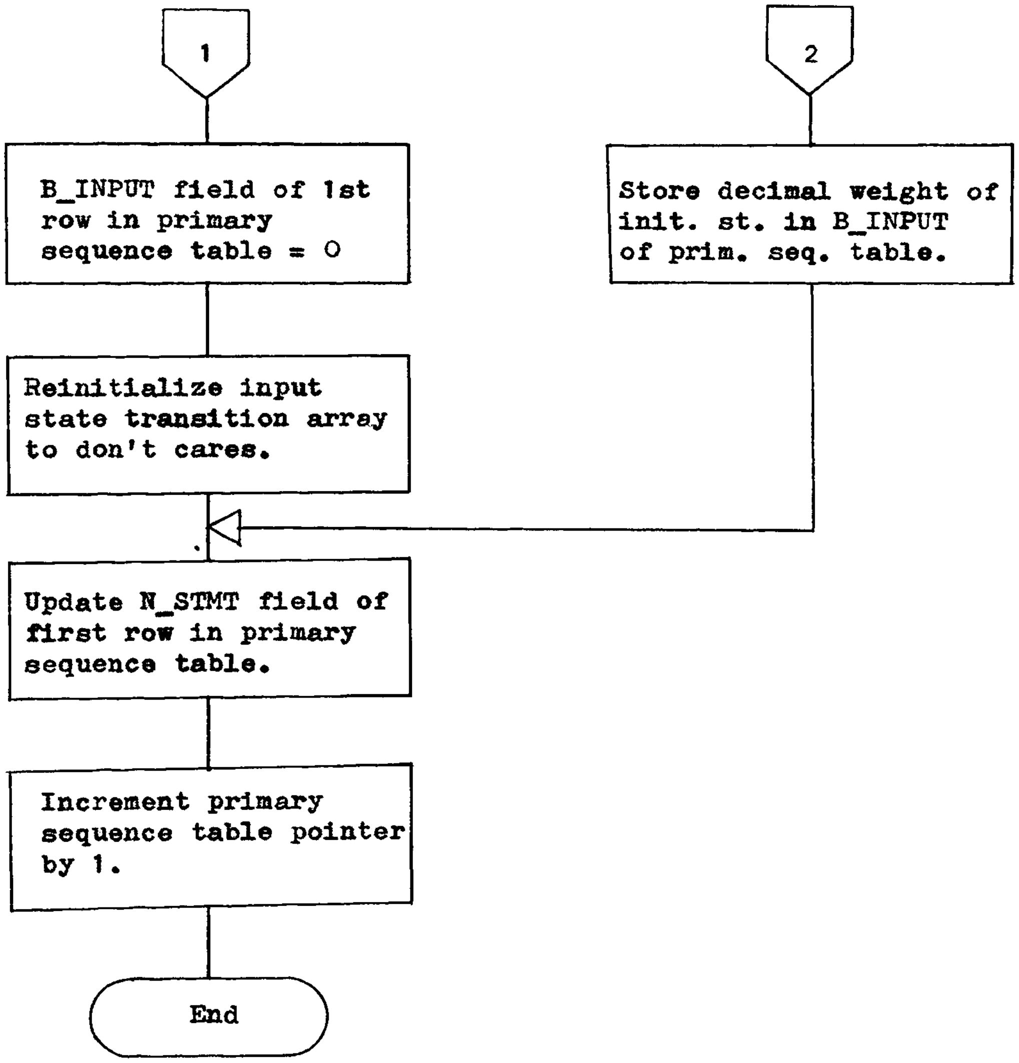
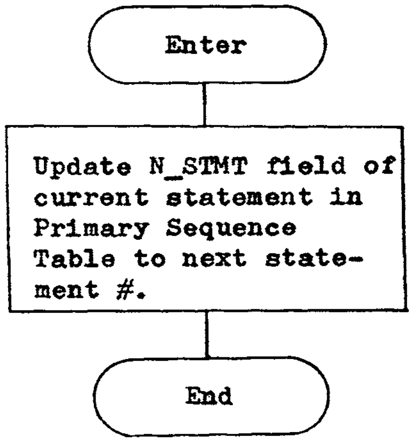

1
2
B_INPUT field of 1st row in primary sequence table = 0
Store decimal weight of init. st. in B_INPUT of prim. seq. table.
Reinitialize input state transition array to don't cares.
Update N_STMT field of first row in primary sequence table.
Increment primary sequence table pointer by 1.
End

Figure 8. (Continued)

Case 86. <TRANSITION STMT> ::= <BASIC TRAN STMT>


Enter
Update N_STMT field of current statement in Primary Sequence Table to next statement #.
End

Case 91. <AUTO LINK> ::= /

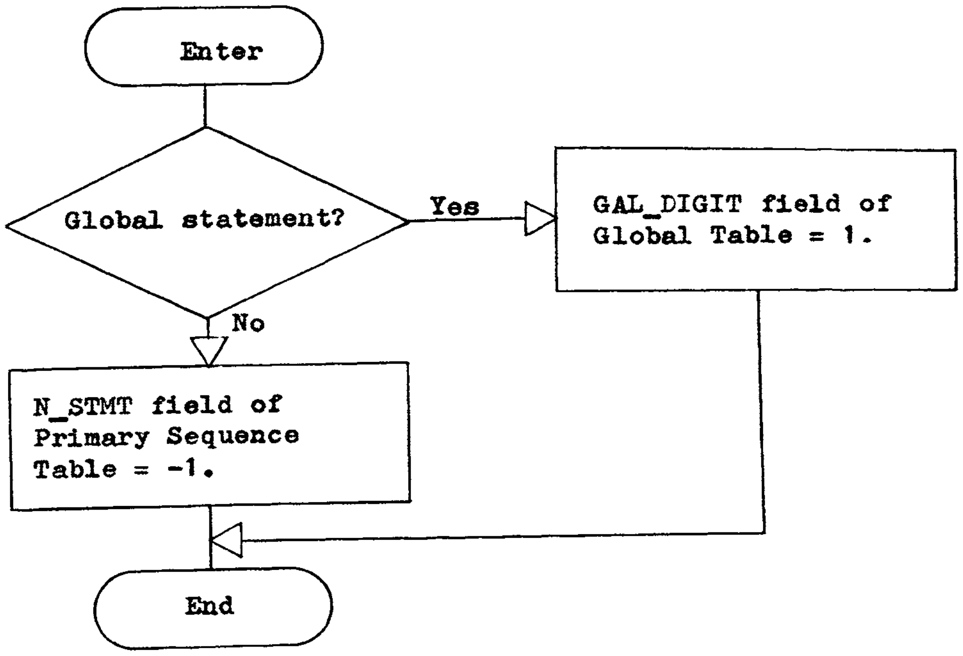

Enter
Global statement?
Yes
GAL_DIGIT field of Global Table = 1.
No
N_STMT field of Primary Sequence Table = -1.
End

Figure 8. (Continued)

Case 92. <AUTO LINK> ::= / <NUMBER>

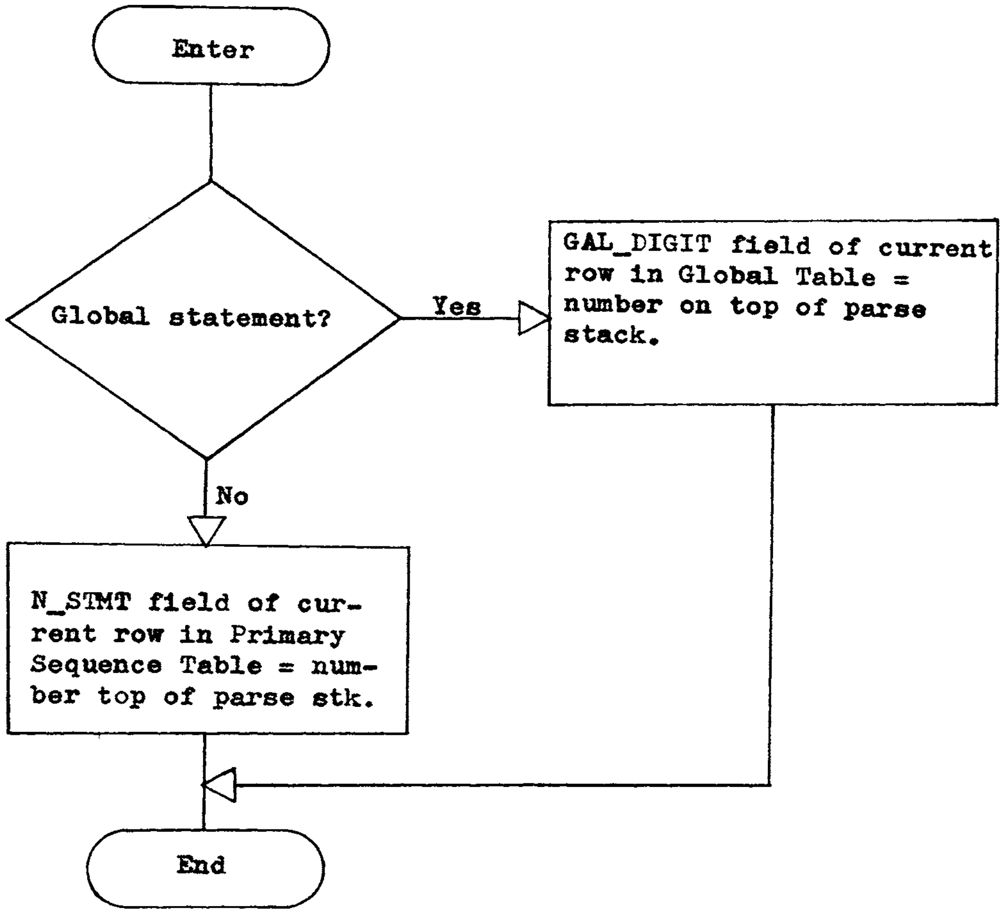

Enter
Global statement?
Yes
GAL_DIGIT field of current row in Global Table = number on top of parse stack.
No
N_STMT field of current row in Primary Sequence Table = number top of parse stk.
End

Figure 8. (Continued)

Case 93. <OUTPUT CHANGE> ::= <IDENTIFIER> <REPLACEMENT OP> <OUTPUT EXPRESSION>

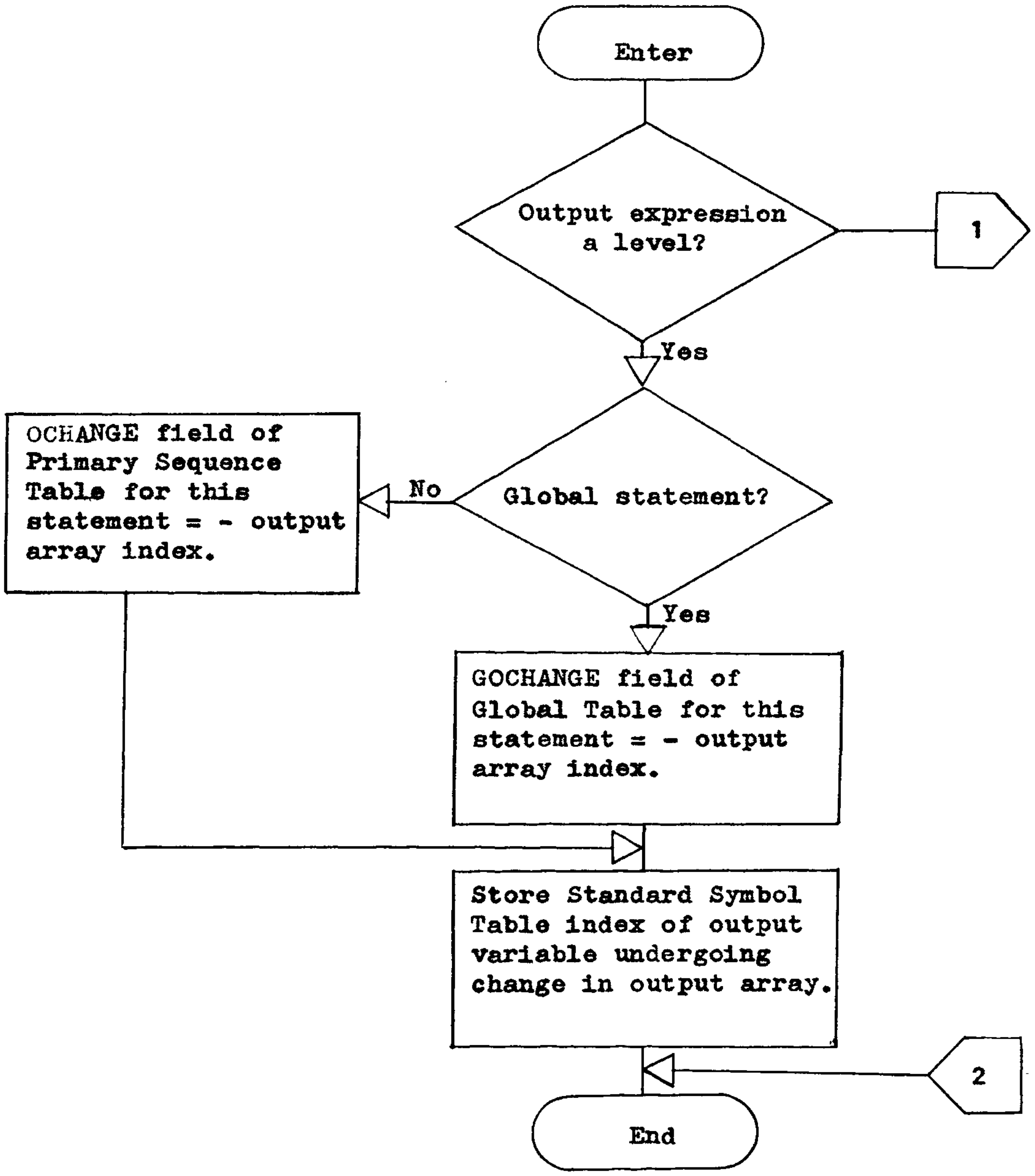

Enter
Output expression a level?
1
Yes
OCHANGE field of Primary Sequence Table for this statement = - output array index.
No
Global statement?
Yes
GOCHANGE field of Global Table for this statement = - output array index.
Store Standard Symbol Table index of output variable undergoing change in output array.
2
End

Figure 8. (Continued)

Case 93. (Continued)

