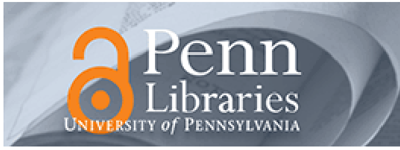


University of Pennsylvania
ScholarlyCommons

Technical Reports (CIS) Department of Computer & Information Science

August 1972

The Table Generating Routines of a Data Description Language Processor

Peter Gross
University of Pennsylvania

Follow this and additional works at: https://repository.upenn.edu/cis_reports

Recommended Citation

Peter Gross, "The Table Generating Routines of a Data Description Language Processor", . August 1972.

University of Pennsylvania Department of Computer and Information Science Technical Report No. MS-CIS-73-01.

This paper is posted at ScholarlyCommons. https://repository.upenn.edu/cis_reports/750
For more information, please contact repository@pobox.upenn.edu.

The Table Generating Routines of a Data Description Language Processor

Abstract

The Data Description Language Processor, designed by J. A. Ramirez, is the compiler for a modified version of the Data Description Language (DDL), written by D. P. Smith.

Two main phases exist in the DDL Processor:

1) The Syntactic Analysis phase and

2) The Code Generation phase

The former phase checks the DDL source for local and global syntactic flaws before passing control to the latter. In order to speed up execution of phase 2, internal tables (one symbol and several data tables), containing encoded versions of the DDL source input, are constructed. The tables, created during syntax analysis, will facilitate global syntax checking (verifying all DDL statement references to be valid), and will permit code generation to operate more quickly by providing it with the "essence" of the source data and, hence, negate the necessity of a second pass over the source input.

Comments

University of Pennsylvania Department of Computer and Information Science Technical Report No. MS-CIS-73-01.

This technical report is available at ScholarlyCommons: https://repository.upenn.edu/cis_reports/750

85

TECHNICAL REPORT

THE TABLE GENERATING ROUTINES OF A DATA DESCRIPTION LANGUAGE PROCESSOR

by

Peter Gross

Prepared for the

Office of Naval Research
Information Systems Branch
Arlington, Virginia

under

Contract N00014-67-A-0216-0014
Research Project No. 049-098

UNIVERSITY OF PENNSYLVANIA
The Moore School of Electrical Engineering
Philadelphia, Pennsylvania 19104

Report No. 73-01

85

TECHNICAL REPORT

THE TABLE GENERATING ROUTINES OF A DATA DESCRIPTION LANGUAGE PROCESSOR

by

Peter Gross

Prepared for the

Office of Naval Research
Information Systems Branch
Arlington, Virginia

under

Contract N00014-67-A-0216-0014
Research Project No. 049-098

UNIVERSITY OF PENNSYLVANIA
The Moore School of Electrical Engineering
Philadelphia, Pennsylvania 19104

Report No. 73-01

University of Pennsylvania
THE MOORE SCHOOL OF ELECTRICAL ENGINEERING
Philadelphia, Pennsylvania 19104

TECHNICAL REPORT

THE TABLE GENERATING ROUTINES OF A DATA DESCRIPTION LANGUAGE PROCESSOR

by

Peter Gross

August 1972

Submitted to the
Office of Naval Research
Information Systems Branch
Arlington, Virginia

under
Contract N00014-67-A-0216-0014
Research Project No. 049-272

Reproduction in whole or in part is
permitted for any purpose of the
United States Government

Moore School Report No. 73-01

Security Classification

DOCUMENT CONTROL DATA - R & D

(Security classification of title, body of abstract and indexing annotation must be entered when the overall report is classified)

1. ORIGINATING ACTIVITY (Corporate author)
The Moore School of Electrical Engineering
University of Pennsylvania
Philadelphia, Pennsylvania 19104

2a. REPORT SECURITY CLASSIFICATION
UNCLASSIFIED

2b. GROUP

3. REPORT TITLE

THE TABLE GENERATING ROUTINES OF A DATA DESCRIPTION LANGUAGE PROCESSOR

4. DESCRIPTIVE NOTES (Type of report and inclusive dates)
Technical Report

5. AUTHOR(S) (First name, middle initial, last name)

Peter Gross

6. REPORT DATE
August 1972

7a. TOTAL NO. OF PAGES
105

7b. NO. OF REFS
7

8a. CONTRACT OR GRANT NO.
N00014-67-A-0216-0014

b. PROJECT NO.
049-098

c.

d.

9a. ORIGINATOR'S REPORT NUMBER(S)
Moore School Report No. 73-01

9b. OTHER REPORT NO(S) (Any other numbers that may be assigned this report)

10. DISTRIBUTION STATEMENT

Reproduction in whole or in part is permitted for any purpose of the United States government.

11. SUPPLEMENTARY NOTES

12. SPONSORING MILITARY ACTIVITY

13. ABSTRACT

The Data Description Language Processor, designed by J. A. Ramirez[7], is the compiler for a modified version of the Data Description Language (DDL), written by D. P. Smith[5].

Two main phases exist in the DDL Processor:

1) The Syntactic Analysis phase and
2) The Code Generation phase

The former phase checks the DDL source for local and global syntactic flaws before passing control to the latter. In order to speed up execution of phase 2, internal tables (one symbol and several data tables), containing encoded versions of the DDL source input, are constructed. The tables, created during syntax analysis, will facilitate global syntax checking (verifying all DDL statement references to be valid), and will permit code generation to operate more quickly by providing it with the "essence" of the source data and, hence, negate the necessity of a second pass over the source input.

Security Classification

14 KEY WORDS	LINK A		LINK B		LINK C	
	ROLE	WT	ROLE	WT	ROLE	WT
DDL (Data Description Language)						
DDL processor						
data definition						
data translation						
DDL compiler						
Syntactic Analysis Program Generator (SAPG)						
lexical phase						
linear structure						
hash structure						
tree structure						
Extended Backus-Naur Form (EBNF)						
Syntactic Analysis Program (SAP)						

DD FORM 1 NOV 65 1473 (BACK)

S/N 0101-807-6821

Security Classification

A-31409

THE TABLE GENERATING ROUTINES OF A DATA
DESCRIPTION LANGUAGE PROCESSOR

Abstract

The Data Description Language Processor, designed by J. A. Ramirez[7], is the compiler for a modified version of the Data Description Language (DDL), written by D. P. Smith[5].

Two main phases exist in the DDL Processor:

1) The Syntactic Analysis phase and

2) The Code Generation phase

The former phase checks the DDL source for local and global syntactic flaws before passing control to the latter. In order to speed up execution of phase 2, internal tables (one symbol and several data tables), containing encoded versions of the DDL source input, are constructed. The tables, created during syntax analysis, will facilitate global syntax checking (verifying all DDL statement references to be valid), and will permit code generation to operate more quickly by providing it with the "essence" of the source data and, hence, negate the necessity of a second pass over the source input.

THE TABLE GENERATING ROUTINES OF A DATA DESCRIPTION LANGUAGE PROCESSOR

by PETER GROSS

ACKNOWLEDGEMENTS

The development of the Data Description Language has been an effort carried out by several individuals on the staff of the project supported by the Office of Naval Research by Contract N00014-67-A-0216-0014. The development of the language itself has been carried out by Dr. Diane Pirog Smith. The first manual for the use of the language was published in April 1971[7]. The language was re-designed and a new manual was included in Dr. Smith's dissertation in December 1971[5].

Subsequently, a definition of the language and the design of a processor for the language was initiated. A first report on the design was published in December 1971[7] by the design team.

The author is a participant in the design of the processor. The other participants are Jesus Ramirez, and A. French. In the interest of completeness, the author has included in this report a view of the entire system. This is a major revision and documentation of the design reported in the December 1971 report.

Certain sections of the document were prepared with the help of other documents or other members of the project. These contributions are outlined below:

Section 1.1 - DDL Annual Report, December 1971[7].

Section 1.2 - DDL Annual Report, December 1971[7].

Section 2.1 - The Syntactic Analysis Program Generator as Designed by J. A. Ramirez and A. French[1]

Section 3.2.1 - EBNF With Subroutine Calls was Designed by J. Ramirez

Section 4.2 - The data table formats were described in the December 1971 Annual Report[7] and

were revised throughout the following
months by J. A. Ramirez and the author.

To define the authors specific contributions, references are made in the following sections of this report indicating the sources of information.

The author acknowledges the advise, support, and direction he has received from everyone in the design team and wishes to thank them for their assistance and aid.

TABLE OF CONTENTS

1.0 INTRODUCTION 1

1.1 Need for a DDL 1

1.2 Summary of DDL Capabilities 1

1.3 Important Features of the Design 3

1.4 Internal Tables in the DDL Processor 4

1.5 Organization of the Thesis 5

2.0 OVERALL DESIGN OF THE DDL PROCESSOR 6

2.1 The Syntactic Analysis Program Generator 6

2.2 The DDL Compiler 6

2.2.1 The Lexical Phase 6

2.2.2 The Syntactic Analysis and Table Generation Phase 6

2.2.3 Code Generation 8

3.0 INTERNAL TABLES 9

3.1 Introduction 9

3.1.1 Linear Structure 9

3.1.2 Hash Structure 10

3.1.3 Tree Structure 10

3.2 Mechanics of Table Generation 11

3.2.1 EBNF with Subroutine Calls 11

3.2.2 Local Syntax Checking 13

3.2.3 Table Generation 13

3.2.4 Global Syntax Checking 17

4.0 TABLE GENERATING ROUTINES FOR DDL 18

4.1 The Symbol Table 18

4.1.1 The Tree Structure and its Uses 18

4.1.2 Growth and Search Tree Algorithms 20

4.1.3 Tree Resturcturing Algorithm 23

4.1.4 Flow Charts 26

4.1.5 Examples 29

4.1.6 The Calling of the Symbol Table Entry Routine 29

4.2 The Data Tables 34

4.2.1 Usage of the Data Tables 34

4.2.2 Data Table Format Design Considerations 34

4.2.3 Subroutine Calls for Data Table Constructions 40

4.2.3.1 File Statement 40

4.2.3.2 Record Statement 40

4.2.3.3 Group Statement 41

4.2.3.4 Field Statement 42

4.2.3.5 Length Statement 43

4.2.3.6 Count Statement 44

4.2.3.7 Card Statement 44

4.2.3.8 Tape Statement 44

4.2.3.9 Disk Statement 45

4.2.3.10 Convert Statement 46

4.2.3.11 Scan Statement 46

4.3 Example of Symbol Table and Data Table Creation 47

5.0 CONCLUSIONS 57

BIBLIOGRAPHY 60

1.0 INTRODUCTION

1.1 The Need For A DDL. The need for an efficient method of converting data for use in different programs or in different computer installations has long been recognized by most EDP users. Organization of data can presently be handled by use of data description facilities contained in operating systems and data management systems, programming languages, or in user-written software. Usually, the method chosen is useful for a particular computer and cannot be transferred to a different system due to hardware and software incompatibilities. In addition, one user's organization of data cannot be efficiently communicated to another as most data organization is implicit in the software used. Other restrictions may force the individual to write special conversion routines in order to accomplish an interchange of data.

The DDL research group collaborated to design and build a utility which would convert data between programs and/or systems, and whose power would be great enough to encompass most existing and most future programming languages and computer systems. This utility was to be a compiler, written in PL/1, built to translate a Data Definition Language (DDL) designed by D. P. Smith[5], with major modifications.

1.2 Summary of DDL Capabilities. The DDL processor was designed to satisfy two important requirements of data interchange:

1) data definition (organization)

2) data translation (movement and/or conversion)

The initial step towards simplifying data interchange was to make data and its organization independent of machines and their processors. This

was accomplished by using a language for describing data separate from the language for processing data. The DDL provides the descriptive language while the DDL processor is a set of programs which will perform the translation of the data described in the language. The capabilities are summarized below*:

a) INTERFACING FILES WITH DIFFERENT PROGRAMS AND PROGRAMMING LANGUAGES

Frequently files created by one program cannot be processed by another program or by another program written in a different programming language. These conflicts can be eliminated using the DDL processor to convert the files into a structure compatible for processing by the other program.

b) INTERFACING FILES WITH DIFFERENT OPERATING SYSTEMS AND DIFFERENT DATA MANAGEMENT SYSTEMS

Files created under one operating system or data management system cannot, in general, be processed by a different operating or data management system. With DDL, the conversion of files for processing by either operating system or data management system can be achieved.

c) INTERFACING FILES WITH NEW INSTALLATIONS

Advancing technology and increased requirements necessitate

---

*For the present implementation read "Sequential Files" for "Files" in sections a) through e)

phasing out of old computers and replacement by new systems. The DDL would enable files to be prepared for such transfers.

d) EXTRACTION OF DATA FROM FILES

If only a small amount of data in a file is used by a program, it is more efficient to create a smaller file consisting only of the useful data. The DDL allows for the creation of many files from one file.

e) INTERFACING FILES TO USE NEW DEVICES

Advances in technology necessitate introduction of new input/output devices which enhance the cost effectiveness of the system in use. The change in the new I/O devices can be facilitated by the DDL processing of the old files onto the new devices.

f) USE AS A HUMAN COMMUNICATION LANGUAGE

It is hoped that DDL will be used as a standard language to describe data structures in a precise manner, just as BNF is now used to describe the syntax of many languages.

1.3 Important Features of the Design. The DDL processor consists of three major parts. The syntactic Analysis Program Generator (SAPG) uses the definitions of the DDL (in EBNF*) to generate the Syntactic Analysis Program (SAP). As its name implies, the SAP parses DDL source

*Extended Backus Naur Form (EBNF) will be discussed in Section 3.2.1

input, scanning for syntactic flaws. Concurrent with this action, subroutines are called to generate internal tables which are encodings of the DDL source statements. These tables are used for global syntax checking and subsequent code generation.

1.4 Internal Tables in the DDL Processor. In designing DDL there were two major philosophies with which the designers had to contend: (a) A multiple-pass compiler in which DDL source would be parsed by a lexical routine, the output of which would be wholly rewritten in storage (core or peripheral) for subsequent syntax analysis, and re-rewritten in storage for final code generation, and (b) A two pass method in which DDL source would be lexically parsed, these units individually passed to a syntax analysis and statement encoding phase, and this data stored for the code generation phase.

DDL designers chose the latter method for two reasons. Firstly, this method allows speedier execution of code generation since the source code is in a simplified form. Noise (delimiters, banks, etc.) units are omitted as only the essence of the statements is retained, and many codes are employed. This eliminates the need for reparsing the DDL source input for code generation. Secondly, and more importantly, when future mechanical techniques are developed to perform code generation, it is most likely that the function of complete syntax checking (local and global) be carried out prior to any code generation. Encoded tables, created at syntax analysis time, permit this complete syntax checking phase to take place, enabling subsequent automatic code generation.

Local syntax analysis consists of a check for proper construction of individual source statements, standing alone. Global syntax analysis is responsible for verifying the legitimacy of references among several DDL source statements. Local syntax analysis can be carried out by simple comparisons between the input source and the EBNF description of the DDL. However, global syntax analysis requires storing of the data in temporary (or permanent, if necessary) tables to enable "walking" through the code to resolve all references.

The major drawback in using method (b) over (a) is that encoding forces an increased overhead in the running of the processor. However, this tradeoff is balanced by the fact that method (a) requires another pass (parse) of the source code during code generation.

1.5 Organization of the Thesis. Section 2 provides a short overview of the design of the processor, describing briefly the three major phases. Section 3 outlines the functions of the internal tables, illustrates, by example, how their designs are arrived at, and runs through the mechanics of the subroutine call facility of EBNF. Section 4 describes the formats of the internal tables (Symbol and Data) and sketches the algorithms for their creation. My conclusions are contained in Section 5.

2.0 OVERALL DESIGN OF THE DDL PROCESSOR

2.1 The Syntactic Analysis Program Generator. As can be readily inferred from its name, the syntactic analysis program generator (SAPG) outputs a PL/1 program (the syntactic analysis program) to perform the syntax checking on the DDL source statements. As seen in figure 1, the input to the SAPG is the description, in EBNF, of the particular DDL to be implemented. With this design, a hypothetical DDL user who wishes to transform his data base in a fashion the current DDL processor cannot handle need only supply the necessary additions to DDL in EBNF, and let the SAPG produce the syntax checking code automatically. Needless to say, the user must also provide the routines to generate internal tables and carry out the code generation.

2.2 The DDL Compiler. The DDL compiler consists of three phases:

a) Lexical

b) Syntactic Analysis and Table Generation

c) Code Generation

2.2.1 The Lexical Phase. The lexical phase is used to speed up the execution of syntax analysis of the DDL source code. It groups the DDL input strings into logical entities, and these units will be parsed by SAP as if they were single characters. Examples of such units are identifiers (names) and punctuation.

2.2.2 The Syntactic Analysis and Table Generation Phase. The syntactic analysis phase is responsible for the detection and flagging of errors

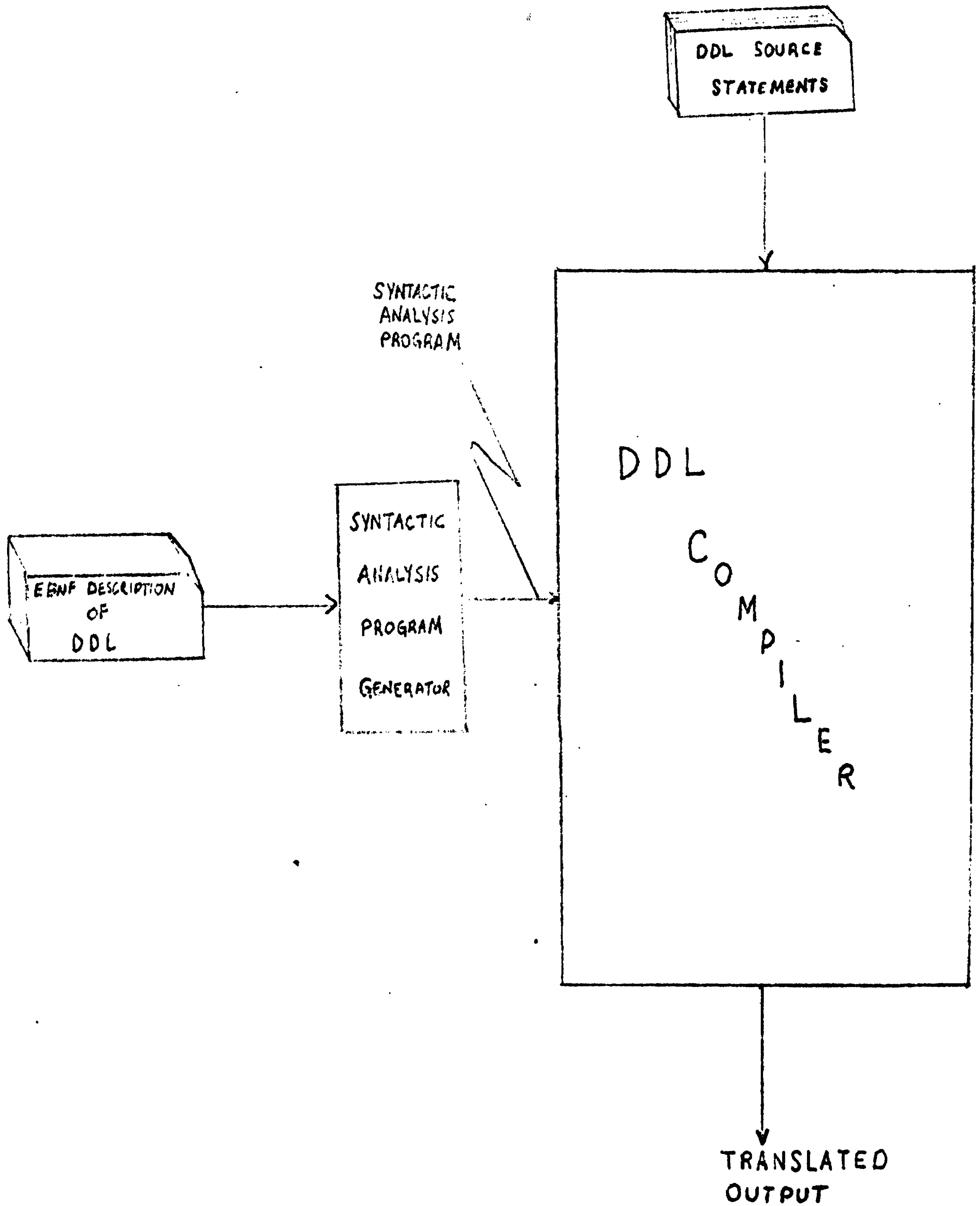


FIGURE 1

in the DDL source input. The SAPG generates PL/1 code to parse the input and, should an error be discovered, certain hand-written syntactic routines will be called to output a message informing the user the location and nature of the misconstruction.

Concurrent with this error detection phase is the table generation phase. At this time, routines are called whose functions are to capture semantic information contained in the DDL source statements and to build tables to preserve this data for use during code generation, as well as detection of global syntax errors. There will be several tables generated at this time: a single Symbol Table and many Data Tables.

2.2.3 Code Generation. After completion of the internal tables, the code generation phase massages the data contained therein and generates PL/1 code to define structures and/or perform data movement on the user's data base.

3.0 INTERNAL TABLES

3.1 Introduction. In most compiler applications internal tables are constructed to hold the pertinent information about the structure or statements contained in the programming language. These tables ordinarily take the form of vectors or matricies, although DDL uses push-downs (stacks) for its storage medium.

A symbol table is a storage device for items each having a unique name or key associated with them. The key is given and, on a table access, a pointer to the information associated with the item is returned by the table accessing mechanism. If this information is small in size, then the information itself may be returned, otherwise a pointer to where the data is stored is returned. The DDL processor uses the latter method.

At the same time the symbol table is created, data tables are also constructed by the DDL compiler. These tables store the information contained in the DDL source necessary for global syntax checking and further compilation by the code generation phase. They may be regarded as "files" in which the "real data" is located, and whose "names" are stored in the symbol table for convenient reference.

While most techniques for data table construction are Ad Hoc, there exist many formal methods of symbol table creation, three of which are described below.

3.1.1 Linear Structure. If successive entires of a symbol table are arranged in an arbitrary fashion, the average number of entries which

must be scanned in a table of length n, in order to locate a component, is n/2. This type of organization is linear since the search time depends linearly on the size of the table. Search time may be considerably shortened by building some structure into the symbol table. Two structures to be considered are Hash and Tree.

3.1.2 Hash Structure. Hashing techniques partition the set of all source codes by applying a function which maps them into a bit pattern with a lesser number of bits. This hashing function is usually chosen to satisfy two criteria:

1) The mapping from a source code to its bit pattern can be readily performed.
2) Source codes are mapped into bit patterns in an unpredictable and random manner.

A hashing function partitions the set of all source code into equivalence classes such that two source codes are equivalent if and only if they have the same bit pattern.

This method requires a function to be found which satisfies criteria 1, however DDL source names may be up to 31 characters in length and a function to perform the hashing would necessarily be very complex and time consuming. For this reason and the reason given in the previous section, Tree structuring of the DDL symbol table was chosen as a compromise.

3.1.3 Tree Structure. An efficient method of searching a structure is by repeated bisection of a list. Unfortunately, when a table is created entry by entry, the midpoint of the list is unknown and the bisection method can-

not be used. However, storing of the list as a binary tree achieves the same effect as structuring it as a "bisectable list". All entries less than the given symbol table entry are reached by going down a branch, and all entries greater, by going up. In spite of the fact that the paths in the tree will be of unequal lengths, the distance of the average node from the root is $\log_2 n$, where n is the number of nodes in the tree.

3.2 Mechanics of Table Generation.

3.2.1 EBNF with Subroutine Calls. The syntactic structure of the DDL is described via Backus-Naur Form with a few minor modifications. Sequences of characters enclosed in the brackets ⟨ ⟩ represent BNF meta-linguistic variables whose values are collections of symbols specified on the right of the "::=". An example of BNF (without the modifications) is:

⟨example⟩::=THIS⟨IS⟩ A ⟨SENTENCE⟩
|HELLO

The non-terminal (values enclosed in ⟨ ⟩) ⟨EXAMPLE⟩ is defined as follows:

("THIS", followed by the definition of ⟨IS⟩ (not supplied) in this example), the terminal "A", followed by the definition of ⟨SENTENCE⟩ (not supplied)) or "HELLO"

Any expression which fails both these alternatives does not belong to the class of statements defined by ⟨EXAMPLE⟩.

The extensions to BNF arise with the introduction of square brackets "[", "]" and the asterisk "*". This extended BNF (EBNF) has the distinction of facilitating human comprehension of repeating entities contained in statement definitions. Items enclosed in square brackets may appear

zero or one times, while square brackets followed by a star ("*") indicate an indefinate number of repititions. Any mark in a formula which is not a meta-linguistic symbol, or which is not enclosed in the meta-linguistic symbols ⟨,⟩, denotes itself. A further extension to EBNF, in the form of subroutine calls, is implemented to allow the compiler writer the flexability of outputting diagnostic messages as well as storing semantic information contained in the DDL source in one pass. The following is an example of EBNF with subroutine calls:

⟨LEFT_SIDE⟩::=⟨RIGHT_SIDE⟩/SUB_CALL/[,⟨RIGHT2⟩/SUB_CALL2/]*

If an input statement is to be identified as a ⟨LEFT_SIDE⟩ it must consist of: the ⟨RIGHT_SIDE⟩ definition, followed by 0 or more occurrences of ",⟨RIGHT2⟩". At syntactic analysis time, after correct recognition of the units ⟨RIGHT_SIDE⟩ and ⟨RIGHT2⟩, subroutines SUB_CALL1 and SUB_CALL2 will be enabled to capture the semantic information currently being scanned. If failure to recognize a unit comes to pass, then the parsing of the statement halts, no further subroutine calls in this production are made, and an error message is generated. The statement is discarded and parsing will continue with the next DDL statement[*].

