University of Pennsylvania
THE MOORE SCHOOL OF ELECTRICAL ENGINEERING

Technical Report

AUTOMATIC GENERATION OF DATA CONVERSION-PROGRAMS USING A DATA DESCRIPTION LANGUAGE (DDL)

VOL. I SYSTEM DESCRIPTION
AND DOCUMENTATION

VOL. II APPENDICES -
USER GUIDE

by

Jesus Arturo Ramirez

Project Supervisor
Noah S. Prywes

May 1973

Prepared for the
Office of Naval Research
Information Systems
Arlington, Va. 22217

under

Contract N00014-67-A-0216-0014
Project No. NR 049-153

DISTRIBUTION STATEMENT

Moore School Report # 73-08

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)*

University of Pennsylvania
The Moore School of Electrical Engineering
Philadelphia, Pennsylvania 19104

2a. REPORT SECURITY CLASSIFICATION

UNCLASSIFIED

2b. GROUP

3. REPORT TITLE

AUTOMATIC GENERATION OF DATA CONVERSION PROGRAMS USING A DATA DESCRIPTION LANGUAGE (DDL) - Vol. 1

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

Technical Report

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

Jesus A. Ramirez

6. REPORT DATE

May 1973

7a. TOTAL NO OF PAGES

Vol. 1, 292

7b. NO. OF REFS

Vol. 1, 34

8a. CONTRACT OR GRANT NO.

N00014-67-A-0216-0014

b. PROJECT NO

NR 049-153

c.

d.

9a. ORIGINATOR'S REPORT NUMBER(S)

Moore School Report No. 73-08

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

Office of Naval Research
Information Systems
Arlington, Virginia 22217

13. ABSTRACT

The report describes a DDL/DML Processor and a methodology to automatically generate data conversion programs. The Processor, accepts as input descriptions of source and target files in a Data Description Language (DDL) and a Data Manipulation Language (DML). It produces an output conversion program in PL/1 capable of converting the source file and producing the target file.

Security Classification

KEY WORDS	LINK A		LINK B		LINK C	
	ROLE	WT	ROLE	WT	ROLE	WT
Compilers, Generators, Problem Oriented Languages, Syntax Analysis, Lexical Analysis, Data Description Languages, Data Manipulation Language, Conversion Programs, Automatic Programming, I/O Utility						

AUTOMATIC GENERATION OF DATA CONVERSION-PROGRAMS USING A DATA DESCRIPTION LANGUAGE (DDL)

Jesus Arturo Ramirez

A DISSERTATION

in

Computer and Information Sciences

Presented to the Faculty of the Graduate School of Arts and Sciences of the University of Pennsylvania in Partial Fulfillment of the Requirements for the Degree of Doctor of Philosophy.

May 1973

Noah Prywes

Supervisor of Dissertation

A.K. Joshi

Graduate Group Chairman

ABSTRACT

AUTOMATIC GENERATION OF DATA CONVERSION-PROGRAMS USING A DATA DESCRIPTION LANGUAGE (DDL)

by Jesus A. Ramirez

Supervisor: Noah S. Prywes

The ultimate goal of the research described in this dissertation is generally directed towards the automatic generation of programs to perform data processing, based on specifications of the desired functions and systems. The dissertation describes a Processor which has been developed to generate such programs in the context of file conversion. The Processor accepts as an input descriptions of Source and Target Files in a Data Description Language (DDL). It produces as an output a conversion program capable of converting the Source File and producing the Target File. To specify validation criteria, data security, summaries and reports, the user utilizes a Data Manipulation Language (DML). This report describes the Data Description and Data Manipulation Languages (DDL/DML), the operation of the Processor, and includes a user guide and full documentation of the processing system. The research was conducted in the Moore School of Electrical Engineering, University of Pennsylvania and utilized the UNICOLL IBM 370/165 computer system.

The contributions of the research are in several directions, primarily in providing a useable Processor that has been

adequately documented and tested for its reliability and dependability. Novel methods of implementation have been used to implement the Processor itself. In particular, methods have been employed to automatically generate major portions of the Processor using compiler-compiler methodology. In conjunction with this latter activity a new meta-language to describe the syntax and encoding of the DDL was developed.

The DDL that has been developed is to a large extent similar to data definition facilities of COBOL or PL/1. The DDL is in that respect English-like; however, it possesses much more extensive facilities to describe data structure and organization than those provided in COBOL or PL/1.

The Processor automatically produces conversion programs in PL/1, and has itself been programmed in PL/1.

ACKNOWLEDGEMENTS

I would like to express my gratitude to my Supervisor, Professor Noah S. Prywes, who first introduced me to this area of research and for his guidance and support throughout its development. I am grateful to the Information System Branch of the Office of Naval Research for supporting this research under contract N00014-67-A-0216-0007 Project No. 049-272.

I would like to thank the many members of the DDL Project, past and present, with whom I have worked over the past three years. In particular, I would like to mention Messrs. Harold Solow and N. Adam Rin with whom I have had many interesting discussions and who helped me in the editing of this dissertation. Also Mr. N. Adam Rin worked with me in the area of Code Generation for the DDL Processor. Thanks are due to Messrs. Andy French and Peter Gross for their assistance in the development of the Syntactic Analysis Program Generator and Internal Table Creation Routines respectively, and to Mr. Tom Reimer for his help in the writing of the User Guide for DDL & DML.

The author would like to express his appreciation of the excellent secretarial help from Miss Barbara Weber.

I would like to acknowledge the support and encouragement of my parents.

Finally, and most importantly, I wish to thank my wife, Glorilu, for she has constantly supported me by her love, her confidence, and her devotion.

INDEX

INDEX (continued)

INDEX (continued)

INDEX (continued)

INDEX (continued)

INDEX (continued)

INDEX (continued)

INDEX (continued)

INDEX (continued)

TABLE OF CONTENTS

VOL. I - SYSTEM DESCRIPTION AND DOCUMENTATION

Page

Page

Page

Page

LIST OF FIGURES

VOL. I

LIST OF FIGURES (continued)

LIST OF FIGURES (continued)

BIBLIOGRAPHY

[AHO 72] Aho, A.V., and Ullman, J.D.: The Theory of Parsing, Translation and Compiling, Vol 1. Prentice Hall, 1972.

[BAC 59] Backus, J.W.: The Syntax and Semantics of the Proposed International Language of the Zurich ACM - Gamm Conference, Proceedings of the International Conference on Information Processing, UNESCO, 1959.

[BRO 63] Brooker and Morris.: The Compiler-Compiler. Annual Review in Automata Programming., Vol. 3, 1963.

[CAR 69] Carr, J.W., and Weiland, J.: A Non-Recursive Method of Syntax Specification, Comm. ACM., Vol. 9, No. 4, 1969.

[CHE 62] Cheatham, T.E., and Warshall, S.: Translation of Retrieval Requests Couched In A "Semiformal" English - Like Language. Comm. ACM., Vol. 5, 1962.

[CODASYL 71] "Codasyl System Committee Technical Report." Feature Analysis of Generalized Data Base Management Systems, May, 1971.

[CON 63] Conway, M.E.: Design of A Separable Transition - Diagram Compiler. Comm. ACM., Vol. 6, No. 7, 1963.

[COR 69] Corbato, F.J.: PL/1 As A Tool For System Programming, DATAMATION., Vol. 15, No. 5, 1969.

[FAN 72] Fang, Isu.: Folds, A Declarative Formal Language Definition System, Ph.D. Dissertation, Stanford University, 1972.

[FEL 68] Feldman, J., and Gries, D.: Translator Writing Systems, Comm. ACM., Vol. 11, 1968.

[FEL 66] Feldman, J.: A Formal Semantics For Computer Language and Its Application In A Compiler-Compiler, Comm. ACM., Vol. 9, 1966.

[FLO 64] Floyd, R.W.: The Syntax Of A Programming Language - A Survey, IEEE Trans., 1964.

[FRE 71] French, A., Ramirez, J.A., Solow, H., and Prywes, N.S.: Design Of The Data Description Language Processor, Annual Report., The Moore School of Electrical Engineering, University of Pennsylvania, 1971.

[FRE 72] French, A.H.: A Syntactic Analysis Program Generator. M.Sc. Thesis, The Moore School of Electrical Engineering, University of Pennsylvania, 1972.

[ING 66] Ingerman, P.: A Syntactic - Oriented Translator, Academic Press, New York, 1966.

[IRO 61] Irons, E.T.: A Syntax Directed Compiler For ALGOL 60, Comm. ACM., Vol. 4, 1961.

[ITU 66] Iturniaga, R., and Standish, T.A.: Techniques and Advantages of Using The Formal Compiler Writing System (FSL) To Implement A Formula ALGOL Compiler. Proc. AFIPS, SJCC., Vol. 28, 1966.

[JOH 68] Johnson, W.L., Porter, J.H., Ackley, S.I., and Ross, D.T.: Automatic Generation Of Efficient Lexical Processors Using Finite State Techniques. Comm. ACM., Vol. 11, 1968.

[KNU 68] Knuth, D.E.: The Art of Computer Programming, Vol. 1/ Fundamental Algorithms. Addison-Wesley Publishing Company, 1968.

[KNU 73] Knuth, D.E.: The Art Of Computer Programming, Vol. 3/ Sorting and Searching. Addison-Wesley Publishing Company, 1973.

[LAN 70] Lang, C.A.,: "Languages For Writing Systems Programs" in Software Engineering Techniques, ed. Buxton and Randell, 1970.

[LEA 64] Leavenworth, B.M.: Fortran IV As A Syntax Language. Comm. ACM., Vol. 7, 1964.

[LED 62] Ledly, and Wilson.: Automatic - Programming Language Translation Through Syntactical Analysis. Comm. ACM., Vol. 5, No. 3, 1962.

[McC 65] McClure, R.M.: TMG - A Syntax - Directed Compiler. Proc. ACM 20th Natl. Conf., 1965.

[McK 70] McKeeman, W.N., Horning, J.J., and Wartman, D.B.: A Compiler Generator Prentice Hall, Inc., 1970.

[NAU 60] Naur, P. (ed). Report On The Algorithm Language ALGOL 60. Comm. ACM., Vol. 3, 1960.

[PRY 72] Prywes, N.S., and Smith, D.P.: "Organization of Information In Annual Review of Information Science and Technology, ed. Carlos A. Cuadra, 1972.

[RAN 64] Randall, B., and Russell, L.J.: ALGOL 60 Implementation. Academic Press, New York, 1964.

[RIC 69] Richards, M.: BCPL: A Tool For Compiler Writing and System Programming. Proc. SJCC, AFIPS, 1969.

[SMI 71] Smith, D.P.: An Approach To Data Description and Conversion, Ph.D. Dissertation, University of Pennsylvania, 1971.

[STE 61] Steel, T.B.: A First Version Of UNCOL. Proc. of the WJCC, ACM, New York, 1961.

[STR 58] Strong, J., Wegstein, J., Tritte, A., Olztym, J., Mock, O., and Steel, T.B.: The Problem of Programming Communication With Changing Machines - A Proposed Solution. *Comm. ACM.*, Vol. 1, Nos. 8 and 9, 1958.

[TRO 67] Trout, R.G.: A Compiler-Compiler System *Proc. ACM 22nd Natl. Conf.*, 1967.

[WEG 72] Wegner, P.: The Vienna Definition Language. ACM Computing Surveys, Vol. 4, 1972.

CHAPTER 1

INTRODUCTION

1.1 Summary of Research Reported

The ultimate goal of the research described in this dissertation is generally directed towards the automatic generation of programs to perform data processing, based on specifications of the desired functions and systems. The dissertation describes a Processor which has been developed to generate such programs for the limited application of file conversion. The Processor accepts as an input descriptions of Source and Target Files in a Data Description Language (DDL). It produces as an output a conversion program capable of converting the Source File and producing the Target File. To specify validation criteria, data security, summaries and reports, the user utilizes a Data Manipulation Language (DML). This report describes the Data Description and Data Manipulation Languages (DDL/DML), the operation of the Processor, and includes a user guide and full documentation of the processing system. The research was conducted in the Moore School of Electrical Engineering, University of Pennsylvania and utilized the UNICOLL IBM 370/165 computer system.

The Processor automatically produces conversion programs in PL/1, i.e., we use PL/1 as an intermediate object language in the compilation process of DDL statements into machine code. The main reason is that in generating PL/1 object code it is much easier for the compiler writer to debug the compiler.

The Processor itself has been programmed in PL/1, basically since it allows us to produce a much better documented system description.

The need for an efficient method of converting data organization in view of new user needs or for use with different programs or different computers has been recognized by the community of EDP users. Presently, a user can re-organize data by either writing his own special software, i.e., writing a new program for each file to be translated, or by using the data description facilities contained in the programming languages - such as COBOL, FORTRAN, PL/1, RPG, etc., operating systems and data management systems available for a particular computer.

The Stored Data Description and Translation (SDDT), task group of the CODASYL Systems Committee have defined Data Translation as:

> "the process whereby data stored in a form that can be processed on one computer (the source file) can be translated into a form (target file) which can be used by the same or different processing systems on a possibly different computer."

There are two main approaches to the solution of the Data Translation (conversion) problem. One consists of building the capability of converting data from external sources and formats into specific data management systems or programming languages. This capability is then limited to the specific computer system and data management system or programming language when it has been incorporated. The other approach which has been adopted here is to develop a general "Utility" which will convert data between programs and/or computer systems. Its power, then, will be general and not limited to a specific programming or computer system.

The first step towards the solution of the general Data Conversion program was the design of a Data Description Language (DDL). As Prywes and Smith [PRE 72] have said:

> "Simply speaking, a DDL is a language which enables a person to describe every aspect of a data organization, from the interrelation among elements of the organization to its representation as a linear string and its positioning on a specific storage medium. Such descriptions can serve as a basis for organizing or converting the respective data bases automatically."

Work on this area is reported by Taylor [TAY 71] and Smith [SMI 71]. Taylor's research was directed toward the specification of the mapping of data structures to linear storage spaces. Smith's research was directed toward the generalized data translation problem. Concurrent to this work the research has been done by the CODASYL Data Base Task Group [CODASYL 71]. They have defined a DDL and a DML using COBOL as their framework.

The DDL that has been developed is to a large extent similar to data definition facilities of COBOL or PL/1. The DDL is in that respect English-like; however, it possesses much more extensive facilities to describe data structure and organization than those provided in COBOL or PL/1.

1.2 Summary of Contributions

The contributions of the research are in several directions, primarily in providing the first useable Processor to convert data using a Data Descriptive language. The Processor has been adequately documented and tested for its reliability and dependability. Novel methods of implementation have been used to implement the Processor itself. In particular, methods have been employed to automatically generate the Syntactic Analysis Program (SAP) using compiler-compiler methodology i.e., a Syntactic Analysis Program Generator (SAPG) was built and in conjunction with this latter activity a new meta-language to describe the syntax and encoding of the DDL language was developed. The SAPG provides an overriding advantage over hand-coded SAP's in that it easily allows changes to the syntax of a language. In the case of DDL, a completely successful change (from the DDL designed by D. Smith [SMI 71] to the DDL used in this dissertation) was accomplished in less than a week. We also found that automatic generation of the Syntax Analysis Program via SAPG greatly speeds up and reduces the work required to implement the Processor.

We did not attempt to use a methodology for automatic code generation in the Processor, since the DDL Processor is built to generate specifically conversion programs, and therefore we concluded that research on automatic generation of code for general purpose Language compilers, such as FORTRAN, would not apply to the DDL Processor. Therefore, we hand-coded the code generation of the DDL Processor in PL/1 We assume that thereby we achieved better efficiency in the generation of the conversion program from the DDL source statements.

At the conclusion of the research we came across the work of Fang [FAN 72] where he presents the system FOLDS. This system is the first attempt at describing the code generation (semantics) of a compiler via a procedural meta-semantic language in addition to a syntactic description. Nevertheless given the nature of the DDL Processor the utilization of such meta-semantic language will not simplify the problem of building the Processor since the compiler writer will end up writing almost exactly the same logic we wrote in PL/1 in the meta-semantic language.

The DDL used in this dissertation is based on a subset of the DDL designed by Smith [SMI 71], but improvement in the latter were made to remove difficulties of use and in implementation. We later modified it for the purpose of simplifying it for ease of use and ease of implementation, and called it DDL/DML version 1.0. By changing the syntax of Smith's DDL we reduced the amount of writing, for example, in the specification of source field to target field association. Moreover, we added new capabilities and flexibility to such things as the movement of repeating fields, and simplified the specification of such items as physical storage.

The concept of default parameters was furthermore introduced. In summary, while the soul of the present DDL language is based on the DDL of Smith, we feel that we have made significant improvements to aid its use and implementations. In changing the DDL syntax we capitalized on the Data Descriptive facilities of COBOL, PL/1 and on the DDL designed by the CODASYL Data Base Task Group.

DML (Data Manipulation Language) was introduced to replace the criteria language which was part of the DDL designed by Smith. In that DDL we found that specifying conversion or validation criteria was not only cumbersome but incomplete in that it was unable to express several real-world problems we faced. In the present version 1.0 on the other hand, DML is a subset of PL/1 to allow maximum manipulative flexibility and it is used to test criteria, perform security procedures - to open "Locks" - before translation of the Source File is attempted, and to aid in the verification of Data Bases. DML can also be used as an aid to report generation. May we also add that by separating DML from DDL, we drew a distinct line between a descriptive language defining data structures (DDL), and a prescriptive procedural language for criteria specification, conversions, and manipulation (DML). In Appendix A of this dissertation we present the User-Guide for DDL and DML.

1.3 Summary of Capabilities of the DDL/DML Processor

The DDL/DML processor is designed to satisfy two requirements of data interchange: (a) data (organization) definition and (b) data translation. The first step toward simplifying data interchange is to make data and its organization explicit and independent of machines and their processors. This can be done by using a language for describing data (separate from the language used to process that data). The second step consists of developing a processor for interpreting the description and translating the data to a format appropriate for the executing machine. The DDL/DML version 1.0 satisfies the first requirement. The DDL/DML processor described in this dissertation is a set of computer programs which will perform the interpretation and translation. It processes data definitions and data translation commands, produces a computer program to perform the required data conversion and then executes this program, thereby doing the data conversion specified.

The capabilities of DDL/DML are summarized below

1.3.1 A Language For Communication Between Humans About Data Structure.

One important application of a DDL is as a means of communication between humans about the structure of data. For example, using a DDL a designer of a Data Base can describe precisely to an applications programmer the exact structure of the data the programmer wants to use. Just an BNF is now used to describe the syntax of a language so can DDL be used to describe data structures.

1.3.2 Restructuring and Conversion of Files

As a utility, the DDL processor in conjunction with DML, enables the user to redefine the structure of his file and have it converted accordingly. Furthermore, conversion of a file can be done selectively; i.e., those portions of a file that meet a user's criteria can be selectively converted or copied to a new file.

1.3.3 Interfacing Files With Different Programs and Programming Languages

Frequently files created by one program cannot be processed by another program or by another program in a different programming language. With the DDL processor, the files can be converted into a structure which can be processed by the other program. In this manner files can be interfaced across programming languages.

1.3.4 Validation of a File

By defining a file in DDL, one can validate it according to user-provided criteria written in DML.

1.3.5 Aid In Report Generation

By defining the source file to be the user data base and defining the target file to be a report form the DDL-DML processor facilitates report generation.

1.4 Important Features of the Design and Implementation

The DDL Processor System consists of three major parts. The first part uses syntax and hand coded definitions to generate the

the second part - the DDL compiler. The DDL compiler formats and translates DDL statements and produces a computer program which will do the file conversion. The third part of the processor actually processes the source file and produces as an output the target file. This three - part structure is illustrated in Figure 1-1a.

Changes in the definition of the DDL language occur only infrequently after an initial language development phase. And the compiler generator (Syntactic Analysis Program Generator) facility is primarily intended for DDL language development.

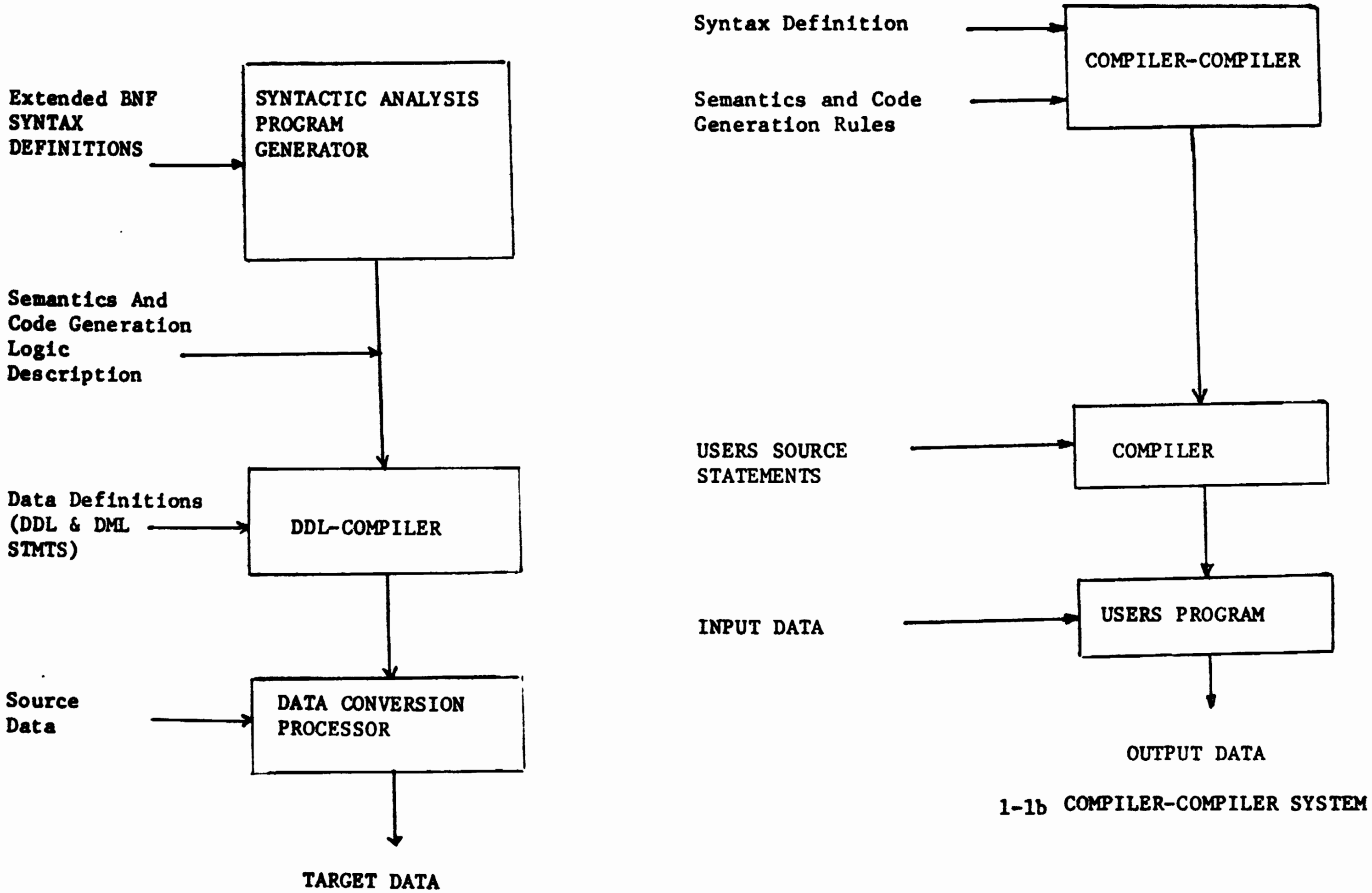
Changes in the definition of a particular data base to which the data conversion processor is applied will occur more frequently. For instance, one may wish to change definitions that are used in the conversion of data between computer systems. In this case only the sequence of DDL statements would be changed and the DDL compiler would be used to create a new Data Conversion Processor.

If the description of the data to be converted does not change, the DDL compiler need not be used, and the previously created Data Conversion Processor may be used again.

There are several features in the design which are of special interest.

(1) The use of EBNF (a meta syntactic language).

The EBNF is to be used as: (a) an aid in specifying the


Extended BNF SYNTAX DEFINITIONS
SYNTACTIC ANALYSIS PROGRAM GENERATOR
Semantics And Code Generation Logic Description
Data Definitions (DDL & DML STMTS)
DDL-COMPILER
Source Data
DATA CONVERSION PROCESSOR
TARGET DATA
Syntax Definition
Semantics and Code Generation Rules
COMPILER-COMPILER
USERS SOURCE STATEMENTS
COMPILER
INPUT DATA
USERS PROGRAM
OUTPUT DATA

1-1a The DDL PROCESSOR

1-1b COMPILER-COMPILER SYSTEM

COMPARISON OF DDL PROCESSOR AND COMPILER-COMPILER
SYSTEMS DESIGNS
FIGURE 1-1

language; i.e., a tool for compiler writers, (b) an aid to implement a syntactic analysis program generator; i.e., for the purpose of recognizing whether or not a given string belongs to the language, (c) as an aid to the user in learning how to use the language correctly (see Appendix A for the User-Guide of DDL & DML).

(2) The applications of the concept of a compiler-compiler to the DDL processor. See Feldman [FEL 68], Irons [IRO 61], Brooker and Morris [BRO 63].

This is illustrated in Figure 1-1 where the design of the DDL Processor is shown on the left (Figure 1-1a) and an analogy is drawn to a design of a compiler-compiler system as shown on the right (Figure 1-1b).

Note that the input is shown as horizontal lines and output is shown as vertical lines, thus the output of the DDL Compiler-Generator is not the input to the DDL compiler; it is, rather, the DDL compiler itself.

The Data Definition (in DDL) of a Data Base is considered analogous to the users source statements which are input to the compiler. The Data Conversion Processor is the analogue of the user's object program, output from the compiler.

(3) Transition - Matrices On Lexical Analysis

The implementation of the ideas of Floyd [FLO 69], Presser [PRE 69] and Conway [CON 63] of using transition - matrices to perform Lexical analysis for EBNF/WSC and for DDL statements.

(4) Use of EBNF with subroutines calls (EBNF/WSC) for encoding the DDL source statements.

Using EBNF/WSC - to describe the syntax of the DDL language-as an input to the Syntactic Analysis Program Generator - facilitates the encoding of DDL statements into internal tables, thus reducing the amount of work performed by code generation.

1.5 Organization of this Dissertation

Chapter 1 has attempted to give an overview of the need for the DDL/DML system, the language, and the techniques used to implement the DDL processor.

After presenting an overview of the DDL/DML processor, the bulk of the text of this dissertation deals with the design and implementation of the DDL/DML Processor System, while the Appendices contain user guides for the use of the system and related detailed documentation.

Chapter 2 provides an overview of the design of the DDL/DML processor. Chapter 3 presents an introduction to BNF, EBNF/WSC and the formal specification of DDL and DML (see also Appendices A and E). Chapter 4 describes the Syntactic Analysis Program Generator (see also Appendix B). The DDL compiler is presented in Chapter 5 (see also Appendix C and D). The Conclusions and Recommendations are given in Chapter 6.

Thus, the reader interested only in the overall description of the DDL and DML and DML Languages and in the DDL/DML Processor System need read only Chapters 2, 3 and 6.

The reader who is interested in a more detailed description of the capabilities of the system and wants to learn how to use it should read the user guide in Appendix A.

Finally, the reader interested in thorough documentation of how the DDL/DML Processor System was built should read Chapter 4 and 5 and the remaining Appendices.

1.6 Selection of Computer and Programming Language

Two computer systems available at the University of Pennsylvania that could satisfy requirements were evaluated, the RCA Spectra 70/46 Time Sharing System, and the IBM 370/75 operating under OS/370. The RCA system includes support for ALGOL, COBOL, FORTRAN, SNOBOL, and Assembler languages. The IBM system supports these and also PL/1, APL, LISP. Since both systems meet the hardware and operating system requirements, the programming language was the determinant. Assembly language programming was unsatisfactory because of programming costs, lack of machine independence, and poor readability. PL/1 was considered, by far, the best, especially in the area of memory allocation, and data management commands. The IBM 370 computer system was selected because of PL/1 availability.

CHAPTER 2

OVERALL DESCRIPTION OF THE "DDL PROCESSOR"

2.1 Major Components

The "DDL Processor" is actually three processors (sets of computer programs). The first is the Syntactic Analysis Program Generator (SAPG), the second is the DDL Compiler, and the third is the Data Conversion Processor (see Figure 1-1). When the term DDL Processor is used, it refers to the latter two processors - the DDL Compiler and the Data Conversion Processor.

An analogy with existing computer programming language systems was made in Figure 1-1; the Data Conversion Processor corresponds to an executable user program, the DDL compiler corresponds to a programming language processor such as a Cobol, Fortran or PL/1 compiler, and the Syntax Analysis Program Generator corresponds to a Compiler-Compiler's - syntax analysis generation modules which are used to produce the Cobol, Fortran or PL/1 syntactic analyzers. The relationships between and the use of the three processors in the DDL systems is readily seen from the analogy. The Data Conversion Processor is the program which reads data from existing files and produces new files. Like most data processing programs, each Data Conversion Processor is designed for a specific function, (e.g. conversion of a file in format A to a file in format B). The DDL Compiler aids in the generation of Data Conversion Processors.

To produce a Data Conversion Processor one writes a Data

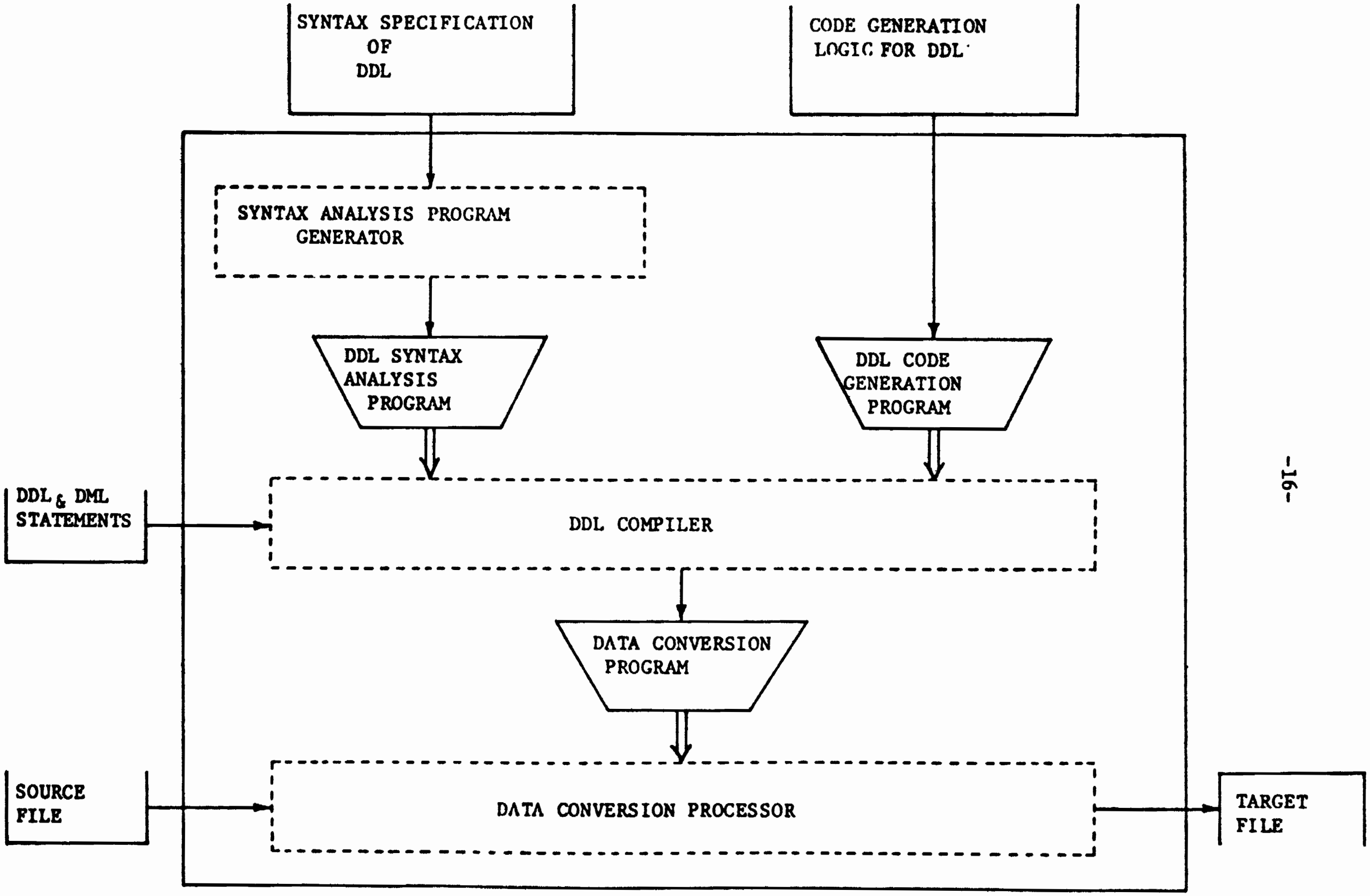
Definition (a series of statements in the DDL language) for each of the source and target files. If Data manipulation is needed a series of routines are written in DML (Data Manipulation Language). These statements are read by the DDL Compiler, which produces a Data Conversion Processor. Just as it is not necessary to compile a Cobol program each time it is used. It is important to note that it is not necessary to create a New Data Conversion Processor each time it is used.

The DDL Compiler, and the Data Conversion Processors produced by it, are the only components of the DDL Processor System that most users will need. The other component, the Syntactic Analysis Program Generator, is the program used to create the Syntax Analysis Program, which is in turn part of the DDL Compiler. The "Generator" is a very valuable tool in the development of the DDL Compiler, and is equally valuable in enhancing and modifying the Compiler. Furthermore, it would be a "stand alone" valuable tool in writing any syntax analysis program.

The components of the DDL Processor System are shown in Figure 2-1. Each processor is surrounded by a broken line. A rectangle with a missing side represents an input, while outputs are shown in trapezoids. When the output is a program, double lines are used to show the destination of the output.

2.2 Overview of The DDL Processor

In Figure 2-2 an overview of the DDL Processor is shown. The inputs to the PL/1[5] compiler to produce the DDL compiler [6] in


SYNTAX SPECIFICATION OF DDL
CODE GENERATION LOGIC FOR DDL
SYNTAX ANALYSIS PROGRAM GENERATOR
DDL SYNTAX ANALYSIS PROGRAM
DDL CODE GENERATION PROGRAM
DDL & DML STATEMENTS
DDL COMPILER
DATA CONVERSION PROGRAM
SOURCE FILE
DATA CONVERSION PROCESSOR
TARGET FILE

MAJOR COMPONENTS OF THE
DDL-PROCESSOR SYSTEM

FIGURE 2-1

in machine code for the IBM/370 are the following:

1) The Lexical Analysis Subroutine for DDL, coded in PL/1. The full description is presented in Section 5.2.1.

2) The Syntax Analysis Program (SAP) produced by the SAPG in PL/1.

3) Supporting subroutines - written in PL/1, perform services required by the SAP. Services include recognition of syntactic elements, diagnostics of syntax errors creation of Internal Tables (symbol and data tables) and if specified a cross-reference table of the identifiers used in the DDL program. (See Section 5)

4) Code Generation Program - written in PL/1, forming the basis of the code generation phase of the DDL compiler. In it is contained the logic required to interpret the information stored in the internal tables and to generate PL/1 statements which will perform the data conversion indicated by the DDL Program.

The output from the PL/1 compiler is the DDL compiler in IBM/370 machine code. (6)

The inputs to the DDL compiler (6) to produce the Data Conversion Program (8) in PL/1 are the following:

- The DDL & DML source statements (7). The DDL statements given by the user describe the structure of his source and target files. The DML statements describe the file manipulations to be prepared on the source file, such as; criteria testing, conversion methods, security testing, report generators and statistical gathering of data.

The Data Conversion Program, in PL/1 is the input to the PL/1 compiler (9) this produces the Data Conversion Program (10) in machine code for the IBM/370.

Finally the Data Conversion Program (10) accepts as input the source file (11) and outputs the target file (12).

2.3 The Syntactic Analysis Program Generator

Figure 2.3 shows the Syntactic Analysis Program Generator (SAPG). The description of SAPG is given in Chapter 4. The SAPG is indeed a compiler in its own. It consists of (a) a Lexical Analysis Module, (b) the Syntax Analysis Module and Code Generation Module. All of these are hand coded in PL/1. They are compiled into IBM/370 machine code using the PL/1 compiler (c), whose output is the Syntax Analysis Program Generator in IBM/370 machine code (e).

The input to SAPG is the DDL syntax specification, written in EBNF with subroutine calls (d). The output (f) is a Syntax Analysis Program (SAP) - in PL/1, which will perform the syntax analysis on statements written in DDL.

2.4 The DDL Compiler

An overview of the DDL Compiler is shown in Figure 2-4. The DDL Compiler contains four major parts:

(1) a Lexical Analysis Program (LEX),

(2) a Syntactic Analysis Program (SAP)

(3) a Code Generation Program (CGP)

(4) a series of supporting subroutines

OVERVIEW OF THE DDL-PROCESSOR

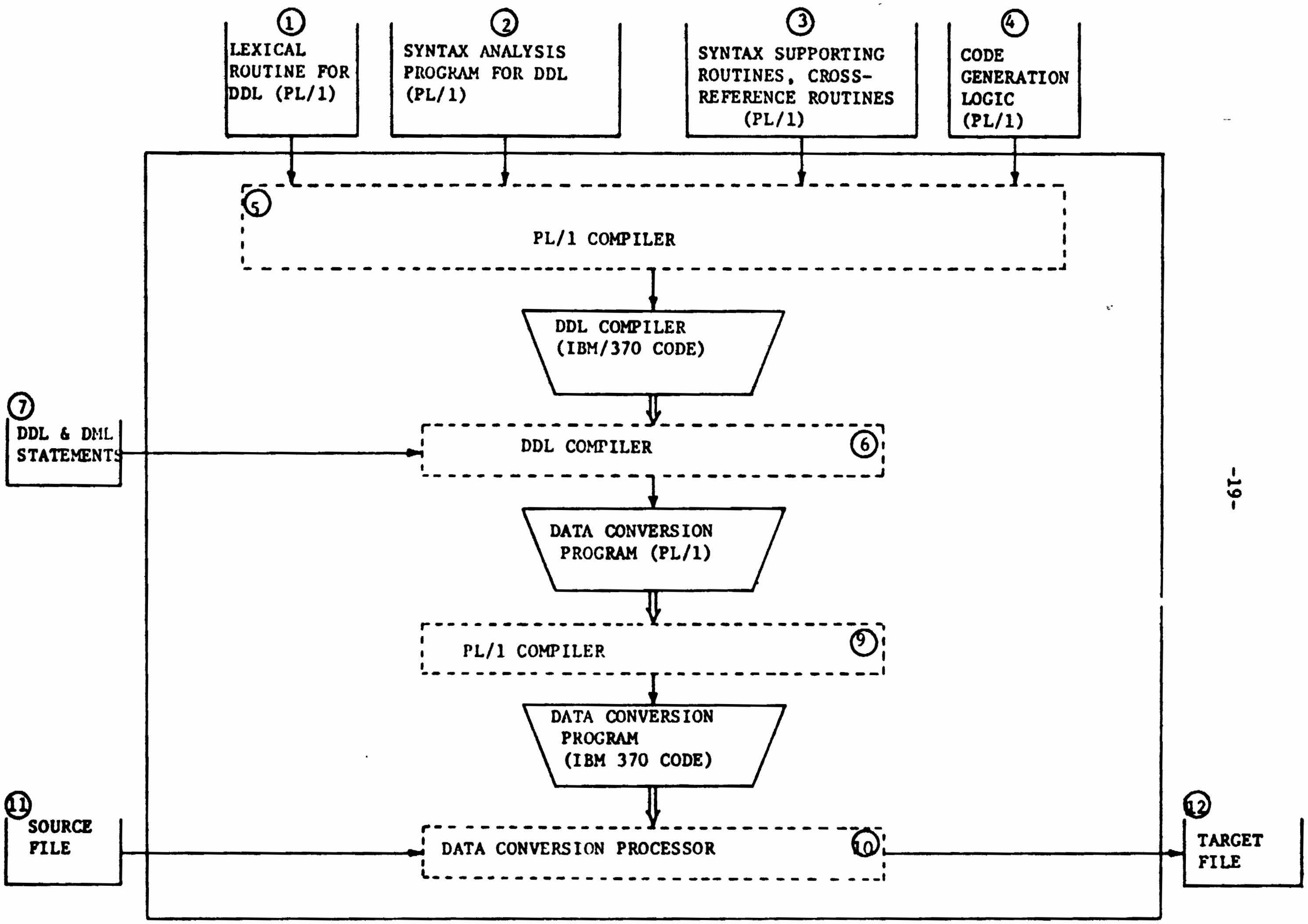

1
LEXICAL ROUTINE FOR DDL (PL/1)
2
SYNTAX ANALYSIS PROGRAM FOR DDL (PL/1)
3
SYNTAX SUPPORTING ROUTINES, CROSS-REFERENCE ROUTINES (PL/1)
4
CODE GENERATION LOGIC (PL/1)
5
PL/1 COMPILER
DDL COMPILER (IBM/370 CODE)
7
DDL & DML STATEMENTS
DDL COMPILER
6
DATA CONVERSION PROGRAM (PL/1)
PL/1 COMPILER
9
DATA CONVERSION PROGRAM (IBM 370 CODE)
11
SOURCE FILE
DATA CONVERSION PROCESSOR
10
12
TARGET FILE

FIGURE 2-2

The full description of the DDL Compiler is given in Chapter 5.