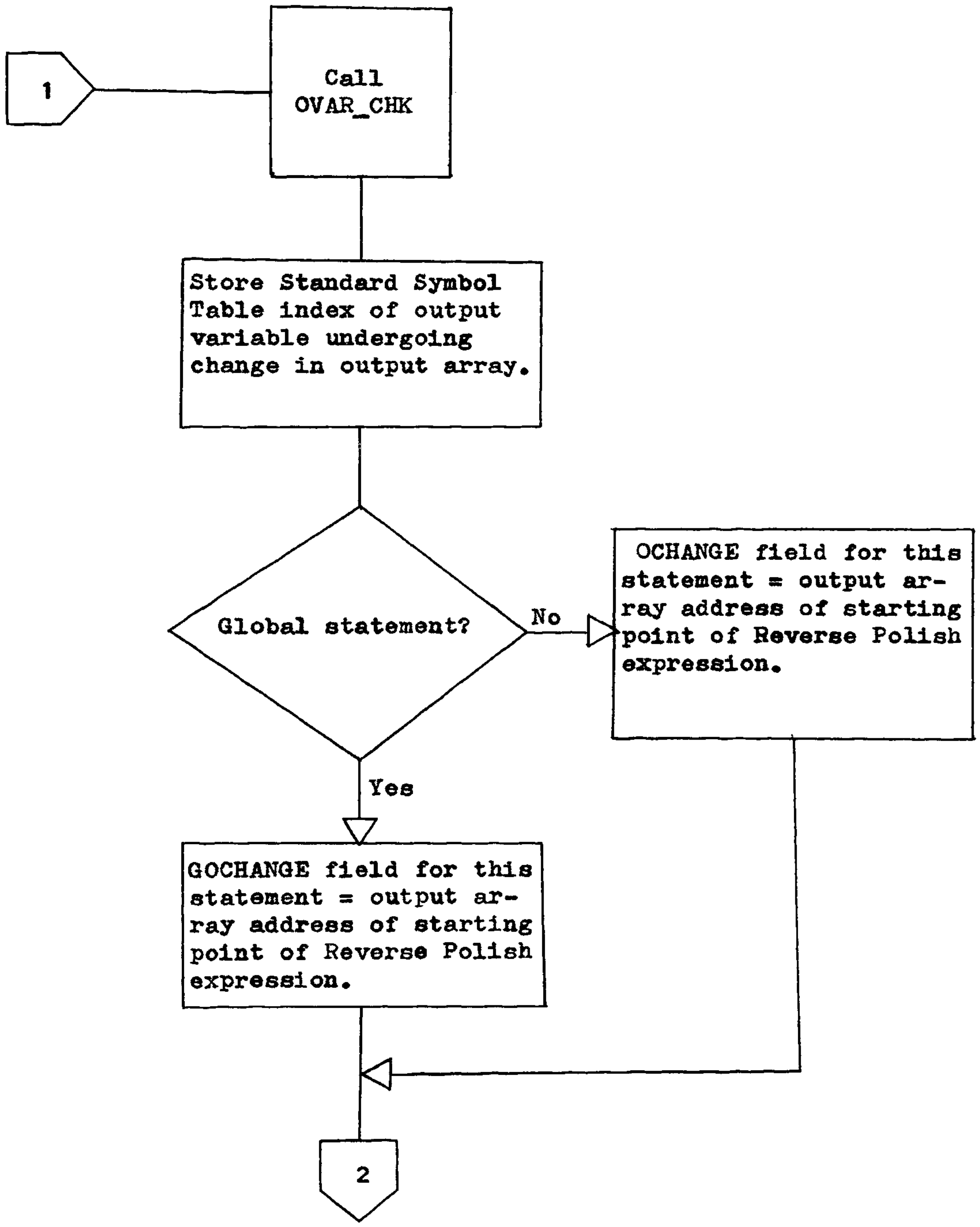

1
Call OVAR_CHK
Store Standard Symbol Table index of output variable undergoing change in output array.
Global statement?
No
OCHANGE field for this statement = output array address of starting point of Reverse Polish expression.
Yes
GOCHANGE field for this statement = output array address of starting point of Reverse Polish expression.
2

Figure 8. (Continued)

Case 94. <OUTPUT CHANGE> ::= <OUTPUT CHANGE> <,1> <IDENTIFIER> <REPLACEMENT OP> <OUTPUT EXPRESSION>

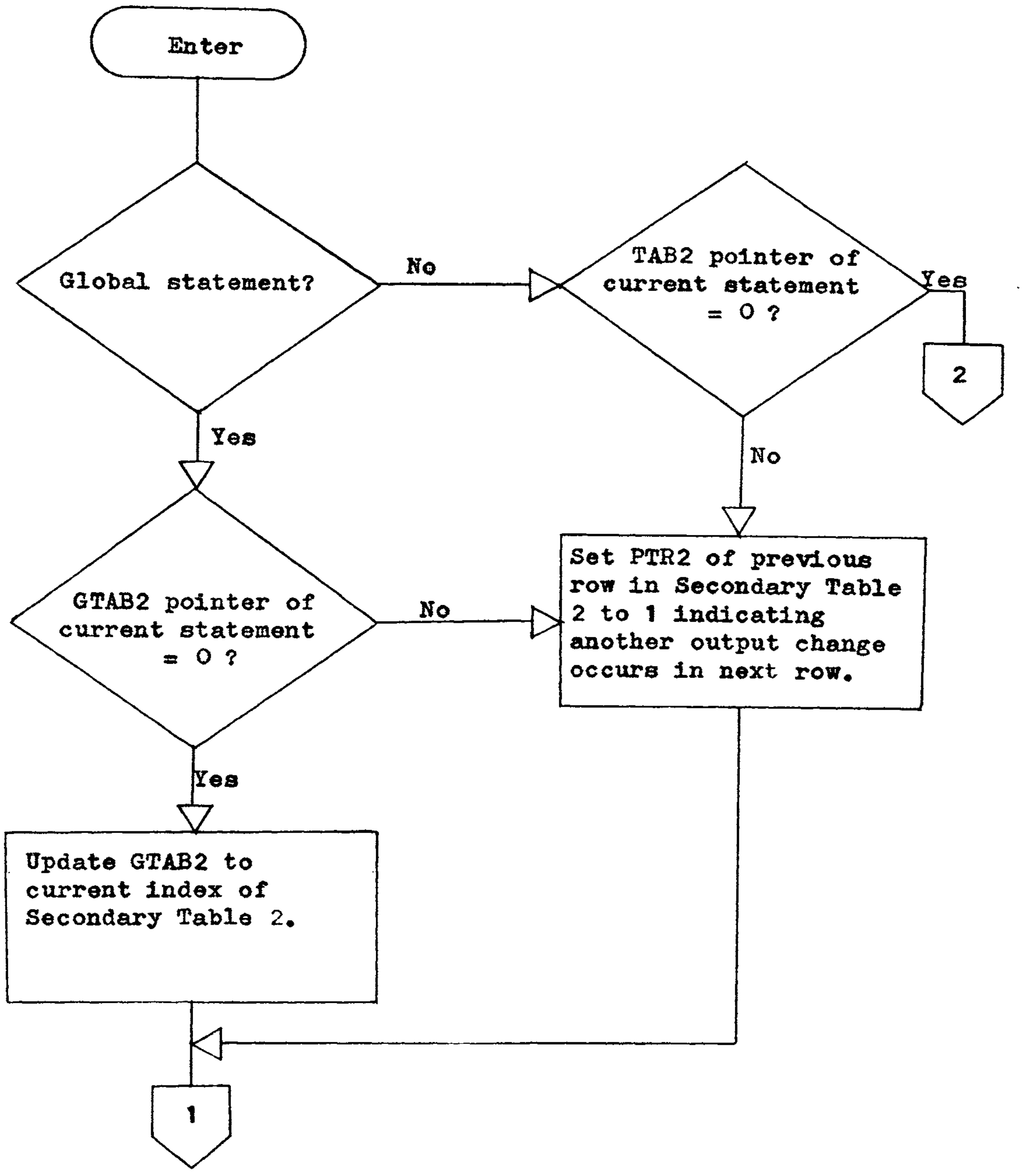

Enter
Global statement?
No
TAB2 pointer of current statement = 0 ?
Yes
2
Yes
No
GTAB2 pointer of current statement = 0 ?
No
Set PTR2 of previous row in Secondary Table 2 to 1 indicating another output change occurs in next row.
Yes
Update GTAB2 to current index of Secondary Table 2.
1

Figure 8. (Continued)

Case 94. (Continued)

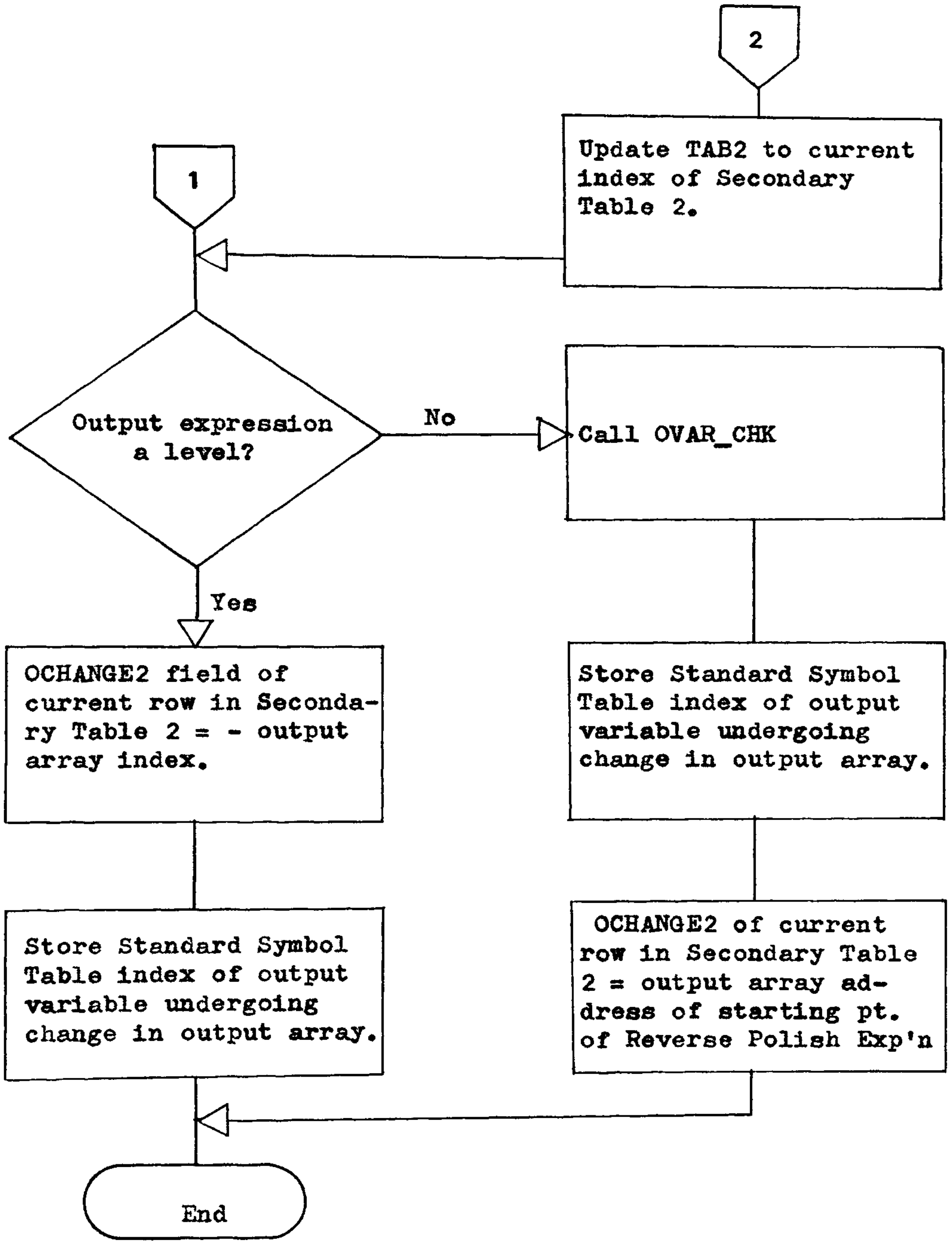
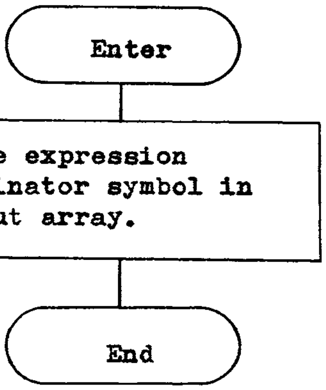

2
Update TAB2 to current index of Secondary Table 2.
1
Output expression a level?
No
Call OVAR_CHK
Yes
OCHANGE2 field of current row in Secondary Table 2 = - output array index.
Store Standard Symbol Table index of output variable undergoing change in output array.
Store Standard Symbol Table index of output variable undergoing change in output array.
OCHANGE2 of current row in Secondary Table 2 = output array address of starting pt. of Reverse Polish Exp'n
End

Figure 8. (Continued)

Case 97. <OUTPUT EXPRESSION> ::= <BOOL EXPR>



Case 104. <LOG PRIMARY> ::= <IDENTIFIER>

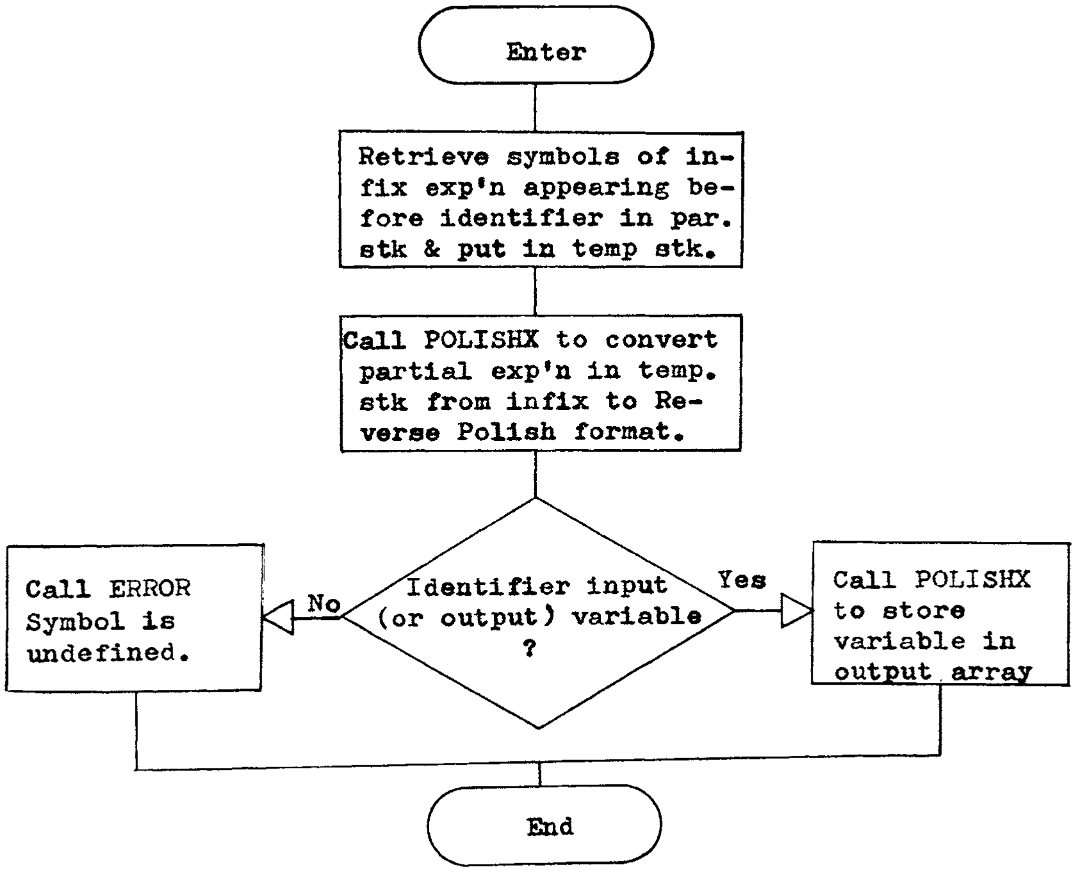
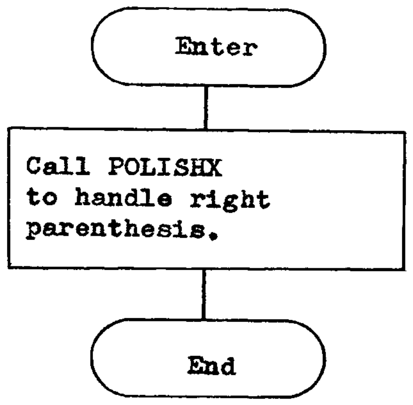
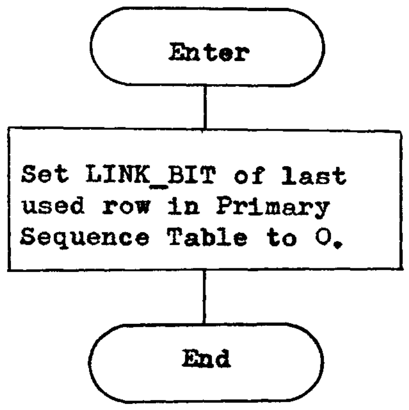