There are two points that should be mentioned concerning the syntax of EBNF:

1) Subroutine calls may appear anywhere except between ⟨and⟩.
2) An EBNF statement line may be nothing more than subroutine calls.

---

[*] If the present production has an "|" (OR) symbol further on, parsing restarts with the definition following the "|".

3.2.2 Local Syntax Checking. Subroutine calls will be placed in appropriate locations in the EBNF in order to prepare the compiler for certain terminal symbol failures. This accomplished by employing a fail stack which is provided with suitable error message codes to be cited should a "failure" in the scanning occur. See French[1].

3.2.3 Table Generation. As in local syntax checking, the internal tables to be used for global syntax checking and code generation are constructed via the subroutine call facility of the EBNF. At appropriate points, calls are made to PL/1 routines which build the symbol table and data table entries. These tables are the stepping stones for the subsequent code generation phase which walks through them to determine the user's data base structure and/or data movement intentions. The tables are PL/1 based structures which are created, only after it has been determined, through local syntax routines, that the DDL statement under consideration is constructed correctly.

In order to fully understand the logic and the mechanics incorporated in the process of table generation, an involved example will be given.

EXAMPLE 1.

The familiar ARRAY declaration contained in many programming languages will be the illustration. In order to provide the global syntax checking phase and the code generation phase with data pertaining to the contents of the ARRAY declaration, table entries consisting of critical information must be constructed. These structures are given in Figure 2.

ARRAY declaration examples:

1) A ARRAY (1);

2) XI ARRAY (2:3);

3) A2B ARRAY (2:4,1:3,5);

Corresponding table entries (see Figure 2 for content definition):

1) STMT IDENTIFIER

ARRAY DECL.	•

STMT DEFINITION

A	1	1	0

2) STMT IDENTIFIER

ARRAY DECL.	•

STMT DEFINITION

X1	1	2	1	3

3) STMT IDENTIFIER

ARRAY DECL.	•

STMT DEFINITION

A2B	3	2	1	4	1	1	3	5	0

Using the contents of the data entries, global syntax routine walk through the information resolving all references, and code generation outputs the code necessary to define the structure. In order to build these entries subroutine calls, embedded at appropriate locations in the EBNF, are written. The EBNF with subroutine calls for the present example is as follows:

STMT IDENTIFIER	STATEMENT TYPE	POINTER TO DEFINITION

STMT DEFINITION	NAME	NUMBER OF DIMENSIONS	FIRST BOUND	FLAG FOR SECOND BOUND	SECOND BOUND	...	FIRST BOUND	FLAG FOR SECOND BOUND	SECOND BOUND

FIGURE 2

⟨ARRAY_DECLARATION⟩::=⟨NAME⟩/ARRAY_NAME/ ARRAY(⟨BOUNDS⟩
/DIMENSION/[,⟨BOUNDS /DIMENSION/]*);

⟨BOUNDS⟩::=⟨INTEGER⟩/FIRST_BOUND/[:⟨INTEGER/SECOND_BOUND/]

⟨NAME⟩::=/NAME_RECOGNIZER/

⟨INTEGER⟩::=/INTEGER_RECOGNIZER/

Explanation of subroutine calls (using example 3):

After recognition of "A2B" as a ⟨NAME⟩, SAP calls

1. ARRAY_NAME

This routine sets the STMT_TYPE entry in the table to the code for an ARRAY declaration statement. Then the POINTER_TO_DEFINITION entry is filled with the pointer value of the location where this statement's definition resides. The NAME entry is filled with "A2B", the name of the array. This value is still available as we have not attempted to scan another unit of input as yet. An initialization of the NO_OF_DIMENSIONS entry to 0 occurs here for subsequent incrementation by the subroutine DIMENSION.

After recognition of "(" and "2" (as an ⟨INTEGER⟩), SAP calls:

2. FIRST_BOUND

This routine fills in the FIRST_BOUND entry of the table with the current lexical unit ("2"). Since no foresight as to the possible occurrence of the second bound is available, the flag FLAG_FOR_SECOND_BOUND is set to 0. It will be overridden by a subsequent subroutine call if the second bound does indeed occur.

After entering the optional clause in ⟨BOUND⟩ **by recognizing** ":", and after recognizing the "4" as an ⟨INTEGER⟩, SAP calls:

3. SECOND_BOUND

This routine changes FLAG_FOR_SECOND_BOUND to a 1, signifying a presence. SECOND_BOUND is assigned the value "4" (the currently scanned unit).

SAP now calls:

4. DIMENSION

This routine increments the NO_OF_DIMENSIONS by one, thus providing the code generation phase with the correct number of dimensions in the array.

The optionality [, BOUNDS /DIMENSION/]* is satisfied by recognition of ",", located between the "4" and the "1" in the source input. The previous routines are re-executed, entering the "1" and the "3" into the table in the same fashion as before. Similarly, the "5" is placed into an entry in the table but, after completion of this entry, the above optionality is not satisfied (no "," in the input stream), SAP skips to recognize the unit following the "*" in the EBNF, accepts the ")" and the ";" as valid input characters, and halts this statement's parse with an indication to the routine of a successful recognition.

3.2.4 Global Syntax Checking. After construction of the tables is completed, control is passed to a routine which walks through the entries just created, verifying that all statement references are valid. For example, if a FILE statement in DDL references something other than a RECORD or Storage statement, an error flag is set. If no data table entry exists for some identifier in the symbol table, the name is flagged. In case of either error type, a message is printed and control is not passed to code generation.

4.0 TABLE GENERATING ROUTINES FOR DDL

4.1 The Symbol Table. As previously mentioned in Chapter 3, the symbol table is a binary tree structure whose entries are lexicographically ordered. The entries are the names of the DDL source statements, located at the statement head. In Figure 3a, the statement identifiers are RCD_NAME,F1, GRP1, GRP2, F2. It is necessary to connect each occurrence of a statement identifier in DDL with the data (the rest of the statement) accompanying it. This is accomplished in two steps:

1. The accompanying data is stored away into a data table entry.
2. A pointer in the symbol table to this storage location is set.

In DDL it is necessary that the data stored in Step 1. also contain a link to its name (thus creating a doubly-chained list). In this fashion, a great deal of intercommunication among DDL statements can occur. These interrelationships are exhibited in Figure 3b. Note that the links from within the data entries reference the names of statements in the symbol table. These references are crucial to both global syntax checking and code generation. The latter phase will use the information contained herein to generate proper structure declarations or data movement commands.

4.1.1 The Tree Structure and its Uses. If an identifier is encountered in the source input, a routine is called to locate the name in the symbol table and, if already present, return its location. Otherwise, an entry is opened for the name and the new location is returned. Using this process with an unstructured symbol table could prove very time consuming and

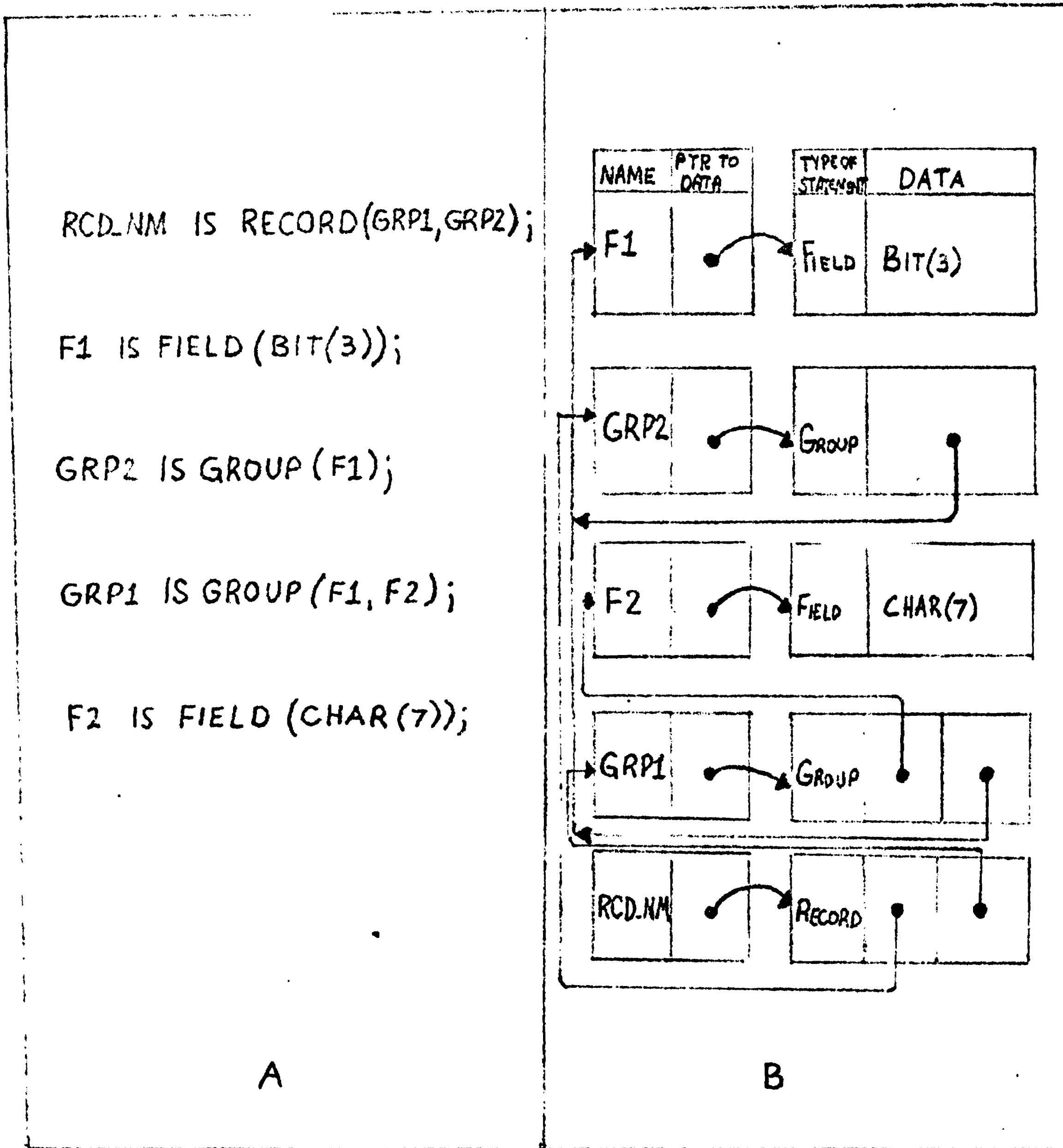


FIGURE 3

therefore, some structure is defined on the symbol table for which the seek time is minimized.

The composition of the symbol table will be a binary tree whose nodes contain the information illustrated in Figure 4. All items of information in the subtrees extending from a given node which are larger* than the item of information at that node will be in the subtree pointed to by the upward pointer. Similarly, all smaller items are in the subtree pointed to by the downward pointer. This structure is illustrated in Figure 5. If a subtree contains no items, a pointer to that subtree is considered to be a pointer to a null node.

4.1.2 <u>Growth and Search Tree Algorithms</u>. Such trees are easy to grow. The first item of information is placed in the tree at the root. Thereafter, each new item is placed in the tree by comparing it with the root and moving up or down depending on whether the new item is larger or smaller. This process is repeated at each node until an attempt is made to move to a null node. The item is then placed at this point in the tree. Section 4.1.4 contains the flow chart of this procedure.

As an example, consider adding the item "H" to the tree in Figure 5.

1) H>A-- move up

2) H<J1-- move down

3) H>E-- move up

4) H>G-- move up

Since there is not item up from G, H is attached at this point. Now some

---

*Any ordered relation may be used-DDL uses lexicographical ordering.

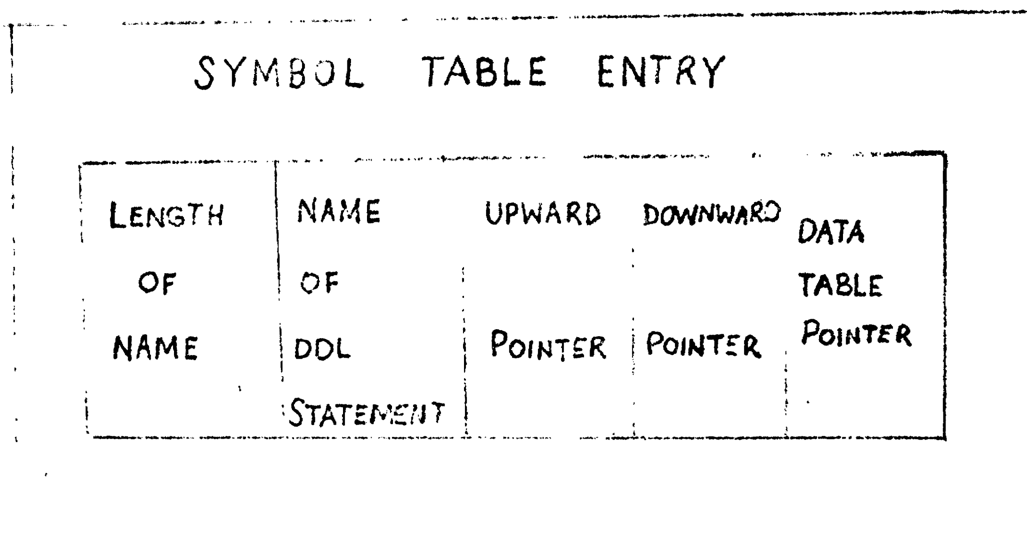


FIGURE 4

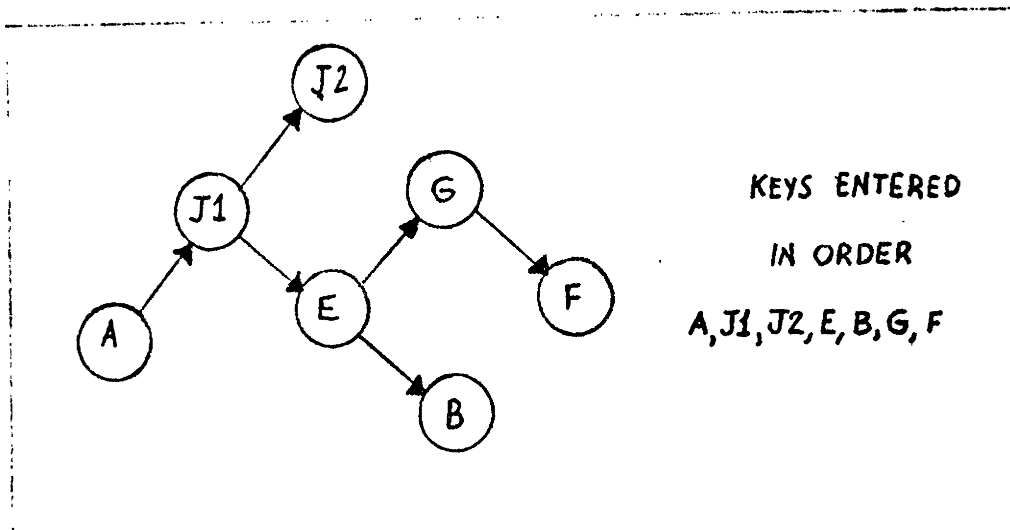


FIGURE 5

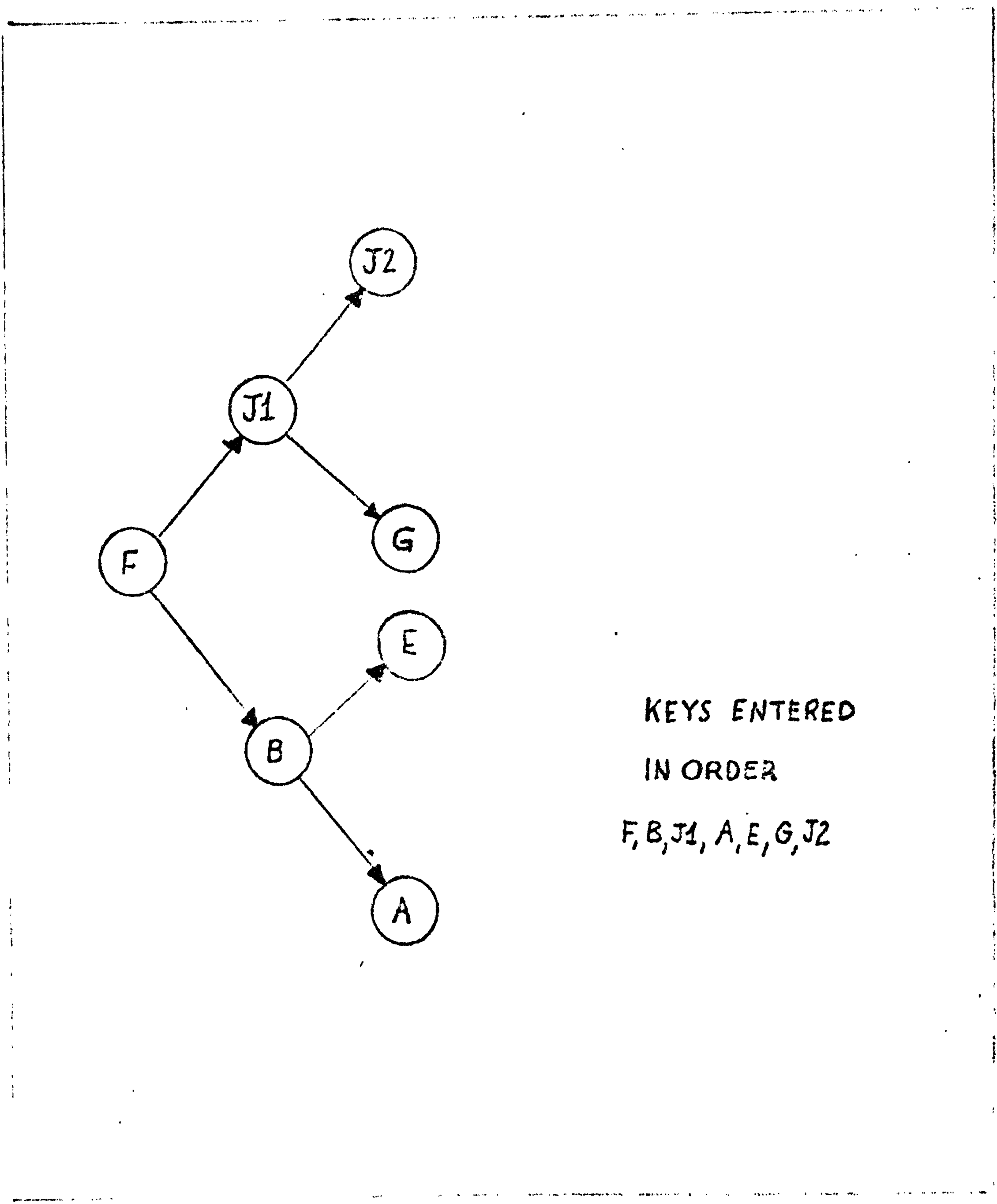


FIGURE 6

mathematical properties of the tree structure grown by this algorithm will be considered.

The shape of a tree containing a given set of n items depends on the order in which the items are encountered. For example, Figure 6 is constructed by considering the same items as in Figure 5 but in a different order. The algorithm thus generates a tree for each of the n! possible arrangements of n items; but not all the trees are distinct, as can be seen from Figure 7. In the analysis to follow it is considered that each of the n! permutations of n items is equi-probable. Thus some trees will be generated more often than others. It can be stated without any contradiction that an item can be searched for following exactly the same steps used to insert that item. It is reasonable to assume that the time required is proportional to the number of nodes visited. It is obvious that in Figure 6 twenty-two nodes must be visited to find each item, while Figure 7 requires 19 visits. Clearly, the tree in Figure 7 is not only better but optimum. An algorithm, whose flow chart is presented in Section 4.1.4, is presented below. It will convert any tree into its optimum tree.

4.1.3 Tree Restructuring Algorithm. The algorithm which resturctures the tree consumes time for execution. A natural question to ask is whether the time saved in searching a reorganized tree is greater than the time required for the conversion from the non-optimal to the optimal form. By referencing W. A. Martins & D. N. Ness[3], it can be deduced that there is some n beyond which application of the restructuring algorithm must result (on the average) in saving. However, in the present application, since the number of accesses from the table is not estimated to be high,

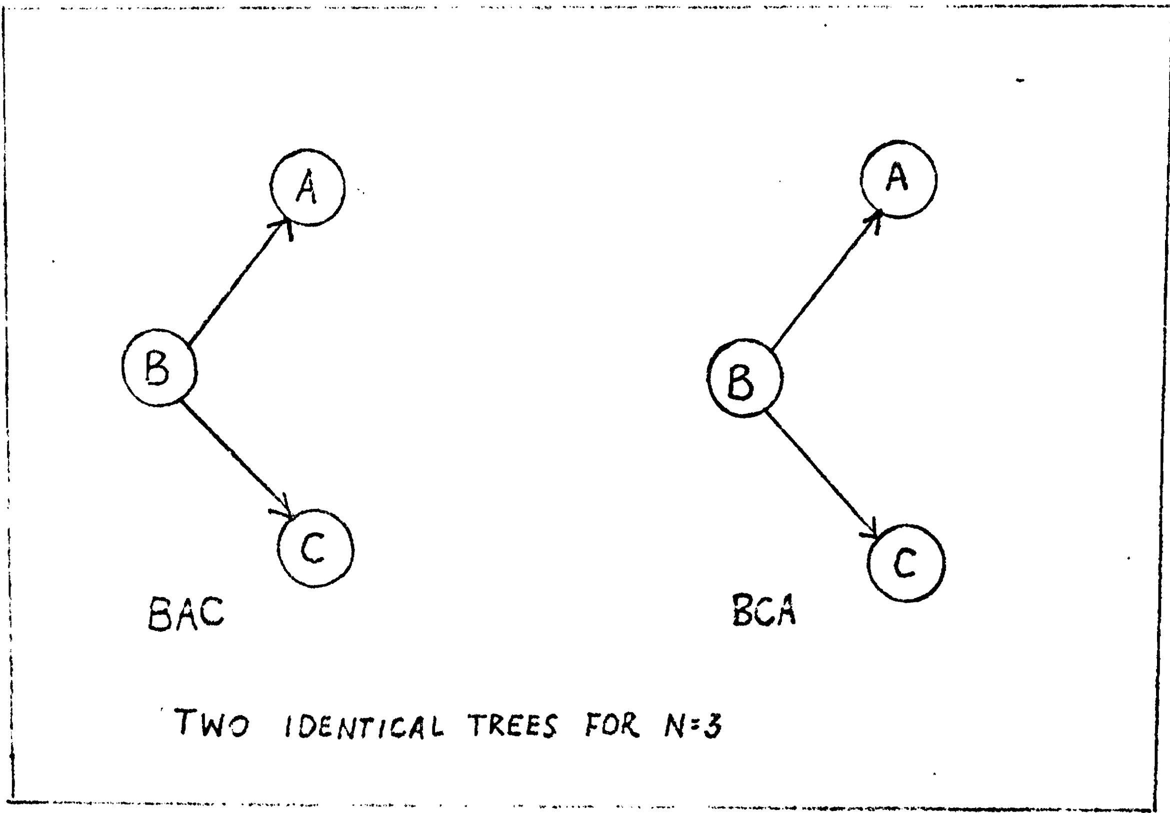
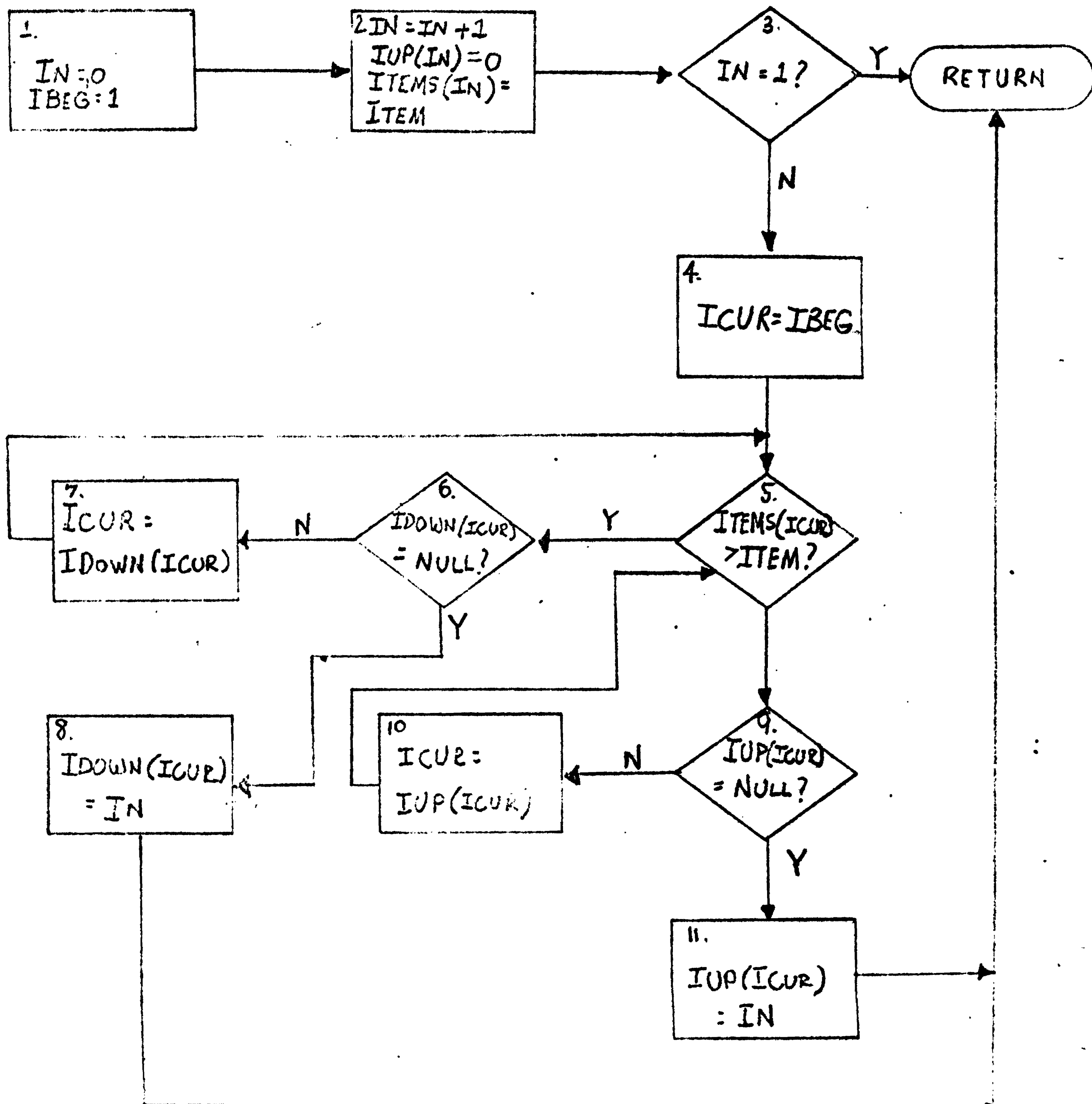


FIGURE 7

restructuring will be attempted on a trial and error basis. Based on this empirical evidence, restructuring will occur only after a certain number of entries have been processed. Present implementations place this number between twenty and thirty and alternates may be used.