2.4.1 Phase 1 of the DDL Compiler

In phase 1, DDL statements are read by the Lexical Analysis Program (LEX) that forms tokens which are in turn the input to the Syntax Analysis Program (SAP). The SAP examines the string of tokens to determine whether or not, the string obeys certain structural conventions explicit in the syntactic definition of the language. Should an error be discovered, the error-diagnostics routines will be called to output a message informing the user the location and nature of the misconstruction.

Concurrent with this error detection is the internal table generation. At this time, routines are called whose functions are the capturing of information contained in the DDL source statements and the building of tables to preserve this data in coded form for use during code generation, as well as in the detection of global syntax errors. The internal tables that are formed are the Symbol and Data Tables. If no errors were detected and if the XREF option was specified to the DDL Compiler, then the cross-reference table is generated.

2.4.2 Phase 2 of the DDL Compiler

Phase 2 is code generation. The first part of Phase 2 is the execution of a series of programs which steps through the Data Table and generates PL/1 declare statements. When compiled, these will produce a Description Table that contains the following for each field of the source record: (1) data descriptor

SYNTAX ANALYSIS PROGRAM
GENERATOR
(SAPG)

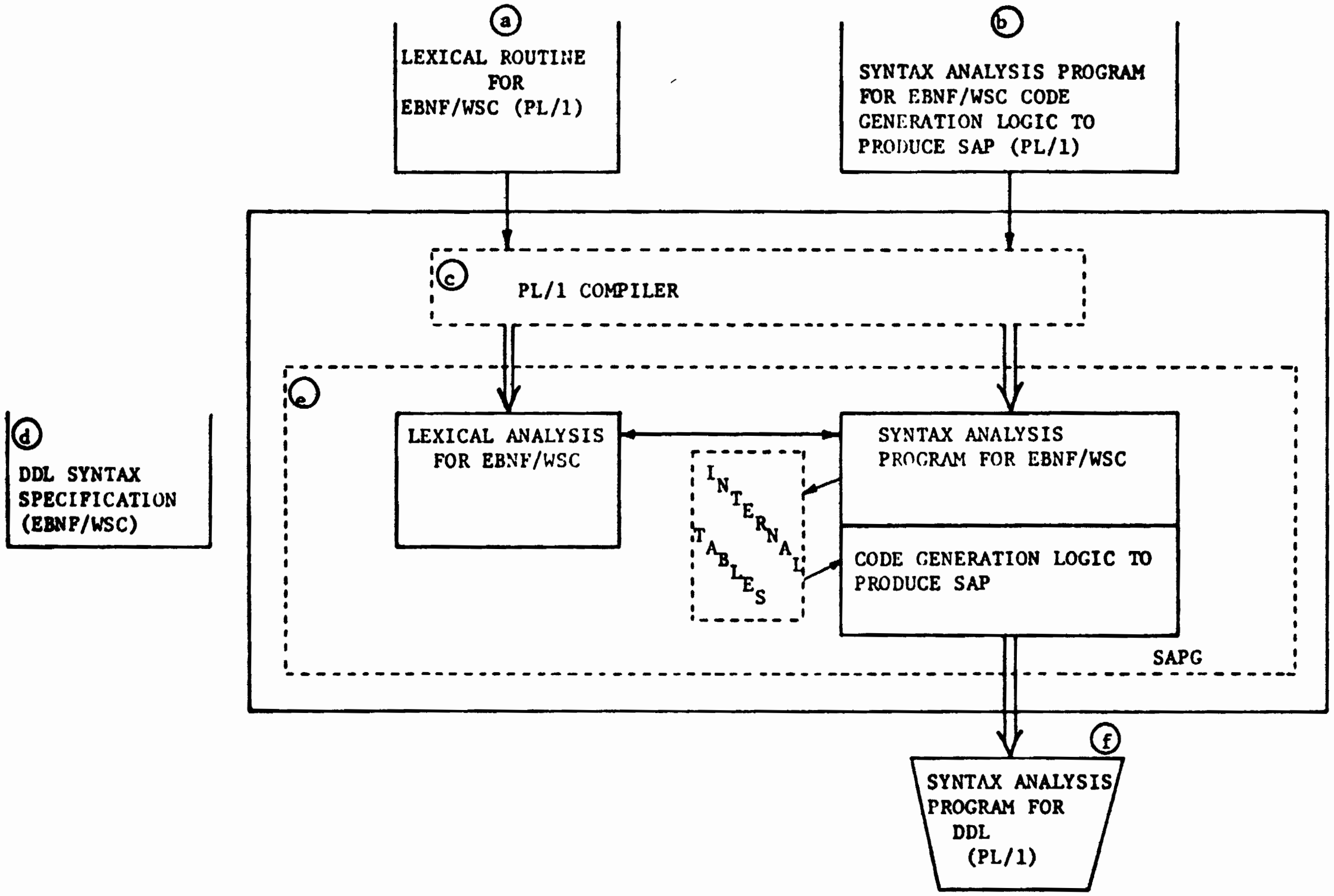

a
LEXICAL ROUTINE FOR EBNF/WSC (PL/1)
b
SYNTAX ANALYSIS PROGRAM FOR EBNF/WSC CODE GENERATION LOGIC TO PRODUCE SAP (PL/1)
c
PL/1 COMPILER
d
DDL SYNTAX SPECIFICATION (EBNF/WSC)
e
LEXICAL ANALYSIS FOR EBNF/WSC
INTERNAL TABLES
SYNTAX ANALYSIS PROGRAM FOR EBNF/WSC
CODE GENERATION LOGIC TO PRODUCE SAP
SAPG
f
SYNTAX ANALYSIS PROGRAM FOR DDL (PL/1)

FIGURE 2-3

DDL-PROCESSOR
DDL COMPILER
DATA CONVERSION PROCESSOR

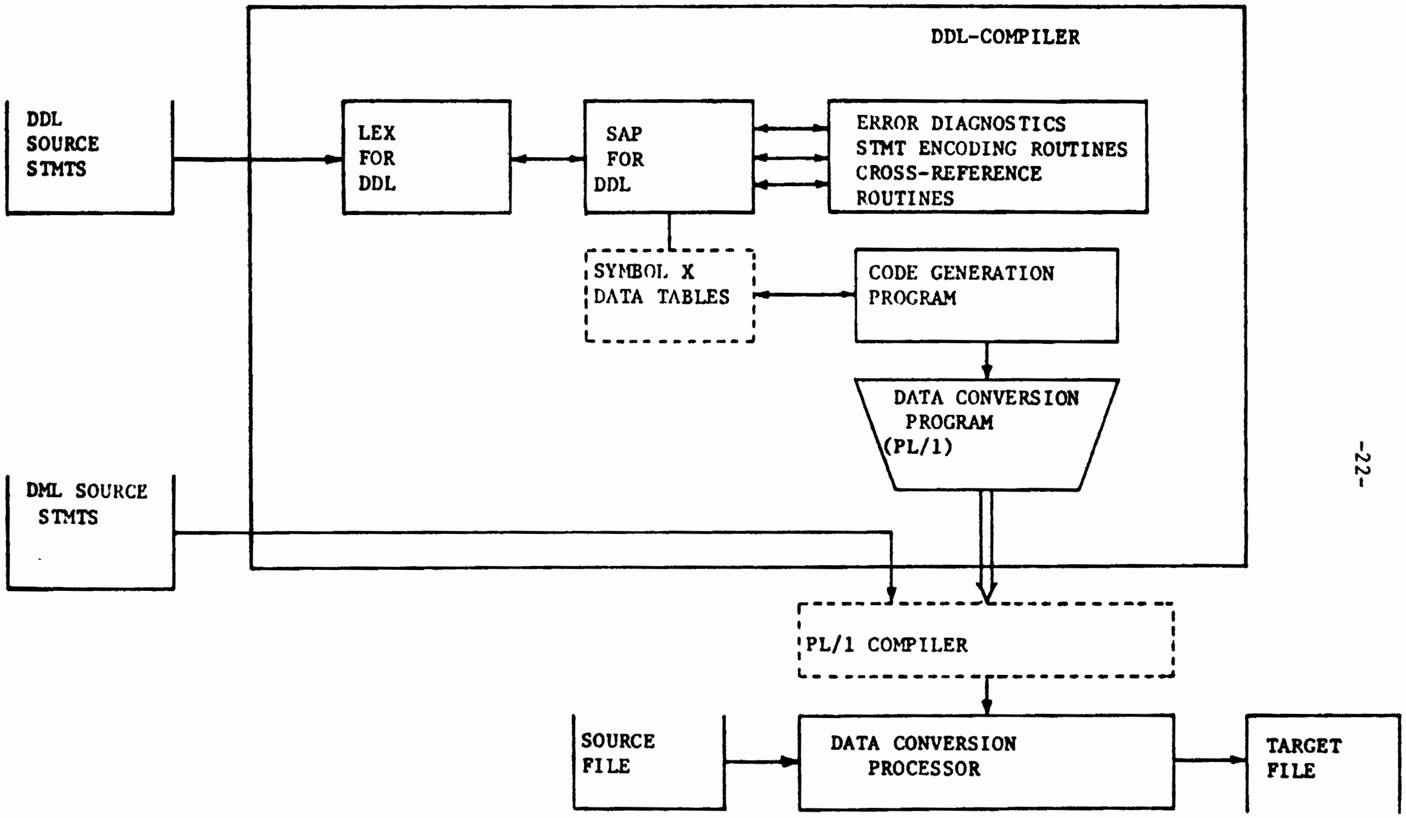


FIGURE 2-4

information, and (2) space for placing pointers to the field at execution time.

After each entry of the Descriptor Table has been produced, a call is made to the Data Parsing code generation routines. These routines generate the code which will fill in the data pointer in the descriptor table entry at run time.

The second part of Phase 2 is a program which uses the Data Table entries for the target file and generates the data movement code. This completes Phase 2 of the DDL compiler. At this point the DML statements are read by the DDL compiler and they are merged with the code produced during code generation phase.

2.4.3 Phase 3 of the DDL Compiler

Phase 3 of the DDL Processor is the creation of the Data conversion program in IBM/370 machine language. This phase is performed by the DDL compiler. The input to the PL/1 compiler is the PL/1 text produced in Phase 2.

2.5 Data Conversion Processor

The Data Conversion Processor is a set of programs and data which was produced by the Code Generation logic of the DDL compiler and the DML routines supplied by the user. It is composed of:

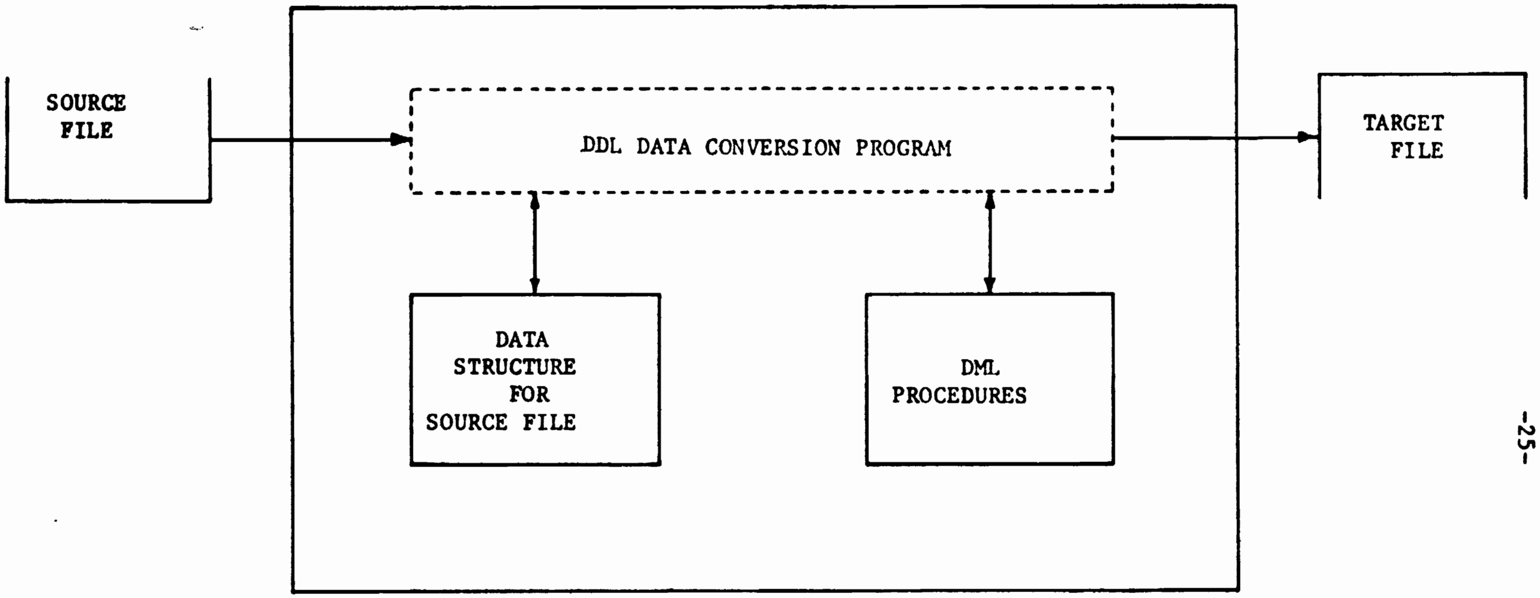
(a) a Data Conversion Program

(b) a Data Structure for the source file, and

(c) a set of DML subroutines

The Data Structure for the source file - which was produced by the DDL Compiler from the DDL statements, is used by the Data Conversion Program to aid in parsing the source data. The other component of the Data Conversion Processor, the DML subroutines, perform functions such as character set conversion, extension or truncation of fields, data type conversion criteria testing, security testing, report generators, gathering of statistical data.

An overview of the Data Conversion Processor is shown in Figure 2-5.



DATA CONVERSION PROCESSOR
FIGURE 2-5

CHAPTER 3

DDL and DML SPECIFICATION

3.1 Introduction: - Extended Backus-Naur Form

During the past decade, one of the most significant problems in the theory of programming has been the search for suitable formalisms for programming language definition.

Backus [BAC 59] has devised a technique for describing the syntax of artificial languages, i.e., a way of specifying precisely and unambiguously the grammatical or well-formed formulas of a language. The method takes the form of a language which has come to be called Backus Normal Form, usually abbreviated BNF. Historically, BNF was devised for the specific purpose of describing the syntax of ALGOL 60, and the ALGOL 60 report [NAU 60] is a conspicuous use of BNF. There are others, e.g. [LEA 64], and [CHE 62].

When describing an artificial language, one must distinguish clearly two levels of language: 1) the language being defined, usually called the object language; and 2) the language in which the describing is done, usually called the meta-language. Thus BNF is really a meta-language.

In BNF the following four meta-linguistic symbols are introduced:

< > | :: =

These are called meta-linguistic symbols since they must not appear in the alphabet of the object language. These symbols make it possible to distinguish the characters of the object language from those of the defining meta-language.

Strings of characters enclosed in the brackets < > are called meta-linguistic variables. A meta-linguistic variable is a name given to a class of strings in the object language. The value of a meta-linguistic variable is a string of symbols. The symbol | may be read as "or" and the symbol ::= may be read as "is defined as."

3.2 EBNF Without Subroutine Calls

Here at the Moore School of Electrical Engineering, the DDL group (H. Solow, A. French and the Author of this Dissertation) have extended BNF, calling it EBNF.

The extension of BNF is through the use of square brackets [ ] and the symbol *. The square brackets are used to indicate that the item enclosed may appear zero or one times. If the right bracket] is followed by a *, the item enclosed may appear zero or more times. Any set of characters appearing in a formula which is neither a variable nor one of the above symbols denotes itself, and concatenation of strings of symbols is denoted by juxtaposition of such characters and/or variables.

It is well-known that BNF can be used to describe the context-free languages. The class of languages describable by EBNF is also the context-free languages. The extensions to BNF do not add to the class of languages describable by BNF, since, as will be shown below, the syntactic extensions only give more flexibility in syntactic description. However, one extension allows information obtained during syntactic analysis to be encoded for more convenient processing during later stages of compilation [FRE1].

The use of the meta-linguistic symbols is best illustrated by example.

Example 1.

< DIGIT > ::= 0|1|2|3|4|5|6|7|8|9|

This meta-linguistic definition may be read as "A member of the meta-linguistic class DIGIT is defined as 0 or 1 or 2 or 3 or 4 or 5 or 6 or 7 or 8 or 9." Thus an occurrence of < DIGIT > in a meta-linguistic definition stands for any one decimal digit.

Both BNF and EBNF allows the use of recursion in a formula. This means in this context that the meta-linguistic class being defined may be included in its definition.

Example 2.

< INTEGER > ::= < DIGIT > | < INTEGER > < DIGIT >

This may be read as "A member of the meta-linguistic class < INTEGER > is defined as a member of the meta-linguistic class DIGIT or a member of the meta-linguistic class < INTEGER > followed by a member of the meta-linguistic class < DIGIT >." More informally one might say "An integer is either a digit or an integer followed by a digit." Alternatively one might say

1. - Every digit is an integer.

2. - If I is an integer and D is a digit, then the string ID is also an integer.

Observe that the meta-linguistic variable INTEGER appears on both the left and right side of the ::= and this makes the definition recursive. But the definition is not circular, although it may appear so at first glance. This is an analogous to the usual recursive definition of factorial:

1! = 1

N! = N X (N-1)!

The difference between BNF and EBNF is that the latter allows repetition. In EBNF the definition of the meta-linguistic variable INTEGER can be stated as:

```
< INTEGER > ::=  < DIGIT > [ < DIGIT > ]*
```

This may be read as "A member of the meta-linguistic class INTEGER is defined as a member of the meta-linguistic class DIGIT followed by zero or more members of the meta-linguistic class DIGIT."

Of course, it is not always possible to avoid recursion in definitions. An example of a definition in which recursion cannot be replaced by repetition is the following:

Example 3:

```
< SIMPLE_ARITH_EXP >  ::=  [ < UNARY_OP > ]
                                  < PRIMARY > [ < ADD_OP> < PRIMARY >]*

< PRIMARY > ::= < INTEGER >
              | NAME
              | ( < SIMPLE_ARITH_EXP> )
```

```
< NAME > ::= < ALPHA_CHAR > [ < ALPHAMERIC > ]*
< ALPHAMERIC > ::=  < ALPHA_CHAR >  |< DIGIT >
< ALPHA_CHAR > ::=  A|B|C|D|E|F|G|H|I|J|K|L|M|N|O
                 |  P|Q|R|S|T|U|V|W|X|Y|Z|$|#|_|@
< ADD_OP > ::= +|-
< UNARY_OP > ::= -|+
```

This definition is recursive since < SIMPLE_ARITH_EXP > is defined in terms of < PRIMARY > and vice versa. Some values of < SIMPLE_ARITH_EXP > are:

16

-324

-A

A+B

(A+B)

-(A+B) + 16 - (-324)

((A+B) + 16) - (A+(C-25))

EBNF includes features of regular expressions, a short discussion of one of these follows. This feature is the Kleene asterisk. It allows a given substring to occur zero or more times to form a string in the object language. For example, (A)* means that the null string (i.e., the string of length zero) is in the object language, the string "A" is in the object language, the string "AA" is in the object language, etc. The language is all strings composed of any number of occurrences, of the character "A" (including zero). This feature has been combined with BNF to form EBNF in the following way. The regular expression characters

"(" and ")" have been replaced by "[" and "]" respectively. These latter characters are always regarded as meta-linguistic symbols. They indicate that the string of characters represented by the terminal and/or non-terminal symbols which appear between them occurs optionally in a string which is a valid result of applying that production. For example:

< example 1 > ::= A [B]

means that < example 1> may be replaced by either "A" or "AB" in any production which contains < example 1 > to the right of the production sign, such as:

< example 2 >::= < example 1 > C

< example 2 > represents either the string "AC" or the string "ABC." An asterisk which appears immediately following the symbol "]" is interpreted as the Kleene asterisk. For example:

< example 3 > ::= A [B]*

< example 4 > ::= < example 3> C

means that < example 4 > represents the strings "AC", "ABC", "ABBC", "ABBBC", etc.

It is well-known that regular expressions represent exactly the set of finite-state languages. Since the finite-state languages are a proper subset of the context-free languages, they are representable by BNF without the extensions found in EBNF. However, EBNF provides an easy way to specify optionality and repetition by means of the optionality brackets, "[" and "]", and the Kleene asterisk. In BNF, this must be accomplished using recursion and making a special case of the production for the null string. This is often cumbersome to write and even more difficult to understand. This point is better illustrated by example.

Example 4.

Let the alphabet A = { 1, 2, 3 }. Let CA be the language over A given by the following BNF specification.

```
CA  ::=  1  |< CA > 2
 2  ::=  2  |< 2 >  3
 3  ::=  3  |< 3 >  CA
```

The purpose of recursion in the above productions is, to achieve repetition. The corresponding specification in EBNF could be written as:

```
< CA > ::=  1 [2[3[  < CA >] *]*]*
```

In this case the number of productions has been reduced from three to one. The structure of the words of the language is more obvious from the EBNF specification than from the BNF specification. The optionality and repetition are clearly indicated. Recognition of good strings in the language can be accomplished in a single pass of the input string, scanning from left to right (in this case with no look ahead) and without the use of a push-down to retain any of the input. Once a character in the input string is successfully recognized, it may be discarded. This will be shown in Chapter 5.

One point must be made concerning the strings of symbols which appear inside the optionality brackets. These symbols are considered as a group, an "optionality group," and as such they must appear in toto or not at all. I.e., the string is bad if only part of the optionality group appears. For example, if < example 5 > ::= A[BC] then < example 5 > represents the strings "A" and "ABC", but not the string "AB". However, optionality groups may be nested inside one another such as: < example 6 > ::= A[B[C]]. In this case < example 6 > represents the strings "A", "AB", and "ABC". There is theoretically no limit to the depth of such nesting. Further, the Kleene asterisk may always appear after the symbol "]", no matter at what level of nesting it appears. (See Example 4).

As in regular expressions A[B[C]*]* is not equivalent to A[B]*[C]*. The latter represents strings containing an "A", followed by any number of "B"'s, followed by any number of "C"'s. The former represents strings containing an "A", followed by any number of strings which contain a "B" followed by any number of "C"'s.

In the following Sections 3.3 and 3.4 we present the formal specification of version 1.0 of DDL and DML respectively.

Version 1.0 of DDL/DML is a modification made to the DDL designed by D. Smith [SM 71]. The syntax of DDL has been changed to the present one in order to simplify its use and the implementation of the DDL Processor. We reduced the amount of writing, for example, in the specification of source field to target field association. Moreover, we added new capabilities and flexibility to such things as the movement of repeating fields, and simplified the specification of such items as physical storage. The concept of default parameters was furthermore introduced. In summary, while the soul of the present DDL language is based on the DDL of Smith, we feel that we have made significant improvements to aid its use and implementation.

In changing the DDL syntax we capitalized on the Data Descriptive facilities of COBOL, PL/1 and the DDL designed by the CODASYL Data Base Task Group [ ].

DML was introduced to replace the criteria language which was part of the DDL designed by Smith. In that DDL we found that specifying conversion or validation criteria was not only cumbersome but incomplete in that it was unable to express several real-world problems we faced. In the present version 1.0 on the other hand, DML is a sub-set of PL/1 to allow maximum manipulation flexibility and it is used to test criteria, perform security procedures - to open "Locks" before translation of Source File is attempted, and to aid in the verification of Data Bases. DML can also be used as an aid to report generation. May we also add that by separating DML from DDL, we drew a distinct line between a descriptive language defining data structures (DDL), and a

```
< POSITION > ::=  < LABEL >

                 | < INTEGER >

< CONVERT_STMT>  ::= CONVERT  (  < FILE_NAME >  INTO< FILE_NAME> )

< FILE_NAME > ::=  < NAME >

< FILE_STMT > ::= FILE  (  < RECORD_NAME >

                          [ ,CHAR_CODE =  < CODE >]

                            ,STORAGE =  < NAME > )

< CODE > ::= BCD

            |  ASCII [ ,OPT_BLOCK_PREFIX =  < INTEGER >]

            |  EBCDIC

< RECORD_STMT > ::=  RECORD (< NAME_LIST > [ ,< NAME_LIST >]*

                                           [ ,LOCK =  < PROC_CALL> ]

                                           [ ,SIZE =< REC_SIZE >])

< REC_SIZE > ::=  FIXED (< INTEGER >)

                |   VARIABLE (< INTEGER > )

< NAME_LIST > ::= < NAME > [< OCCURRENCE >]

< OCCURRENCE > ::= ( < MIN_OCC > )

                  |  (< MIN_OCC > : < MAX_OCC >)  < CRITERION >

< MIN_OCC > ::=   < INTEGER >

< MAX_OCC >::= < INTEGER >

< CRITERION > ::= ,PRE_CRIT = '< NAME >'

                 | ,POST_CRIT = '< NAME >'

< GROUP_STMT > ::= GROUP  ( < NAME_LIST > [ ,< NAME_LIST > ]*)

< FIELD_STMT > ::= FIELD (< TYPE > [ < DELIMETER >] [< CONV >])
< TYPE > ::= BIT  (< LENGTH >) [ < BIT_ASSGN >]
           | CHAR (< LENGTH >) [< CHAR_ASSGN >]
           | NUM_PICTURE = '< NUM_PICTURE_SPEC >' [< NUM_ASSGN >]
           | CHAR_PICTURE = ' < CHAR_PICTURE_SPEC >' [< CHAR_ASSGN>]
< LENGTH > ::= *
             | < INTEGER >
             | < PARAM_STMT >
             | < REF_NAME >
             | < PROC_CALL >
< NUM_PICTURE_SPEC > ::= < INTEGER_SPEC >
                       | < SIGNED_INTEGER_SPEC >
                       | < FIXED_NUM_SPEC >
                       | < FLOT_NUM_SPEC >
< INTEGER_SPEC > ::= 9[9]*
< SIGNED_INTEGER_SPEC > ::= [S]< INTEGER_SPEC >
< FIXED_NUM_SPEC > ::= [S] [< INTEGER_SPEC >] [< P>]
                          [< INTEGER_SPEC >]
< FLOAT_NUM_SPEC > ::=< FIXED_NUM_SPEC >< EXP>< SIGNED_INTEGER_SPEC >
< P >::= .| V
< EXP >::=  E|K
< CHAR_PICTURE_SPEC > ::=< SINGER_CHAR > [< SINGLE_CHAR >]*
< SINGLE_CHAR >::= A|X|9
< CHAR_ASSGN >::=< - '< CHAR_STRING >'
               |< = '< SOURCE_NAME >
```

```
< BIT_ASSGN > ::=  < -  '< BIT_STRING > '

                   |  <  =  '< SOURCE_NAME >'

< NUM_ASSGN > ::=  < -  '< NUM_STRING > '

                   |  <  =  '< SOURCE_NAME >'

< DELIMETER > ::= ,DELIM= '<PUNCT_MARK >'

< PUNCT_MARK >::=   < CHAR_STRING >

< CONV >::=   ,CONV=  < PROC_CALL >

< SOURCE_NAME > ::=  < PARAM_STMT >

                   | < NAME > [  < SUBSCRIPT >]

                     [.  < NAME > [<  SUBSCRIPT > ]]*

< SUBSCRIPT > ::=< BOUND > [: < BOUND >]

< BOUND > ::=  < INTEGER >

              |  <  PARAM_STMT >

              |  <  REF_NAME >

< PARAM_STMT > ::=  < LENGTH_STMT >

                   |  <  COUNT_STMT >

< LENGTH_STMT > ::= DDL_LENGTH  (< DATA_NAME > )

< COUNT_STMT >::= DDL_COUNT  (< DATA_NAME > )

< DATA_NAME > ::=  < REF_NAME >

< CARD_STMT > ::=  CARD

< TAPE_STMT > ::=   TAPE  (< TAPE_DATA_CTL_BLOCK >)

< TAPE_DATA_CTL_BLOCK > ::=  < RECORD_FORMAT >

                              ,VOL_NAME =  < NAME >

                             [,NO_TRKS =  < NO_TRKS >]

                             [,DENSITY =  < DENSITY > ]
```

```
                              [,REC_MODE =  < REC_MODE >

                              [,TAPE_LABEL =  < TAPE_LABEL > ]

                              [,START_FILE =  < INTEGER > ]

                              [,CTL_CHAR = < CTL_CHAR > ]

< RECORD_FORMAT > ::= FIXED (< BLOCK_SIZE > [, < RECORD_SIZE >

                            [,PAD = < NAME > ]])

                  |  VARIABLE (< MAX_BLOCK_SIZE >[, < MAX_RCD_SIZE >])

                  |  VAR_SPANNED (< MAX_BLOCK_SIZE >

                                  [, < MAX_RCD_SIZE >])

                  |  UNDEFINED (< MAX_BLOCK_SIZE >)

< NO_TRKS > ::= 7|9

< PARITY > ::= ODD|EVEN

< DENSITY > ::= 200|556|800|1600

< REC_MODE > ::= ALL_BIN |ALL_CHAR |MIXED

< TAPE_LABEL > ::= IBM_STD ,INT_NAME =  < NAME >

                 |  ANSI_STD ,INT_NAME = < NAME >

                 |  NONE

                 |  BYPASS [,INT_NAME =  < NAME >

< CTL_CHAR > ::= A|M

< BLOCK_SIZE >::= < INTEGER >

< RECORD_SIZE > ::= < INTEGER >

< MAX_BLOCK_SIZE >::= < INTEGER >

< MAX_RCD_SIZE > ::= < INTEGER >
< DISK_STMT > ::= DISK (< DISK_DATA_CTL_BLOCK >)

< DISK_DATA_CTL_BLOCK > ::= < RECORD_FORMAT >,VOL_NAME =< NAME >

                            ,INT_NAME =< NAME >

                            [,UNIT =  < TYPE DSK >]

                            [,SPACE = ( < PARA METERS >]]

< PARAMETERS > ::= < UNITS >,< QUANTITY > [,< INCREMENT >

                   [,RISE]

< UNITS > ::= TRACKS

            |  CYLINDERS

            |  INTEGER

< QUANTITY > ::= < INTEGER >

< INCREMENT > ::= < INTEGER >

< TYPE DSK >::= 2314|3330|2305

< BIT_STRING > ::=< BIT >[< BIT >]*

< CHAR_STRING > ::= < FULL_CHAR_SET > [< FULL_CHAR_SET > ]*

< NUM_STRING > ::= [< SIGN >] [< DIGIT >]* [.] [< DIGIT >]*

                   [ E [< SIGN > ] <DIGIT> [< DIGIT > ]*]

< INTEGER>::=< DIGIT > [< DIGIT > ]*

< NAME > ::= < ALPHA_CHAR > [< ALPHMERIC > ]*

< SUB_NAME > ::= < NAME >[(< SUBSCRIPT > )]

< REF_NAME > ::= < SUB_NAME > [.< SUB_NAME > ]*

< SUBSCRIPT > ::= *| < INTEGER >

< LABEL > ::= < NAME >

< PROC_CALL > ::= '< LABEL >'

< SIGN > ::= +|-
< FULL_CHAR_SET > ::= < ALPHAMERIC >

                      | < SPEC_CHAR >

< ALPHAMERIC > ::=  < ALPHA_CHAR >

                   | <  DIGIT >

< ALPHA_CHAR > ::= A|B|C|D|E|F|G|H|I

                   | J|K|L|M|N|O|P|Q|R

                   | S|T|U|V|W|X|Y|Z|$|-|#|@

<DIGIT > ::= 0|1|2|3|4|5|6|7|8|9

<BIT > ::= 0|1

<SPEC_CHAR > ::=  ,|;|:|.|?|!|'|"

                 | (|)| |#|X|*|@|$|-

                 | +|-|/|=|< | >| %|¬
```

3.4 FORMAL SPECIFICATION OF DML

```
< DML_PROGRAM >::= [< DML_BODY_STMTS > ]* < DML_PROGRAM >

< DML_BODY_STMTS > ::=  < DATA_MANIPULATION_STMTS >

                        | < CONTROL_STMTS >

                        | < END_STMT >

< DATA_MANIPULATION_STMTS > ::=  < ASSIGNMENT_STMTS >

                                 | <  DECLARE_STMT >

< CONTROL_STMTS > ::=  < GO_TO_STMT >

                      | <  DO_GROUPS >

                      |  < IF_STMT >

                      | <  PROCEDURE_STMT >

                      | <  CALL_STMT >

                      | <  RETURN_STMT >
```

```
< ASSIGNMENT_STMTS > ::= < VAR_ASSIGN_STMT >
                       | < POINTER_ASSIGN_STMT >

< VAR_ASSIGN_STMT > ::= [ < STMT_LABEL >:] < VARIABLE >=
                                            < EXPRESSION > ;

< POINTER_ASSIGN_STMT > ::= [< STMT_LABEL > :] < SIMPLE_POINTER >
                              = < POINTER_EXP > .PTR;

< POINTER_EXP > ::= < VARIABLE > [. < VARIABLE > ]*

< DECLARE_STMT > ::=  DCL < ARRAY_NAME > [ < ATTRIBUTES > ]
                      [BASED (< SIMPLE_POINTER > )];
                   |  DCL < FUNCTION_NAME > ENTRY
                                           [RETURNS (< ATTRIBUTES>)];

< ARRAY_NAME  >::= < NAME > [(< INTEGER > [:< INTEGER >])]

< ATTRIBUTE > ::=  BIT (< STRING_LEN >) [VARYING]
                |  CHAR (< STRING_LEN >) [VARYING]
                |  PICTURE '< PICTURE_SPEC >'

< STRING_LEN > ::= *
                | < EXPRESSIONS >

< PICTURE_SPEC > ::=  < NUM_PICTURE_SPEC >
                   |  < CHAR_PICTURE_SPEC >

< SIMPLE_POINTER > ::= < NAME > [. < NAME > ]*

< EXPRESSION > ::= < ARITH_EXP >
                 | < STRING_EXP >

< STRING_EXP > ::= < STRING_TERM > [||< STRING_TERM > ]*
< STRING_TERM > ::= < STRING  >

                  | <  VARIABLE >

                  | <  FUNCTION_CALL >

< ARITH_EXP > ::= [ < UNARY_OP >] < TERM > [ < ADD_OP > < TERM > ]*

< TERM > ::= < FACTOR > [ < MULT_OP > < FACTOR > ]*

< FACTOR > ::= < PRIMARY > [** < PRIMARY > ]*

< PRIMARY > ::= < UNSIGNED_NUMBER >

              |  < VARIABLE >

              |  < FUNCTION_CALL >

              |  ( < ARITH_EXP > )

< FUNCTION_CALL > ::= < FUNCTION_NAME > [( < ARG_LIST > )]

< FUNCTION_NAME > ::= < LABEL >

< ARG_LIST >  ::= < EXPRESSION > [, < EXPRESSION > ]*

< UNARY_OP > ::= +|-

< ADD_OP > ::= +|-

< MULT_OP > ::= *|/

< GØ_TØ_STMT > ::= [< STMT_LABEL > :] GO TØ < STMT_LABEL >;

< DO_GROUP > ::= < DO_STMT > < PHRASE > [ < PHRASE >]* < END_STMT >

< DO_STMT > ::=  [ < STMT_LABEL > :] DO;

              |  [ < STMT_LABEL > :] DO < VARIABLE > = < INIT_VALUE >

                                         TO < MAX_VALUE >

                                         [BY < INCREMENT > ];

< PHRASE > ::= < DML_BODY_STMT >

             |  < DO_GROUP >
```

```
< INIT_VALUE > ::=  < ARITH_EXP >

< MAX_VALUE > ::=  < ARITH_EXP >

< INCREMENT > ::=  < ARITH_EXP >

< IF_STMT > ::= [ < STMT_LABEL> :] IF  < COND_EXP > THEN  < CLAUSE > ;

                                  [ELSE [  < CLAUSE > ];]

< COND_EXP > ::=  < ARITH_EXP >  < COND_OP > < ARITH_EXP>

< COND_OP >  ::=  = | < | > | ¬ = | ¬ < | ¬ > | < = | < = | > =

< CLAUSE > ::=  < DML_BODY_STMT >

           |   < DO_GROUP >

< PROCEDURE_STMT > ::=  < PROC_NAME > : PROC [(< PARAM_LIST > )]

                                     [RETURNS(< ATTRIBUTES > )];

< PROC_NAME > ::=  < LABEL >

< PARAM_LIST > ::=  < NAME > [, < NAME  >]*

< CALL_STMT > ::= [< STMT_LABEL > :] CALL  < PROC_NAME >

                                   [(  < ARG_LIST > )];

<  PROC_NAME > ::=  < LABEL >

<  ARG_LIST > ::=  < EXPRESSION > [,< EXPRESSION > ]*

<  RETURN_STMT > ::= [< STMT_LABEL > :] RETURN [(< VALUE > )];

<  VALUE > ::=  < EXPRESSION >

<  END_STMT > ::= [  < STMT_LABEL > :] END;
```

3.5 EBNF With Subroutine Calls (EBNF/WSC)

EBNF/WSC is the input to the Syntactic Analysis Program Generator, it is a prescriptive as well as descriptive meta-language. By using EBNF/WSC to describe DDL, the compiler writer has a powerful technique to create and maintain internal tables as is explained in Section 5.3.6, this is carried out by the Syntactic Analysis Program (SAP), which is in turn generated by the SAPG whose input is EBNF/WSC.