Enter
Retrieve symbols of infix exp'n appearing before identifier in par. stk & put in temp stk.
Call POLISHX to convert partial exp'n in temp. stk from infix to Reverse Polish format.
Identifier input (or output) variable ?
No
Call ERROR Symbol is undefined.
Yes
Call POLISHX to store variable in output array
End

Figure 8. (Continued)

Case 105. <LOG PRIMARY> ::= <(2> <BOOL EXPR>)



Case 106. <LINK STMT> ::= LINK <PARAMETER LIST>


Enter
Set LINK_BIT of last used row in Primary Sequence Table to 0.
End

Case 107. <PARAMETER LIST> ::= <NO TESTS>

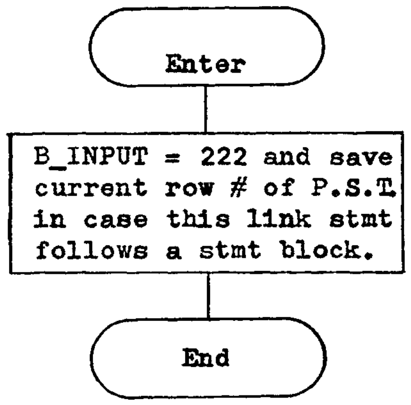
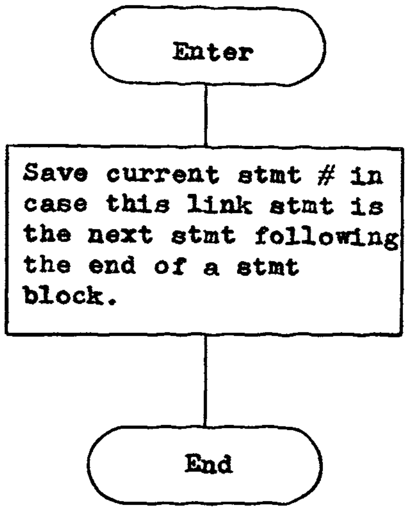

Enter
B_INPUT = 222 and save current row # of P.S.T. in case this link stmt follows a stmt block.
End

Figure 8. (Continued)

Case 115. <1 TEST> ::= <(1> <TEST CONDITION>


Enter
Save current stmt # in case this link stmt is the next stmt following the end of a stmt block.
End

Case 128. <TEST CONDITION> ::= <TRANSITION EXPRESSION>

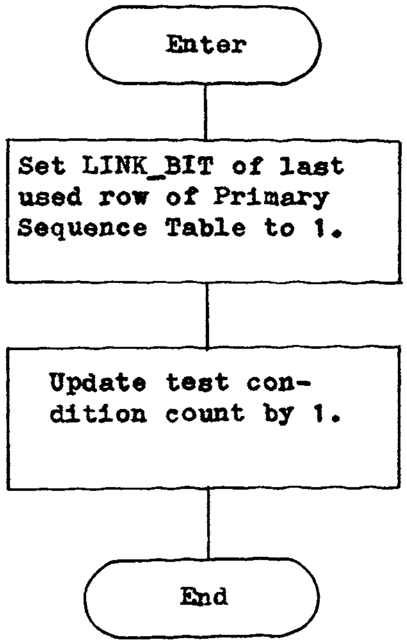

Enter
Set LINK_BIT of last used row of Primary Sequence Table to 1.
Update test condition count by 1.
End

Figure 8. (Continued)

Case 129. <TEST CONDITION> ::= <LEVEL FACTOR>

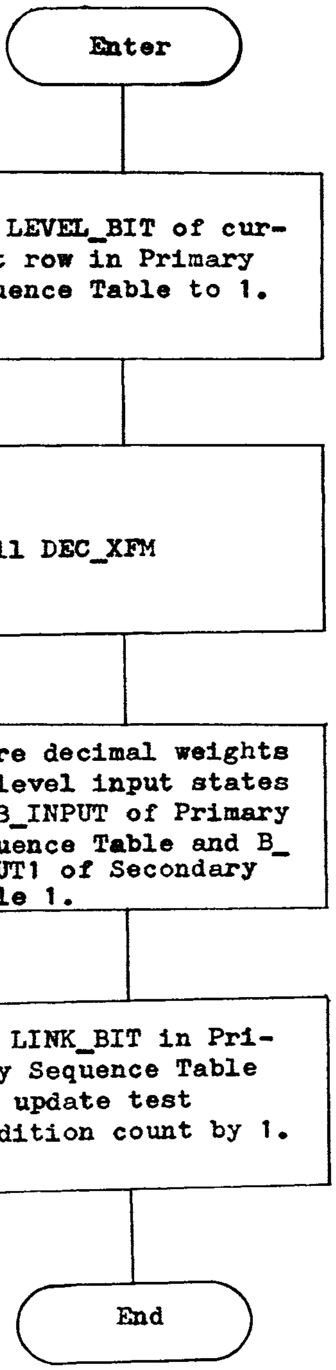
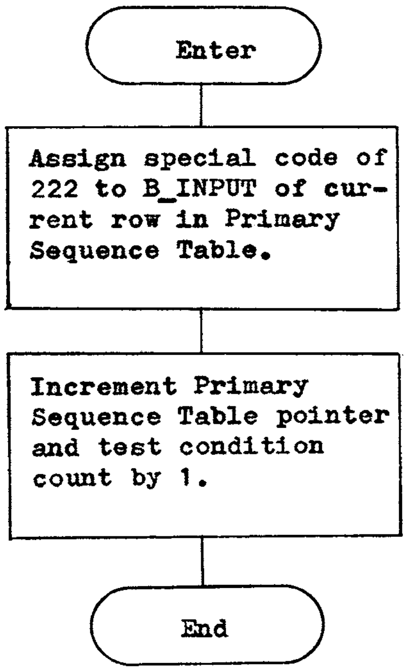

Enter
Set LEVEL_BIT of current row in Primary Sequence Table to 1.
Call DEC_XFM
Store decimal weights of level input states in B_INPUT of Primary Sequence Table and B_INPUT1 of Secondary Table 1.
Set LINK_BIT in Primary Sequence Table and update test condition count by 1.
End

Figure 8. (Continued)

Case 130. <TEST CONDITION> ::= ELSE


Enter
Assign special code of 222 to B_INPUT of current row in Primary Sequence Table.
Increment Primary Sequence Table pointer and test condition count by 1.
End

Case 131. <LIST STMT> ::= LIST <LIST>

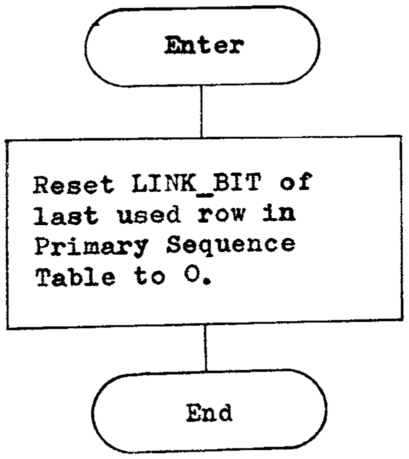
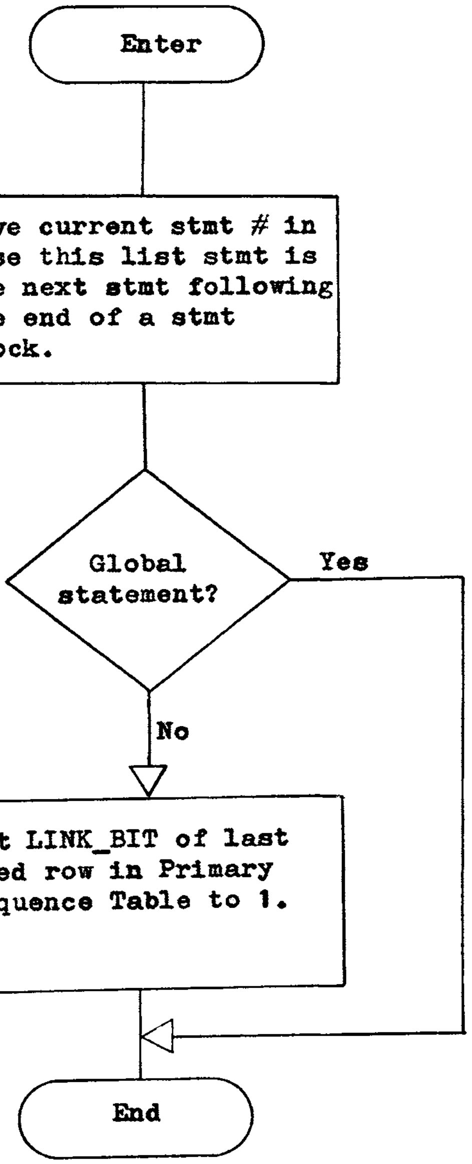

Enter
Reset LINK_BIT of last used row in Primary Sequence Table to 0.
End

Figure 8. (Continued)

Case 132. <LIST> ::= <AUTO LINK TRAN STMT>



Enter
Save current stmt # in case this list stmt is the next stmt following the end of a stmt block.
Global statement?
Yes
No
Set LINK_BIT of last used row in Primary Sequence Table to 1.
End

Figure 8. (Continued)

Case 133. <LIST> ::= <LIST> <,1> <AUTO LINK TRAN STMT>

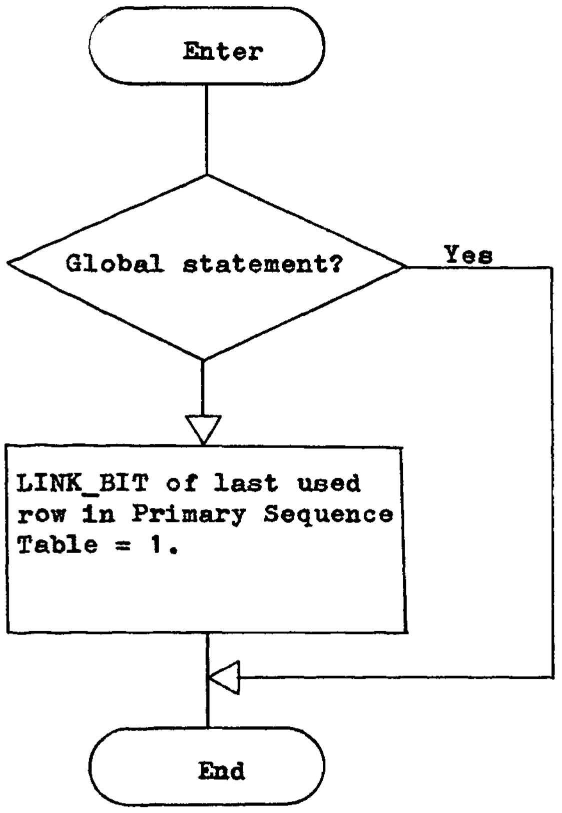


Figure 8. (Continued)

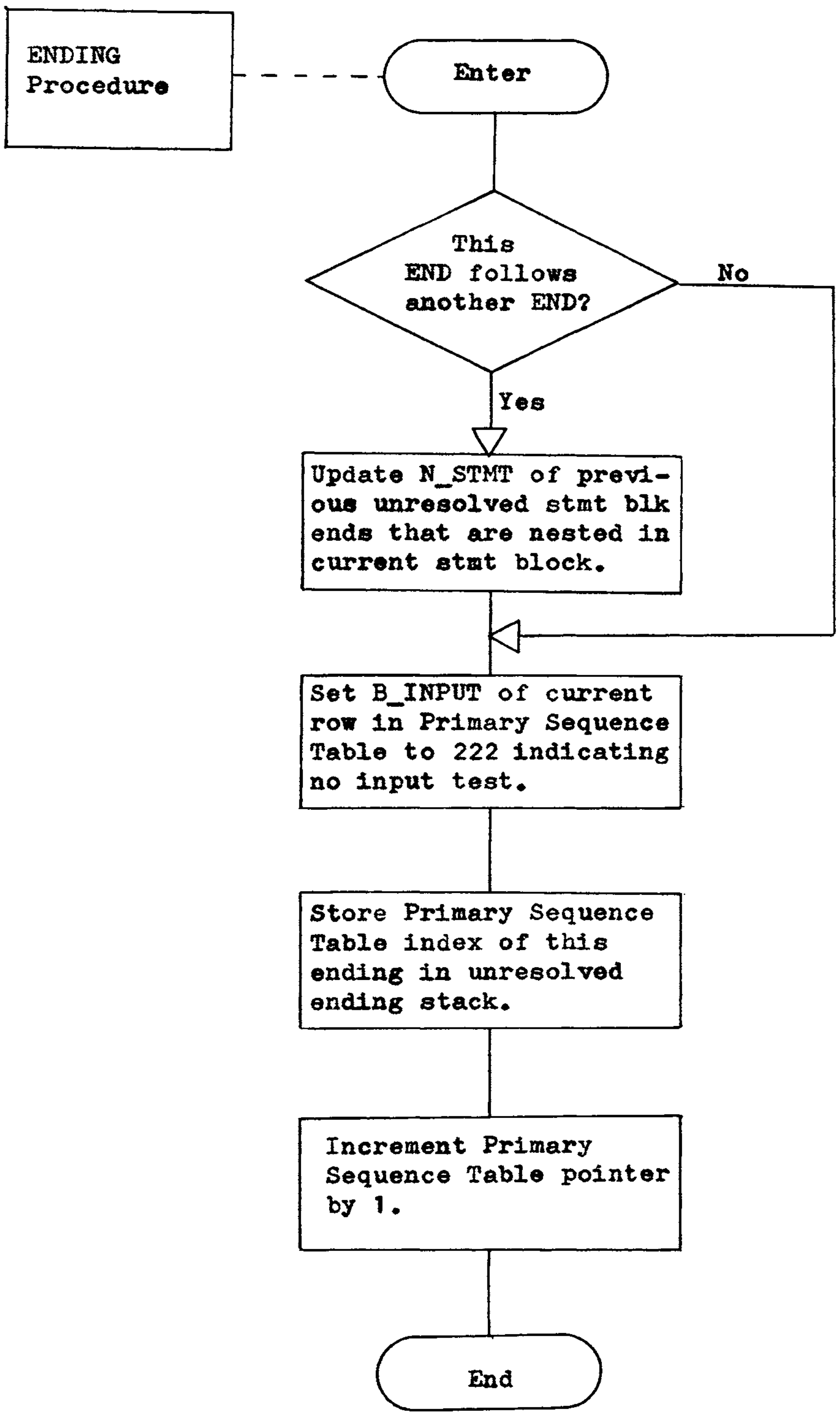

ENDING Procedure
Enter
This END follows another END?
No
Yes
Update N_STMT of previous unresolved stmt blk ends that are nested in current stmt block.
Set B_INPUT of current row in Primary Sequence Table to 222 indicating no input test.
Store Primary Sequence Table index of this ending in unresolved ending stack.
Increment Primary Sequence Table pointer by 1.
End