4.1.4 Flow Charts.

a) Growth and Search Tree Algorithm



Explanation:

The data items input to the algorithm are stored in the variable ITEMS. Upward pointers are in IUP, an array. Similarly, downward pointers are in IDOWN. The current number of items is in IN, the beginning of the tree in IBEG, the current node to be used is labeled ICUR.

BOX LABELS	EXPLANATION
1.	Initialization of program variables.
2.	Put the input word into the array of nodes.
3.	Is this the first input? If yes-return.
4.	If not, place item in list.
5.	Determine if ITEM goes into upper sub-tree or lower sub-tree.
6.	Lower sub-tree is determined. Is node null?
7.	If not, set current node to this non-null node and return to 5 to restart.
8.	If line 6 is Yes, then insert value at this node. Return.
9.	If the answer to 6 is no, then an upward sub-tree is required for the placing of ITEM. Is the pointer to the upper sub-tree null?
10.	If not, set current node to point to this non-null node, and return to 5 to restart.

11. If answer to 9 is yes, insert the value into that node- Return.

b) Tree Restructuring Algorithm

Owing to the fact that the procedure contains recursive routine, I won't endeavor to flow chart this algorithm in the same detail as in the previous case. An English description of the steps to be followed will be provided.

The procedure IBEST returns as its answer, a pointer to the root node of the restructured tree. This procedure also establishes the environment for the other subroutines, IGROW and INEXT. IBEST computes IGROW(n), where n is the number of nodes in the tree to be restructured. It returns the result of this computation (which is the restructured tree) as its answer. The procedure IGROW(n) is responsible for constructing an optimum tree containing n nodes. It may be recursive, as it may call itself. It uses the procedure NEXT, which returns a pointer to the smallest node in the old tree the first time it is called, and a pointer to the smallest node, not previously returned, on each successive call. IGROW(n) can take three courses of action:

1) If n=0, return a pointer to a NULL node.
2) If n=1, call NEXT and return its result.
3) If n>1,
   a) Call IGROW($\lfloor (n-1)/2 \rfloor$)
   b) Call NEXT
   c) Call IGROW($\lceil (n-1)/2 \rceil$)

Then after the node pointed to as a result of b) by replacing its down pointer with the result of a), and its up pointer with the result of c). The procedure Next is given a pointer to the root of the original tree by IBEST. Each time it is called by IGROW it moves one step through the tree and returns the next node in ascending sequence. It also saves place in the tree for the next call by IGROW. Given a sub-tree, NEXT returns the nodes in the lower branch by calling itself recursively with this branch as an argument, then it returns the root node of the subtree, and then the nodes in the upper branch.

4.1.5 Examples. The PL/1 output contained in Figures 8 & 9 represents the tree structures shown in Figures 10 & 11 respectively. Data for Figure 8(10) was input and restructured, resulting in the structure of Figure 9(11).

4.1.6 The Calling of the Symbol Table Entry Routine. Statements in DDL are generally input with the following format (EBNF): ⟨IDENTIFIER⟩ IS ⟨STATEMENT⟩;

For example:

```
RCD IS RECORD(G1,G2);
GRP IS GROUP(F1,F2,F3);
```

To enter the names RCD, GRP into the symbol table, a routine called ENTESYM is called after recognition of the identifiers. Thus the EBNF with subroutine calls for the majority of the DDL source statements is:

⟨IDENTIFIER⟩/ENTESYM/ IS ⟨STATEMENT⟩;

ENTESYM returns a pointer to the location in the symbol table of the iden-

LOCATION:	UP_PTR:	DOWN_PTR:	DT_PTR:	SIZE	KEY
[illegible] → 523384	519152	519200	4278190080	31	GLORILU
519200	518624	519104	4278190080	31	CARLOS
519152	518672	518336	4278190080	31	JOSE_LUIS_123
519104	518576	4278190080	4278190080	31	ARTURO_RAMIREZ
518672	518528	518240	4278190080	31	NOMBRE_1[illegible]
518624	518384	4278190080	4278190080	31	EXAMPLE_1
518576	4278190080	4278190080	4278190080	31	BLANCO
518528	518480	4278190080	4278190080	31	PRUEBA
518480	518432	4278190080	4278190080	31	RAMIREZ
518432	518048	518288	4278190080	31	SOLOW
518384	4278190080	4278190080	4278190080	31	FRENCH
518336	518000	518144	4278190080	31	GROSS
518288	518096	518192	4278190080	31	RIN
518240	4278190080	4278190080	4278190080	31	MULLER
518192	4278190080	4278190080	4278190080	31	RECIO
518144	4278190080	517952	4278190080	31	GONZALES
518096	4278190080	4278190080	4278190080	31	SMITH
518048	4278190080	4278190080	4278190080	31	VALDEZ_IS COMMING
518000	4278190080	4278190080	4278190080	31	JORGE_GANA
517952	4278190080	4278190080	4278190080	31	GOD_FATHER

IBEG NOW POINTS TO 518000

LOCATION:	UP_PTR:	DOWN_PTR:	DT_PTR:	SIZE	KEY
523384	4278190080	4278190080	4278190080	(31)	GLORILU
519200	518624	4278190080	4278190080	31	CARLOS
519152	4278190080	4278190080	4278190080	31	JOSE_LUIS_123
519104	4278190080	4278190080	4278190080	31	ARTURO_RAMIREZ
518672	518528	4278190080	4278190080	31	NOMBRE_1[illegible]
518624	4278190080	4278190080	4278190080	31	EXAMPLE_1
518576	519200	519104	4278190080	31	BLANCO
518528	4278190080	4278190080	4278190080	31	PRUEBA
518480	518096	518240	4278190080	31	RAMIREZ
518432	518048	4278190080	4278190080	31	SOLOW
518384	517952	518576	4278190080	31	FRENCH
518336	4278190080	4278190080	4278190080	31	GROSS
518288	4278190080	4278190080	4278190080	31	RIN
518240	518672	519152	4278190080	31	MULLER
518192	518288	4278190080	4278190080	31	FECIO
518144	518336	4278190080	4278190080	31	GONZALES
518096	518432	518192	4278190080	31	SMITH
518048	4278190080	4278190080	4278190080	31	VALDEZ_IS COMMING
ROOT → 518000	518480	518384	4278190080	31	JORGE_GANA
517952	518144	523384	4278190080	31	GOD_FATHER

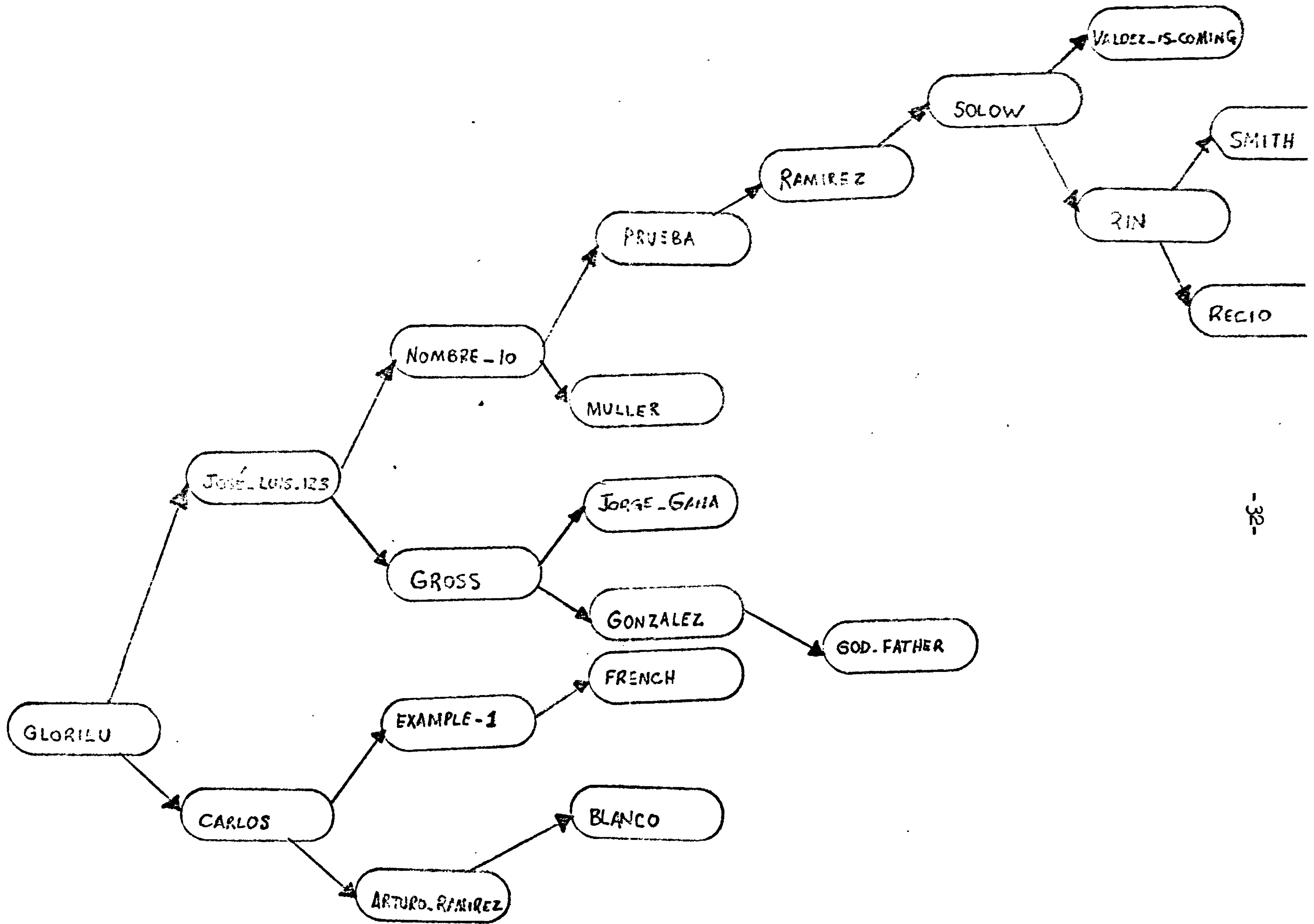


FIGURE 10

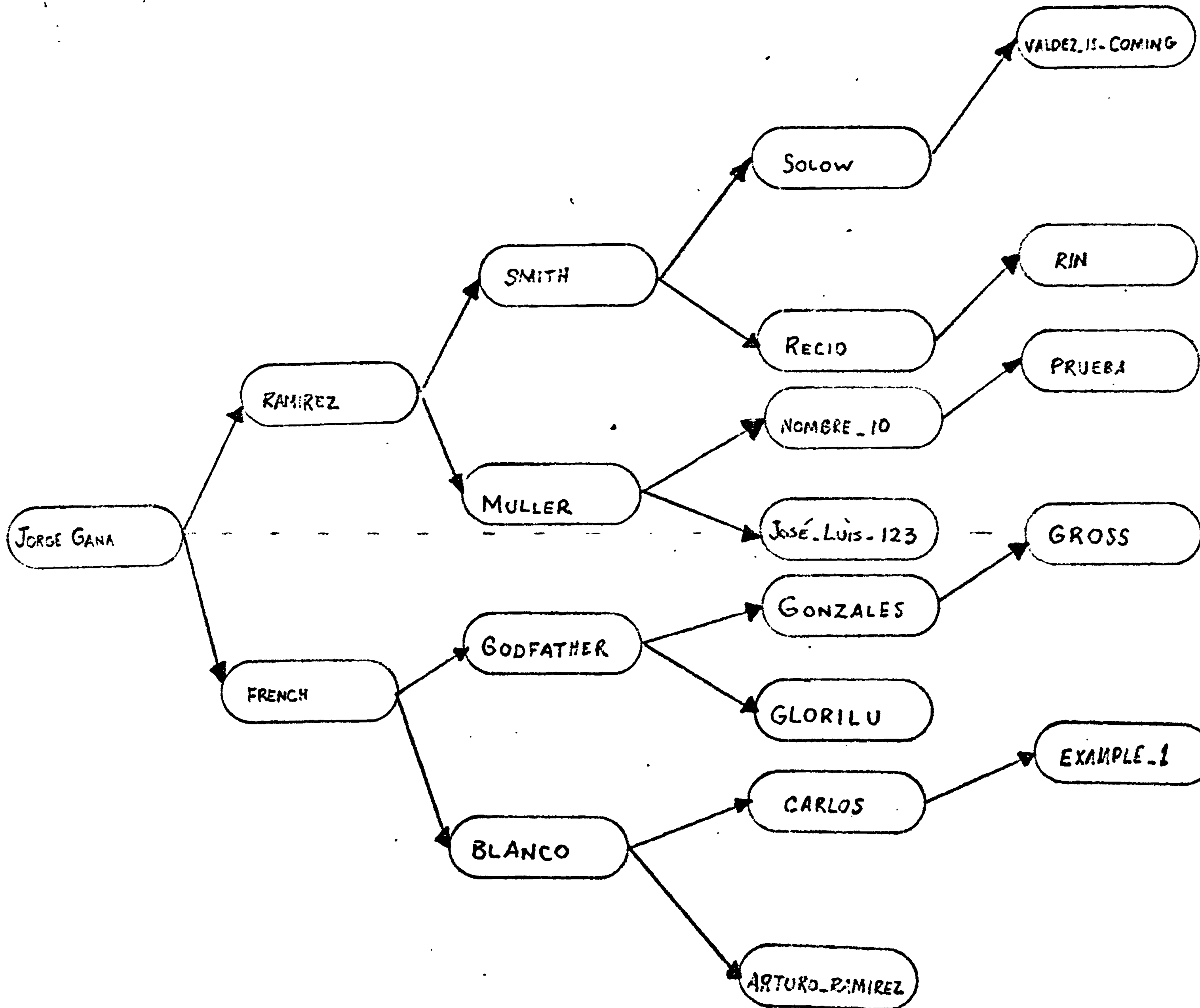


FIGURE 11

tifier just recognized.

4.2 The Data Tables.

4.2.1 Usage of the Data Tables. For every DDL statement certain relevant information must be stored in core for later use in the global syntax checking phase as well as in the code generation phase. To this end, each instance of a source statement initiates procedures which open data table entries whose function is the preservation of the pertinent information. The DDL data table designs are by no means unique and, given different designers/programmers, different designs for these tables most likely would be conceived. As long as the tables generated by the internal routines correspond exactly to the tables expected by the code generation phase any applicable construct may be used.

4.2.2 Data Table Format Design Considerations. If the code generation phase is to generate declaration and/or translation statements for a user's file structure, handwritten subroutines must utilize the appropriate data table information. Up to this point in the thesis no allusion to actual DDL source statements has been made. However, in order to relate the evolution of the data table designs, certain examples of DDL source will be examined.

Assume the following DDL source was input to the compiler:

```
(a) RCD IS RECORD(GRP1(2),GRP2);
(b) GRP1 IS GROUP(F1,F2(3));
(c) GRP2 IS GROUP(F1(2),F2);
(d) F1 IS FIELD(BIT(3)←'100');
(e) F2 IS FIELD(CHAR(2));
```

Statement (a) describes a record of a user's file composed of two groups: GRP1, which occurs twice and GRP2. GRP1 is specified in statement (b) wherein it is defined as a group consisting of two members: F1 and F2 which repeats three times. GRP2 is a group with members F1, repeating twice, and F2. F1 is a field of three bits initially assigned the value "100". F2 is a field of two characters with no initial assignment. This structure is pictorally represented in Figure 12.

In order to preserve this structure, the DDL compiler will create individual data tables, unique for each source statement. The global syntax checking routine will walk through these tables, verifying that all references to any statement are valid. The code generation phase is saddled with the responsibility of declaring the PL/1 structure representative of this description, using the encoded tables as guidelines. For this example, it is fairly obvious that the following PL/1 declaration describes the file presented in the example:

```
DCL 1 RCD,
      2 GRP1(2),
         3 F1 BIT(3) INITIAL('100'),
         3 F2 CHAR(2),
      2 GRP2,
         3 F1 BIT(3) INITIAL('100'),
         3 F2 CHAR(2);
```

It is necessary that the compiler provide the code generation phase with

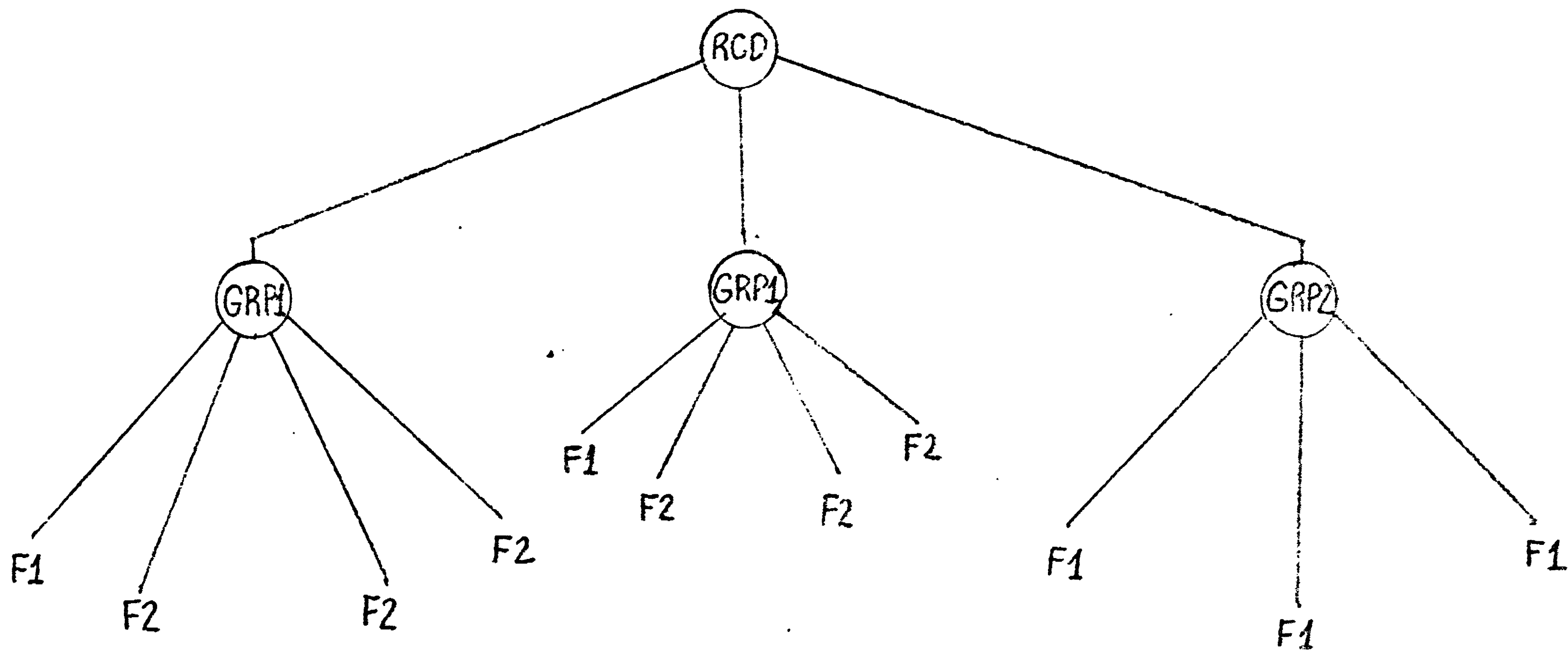


FIGURE 12

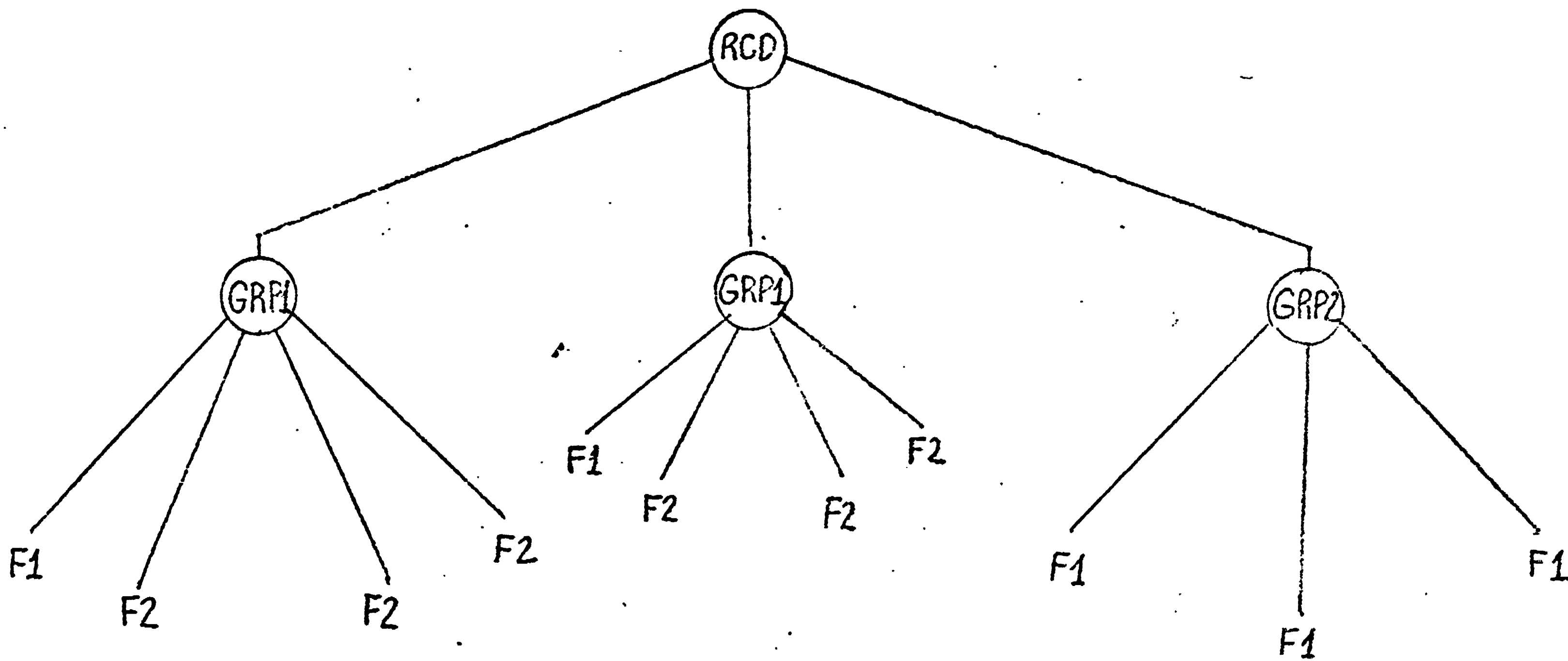


FIGURE 12

the encoded table containing sufficient data to declare the above structure.

As the record statement provided in the example now stands, Figure 13 furnishes a prototype of a RECORD STMT data table entry. Referring to this diagram, a simulation of the steps the code generation phase would take upon encountering this entry is given.

1. From examination of the CODE entry, determination of the TYPE (RECORD) of data takes place.

2. The pointer to the symbol table entry which contains the "name" of the source statement is followed, and using this value, the "DCL 1 RCD," line is generated.

3. The "number of groups" entry alerts the code generation phase to the number of accesses of contained group entries that must be performed.

4. The group entries are pointers to a symbol table entry containing their "names". This value, along with the subsequent "repitition number" entries, allow generation of the lines "GRP1(2)" and "GRP2".

5. The declaration is not complete as we must travel the pointers to locate the group entries' data tables as well as the field entries' data table. This is done to acquire the remaining information about the structure.

It must be pointed out that the RECORD STMT data table entry provided in the above example is a restriction of the actual entry used in DDL. It was used to simplify the discussion, and the reader is advised to consult APPENDIX B where he will find the expanded form used in the DDL processor.

DATA TABLE ENTRY FOR PROTOTYPE RECORD STATEMENT

CODE FOR RECORD STMT	PTR TO NAME OF RECORD STMT	NUM. OF GROUPS	GROUP #1	REPETITION NUMBER	GROUP #2	OUP #N	REPETITION NUMBER

FIGURE 13

4.2.3 Subroutine Calls for Data Table Constructions. The succeeding sections provide the EBNF with subroutine calls for the set of DDL statements. In order for the reader to comprehend the logic used for placement of the calls, it may be beneficial to consult Appendicies A and B. In addition, DDL source statements corresponding to the EBNF will be provided.

4.2.3.1 File Statement. With this statement the user describes his overall file structure. This consists of the record names contained therein, the code used (EBCDIC,BCD, or ASCII), and the medium of storage used (TAPE, DISK or CARD).