It is the addition of subroutine calls to BNF that gives EBNF the capability of permitting the encoding of information from syntactic analysis for later processing. This is done by embedding these calls in the syntactic specification of the language. Doing so provides the capability to branch to a subroutine upon successful recognition of a syntactic unit in order to allow it to be encoded in a form convenient to the compiler writer. This capability can also be used in a Syntactic Analysis Program (SAP) to produce error diagnostics (see Section 5.2.3.1). The use of subroutine calls is illustrated by the following example:

```
< FILE_STMT > ::= FILE/DFILE /(< RECORD_NAME > /FRN/
                  [,CHAR_CODE =  < CODE >]
                   ,STORAGE =   NAME   /FSN/)
< CODE > ::= / FILCODE / BCD / FC3
           |    ASCII / FC 2 / [,OPT_BLK_PREFIX = /BLKPRF /
              < INTEGER > / OC / ]
           |    IBCDIC / FC1 /
```

After successfully recognizing the keyword FILE the SAP calls on DFILE, this routine is used to fill the error stack to produce error diagnostics, if any. The error codes that DFILE pushes into the error stack are the following:

Error Code	Meaning
FIL_01	open parenthesis missing after keyword 'FILE
FIL_02	invalid record name specification in FILE stmt
FIL_03	keyword ",STORAGE=" not found in FILE stmt
FIL_04	invalid storage specification in FILE stmt.
FIL_05	close parenthesis for FILE stmt missing

Then DFILE calls on the routine DFILETG which is used to allocate an entry to store the pertinent information in the FILE statement. This routine also sets the type of the entry just allocated to FILE and the character code to its default value EBCDIC. After (" has been recognized the error stack is popped-up (i.e., FIL_02 is now at the top of the stack). Then if < RECORD_NAME > is also recognized the error stack is popped-up and SAP calls on the FSH. FSH is a routine that stores (in the entry for the FILE statement) the record name just recognized. If the routine that recognizes < RECORD_NAME > exists false, then give error code in top of stack (FIL_02) and stop the recognition process for < FILE_STMT >.

If the next unit to be recognized is ",CHAR_CODE=" then SAP calls on the routine to recognize < CODE > . The procedure CODE calls on the routine FILCODE, which pushes into the error stack the following error code:

FIL_06	Invalid character code specification.

If < CODE > is BCD the error-stack is pop-up and the routine FC3 is called, this routine sets the character code field in the FILE entry to BCD. If < CODE > is ASCII the error stack is pop-up and the routine FC2 is called. This routine sets the character code in the FILE entry to ASCII, if ",OPT_BLK_PREFIX=" is found in the input then the SAP calls on BLK PRF which is a routine that pushes the following error code into the error stack:

FIL_07 invalid block prefix specification for ASCII in FILE statement.

BLK PRF also set a flag in the FILE entry to indicate that the optional prefix block is present. Then after < integer > has been recognized the routine OC stores the integer in the FILE entry. Note that failure to recognize ",OPT_BLK_PREFIX=" does not cause the recognition process for < FILE_STMT > to terminate. However, if ",OPT_BLK_PREFIX=" was recognized, then failure to recognize < INTEGER > would cause SAP to give an error code (the code on top of the error stack FIL_07) and the process to recognize < FILE_STMT > to stop. Finally in recognizing < CODE > if it was EBCDIC the routine FC2 is called. This routine sets the character code in the FILE entry to EBCDIC. Note again that failure to recognize ",CHAR_CODE=" does not cause the recognition process of < FILE_STMT > to terminate.

Finally after ",STORAGE=" has been recognized the error stack is popped-up (i.e., FIL_04 is now on top) and the routine to recognize NAME is called. If < NAME > was recognized, the error-stock is popped-up (i.e., FIL_05 is now on top) and the routine FSN is called. This routine stores the "storage name" in the FILE entry. Successful recognition of ")" causes SAP to pop-up the error-stack (i.e., now it is empty) this means that < FILE_STMT > has successfully been recognized.

Two points should be mentioned with regard to subroutine calls and EBNF syntax:

(1) Subroutine calls may appear anywhere except between "< " and "> ".

(2) An EBNF production may consist of nothing but subroutine calls as the right side of the production.

Note that subroutine calls are indicated by enclosing the subroutine name in "/"'s. Use of the "/" in any other context causes it to be treated literally, i.e., as a terminal symbol in the object language.

Internal table creation and maintenance is completely at the compiler writer's discretion. See Section 5.3.6. Subroutine calls are made only if everything specified by the EBNF production in which the subroutine call appears has been successfully recognized up to the point of the subroutine call. This information should enable the compiler writer to plan what he must do with a syntactic unit after it has been recognized.

CHAPTER 4

THE SYNTACTIC ANALYSIS PROGRAM GENERATOR

4.1 INTRODUCTION

One of the main problems in using computers is that of effective programming. An important advance was made with the introduction of mechanical translators as an aid in preparing programs. An easy way to use artificial language was developed and a translator written to translate that language into a machine language. Initially these translators were handwritten in an "ad-hoc" manner for a particular machine and language. But using the theory of automata and formal linguistics as tools compiler writers were able to develop better techniques for translator construction. An important step was the development of a formal language in which to describe the syntax of a programming language.

The definition of the ALGOL syntax [NAU60] was an early and successful attempt to describe programming languages in a formal way. The automatic construction of compilers is based on such a formalization.

Table driven techniques included in this broad category are all of the techniques reviewed by Feldman and Gries [FEL 68]. Broadly speaking, some formal description of the language to be compiled with perhaps the semantics of each production, usually in table form, serve as one input to the compiler. The compiler uses this input to parse the source statements.

The main advantage of this approach is that the tables can be changed to those for another language; these tables are usually highly stylized and rewriting them requires much less work than the rewritting of an entire compiler.

Such compilers therefore can be looked upon as the framework for the production of a processor for any particular language. Table driven compilers essentially use the tables in an interpretive way, i.e., scanning the source and matching it with production in the tables.

The following excerpt from a paper by J.A. Feldman [FEL 66] on Formal Semantic Language (FSL), will help us in the explanation of the operation of a compiler writing system.

> "When a compiler for some language, L, is required, the following steps are taken. First the formal syntax of L, expressed in a syntactic meta-language, is fed into the syntax loader. This program builds tables which will control the recognition and parsing of programs in the language L. Then the semantics of L, written in a semantic meta-language, is fed into the semantic loader. This program builds another table, this one containing a description of the meaning of statements in L. Finally, everything to the left of the double line [in Figure 4-1] is discarded leaving a compiler for L."

The compiler writing system uses two formal languages to describe a compiler. First, a syntax analyzer for the source language L is written as a program in the "production language." This program is processed by a translator called production loader producing as output a set of driving tables which are stored for later use. Second, a collection of semantic routines is defined prescriptive procedural language for criteria specification, conversions, and manipulation (DML).

In Appendix A of this dissertation we present the User Guide for DDL and DML.

3.3 FORMAL SPECIFICATION OF DDL

```
< DDL_PROGRAM > ::= [< DDL_BODY_STMTS >]* < DDL_PROGRAM >
< DDL_BODY_STMTS > ::= END;
                     | < COMMAND_STMTS >;
                     | < NAME > IS < DESCRIPTION_STMTS >;
< COMMAND_STMTS > ::= < CONVERT_STMT >
                    | < SCAN_STMT >
< DESCRIPTION_STMTS > ::= < RECORD_SPEC_STMT >
                        | < FILE_SPEC_STMT >
                        | < STORAGE_SPEC_STMT >
< RECORD_SPEC_STMT > ::= < FIELD_STMT >
                       | < GROUP_STMT >
                       | < RECORD_STMT >
< FILE_SPEC_STMT > ::= < FILE_STMT >
< STORAGE_SPEC_STMT > ::= < CARD_STMT >
                        | < TAPE_STMT >
                        | < DISK_STMT >

< SCAN_STMT > ::= SCAN (REC = < RECORD_NAME > : < GROUP_NAMES >
                  [ , < GROUP_NAMES >]*)
< RECORD_NAME > ::= < NAME >
< GROUP_NAMES > ::= < NAME > [(< POSITION >)]
```

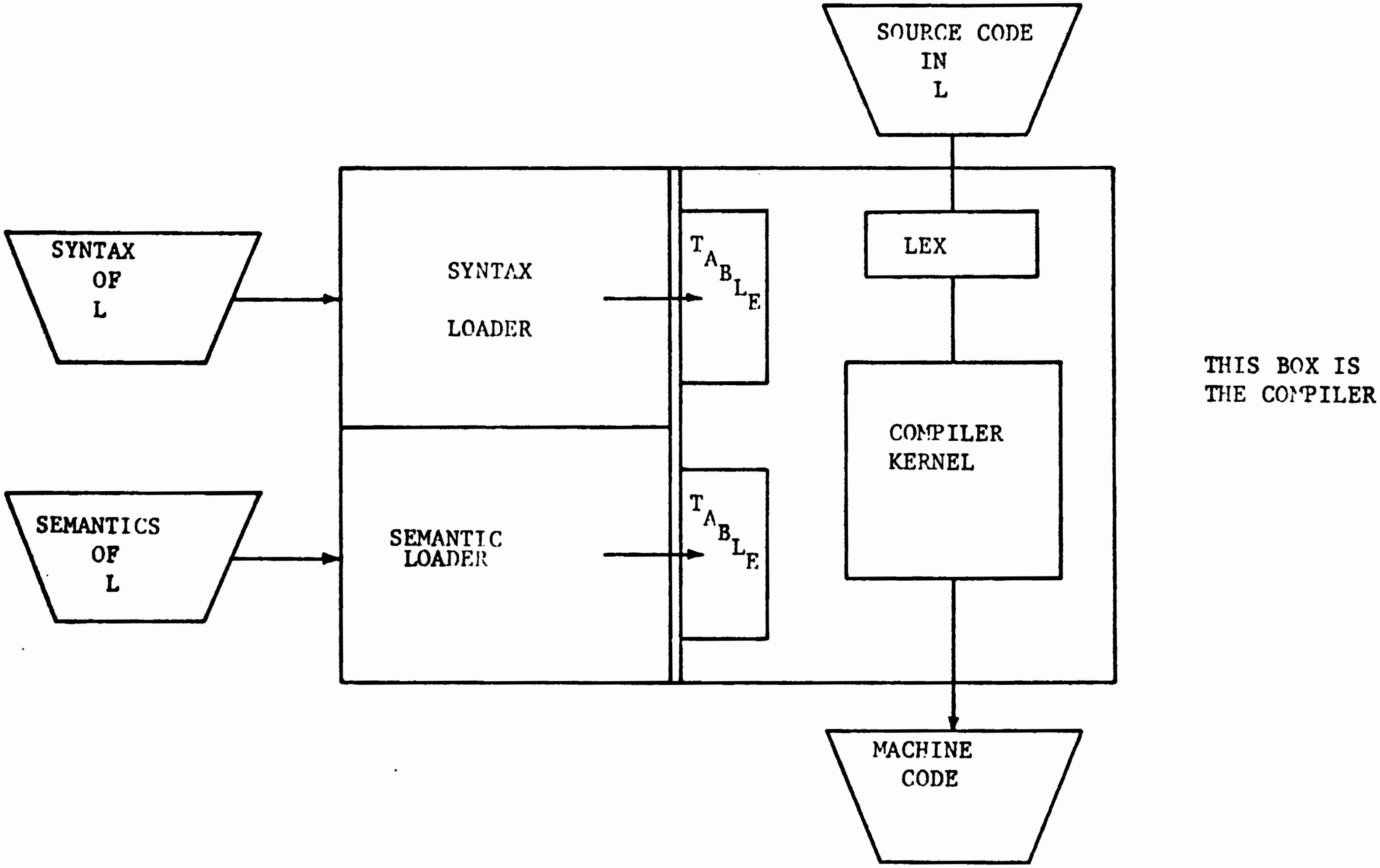


FIGURE 4-1
A COMPILER-COMPILER [FEL 66]

by writing a program in the "formal semantic language" (FSL). Another translator called the semantic loader then translates this collection of routines into a set of tables and a block of code, which code is compiled for use as part of the compiler itself. This output is also intermediate and stored for latter use.

The compiler itself (Figure 4.1) is another program which reads in both the syntax tables and the semantic tables and code, and by using these translates a source language program into an object program. For the sake of efficiency a preliminary lexical transformation is performed on source language text as it is read in by a routine called LEX.

The compiler which is the result of this process is a table-driven translator. The compiler kernel includes input-output, code generation routines, and other facilities used by all translators. Examples of this type may be found in the work of E.T. Irons [IRO 61]; Ledly and Wilson [LED 62]; Brooker and Morris [BRO 63]; M.E. Conway [CON 63]; R.W. Floyd [FLO 64]; R.M. McClure [McC 65]; W.L. Johnson, J.H. Porter, S.I. Ackley and D.T. Ross [JOH 68]; R. Iturniaga and T.A. Standish [ITU 66]; P. Ingerman [ING 66]; R.G. Trout [TRO 67]; M. Richards [RIC 69] and W.M. McKeeman, J.J. Harning and D.B. Wortman [McK 70].

The Syntactic Analysis Program Generator (SAPG) corresponds to part of the compiler-compiler described by Feldman. The formal meta-language for describing syntax is Extended Backus-Naur Form (EBNF) with subroutine calls. Figure 4-2 shows this correspondence. Everything to the left of the double line in Figure 4-? is discarded leaving a compiler for DDL.

Section 3.1 presents the syntax of the meta-language EBNF, along with examples of its use. Section 3.2 describes its use for purely describing syntax. Section 3.5 presents its use for internal table creation and maintenance and producing error diagnostics.

4.1.1 Comparison of SAPG To The XPL System

Similar principles as those found in SAPG can be found in the XPL system [McK 70]. The syntax analysis is done in XPL by using two modules; the ANALYZER and the SKELETON. The ANALYZER, is a program which reads a BNF grammar, determines whether it is acceptable, and constructs parsing decision tables for that algorithm.

The input to the ANALYZER is the BNF grammar (in a somewhat machine oriented form) punched one production per card. It requires that all cards containing productions with the same left part be grouped together and restricts right parts to strings of one to five symbols. The beginning of a production must start in column 1.

The input to the SAPG is the EBNF/WSC grammar also via punched cards, but it is more human oriented. A production is punched in free format (extra blanks between symbols are ignored) and the user can use as many cards as necessary to describe a production.

The name of non-terminal symbols can be up to 31 symbols. The beginning of each production must start in column 1.

One interesting feature of the XPL system is specific the ANALYZER is that it reduces the tables for lexical analysis from the BNF grammar. In the case of DDL the compiler writer must design the tables for lexical analysis.

The function of the SKELETON is to check the input (source language) according to the BNF grammar. The ANALYZER produces a set of tables for SKELETON. The tables are physically inserted into the body of the SKELETON program.

The function of the SAP presented in this dissertation is to check the input (source DDL statements) and to produce the internal tables (to be used during code generation) according to the EBNF/WSC grammar. The SAPG produces the program SAP.

In conclusion, while the XPL system has some features not present in SAPG (e.g. automatic generation of lexical analysis tables, left recursivity, etc.), SAPG has the following advantages: it produces a specific ad-hoc syntax analysis program (SAP) for a particular language, as opposed to XPL's SKELETON which is a general purpose syntax analyzer; furthermore, with the subroutine call features, it facilitates encoding of source statements yielding greater flexibility to the compiler-writer and facilitates code-generation.

4.2 The Syntactic Analysis Program Generator

The Syntactic Analysis Program Generator, is a program which inputs a formal description of the syntax of a Language "L" (e.g., DDL), outputs code (in PL/1) to perform syntax analysis on statements in the Language "L" and coordinates the creation and maintenance of internal tables for use in code generation phases of compilation of the language "L", if required.

For example, assume that the SAPG is to be used to produce a SAP for the simple arithmetic expression used in ALGOL 60. The input to SAPG is the formal description of simple arithmetic expression in EBNF with subroutine calls, it is the following:

```
< SIMPLE_ARITH_EXP  >  ::= [ < ADD_OP > ] < TERM > [< ADD_OP >
                                                      < TERM >]*
< TERM > ::= < PRIMARY > [< MULT_OP > <  PRIMARY > ]*
< PRIMARY >  ::= < INTEGER >
               | <  NAME >
               |    (< SIMPLE_ARITH_EXP > )
```

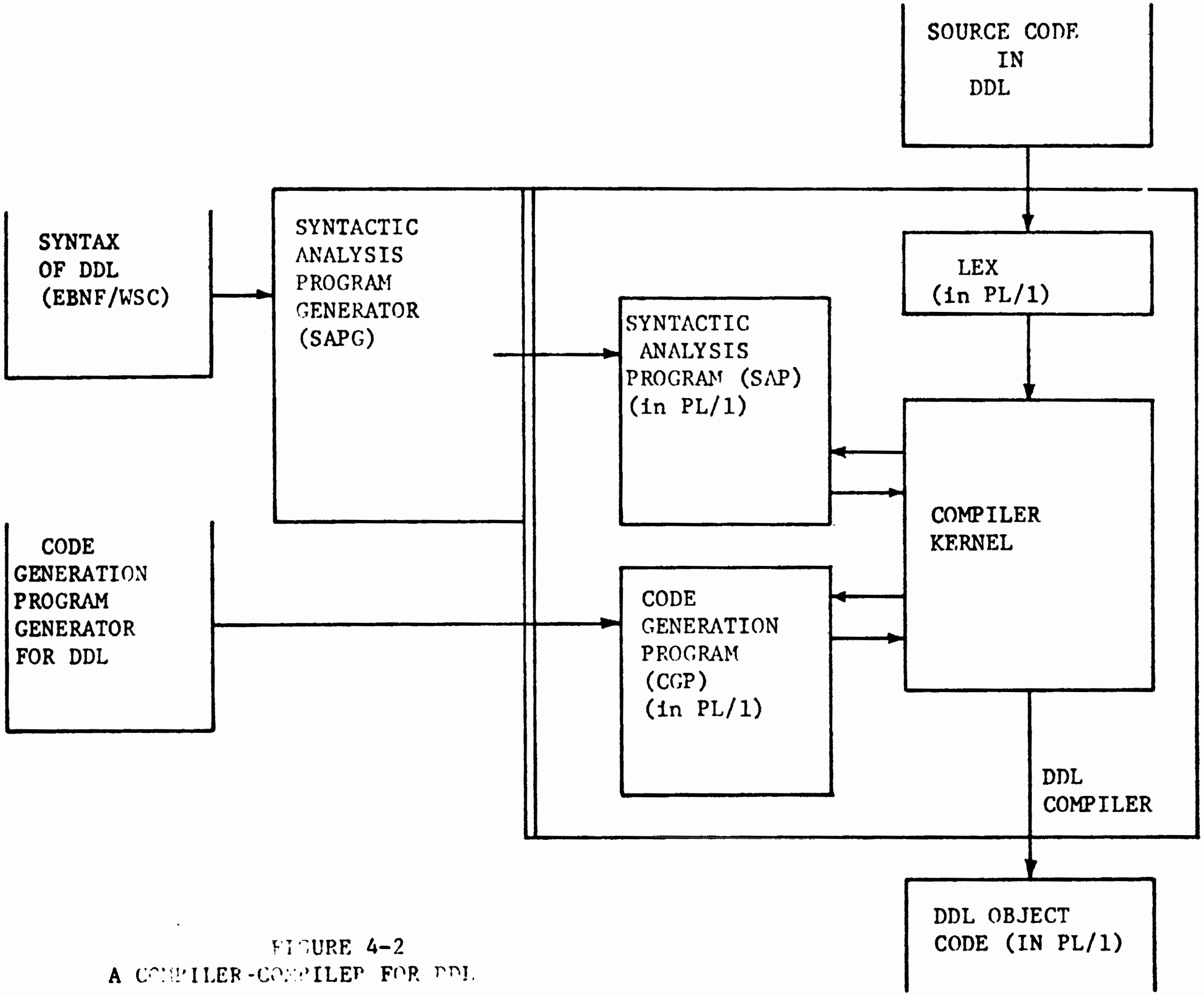


FIGURE 4-2
A COMPILER-COMPILER FOR DDL

< MULT_OP > ::= *|/

< ADD_OP > ::= +|-

< INTEGER >::= / INTREC / < NAME > ::= /NAMEREC/

Note: < INTEGER > represents an integer constant; < NAME > represents a variable name conforming to the ALGOL 60 naming conventions. Both INTREC and NAMEREC are recognizer routines which recognize valid integers and ALGOL variable names, respectively. They would have to be written, compiled, and incorporated into the SAP module in order for the SAP to run properly.

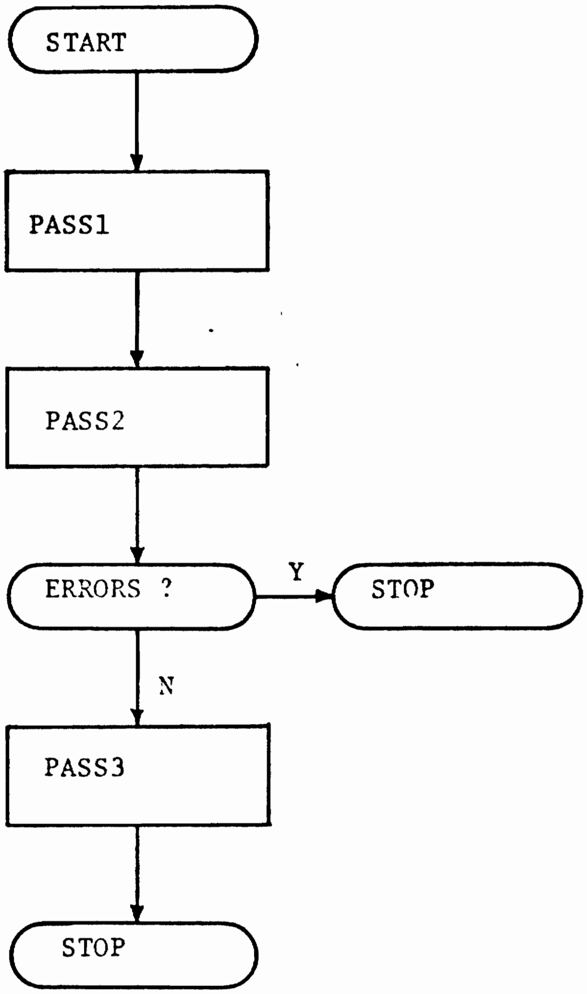
The output of SAPG, i.e., the SAP is presented in Figure 4-8 at the end of Section 4.3.

The SAPG is a three-pass compiler (Figure 4-3). These three passes, along with a description of the generated code and necessary supporting routines, are described in the following sections.

4.2.1 Lexical Analysis for EBNF/WSC

Lexical analysis with some exceptions, has been regarded as a minor part of the implementation of computer language translator (see[WAL 68]), and lexical analysis has been incorporated as an incidental part of Syntax Analysis Programs. Floyd [FLO 69] points out some good reasons for separating these functions, both logically and programmatically, they are:

1) A large portion of compiler time is consumed in lexical analysis, making it essential that this function be as efficient (fast) as possible. See Presser [PRE 69] and Conway [CON 63].



THE SAPG

FIGURE 4-3

2) Separation allows the development of systems for automatic syntactic and lexical analysis.

The last point was very important in this work since the existence of such systems allowed us to experiment with various lexical and syntactic schemes, without the burden of the immense programming times which would otherwise be required.

It is the job of the lexical analyzer to group together certain terminal characters into single syntactic entities called tokens. A token is a string of terminal symbols with which we associate a lexical structure consisting of a pair (token type, content). For a given language the number of token types will be presumed finite. Thus the lexical analyzer is a translator whose input is the string of symbols representing the source language (e.g. for the Processor SAPG the source language is EBNF/WSC and for the DDL compiler the source language is DDL), and whose output is a stream of tokens. This output forms the input to the Syntax Analysis.

Lexical analysis is important in compilations for several reasons. Perhaps most significant, replacing identifiers and constants in a program by single tokens makes the representation of a program much more convenient for later processing. Lexical analysis further reduces the length of the representation of the program by removing irrelevant blanks and comments from the representation of the source program. During subsequent stages of compilation, the compiler may make several passes over the internal representation of the program. Consequently, reducing the length of this representation by lexical analysis can reduce the overall compilation time.

The Lexical analyzer implemented here is based on the finite state machine concept [CON 63]. Each state of a "machine" correspond to a unique condition in the lexical processing of a character string. At each state a character is read, and the machine changes to a new state. At each transition, appropriate actions are taken based on the particular character read, a process controlled via a transition Matrix.

In the construction of such processor our goals were to:

a) Completely eliminate backup or re-reading,

b) Perform analysis of the language to detect lexical errors, and

c) Make it convenient and easy to use.

The lexical analyzer used in SAPG is called LEXEBNF and the lexical analyzer used by the DDL compiler is called LEX.

The implementation of LEXEBNF is given below, the implementation of LEX is given in Section 5.2 .

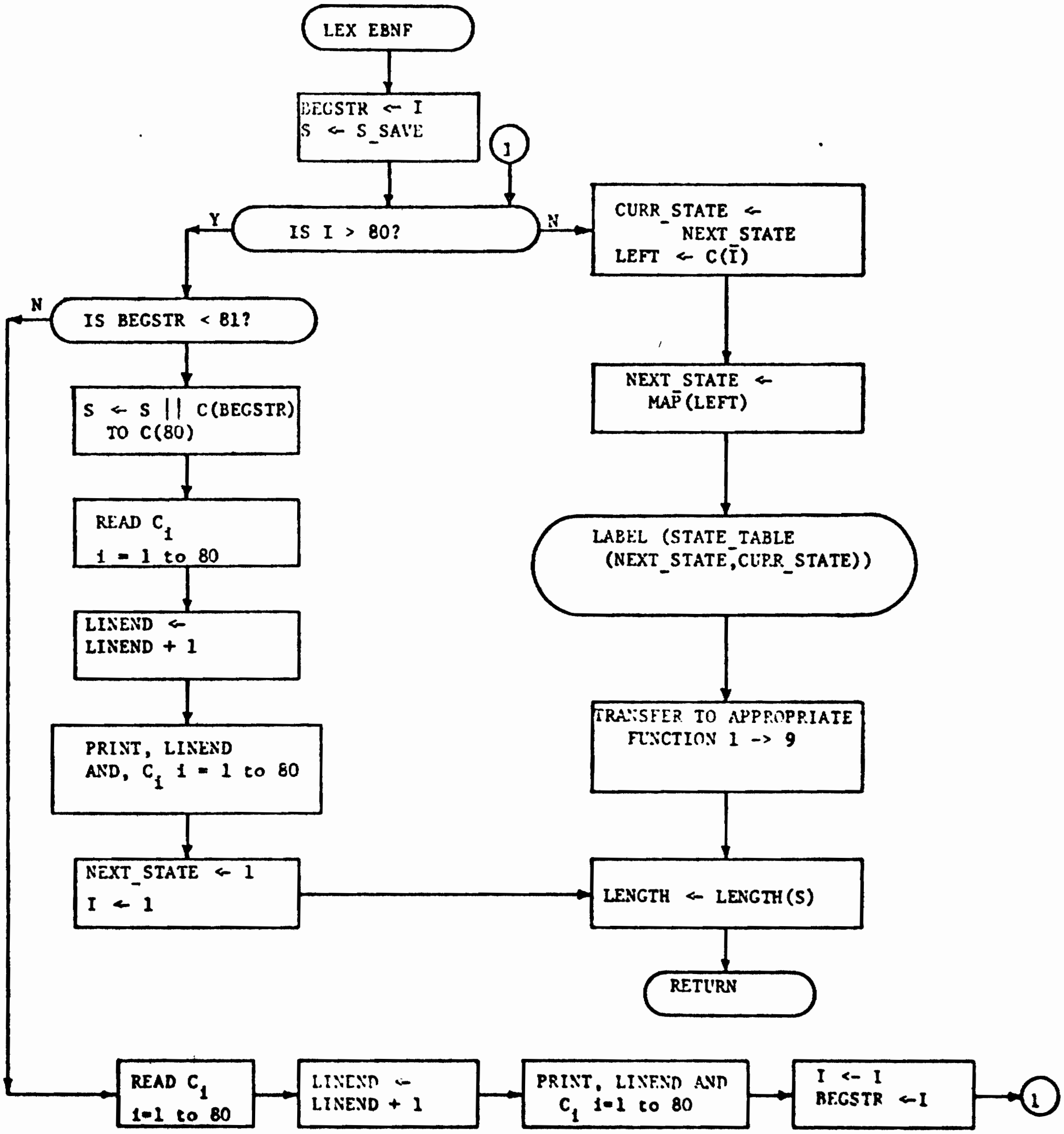
LEXEBNF is the lexical analyzer for EBNF with subroutine calls. (see Figure 4-4) This routine is called from pass-1 in SAPG and it returns the next token in the input stream as well as an indication of the beginning of a new production and the end of input. For LEXBNF a token has been defined as

a) metalinguistic symbol,

b) a non-terminal symbol - a string of characters beginning with " < " and ending with " > ",

c) a separator - a character such as ",", "(", ")", ";", and

d) a terminal symbol - any string of characters which does



LEXICAL ANALYSIS FOR EBNF/WSC
(LEXBNF)

FIGURE 4-4

not include a metalinguistic symbol, a non-terminal symbol, a separator, or embedded blanks. LEXEBNF ignores all blanks.

LEXEBNF contains two tables which it uses in lexical analysis. The first of these is MAP. This array partitions the character set into classes, including the class of invalid characters. The separators mentioned in the previous paragraph are indicated in this table. As the members of this class change due to differing languages, this table must be updated to reflect the differences. The table is indexed by the internal character code representation, in this case Extended Binary Coded Decimal Interchange Code (EBCDIC). The position in the table corresponding to a character contains the number of the partition class of which it is a member. If a character is invalid the message printed will be "INVALID CHARACTER OR COMBINATION OF CHARACTERS IN COLUMNS i, j OF LINE NUMBER k. PRODUCTION DISCARDED." Column "j" contains the offending character and "k" is the line number which appears immediately to the left of the input record in which the error occurred in the source listing. (Note: LEXEBNF assumes fixed length records of 80 bytes for its source program input stream.). The entire production in which such an error occurs is eliminated by the routine DISCARD.

The second table used by LEXEBNF is STATE TABLE. This array is indexed by the partition class numbers of the current input character and the immediately preceding one. This doubly-indexed array determines the function performed by LEXEBNF, such as concatenating the current input character to the string obtained so far by previous iterations or returning the current input characters. In otherwords, in EBNF (as well as in DDL) two adjacent characters suffice to determine the appropriate action to be taken in order to produce the token. If the lexical functions are changed, this table may also have to be changed.

If an illegal combination of partition class numbers occurs, the message which occurs for an illegal character is printed. In this case "i" and "j" refer to the card columns in which the two characters forming the offending combination may be found.

LEXEBNF inputs source records as necessary to fulfill the requests of SAPG for a new token. The line number count is incremented after each source record is read. Every error encountered by LEXEBNF causes an error count to be incremented. This error count is used by SAPG.

The Transitions Matrix (STATE-TABLE) For LEXBNF'S:

	0	1	2	3	4	5	6	7	8	9	10	11	12
0	1	2	4	1	1	4	4	4	4	4	4	4	8
1	1	3	1	4	5	1	1	9	1	5	5	1	8
2	1	8	8	8	7	8	8	8	8	8	7	8	8
3	4	2	4	8	4	4	4	4	4	4	4	4	8
4	4	2	4	8	8	4	4	4	8	4	8	8	8
5	4	2	4	8	8	8	7	4	8	4	8	4	8
6	8	8	8	8	8	7	8	8	8	8	8	8	8
7	6	2	8	8	8	8	8	6	8	8	8	8	8
8	4	2	4	8	7	4	4	4	1	8	8	8	8
9	4	2	4	8	8	4	4	4	4	4	4	4	8
10	4	2	4	8	8	4	4	4	8	4	8	8	8
11	1	2	4	8	8	4	4	4	8	4	8	8	8
12	8	8	8	8	8	8	8	8	8	8	8	8	8

The character mapping table is as follows:

CLASS	CHARACTER
0	ABC...z 01....9_$
1	b (space)
2	<
3	>
4	=

CLASS	CHARACTER
5	"
6	multipunch $\overset{o}{12}$
7	/
8	:
9	+ # ' ( ) ; * | .
10	-
11	,
12	"any other character non-mentioned above"

The lexical functions are determined by the entries in the transition matrix given above. These entries are used to index an array of labels. The contents of this array is as follows:

LABEL (1) = F1

LABEL (2) = F2

LABEL (3) = F3

LABEL (4) = F4

LABEL (5) = F5

LABEL (6) = F6

LABEL (7) = F7

LABEL (8) = F8

LABEL (9) = F9

The lexical functions performed are:

```
F1:  I=I+1;

F2:  S=S || SUBSTR (D, BEGSTR,I-BEGSTR); I=I+1
     S-SAVE=''; GO TO RETURN,

F3:  DO I=I+1 TO 80;
     IF C(I) ¬= '' THEN DO; BEGSTR=I; GO TO SCAN; END; END;
     I=0; GO TO F3;
```

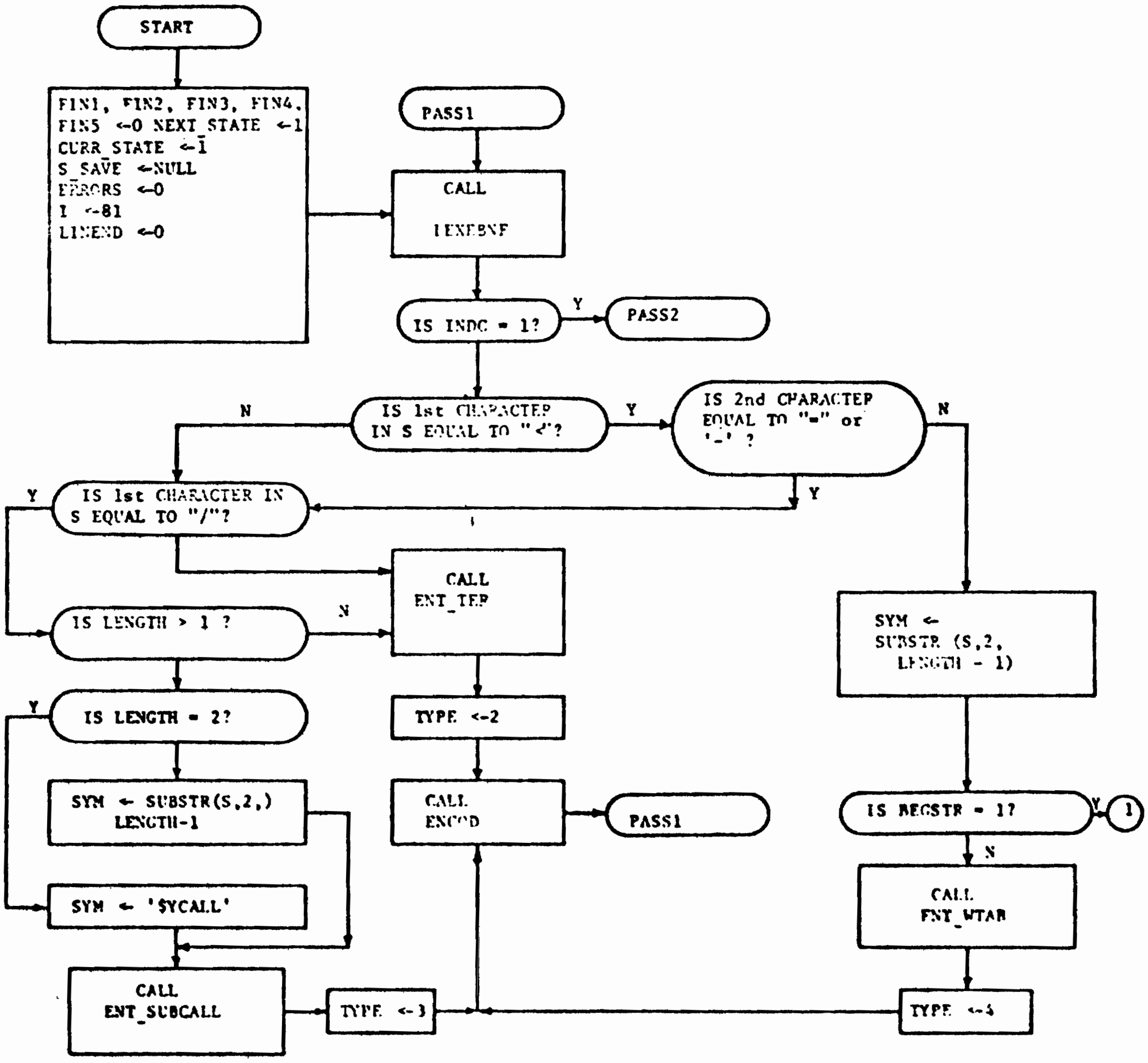
```
F4:  S=S || SUBSTR(D,BEGSTR,I-BEGSTR);
     I=I+1;
     S-SAVE=LEFT;
     GØ TO RETURN;

F5:  S=LEFT; LENGTH,NEXT_STATE=1; I=I+1; S-SAVE=''; RETURN;

F6:  J=INDEX (SUBSTR (D,I,8),'/');
     IF J=0 THEN DO; CALL DISCARD; ERRORS=ERRORS+1;
     FIN5=FIN5-1; FIN1=FIN1=1;
     IF - INDC THEN GO TO SCAN; ELSE RETURN; END;
     ELSE DO;
     S=S || SUBSTR (D,I,J); NEXT_STATE=1; S-SAVE= '';
     I=I+J; GO TO RETURN; END;

F7:  I=I+1; S=S || SUBSTR(D,BEGSTR,I-BEGSTR);
     S-SAVE= ''; NEXT_STATE=1; GO TO RETURN;

SE:  IF BEGSTR -= 1 THEN DO; FIN1=FIN1-1; FIN5=FIN5-1; END;
"    ELSE IF SUBSTR (D,1,1) = ' < ' THEN DO; FIN1=FIN1-1;
F8:  FIN5=FIN5-1; END;
     CALL DISCARD; ERRORS=ERRORS+1;
     IF - INDC THEN GO TO SCAN; ELSE RETURN;

F9:  S=LEFT; BEGSTR,I=I+1; GO TO SCAN;
```

RETURN is a label which begins the following code:

```
     RETURN:  BEGIN;
     DCL LENGTH BUILTIN;
     J=LENGTH(S); END;
     LENGTH=J; RETURN
```

(Note: the code which generates error messages and listings has been omitted from the above code. SCAN is the label which begins the loop which performs the scanning of the individual characters in the input.)