Figure 9. Flow Chart of ENDING Procedure

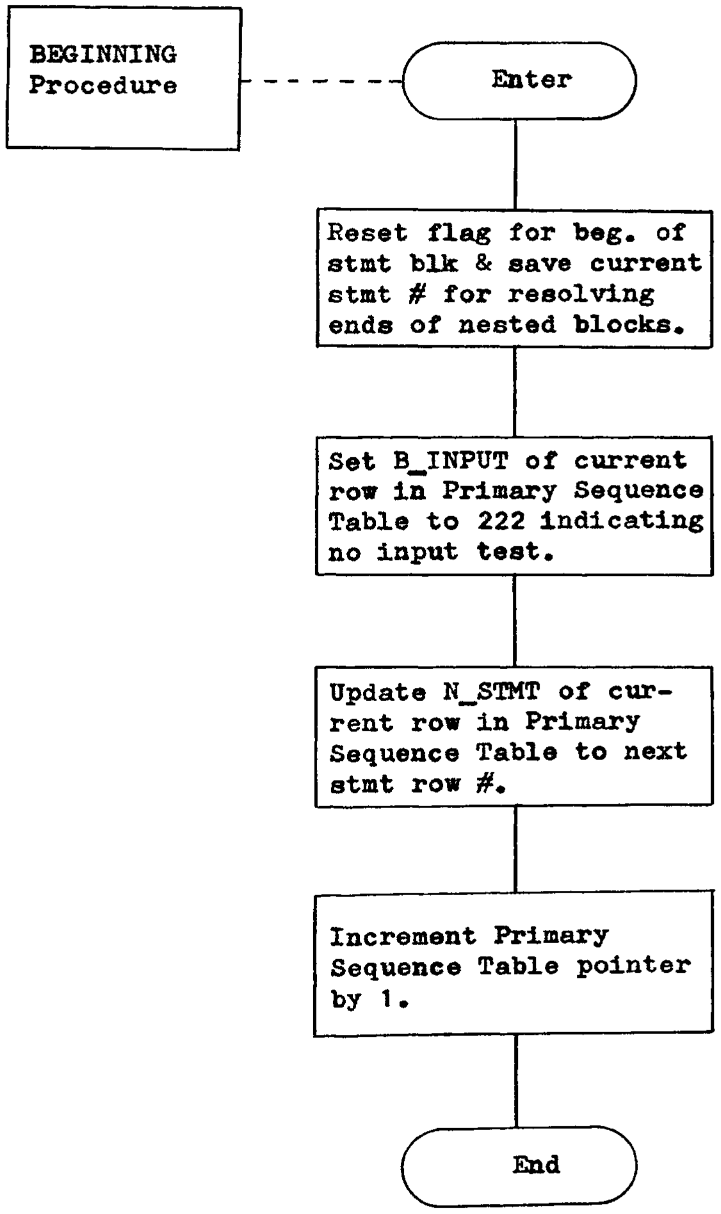

BEGINNING Procedure
Enter
Reset flag for beg. of stmt blk & save current stmt # for resolving ends of nested blocks.
Set B_INPUT of current row in Primary Sequence Table to 222 indicating no input test.
Update N_STMT of current row in Primary Sequence Table to next stmt row #.
Increment Primary Sequence Table pointer by 1.
End

Figure 10. Flow Chart of BEGINNING Procedure

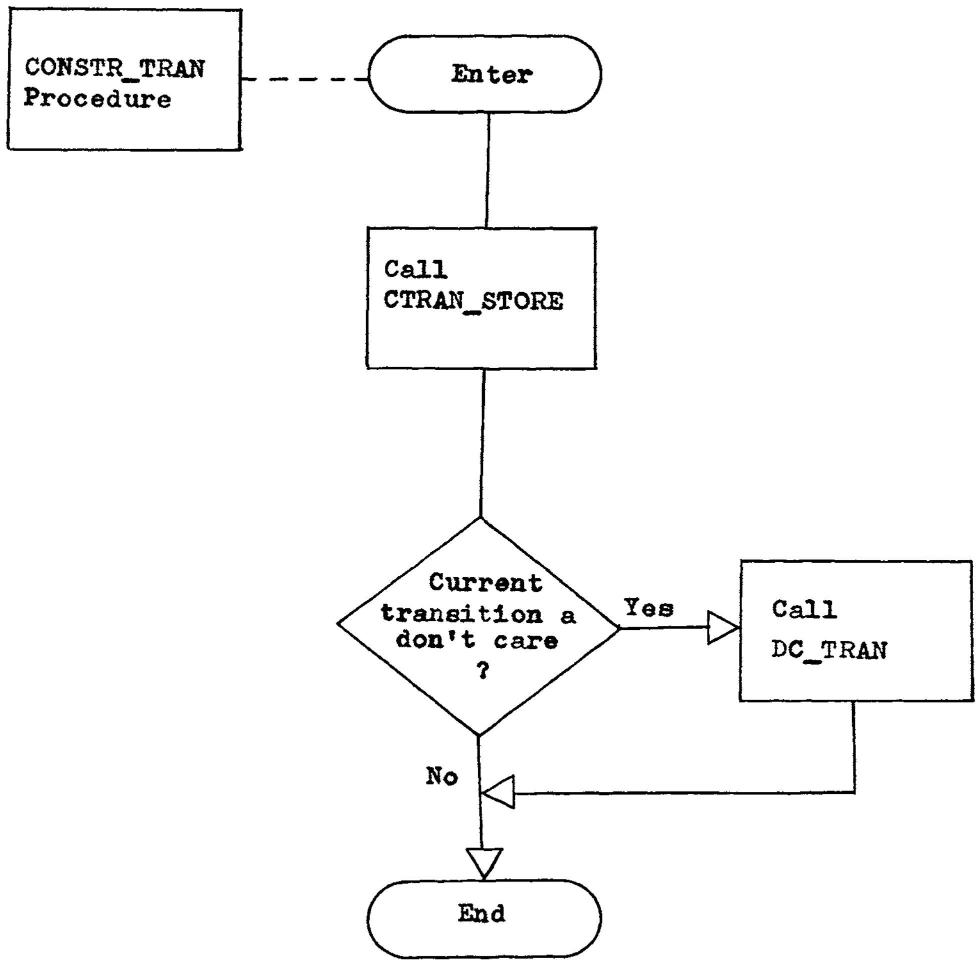


Figure 11. Flow Chart of CONSTR_TRAN Procedure

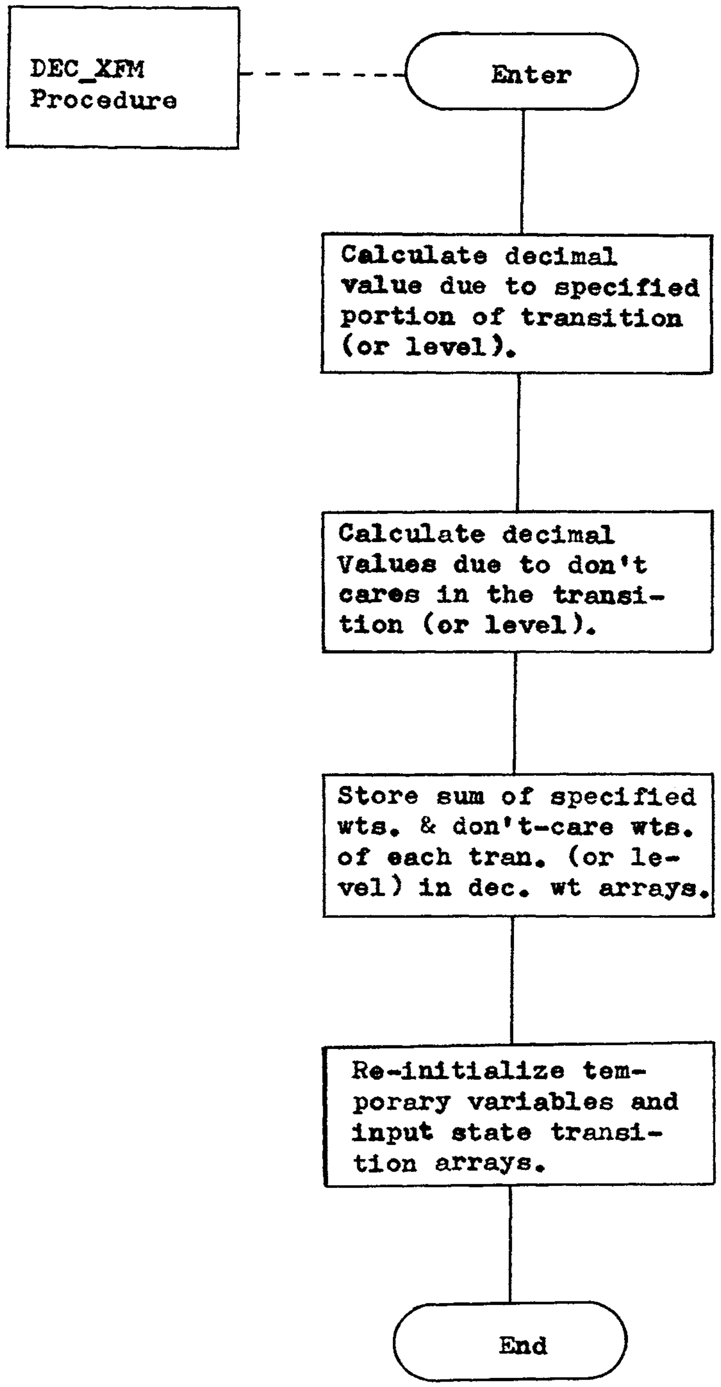

DEC_XFM Procedure
Enter
Calculate decimal value due to specified portion of transition (or level).
Calculate decimal Values due to don't cares in the transition (or level).
Store sum of specified wts. & don't-care wts. of each tran. (or level) in dec. wt arrays.
Re-initialize temporary variables and input state transition arrays.
End

Figure 12. Flow Chart of DEC_XFM Procedure

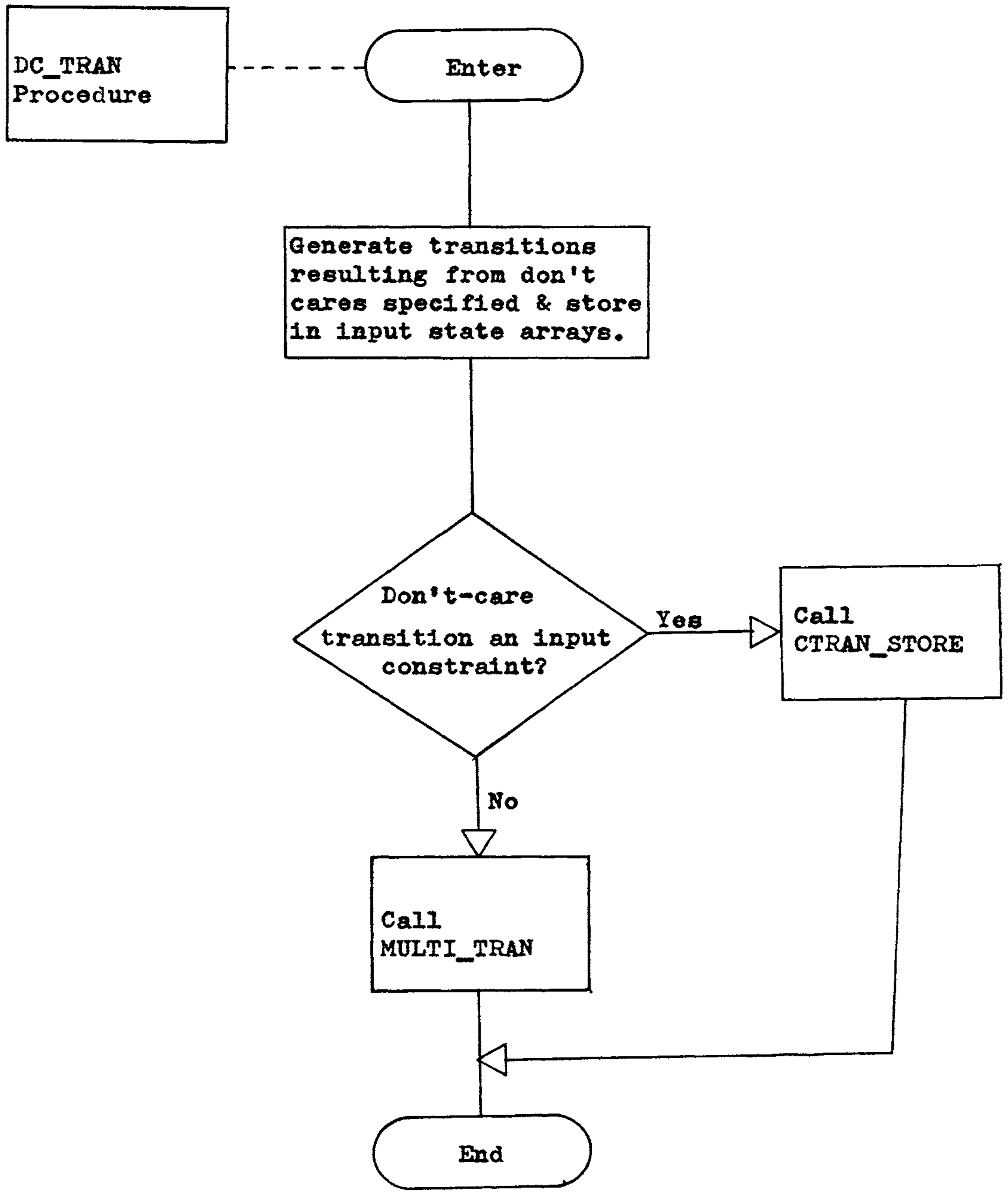

DC_TRAN Procedure
Enter
Generate transitions resulting from don't cares specified & store in input state arrays.
Don't-care transition an input constraint?
Yes
Call CTRAN_STORE
No
Call MULTI_TRAN
End

Figure 13. Flow Chart of DC_TRAN Procedure

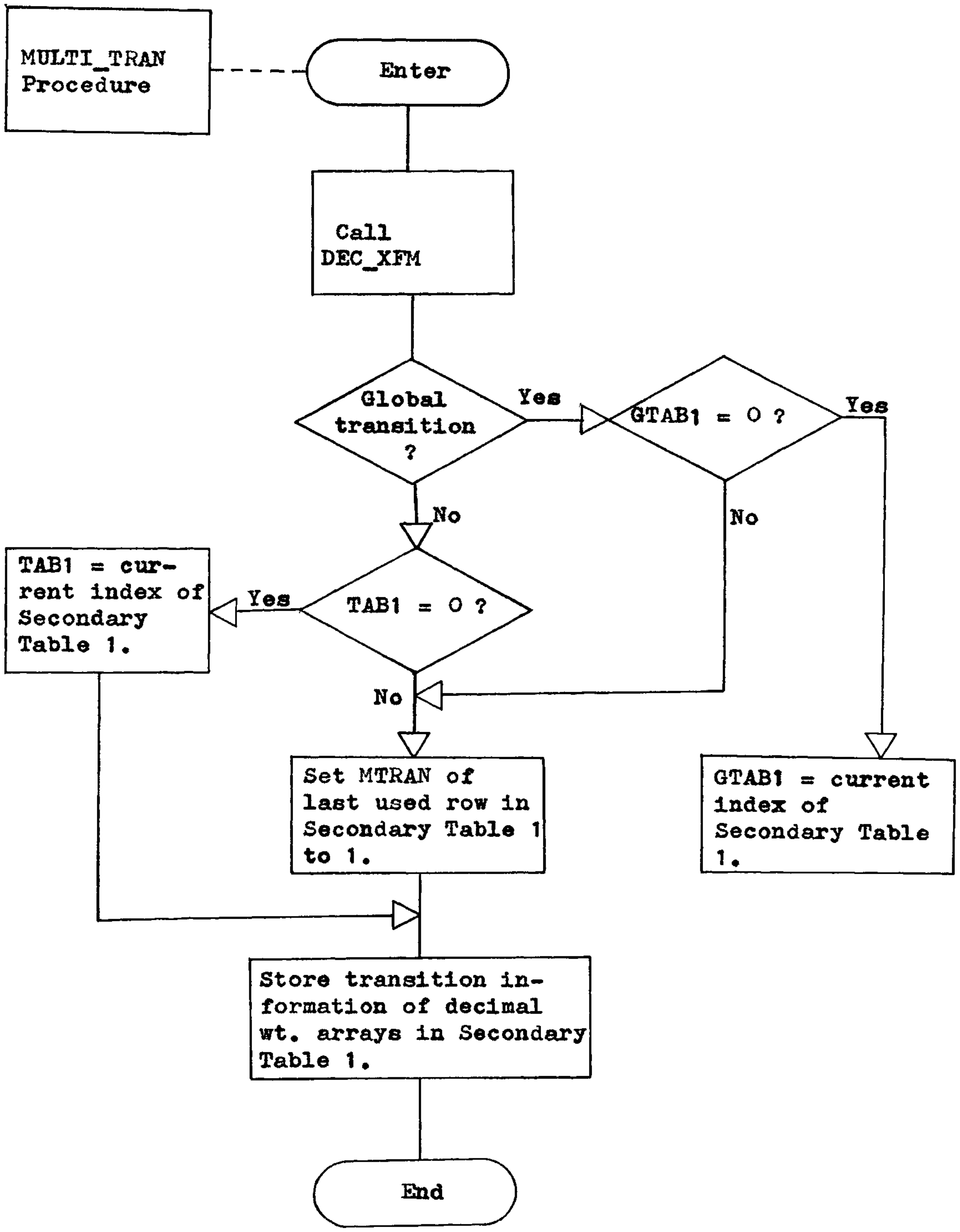

MULTI_TRAN Procedure
Enter
Call DEC_XFM
Global transition ?
Yes
GTAB1 = 0 ?
Yes
No
No
TAB1 = cur-rent index of Secondary Table 1.
Yes
TAB1 = 0 ?
No
Set MTRAN of last used row in Secondary Table 1 to 1.
GTAB1 = current index of Secondary Table 1.
Store transition in-formation of decimal wt. arrays in Secondary Table 1.
End