EBNF:

```
⟨FILE_STMT⟩::= FILE/DFILETG/(⟨RECORD_NAME⟩/FRN/,CHAR_CODE=
                    ⟨CODE⟩,STORAGE=⟨NAME⟩/FSN/)
⟨CODE⟩::= BCD/FC3/
         |ASCII/FC2/
         |EBCDIC/FC1/
```

EXAMPLE:

```
FILE_NAME IS FILE(REC_NAME,CHAR_CODE=EBCDIC,STORAGE=
                    TAPE_NAME);
```

4.2.3.2 Record Statement. The RECORD STMT describes the user's record structure by allowing specification of the groups contained in the file. The fields which are subordinate to no group are also specified in this statement. A password to the record structure may also be provided by the user.

EBNF:

```
⟨RECORD_STMT⟩::= RECORD/DRCDTG/(⟨NAME_LIST⟩NL_R/[,⟨NAME_LIST⟩
                 /NL_R/]*[,LOCK='⟨NAME⟩/RCRIT/'],REC_SIZE=
                 ⟨REC_SIZE⟩)/ALLO_R/
⟨NAME_LIST⟩::=⟨NAME⟩/NL1/[⟨OCCURRENCE⟩]
⟨OCCURRENCE⟩::= (⟨MIN_OCC⟩/NLF/[:⟨MAX_OCC⟩/NLSB/⟨CRITERION⟩])
⟨MIN_OCC⟩::=⟨INTEGER⟩/NLI/
⟨MAX_OCC⟩::=⟨INTEGER⟩/NLI/
⟨CRITERION⟩::= PRE_CRIT='⟨NAME⟩/PC/
              |POST_CRIT='⟨NAME⟩/NLC/
⟨REC_SIZE⟩::= FIXED(⟨INTEGER⟩/RCDBLK/)
             |VARIABLE(⟨INTEGER⟩/RCD_V/)
```

EXAMPLE:

```
RECORD_NAME IS RECORD(GRP1(1:8),PRE_CRIT='LABCRT',GRP2(2),
                        LOCK='LOCKCRT',REC_SIZE=VARIABLE(7));
```

4.2.3.3 Group Statement. The group statement function much in the same way as the record statement except at a lower level in the file structure. It may contain groups and fields.

EBNF:

```
⟨GROUP_STMT⟩::= GROUP/DGRPTG/(⟨NAME_LIST⟩/NL_G/[,⟨NAME_LIST⟩
                /NL_G/]*)/ALLO_G/
⟨NAME_LIST⟩::= same as in RECORD STMT.
```

EXAMPLE:

```
GRP IS GROUP(GRP2(1),FLD(1:8),POST_CRIT='CRT');
```

4.2.3.4 <u>Field Statement.</u> This lengthy (in EBNF) statement describes the lowest level of the DDL file structure-the fields. There are many options that can be specified, and by careful scrutiny of the EBNF most can be located and understood.

```
EBNF:

<FIELD_STMT>::= FIELD/DFLDTG/(<TYPE>[<DELIMITER>][<CONVERSION>])
                  /ALLO_F/

<TYPE>::= BIN/B1/[(<LENGTH>)][<BIT_ASSGN>]
          |CHAR/C1/[(<LENGTH>)][<CHAR_ASSGN>/C2/]
          |NUM_PICTURE=<NUM_PICTURE_SPEC>/N/[<NUM_ASSGN>]
          |CHAR_PICTURE=<CHAR_PICTURE_SPEC>/C/[<CHAR_ASSGN>/CP2/]

<LENGTH>::= */L1/
            |<LABEL>/L5/
            |<REF_NAME>/L2/
            |<PARAM_STMT>L3/
            |<INTEGER>/L4/

<CHAR_ASSGN>::= ←'<CHAR_STRING>/CA/'
                |←'<SOURCE_NAME>/BA1/'

<BIT_ASSGN>::=  '<BIT_STRING>/BA/'
               |←'<SOURCE_NAME>/BAT/'

<NUM_ASSGN>::= '←<NUM_STRING>/NA/'
               |←'<SOURCE_NAME>/BA2/

<BIT_STRING>::=/BITSTRING recognizer/

<CHAR_STRING>::=/CHAR_STRING recognizer/

<NUM_STRING>::=/NUM_STRING recognizer/

<LABEL>::=/LABEL recognizer/

<NUM_PICTURE_SPEC>::=/NUM_PICTURE recognizer/

<CHAR_PICTURE_SPEC>::=/CHAR_PICTURE recognizer/
```

```
⟨DELIMITER⟩::= ,DELIM='⟨PUNCT_MARK⟩/DLM/'

⟨CONVERSION⟩::=⟨NAME⟩

⟨SOURCE_NAME⟩::=⟨PARAM_STMT⟩/PAR/
                |⟨NAME⟩/NS/[(⟨SUBSCRIPT⟩)]
                 [ .⟨NAME⟩/NS2/[(⟨SUBSCRIPT⟩)]]*/ALLO_T/
                  [,POS=⟨POS⟩]

⟨SUBSCRIPT⟩::=⟨BOUND⟩/LS/[:⟨BOUND⟩/US/

⟨BOUND⟩::=⟨REF_NAME⟩/RN/
         |⟨PARAM_STMT⟩/PM/
         |⟨INTEGER⟩/INT/

⟨POS⟩::=⟨REF_NAME⟩/RNP/
       |⟨PARAM_STMT⟩/PMP/
       |⟨INTEGER⟩/INTP/
       | */SP/
       |⟨LABEL⟩/LBP/
```

EXAMPLE:

```
FIELD_NAME IS FIELD(BIN⟨=A(1:1).B(0:2),POS=7);
```

4.2.3.5 Length Statement. Occasionally some field may have a value which is based upon the length of another record, field, or group. This statement allows the user to specify these possibilities.

EBNF:

```
⟨LENGTH_STMT⟩::= DDL_LENGTH/DLGNTG/(⟨DATA_NAME⟩/LDN/)

⟨DATA_NAME⟩::=⟨REF_NAME⟩
```

EXAMPLE:

```
FIELD_NAME IS FIELD(CHAR(7)⟨=LENGTH(GRP1.FLD3(9));
```

4.2.3.6 Count Statement. As in the case of the Length Stmt.. this allows the user to provide a value based on the number of occurrences of another record, group, or field.

```
EBNF:
⟨COUNT_STMT⟩::=DDL_COUNT/DCNTTG/(⟨DATA_NAME⟩/CDN/)
⟨DATA_NAME⟩::=⟨REF_NAME⟩
EXAMPLE:
FIELD_NAME IS FIELD(CHAR(7)⟨=COUNT(GRP4.FLD7(8));
```

4.2.3.7 Card Statement. Information as to the medium of storage for the file must be passed to the processor. This statement specifies card storage.

```
EBNF:
⟨CARD_STMT⟩::= CARD/DCARDTG/
EXAMPLE:
CARD_NAME IS CARD;
```

4.2.3.8 Tape Statement. Tape storage is the medium used for the file.

```
EBNF:
⟨TAPE_STMT⟩::= TAPE/DTAPETG/(⟨TAPE_DATA_CTL_BLOCK⟩)
⟨TAPE_DATA_CTL_BLOCK⟩::=⟨RECORD_FORMAT⟩,VOL_NAME=⟨NAME⟩/VOL/
                        [,NO_TRKS=⟨NO_TRKS⟩/TT/]
                        [,PARITY=⟨PARITY⟩/TE/]
                        [,DENSITY=⟨DENSITY⟩/TL/]
                        ,
```

[,REC_MODE=⟨REC_MODE⟩/F1/]

[,TAPE_LABEL=⟨TAPE_LABEL⟩/TPL/]

[,START_FILE=⟨INTEGER⟩/INT1/]

[,CTL_CHAR=⟨CTL_CHAR⟩/CC/]

⟨RECORD_FORMAT⟩::= FIXED/REF/(⟨BLOCK_SIZE⟩/FBLKS/[,⟨RECORD_SIZE⟩
/FRSIZE/])

|VARIABLE/RFV/(⟨MAX_BLOCK_SIZE⟩/VBLKS/
[,⟨MAX_RECORD_SIZE⟩/VRSIZE/])

|VAR_SPANNED/RFVS/(⟨MAX_BLOCK_SIZE⟩/VBLKS/
[,⟨MAX_RECORD_SIZE⟩/VSRSIZE/])

|UNDEFINED(⟨MAX_BLOCK_SIZE⟩/UBLKS/)

EXAMPLE:

TAPE_NAME IS TAPE(FIXED(8,2),VOL_NAME VOL003,NO_TRKS#7
,PARITY=ODD,DENSITY=800,TAPE_LABEL=LBL003
,START_FILE=2,CTL_CHAR=@)

4.2.3.9 <u>Disk Statement</u>. As in the previous structure, this statement is referenced via the file statement and its function is to describe the disk storage of the user's file.

EBNF:

⟨DISK_STMT⟩::=DISK/DDISKTG/(⟨DISK_DATA_CTL_BLOCK⟩)

⟨DISK_DATA_CTL_BLOCK⟩::=⟨RECORD_FORMAT⟩/DRF/,VOL_NAME=
⟨NAME⟩/VOL2/

[,INT_NAME=⟨NAME⟩/DNM1/]

[,UNITS=⟨TYPEDSK⟩/DTYP/]

[,SPACE=⟨PARAMETERS⟩/DPARS/]

⟨RECORD_FORMAT⟩::= same as in TAPE STMT.

⟨PARAMETERS⟩::=⟨UNITS⟩,⟨QUANTITY⟩/PQ/[,⟨INCREMENT⟩/I1/][,RLSE/RLS/]

⟨UNITS⟩::= TRACKS

|CYLINDERS/CY/

|⟨INTEGER⟩/INT2/

⟨QUANTITY⟩::=⟨INTEGER⟩

⟨INCREMENT⟩::=⟨INTEGER⟩

TYPEDSK ::= 2314

|2311

|3300

EXAMPLE:

DISK_NAME IS DISK(FIXED(5),VOL_NAME=VOL006,INT_NAME=NM);

The following two statements are data movement commands, and provide code generation with data relevant to the structure mappings from the user's source to target file.

4.2.3.10 Convert Statement. This statement alets the compiler to the files (source and target) that the user is employing for his conversion.

EBNF:

⟨CONVERT_STMT⟩::=CONVERT/DCONVTG/(⟨FILE_NAME⟩/CS/ INTO ⟨FILE_NAME⟩/CT/

⟨FILE_NAME⟩::=⟨NAME⟩

EXAMPLE:

CONVERT(FILEA INTO FILES);

4.2.3.11 Scan Statement. Although no new entry is created by the routines in this statement, the RECORD data table entry is modified. The SCAN STMT specifies the order that groups within the record are to be parsed at code generation time. This information must necessarily be

provided if a field's values depend on another field, or combination of others. The position within the field at which scanning must occur is also provided in this statement.

EBNF:

```
⟨SCAN_STMT⟩::= SCAN/DSCANTG/(REC=⟨RECORD_NAME⟩/SCREC/
                                     :⟨GROUP_NAMES⟩[,⟨GROUP_NAMES⟩]*)
                                     /ALLSCAN/
⟨GROUP_NAMES⟩::=⟨NAME⟩/GN1/[(⟨POSITION⟩)]
⟨POSITION⟩::=⟨LABEL⟩/P1/
            |⟨INTEGER⟩/P3/
⟨RECORD_NAME⟩::=⟨NAME⟩
```

EXAMPLE:

```
SCAN(REC=REC_NAME:GRP3(2),GRP1,GRP2(2));
```

4.3 Example of Symbol Table and Data Table Creation. The DDL source for this example follows immediately and the files described therein are illustrated in Figure 14. Figures 15a and 15b portray graphically the symbol table and data table structures that would be created by the table generating routines.

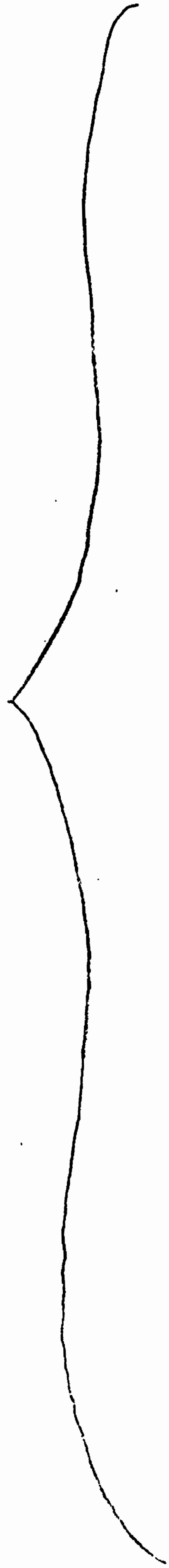
```
SFLE IS FILE(SRCD,CHAR_CODE=BCD,STORAGE=SRC_CRD);
SRCD IS RECORD(SGRP1(2),SGRP2(3));
SGRP1 IS GROUP(SFLD1(4),SFLD2(2));
SGRP2 IS GROUP(SFLD1(3),SFLD3(2));
SFLD1 IS FIELD(BIN(SFLE.SGRP2(1).SFLD1(2));
SFLD2 IS FIELD(CHAR(7));
```

```
SFLD3 IS FIELD(CHAR_PICTURE='AAX');
SRC_CRD IS CARD;


TFLE IS FILE(TRCD,CHAR_CODE EBCDIC,STORAGE=TARTAPE);
TRCD IS RECORD(TGRP1(3),TGRP2,TGRP2(2));
TGRP1 IS GROUP(TFLD1(2));
TGRP2 IS GROUP(TFLD2(2),TFLD3));
TGRP3 IS GROUP(TFLD3,TFLD4);
TFLD1 IS FIELD(BIN<-'100');
TFLD2 IS FIELD(CHAR<=SGRP1(2).SFLD2(1));
TFLD3 IS FIELD(CHAR_PICTURE='AAX'<=SGRP2(3).SFLD3(2));
TFLD4 IS FIELD(CHAR<-'ABC');
TARTAPE IS TAPE(FIXED(80),VOL_NAME=VOL003);


SCAN(REC=SRCD:SGRP2,SGRP1,SGRP3);
CONVERT(SELE INTO TFLE);
```

SRCD
SFLE



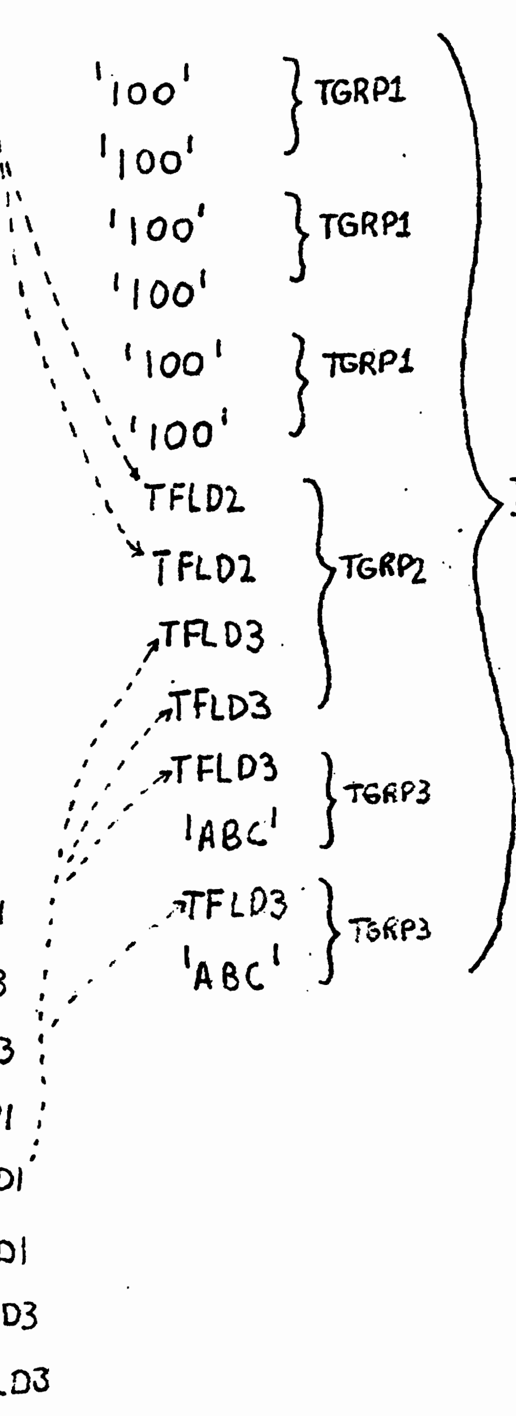
SGRP1
SFLD1
SFLD1
SFLD1
SFLD1
SFLD2
SFLD2

SGRP1
SFLD1
SFLD1
SFLD1
SFLD1
SFLD2
SFLD2

SGRP2
SFLD1
SFLD1
SFLD1
SFLD3
SFLD3

SGRP2
SFLD1
SFLD1
SFLD1
SFLD3
SFLD3

SGRP2
SFLD1
SFLD1
SFLD1
SFLD3
SFLD3



Loc	DDL STATEMENT NAME	UP POINTER	DOWN POINTER	DATA TABLE POINTER
S1	SFLE	NOT SUPPLIED FOR THIS ILLUSTRATION	NOT SUPPLIED FOR THIS ILLUSTRATION	D1
S2	SRCD			D2
S3	SRC_CRD			D11
S4	SGRP1			D3
S5	SGRP2			D4
S6	SFLD1			D5
S7	SFLE.GRP2(1).SFD1(1)			NULL
S8	SFLD2			D7
S9	SFLD3			D9
S10	TFL3			D12
S11	TRCD			D13
S12	TARTAPE			D29
S13	TGRP1			D14
S14	TGRP2			D15
S15	TGRP3			D16
S16	TFLD1			D17
S17	TFLD2			D20
S18	TFLD3			D23
S19	TFLD4			D26

FIGURE 15a

Loc

D1	FILE	S1	0	S2	S3	2

Loc

D2	RECORD	S2	0	NONE	2	S4	1	2	NULL
0	0	NULL	0	NONE	0	NONE	0		
0	NULL	S5	1	3	NULL	0	0	NULL	
0	NONE	0	NONE	0	0	NULL			

Loc

D3	GROUP	S4	2	S6	1	4	NULL	0	0
NULL	0	NONE	0	NONE	S7	1	2	NULL	
0	0	NULL	0	NONE	0	NONE			

DATA TABLES

FIGURE 15 b

Loc

D4	GROUP	S5	2	S6	1	3	NULL	0	0

NULL	0	NONE	0	NONE	S8	1	2	NULL

0	0	NULL	0	NONE	0	NONE

Loc

D5	FIELD	S6	0	D6	0	NONE	0	0

Loc

D6	0	3	S7	-1	NONE	0	NULL

Loc

D7	FIELD	S8	0	D8	0	NONE	0	0

Loc

D8	1	1	NULL	7	NONE	0	NULL

Loc

D9	FIELD	S9	0	D10	0	NONE	0	0

Loc

D10	1	1	NULL	3	NONE	0	NULL

FIGURE 15b (CONT.)

Loc

D11	CARD	S3	0	80	80

Loc

D12	FILE	S10	0	S11	S12	0

Loc

D13	RECORD	S11	0	NONE	3	S13	1	3	NULL	0

0	NULL	0	NONE	0	NONE	0	0	NULL	S14

0	0	NULL	0	0	NULL	0	NONE	0	NONE

0	0	NULL	S15	1	2	NULL	0	0	NULL

0	NONE	0	NONE	0	0	NULL

Loc

D14	GROUP	S13	1	S16	1	2	NULL	0	0

NULL	0	NONE	0	NONE

FIGURE 15b (CONT.)

Loc

D15	GROUP	S14	2	S17	1	2	NULL	0	0	NULL

0	NONE	0	NONE	S18	0	0	NULL	0	0

NULL	0	NONE	0	NONE

Loc

D16	GROUP	S15	2	S18	0	0	NULL	0	0	NULL

0	NONE	0	NONE	S19	0	0	NULL	0	0

NULL	0	NONE	0	NONE

Loc

D17	FIELD	S16	0	D18	0	NONE	0	0

Loc

D18	0	-1	NULL	0	NONE	1	D19

Loc

D19	3	100

FIGURE 15b (CONT.)

Loc

D20	FIELD	S17	0	D21	0	NONE	0	0

Loc

D21	1	-1	NULL	0	NONE	3	D22

Loc

D22	NULL	-1	0	NULL	NONE	2	S4	1
1	NULL	2	0	0	NULL	0	S8	1	1
NULL	1	0	0	NULL	0				

Loc

D23	FIELD	S18	0	D24	0	NONE	0	0

Loc

D24	1	1	NULL	3	NONE	3	D25

Loc

D25	NULL	-1	0	NULL	NONE	2	S5	1
1	NULL	3	0	0	NULL	0	S9	1	1
NULL	2	0	0	NULL	0				

FIGURE 15b (CONT.)

Loc								
D26	FIELD	S19	0	D27	0	NONE	0	0

Loc							
D27	1	-1	NULL	0	NONE	1	D28

Loc		
D28	3	ABC

Loc									
D29	TAPE	S12	0	80	0	2	0	2	0
	VOL003	0	ƀƀƀƀ	0	A				

Loc			
D30	CONVERT	S10	S1

FIGURE 15b (CONT.)

5.0 CONCLUSION

Throughout the evolution of the DDL compiler certain ambivalences kept cropping up in project discussions. One particular area of concern was whether encoding of the source statements was economically beneficial (were internal tables needed?). The answer to this question is certainly not cut and dry and, in reality, can only be supplied if and when some other team tries to implement a compiler for DDL which does not encode the source but which reparses. Nevertheless, encoding of the input reduces the amount of work performed by code generation and permits the use of global syntax checking routines, separate from code generation. This dichotomy (local and global syntax checking separate from code generation) permits modifications to the compiler to be performed modularly, simplifying matters considerably.

However, economics was not the sole reason for performing this shuffling of data. There is one very crucial consideration that perhaps outweighs even the economic question-communication between our routines and the future users or compiler writers who will inevitably modify these routines. The internal tables have been designed to facilitate comprehension of the logic in global syntax checking and data preservation. For purposes of clarity, PL/1 structures are created to contain the encoded statements. Subsequent code generation routines refer to the data contained in these tables BY NAME. As an example, the entry for the record name in the FILE data table structure is referenced by FILE.RECORD_NAME. Seeing this qualified name in the code is enough of a clue to identify which structure is currently being dealt with.

Ordinarily, certain array positions, transparent to the reader unless well documented, would be agreed upon, by convention, to contain the information. Any individual who has had the unfortunate task of debugging someone else's logic will concur with my claims to the advantages of referencing data by name.

Storage optimization is always foremost in the minds of compiler designers as excessive storage will result in a very expensive processor. For this reason, certain techniques for space saving were employed. In the data table formats (APPENDIX A) are found many pointer entries referencing various DDL names. The pointers are used instead of the name themselves because they occupy only 1 word while names may be up to 32 characters. Thus a substantial saving of space may be realized if a name is frequently referenced.

In many instances, data table entries do not have a fixed structure (see REFER option, PL/1 F Compiler, Language Reference Manual). This means that they are allocated space only after it has been determined just how much information is to be stored in them. It is apparent that collection of this data must occur by way of temporary storage. These temporals were chosen to be PL/1 controlled variables (variables which act like pushdowns and whose allocation and de-allocation is toatlly programmer controlled) so that, after all information has been amassed, their storage allocation would be freed, thus reducing the amount of unused storage in the processor.

The use of EBNF with subroutine calls in the DDL compiler allowed every data table used for global syntax checking and code generation to be created in the same pass in which both lexical and local syntax analysis were performed. This meant that one pass over the source was performed in the ENTIRE compiler.

By designing the symbol table and data tables as doubly chained lists, the code necessary for walking through the structures was immensely simplified. Links were travelled from statement identifier to statement data and back again with relative ease and efficient speed.

It is hoped that the choice of encoding source statements will prove the right one. Whether the tradeoffs were beneficial or not it must be pointed out that, when future automatic programming techniques are developed, our DDL compiler has a distinct syntax phase and code generation phase, a separation which enhances the possibilities of mechanical code generation.

BIBLIOGRAPHY

1. French, A., "A Syntactic Analysis Program Generator", M.Sc Thesis, The Moore School of Electrical Engineering, University of Pennsylvania, August 1972. [In preparation]

2. Hopgood, F. R. A., "Compiling Techniques", Macdonald/Elsevier Computer Monographs, 1970.

3. Martin, W. A. and Ness, D. N., "Optimizing Binary Trees Grown With A Sorting Algorithm", Communications of the ACM, Vol. 15, No. 2, February 1972.

4. McKeeman, W. M. and Horning, J. J. and Wortman, D. B., "A Compiler Generator", Prentice-Hall, 1970.

5. Smith, D. P., "An Approach to Data Description and Conversion", Ph.D Dissertation, The Moore School of Electrical Engineering, University of Pennsylvania, Moore School Report #72-20, December 1971.

6. Wegener, P., "Programming Languages, Information Structures, and Machine Organization", McGraw-Hill, 1968.

7. French, A. and Ramirez, J. and Solow, H. and Prywes, N. S., "Design of the Data Description Language Processor", Annual Report, The Moore School of Electrical Engineering, University of Pennsylvania, Moore School Report #72-19, December 1971.

APPENDIX A

DATA TABLE FORMATS

The data table formats corresponding to the EBNF statements in Section 4.2.3 and the subroutines flow-charted in APPENDIX B appear in this section.