THE SYNTACTIC ANALYSIS PROGRAM GENERATOR



PASS1 OF SAPG

FIGURE 4-5

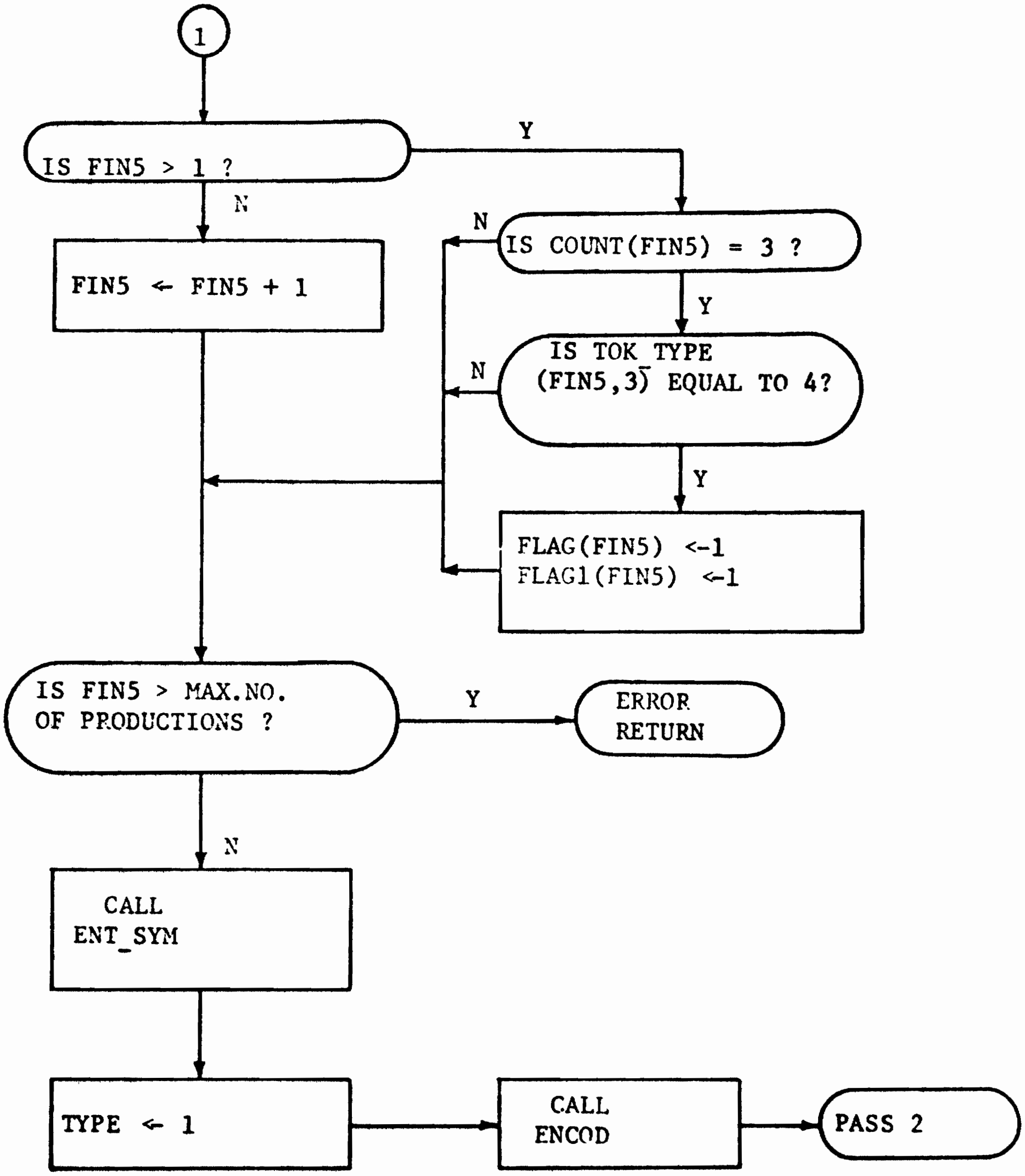


FIGURE 4-5 (CONT)
PASS1 OF SAPG

Pass 1 of the SAPG (see Figure 4-5) performs the lexical analysis of the EBNF/WSC source statements (productions) by calling the routine LEXEBNF and encodes such productions in a table called the "Encoded Table." (This table is one of the five internal tables which the SAPG maintains).

The Encoded Table is organized by production. Tokens (lexical units) are placed in the encoded table by calling the routine ENCOD (see Figure 4-5a) as they are encountered. The tokens are divided into the following classes:

CLASS	TYPE
a) Terminal symbols	2
b) Subroutine calls	3
c) non-terminal symbols	1 and 4

The entry for each lexical unit in the Encoded Table consist of two parts:

TYPE	POINTER

There is a table corresponding to each type, and the pointer is used to index the appropriate table. This is done to conserve space in the tables, since the same entry in a table may be referred to by different entries in the Encoded Table. Non-terminals appearing on the left side of a production are typed as 1 and placed in the Symbol Table by the routine ENT_SYM (see Figure 4-5c). Non-terminals appearing on the right side of the production symbol are typed as 4 and placed in the Work Table by the routine ENT_WTAB (see Figure 4-5b). Entries in the Work Table are references to other productions in the language specification

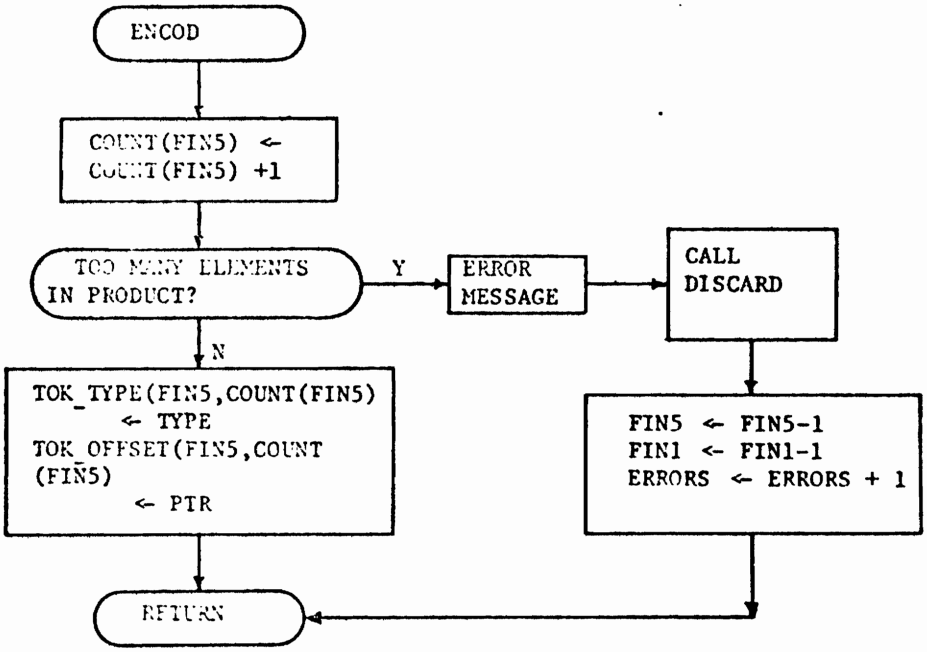


FIGURE 4-5A
ENCOD

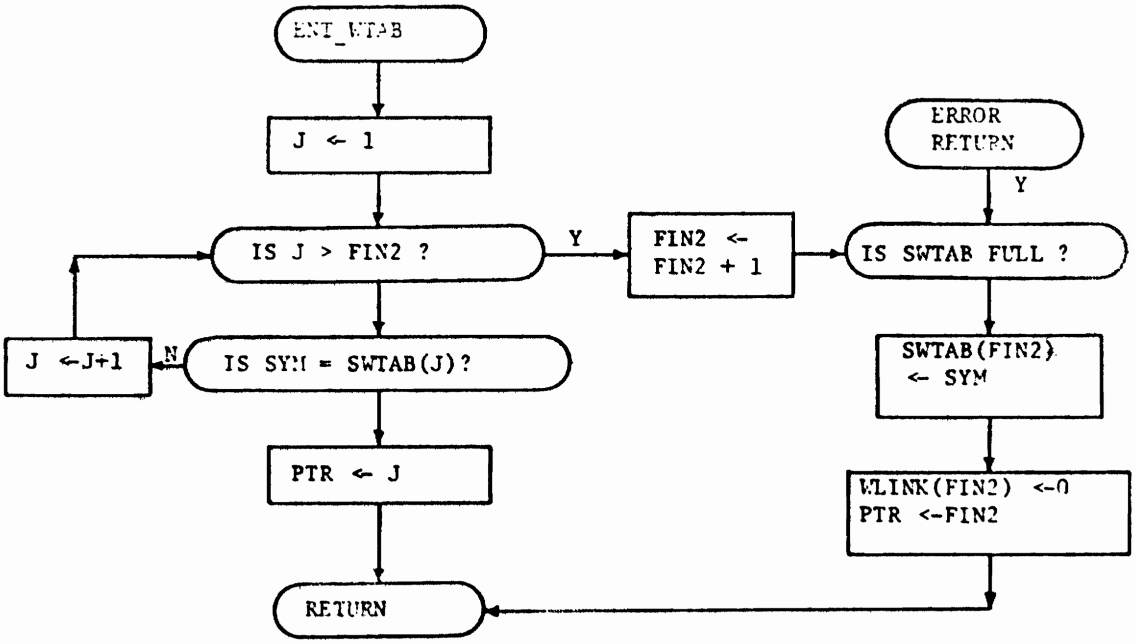


FIGURE 4-5B
ENT_WTAB

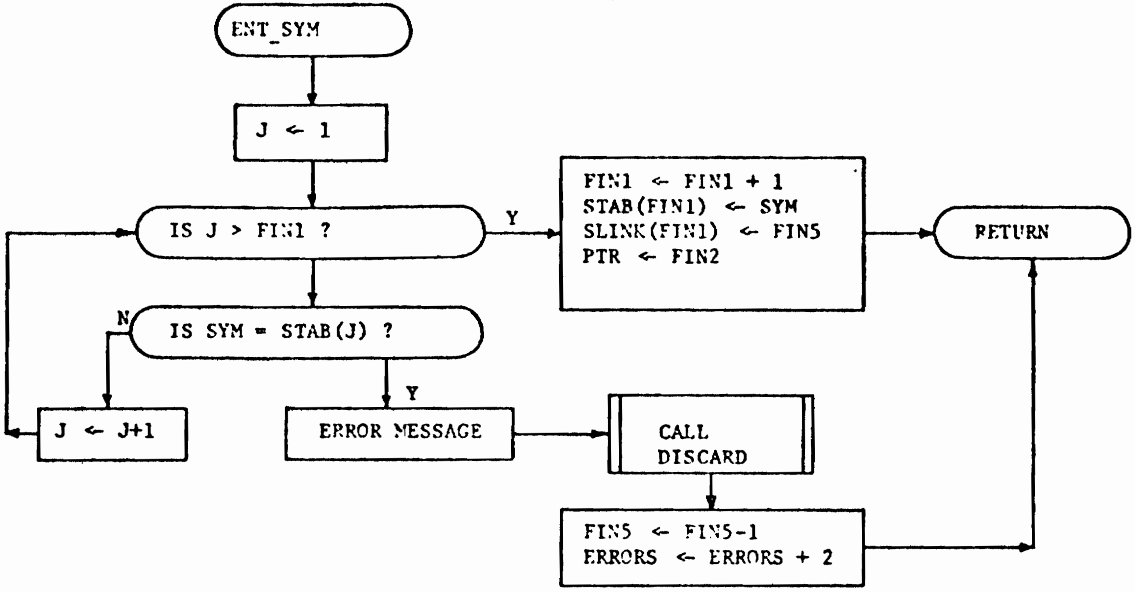


FIGURE 4-5C

ENT_SYM

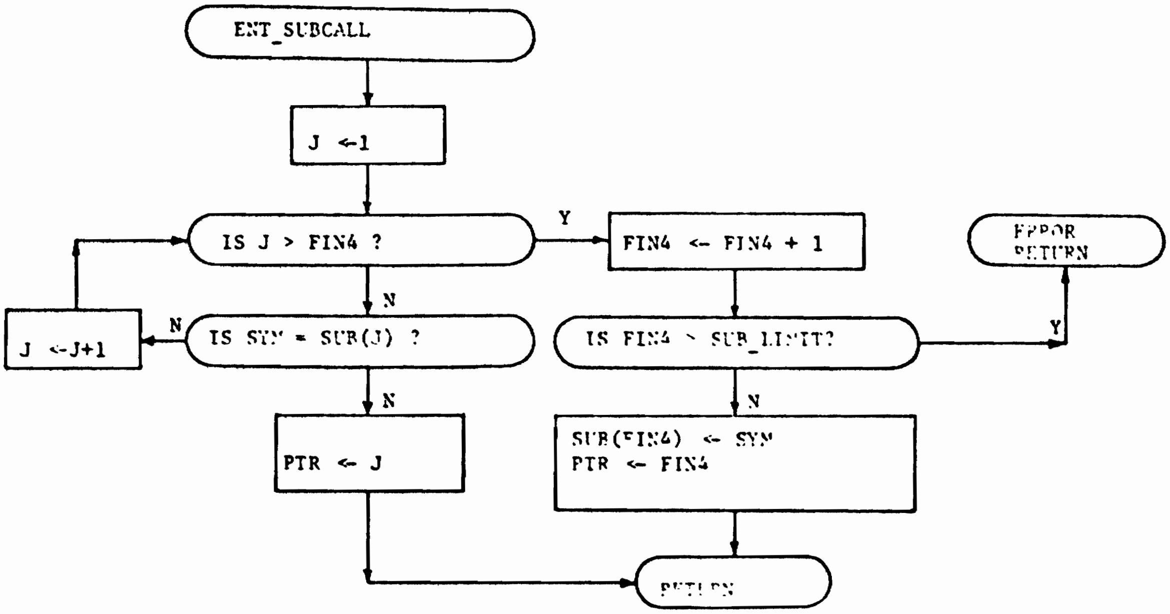


FIGURE 4-5D

ENT_SUBCALL

which must be resolved before the code generation phase (Pass 3) of the SAPG can proceed. Subroutine calls and terminal symbols are typed as indicated above and placed in separate tables by the routines ENT_SUBCALL (see Figure 4-5d) and ENT_TER, (see Figure 4-5e) respectively. (Note: in the current SAPG these tables are implemented as arrays of size 100. Each production is limited to 75 elements.

Each of the entries appearing in the tables, except for the Encoded Table, is unique. The tables are organized sequentially, so that a simple sequential search will determine if an entry already exists in the table. If it does not, it is placed at the end of the appropriate table.

The entries in the Symbol Table are the syntactic units of the language being specified. Entries in the Symbol Table are always the first symbol in a production. LEXEBNF recognizes them as such because the symbol defining a new production must always begin in column 1 of the input record. If a symbol already appears in the Symbol Table, an ambiguity is present in the language, and the entire production is rejected by the routine DISCARD (see Figure 4-5f). (Rejected productions are noted, but not analyzed, during Pass 2 of the SAPG). The message "MULTIPLY DEFINED PRODUCTION < symbol >. PRODUCTION DISCARDED." will appear on the listing when this occurs, where "symbol" is the name of

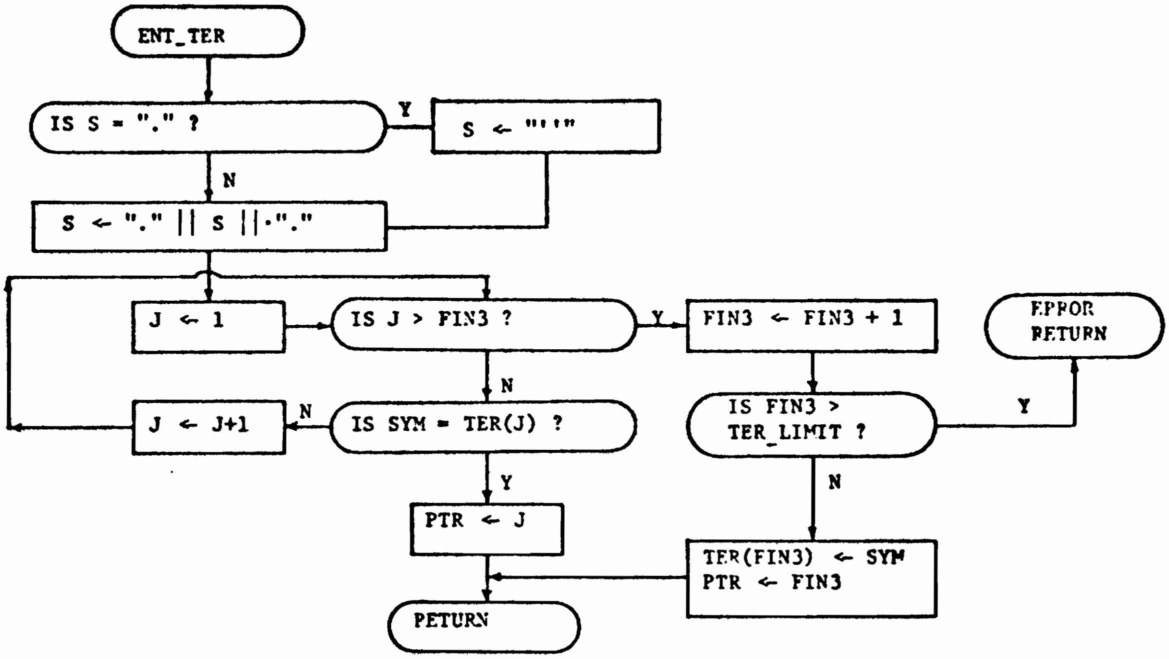


FIGURE 4-5E
ENT_TER

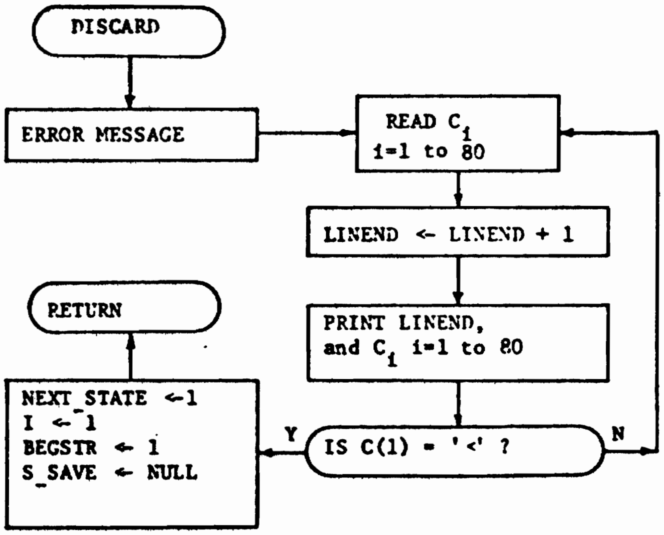


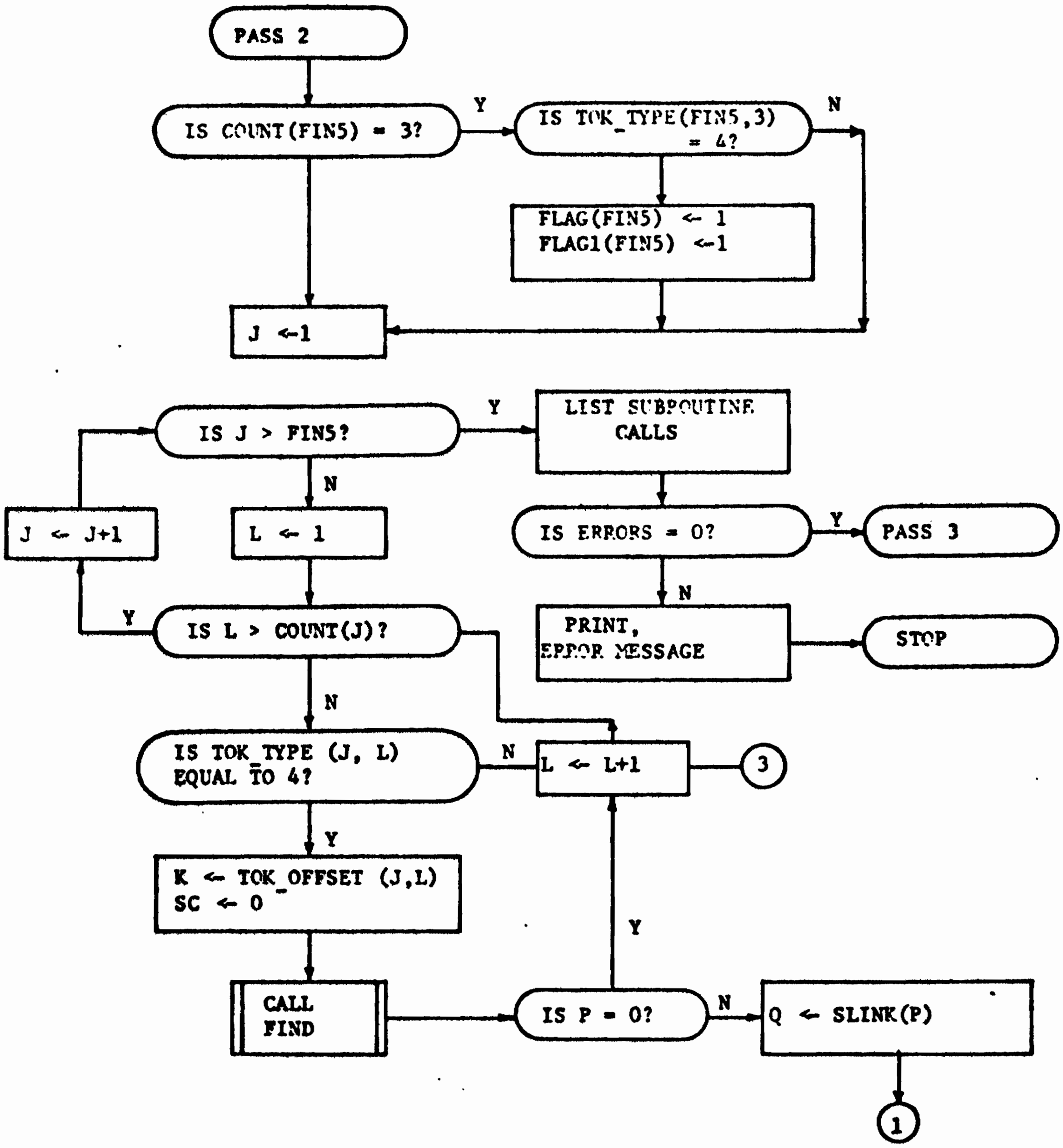
FIGURE 4-5F
DISCARD

the offending production.

Each production is also checked to see if it is "singular". A singular production is one of the form < production1 > ::= < production2 >. This type of production makes < production1 > and < production2 > equivalent. A reference to < production2 > may be substituted for a reference to < production1 > in any other production which makes a reference to the latter. Doing so eliminates unnecessary intermediate levels of the language specification and wasted time during the execution of the SAP, since each EBNF production is translated into a PL/1 procedure (see Section 4.2.3). Singular productions are flagged for PASS 2 of the SAPG so that all possible intermediate levels may be removed. Note that productions of the form < production3 > ::= TERMINAL-SYMBOL and < production4 > ::= /SUBCALL/ are not treated as singular productions, even though no code will be generated for the latter (see Section 5.3.2.2).

4.2.2 Pass 2 of the SAPG (Syntax Analysis of EBNF)

Pass 2 (see Figure 4-6) of the SAPG scans the Encoded Table output by Pass 1 to resolve all symbolic references that were placed in the Work Table. Each entry in the Work Table has a link to the Symbol Table. This link is initially set to zero. Scanning begins with the first production in the Encoded Table. Each reference to an entry in the Work Table (indicated by a type 4 entry) causes Pass 2 to call on the routine FIND (see Figure 4-6a). If the link to the Symbol Table is zero, the reference has not yet been resolved. The Symbol Table is searched for the symbol



PASS2 OF THE SAPG
FIGURE 4-6

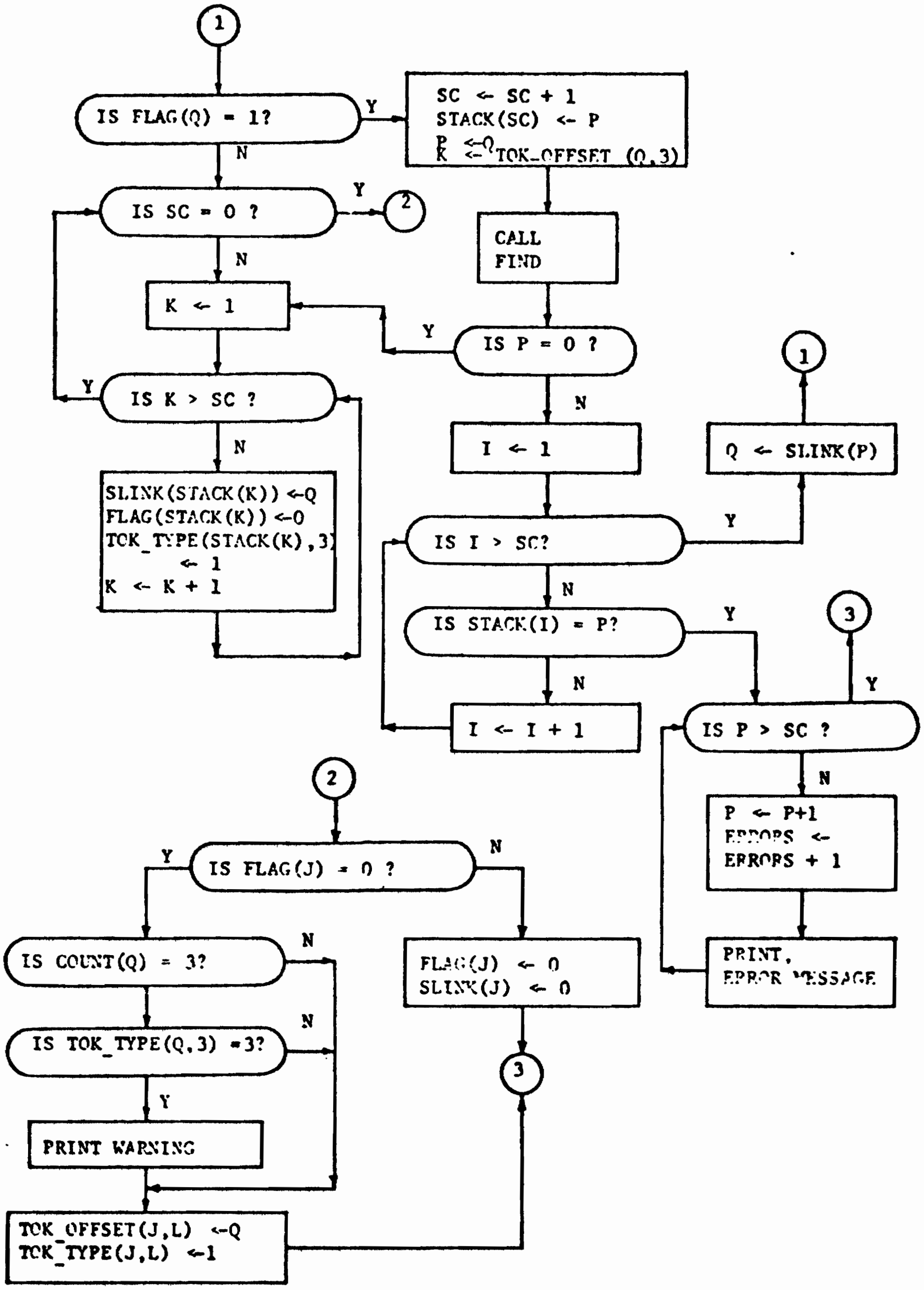


FIGURE 4-6 (CONT)

PASS2 OF THE SAPG

in the Work Table. If a match is found, FIND returns a pointer to the appropriate entry in the Symbol Table. If a match is not found, the symbol is undefined. The message "< symbol > IN PRODUCTION < production > IS UNDEFINED." is printed, where "symbol" is the Work Table entry which is unmatched in the Symbol Table and "production" is the production currently being scanned. In this latter case a pointer of zero is returned to Pass 2.

If the production referenced is defined and is singular, (indicated by a flag set during Pass 1), a pointer to the referenced production is placed on a simulated stack and an attempt is made to resolve the reference made by the singular production using the resolution process just described. The process terminates either by encountering an undefined symbol or by a reference to a non-singular production. (During the stacking process, a check is made to see if the production just referenced is already on the stack. If it is a condition of circular definition exists in the language. The name of every production on the stack from the point at which the match is found to the top of the stack is printed in the following error message: "PRODUCTION < name > IS A MEMBER OF A RING OF CIRCULAR, SINGULAR PRODUCTIONS.". A Symbol Table pointer of zero is returned to Pass 2.)

Every entry in the Symbol Table has a link to the Encoded Table pointing to the production which defines it. Upon termination of the resolution process, one of the singularity flags of each singular production on the stack (there are two such flags associated with each production) is reset, the link to the Encoded Table is set to the value returned by FIND (see Figure 4-6a), and the type of the symbol on the right side of the production is set to 1.

The singularity flag of the original production which started the resolution process is checked. If the flag is set, the production is treated in the same manner as the singular productions which were on the stack. If the production is not singular, the pointer of the symbol currently being scanned is set to the value returned by FIND and the type is set to 1. Scanning proceeds with the next type 4 symbol in the current production, if there is such a symbol. In this case, if the production at the lowest level of reference is of the form < production > ::= /subcall/, where "subcall" is a subroutine call, then the following warning is printed on the data set referenced by the ddname WARNING: "AN ENTRY ON THE ERROR STACK MAY BE REQUIRED FOR < symbol > IN PRODUCTION < current production >.", where "symbol" is the symbolic reference just resolved and "current production" is the production in the Encoded Table currently being scanned. (A discussion of the Error Stack is given in Section 5.3.1). Note that resetting the singularity flag of the singular productions prevents a future erroneous attempt at resolution should they be referenced again, and setting the type to 1 prevents a future erroneous resolution attempt

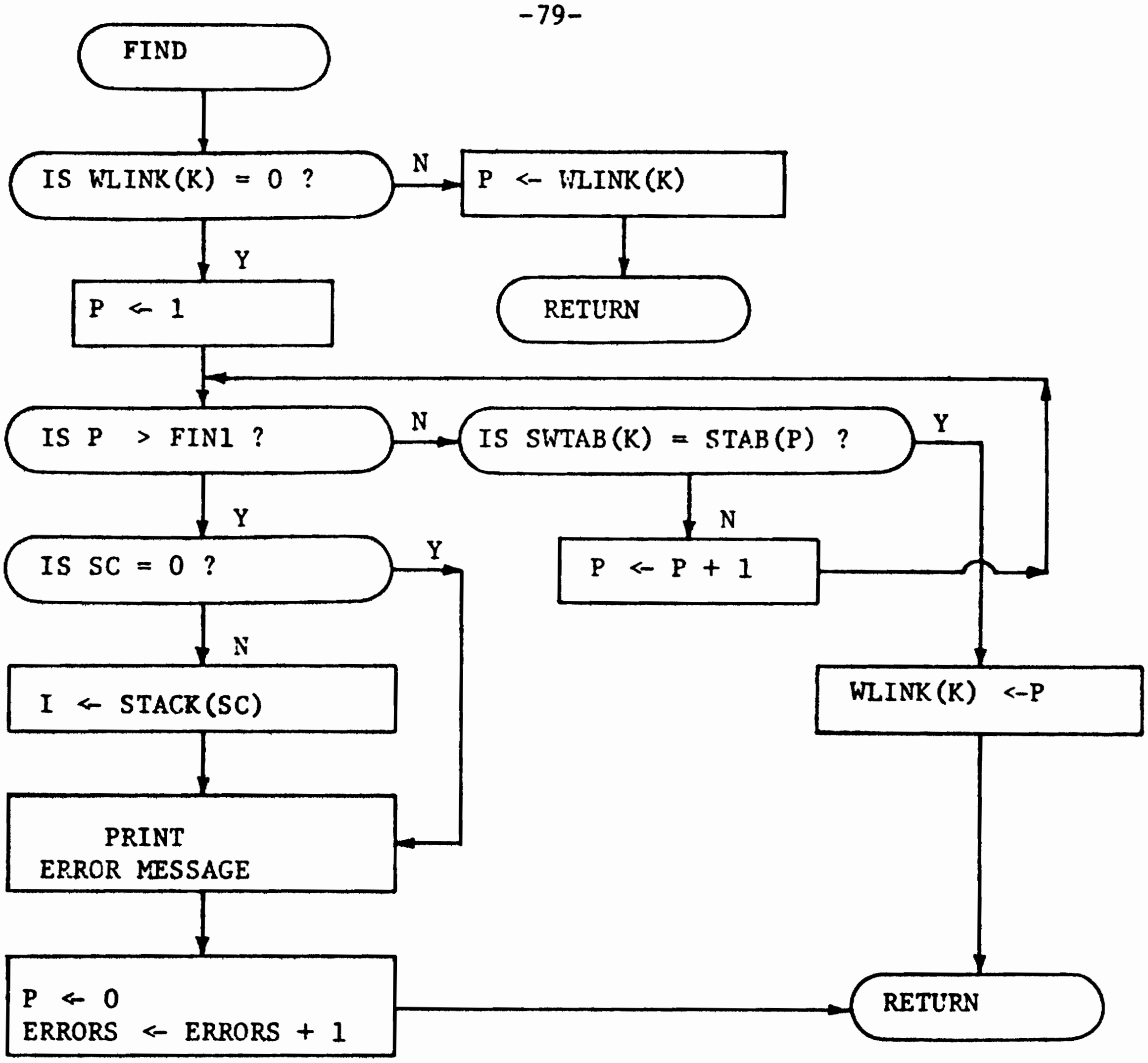


FIGURE 4-6A
FIND

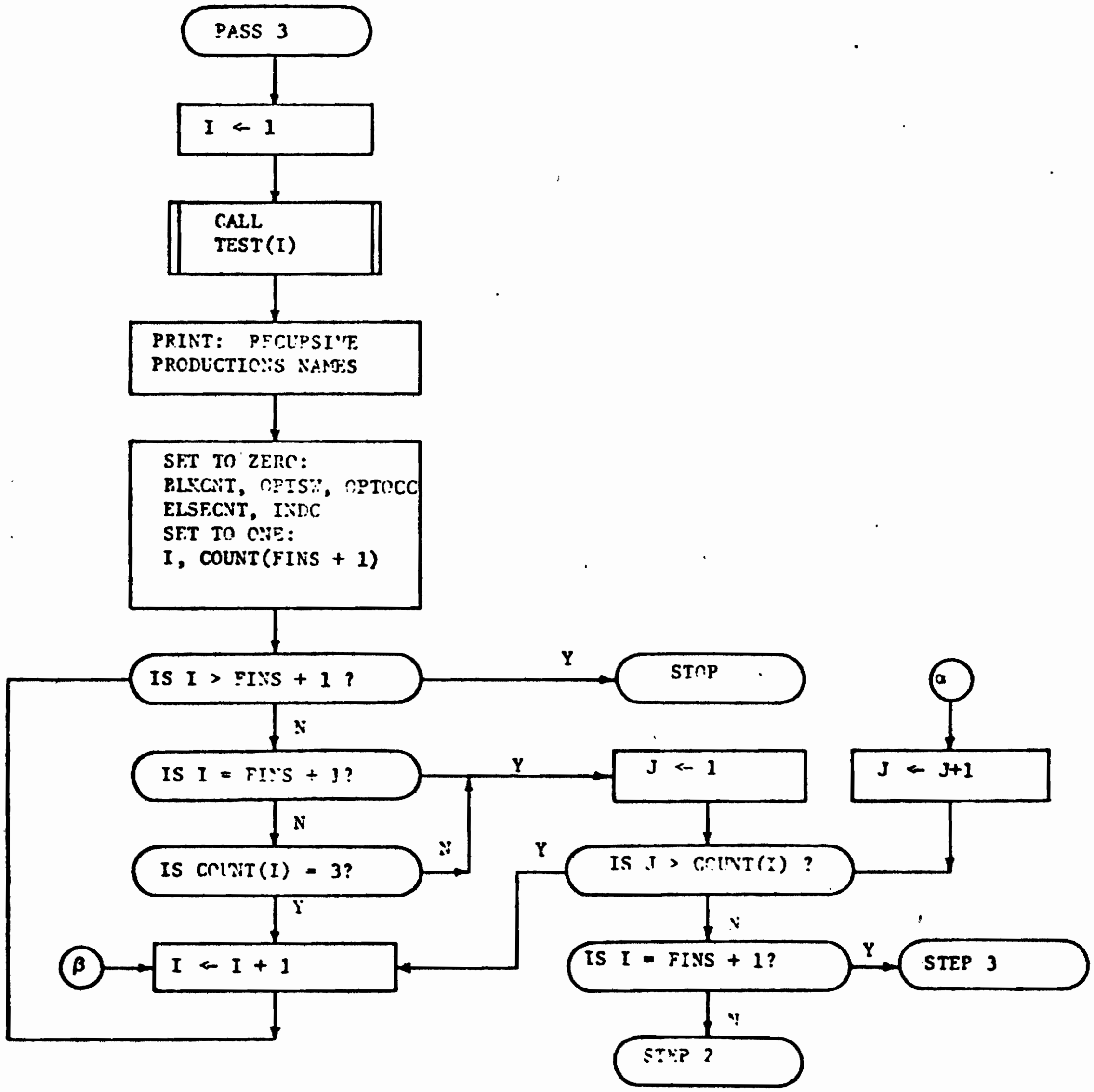
should they be encountered later during the sequential scan of the Encoded Table.

If no errors are encountered by the first two passes, processing continues with code generation phase, Pass 3, of the SAPG. Otherwise, processing terminates with an error message indicating the number of errors detected and returning a non-zero code to the operating system.

4.2.3 Pass 3 of the SAPG (Code Generation)

Pass 3 of the SAPG is the code generation phase; it outputs the PL/I-coded SAP. Pass 3 operates under the assumption of an error-free environment. Thus, it is not called unless the first two passes have detected no errors. The SAP depends heavily upon the PROCEDURE- and DO-blocks and IF-THEN-ELSE-clause features of PL/I. Each production is encoded as a PROCEDURE which returns a bit string of length 1. The returned value is 0 if recognition fails on the first syntactic unit of the production. Otherwise, the returned value is 1, with an error message being printed if an error is detected after the first syntactic unit is recognized (see Section 5.3.1).

The scan for syntactic units is accomplished by nested IF-THEN-ELSE clauses using DO-groups. (Optionality groups also use the GO TO statement to cause the SAP to scan for the first syntactic item of the group again). The exclusive nature of the EBNF descriptions is implemented by the exclusive nature of the THEN-ELSE clauses of the PL/I IF statement.