Figure 14. Flow Chart of MULTI_TRAN Procedure

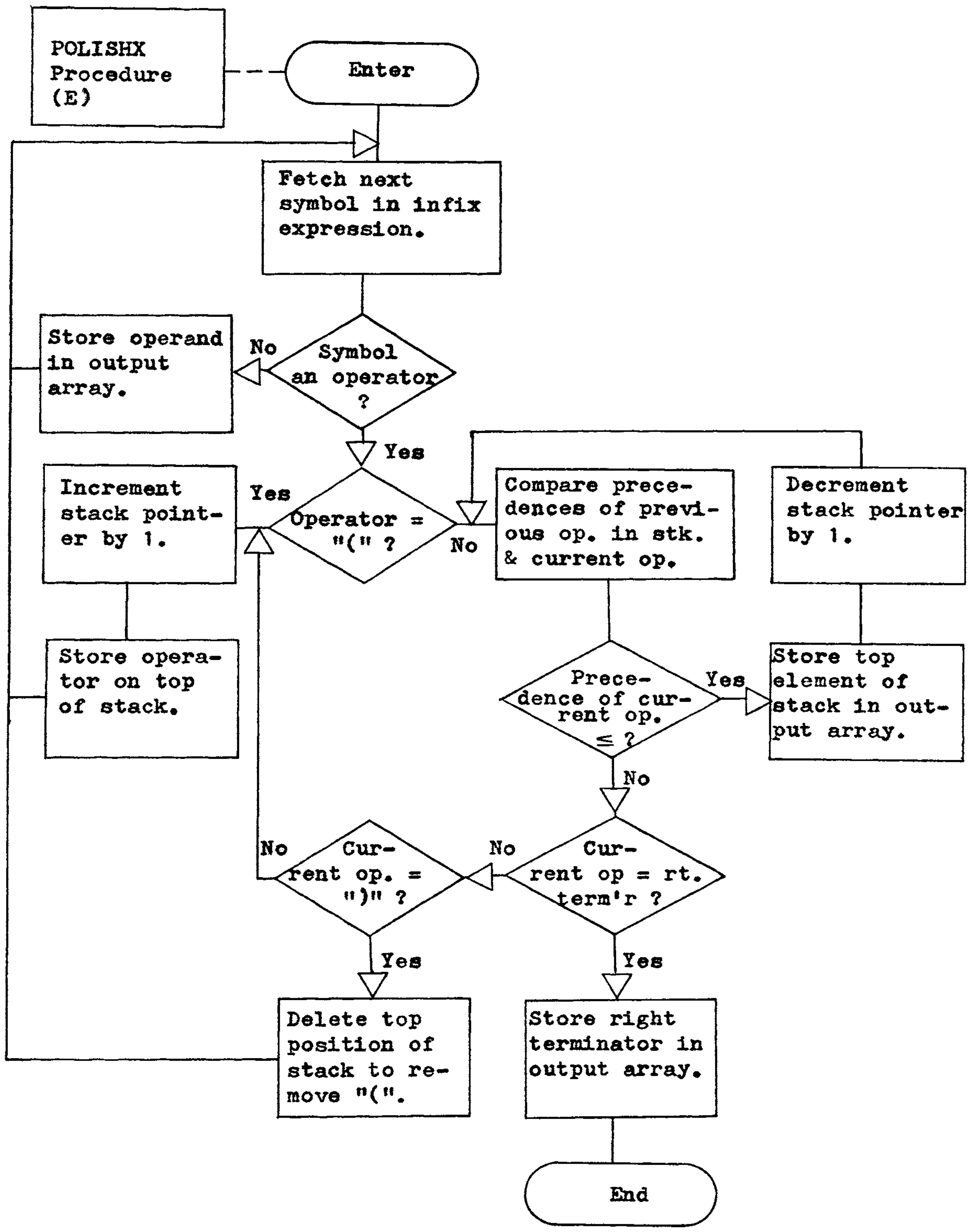

POLISHX Procedure (E)
Enter
Fetch next symbol in infix expression.
Symbol an operator ?
No
Store operand in output array.
Yes
Operator = "(" ?
Yes
Increment stack pointer by 1.
Store operator on top of stack.
No
Compare precedences of previous op. in stk. & current op.
Decrement stack pointer by 1.
Precedence of current op. ≤ ?
Yes
Store top element of stack in output array.
No
Current op = rt. term'r ?
No
Current op. = ")" ?
No
Yes
Delete top position of stack to remove "(".
Yes
Store right terminator in output array.
End

Figure 15. Flow Chart of POLISHX Procedure

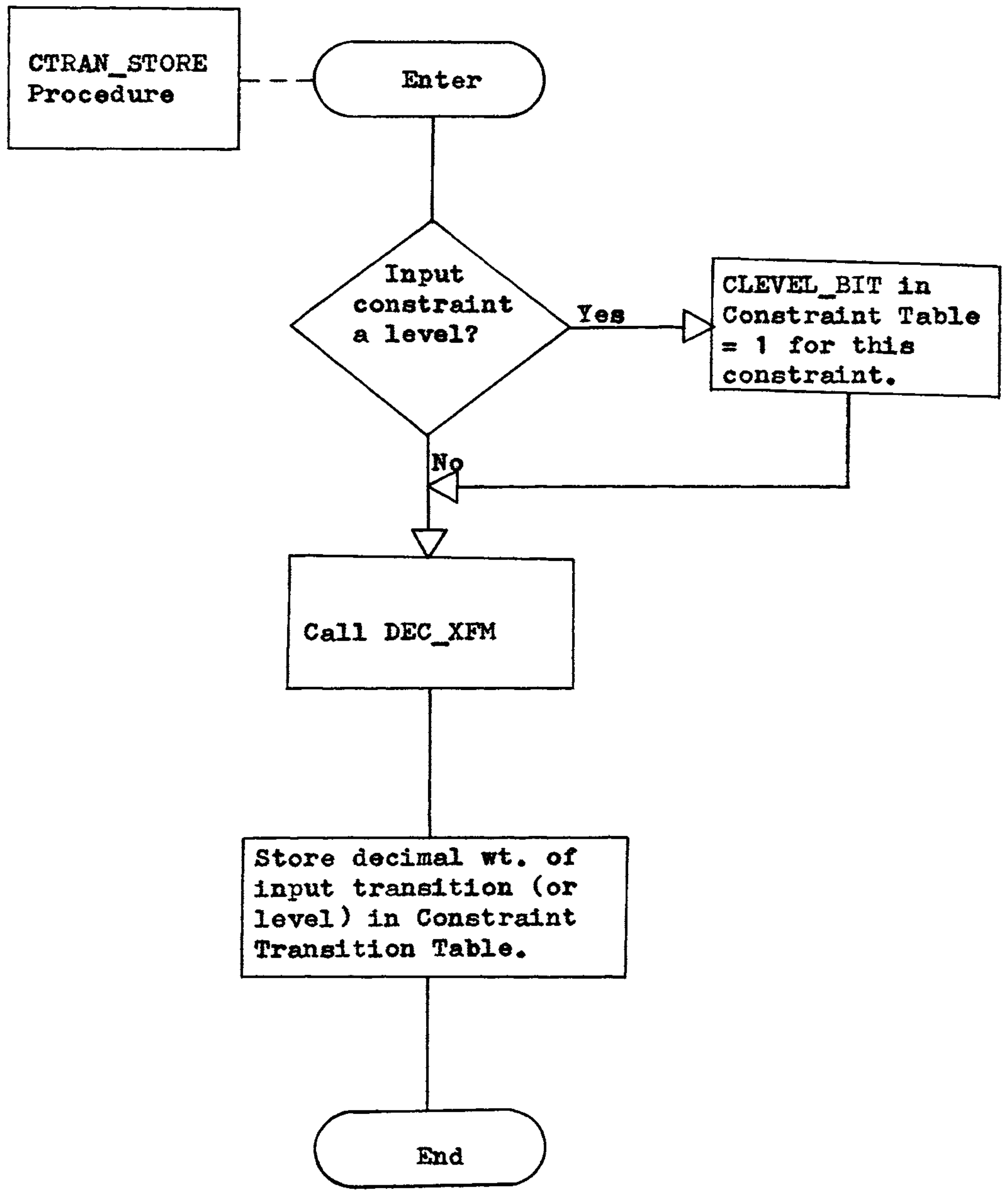

CTRAN_STORE Procedure
Enter
Input constraint a level?
Yes
CLEVEL_BIT in Constraint Table = 1 for this constraint.
No
Call DEC_XFM
Store decimal wt. of input transition (or level) in Constraint Transition Table.
End

Figure 16. Flow Chart of CTRAN_STORE Procedure

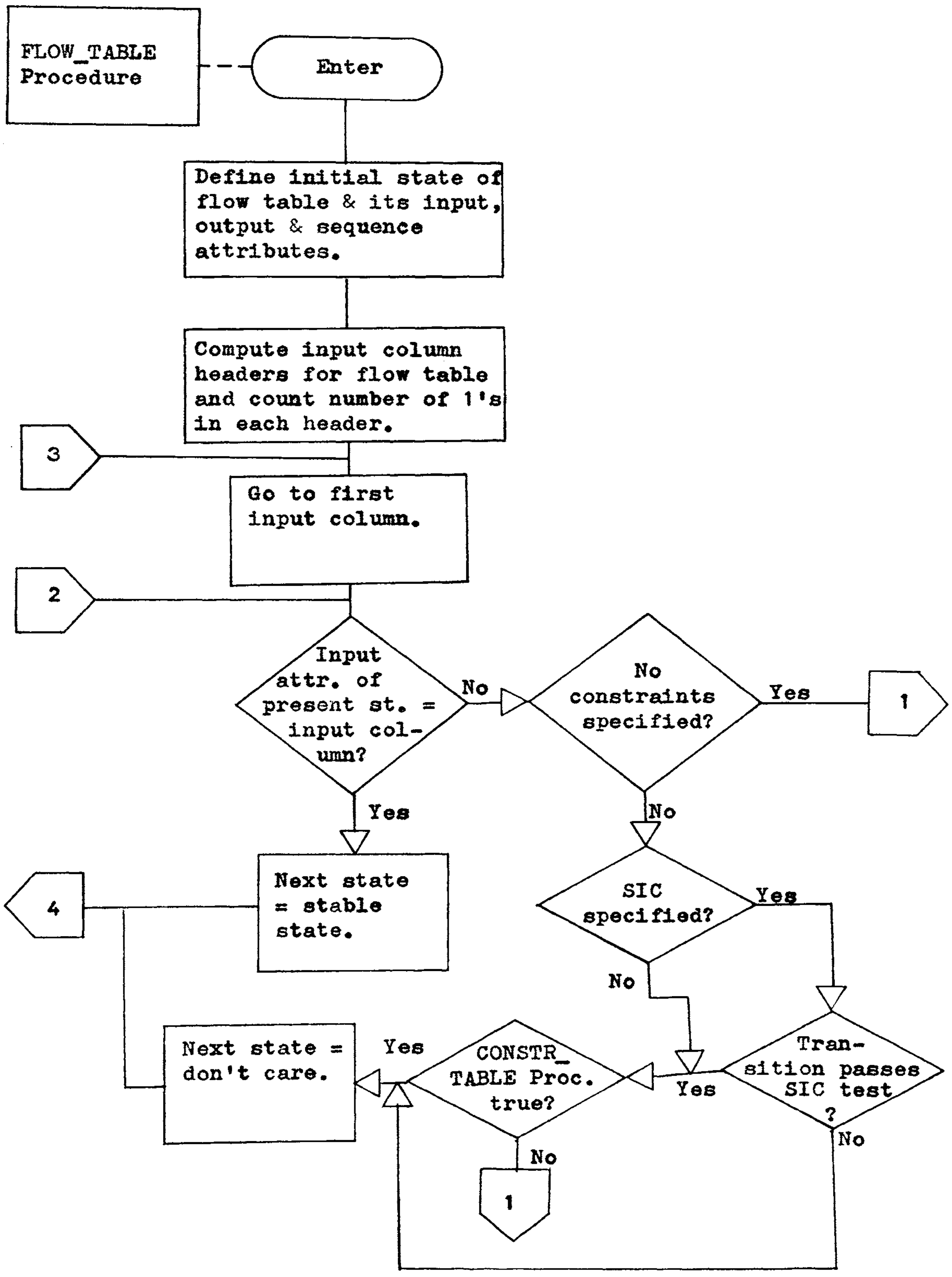
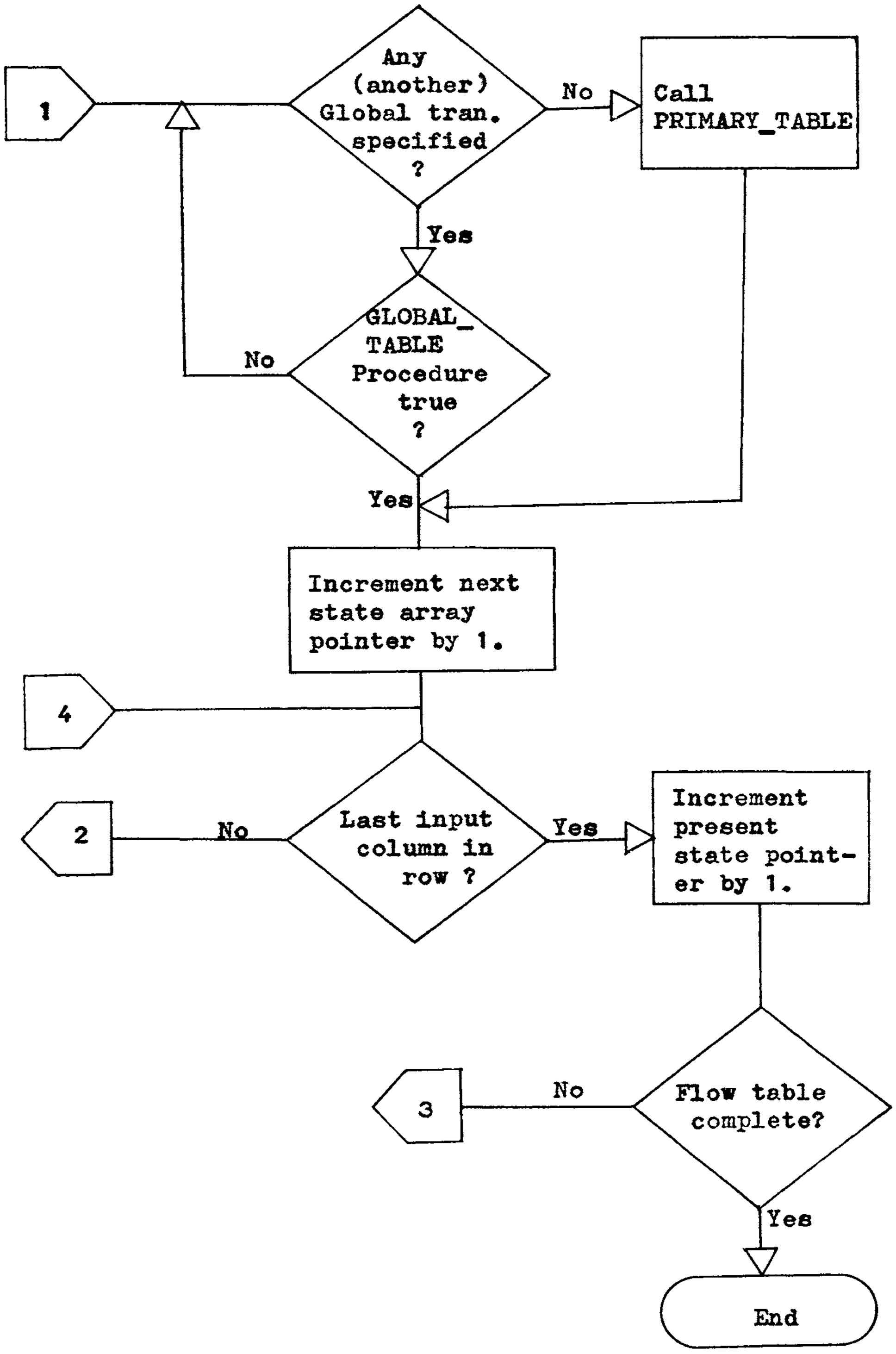