SYMBOL TABLE ENTRY.

```
DCL 1 ST_ENTRY BASED(ST_PTR),
      2 UP_PTR POINTER,
      2 DOWN_PTR POINTER,
      2 DT_PTR POINTER,
      2 SIZE  FIXED BINARY,
      2 KEY  CHAR(KEY_SIZE REFER(ST_ENTRY.SIZE));
```

DATA TABLE ENTRY FOR LENGTH AND COUNT STMT'S.

```
DCL 1 LENGTH_DDL    BASED(DTPTR),
      2 TYPE   FIXED BIN,
      2 DATA_NAME   POINTER;


DCL 1 COUNT_DDL   BASED(DTPTR),
      2 TYPE FIXED BIN,
      2 DATA_NAME    POINTER;
```

DATA TABLE ENTRY FOR CONVERT STMT.

```
DCL 1 CONVERT  BASED(DTPTR),
      2 TYPE  FIXED BIN,
      2 TARGET  POINTER,
      2 SOURCE  POINTER;
```

DATA TABLE ENTRY FOR FILE STATEMENT.

```
DCL 1 FILE BASED(DTPTR),
      2 TYPE FIXED BIN,
      2 SYM POINTER,
      2 BUFOFF FIXED BIN,
      2 RCD_NAME POINTER,
      2 STORAGE POINTER,
      2 CHAR_CODE FIXED BIN;
```

DATA TABLE FOR RECORD STMT.

```
DCL 1 RECORD BASED(DTPTR),
      2 TYPE FIXED BIN,
      2 SYM POINTER,
      2 LOCK CHAR(7),
      2 NO_MEM FIXED BIN,
      2 MEMBERS (NDUMMY REFER (RECORD.NO_MEM)),
        3 MEM_NAME POINTER,
        3 F_SUB_TYPE FIXED BIN,
        3 F_SUB_CONST FIXED BIN,
        3 F_SUB_VAR POINTER,
        3 S_SUB_TYPE FIXED BIN,
        3 S_SUB_CONST FIXED BIN,
        3 S_SUB_VAR POINTER,
        3 PRE_CRIT_FLAG BIT(1) ALIGNED,
        3 PRE_CRITERION CHAR(7),
        3 POST_CRIT_FLAG BIT(1) ALIGNED,
        3 POST_CRITERION CHAR(7),
        3 POS_FLAG FIXED BIN,
        3 POS_CONST FIXED BIN,
        3 POS_VAR CHAR(7);
```

DATA TABLE ENTRY FOR CARD STMT.

```
DCL 1 CARD BASED(DTPTR),
      2 TYPE FIXED BIN,
      2 SYM POINTER,
      2 FORMAT BIT(1) ALIGNED,
      2 NO_CARDS FIXED BIN,
      2 MODE_TYPE FIXED BIN;
```

DATA TABLE ENTRY FOR GROUP STMT.

```
DCL 1 GROUP BASED(DTPTR),
      2 TYPE FIXED BIN,
      2 SYM POINTER,
      2 NO_MEM FIXED BIN,
      2 MEMBERS (NDUMMY REFER (GROUPµNO_MEM)),
        3 MEM_NAME POINTER,
        3 F_SUB_TYPE FIXED BIN,
        3 F_SUB_CONST FIXED BIN,
        3 F_SUB_VAR POINTER,
        3 S_SUB_TYPE FIXED BIN,
        3 S_SUB_CONST FIXED BIN,
        3 S_SUB_VAR POINTER,
        3 PRE_CRIT_FLAG BIT(1) ALIGNED,
        3 PRE_CRITERION CHAR(7),
        3 POST_CRIT_FLAG BIT(1) ALIGNED,
        3 POST_CRITERION CHAR(7);
```

DATA TABLE ENTRY FOR DISK STMT.

```
DCL 1 DISK BASED(DTPTR),
       2 TYPE FIXED BIN,
       2 SYM POINTER,
       2 DISK_FORMAT,
         3 RCD_FORMAT_TYPE FIXED BIN,
         3 BLOCK_SIZE FIXED BIN,
         3 RECORD_SIZE FIXED BIN,
      2 SPACE,
        3 UNITS FIXED BIN,
        3 QUANTITY FIXED BIN,
        3 INCREMENT FIXED BIN,
        3 RLSE BIT(1) ALIGNED,
      2 VOL_NAME CHAR(7),
      2 DSNAME CHAR(30),
      2 DISK_TYPE FIXED BIN,
      2 REC_MODE FIXED BIN,
      2 CTL_CHAR BIT(1) ALIGNED;
```

DATA TABLE ENTRY FOR TAPE STMT.

```
DCL 1 TAPE BASED(DTPTR),
      2 TYPE FIXED BIN,
      2 SYM POINTER,
      2 TAPE_FORMAT,
        3 RCD_FORMAT_TYPE FIXED BIN,
        3 BLOCK_SIZE FIXED BIN,
        3 RECORD_SIZE FIXED BIN,
      2 DENSITY CHAR(1),
      2 NO_TRKS BIT(1) ALIGNED,
      2 LABEL_TYPE FIXED BIN,
      2 START_FILE FIXED BIN,
      2 VOL_NAME CHAR(6),
      2 PARITY BIT(1) ALIGNED,
      2 DSNAME CHAR(30),
      2 REC_MODE FIXED BIN,
      2 CTL_CHAR BIT(1) ALIGNED;
```

DATA TABLE ENTRY FOR FIELD STMT.

```
DCL 1 FIELD BASED(DTPTR),
      2 TYPE FIXED BIN,
      2 SYM POINTER,
      2 FLD_TYPE BIT(1) ALIGNED,
      2 FLD_DESC POINTER,
      2 FLAG_CONV BIT(1) ALIGNED,
      2 CONVERSION CHAR(7),
      2 FLAG_DELIM BIT(1) ALIGNED,
      2 DELIM_SIZE FIXED BIN,
      2 DELIMITER CHAR (NDUMMY REFER (DELIM_SIZE));

DCL 1 DESC BASED(DTPTR1),
      2 TYPE BIT(1) ALIGNED,
      2 LENGTH_TYPE FIXED BIN,
      2 LENGTH_PARAM POINTER,
      2 LENGTH_CONST FIXED BIN,
      2 LENGTH_LAB CHAR(7),
      2 ASSG FIXED BIN,
      2 ASSG_PTR POINTER;

DCL 1 BIT_ATT BASED(DTPTR2),
      2 SIZE FIXED BIN,
      2 BIT_STRING (NDUMMY REFER (BIT_ATT.SIZE));

DCL 1 CHAR_ATT BASED(DTPTR2),
      2 SIZE FIXED BIN,
      2 CHAR_STRING CHAR(NDUMMY REFER (CHAR_ATT.SIZE));

DCL 1 NUM_PICTURE BASED(DTPTR1),
      2 ASSG FIXED BIN,
      2 PIC_SOURCE_NAME POINTER,
      2LENGTH_CONST FIXED BIN,
      2 SIZE_PIC_SPEC FIXED BIN,
      2 PIC_SPEC CHAR( NDUMMY REFER (SIZE_PIC_SPEC));
```

```
DCL 1 SOURCE_NAME BASED(DTPTR3),
      2 PARAM_STMT POINTER,
      2 POS_FLAG FIXED BIN,
      2 POS_CONST FIXED BIN,
      2 POS_VAR POINTER,
      2 POS_LAB CHAR(7),
      2 NO_NAMES FIXED BIN,
      2 DDL_NAME(NDUMMY REFER (NO_NAMES));
        3 NAME POINTER,
        3 LOWER_SUB BIT(1) ALIGNED,
        3 LOWER_TYPE FIXED BIN,
        3 LOWER_PARAM POINTER,
        3 LOWER_CONST FIXED BIN,
        3 UPPER_SUB BIT(1) ALIGNED,
        3 UPPER_TYPE FIXED BIN,
        3 UPPER_PARAM POINTER,
        3 UPPER_CONST FIXED BIN;
```

APPENDIX B

FLOW CHARTS FOR THE DATA TABLE CONSTRUCTIONS

The routines that generate the PL/1 structures to be used for global syntax checking and code generation are presented in this section.

FILE STMT PROCEDURE

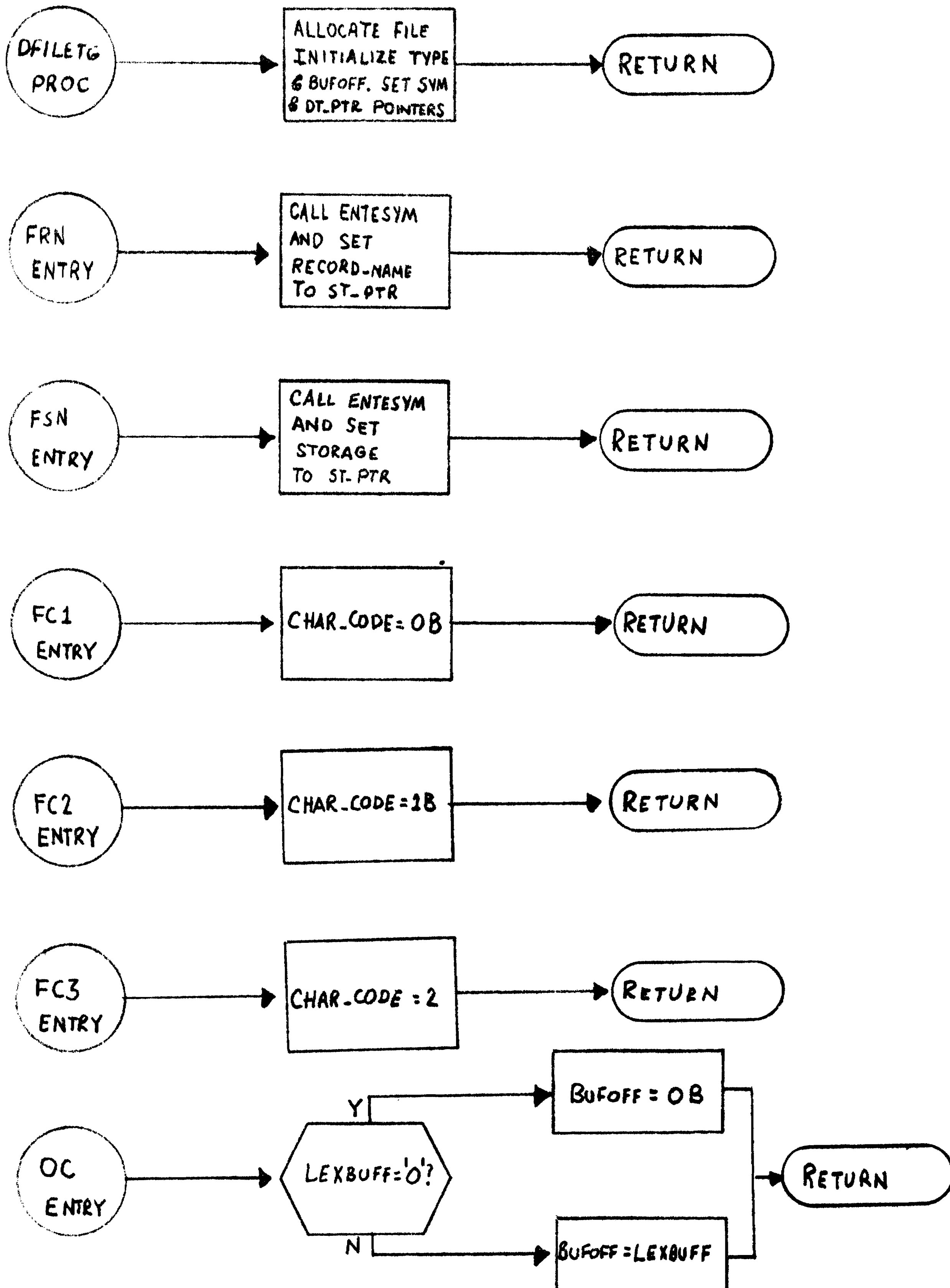
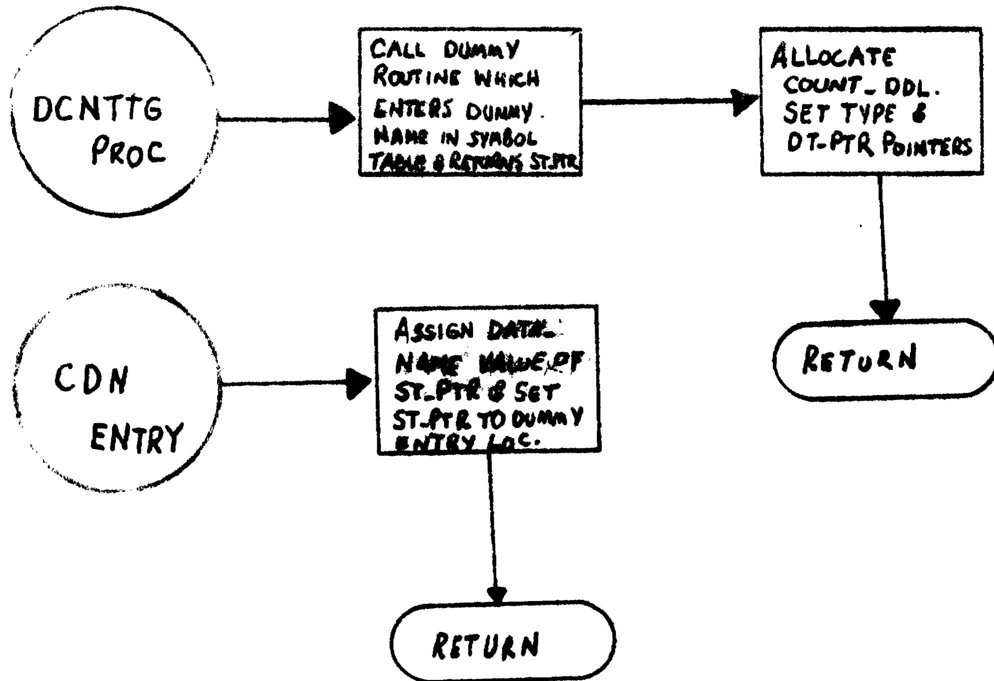


FIGURE B-1

COUNT STMT PROCEDURE



LENGTH STMT PROCEDURE

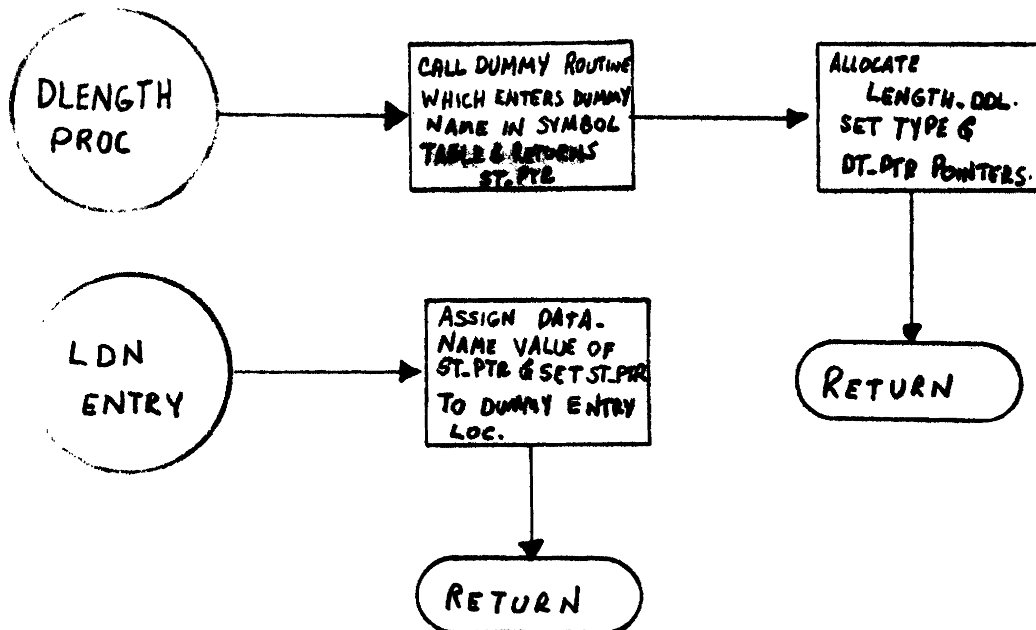
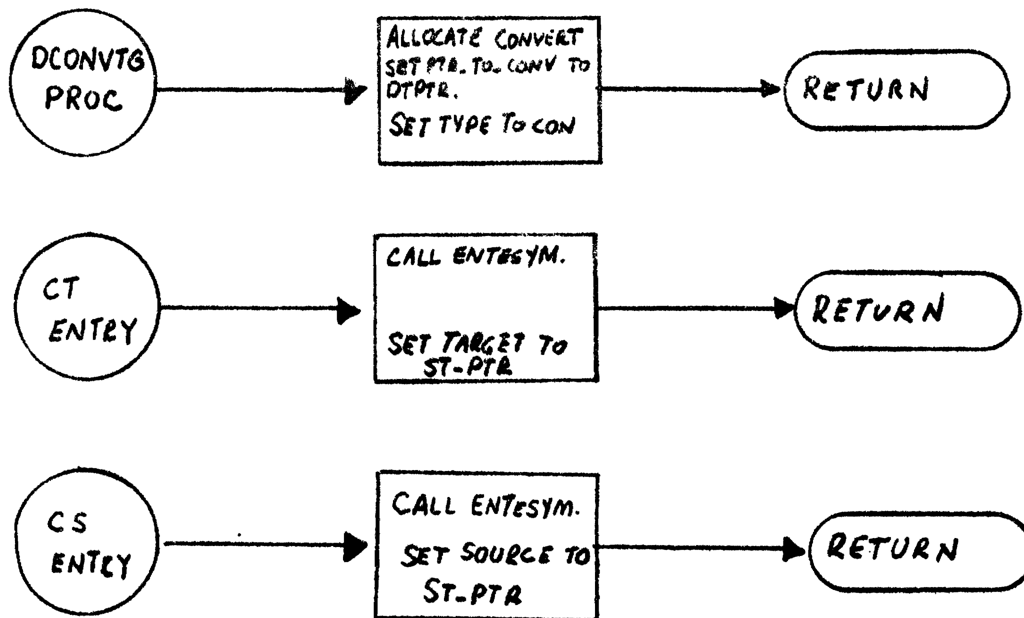


FIGURE B-2

CONVERT STMT PROCEDURE



CARD STMT PROCEDURE

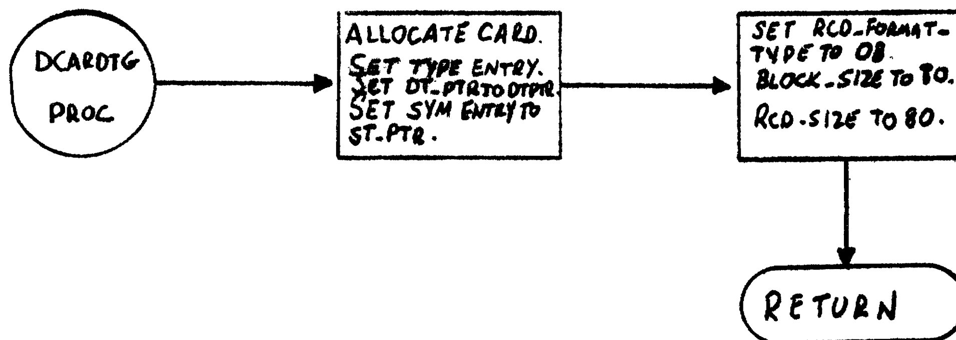


FIGURE B-3

RECORD FORMAT PROCEDURE

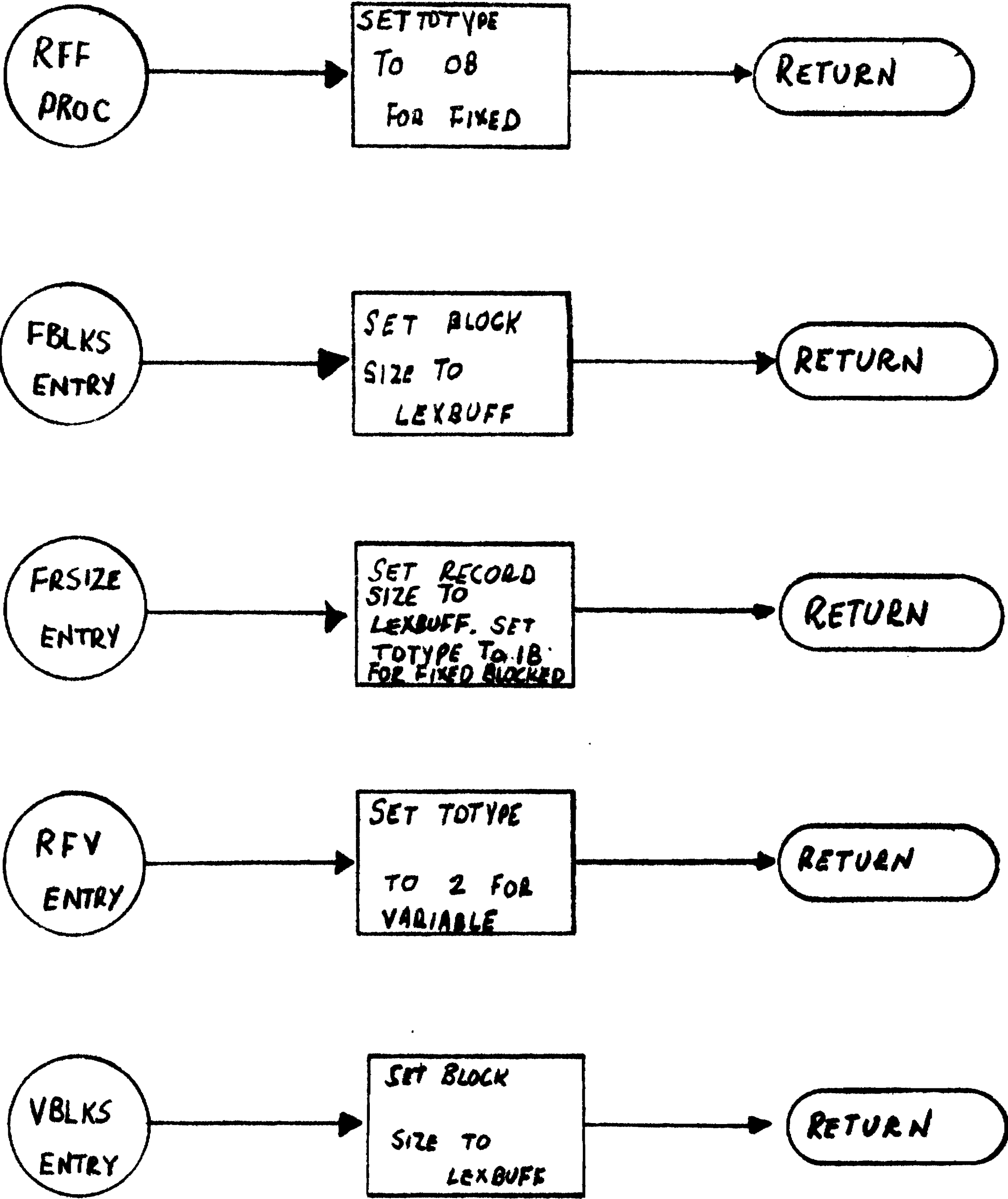


FIGURE B-4

RECORD FORMAT PROCEDURE (CONT.)

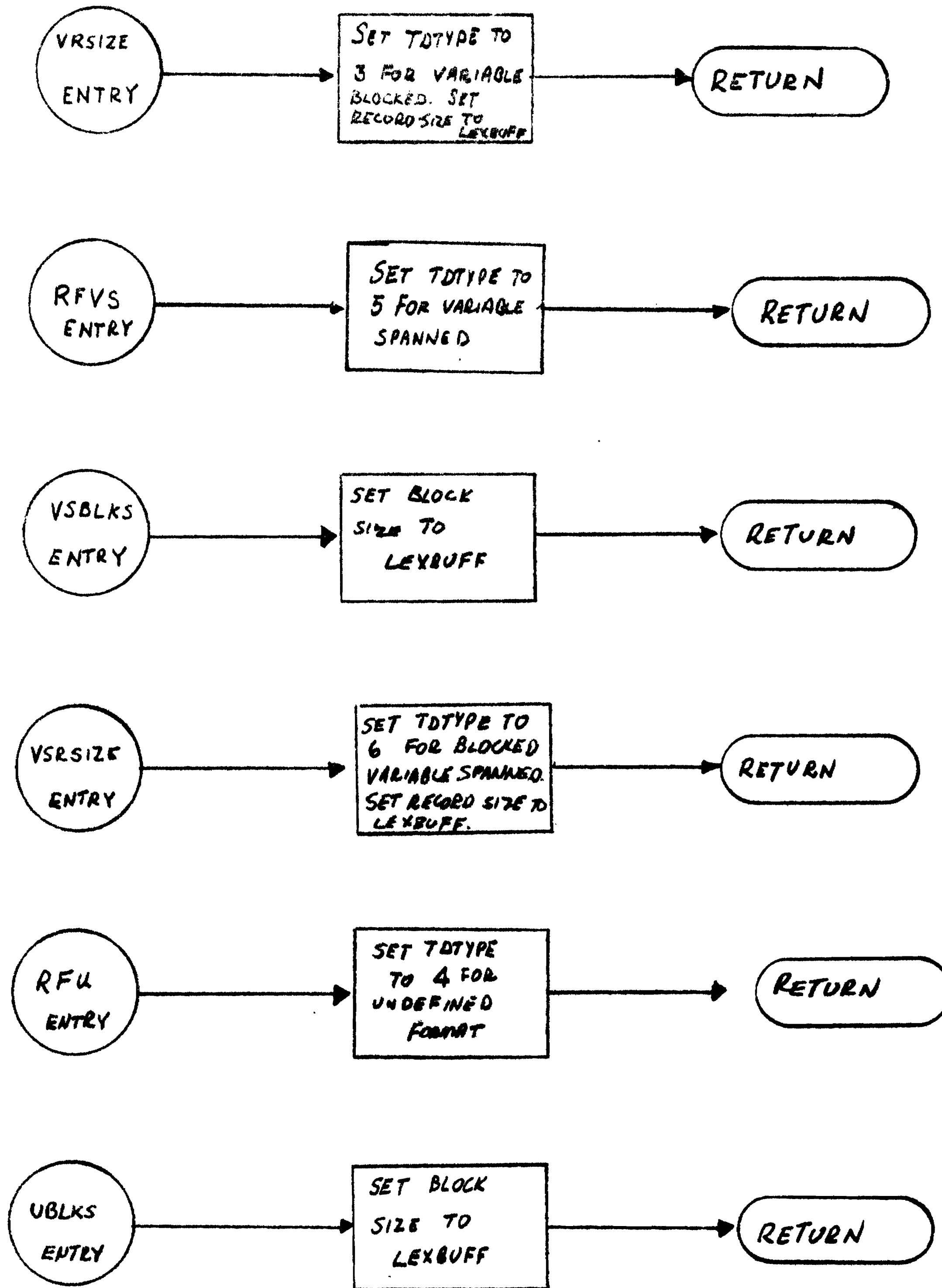


FIGURE B-4 (CONT.)