STEP1 OF PASS 3 OF SAPG
FIGURE 4-7A

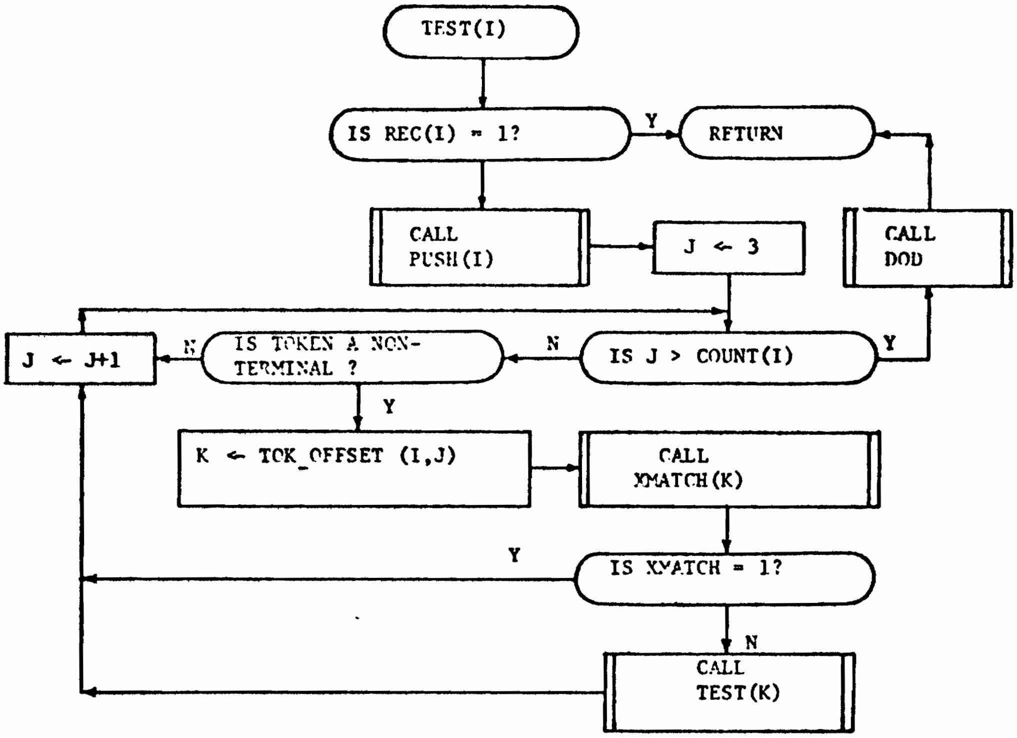


FIGURE 4-7A-1 - TEST(I)

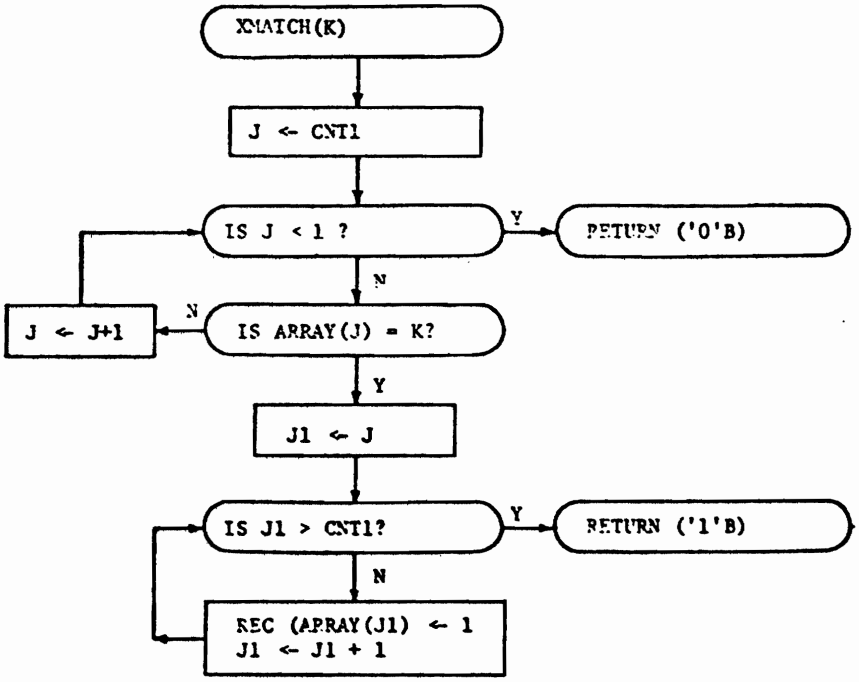


FIGURE 4-7A-1
XMATCH(K)

The following subsections describe the algorithm used for converting the Encoded Table into PL/I code.

4.2.3.1 Step 1 (see Figure 4-7a)

This step is the initialization step for Pass 3. Storage which contains the following information is allocated and/or initialized:

(1) A count of the number of syntactic units to be recognized within a given alternative or optionality group (BLKCNT);

(2) An indicator for recording the occurrence of the optionality brackets "[" and "]" (OPTSW);

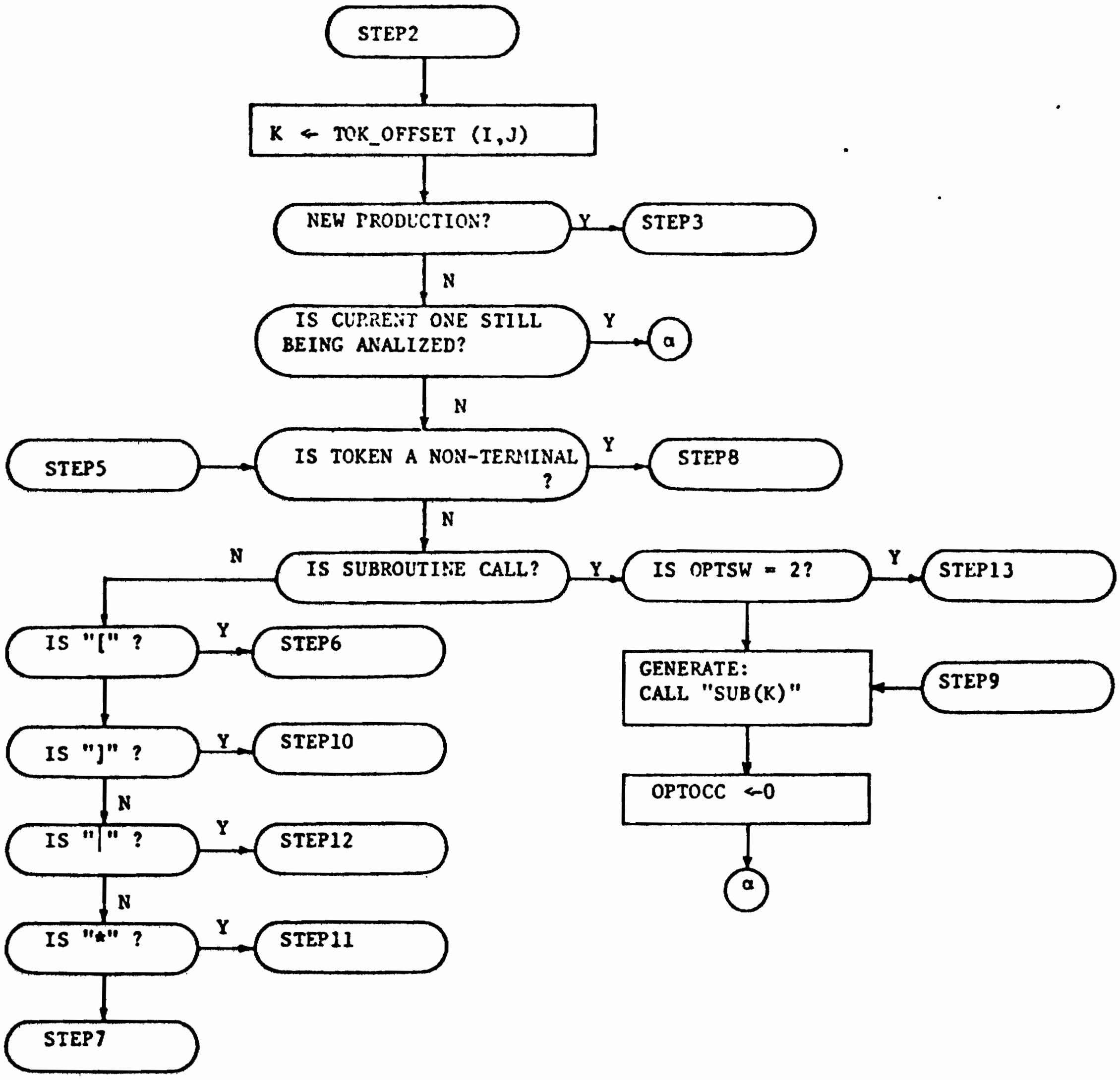
(3) An indicator for recording the immediate occurrence of the optionality bracket "[" (OPTOCC);

(4) A count of the number of alternatives in the current production (ELSECNT);

(5) A table of indicators for the type of each syntactic unit in the alternative or optionality group, i.e., terminal symbol or recognizer routine (see Section 5.3.2.2) or neither of these two.

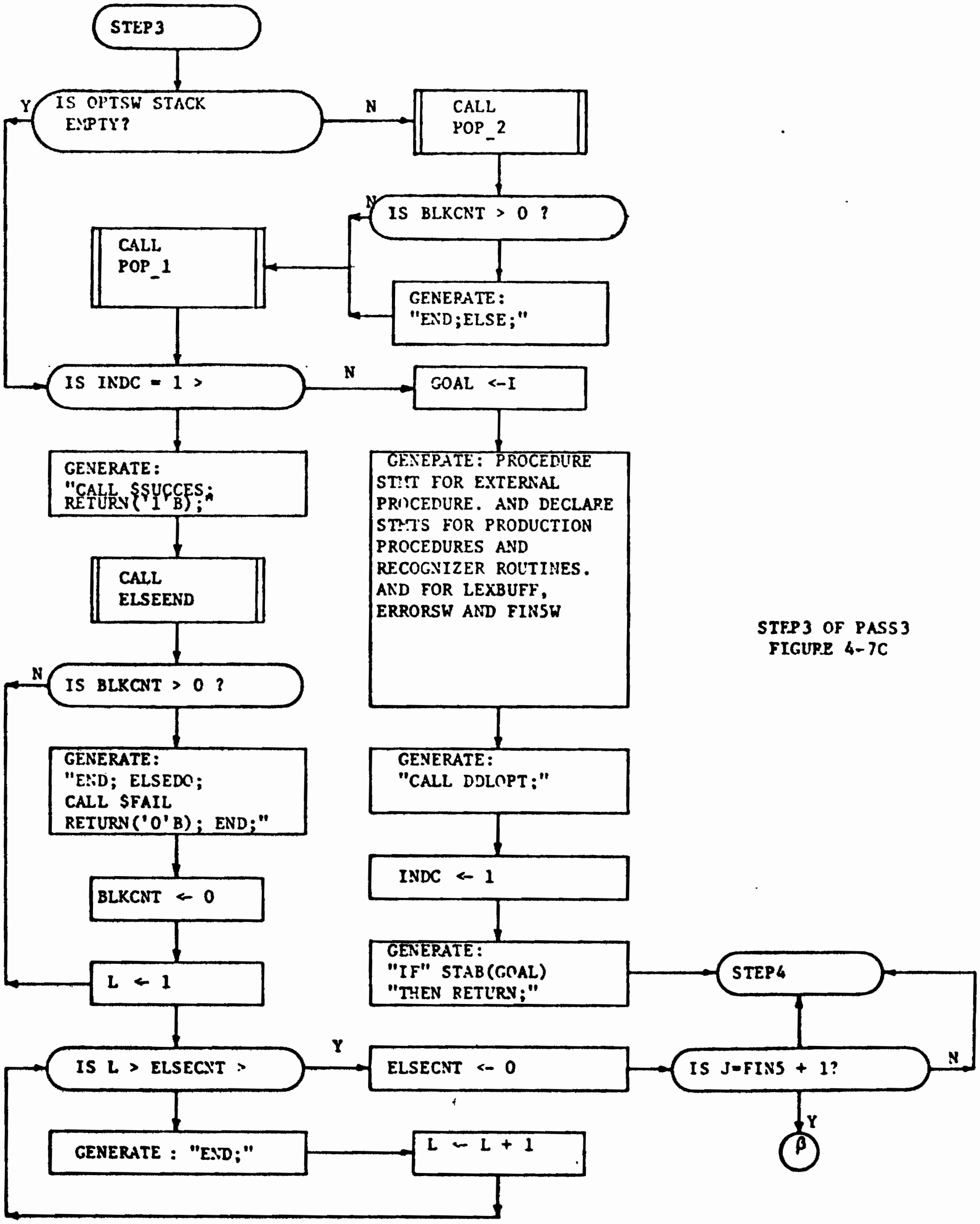
4.2.3.2 Step 2 (see Figure 4-7B)

Step 2 gets the next Encoded Table entry and determines if a new production is being started or if the current one is still being analyzed. In the former case the algorithm proceeds with Step 3. Otherwise, control is transferred to Step 5 for a determination of the type of the Encoded Table unit currently being



STEP2 & STEP5 & STEP9 OF PASS3

FIGURE 4-7B


STEP3
IS OPTSW STACK EMPTY?
Y
N
CALL POP_2
IS BLKCNT > 0 ?
N
CALL POP_1
GENERATE: "END;ELSE;"
IS INDC = 1 >
N
GOAL <-I
GENERATE: "CALL $SUCCES; RETURN('1'B);"
GENERATE: PROCEDURE STMT FOR EXTERNAL PROCEDURE. AND DECLARE STMTS FOR PRODUCTION PROCEDURES AND RECOGNIZER ROUTINES. AND FOR LEXBUFF, ERRORSW AND FINSW
CALL ELSEEND
IS BLKCNT > 0 ?
N
GENERATE: "END; ELSEDO; CALL $FAIL RETURN('0'B); END;"
GENERATE: "CALL DDLOPT;"
INDC <- 1
BLKCNT <- 0
GENERATE: "IF" STAB(GOAL) "THEN RETURN;"
STEP4
L <- 1
IS L > ELSECNT >
Y
ELSECNT <- 0
IS J=FINS + 1?
N
Y
β
GENERATE : "END;"
L <- L + 1

STEP3 OF PASS3
FIGURE 4-7C

scanned. Productions which are singular or involve only a single subroutine call are skipped, i.e., no code is generated for them.

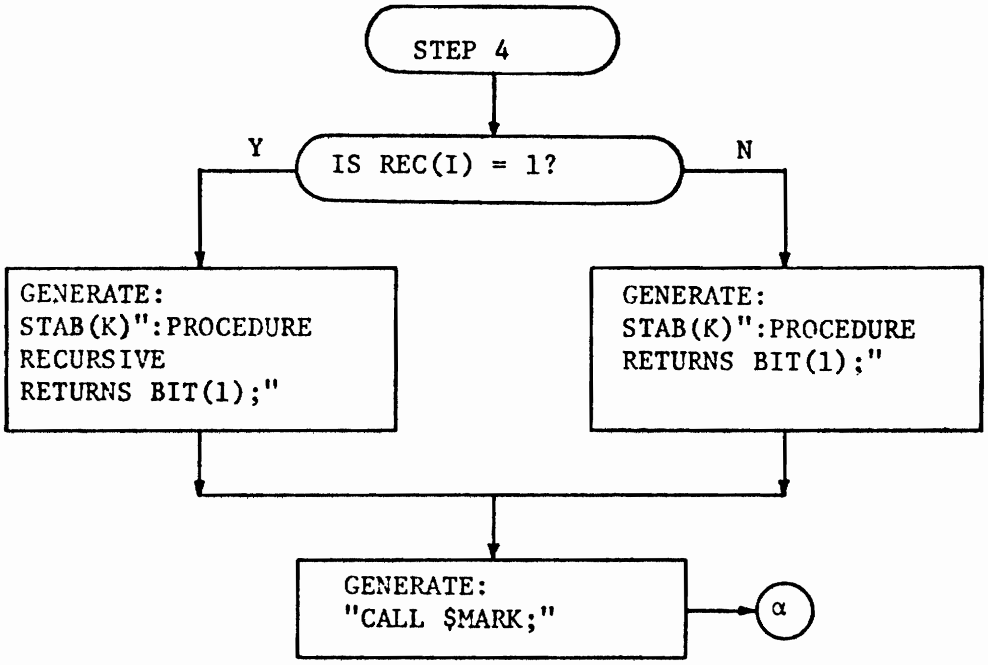
4.2.3.3 Step 3 (see Figure 4-7c)

Step 3 first checks to see if the last unit of the previous production was "]". If so, the code described in Step 12 (see Section 4.2.3.12) is generated. Otherwise, a check is made to see if any code has been generated, i.e., if the current production is the first non-singular one. Before code for the first production can be output, preliminary PL/I declarations must be made. Each production procedure and recognizer routine (see Section 4.2.3.12) is declared to return a bit string value of length 1. This is to prevent the PL/I compiler from assuming default attributes for the procedure name when the procedure is invoked. The global variables LEXBUFF, ERRORSW, and FINSW are then declared. (LEXBUFF contains the lexical unit currently being scanned. Its maximum length is 31 characters. ERRORSW is set when an error is detected and must be reset by subroutine call before the next statement is analyzed. FINSW is set upon the detection of the end-of-program statement). Finally, code to call the SAP initialization routine, DDLOPT, and invoke the goal procedure are generated. (The goal procedure is the procedure generated for the first non-singular production). The algorithm proceeds with Step 4.

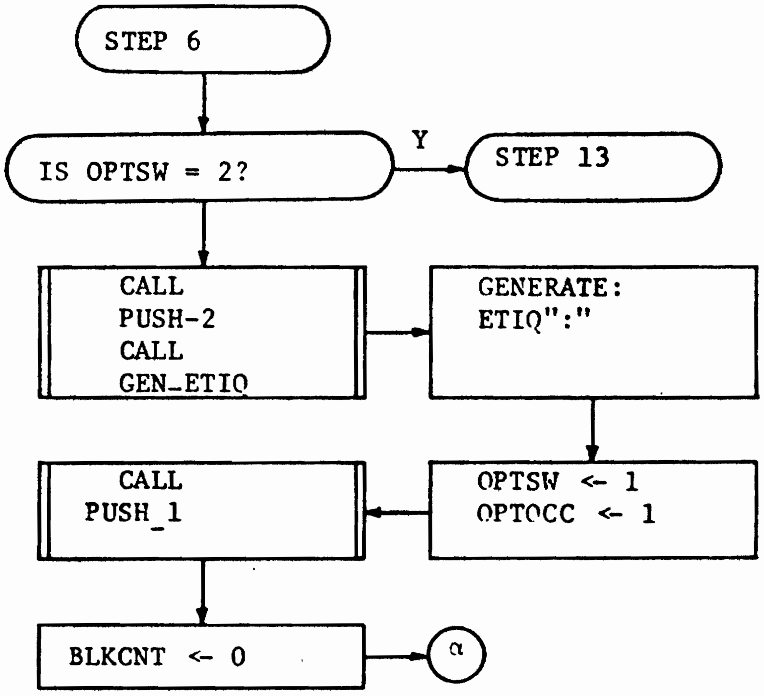
If the code described above has already been generated, i.e., the current production is not the goal production, code to end successful execution of the procedure ("CALL $SUCCES; RETURN('1'B);") is generated, and every generated code block which was opened is closed. (The routine $SUCCES is described in Section 5.3.1). Each opened DO-block within the last alternative, except the first, is closed with the following code: "END; ELSE DO; CALL end-routine; RETURN('1'B); END;", where "end-routine" is $FAIL (see Section 5.3.1) if the DO-block was opened for a terminal symbol or recognizer routine and $SUCCES otherwise. The PROCEDURE-block is closed with "END; ELSE DO; CALL $FAIL; RETURN('1'B); END; count-ends END proc-name;", where "proc-name" is the name of the procedure and "count-ends" is the string "END;" repeated a number of times equal to the number of alternatives in the production minus 1. If the current production is the last one, compilation terminates with outputting the code to terminate the external procedure of the SAP and returning a return code of 0 to the operating system.

4.2.3.4 Step 4 (see Figure 4-7d)

This step opens the PROCEDURE-block for each production. The code generated is as follows: "name: PROCEDURE RETURNS (BIT(1));", where "name" is the name of the production. If the production is recursive, as determined during the prologue to Pass 3 by the routine TEST (see Figure 4-7a-1), the procedure



STEP4 OF PASS3
FIGURE 4-7D



STEP6 OF PASS3
FIGURE 4-7E

is also given the RECURSIVE attribute. Code to call the production procedure initialization routine, $MARK (see Section 5.3.1.1), is also generated.

4.2.3.5 Step 5 (see Figure 4-7b)

This step directs control to steps 6 through 12 upon the type of the Encoded Table unit currently being scanned, and, in the case of terminal symbols (type 2), upon the symbol itself. (This is because metalinguistic symbols are typed as terminal symbols in the Encoded Table). Following is the decision table for transfer of control.

Encoded Table Symbol	Algorithm Step
"["	6
Non-metalinguistic terminal symbol	7
Non-terminal symbol	8
Subroutine call	9
"]"	10
"*"	11
" "	12

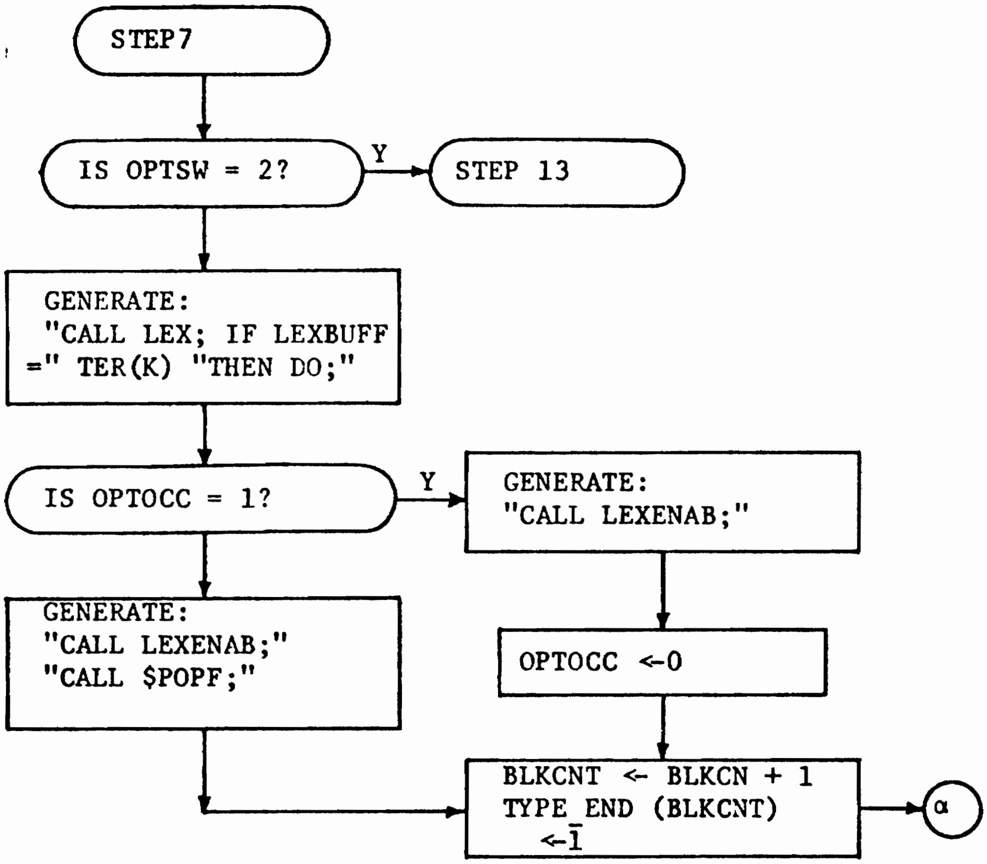
Table 1
Pass 3 Branch Table

4.2.3.6 Step 6 (see Figure 4-7e)

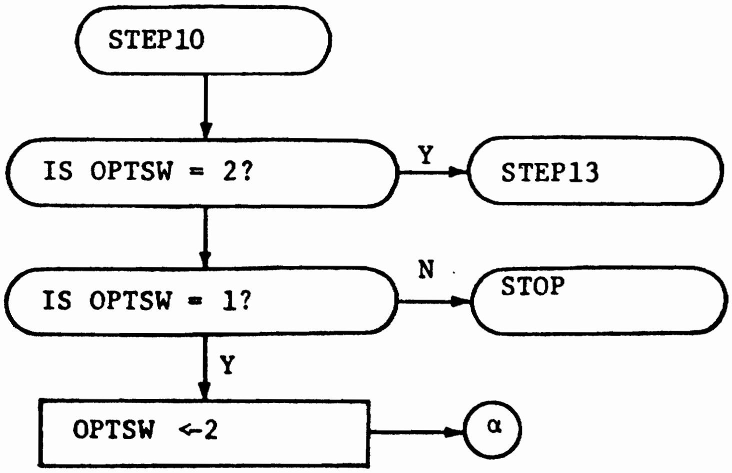
Algorithm control is transferred to this step when the opening optionality bracket "[" is encountered. If the previous unit was the closing optionality bracket "]", then control is transferred to Step 13. Otherwise, items 1, 2, 3, and 5 described in Section 4.2.3.1, along with a system label variable, ETIQ, are pushed onto stacks. A unique label for the optionality group being opened is generated and placed in the system label variable. The code "label:" is generated, where "label" is the current value of ETIQ, i.e., the value just generated. OPTSW and OPTOCC (see Section 4.2.3.1) are initialized to indicate detection of the opening optionality bracket, and BLKCNT (see Section 4.2.3.1) is initialized to zero. The algorithm proceeds with Step 2.

4.2.3.7 Step 7 (see Figure 4-7f)

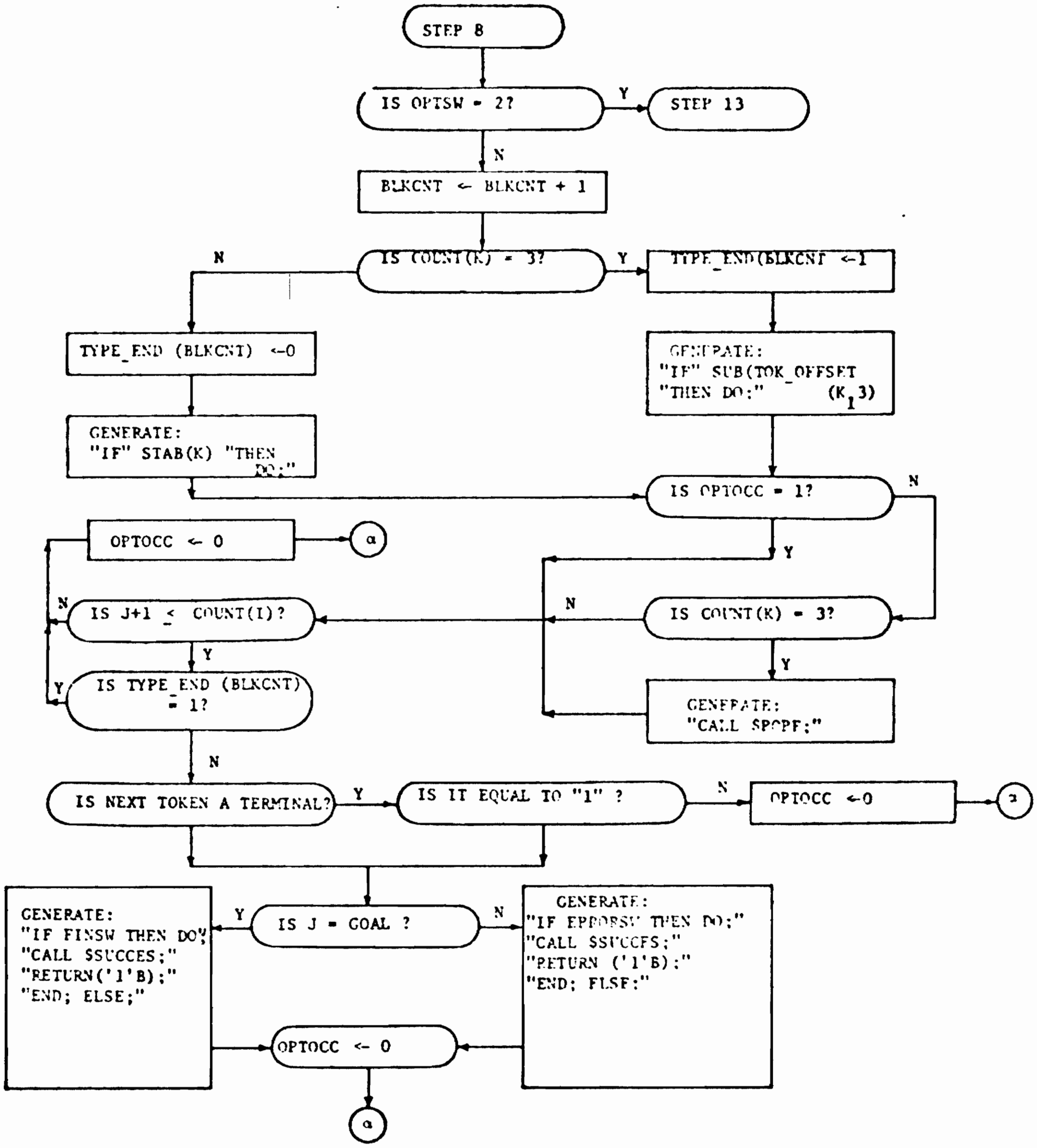
Algorithm control is transferred to this step when a terminal symbol is encountered. Since the previous unit may have been a closing optionality bracket, a check of OPTSW is made to determine if this is the case. If so, algorithm control is transferred to Step 13. Otherwise, recognition code of the following form is generated: "CALL LEX; IF LEXBUFF = 'terminal' THEN DO; CALL LEXENAB;", where "terminal" is the character string to be recognized as was indicated in the EBNF specification. The switch OPTOCC is checked. If it is set, the opening optionality bracket immediately precedes this symbol. This means that this terminal symbol determines whether or not checking for the rest of the optionality group is to be performed. There should be no entry on the error stack (see Section 4.2.4.2) for this terminal symbol and thus no code to call $POPF (see Section 4.2.4.2) is generated. In all other cases this call is generated in the following form: "CALL $POPF;". The indicator for this block (see item (5) in Section 4.2.3.1) is set to indicate a terminal symbol. OPTOCC is reset, and control returns



STEP7 OF PASS3
FIGURE 4-7F



STEP10 OF PASS3
FIGURE 4-7G



STEP 8 OF PASS 3
FIGURE 4-7H

to Step 2.

4.2.3.8 Step 8 (see Figure 4-7h)

This step is entered when a non-terminal symbol is encountered. There are two types of non-terminal symbols: namely, those which reference a recognizer production (i.e., a production which involves only a subroutine call) and those which do not. The former are treated as terminal symbols (see Section 4.2.3.7), except for the first part of the recognition code. The code generated is: "IF recognizer THEN DO;", where "recognizer" is the name of the recognizer routine. (Note that the recognizer routine must return a bit string value of length 1 for proper execution of the SAP).

In the latter case the code generated is: "IF prod-name THEN DO;", where "prod-name" is the production name which is being referenced. This name is the name of a procedure which is internal to the SAP. The code for it already has been or will be generated by the SAPG. If the current Encoded Table entry does not immediately precede either the end of the production or an entry for the metalinguistic symbol "|" then the following code is also generated: "IF switch THEN DO; CALL $SUCCES; RETURN('1'B); END; ELSE;", where "switch" is the string "FINSW" if the current production is the goal production and "ERRORSW" otherwise. The former is to terminate syntactic analysis when the end-of-program statement is recognized, and the latter is to terminate further syntactic analysis of the current statement if an error has been detected in the statement.

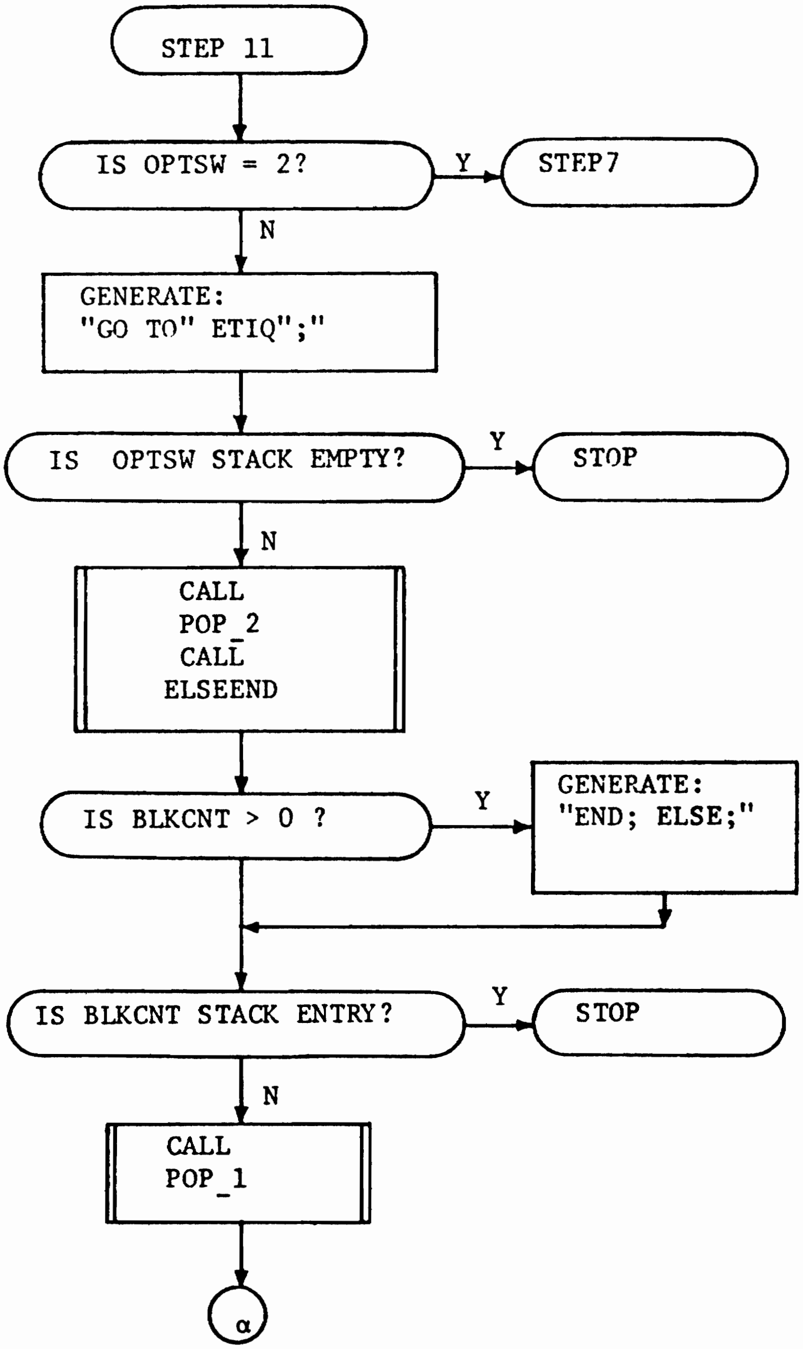
The indicator for the block is set to indicate a non-terminal symbol. In either case, OPTOCC is reset and control is returned to Step 2.

4.2.3.9 Step 9 (see Figure 4-7b)

Algorithm control is transferred to this step when a subroutine call is recognized. The switch OPTOCC is reset, in case the subroutine call immediately follows the opening optionality bracket. The code to call the subroutine is generated as follows: "CALL sub-name;", where "sub-name" is the name of the subroutine to be called. Note that these subroutines will, in general, be external subroutines. For a System/370 implementation the length of this name is restricted to a maximum of 7 characters. Pass 1, in particular LEXEBNF, will reject any production which contains a subroutine name of length greater than 7. Control then returns to Step 2.

4.2.3.10 Step 10 (see Figure 4-7g)

This step is entered whenever a closing optionality bracket is recognized. A check is first made to see if the preceding character was also a closing optionality bracket. If so, control is transferred to Step 13 to finish generating the necessary code for the optionality group. (This is done by checking OPTSW). If not, a check is then made to see if an opening optionality bracket was detected previously in this production, again by checking OPTSW. If so, OPTSW is updated to indicate that a closing optionality bracket was recognized, and control returns



STEP11 OF PASS3
FIGURE 4-71

to Step 2. Otherwise, an error has occurred due to an unmatched closing bracket. An error message is printed and SAPG terminates.

4.2.3.11 Step 11 (see Figure 4-7i)

Control is transferred to this step when an asterisk is detected. Since the asterisk could stand for itself (i.e., by a terminal symbol) or by the Kleene asterisk, OPTSW is checked. If OPTSW indicates that the preceding symbol was the closing optionality bracket, then the asterisk is interpreted as the Kleene asterisk. Otherwise, it is treated as a terminal symbol, and control is transferred to Step 7. These interpretations impose the restriction that the asterisk cannot be interpreted as a terminal symbol immediately following an optionality group.

If the asterisk is the Kleene asterisk, then "GO TO label;" is generated, where "label is the current value of the system label variables, ETIQ. When the SAP executes this instruction it will cause syntactic scanning to begin again with the first syntactic unit of the optionality group which immediately precedes the asterisk. Each opened DO-block within the optionality group, except the first, is closed as described in Section 4.2.3.3. The first DO-block is closed with "END; ELSE;". The stacks which were pushed in Step 6 are popped and control returns to Step 2.

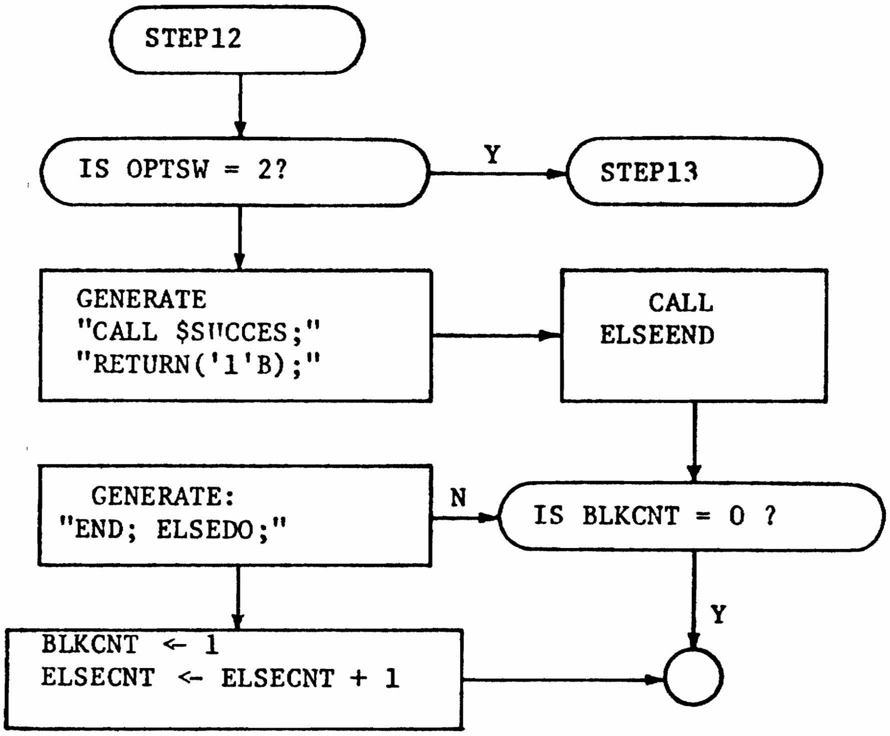
4.2.3.12 Step 12 (see Figure 4-7j)

Step 12 is entered when the code for an alternative of the production has been generated and there is still at least one more alternative to be processed, indicated by the metalinguistic symbol "|". Code to indicate successful recognition of the previous alternative is generated in the form: "CALL $SUCCES; RETURN('1'B);". As in other steps, if the preceding alternative ends with an optionality group, this group must be closed as described in Section 4.2.3.11. Then code to close the DO-blocks for the required syntactic units, if any, is generated as described in Section 4.2.3.3. However, the first block of the alternative is closed with "END; ELSE DO;". This is because there are still other possibilities to be checked. Note that this ELSE-clause will be executed only if the first syntactic unit of each of the preceding alternative fails to be recognized. (It will not be executed if recognition fails internally in a preceding alternative). The arrangement of the alternatives within a production will affect the execution time of the SAP, since the analysis takes place sequentially in the order in which the alternatives are listed. Execution time may be improved by listing the alternatives of a production in decreasing order of their frequency of occurrence, i.e., with the most frequently occurring one listed first.