FLOW_TABLE Procedure
Enter
Define initial state of flow table & its input, output & sequence attributes.
Compute input column headers for flow table and count number of 1's in each header.
3
Go to first input column.
2
Input attr. of present st. = input column?
No
No constraints specified?
Yes
1
Yes
No
Next state = stable state.
4
SIC specified?
Yes
No
Next state = don't care.
Yes
CONSTR_ TABLE Proc. true?
Yes
Transition passes SIC test ?
No
No
1

Figure 17. Flow Chart of FLOW_TABLE Procedure



1
Any (another) Global tran. specified ?
No
Call PRIMARY_TABLE
Yes
GLOBAL_ TABLE Procedure true ?
No
Yes
Increment next state array pointer by 1.
4
2
No
Last input column in row ?
Yes
Increment present state point- er by 1.
3
No
Flow table complete?
Yes
End

Figure 17. (Continued)

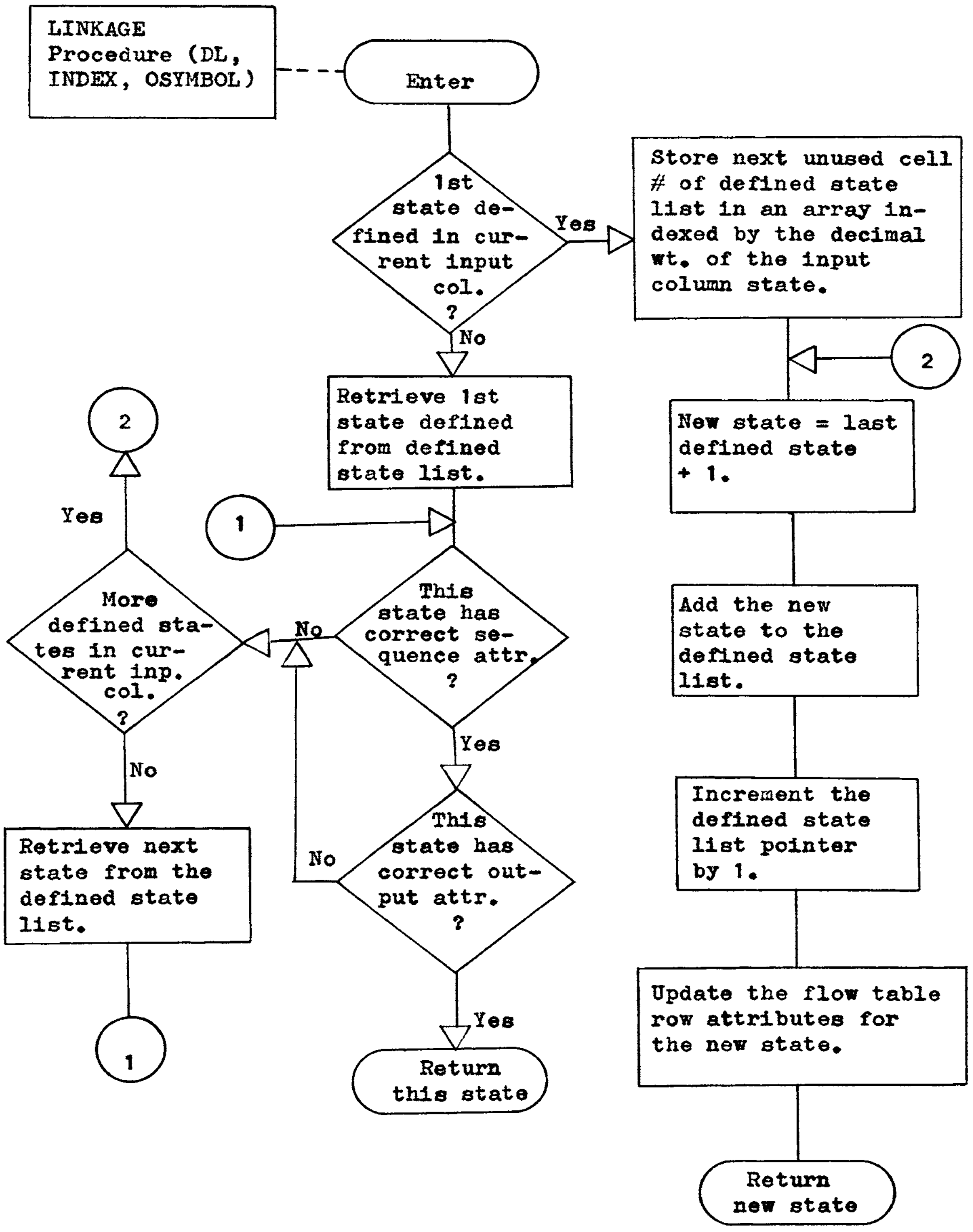

LINKAGE Procedure (DL, INDEX, OSYMBOL)
Enter
1st state defined in current input col. ?
Yes
No
Store next unused cell # of defined state list in an array indexed by the decimal wt. of the input column state.
2
Retrieve 1st state defined from defined state list.
New state = last defined state + 1.
1
This state has correct sequence attr. ?
No
Yes
More defined states in current inp. col. ?
Yes
No
Add the new state to the defined state list.
Retrieve next state from the defined state list.
This state has correct output attr. ?
No
Yes
Increment the defined state list pointer by 1.
Update the flow table row attributes for the new state.
Return this state
Return new state
1
2

Figure 18. Flow Chart of LINKAGE Procedure

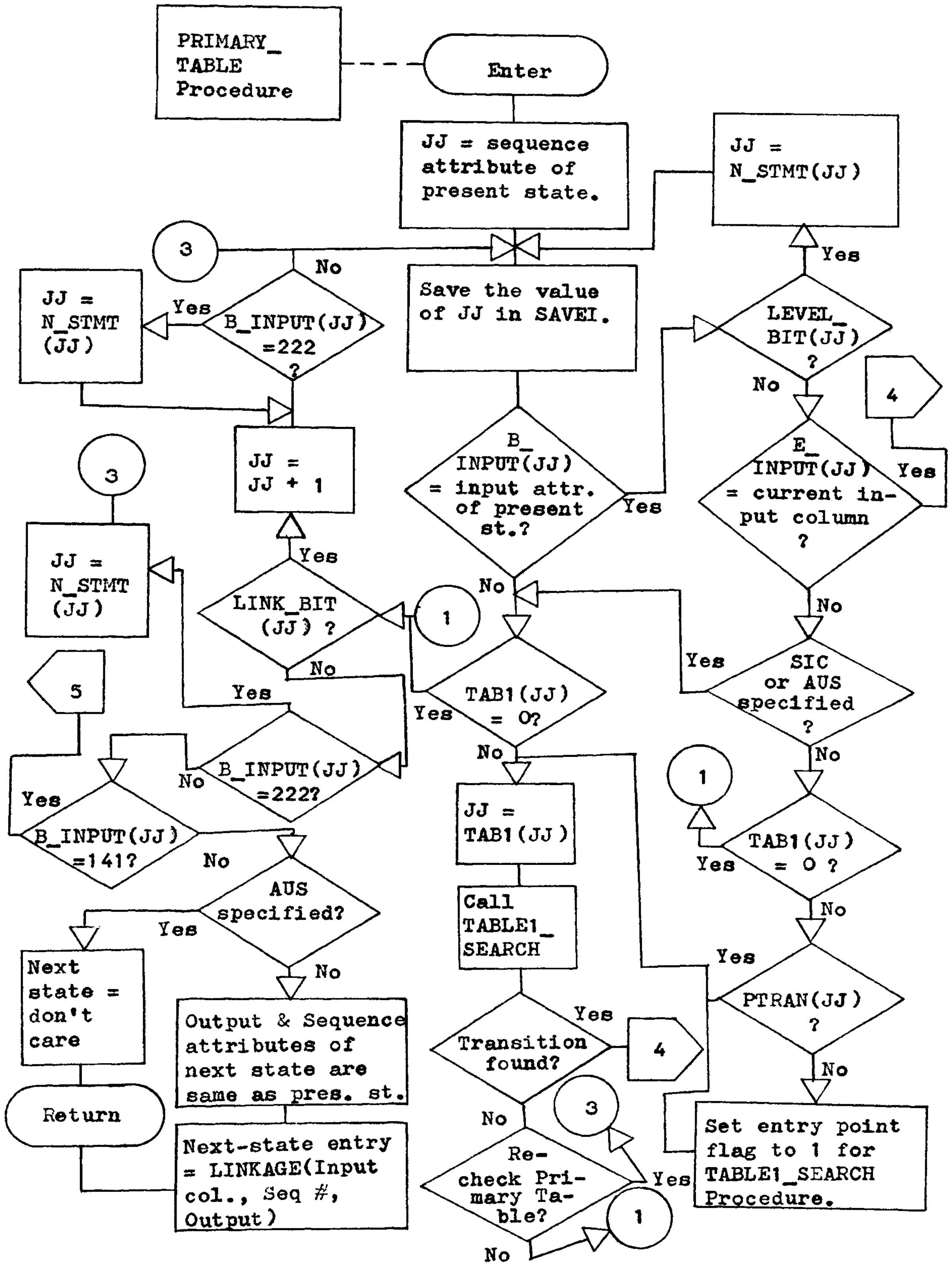


Figure 19. Flow Chart of PRIMARY_TABLE Procedure

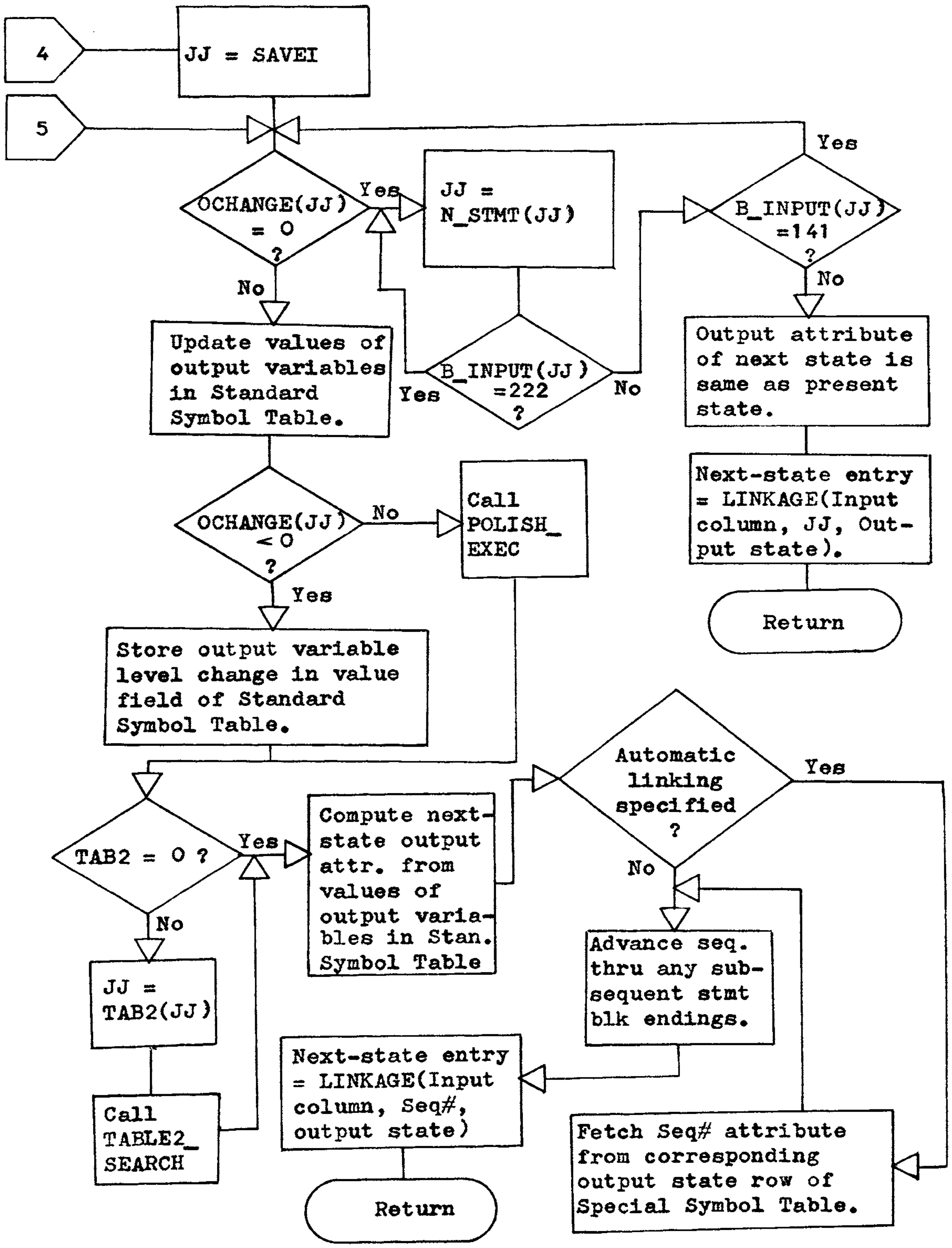

4
5
JJ = SAVEI
OCHANGE(JJ) = 0 ?
Yes
No
JJ = N_STMT(JJ)
B_INPUT(JJ) =222 ?
Yes
No
B_INPUT(JJ) =141 ?
Yes
No
Output attribute of next state is same as present state.
Next-state entry = LINKAGE(Input column, JJ, Output state).
Return
Update values of output variables in Standard Symbol Table.
OCHANGE(JJ) < 0 ?
No
Yes
Call POLISH_ EXEC
Store output variable level change in value field of Standard Symbol Table.
TAB2 = 0 ?
Yes
No
JJ = TAB2(JJ)
Call TABLE2_ SEARCH
Compute next-state output attr. from values of output variables in Stan. Symbol Table
Automatic linking specified ?
Yes
No
Advance seq. thru any subsequent stmt blk endings.
Next-state entry = LINKAGE(Input column, Seq#, output state)
Return
Fetch Seq# attribute from corresponding output state row of Special Symbol Table.