TAPE STMT PROCEDURE

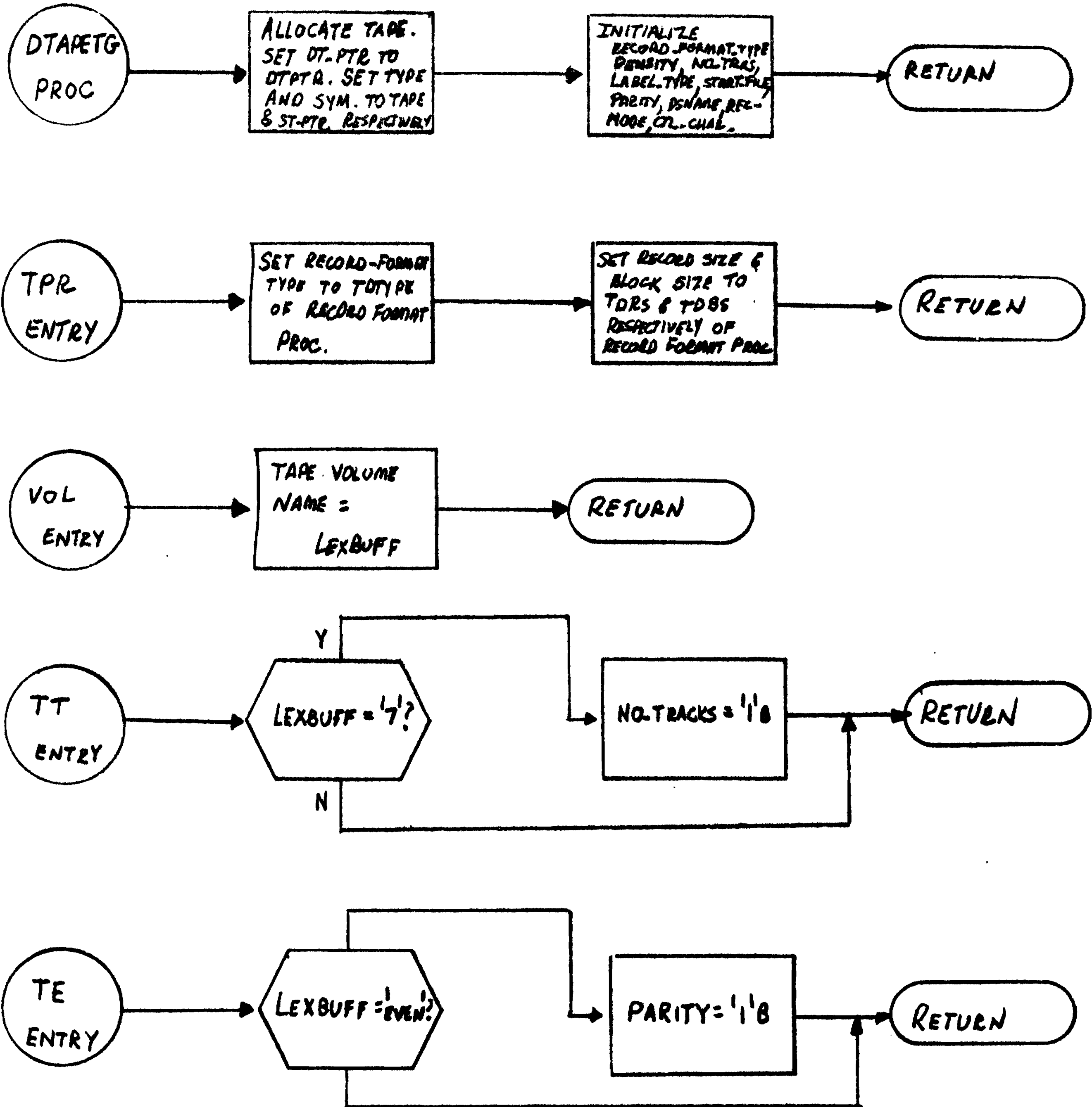
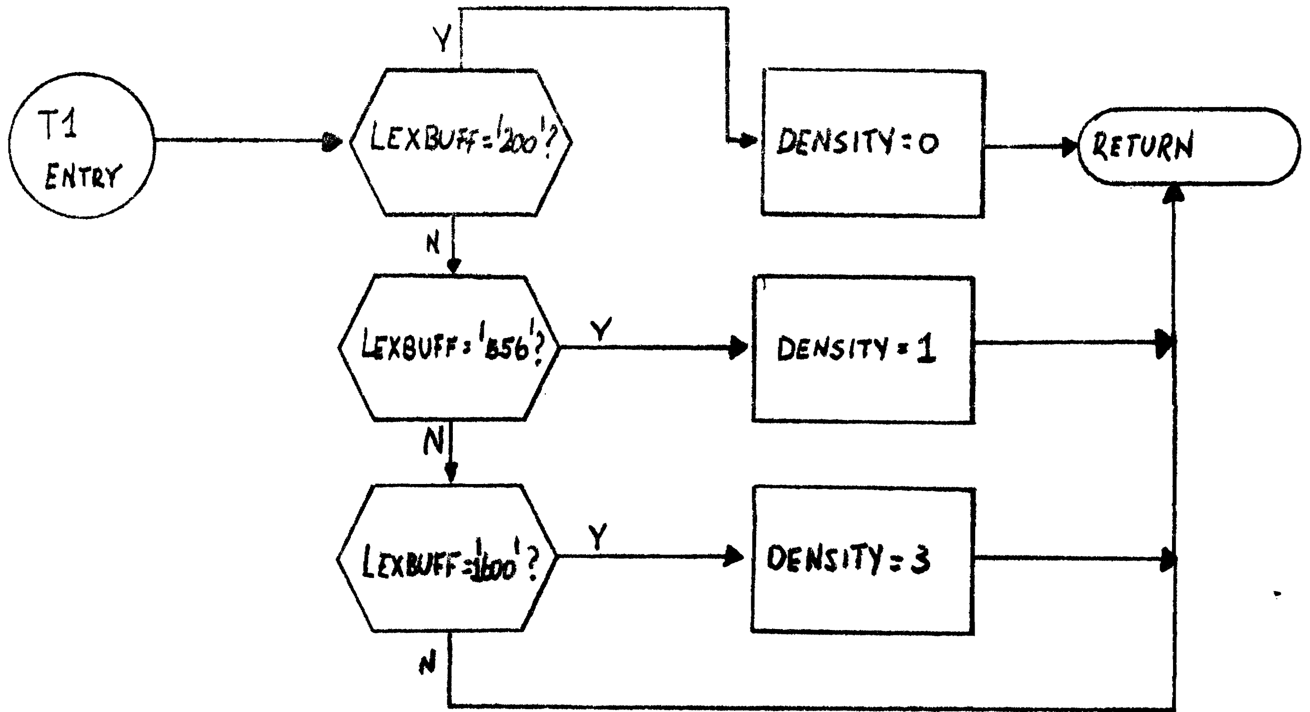


FIGURE B-5

TAPE STMT PROCEDURE (CONT.)



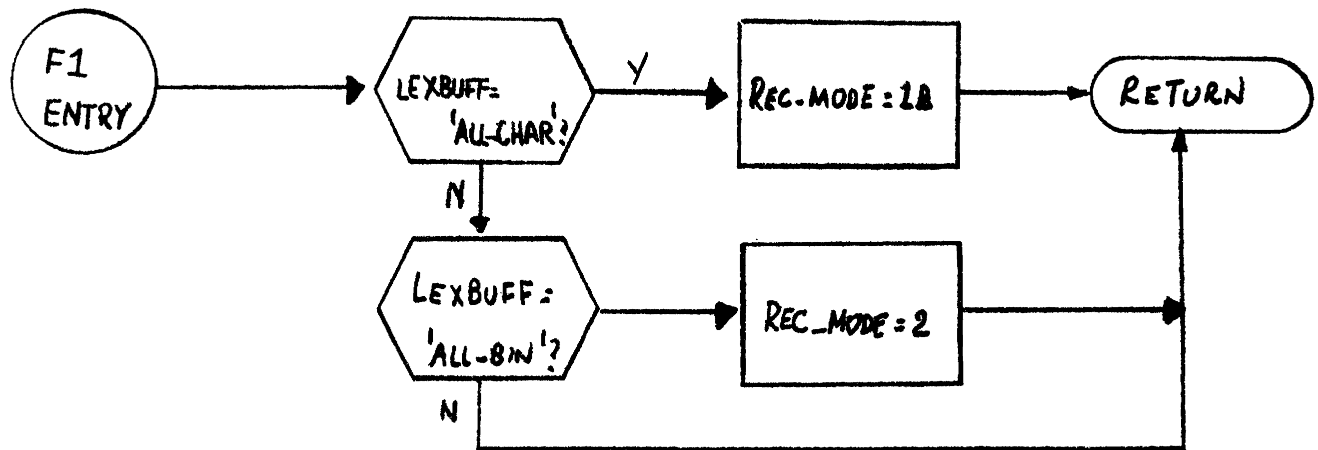


FIGURE B-5 (CONT.)

TAPE STMT PROCEDURE (CONT.)

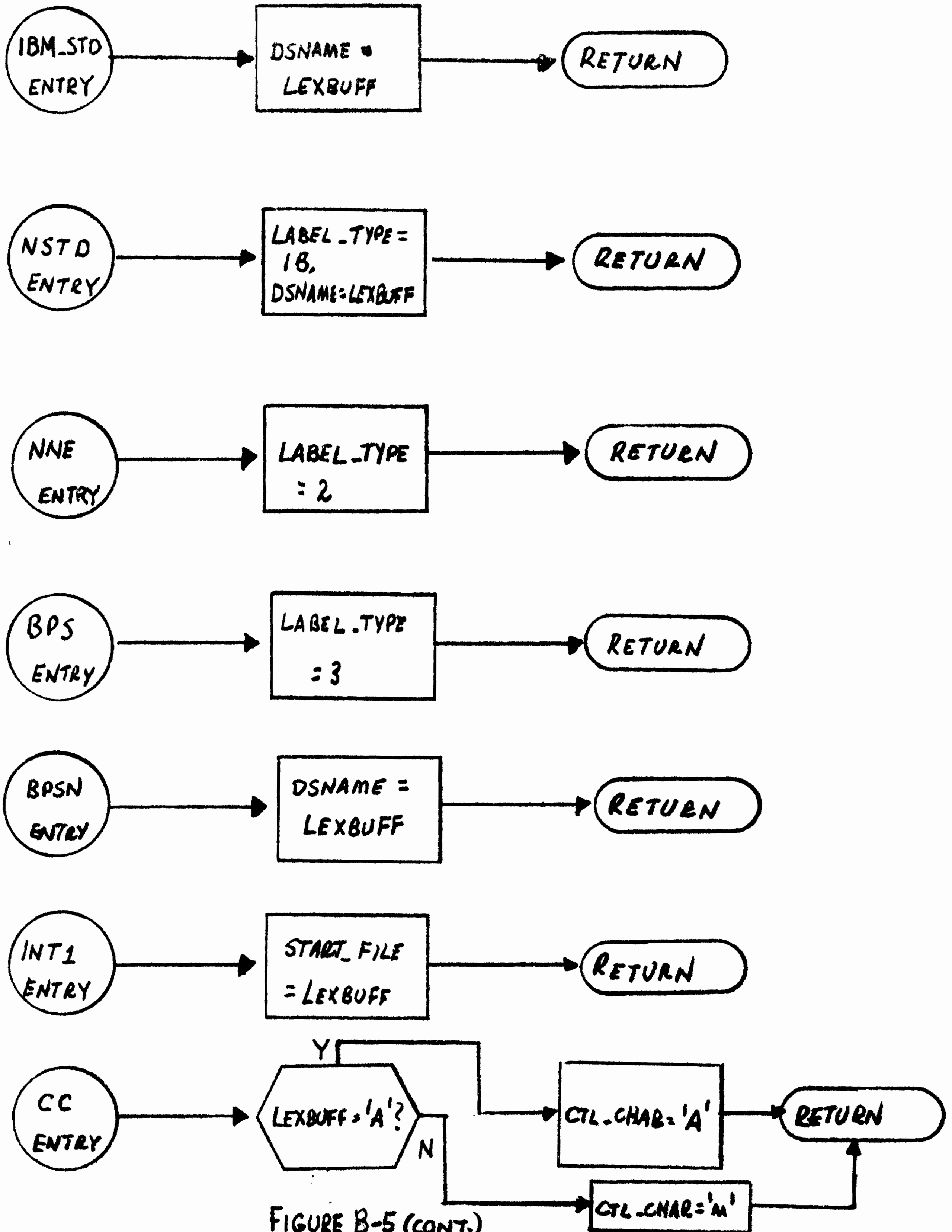


FIGURE B-5 (CONT.)

NAME LIST PROCEDURE

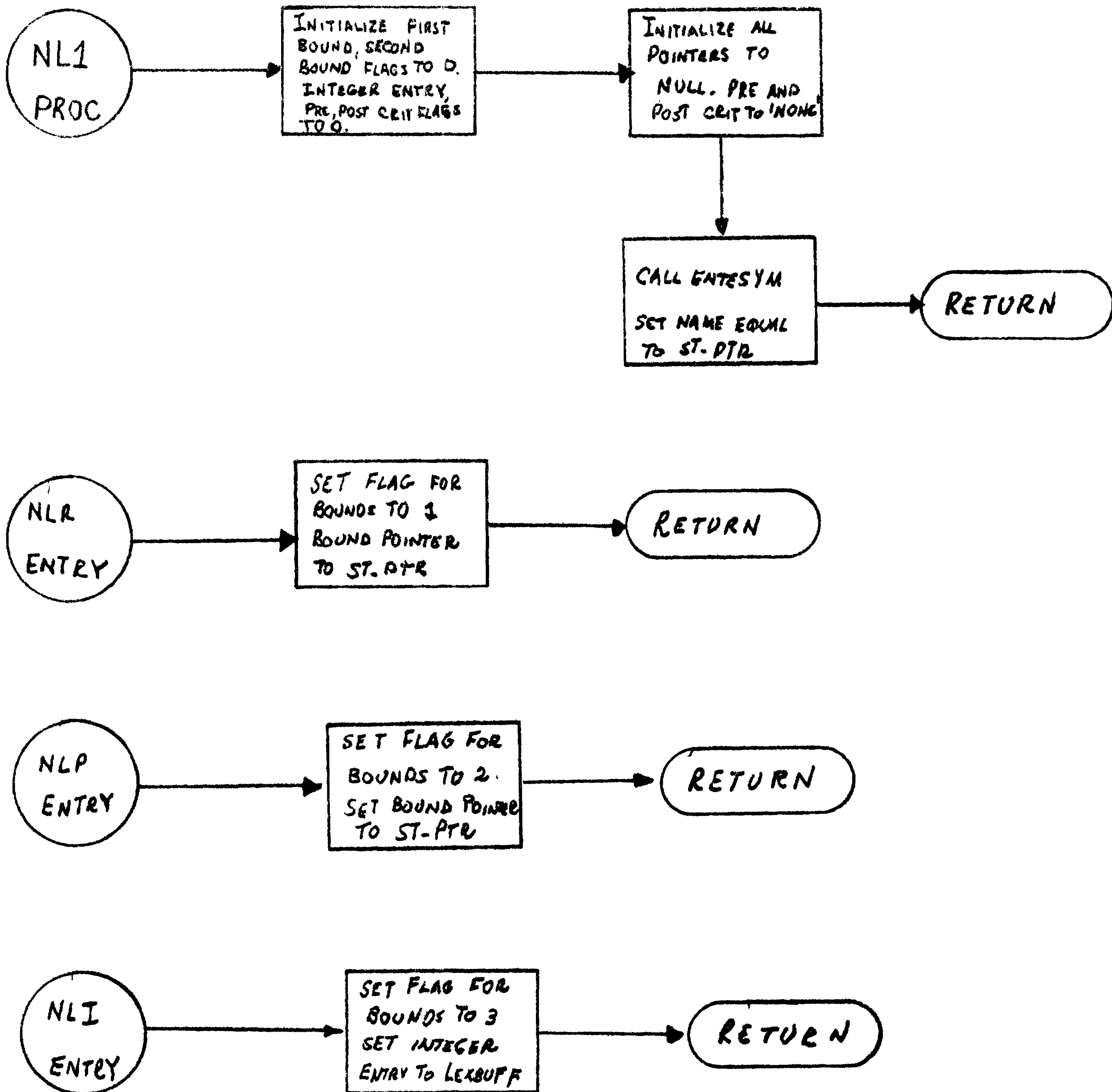


FIGURE B-6

NAME LIST PROCEDURE (CONT.)

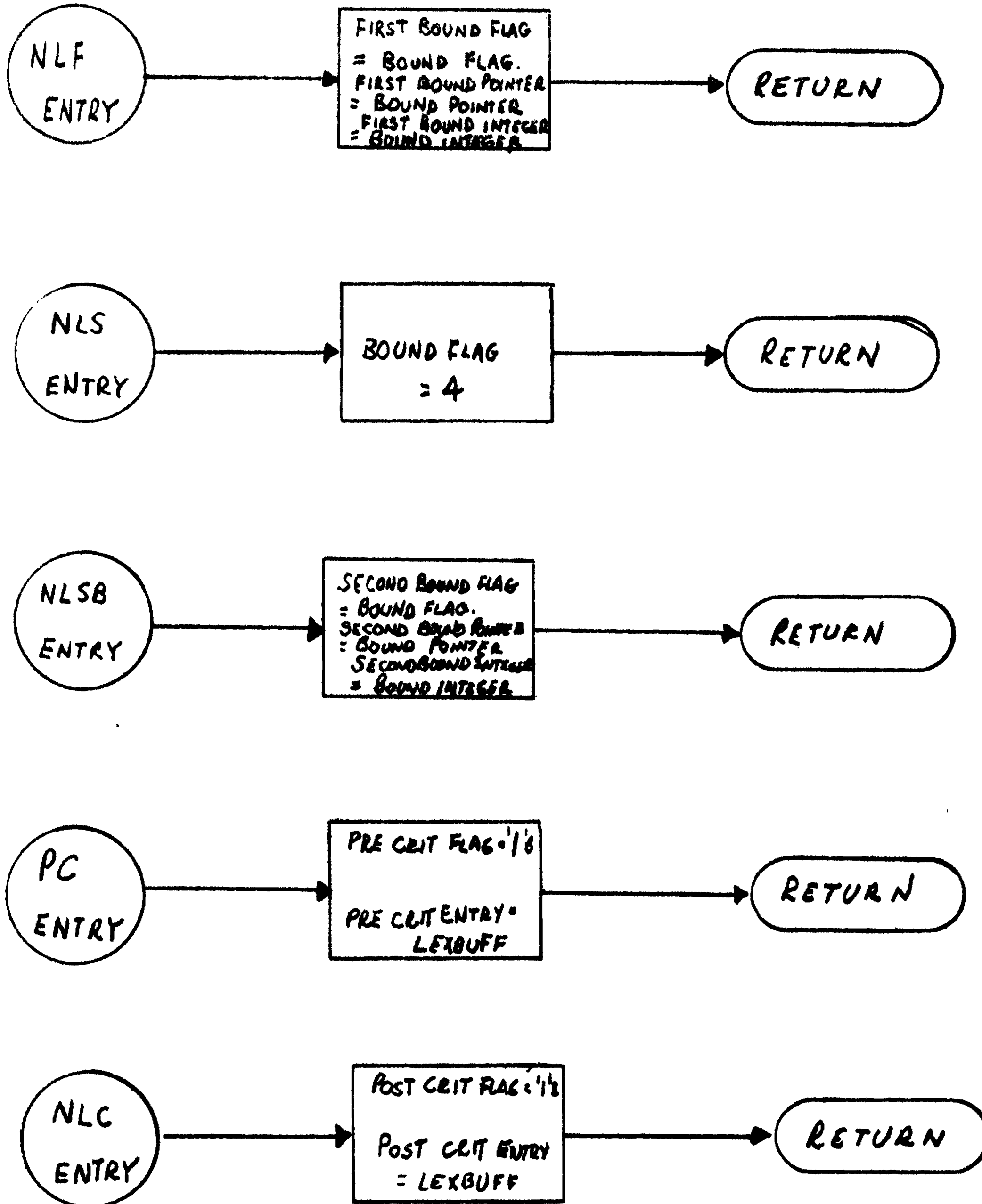


FIGURE B-6 (CONT.)

GROUP STMT PROCEDURE

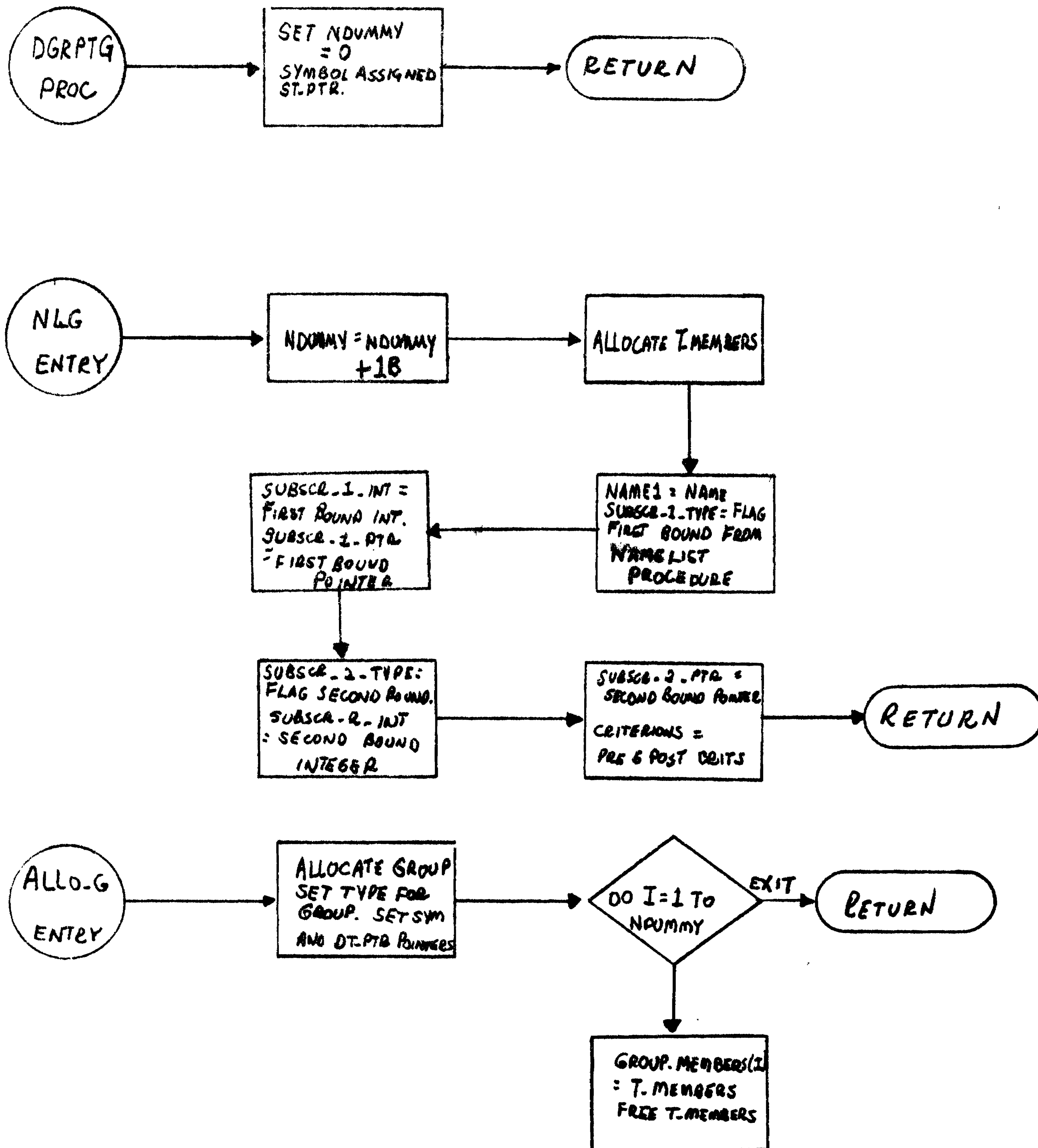


FIGURE B-7

RECORD STMT PROCEDURE

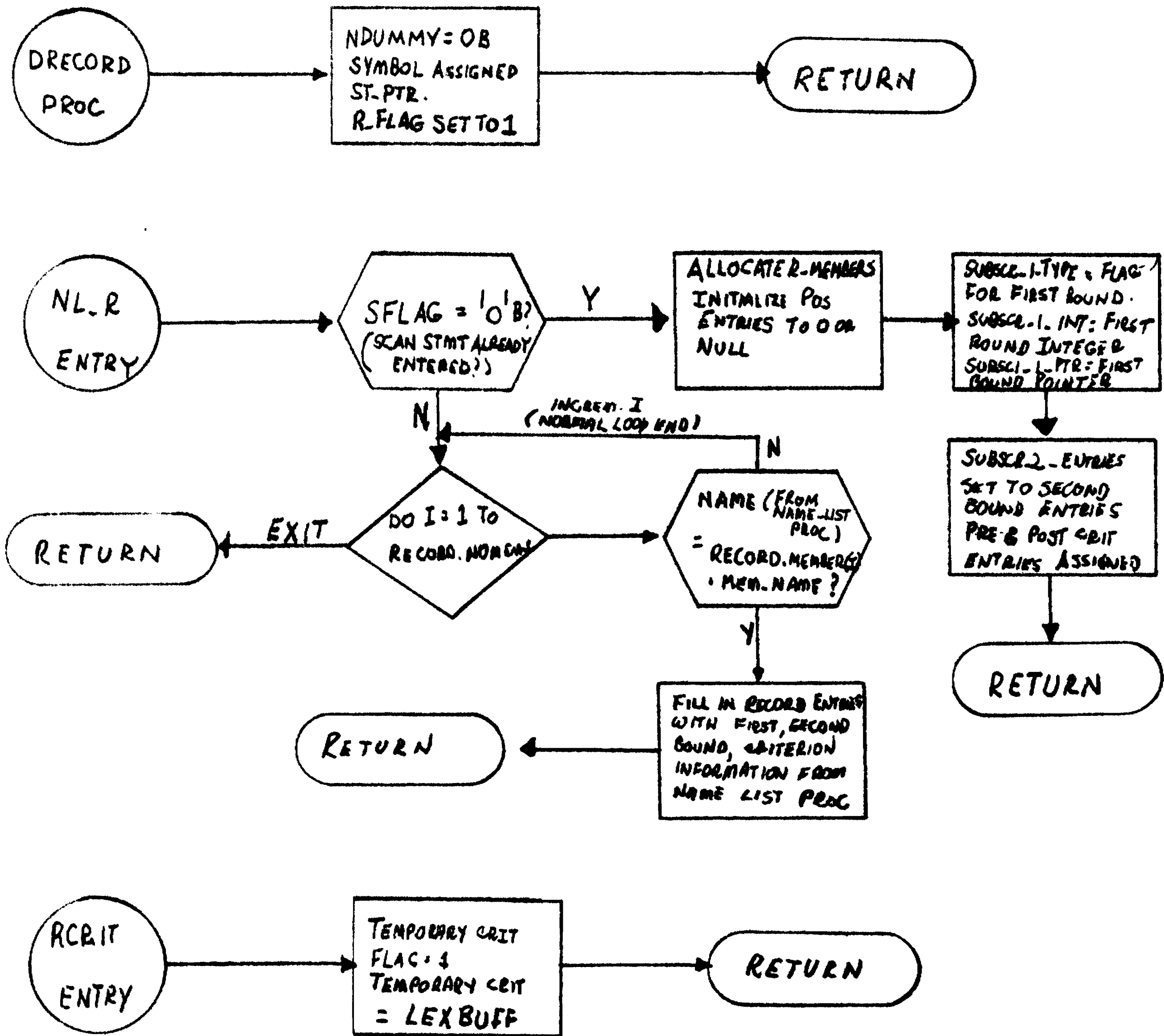


FIGURE B-8

RECORD STMT PROCEDURE (CONT.)