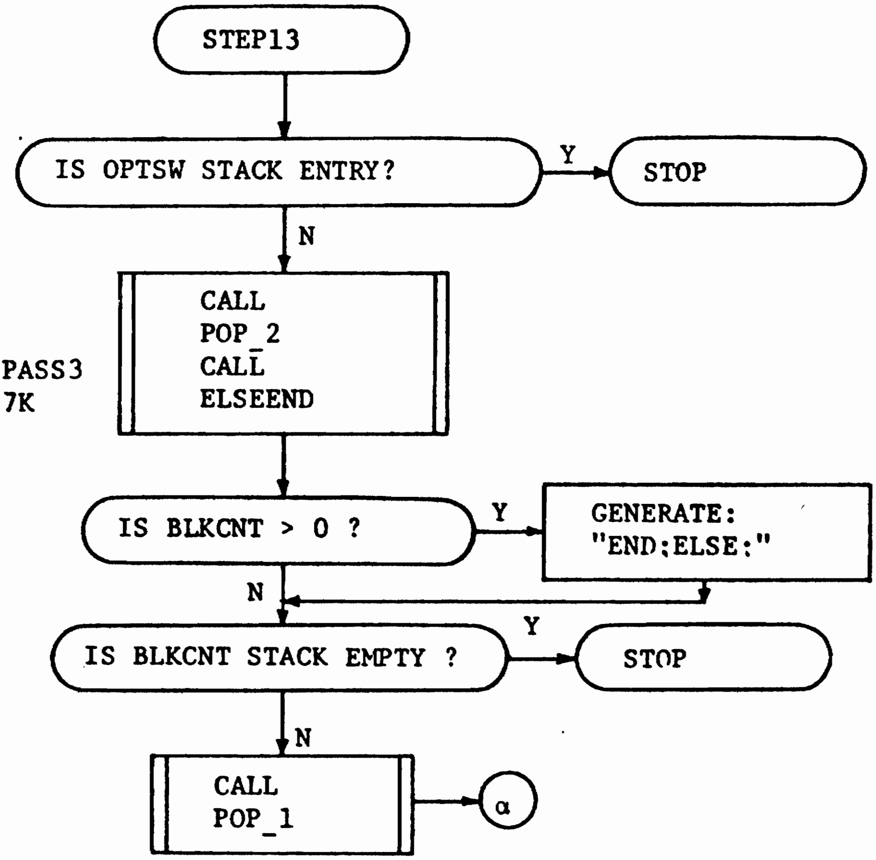
BLKCNT is reinitialized to zero and ELSECNT is incremented by one. (ELSECNT is used to generate the appropriate number of "END;"'s for the "ELSE DO;"'s generated by this step). Control then is transferred to Step 2.

4.2.3.13 Step 13 (see Figure 4-7k)

This step is required to close all optionality groups which



STEP12 OF PASS3
FIGURE 4-7J



STEP13 OF PASS3
FIGURE 4-7K

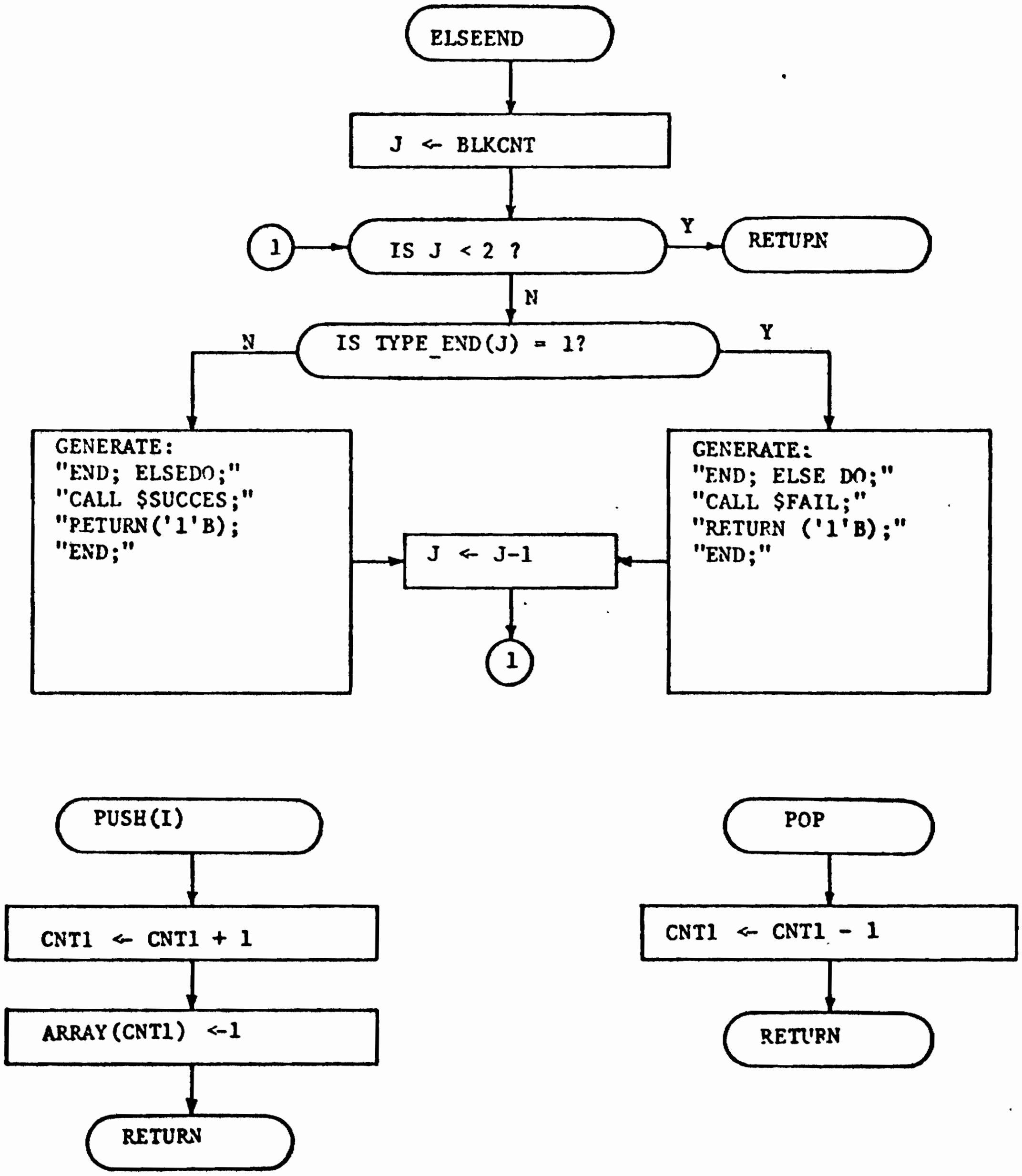


FIGURE 4-7J-1

ELSEEND,PUSH(I), POP

are not followed by the Kleene asterisk, except one which is last in a production. That is, there is nothing else following it in the production. Step 13 is identical to Step 11, except that the code of the form: "GO TO label;" is not generated, and control is transferred to Step 5, instead of Step 2, at the completion of the step.

4.3 SAPG Implementation Restrictions on EBNF

Due to the way in which the SAPG has been implemented, there are two restrictions on the way in which ENBF may be used to specify a language when a SAP is desired as output. The first of these is due to the strictly sequential nature of the syntactic analysis of the SAP. This means no production in the EBNF specification can involve left recursion. A production is left recursive if the first symbol of its definition is a non-terminal symbol and either it or a valid substitution of any of its possible alternatives produces a result which makes an immediate recursive reference to it. For example, the following sets of productions are left recursive.

(1) < production1 > ::= < production1> < production2>

(2) < production3 > ::= < production4> < production5>
< production4 > ::= < production5> < production6>
< production5 > ::= < production3> < production6>

Note that in the second example the recursive call on each production is made two levels away, but the productions are still to be considered left recursive. If the code which would be generated for a SAP which included productions such as these, using the current implementation of SAPG, were executed, the program would loop until memory for storage areas for the invocation of procedures were exhausted. A solution to this problem will not be presented here, but it would involve the use of a run-time stack. The reader may wish to refer to Carr and Weiland [CAR 69] for a discussion of a method by which as much recursion as possible can be eliminated from an EBNF specification.

A second restriction is that an optionality group must be distinguishable by its first element. That is, the first syntactic unit which follows the optionality group, whether that unit be required or itself be the first element of an optionality group, must be different from the first element of the optionality group which it follows. This is the reason for the special treatment of the character "," as described in Chapter 3 . Otherwise, a unique separator would have to be required as the first character in each optionality group in a sequence. For example, if "[,KEYWORD1 = < name >] [,KEYWORD2 = < name >]" appeared in an EBNF production and the character "," were not part of the keyword but a lexical unit by itself, it would be impossible to determine by scanning the comma which optionality group the comma was beginning. This restriction would not be necessary if the lexical routine were not strictly sequential but could be made to back up in its scan, either in actuality or through a run-time stack. Making the comma part of the keyword distinguishes the optionality group by its first element.

These restrictions should not present too great a hindrance to the SAPG user. The left recursion restriction can be circumvented when it is used to achieve repetition by using the repetition feature of EBNF. One way to solve the problem of distinguisable optionality groups has been described above and is being used in the current implementation of DDL.

SAP produced by SAPG for the example given in Section 4.2

```
SAP:  PROCEDURE OPTION (MAIN) RECORDER;
DCL SIMPLE ARITH_EXP RETURNS (BIT(1)) REDUCIBLE;
DCL TERM RETURNS (BIT(1)) REDUCIBLE;
DCL MULT_OP RETURNS (BIT(1)) REDUCIBLE
DCL ADD_OP RETURNS (BIT(1)) REDUCIBLE;
DCL INTREC RETURNS (BIT(1)) REDUCIBLE;
DCL NAMEREC RETURNS (BIT(1)) REDUCIBLE;
DCL LEXBUFF CHAR(31) VARYING EXTERNAL;
DCL ERRORSW BIT(1) EXTERNAL;
FINSW BIT(1) EXTERNAL;
CALL DDLOPT;
IF SIMPLE_ARITH_EXP THEN RETURN;

SIMPLE_ARITH_EXP:PROCEDURE RECURSIVE RETURNS (BIT(1));
CALL $MARK;
$SYS_001:
IF ADD_OP THEN DO;
IF FINSW THEN DO; CALL $SUCCES; RETURN('1'B); END; ELSE;
END; ELSE;
IF TERM THEN DO;
IF FINSW THEN DO; CALL $SUCCES; RETURN('1'B); END; ELSE;
$SYS_002;
IF ADD_OP THEN DO;
IF FINSW THEN DO; CALL $SUCCES; RETURN('1'B); END; ELSE;
IF TERM THEN DO;
IF FINSW THEN DO; CALL $SUCCES; RETURN('1'B); END; ELSE;
GO TO $SYS_002;
END; ELSE DO; CALL $SUCCES; RETURN('1'B); END;
END; ELSE;
CALL $SUCCES; RETURN('1'B);
END; ELSE DO; CALL $FAIL; RETURN('0'B); END;
END SIMPLE_ARITH_EXP;

TERM:PROCEDURE RECURSIVE RETURNS(BIT(1));
CALL $MARK;
IF PRIMARY THEN DO;
IF ERRORSW THEN DO; CALL $SUCCES; RETURN('1'B); END; ELSE;
$SYS_003:
IF MULT_OP THEN DO;
IF ERRORSW THEN DO; CALL $SUCCES; RETURN('1'B); END; ELSE;
IF PRIMARY THEN DO;
IF ERRORSW THEN DO; CALL $SUCCES; RETURN('1'B); END; ELSE;
GO TO $SYS_003;
END; ELSE DO; CALL $SUCCES; RETURN('1'B); END; ELSE;
END; ELSE;
CALL $SUCCES; RETURN('1'B);
END; ELSE DO; CALL $FAIL; RETURN('0'B); END;
END TERM;
```

FIGURE 4-8

```
PRIMARY:PROCEDURE RECURSIVE RETURNS(BIT(1));
CALL $MARK;
IF INTREC THEN DO; CALL $POPF;
CALL $SUCCES; RETURN('1'B);
END; ELSE DO;
IF NAMEREC THEN DO; CALL $POPF;
CALL $SUCCES; RETURN('1'B);
END; ELSE DO;
CALL LEX; IF LEXBUFF = '(' THEN DO;
CALL LEXENAB; CALL $POPF;
IF SIMPLE_ARITH_EXP THEN DO;
IF ERRORSW THEN DO; CALL $SUCCES; RETURN('1'B); END; ELSE;
CALL LEX; IF LEXBUFF = ')' THEN DO;
CALL LEXENAB; CALL $POPF;
CALL $SUCCES; RETURN('1'B);
END; ELSE DO; CALL $FAIL; RETURN('1'B); END;
END; ELSE DO; CALL $SUCCES; RETURN('1'B); END;
END; ELSE DO; CALL $FAIL; RETURN('0'B); END;
END;
END
END PRIMARY;

MULT_OP:  PROCEDURE RETURNS(BIT(1));
CALL $MARK;
CALL LEX; IF LEXBUFF = '*' THEN DO;
CALL LEXENAB; CALL $POPF;
CALL $SUCCES; RETURN('1'B);
END; ELSE DO;
CALL LEX; IF LEXBUFF = '/' THEN DO;
CALL LEXENAB; CALL $POPF;
CALL $SUCCES; RETURN('1'B);
END; ELSE DO; CALL $FAIL; RETURN('0'B); END;
END;
END MULT_OP;

ADD_OP:PROCEDURE RETURNS(BIT(1));
CALL $MARK;
CALL LEX; IF LEXBUFF = '+' THEN DO;
CALL LEXENAB; CALL $POPF;
CALL $SUCCES; RETURN('1'B);
END; ELSE DO;
CALL LEX; IF LEXBUFF = '-' THEN DO;
CALL LEXENAB; CALL $POPF;
CALL $SUCCES; RETURN('1'B);
END; ELSE DO; CALL $FAIL; RETURN('0'B); END;
END;
END ADD_OP;
END SAP;
```

FIGURE 4-8
-continued -

CHAPTER 5

THE DDL COMPILER

5.0 Introduction - An Overview of Compiling

5.1 The Components of a Compiler

The conversion of the source program (e.g. DDL), which is nothing more than a string of characters, into the object program can be regarded as a series of simple transformations rather than as one complex transformation. A compiler ultimately converts the string of characters into a string of bits, the object code. The DDL compiler is a "multi-pass" translator in which a sequence of transformations is applied to the program as a whole.

The technique of transforming the source program into the object program by means of a sequence of simple transformations leads naturally to the idea of "intermediate languages." For example, the DDL compiler involves the translation of DDL into PL/1, and then it uses the PL/1 compiler to translate PL/1 to the machine language of the IBM/370. The DDL compiler takes advantage of the existence of the PL/1 compiler by including it as the last stage of the translation process. Thus the DDL compiler can avoid machine-oriented problems such as storage allocation, register management, operating system interface, etc., by assuming that the object language it generates is to be processed by a "PL/1 machine."

In this compilation process, subprocess with the following names can often be identified:

(1) Lexical Analysis

(2) Syntax Analysis

(3) Bookkeeping or Symbol and Data Table Creation

(4) Code Generation or Translation to Intermediate Code (e.g. DDL → PL/1)

(5) Object Code Generation (e.g. PL/1 → Machine Code)

The monitor of the DDL compiler (see Figure 5.1) is called DDLCOMP. DDLCOMP is the first module to be loaded into core when the DDL compiler is called by the user. The task of DDLCOMP is to call on the different phases of the compiler. The first thing DDLCOMP does is to open the internal working files, which are:

a) SAPERR - where the error messages (if any) produced by the SAP, are to be stored

b) TABERR - where the error messages (if any) found by the table generating routines, (i.e., global syntax checking) are kept

c) CODERR - where the error messages (if any) found at code generation time, (i.e., final part of global syntax checking) are to be stored.

d) SAPLIST - where the DDL source statements are stored

e) XREFTAB - where the cross-reference table is kept

f) OUT7 - where the user written DML routines are stored.

g) OPTLIST - where the parameters given to the DDL compiler

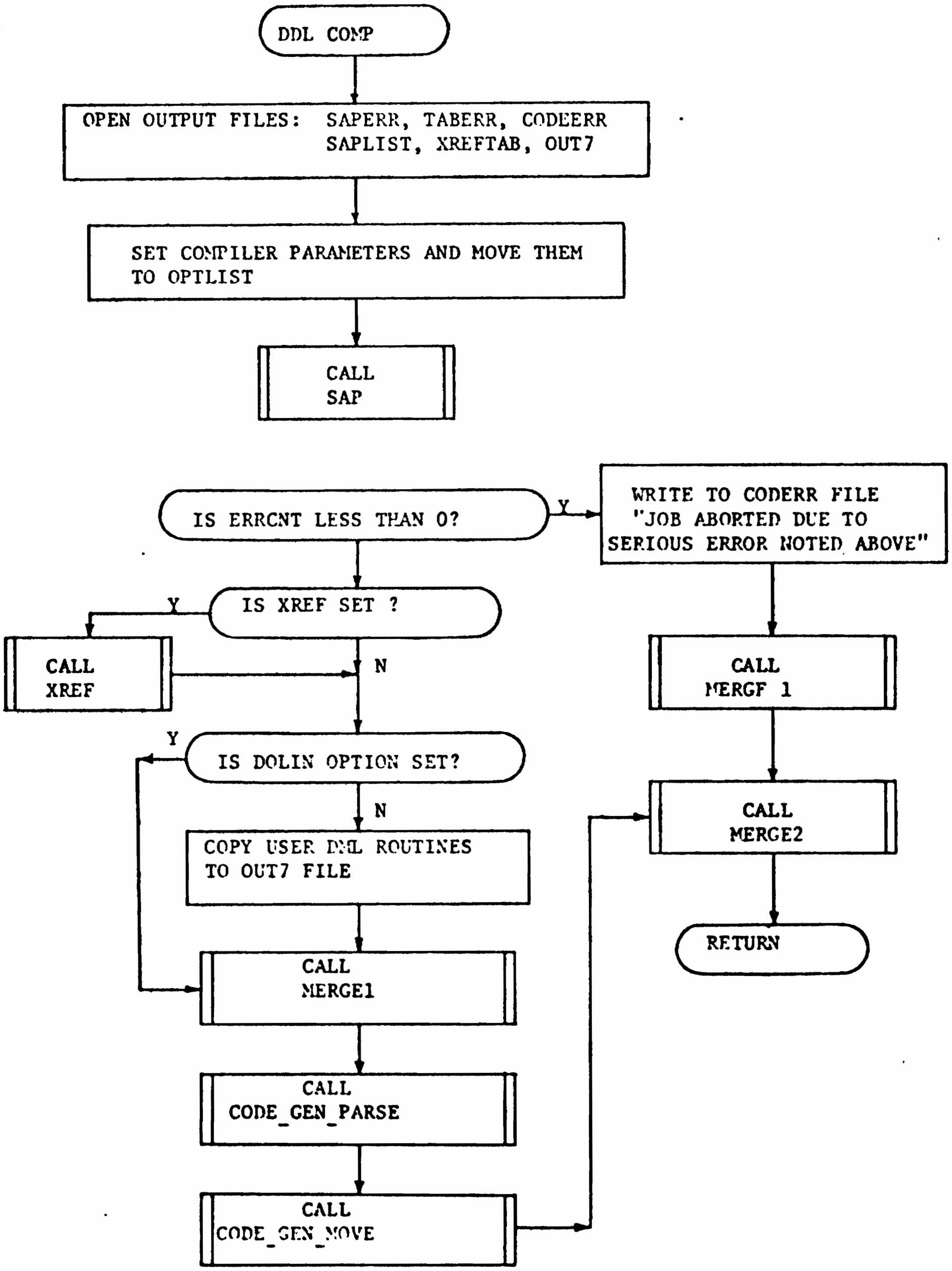

DDL COMP
OPEN OUTPUT FILES: SAPERR, TABERR, CODEERR SAPLIST, XREFTAB, OUT7
SET COMPILER PARAMETERS AND MOVE THEM TO OPTLIST
CALL SAP
IS ERRCNT LESS THAN 0?
Y
WRITE TO CODERR FILE "JOB ABORTED DUE TO SERIOUS ERROR NOTED ABOVE"
IS XREF SET ?
Y
CALL XREF
N
CALL MERGF 1
Y
IS DOLIN OPTION SET?
N
COPY USER DML ROUTINES TO OUT7 FILE
CALL MERGE2
RETURN
CALL MERGE1
CALL CODE_GEN_PARSE
CALL CODE_GEN_MOVE

FIGURE 5.1

DDLCOMP

are stored.

After the working files are opened DDL COMP sets the compiler parameters and moves them to the OPTLIST file. The SAP (see Section 5.3) which is called next, in turn will call on LEX (see Section 5.2) to perform the lexical analysis. If there were no errors found during execution of the SAP and if the XREF option was specified then XREF is called (see Section 5.4) to produce a cross-reference table. If errors were detected the following message is written on the CODERR file:

"JOB ABORTED DUE TO SERIES ERROR NOTED ABOVE". Then MERGE 1 and MERGE 2 are called and the process terminates.

After the XREF procedure has been called, a test is made to see whether or not the user has supplied DML routines, in which case they are written into the OUT7 file. In order to merge the error and listing files, MERGE 1 is invoked followed by the code generation procedures of CODE_GEN_PARSE (see Section 5.5) and CODE_GEN_MOVE (see Section 5.6). Finally, MERGE 2 is called and the process terminates.

MERGE 1 and MERGE 2 are entry points of the procedure named MERGE_OUT_FILES, which merge all the output and listing files. The functions of MERGE 1 and MERGE 2 are given in Section 5.6.14.

We shall describe the first four phases of compilation in the following sections.

5.2 Lexical Analysis

The lexical analysis phase comes first. The input to the compiler and hence to the lexical analyzer is a string of symbols from an alphabet of characters. In the DDL language for example, the terminal symbol alphabet contains the following 60 symbols

A B C ... Z $ # @ -
0 1 2 ... 9
= + - * / ( ) , . ; : ' & | - > < ? %

In a DDL program, certain combinations of symbols are treated as a single entity, called a "token", such as the following:

(a) A string of one or more blanks is normally treated as a separator.

(b) Comments i.e., those strings of characters enclosed between "/*" and "*/" are treated as a single blank.

(c) DDL has keywords such as ",PRE_CRIT", ",SIZE=", ",CONV=", ",LOCK=" etc. which are treated as single entities.

(d) Strings representing numerical constants and strings of characters enclosed in quotes are treated as single items.

(e) Identifiers (DDL names) for Groups, Fields, Records, DML Procedures and the like are also treated as single lexical units in DDL.

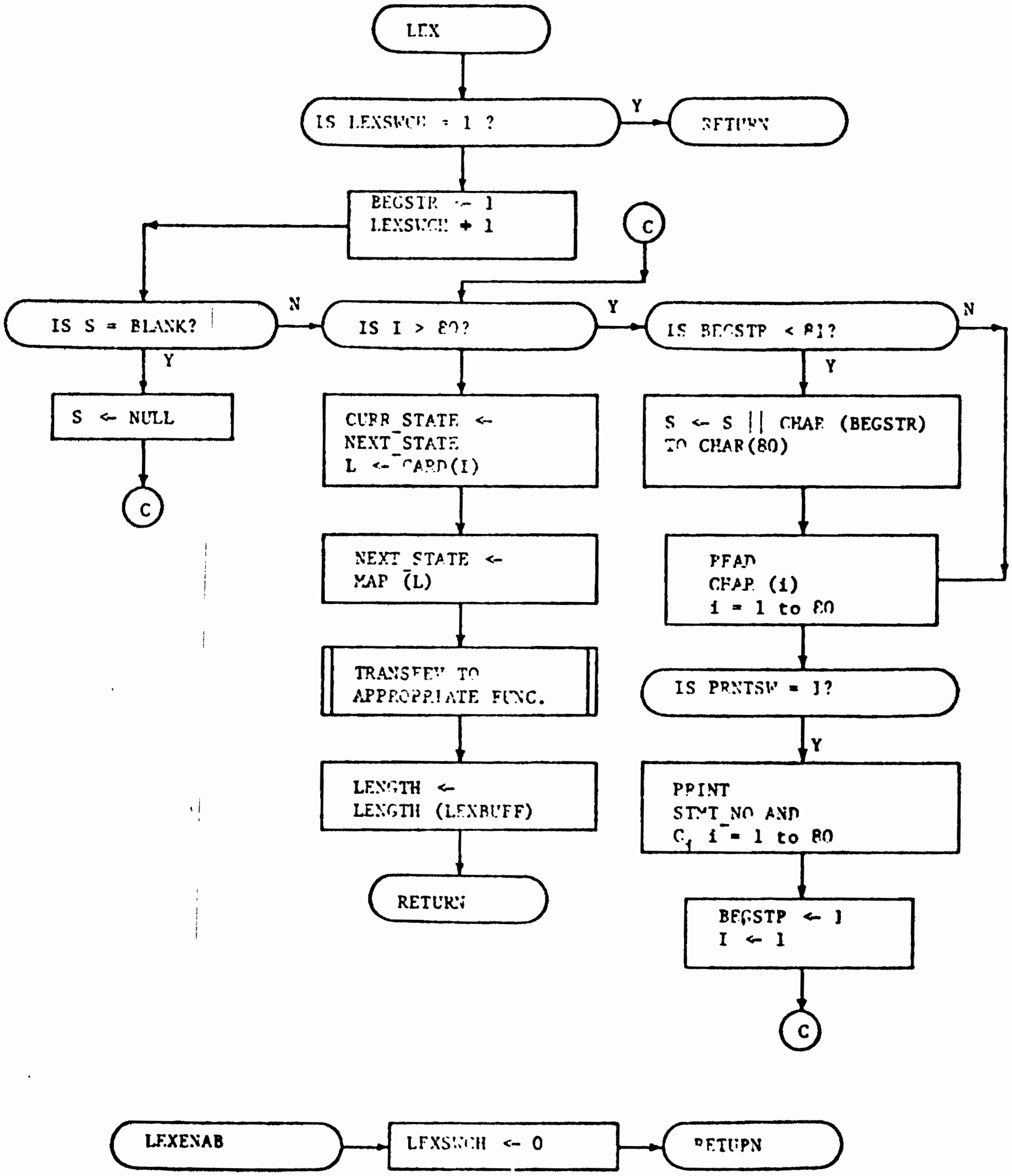
The Lexical analyzer used in the DDL compiler is called LEX.

LEX for the current implementation of DDL is written after the same manner as LEXEBNF. The completely metalinguistic symbols, e.g., "|", "[", "]", are treated as invalid characters by LEX. Otherwise, the two routines are the same as far as lexical analysis is concerned, except in one point. Since a production may have several alternatives, the SAP may have to check the same value of LEXBUFF several times before it determines the correct production alternative to be analyzed further. Since a call to LEX is automatically generated for each terminal symbol, the lexical routine must have the capability, in effect, to back up in order to use the same lexical unit again. This is accomplished by having LEX disable itself by setting an internal switch after it produces a lexical unit. When this switch is set, a call to LEX will not cause further lexical analysis to take place. LEX simply returns to the calling routine, and LEXBUFF contains the same value as it did before LEX was called.

When successful recognition of the contents of LEXBUFF occurs, LEX must be enabled again to allow further analysis. This is accomplished by a call to another entry point in the lexical routine, LEXENAB. A call to LEXENAB merely resets the internal switch which disables LEX. This call, as the call to LEX, is automatically generated for EBNF terminal symbols but must by issued specifically by a recognizer routine.

One restriction to LEX (and LEXEBNF) was made for the DDL implementation. This is in the area of keywords, which may be best explained by example. In DDL, the string, ",KEYWORD=" will be treated as a single keyword. That is, if the character "," immediately precedes an alphanumeric character string, it will be treated as part of that character string. If the character "=" immediately follows an alphanumeric character string, it will be treated as part of the character string. The "," will terminate the character string which immediately precedes it, and the "=" will terminate the character string which it follows. If the single characters "," and "=" are desired, they must be succeeded and preceeded, respectively, by the blank character. It is important to remember this restriction when commas are used to separate items in a list. A blank must immediately follow the separating commas, or the comma will be treated as part of the next item in the list and a compilation error will result.

There are nine entry points to the lexical routine, including the main entry point, LEX. The others are LEXENAB, STMT_FL, CHARSTR, BITSTR, NUMSTR, GETCHAR, GETBIT, and GETNUM. (The last three are the same). LEX, LEXENAB, and STMT_FL are used during normal lexical analysis. CHARSTR, NUMSTR, and BITSTR are entry points for obtaining special character strings which cannot be analyzed properly by invoking the main entry point, LEX. GETCHAR, GETNUM, and GETBIT are used by the DDL table creation routines to obtain the character strings for encoding purposes. Each of these entry points is described in more detail below.



LEX FLOW DIAGRAM
FIGURE 5-2

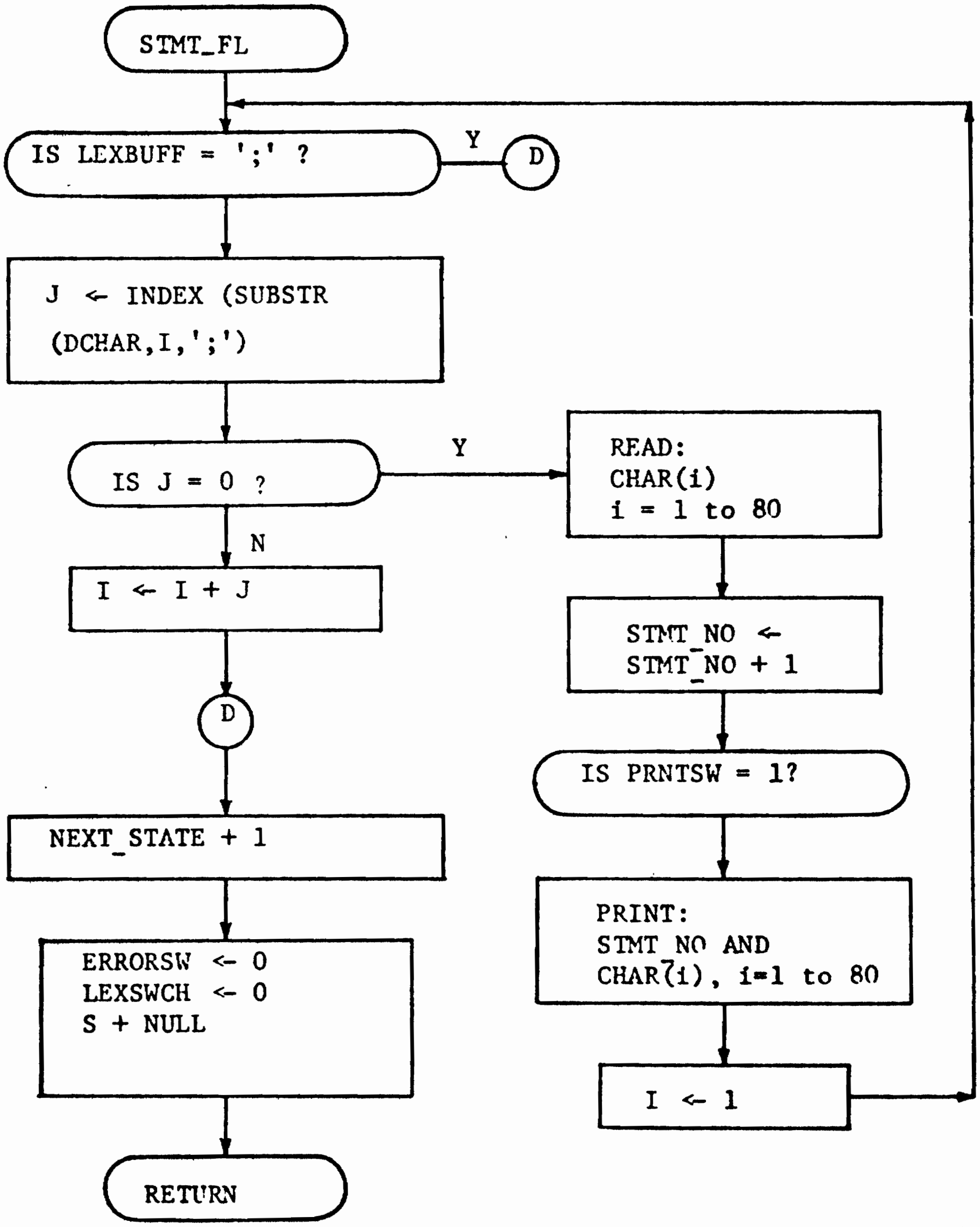


FIGURE 5-2A
STMT_FL

During syntactic analysis the SAP calls upon the routine LEX to return the next token in the input stream, referred to by the ddname DDLIN. This unit is returned in the global variable LEXBUFF as a character string (maximum length of 31 characters); its length is returned in the global variable LENGTH. LEX inputs source records as necessary to fulfill these requests and scans all 80 columns of the records. The contents of LEXBUFF are compared with the character string specified by the EBNF description (see Section 3.3 and 3.5). If the string is specified as a terminal symbol, the comparison is performed by the main SAP procedure, i.e., by the generated code. If the string is to be recognized by a recognizer routine, this routine must issue the call to LEX to obtain the lexical unit for analysis.

There are other entry points to the DDL implementation of LEX. The first of these is STMT_FL. This entry point is called when an error occurs. The rest of the current input record is scanned for the (;) end-of-statement character. If it is found, LEX is set up to begin further lexical analysis with the character which immediately follows the end-of-statement character. If it is not found, another input record is read and scanned in the same manner. This process continues as long as necessary unless an end-of-file condition occurs on the input data set. In this case an end-of-program record is placed in the input buffer to assure termination of compilation. (Input records are placed, along with a statement number, in the data set named by the ddname LISTING if the global switch PRNTSW is set. Otherwise, they are not. In the current DDL implementation, this switch is set by the routine DLLOPT.) LEX is enabled and put in the state of having just scanned a blank character, and ERRORSW is reset.

Since DDL permits blank characters to occur in character string constants, special entry points in LEX are provided to scan for character (CHARSTR), numeric (NUMSTR), and bit (BITSTR) strings. (These strings are designated by surrounding apostrophes). The maximum length allowed for these strings is 32,767. The entry points NUMSTR and BITSTR also validate the character string. Since these strings may be longer than 31 characters, the maximum length of LEXBUFF, the entry points GETCHAR, GETNUM, and GETBIT are provided for the return of the actual character string. These entry points have a character string argument in which the character string will be placed. The required storage for the character string may be obtained from the global variable LENGTH, since the length of the character string is placed there before returning from CHARSTR, NUMSTR, or BITSTR.

Source records are input as necessary until the character string is terminated. If an end-of-file condition occurs, an end-of-program record is placed in the input buffer and the routines return a length-1 bit string value of 0.

The Transition Matrix (STATE-TABLE) for LEX is

	0	1	2	3	4	5	6	7	8	9	10
0	1	2	4	4	7	4	1	4	7	4	7
1	1	3	5	1	7	1	5	1	7	5	7
2	4	2	4	4	4	4	7	4	7	4	7
3	7	7	7	4	6	7	7	7	7	7	7
4	7	2	4	4	7	7	7	7	7	7	7
5	7	7	7	7	7	7	1	7	1	7	7
6	4	2	7	4	4	7	7	7	7	4	7
7	1	2	4	4	7	7	7	7	7	7	7
8	4	2	7	4	7	7	7	7	7	4	7
9	4	2	4	4	4	4	4	4	4	4	4
10	7	7	7	7	7	7	7	7	7	7	7

The character mapping table is as follows:

CLASS	CHARACTER
0	ABC...2 01...9_$
1	b (space)
2	; : ( ) . #
3	/
4	*
5	<
6	=
7	,
8	-
9	.

The lexical functions performed are determined by the entries in the decision table given above. These entries are used to index an array of labels. The contents of this array are as follows:

```
LABEL(1)  =  F1

LABEL(2)  =  F2

LABEL(3)  =  F3

LABEL(4)  =  F4

LABEL(5)  =  F5

LABEL(6)  =  F6

LABEL(7)  =  F7
```

The lexical functions performed are:

```
F1: I=I+1

F2: F4:  LEXBUFF=S  || SUBSTR(DCHAR,BEGSTR,I-BEGSTR); I=I+1;
    S=L; GO TO RETURN;

F3:  DO I=I+1 TO 80;
     IF CHAR(I) ¬= '' THEN DO; BEGSTR=I; GO TO SCAN; END;
     END;
     READ FILE(DDLIN) INTO (DCHAR);
     I=0; GO TO F3;

F5:  LEXBUFF=L; LENGTH,NEXT_STATE=1; I=I+1; RETURN;

F6:  I=I+1;
     J=INDEX(SUBSTR(DCHAR,I),'*/');
     IF J=0 THEN DO;
     READ FILE(DDLIN) INTO (DCHAR);
     I=0; GO TO F6; END;
     ELSE DO; I=I+J+2; S=''; NEXT_STATE=1; GO TO SCAN; END;

F7:  LEXBUFF=S || SUBSTR(DCHAR,BEGSTR,I+1-BEGSTR);
     NEXT_STATE=1; S=''; GO TO RETURN;
```

RETURN is a label which begins the following code:

```
RETURN:  BEGIN;
DCL LENGTH BUILTIN;
J=LENGTH(LEXBUFF); END;
LENGTH=J; RETURN;
```

(Note: the code which generates the output listing has been omitted from the above code. SCAN is the label which begins the loop which performs the scanning of the individual characters in the input).

5.3 Phase 1 of the DDL compiler

Phase 1 includes the syntax analysis and the symbol and data table creation.

The syntax analysis of DDL statements is carried out by the SAP generated by the SAPG (see Chapter 4). The syntax analysis is performed using the top-down technique. That is, syntax analysis is achieved by checking the input statements for satisfaction of the definition of the goal symbol, i.e., a "program" written in the DDL language. It can also be considered to be checking to see if the input is a "good word" of the language.

Therefore, the syntactic analysis phase is responsible for the detection and flagging of errors on the DDL source input. Concurrent with the syntactic analysis is table generation. At this time, the subroutines used in the EBNF specification for DDL are called. The functions of these routines are table building (i.e., encoding the DDL statements) to preserve the necessary information to be used during code generation, as well as in the detection of global syntax errors. Table generation is presented in Section 5.3.6 and the format of Symbol Table and Data Table in Sections 5.3.4 and 5.3.5, respectively.

The following Sections describe the syntax analysis phase.

5.3.1 Entry Points To The SAP

There are six entry points to this routine. They are $MARK, $POPF, $SUCCESS, $FAIL, CLRERRF, and $PUSH_F. Calls to the first five are generated automatically by SAPG according to the description of the DDL grammar. $PUSH_F must be called explicitly by the compiler writer from an error-stacking routine (see Section 5.3.2.1).

5.3.1.1 The Entry Point $MARK (see Figure 5-3a)

This entry point is called upon entry to a production procedure to mark the beginning of errors which may be pushed onto the error stack by routines called during the execution of the procedure. This is done by calling $PUSH_F which pushes value of -1 onto the error stack.

5.3.1.2 The Entry Point $POPF (see Figure 5-3b)

$POPF is called after successful recognition of a terminal symbol or upon a successful return from a recognizer routine. The effect of calling this routine is to pop the error stack, i.e., to remove the top entry. If the entry at the top of the error stack is -1, the error stack is not popped. Only a call to $SUCCES can pop this type of entry.