Figure 19. (Continued)

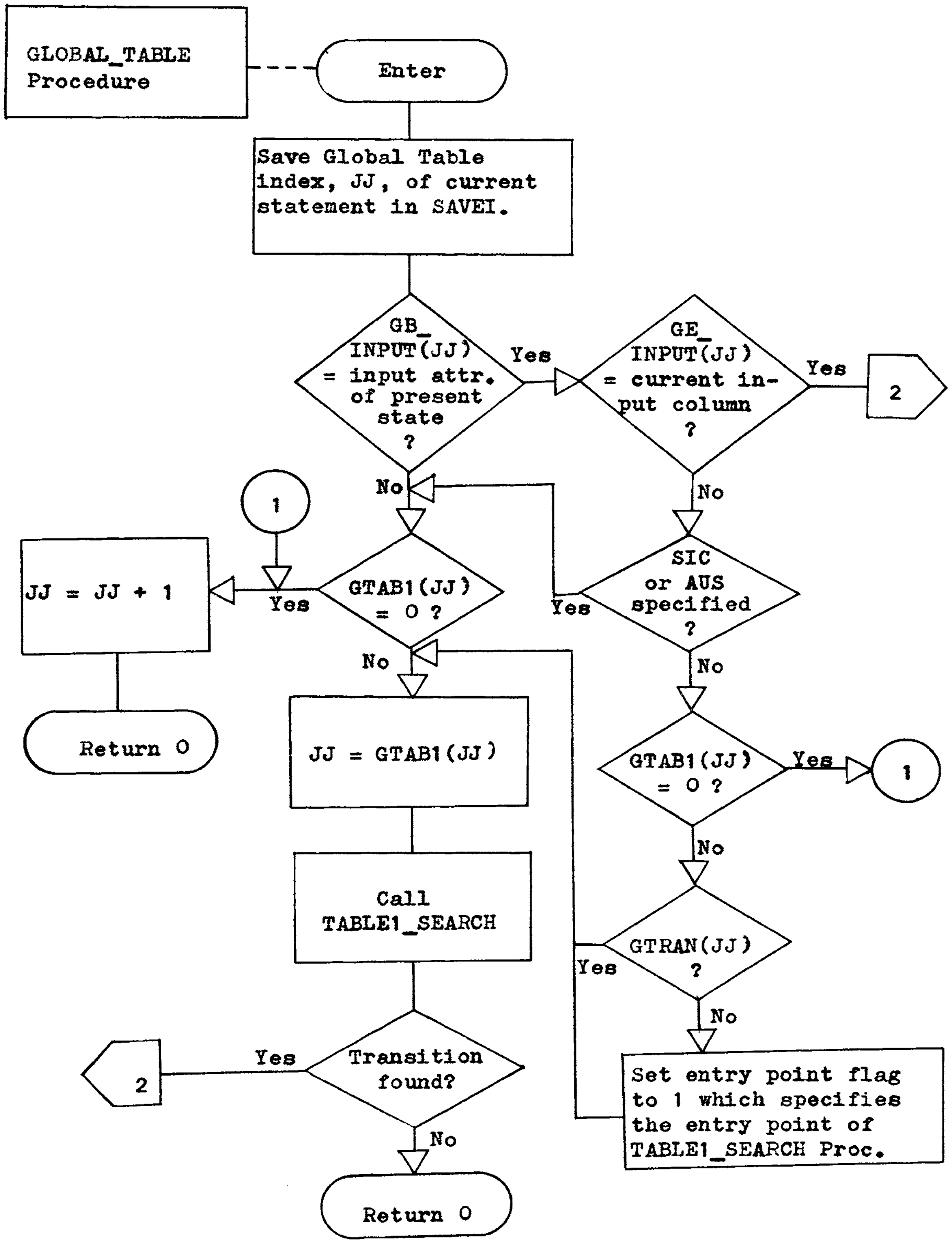

GLOBAL_TABLE Procedure
Enter
Save Global Table index, JJ, of current statement in SAVEI.
GB_ INPUT(JJ) = input attr. of present state ?
Yes
GE_ INPUT(JJ) = current input column ?
Yes
2
No
1
No
JJ = JJ + 1
Yes
GTAB1(JJ) = 0 ?
Yes
SIC or AUS specified ?
No
No
Return 0
JJ = GTAB1(JJ)
GTAB1(JJ) = 0 ?
Yes
1
No
Call TABLE1_SEARCH
GTRAN(JJ) ?
Yes
No
2
Yes
Transition found?
Set entry point flag to 1 which specifies the entry point of TABLE1_SEARCH Proc.
No
Return 0

Figure 20. Flow Chart of GLOBAL_TABLE Procedure

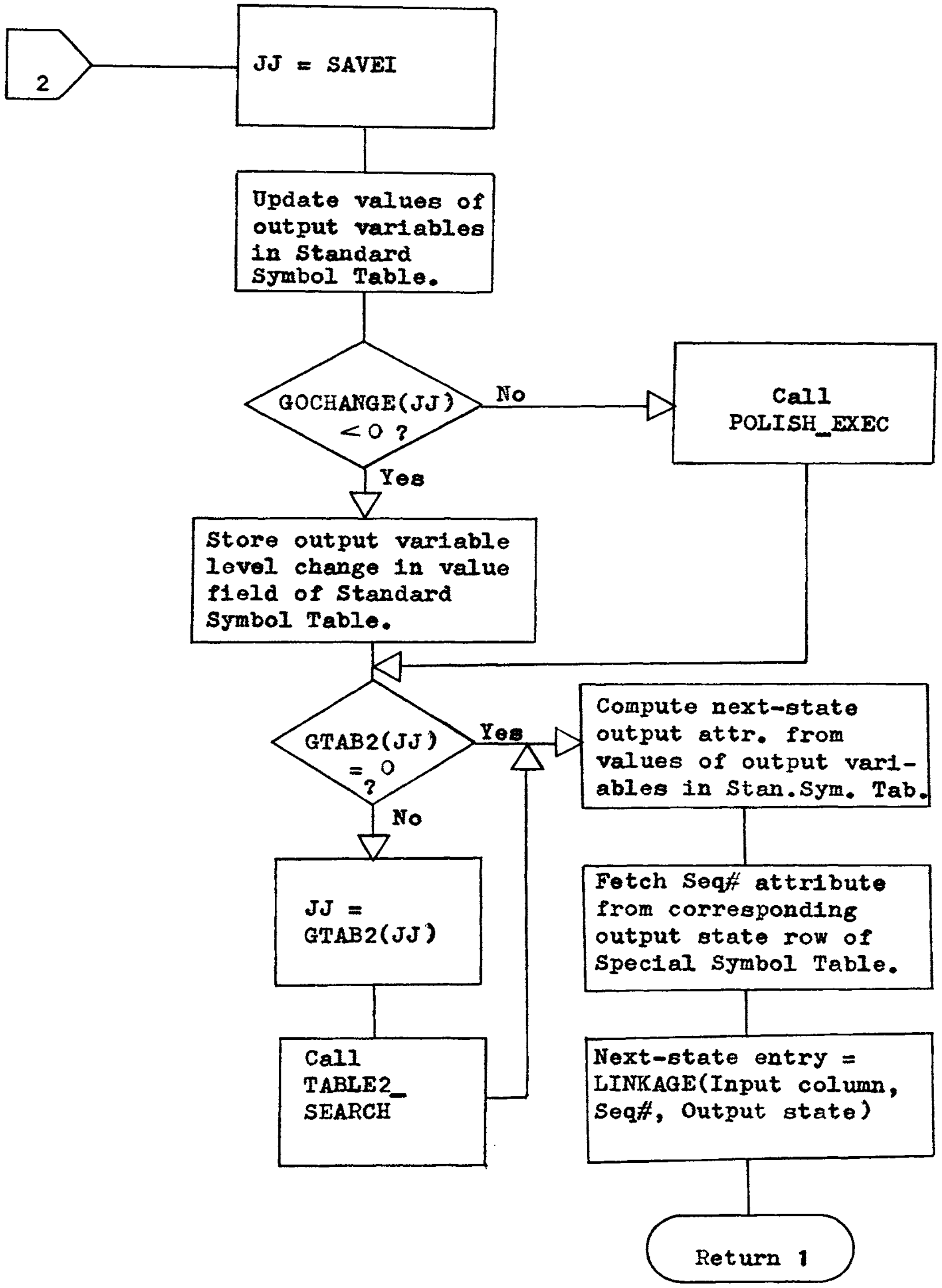

2
JJ = SAVEI
Update values of output variables in Standard Symbol Table.
GOCHANGE(JJ) < 0 ?
No
Call POLISH_EXEC
Yes
Store output variable level change in value field of Standard Symbol Table.
GTAB2(JJ) = 0 ?
Yes
Compute next-state output attr. from values of output variables in Stan.Sym. Tab.
No
JJ = GTAB2(JJ)
Fetch Seq# attribute from corresponding output state row of Special Symbol Table.
Call TABLE2_SEARCH
Next-state entry = LINKAGE(Input column, Seq#, Output state)
Return 1

Figure 20. (Continued)

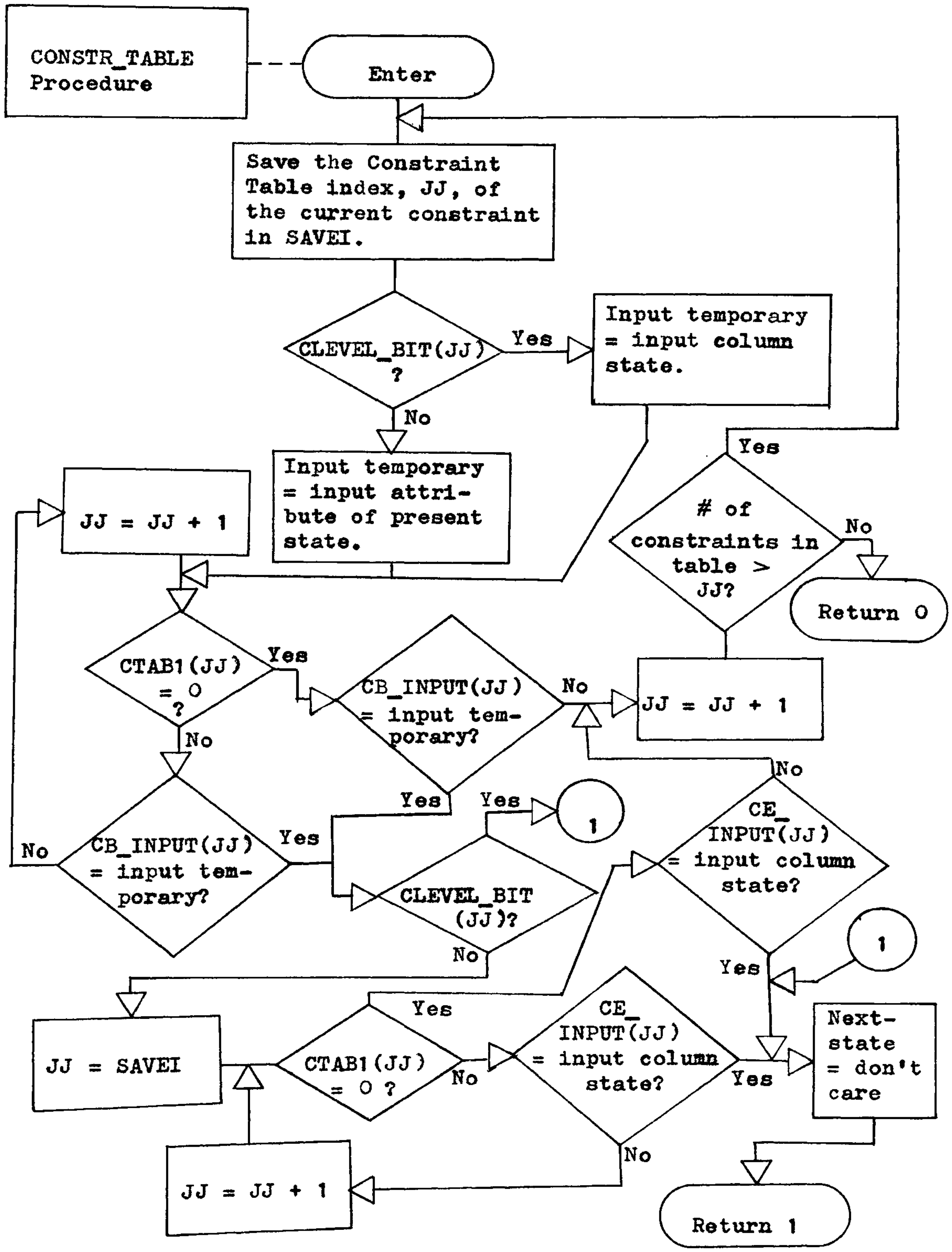

CONSTR_TABLE Procedure
Enter
Save the Constraint Table index, JJ, of the current constraint in SAVEI.
CLEVEL_BIT(JJ) ?
Yes
Input temporary = input column state.
No
Input temporary = input attribute of present state.
JJ = JJ + 1
Yes
# of constraints in table > JJ?
No
Return 0
CTAB1(JJ) = 0 ?
Yes
CB_INPUT(JJ) = input temporary?
No
JJ = JJ + 1
No
No
CB_INPUT(JJ) = input temporary?
Yes
Yes
Yes
1
No
CE_ INPUT(JJ) = input column state?
CLEVEL_BIT (JJ)?
No
1
Yes
JJ = SAVEI
Yes
CTAB1(JJ) = 0 ?
No
CE_ INPUT(JJ) = input column state?
Yes
Next-state = don't care
No
JJ = JJ + 1
Return 1