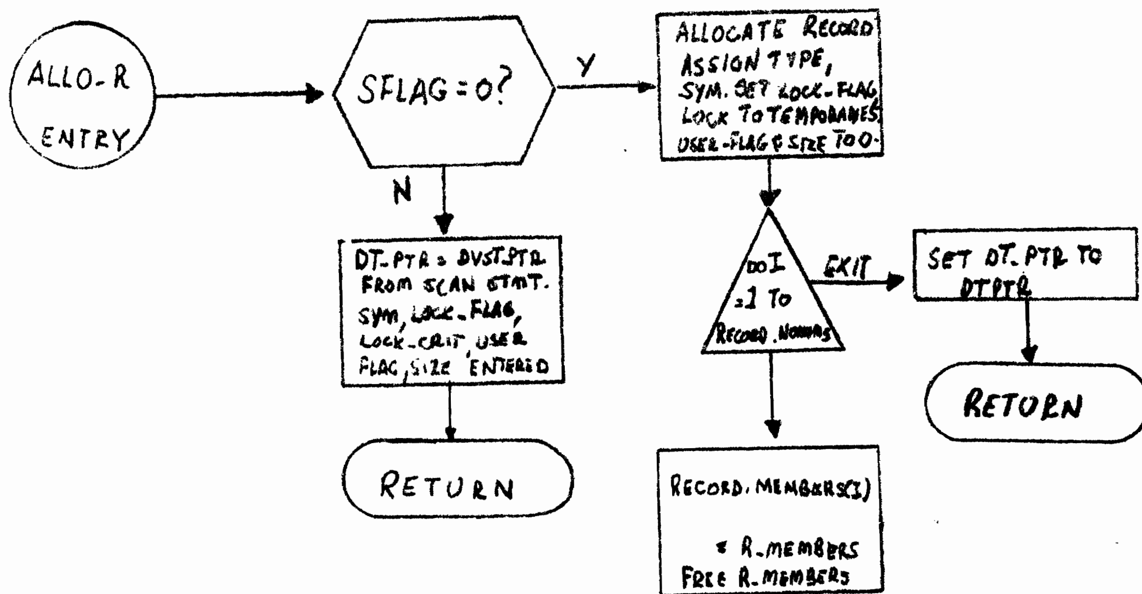


FIGURE B-8 (CONT.)

SCAN STATEMENT PROCEDURE.

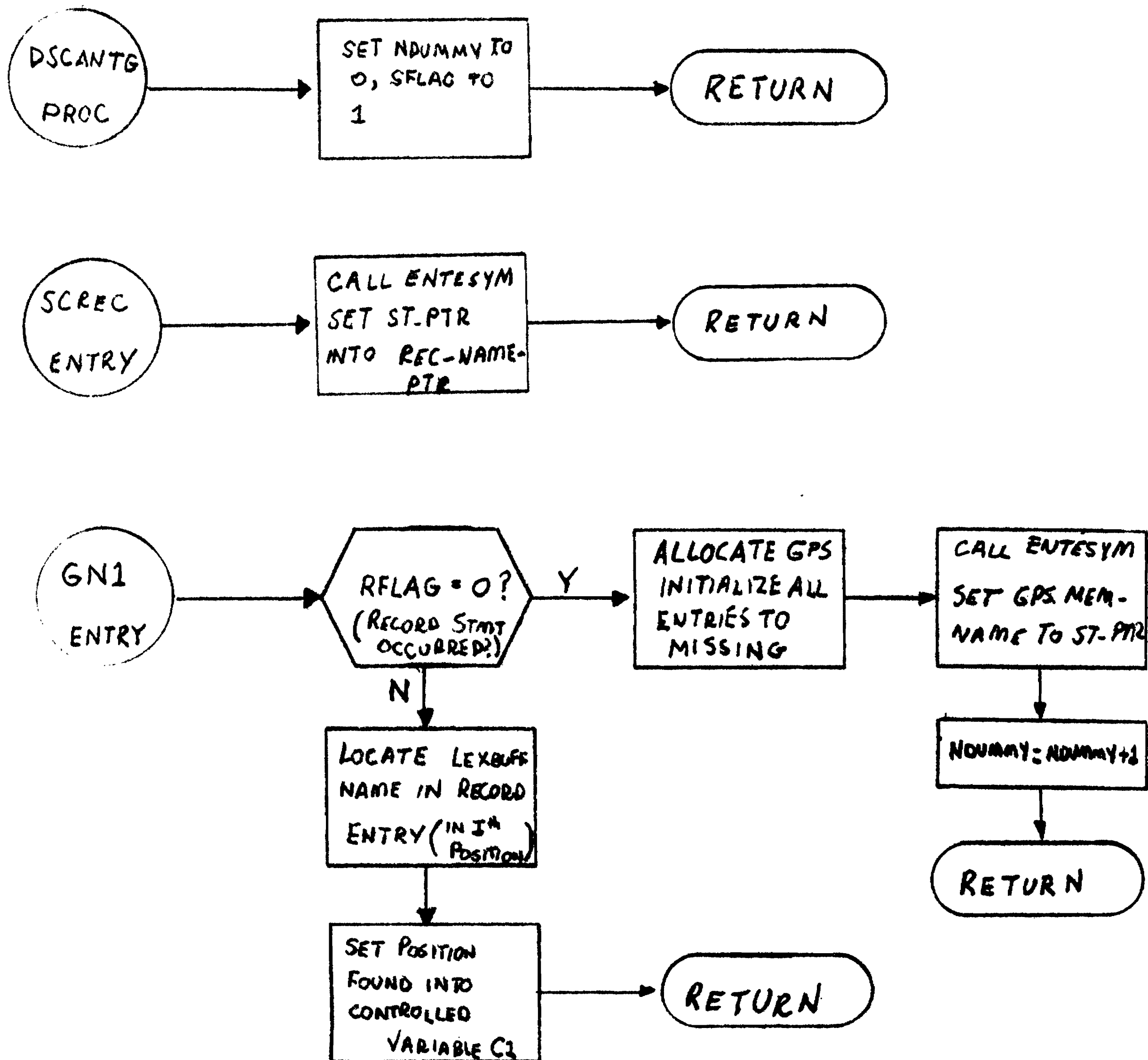
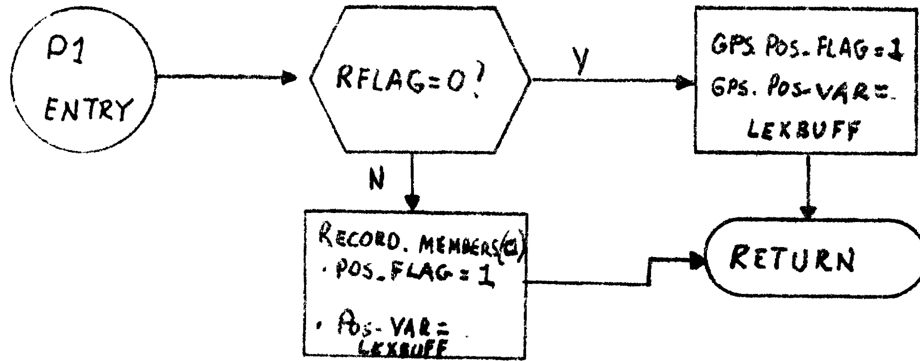
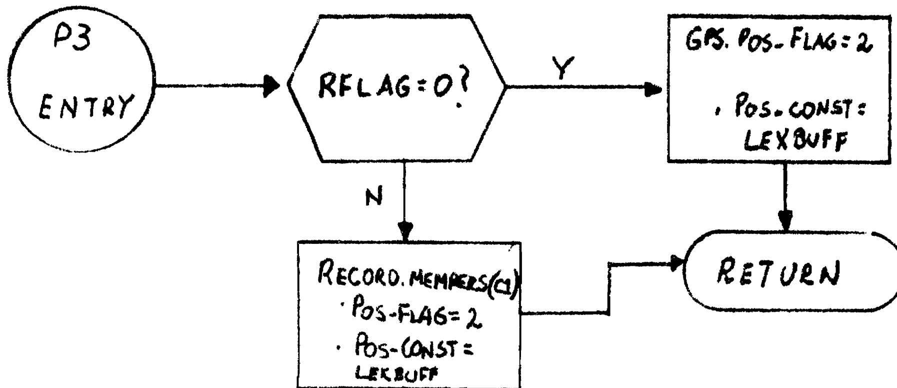


FIGURE B-9

SCAN STMT PROCEDURE (CONT.)





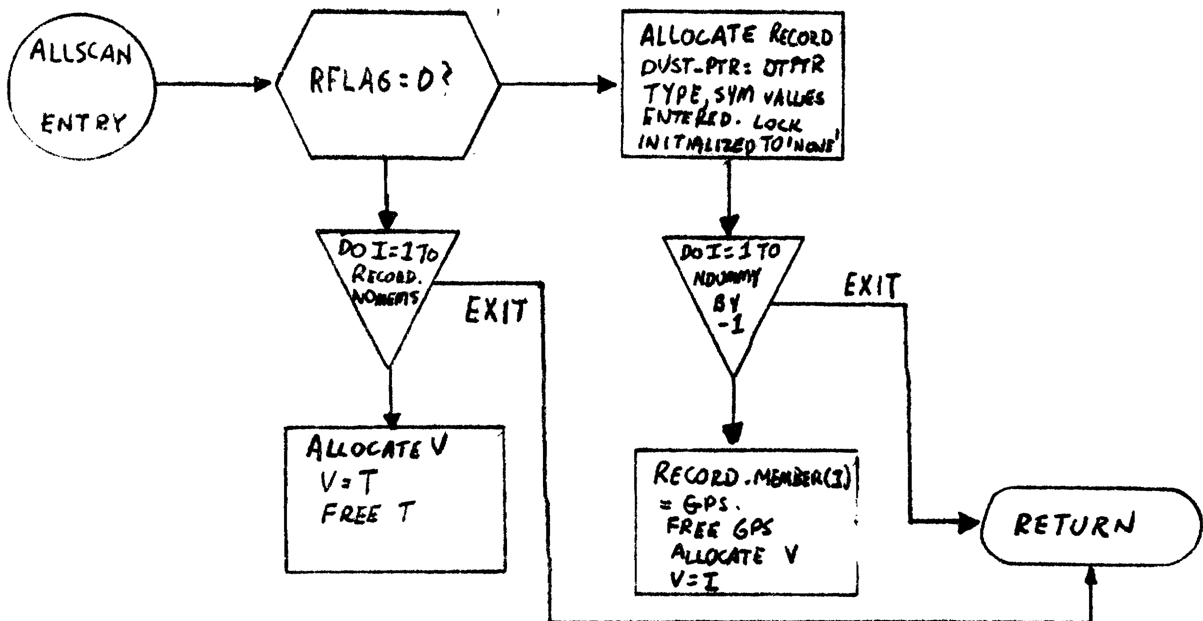


FIGURE B-9 (CONT.)

FIELD STMT PROCEDURE

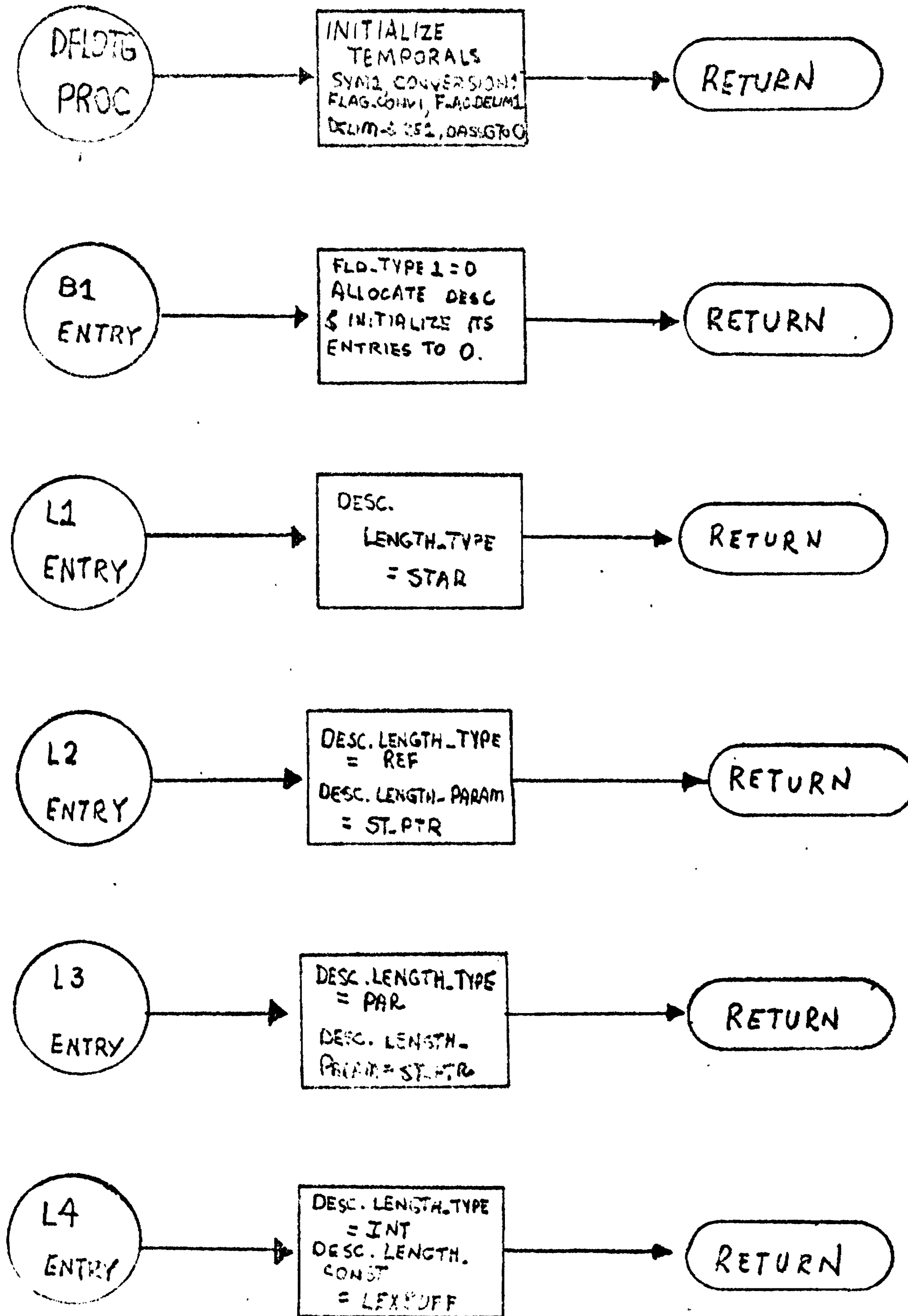


FIGURE B-10

FIELD STATEMENT PROCEDURE (CONT.)

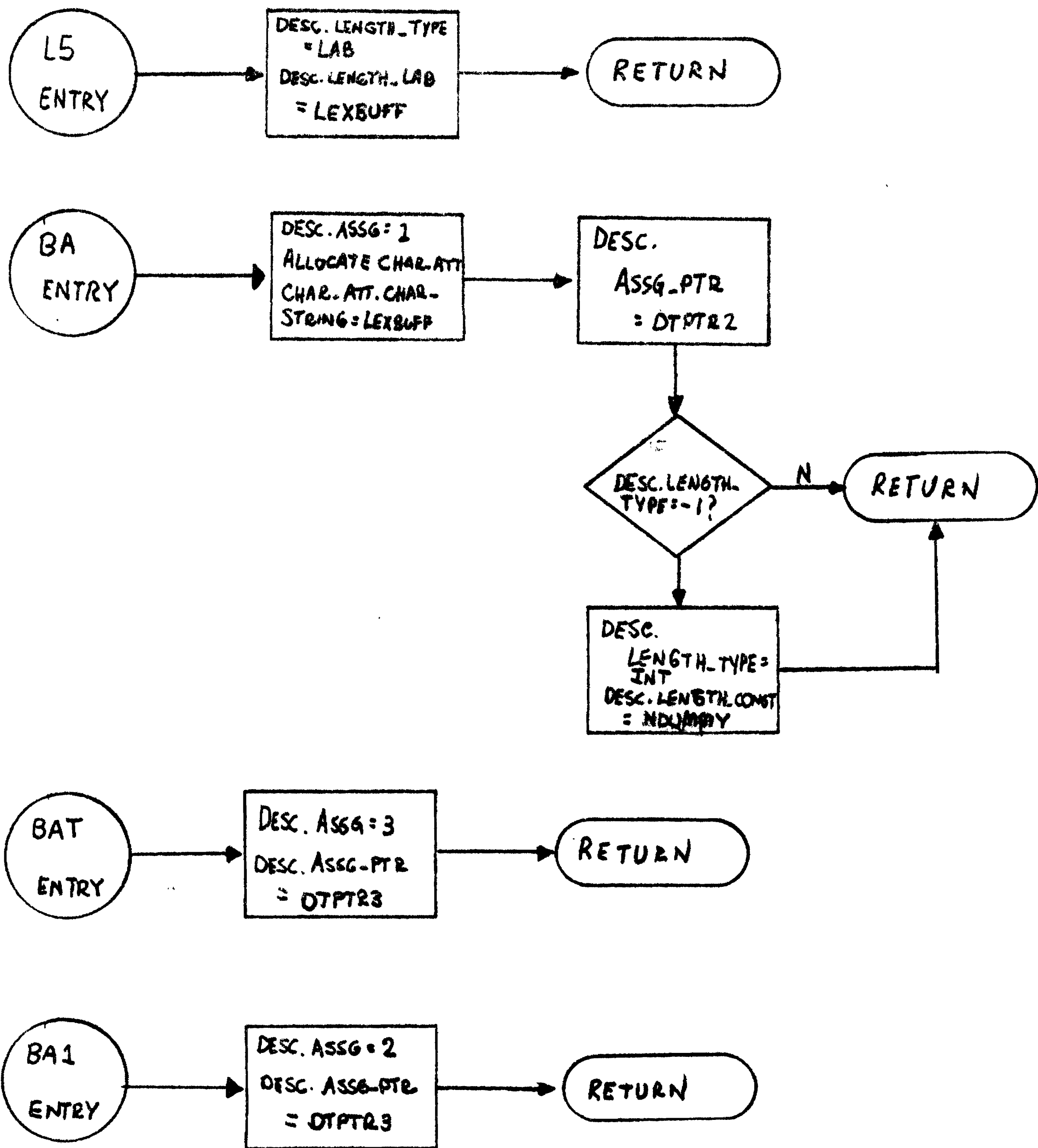


FIGURE B-10 (CONT.)

FIELD STATEMENT PROCEDURE (CONT.)

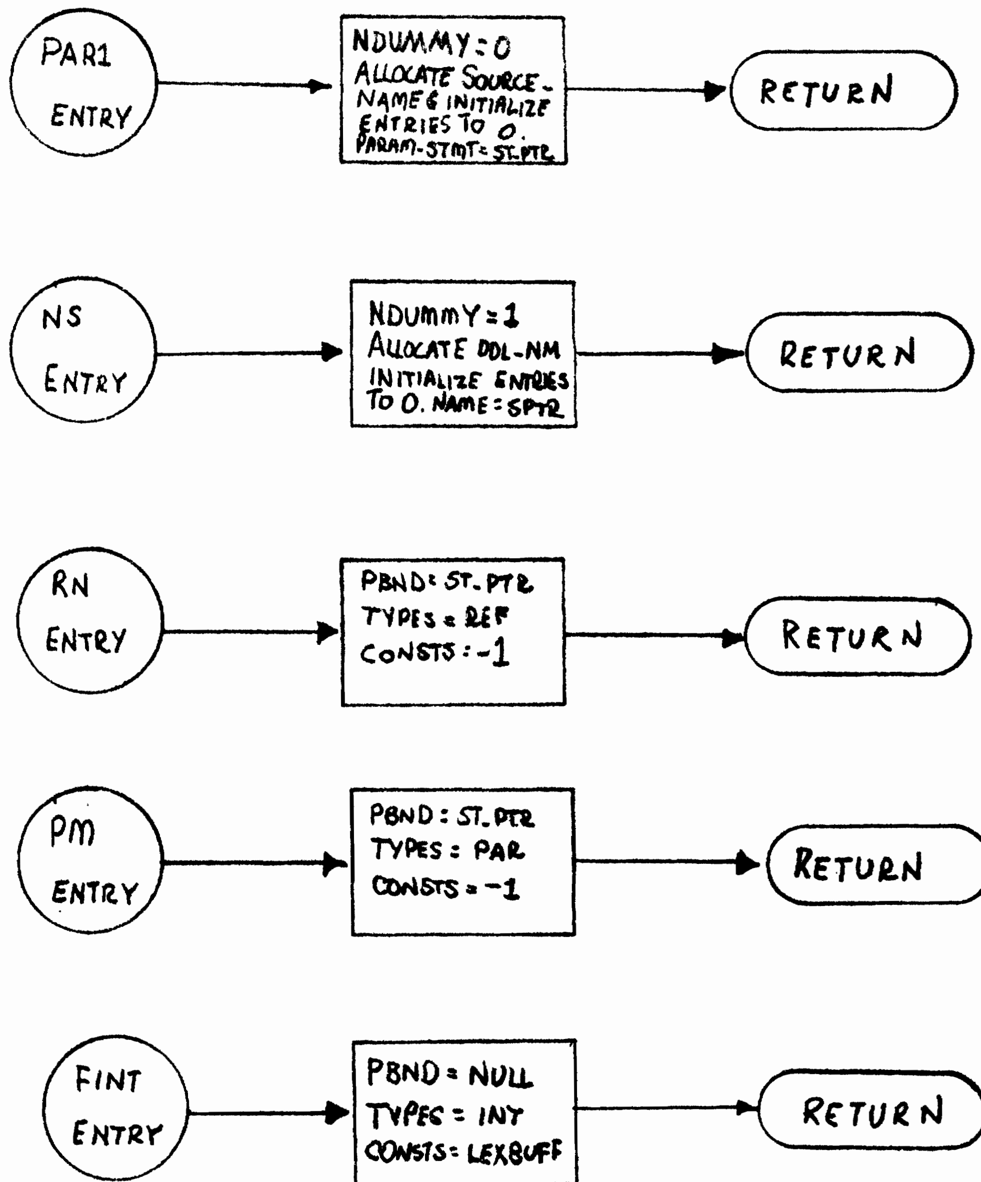


FIGURE B-10 (CONT.)

FIELD STMT PROCEDURE (CONT.)