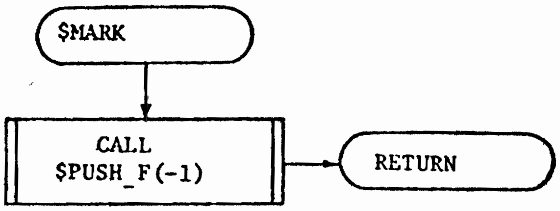


FIGURE 5-3A
$MARK

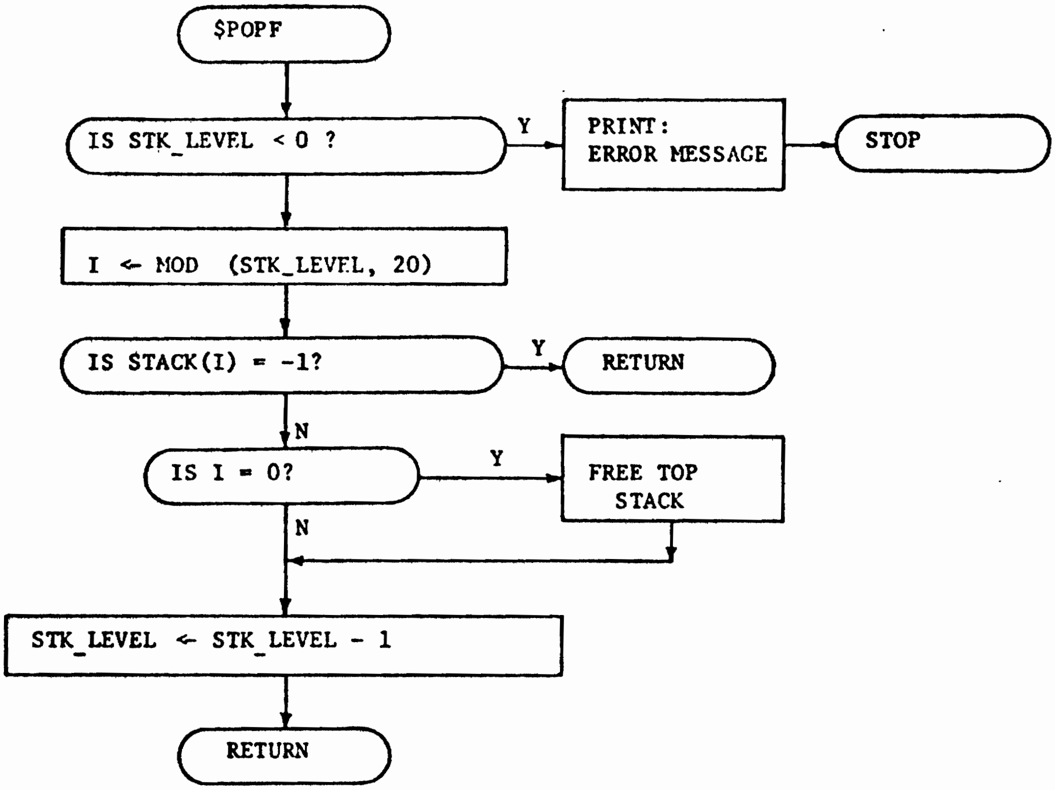


FIGURE 5-3B
$POPF

5.3.1.3 The Entry Point $SUCÇES (see Figure 5-3c)

This entry is called upon termination of a production procedure, whether successful or not, to restore the error stack to the way it was in before the procedure was invoked. This is done by removing all entries from the top of the stack up to and including the entry -1 which is nearest the top of the error stack.

In both entries $POP-F and $SUCCES if more than one stack has been allocated and popping the top stack empties it, the top stack is freed, bringing the previous error stack to the top. If the stack underflows the error message "ERROR STACK UNDERFLOW. COMPILATION DISCONTINUED," is printed and compilation immediately terminates.

5.3.1.4 The Entry Point $FAIL (see Figure 5-3d)

$FAIL is called from SAP when a local error occurs during syntactic analysis in order to print out an error message.

The call to $FAIL is generated by the SAPG as part of SAP for all terminal symbols and recognizer routine references (see Section 5.3.2.2)

When $FAIL is called, the following message is printed:
" < ERROR_CODE > ERROR. INVALID TEXT BEGINNING '< BAD_TEXT > IN STATEMENT NUMBER < STMT_NO >.

Where

< ERROR_CODE > is the entry at the top of the error stack,

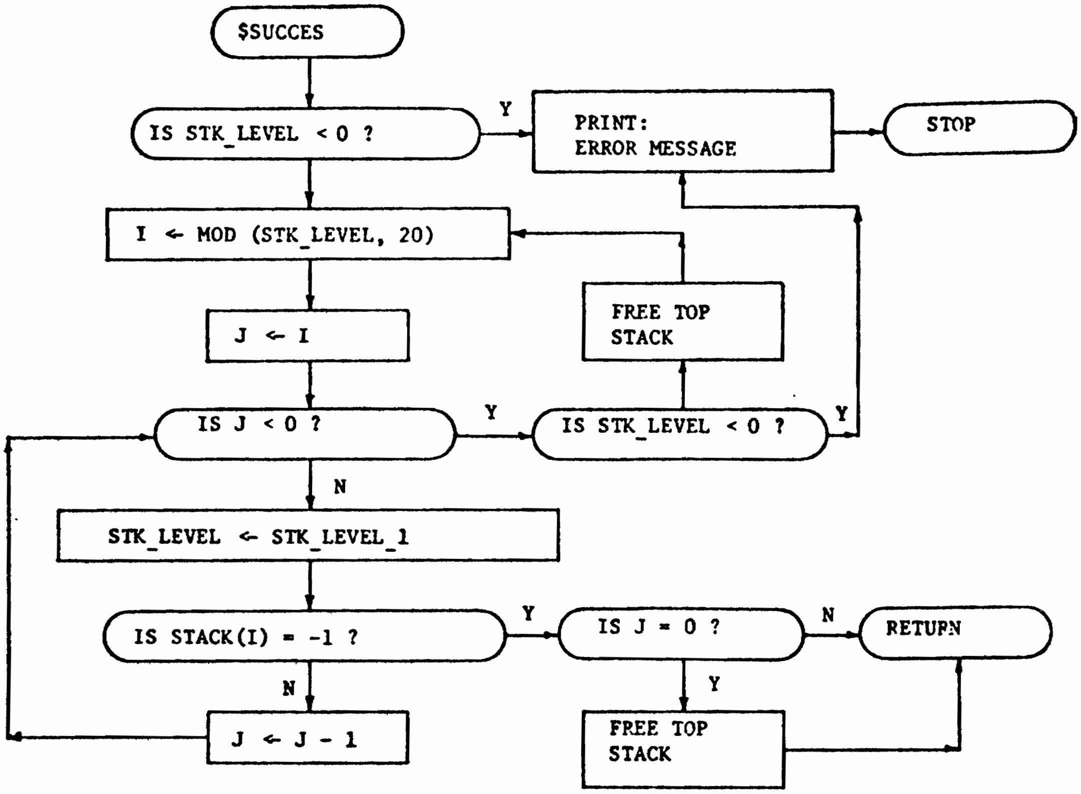


FIGURE 5-3C
$SUCCES

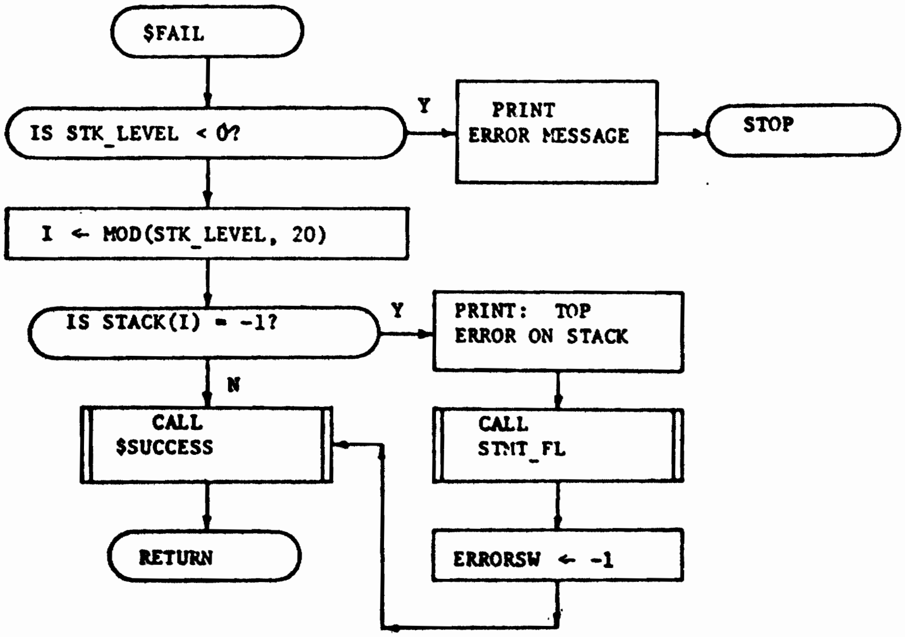


FIGURE 5-3D
$FAIL

< BAD_TEXT > is the current contents of LEXBUFF, and

< STMT_NO >is the current statement number.

Error codes are character strings of maximum length 6. Any other value results in no message being printed. In either case, ERRORSW is set, $SUCCES is called to restore the stack to its contents upon entry to the production procedure, and STMT_FL is called to prepare for the continuation of syntax analysis of the next statement. If ERRORSW is set then it causes all further analysis of the current statement to cease.

5.3.1.5 The Entry Point $PUSH_F (see Figure 5-3e)

This entry point is called by the error-stacking routines (see Section 5.3.2.1) and by the entry point $MARK.

This entry point has a one-dimensional array whose elements are characters strings of length six (where the error codes are to be stored).

The array is pushed onto the error stack in reverse order with the last element being pushed first. The result is that the first element of the array will be at the top of the error stack. This allows the compiler writer to code error messages for a production in a left-to-right manner.

5.3.1.6 The Entry Point CLRERRF (see Figure 5-3f)

When ERRORSW is set (see Section 5.3.1.4), no further syntactic analysis will take place until it is reset. The entry CLRERRF is provided for this purpose. Its placement in the EBNF specification is described in Section 5.3.3.

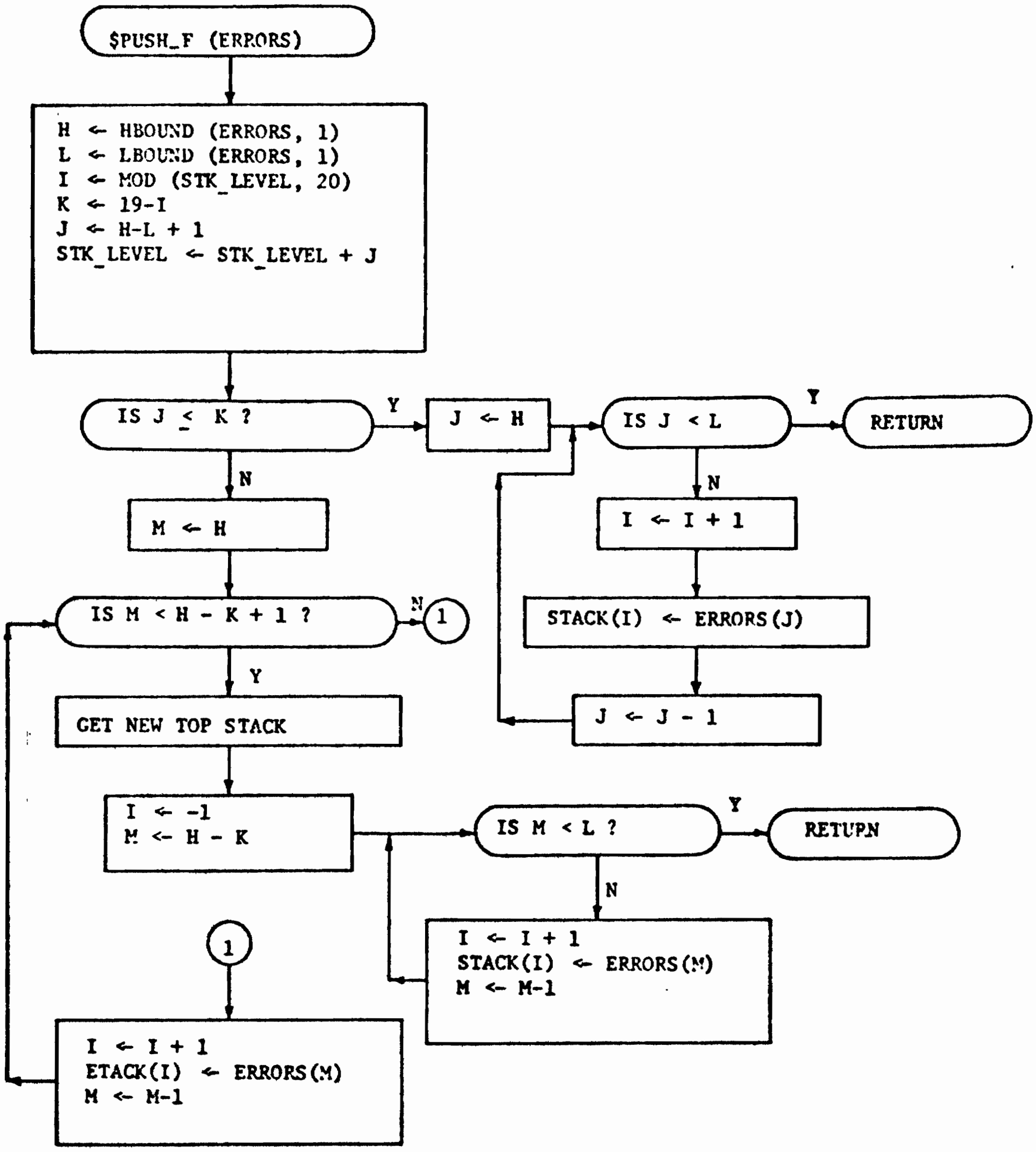


FIGURE 5-3E
$PUSH_F(ERRORS)

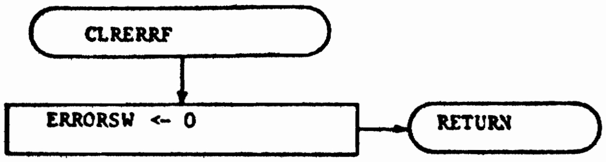


FIGURE 5-3F CLRERRF

5.3.2 Syntactic Supporting Routines

The syntactic supporting routines are routines which are external to the SAP and hand-written by the compiler writer but used by the SAP during syntax analysis. They are designated in EBNF as are all subroutines, i.e., surrounded by "/"'s. The syntactic supporting routines are of two types: (1) error stackers, which are treated as subroutines; and (2) recognizers, which are treated as functions. These routines are described in the following subsections.

5.3.2.1 Error - Stacking Routines

These routines are indicated by subroutine calls embedded in productions. For syntax analysis these routines are called upon entry to a production procedure or after recognition of the first syntactic unit of an optionality group. These routines call on $PUSH_F to push error codes onto the error stack for all the required terminal symbols and recognize routine calls in the production or optionality group. They are usually very simple and brief. In the DDL implementation they have the following general form:

```
< Name  >:  PROCEDURE;
            DCL ERRORS (N) CHAR (6)
            STATIC INIT ('c1', 'c2',..., 'cn');
            CALL $PUSH_F (ERRORS);
            RETURN;
```

```
END < Name > ;
```

Where < Name > is the EBNF subroutine call reference, ERRORS is the name of the array containing the error codes to be stacked, N is the number of error codes to be stacked (in the form of a constant), and "c_i" $1 < i < N$, are the actual error codes enclosed in quotes.

The use of ERROR-stacking routines is best explained by the following example:

The EBNF description for the CONVERT stmt in DDL is: < CONVERT_STMT > ::= CONVERT/DCONV/(< FILE_NAME > INTO < FILE_NAME >);

Thus DDL convert stmts look like:

a) CONVERT (FILE1 INTO FILE2);

b) CONVERT (SOURCE INTO TARGET);

In the EBNF specification of the CONVERT statements, DCONV is the name of the ERROR-stacking routine, which is surrounded by "/"'s. The PL/1 code for DCONV is:

```
DCONV:  PROCEDURE;
        DCL ERRORS(5), CHAR(6) STATIC INIT
        ('CNV-01', 'CNV-02', 'CNV-03, 'CNV-04', 'CNV-05');
        CALL $PUSH_F (ERRORS);
        RETURN;
END DCONV;
```

The meaning of the error codes for the CONVERT statement are:

CNV-01 - Open parenthesis missing after keyword 'CONVERT'.

CNV-02 - Invalid source file name in CONVERT stmt.

CNV-03 - Keyword 'INTO' missing in CONVERT stmt.

CNV-04 - Invalid target file name in CONVERT stmt.

CNV-05 - Closing parenthesis missing in CONVERT stmt.

The complete list of Error codes for the DDL language is given in Appendix A Section of the User Guide.

In addition, calls to initialization routines for internal table creation and maintenance, such as the symbol and data tables, can be made from these routines, although two succesive subroutine calls in the EBNF specification will accomplish the same thing. For example, if DCONVTG (routine to allocate the entry for a convert stmt) were to be called, then insert "CALL DCONVTG;" immediately before the RETURN statement in Procedure DCONV.

However, writing:

```
< CONVERT_STMT > ::= CONVERT/DCONV//DCONVTG/ (< FILE_NAME >
                                  INTO< FILE_NAME >);
```

would produce the same results. Note that EBNF subroutine calls cannot have arguments. If arguments are desired then the first alternative must be used.

5.3.2.2 Recognizer Routines

These routines are indicated by EBNF productions which involve a single subroutine call and nothing else, i.e., productions of the form:

```
< PRODUCTION >::= /Subcall/,
```

where "production" is the name of the production and "subcall" is the name of the routine to be called. For example

< NAME >::= /N_REC/.

These routines are treated as functions which return a bit string value of length 1, representing the logical truth values. These routines are used very often when it would be clumsy to have SAPG-generated code to do the required analysis. For example, analyzing a character string to determine if it is a valid name might require as many iterations as the maximum length of a name. A recognizer routine can analyze it in a single pass and even determine such information as the length of the name being less than a certain limit. In this manner, recognizer routines can be used to speed up the execution of the SAP. The routines are coded by hand and must perform the necessary lexical functions that the generated code of the SAP performs. Upon entry to the routine, LEX must be called to obtain a lexical unit to be analyzed. After the necessary analysis, LEXENAB must be called to enable the lexical routine and a code of '1'B must be returned. Otherwise, '0'B should be returned.

It is also possible for these routines to push error numbers onto the error stack, or even to analyze more than one syntactic unit if the code to accomplish this is written. However, care must be exercised so that all the system conventions are followed.

5.3.3 Error Recovery

As has been pointed out, to produce a SAP, the compiler writer must first express the syntax of the language that the SAP is to analyze in EBNF. In doing this, at appropriate places, subroutines calls should be inserted for error diagnostics and for calling the internal table routines.

The Goal production accepted by SAPG should be of the following form:

```
GOAL ::= end-of-program /done-call/
       | < production-1 >
              .
              .
              .
       | < production-n >
```

Where < production-i >, $1 < i < n$, are productions which define valid statements in the language. "end-of-program" may be a terminal symbol or a production defining the end-of-program statement, although the former will probably be used more frequently. "Done-call" is a subroutine which sets FINSW to indicate the end of syntactic analysis.

To achieve syntax analysis of the remaining statements, after an error has occurred in the statement being analyzed a new goal production should be defined in the following way:

< NEW_GOAL > ::= [< GOAL >/CLRERRF/]*/STMT_FL/< NEW_GOAL >

Thus, the SAP generated by SAPG with the above specification will be able to continue to analyze source statements until the end-of-program is detected and to reinitialize appropriately after detecting an error.

5.3.4 Symbol Table

The symbol table is created during the syntax analysis phase. In DDL, as in most programming languages, it is necessary to connect each occurrence of an < Identifier > (or < NAME >) with its declaration. The DDL compiler accomplishes this connection by means of a symbol table and a data table. The symbol table contains the relevant information about all active < Identifier>s. This includes the name, the line (statement number on which the variable was declared) a count of the references to each < identifier > and a pointer to the data table where the information required for code generation is to be stored (see Section 5.3.5 where a full description of the data table is given).

While most techniques for Data Table construction are Ad-Hoc, there exist many formal methods of Symbol Table creation, three are among the most common ones, they are:

a) Linear Structure

b) Hash Structure and,

c) Tree Structure

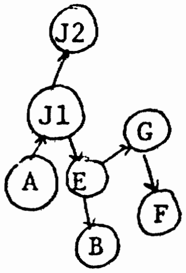
The one used by the DDL compiler is the Tree Structure

5.3.4.1 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 mid-point of the list is unknown and the bisection method cannot 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 a balanced tree, the distance from the root to any node in the tree is in the average $\log_2 n$, where n is the number of nodes in the tree. The nodes contain the information illustrated in Figure 5-3G.

DOWN PTR	UP PTR	DATA TABLE PTR	NO OF REFS	STMT NO	LENGTH OF KEY	KEY IDENTIFIER

All the< intentifier >'s in the subtree extending from a given node which are larger - DDL uses lexicographical ordering, than the key at that node will be in the subtree pointed to by the "upward pointer." Similarly, all smaller keys are in the subtree pointed to by the downward pointer. A subtree can be empty, i.e., if a subtree contains no items, a pointer to that subtree is the null pointer. This tree structure is illustrated in Figure 5-3H.



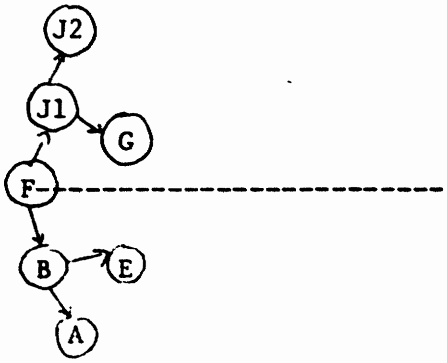
Keys entered in Order A,J1,J2,F,B,G,F

Figure 5-3H

5.3.4.2 Growth and Search Tree Algorithms

Such trees are easy to grow. The first key entered is placed at the root in the tree. 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 key is larger or smaller than the root. This process is repeated at each node being compared against until such a node has a null pointer. The key is then placed by allocating a new node at its point of insertion. The null pointer in the node just compared against is changed to point to the new node just allocated, while the newly allocated node has its up and down pointers set to null.

As an example, consider adding the key "H" to the tree in Figure 5-3I



Keys entered in order F,B,J1,A,E,G,J2

Figure 5-3I (before addition of Key "H").

The root in the above tree is "F".

Step 1: Compare H with F : H>F => move up

Step 2: Compare H with J1 : H<J1 => move down

Step 3: Compare H with G : H>G => move up,

At this point the upward pointer in node "G" is null, therefore a new node is allocated for H. The upward pointer of node G is changed to point to the just allocated node, while the pointers of H are set to null. The new tree structure is given in Figure 5-3J.

Notation

The data items input to the algorithm are stored in the variable LEXBUFF. ST_PTR is a pointer variable which points to the node just allocated. IBEG is a pointer variable which points to the root of the tree. ICUR is a pointer variable pointing to the current node being compared against. DOWN_PTR is the downward pointer and UP_PTR is the upward pointer. SW is a logical variable set initially to 1. KEY_SIZE is a variable where the length of the data item is stored.

Box Labels (Step)	Explanation
1	Is this the first input? If yes go to step 2, otherwise go to step 3.
2	Reset SW and set IBEG to null.
3	Save in KEY_SIZE the length of the data item.

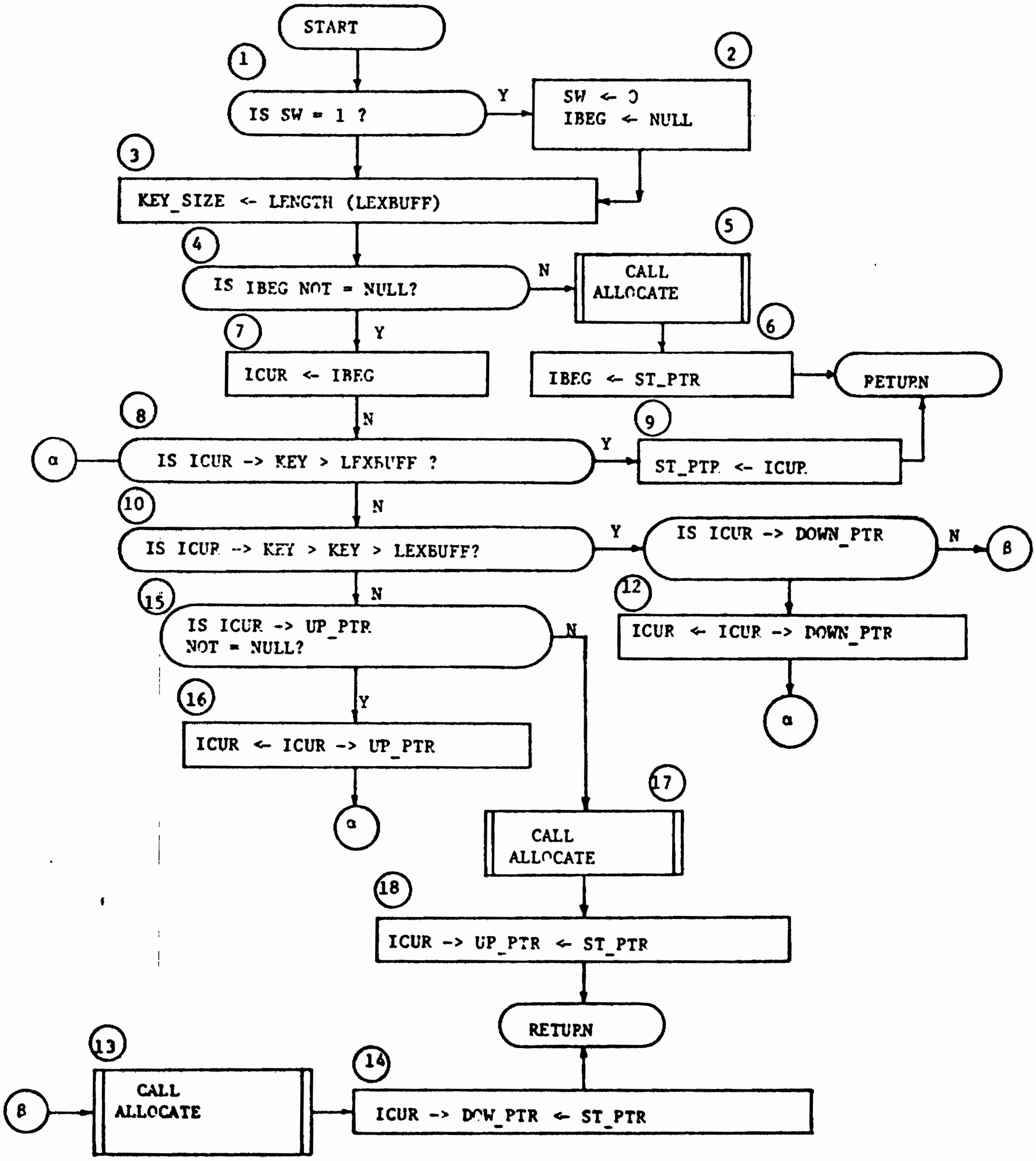


FIGURE 5-3JJ
GROWTH AND SEARCH ALGORITHM

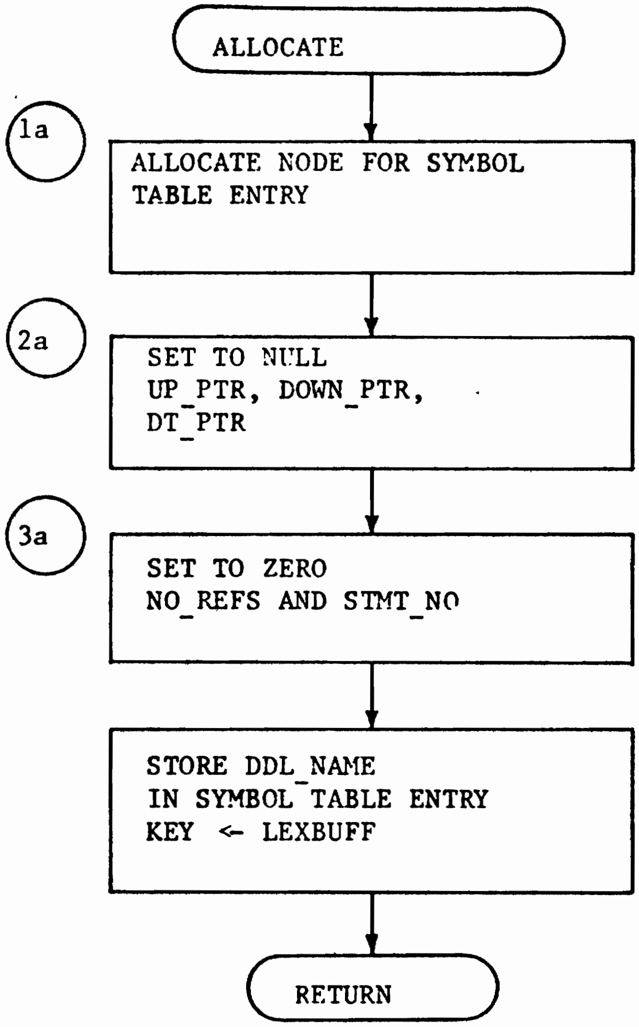

ALLOCATE
1a
ALLOCATE NODE FOR SYMBOL TABLE ENTRY
2a
SET TO NULL UP_PTR, DOWN_PTR, DT_PTR
3a
SET TO ZERO NO_REFS AND STMT_NO
STORE DDL_NAME IN SYMBOL_TABLE ENTRY KEY <- LEXBUFF
RETURN

FIGURE 5-3JJ (cont)

GROWTH AND SEARCH ALGORITHM

Box Labels (Step)	Explanation
4	Is IBEG not equal to null? (i.e., is this not the first input?) then go to step 5, otherwise go to step 7.
5	Call routine to allocate new node.
6	Set IBEG (i.e., root of Tree) to the node allocated in step 5, and return.
7	Set ICUR to point to root of tree i.e., to IBEG.
8	Determine if contents of LEXBUFF is already in tree. If yes go to step 9, otherwise go to step 10.
9	Contents of LEXBUFF is already in the tree ICUR is pointing to such node, set ST_PTR to ICUR and return.
10	Determine if contents of LEXBUFF goes (or is to be found) in lower subtree. If yes go to step 11, otherwise go to step 15.
11	Determine if current node has a lower branch, if so go to step 12, otherwise go to step 13.
12	Set ICUR to point to lower branch of current node and go to step 8.
13	Current node does not have a lower branch, thus call routine to allocate new node and go to step 14.
14	Set down-pointer of current node to point to the node allocated in step 13, and return.
15	Determine if content of LEXBUFF goes (or is to be found) in upper subtree, if yes go to step 16, otherwise go to step 17.

Box Labels (Step)	Explanation
16	Set IWR to point to upper branch of current node and go to step 8.
17	Current node does not have a upper branch, therefore call routine to allocate new node and go to step 18.
18	Set upper-pointer of current node to point to the node allocated in step 17, and return.
1a	In subroutine allocate, allocate space for a new entry in symbol table.
2a	Set upper-pointer, down-pointer, pointer to data table to null.
3a	Set NO_REFS to zero.
4a	Set KEY (in symbol table entry) to contents of LEXBUFF.

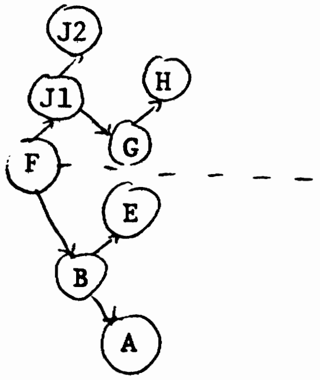
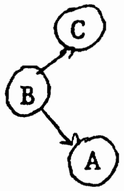


Figure 5-3J (after the key "H" has been added).

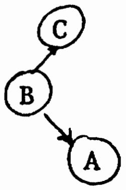
The Growth and Search algorithm is given in Figure 5-3JJ

Now some mathematical properties of the tree structure grown by this algorithm will be considered. The shape of a tree containing a given set of n keys depends on the order in which the items are encountered. For example, Figure 5-3K is formed by inputing the same keys as in Figure 5-3L but in different order. The algorithm thus generates a tree for each of the n! possible arrangements of n keys; but not all the trees are distinct, as can be seen from Figures 5-3K and 5-3L.



Order BAC

Figure 5-3K



Order BCA

Figure 5-3L

Here the assumption that each of the n! permutation of n keys is equi-probable is made. Thus some trees will be generated more often than others.

By using the growth algorithm presented above, one can produce trees which are not balanced; i.e., the upper branch may be heavier than the lower branch or vice-versa. Then if the structure of the tree is unchanged, it can be stated without any contradiction that a key can be searched for, following exactly the same step used to insert that key. An algorithm based on the one designed by W.A. Martins and D.N. Ness [MAR 72] to convert any tree formed using the growth algorithm into a balanced tree is presented in the following subsection.

5.3.4.3 The Tree Restructuring Algorithm

The algorithm which restructures the tree is time consuming.

A natural question to ask is whether the time saved in searching a balanced tree is greater than the time required for the translation from the non-balanced to the balanced tree. Martins and Ness [MAR 72] have shown that the restructuring algorithm, as far as, time is concerned is equivalent to an algorithm in which the tree grows in a balanced manner. Since the number of accesses from the table is not estimated to be high in the present application, however, restructuring will occur only after a certain number of entries have been processed in order to save restructuring time. This number has been arbitrarily placed at twenty for the present implementation.

For the DDL Compiler the restructuring algorithm is called IBEST. This procedure calls on the procedure IGROW which in turn calls INEXT. IBEST returns as its answer, a pointer to the root of the restructured tree. The procedure IGROW(N) is responsible for constructing an optimum tree containing n nodes. IGROW is recursive, as it may call itself. IGROW uses the procedure INEXT, 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 null pointer

2) If n=1 call INEXT and return its result

3) If n > 1

a) Call IGROW ($\lfloor (n-1)/2 \rfloor$)

b) Call INEXT

c) Call IGROW ($\lceil (n-1)/2 \rceil$)

Then it replaces the down pointer of the node pointed by the result of (b), with the result of (a), and its up pointer with the result of (c).

The procedure INEXT 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. Given a sub-tree INEXT returns the nodes in the lower branch by calling itself recursively with this branch as an argument, then it returns the root of the sub-tree, and finally the nodes in the upper branch.

As an example consider that the following items are given to the "Growth and Search Algorithm":
123, 875, 023, 653, 986, 523, 741, 258, 302, 002, 005, 532, 965, 142, 562, 368, 987, 664, 321, 578.

The result is an unbalanced tree where the root is the node containing the "123". The tree which result is shown in Figure 5-3M. Then if we call on the restructuring algorithm, the resulting tree is a balanced one and it is shown in Figure 5-3N.

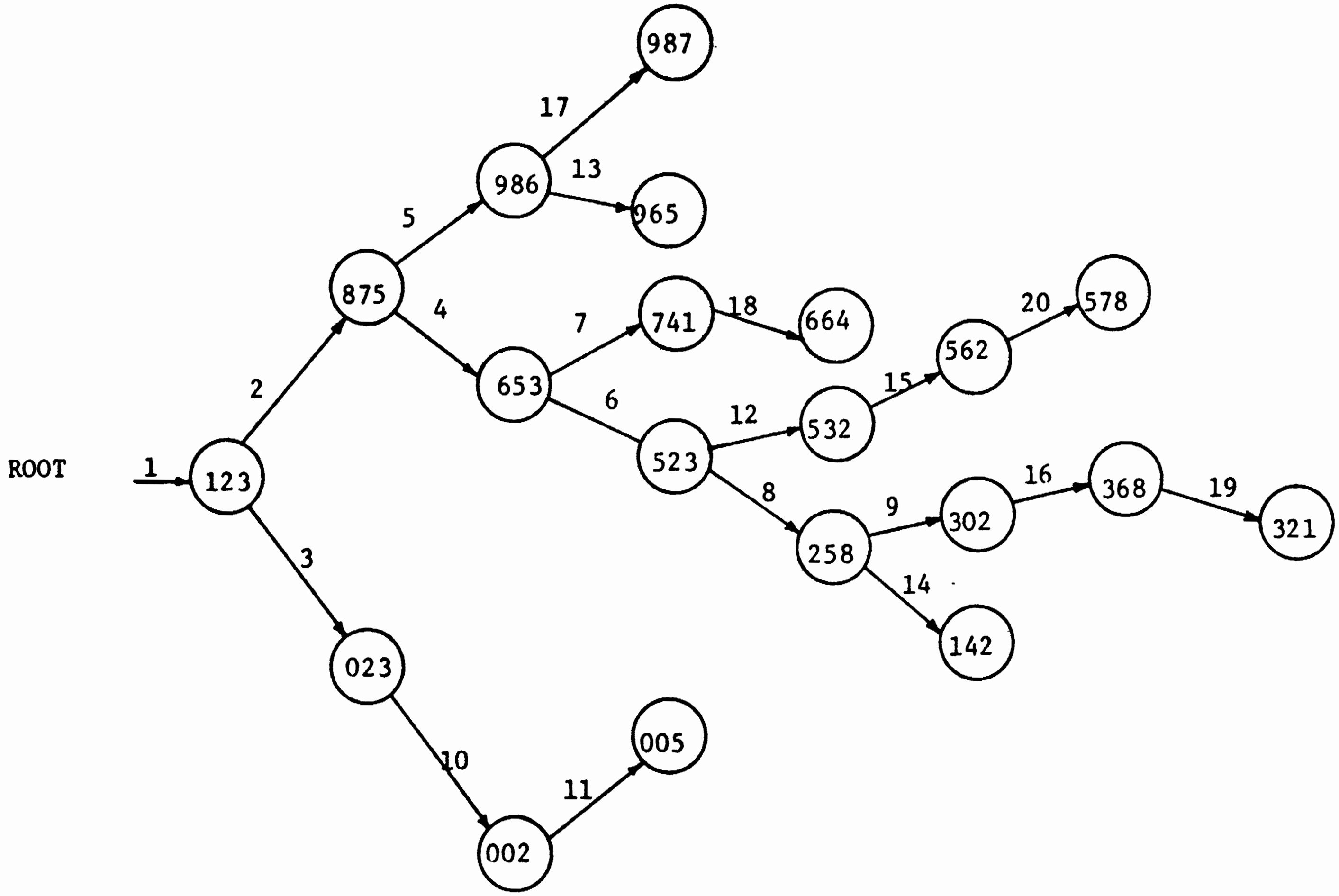


FIGURE 5-3M
UNBALANCED TREE

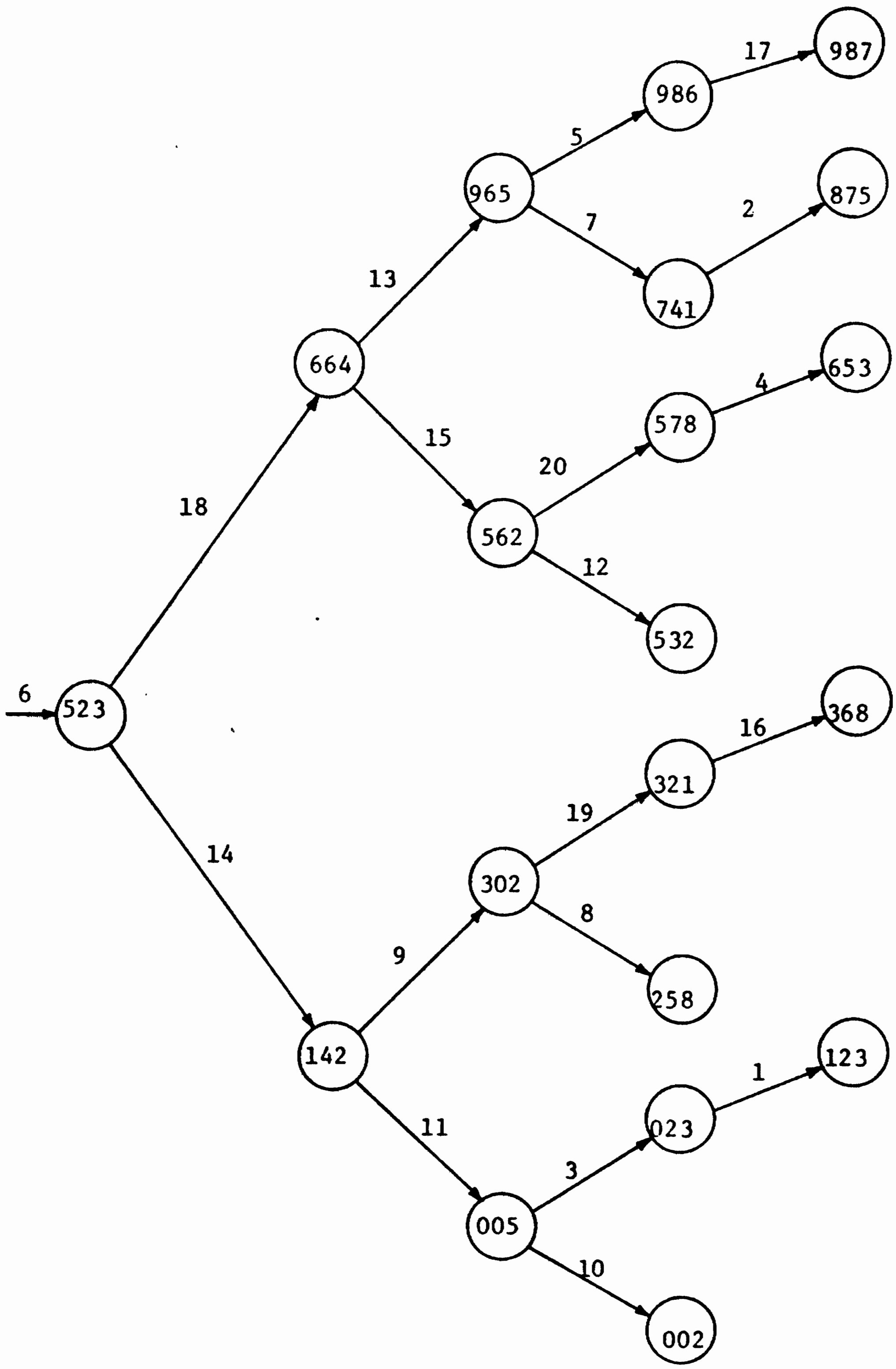


FIGURE 5-3N
BALANCED TREE

5.3.5 Data Table

The data table entries store the information contained in the DDL statements, such information is to be used for global syntax checking,cross-reference and for code generation. To this end, each instance of a DDL source statement initiates procedures which allocate data table entries whose function, as we already mention, is the preservation of the pertinent information. The format of the DDL data table entries are by no means unique and, given different compiler writers, different designs for these entries most likely would be conceived. As long as the entries generated by the internal routines correspond exactly to the tables expected by the code generation phase, global syntax checking and cross reference, any applicable construct may be used.