Figure 21. Flow Chart of CONSTR_TABLE Procedure

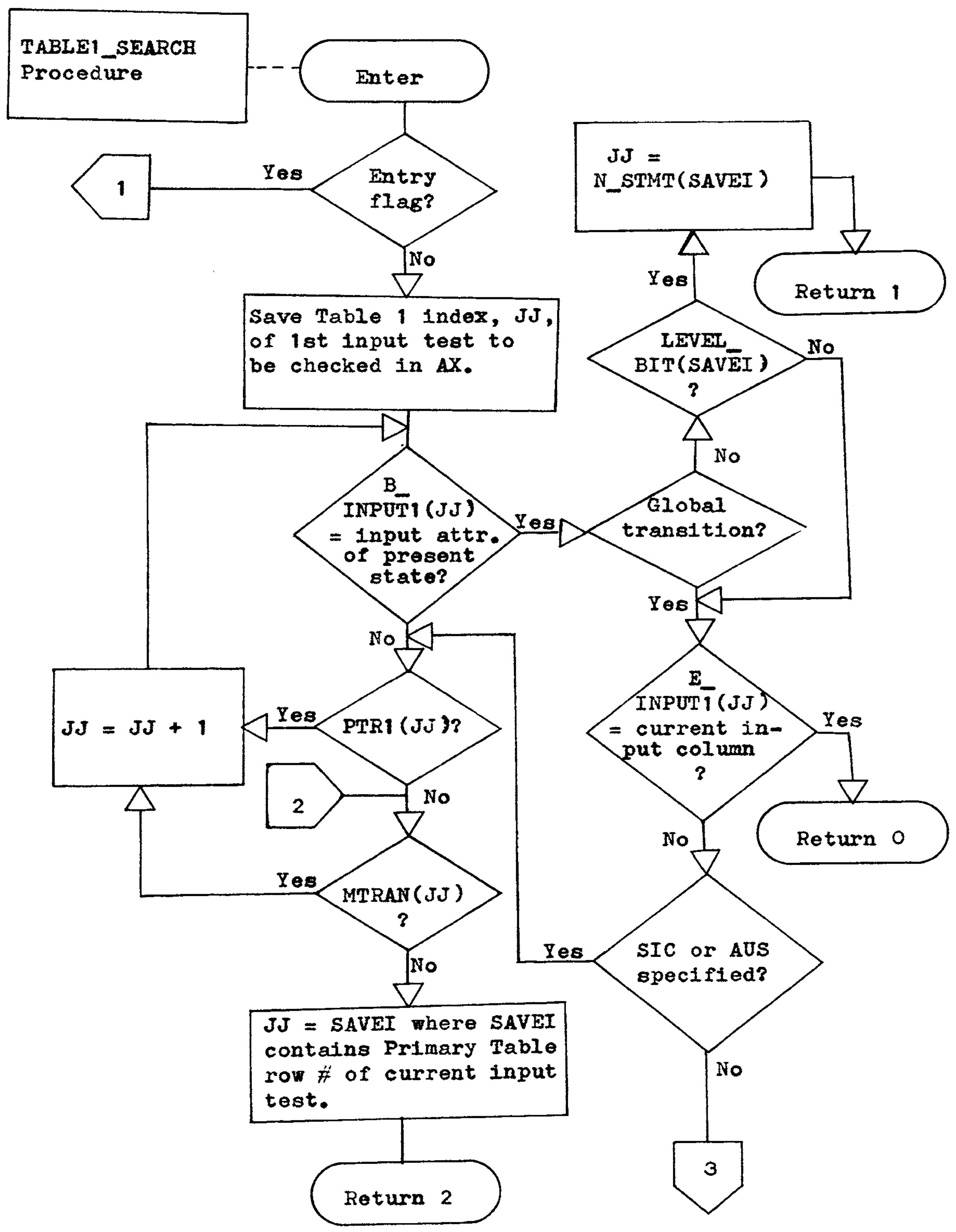


TABLE1_SEARCH Procedure
Enter
Entry flag?
Yes
No
1
Save Table 1 index, JJ, of 1st input test to be checked in AX.
B_ INPUT1(JJ) = input attr. of present state?
Yes
No
PTR1(JJ)?
Yes
No
JJ = JJ + 1
2
MTRAN(JJ) ?
Yes
No
JJ = SAVEI where SAVEI contains Primary Table row # of current input test.
Return 2
Global transition?
No
Yes
LEVEL_ BIT(SAVEI) ?
Yes
No
JJ = N_STMT(SAVEI)
Return 1
E_ INPUT1(JJ) = current input column ?
Yes
No
Return 0
SIC or AUS specified?
Yes
No
3

Figure 22. Flow Chart of TABLE1_SEARCH Procedure

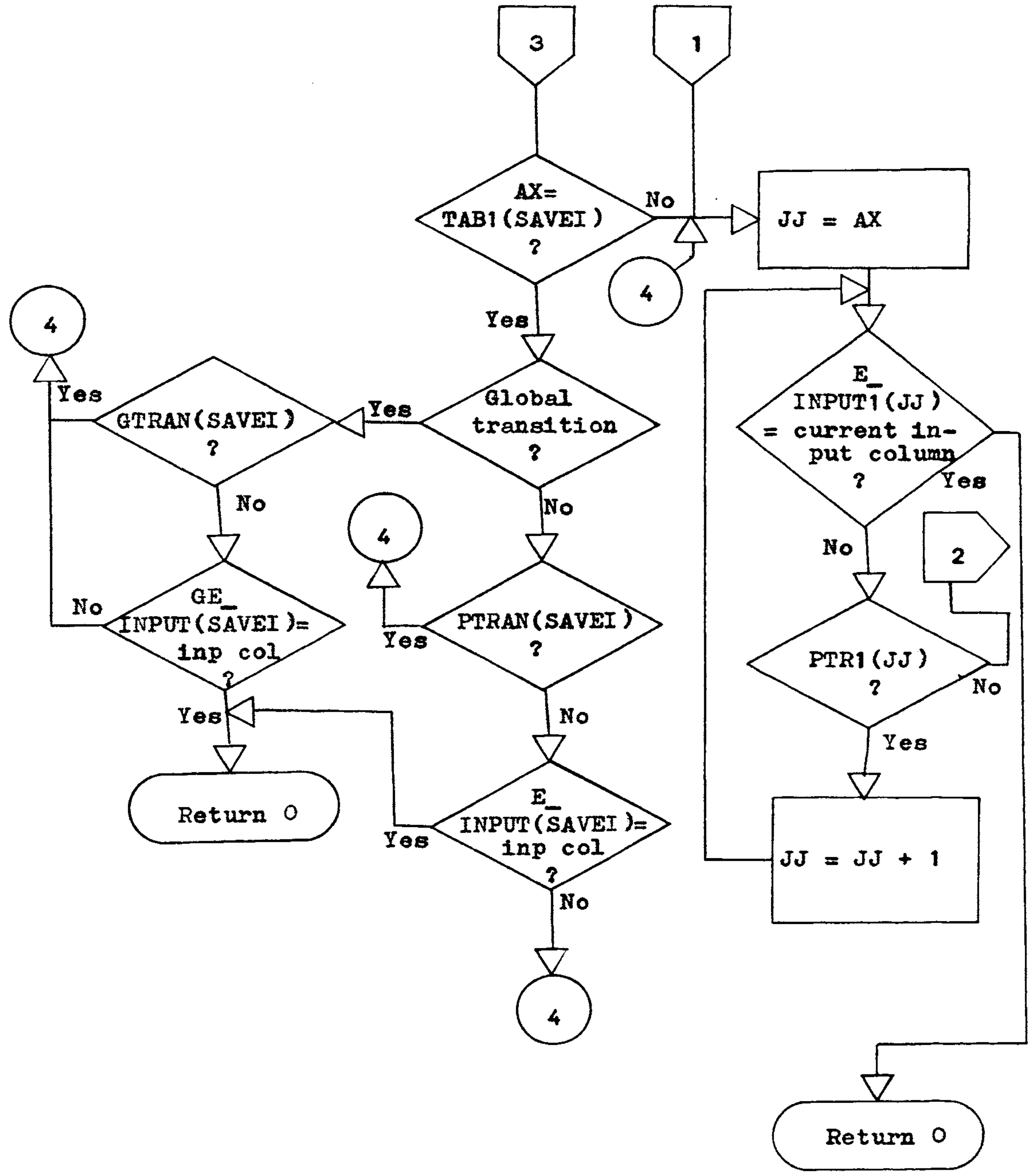


Figure 22. (Continued)

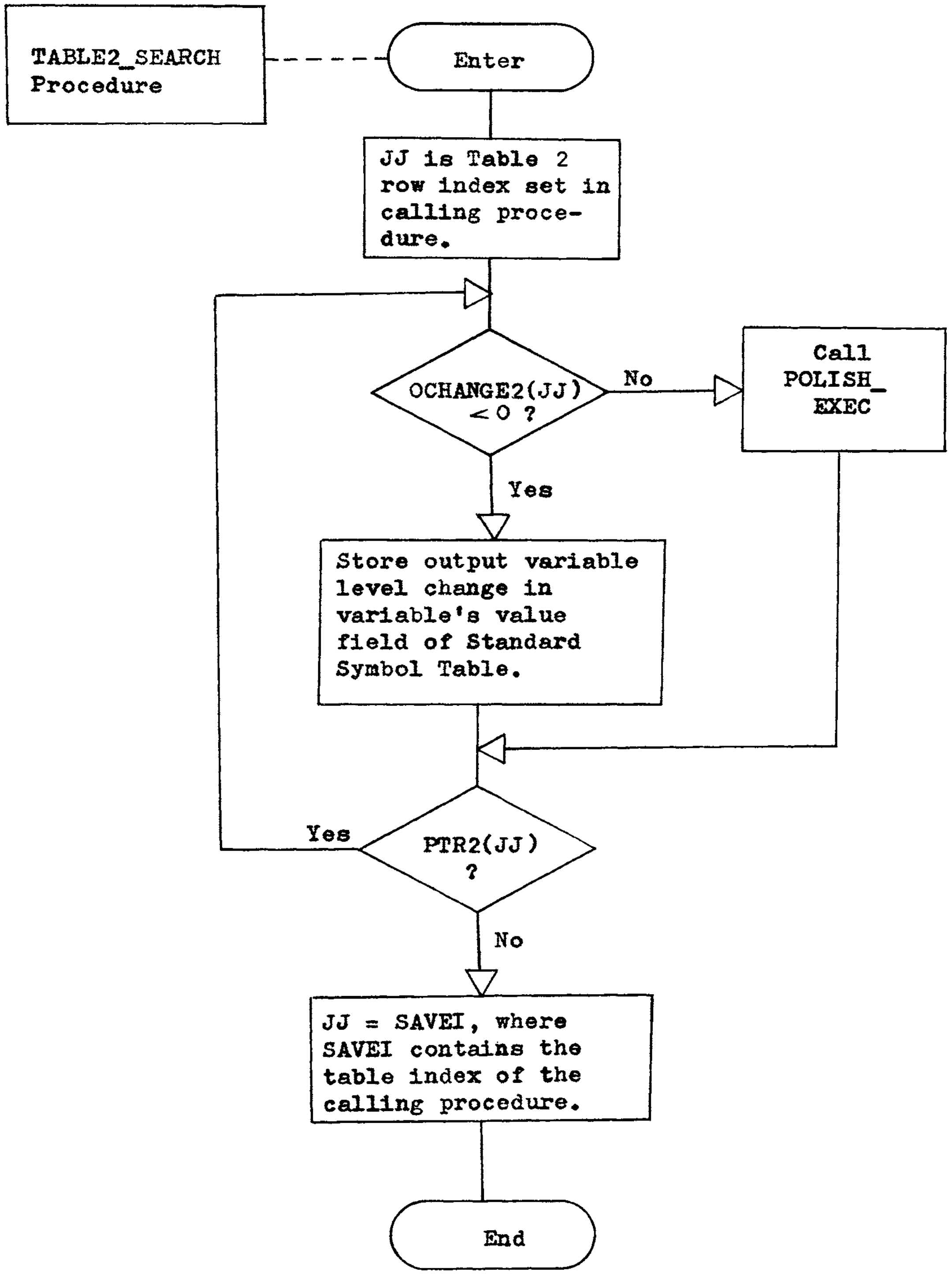


TABLE2_SEARCH Procedure
Enter
JJ is Table 2 row index set in calling procedure.
OCHANGE2(JJ) < 0 ?
No
Call POLISH_ EXEC
Yes
Store output variable level change in variable's value field of Standard Symbol Table.
Yes
PTR2(JJ) ?
No
JJ = SAVEI, where SAVEI contains the table index of the calling procedure.
End

Figure 23. Flow Chart of TABLE2_SEARCH Procedure

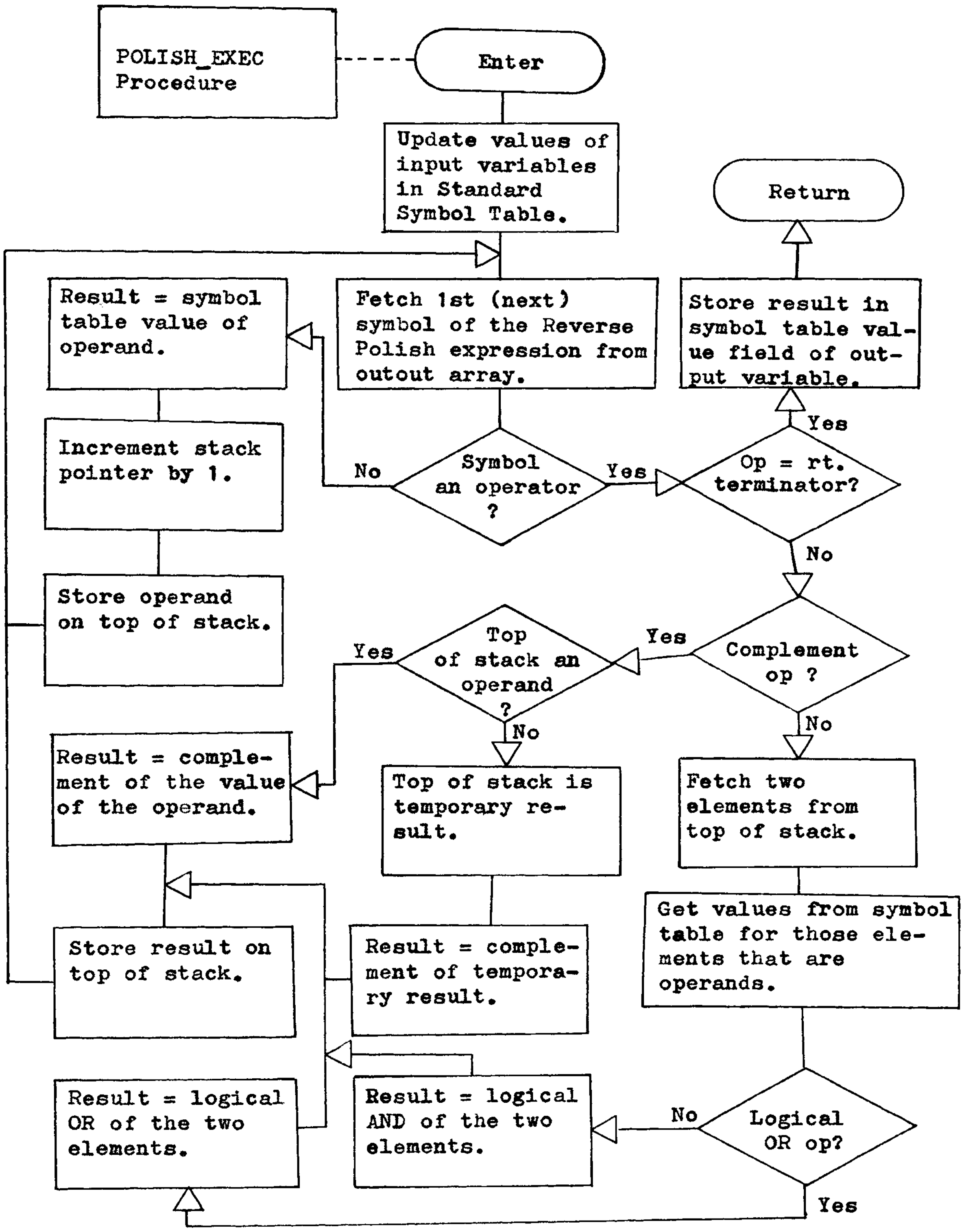

POLISH_EXEC Procedure
Enter
Update values of input variables in Standard Symbol Table.
Return
Result = symbol table value of operand.
Fetch 1st (next) symbol of the Reverse Polish expression from outout array.
Store result in symbol table value field of output variable.
Yes
Increment stack pointer by 1.
No
Symbol an operator ?
Yes
Op = rt. terminator?
No
Store operand on top of stack.
Yes
Top of stack an operand ?
Yes
Complement op ?
No
No
Result = complement of the value of the operand.
Top of stack is temporary result.
Fetch two elements from top of stack.
Get values from symbol table for those elements that are operands.
Store result on top of stack.
Result = complement of temporary result.
Result = logical OR of the two elements.
Result = logical AND of the two elements.
No
Logical OR op?
Yes

Figure 24. Flow Chart of POLISH_EXEC Procedure