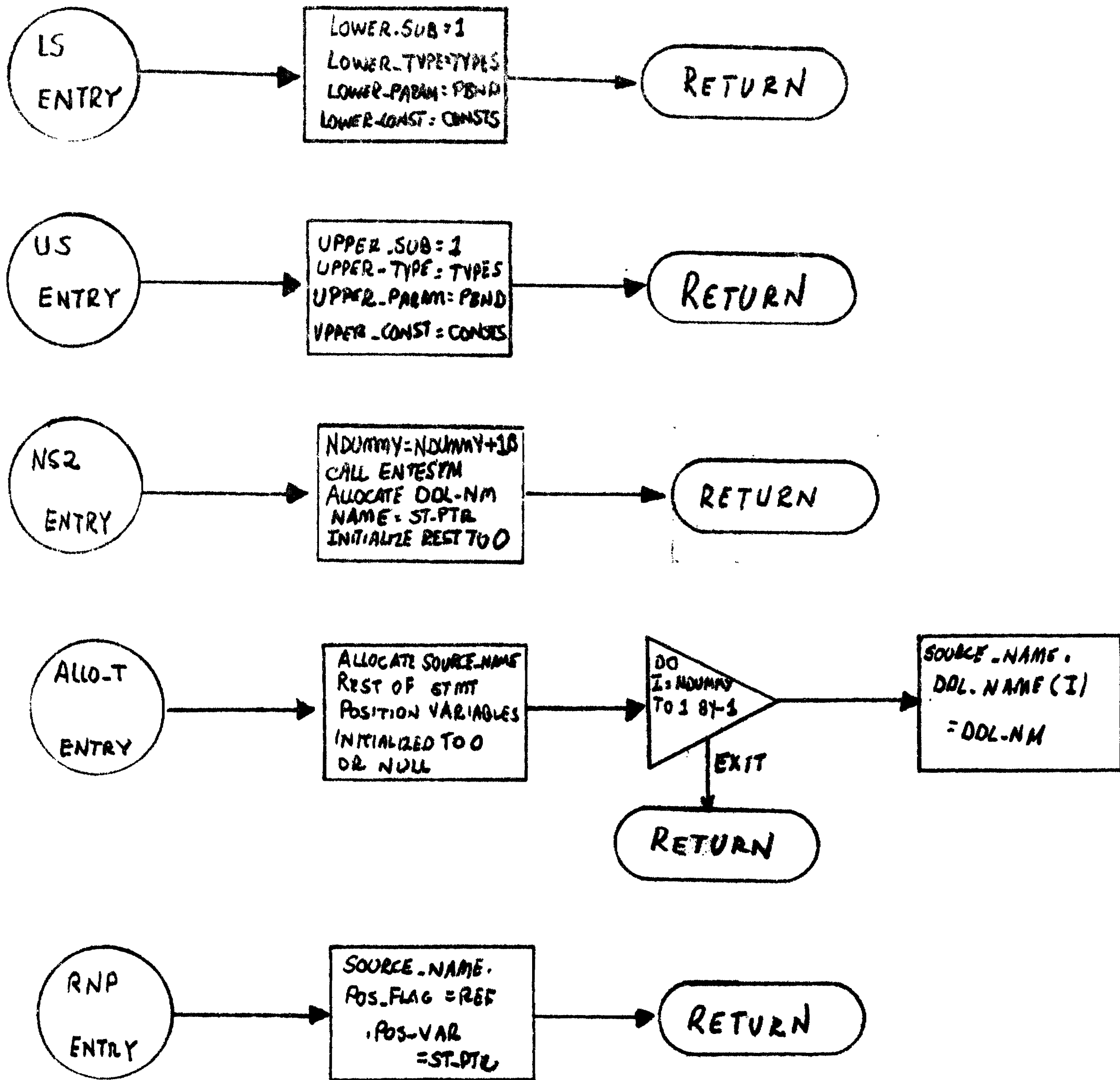


FIGURE B-10 (CONT.)

FIELD STMT PROCEDURE (CONT.)

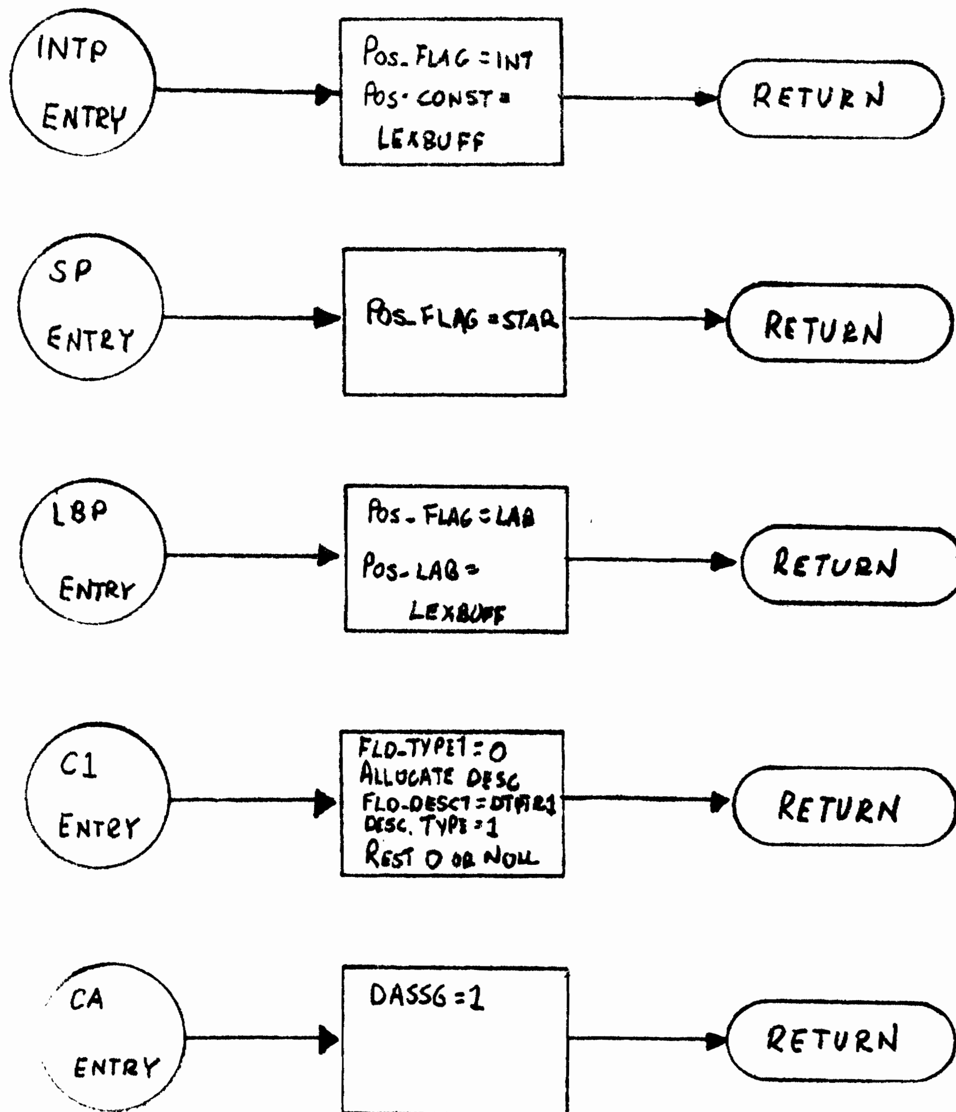


FIGURE B-10 (CONT.)

FIELD STMT PROCEDURE (CONT.)

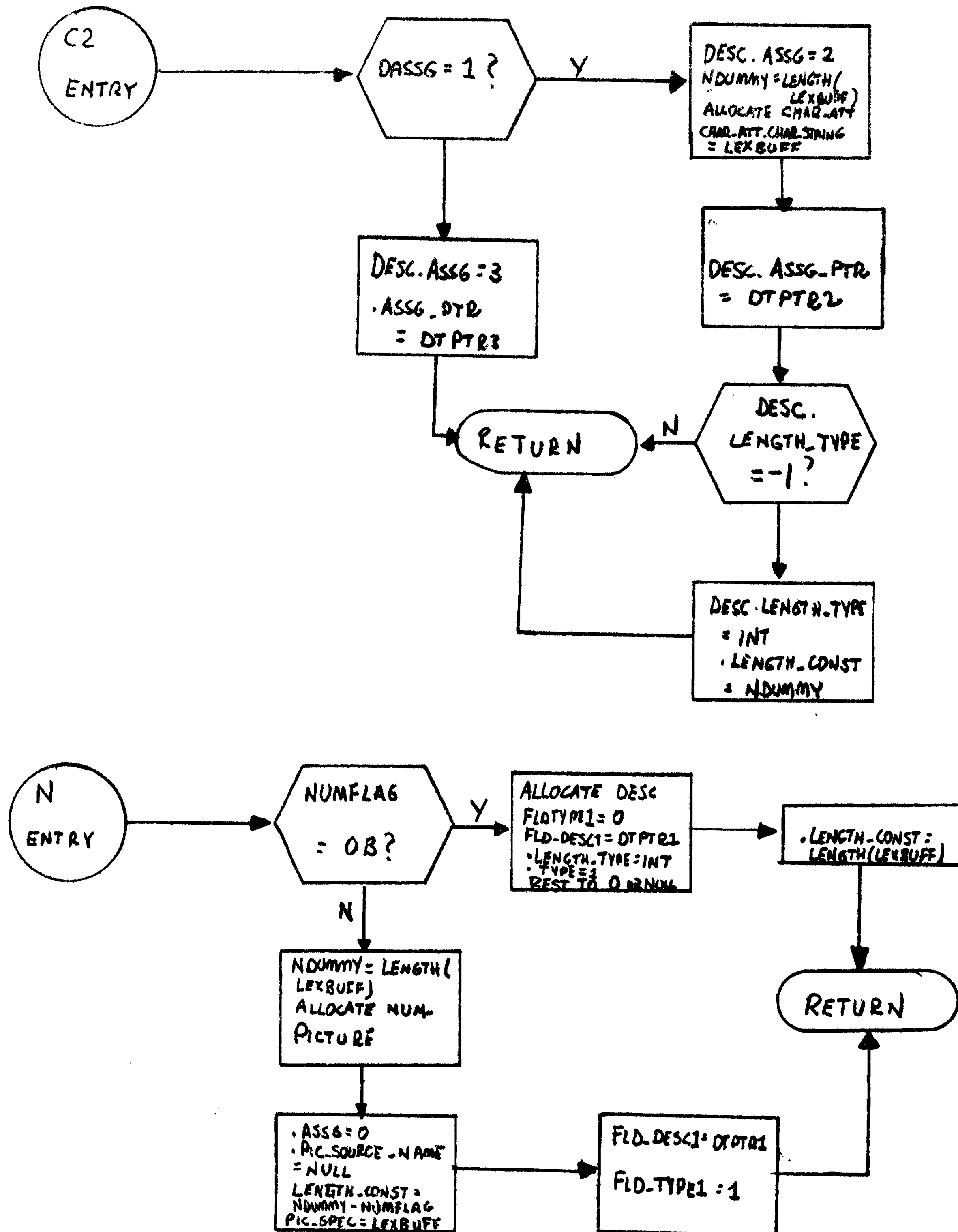


FIGURE B-10 (CONT.)

FILE STMT PROCEDURE (CONT.)

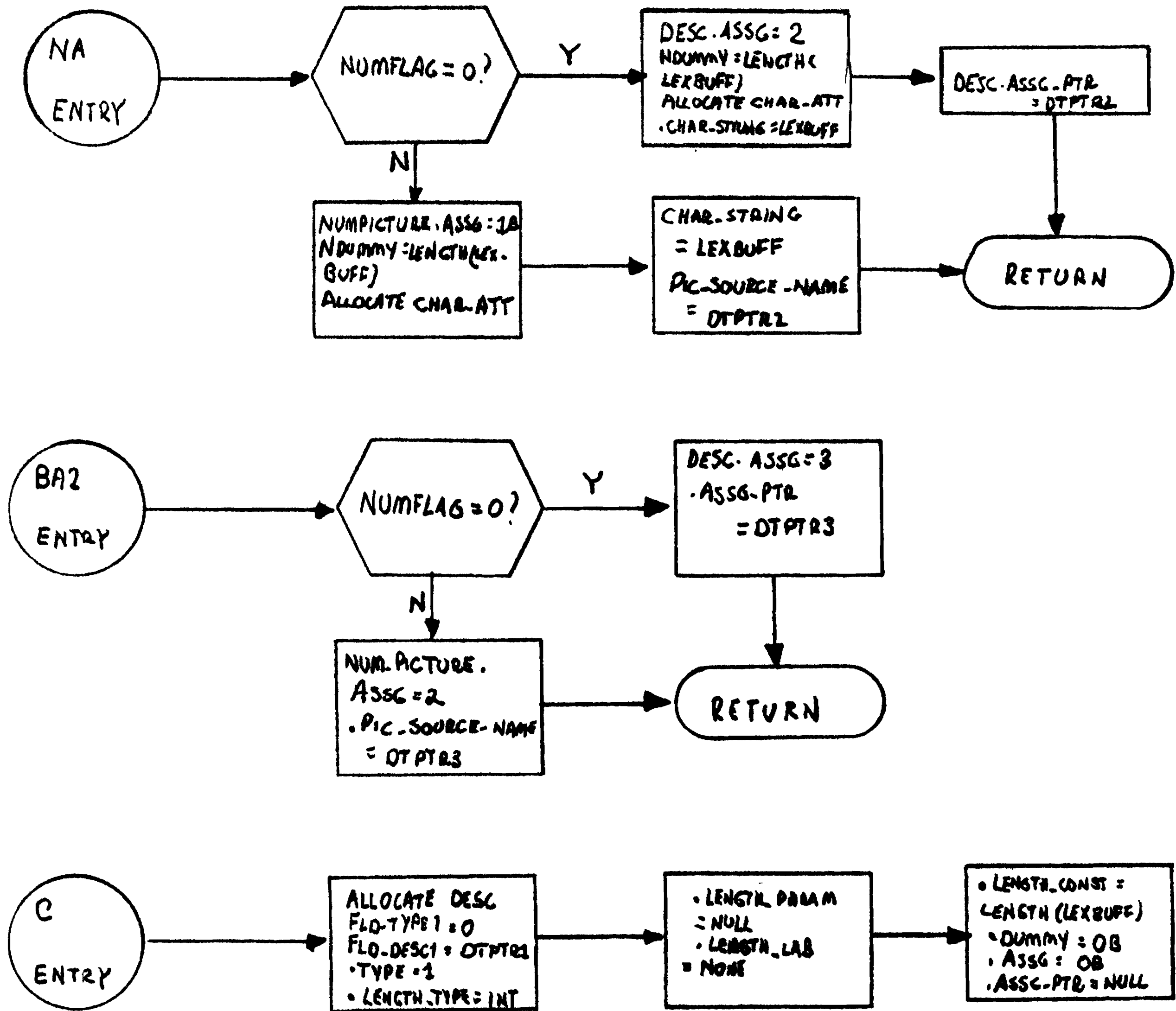


FIGURE B-10 (CONT.)

FIELD STMT PROCEDURE (CONT.)

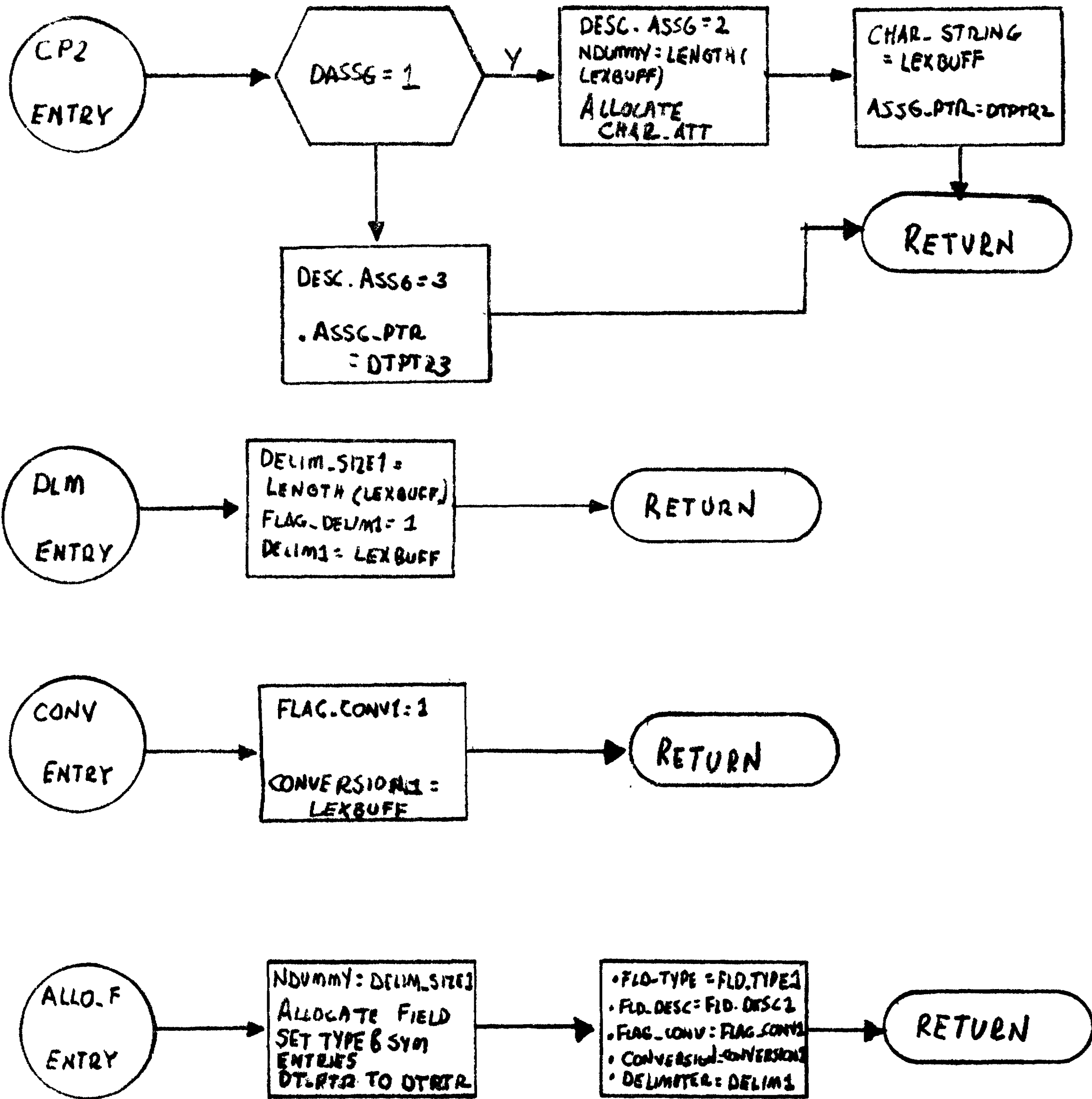


FIGURE B-10 (CONT.)

DISK STMT PROCEDURE

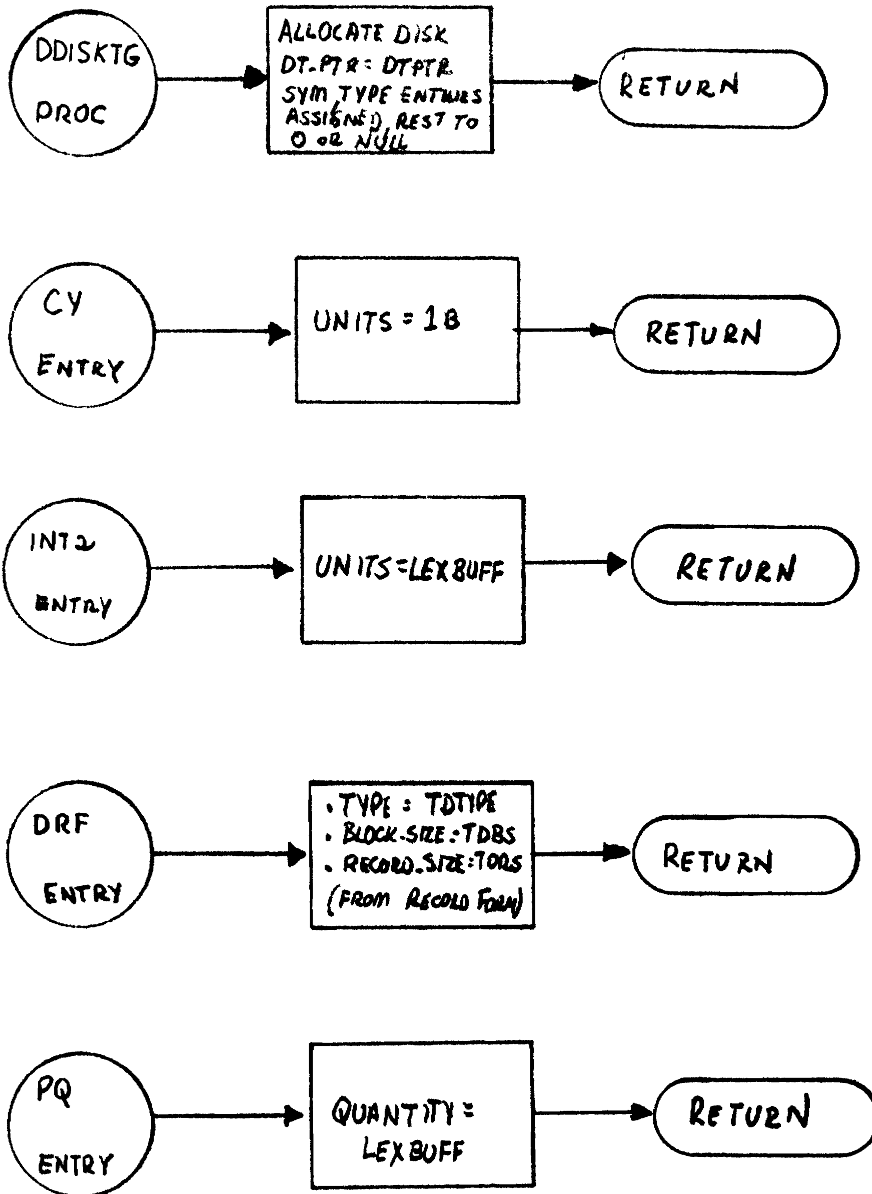


FIGURE B-11

DISK STMT PROEDURE (CONT.)

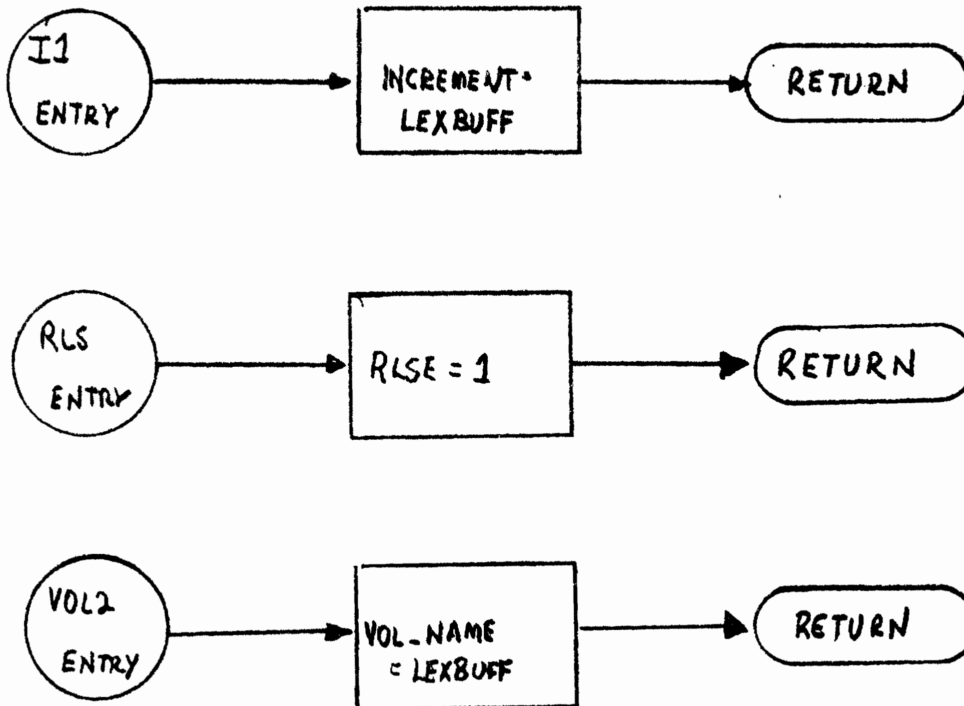


FIGURE B-11 (CONT.)

DISTRIBUTION LIST

Defense Documentation Center Cameron Station Alexandria, Virginia 22314	12 copies
Office of Naval Research Department of the Navy Information Systems Program Code 437 Arlington, Virginia 22217	2 copies
Office of Naval Research Branch Office/Boston 495 Summer Street Boston, Massachusetts 02210	1 copy
Office of Naval Research Branch Office/Chicago 536 South Clark Street Chicago, Illinois 60605	1 copy
Office of Naval Research Branch Office/Pasadena 1030 East Green Street Pasadena, California 91101	1 copy
Director, Naval Research Laboratory ATTN: Library, Code 2029 (ONRL) Washington, D. C. 20390	6 copies

U.S. Naval Research Laboratory
Technical Information Division
Washington, D. C. 20390 — 6 copies

Commandant of the Marine Corps (Code AX)
Dr. A. L. Slafkosky
Scientific Advisor
Washington, D. C. 20380 — 1 copy

Office of Naval Research
Code 455
Arlington, Virginia 22217 — 1 copy

Office of Naval Research
Code 458
Arlington, Virginia 22217 — 1 copy

Naval Electronics Laboratory Center
Computer Science Department
San Diego, California 92152 — 1 copy

Naval Ship Research & Development Ctr.
Dr. G. H. Gleissner
Computation & Mathematics Department
Bethesda, Maryland 20034 — 1 copy

Office of Naval Research
New York Area Office
Dr. J. Laderman
207 West 24th Street
New York, New York 10011 — 1 copy