The format of the data table entries for all the DDL statements is given in Appendix D. For purposes of a better communication in the mechanics of table creation we present here a few of them.

Example (a)

Take for example the CONVERT statement, its EBNF description is :

< CONVERT_STMT > ::= CONVERT (< FILE_NAME > INTO <FILE_NAME>)

the corresponding entry in the data table for this statement is:

Stmt Type	Pointer to Target File	Pointer to Source File	Stmt Number

The PL/1 declaration for this entry is

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

where:

1) TYPE is a fixed binary field (two bytes) where the corresponding code for CONVERT is to be stored.

2) TARGET is a pointer (1 word) which points to the symbol table where the name of the Target File is stored.

3) SOURCE is a pointer (1 word), which points to the symbol table where the name of the Source File is stored.

4) STMT_NO is a fixed binary field (2 bytes) were the statement number of the CONVERT stmt in the DDL program is to be stored.

Example (b):

The FILE_STMT is described in EBNF as follows:

```
< FILE_STMT > ::= FILE (< RECORD_NAME > [,CHAR_CODE =  < CODE >
                                            ,STORAGE = < NAME > );
```

The format for the data table corresponding to the FILE stmt is:

STMT TYPE	POINTER TO FILE NAME IN SYMBOL TABLE	POINTER TO RECORD NAME IN SYMBOL TABLE	POINTER TO STORAGE NAME IN SYMBOL TABLE	BUFOFF OPTION	CHAR CODE

The PL/1 declaration for the above entry is:

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

where;

1) TYPE is a fixed binary field where the corresponding code for FILE is to be stored.

2) SYM is a pointer, which points to the Symbol Table entry where the name of the FILE stmt is kept.

3) RCD_NAME is a pointer, which points to the Symbol Table entry where the name of the Record referred in the FILE stmt is kept.

4) BUFOFF is a fixed binary field used when the character code is ASCII and the user specified the BUFOFF option.

5) CHAR_CODE is a fixed binary field where the code corresponding to EBCDIC, ASCII or BCD is to be stored.

Example 3.

The RECORD STMT is described in EBNF as:

```
< RECORD_STMT > ::=  RECORD (< NAME_LIST > [, < NAME_LIST >]*
                              [,LOCK =  < PROC_CALL >]
```

```
                                [,SIZE = < USER_RECORD_FORMAT > ]);

< NAME_LIST   ::=  < MEMBER_NAME > [  < OCCURRENCES > ]
```

The format for the Record entry in the Data table is

STMT TYPE	POINTER TO RECORD NAME IN SYMBOL TABLE	LOCK INFORMATION	SIZE INFORMATION	NO. OF MEMBERS	MEMBER 1 INFORMATION	..
					...	MEMBER n INFORMATION

For the purpose of our present discussion we will present only a subset of the RECORD stmt in the PL/1 declaration, but the user is referred to Appendix D where the full description of the RECORD entry is given. The PL/1 description for a subset of the RECORD stmt is:

```
     DCL 1  RECORD BASED (PTR)

            2 TYPE    FIXED BIN,

            2 SYM     POINTER

            ( Lock and Size
            (
            ( Declarations

            2 NO_MEMS   FIXED BIN

            2 MEMBERS (NDUMMY REFER (RECORD.NO_MEMS),

                 3 MEM_NAME POINTER,

                 ( rest of declaration
                 (
                 ( for information          ;
                 (
                 ( about member
```

where;

1) TYPE is a fixed binary field where the corresponding code for RECORD is to be stored.

2) SYM is a pointer, which points to the symbol table entry where the name of RECORD stmt is kept.

3) NO_MEMS is a fixed binary field where the number of members described in the RECORD stmt is kept.

4) MEMBERS is a substructure in the RECORD structure where the information about the members is kept.

5) MEM_PTR is a pointer, which points to the symbol table entry where the name of the corresponding member is kept.

5.3.6 Mechanics of Table Generation

In Section 3.5 we introduced EBNF/WSC (Extended Backus-Normal-form with subroutine calls), using it the syntactic structure of the DDL is described as well as the subroutines that are to be called upon successfull recognition of a syntactic unit. In Appendix E we present the definition of DDL in EBNF/WSC.

In describing DDL in EBNF/WSC, at appropriate points, the names of the PL/1 routines are given. Such routines will build the Symbol Table and the Data Table entries. Subroutine calls are made only if everything specified by the EBNF/WSC production in which the subroutine call appears has been successfully recognized up to the point of the subroutine call.

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

Example 1

```
STMT_NO 1     RC1 IS RECORD (MEM1,  MEM2);
        2         MEM1 IS GROUP (MEM11,  MEM12);
        3              MEM11 IS FIELD (CHAR(2));
        4              MEM12 IS FIELD (NUM_PICTURE = '99');
        5         MEM12  IS FIELD (BIT(8));
```

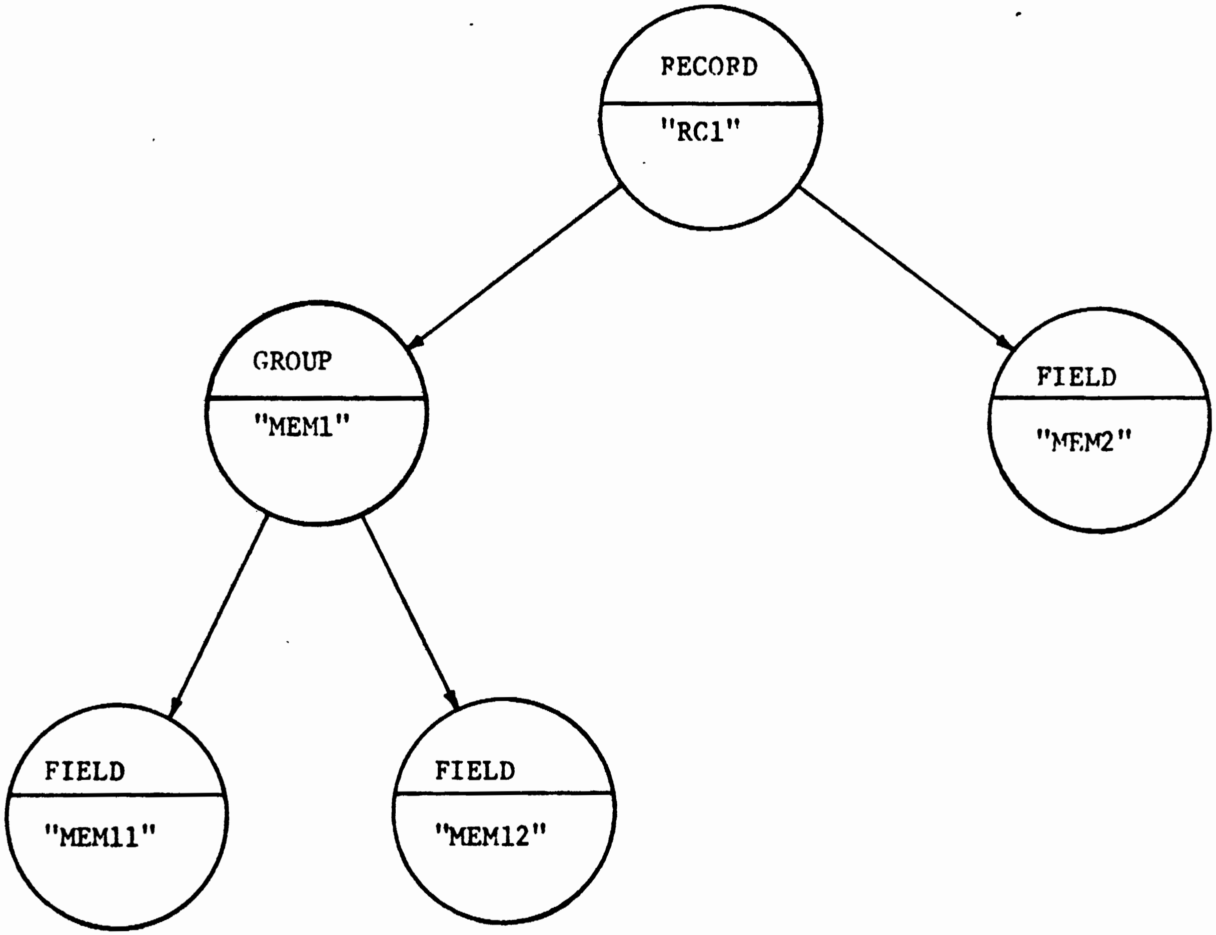
Statements in DDL are generally described in EBNF/WSC in the following form (see Section 3.3, 3.4 and 3.5);

< NAME >IS < DDL_STATEMENT > , therefore to enter the names RC1, MEM1, MEM11, MEM12 and MEM2 (which are the names of the DDL stmts) into the symbol table, a routine called ENTESYM is called after recognition of the < name >. Thus the EBNF/WSC for the DDL statements is:

< NAME >/ENTESYM/ IS < DDL_STATEMENT > ;

The subroutine ENTESYM returns a pointer to the location (in the symbol table) where the < name > is stored if the name was already in the symbol table, if not in the symbol table an entry for that name is made (in the symbol table) and the up-pointer, down pointer, DT-pointer are set to null, similarly the number of references and the statement number are set to zero. In order to store in the symbol table just allocated the DDL statement number the subroutine STMTENT is called, to this end the EBNF/WSC for the DDL statement is coded as:

< NAME >/ENTESYM/IS /STMTENT/ < DDL_STATEMENT >;



TREE STRUCTURE
CORRESPONDING TO DDL
EXAMPLE 1 IN SECTION 5.3.6

FIGURE 5-3P

The foregoing DDL statements in Example 1 denote the tree structure Record RC1 shown in Figure 5-3P.

In order to preserve such structure, the DDL compiler will allocate individual data tables, unique for each source statement, and the DDL compiler will link them in such a way that the intended structure is preserved.

Following with our discussion of the mechanics of table generation, let us now concentrate in the Data table entries. In general < DDL_STATEMENT > can be described in EBNF/WSC as follows:

< DDL_STATEMENT > ::=   STMT_TYPE   (< STMT_INFO >);

where STMT_TYPE can be RECORD, FILE, GROUP etc. And < STMT_INFO > is the EBNF/WSC description of the syntax of the statement, along with the names of the subroutine to be called.

In order to allocate the appropriate entry in the Data table for the DDL statement being processed in all of the DDL statements, (with the exception of RECORD, GROUP and SCAN; later in this Section we will explain how these statements are treated) after the STMT_TYPE has been successfully recognized a subroutine call is made to perform the allocation of the appropriate entry. In general the EBNF/WSC for the DDL statements (up to this point in our discussion) will look like

< DDL_STATEMENT > ::= STMT_TYPE/ Name of subroutine to allocate
entry/ (< STMT_INFO>);

Let us take once more the CONVERT statement, its EBNF/WSC is:

< CONVERT_STMT > ::= CONVERT/DCONV/(< STMT_INFO >);

The subroutine DCONV will perform the following tasks:

1) Push onto the error stack the error codes for all the terminal symbols for the convert statement, see Section 5.3.2.1.

2) Call on DCONVTG which is the routine that will allocate the CONVERT data table entry. After the allocation has been made the TYPE is set to CONVERT and the STMT_NUMBER is set to contain the current stmt numbers provided by LEX.

Let us now take the FILE statement, its EBNF/WSC is:

< FILE_STMT > ::= FILE/DFILE/(< STMT_INFO >);

The subroutine DFILE will perform the following tasks:

1) Push onto the error stack the error codes for all terminal symbols for the FILE statement.

2) Call on DFILETG which is the routine that will allocate the FILE data table entry.

DFILETG after the allocation for the FILE data table entry has been made will set TYPE to FILE, and the SYM to point to the symbol table entry where the < name> of the FILE statement is kept. Also it will set all the parameters of the FILE statements with the default values. Finally, DT_PTR in the symbol table entry is set to point to the data table entry just allocated.

Up to this point we have explained how < NAMES > are entered into the symbol table and the appropriate data tables entries are allocated. In the following paragraphs we will explain how the < STMT_INFO > is encoded in the data tables entries.

Consider once more the CONVERT statements, its complete EBNF/WSC is:

< CONVERT_STMT > ::= CONVERT/DCONV/(< STMT_INFO >);

where:

< STMT_INFO > ::=< FILE_NAME > /CS/ INTO < FILE_NAME > /CT/

CS is the routine which will call on ENTESYM if the < FILE_NAME> was successfully recognized to enter such name in the symbol table. And it will call in a routine INCR_REF to increment the number of times the name is being referred in a DDL statement. Finally we will set the SOURCE field in the CONVERT entry to point the symbol table where the< FILE_NAME > is stored.

CT performs similar tasks as CS, the difference is that it set the TARGET field in the CONVERT entry to point the symbol table where the name is stored.

Let us now take the GROUP statement.

< GROUP_STMT > ::= GROUP/DGROUP/(< STMT_INFO >);

< STMT_INFO > ::= (< NAME_LIST> /NL_G/[, < NAME_LIST > /NL_G]*)
/ALL 0_G/

In a previous paragraph we mention that after the successful recognization of the STMT_TYPE a routine will be called to allocate the appropriate data table entry. We mention that the exceptions are the RECORD, GROUP and SCAN statements. The reason is that these statements can contain variable information, and the goal is to allocate just the necessary core to encode the information in that particular occurrence of the statement. Therefore control storage will be used to save the information for each member in the GROUP, RECORD or SCAN statement. A counter of the member of members found in the statement is kept and after the last member is processed then the allocation of the appropriate entry is made, since then we do know how much core to allocate. Then the information saved in the controlled storage is stored into the data table entry and the controlled storage is freed.

In the above specification of the GROUP statement the routine NL_G will store in the controlled storage all the pertinent information of the member. NL_G will call on ENTESYM to enter the < NAME > of the member into the symbol table and also will call on INCR_REF to increment the number of times this < NAME > has been referred in the DDL statements. The routine ALLO_G is called after the last member has been processed and it allocates the entry for the Group statement, set its TYPE to GROUP and moves from the controlled storage into the entry just allocated the information of all its members, then the controlled storage is freed and the DT_PTR entry in the symbol table corresponding to the < name > of the GROUP statement is set to point to the data table just allocated. Finally the SYM field is set to point to the symbol table where the < NAME> of the GROUP statement is stored.

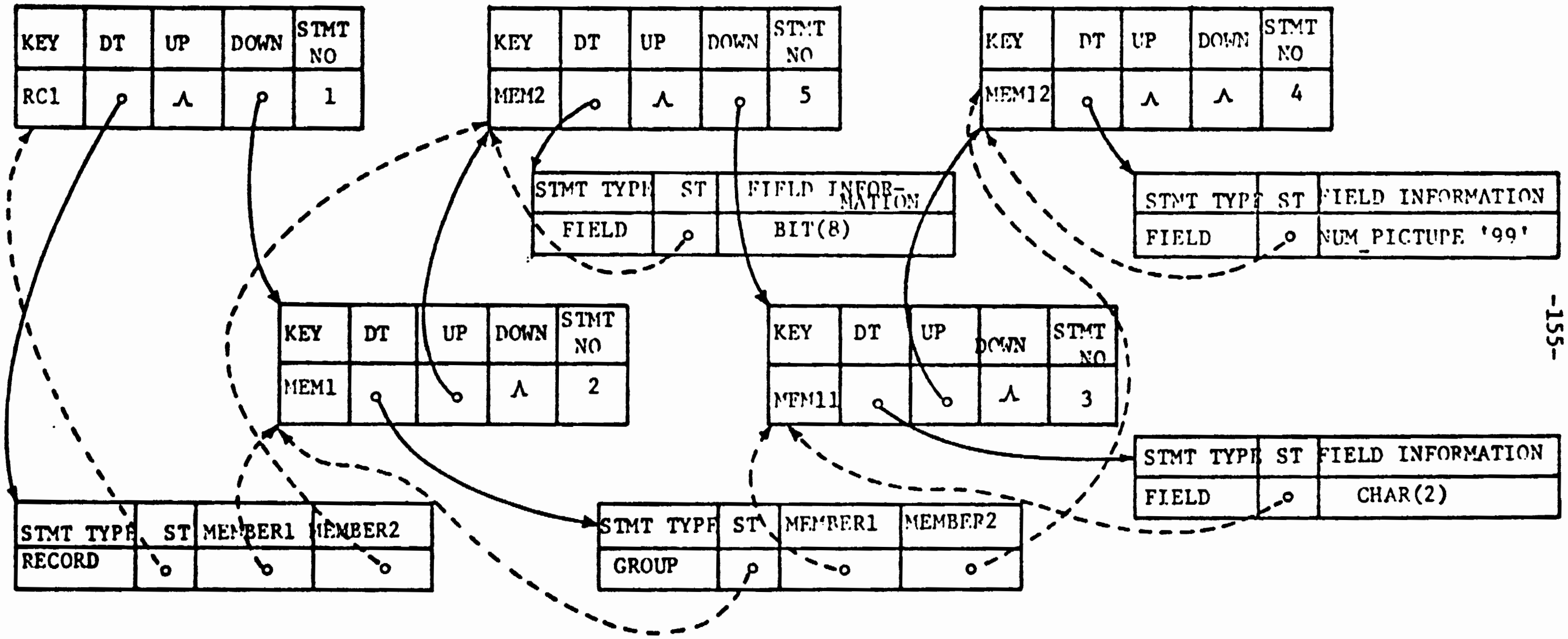
To illustrate how the link between Symbol Table entries and Data Table entries we refer the reader to Example 1 which was:

```
STMT 1   RC1 IS RECORD (MEM1,  MEM2);
STMT 2       MEM1 IS GROUP (MEM11,  MEM12);
STMT 3            MEM11 IS FIELD (CHAR(2));
STMT 4            MEM12 IS FIELD (NUM_PICTURE = '99');
STMT 5       MEM2 IS FIELD (BIT(8));
```

The corresponding Symbol Table entries an Data Table entries for Example 1 are given in Figure 5-3Q.

5.4 Cross-reference and Global Syntax Checking

When the XREF option was specified to the DDL compiler, a cross-reference table that lists all the identifiers (DDL-Names) and its attributes in the Source program with the numbers of the source statements in which they appear is generated. The XREF option also requests the listing of all the calls to DML procedures made in the DDL source statements. The XREF procedure, see Section 5.4.1 is called only if no local syntax errors were detected during the syntax analysis phase.



KEY	DT	UP	DOWN	STMT NO
RC1	○	Λ	○	1

KEY	DT	UP	DOWN	STMT NO
MEM2	○	Λ	○	5

KEY	DT	UP	DOWN	STMT NO
MEM12	○	Λ	Λ	4

STMT TYPE	ST	FIELD INFORMATION
FIELD	○	BIT(8)

STMT TYPE	ST	FIELD INFORMATION
FIELD	○	NUM_PICTURE '99'

KEY	DT	UP	DOWN	STMT NO
MEM1	○	○	Λ	2

KEY	DT	UP	DOWN	STMT NO
MEM11	○	○	Λ	3

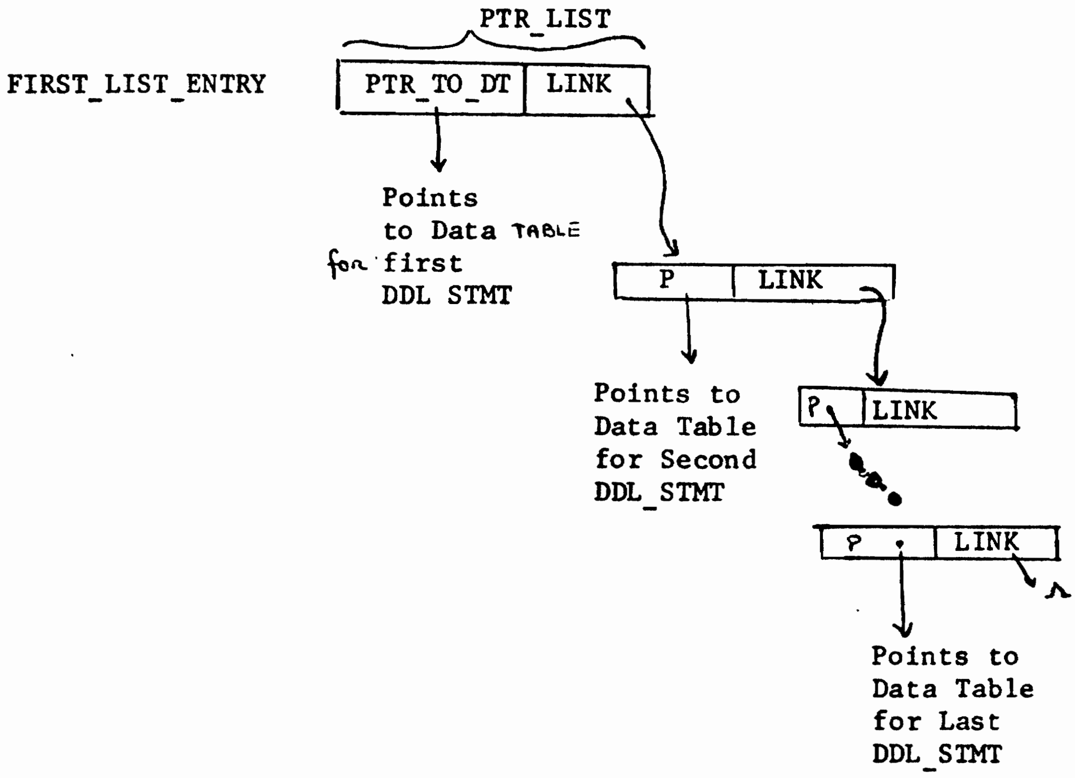
STMT TYPE	ST	FIELD INFORMATION
FIELD	○	CHAR(2)

STMT TYPE	ST	MEMBER1	MEMBER2
RECORD	○	○	○

STMT TYPE	ST	MEMBER1	MEMBER2
GROUP	○	○	○

FIGURE 5-30
SYMBOL AND DATA TABLES

During the syntax analysis phase a linked list (PTR_LIST) of pointers to the Data Table entries for each DDL statement is form, this is done by calling DT_PTR_LIST_LINK every time a Data Table has been allocated for a DDL statement. The linked list that DT_PTR_LIST_LINK forms have the following general form:


PTR_LIST
FIRST_LIST_ENTRY
PTR_TO_DT
LINK
Points to Data TABLE for first DDL STMT
P
LINK
Points to Data Table for Second DDL_STMT
P
LINK
P
LINK
Points to Data Table for Last DDL_STMT

When DT_PTR_LINK_LIST (see Figure 5-4) is called the External Pointer Variable DTPTR is pointing to the Data Table entry corresponding to the current DDL statement being processed. Upon entry to this procedure the item PTR_LIST which consist of two elements, the first one PTR_TO_DT is a pointer where the address of the Data Table corresponding to the current statement is to be stored, the second element in PTR_LIST is LINK which is also a pointer variable where the

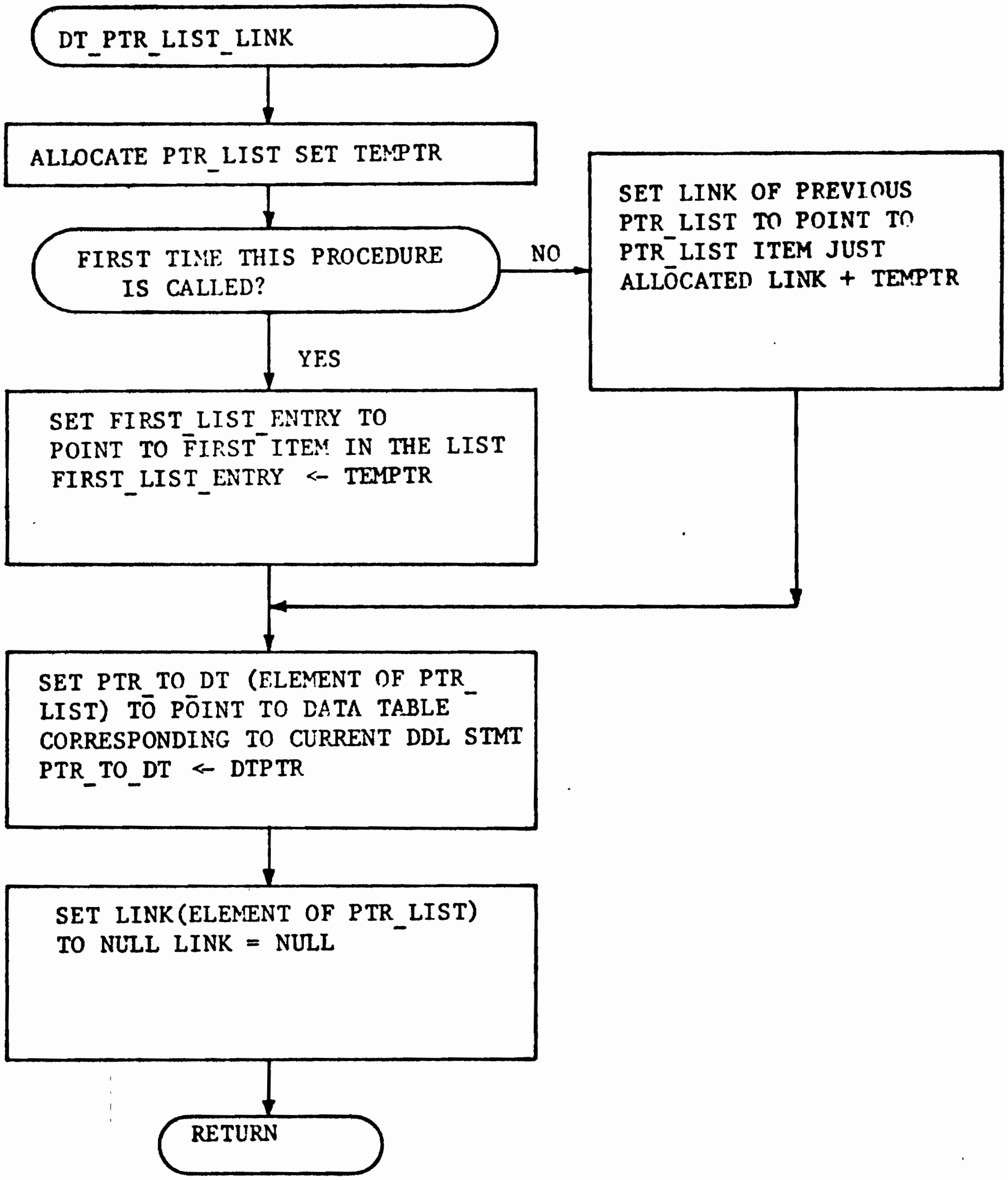

DT_PTR_LIST_LINK
ALLOCATE PTR_LIST SET TEMPTR
FIRST TIME THIS PROCEDURE IS CALLED?
NO
SET LINK OF PREVIOUS PTR_LIST TO POINT TO PTR_LIST ITEM JUST ALLOCATED LINK + TEMPTR
YES
SET FIRST_LIST_ENTRY TO POINT TO FIRST_ITEM IN THE LIST FIRST_LIST_ENTRY <- TEMPTR
SET PTR_TO_DT (ELEMENT OF PTR_ LIST) TO POINT TO DATA TABLE CORRESPONDING TO CURRENT DDL STMT PTR_TO_DT <- DTPTR
SET LINK(ELEMENT OF PTR_LIST) TO NULL LINK = NULL
RETURN

FIGURE 5-4
DT_PTR_LIST_LINK

address of the next item of the list is to be stored, LINK is initially set to NULL. After the allocation of PTR_LIST takes place a test is made to determine if this is the first time DT_PTR_LINK_LIST is being called, if so FIRST_LIST_ENTRY which is declared as a external pointer variable is set to point to the PTR_LIST just allocated (which happens to be the first element of the list). Then PTR_TO_DT is set to DTRTR and LINK is set to NULL. If this is so the first time DT_PTR_LINK_LIST is being called then the Link of the previous PTR_LIST is set to the item (PTR_LIST) just allocated and LINK is set to NULL.

5.4.1 XREF

The external variables that the procedure XREF uses are the following:

- MAX_REFS -- where the maximum number of times a DDL_NAME was referenced is kept
- FIRST_LIST_ENTRY -- pointer to the first element of the list of PTR_LIST
- STMT_NO -- variable whose value is the number of DDL statements in the program being compiled.

Since one of the jobs of XREF is to produce a list of DML routines being called from DDL statements, XREF allocates three different structures to keep this information, the first one FLD_LENGTHS is where all the names of DML procedures specified by the user as the length of some fields are to be

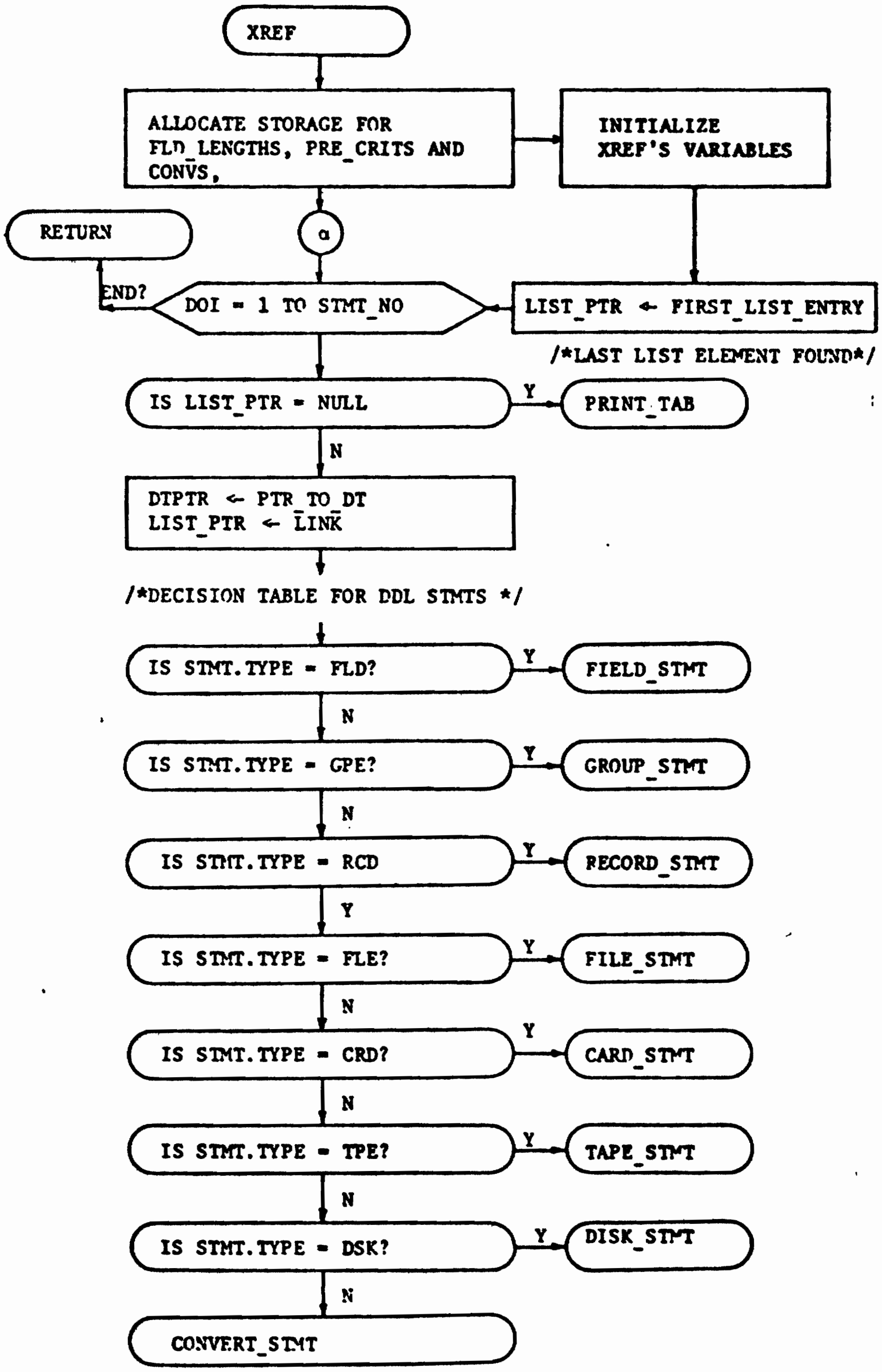


FIGURE 5-4A
XREF

stored, the second one PRE_CRITS is where all the names of the DML routines specified in DDL as PRE_CRITS procedures are stored, and the third one CONVS is where all the names of DDL routines specified in DDL as conversion routines are stored.

After the above allocation takes place XREF (see Figure 5-4A) uses the external variable FIRST_LIST_ENTRY to get the first element of the list of pointers to the Data Table where the information of the DDL statements have been encoded. The pointer variable LIST_PTR is set to FIRST_LIST_PTR and a loop is initiated to process all the Data Table entries. When LIST_PTR is NULL indicates that all the Data Tables have been processed by XREF and the procedure PRINT_TAB is called (see Section 5.4.2).

If LIST_PTR is not NULL then DTPTR (pointer on which the Data Tables entries are based) is set to PTR_TO_DT and LIST_PTR is set to LINK i.e., LIST_PTR is now pointing to next element in the list of pointer to Data Tables.

Next a check is made to determine the TYPE of Data Table entry, i.e., if Data Table corresponds to a CONVERT, RECORD, GROUP etc., and an appropriate branch to a label where such statement is to be processed takes place.

5.4.1.1 CONVERT_STMT Entry (see Figure 5-4B)

Since CONVERT_STMT is a Command statement i.e., there is no name associated to it, the DDL compiler keeps the statement number in the Data Table rather than in the Symbol Table

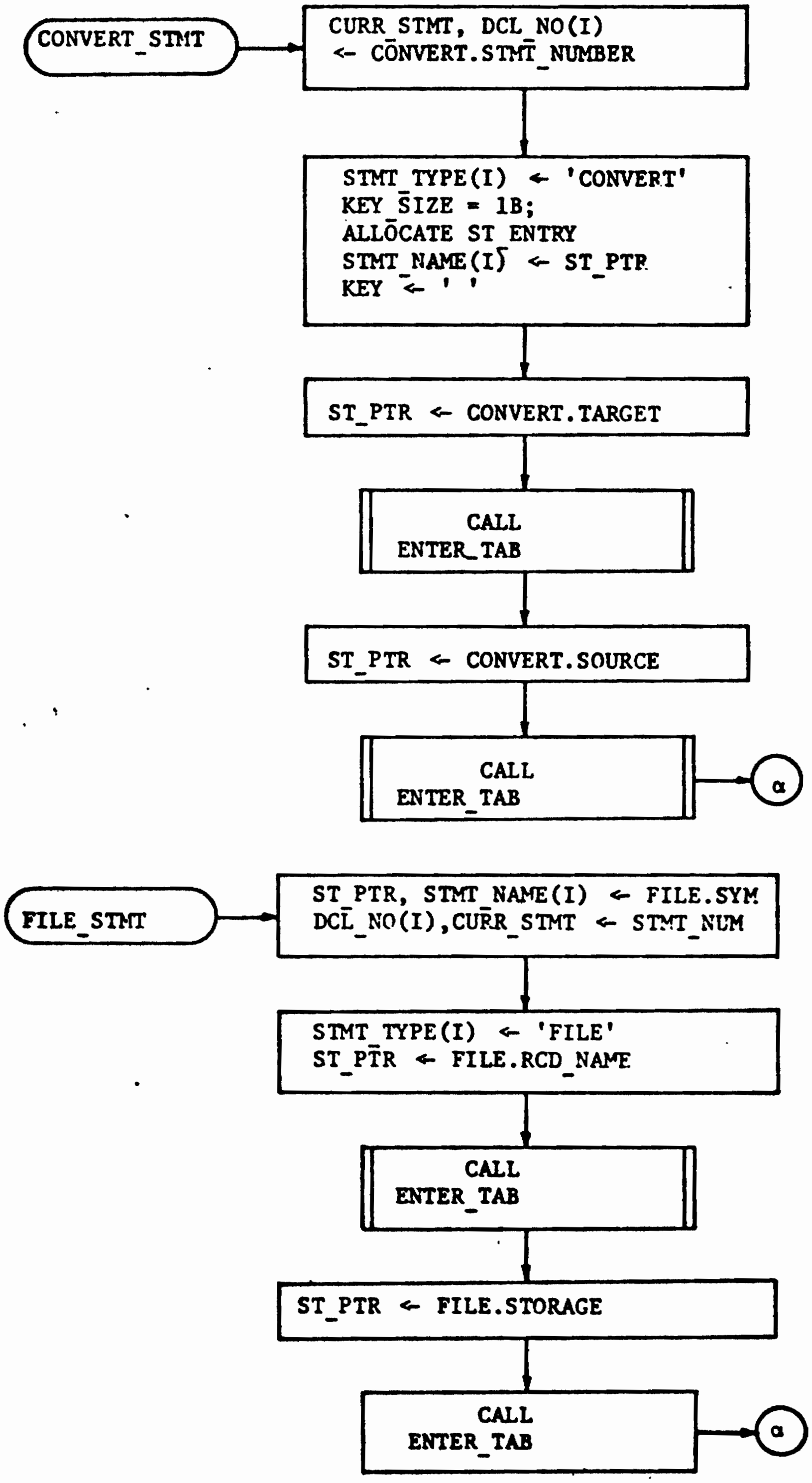

CONVERT_STMT
CURR_STMT, DCL_NO(I)
<- CONVERT.STMT_NUMBER
STMT_TYPE(I) <- 'CONVERT'
KEY_SIZE = 1B;
ALLOCATE ST_ENTRY
STMT_NAME(I) <- ST_PTR
KEY <- ' '
ST_PTR <- CONVERT.TARGET
CALL
ENTER_TAB
ST_PTR <- CONVERT.SOURCE
CALL
ENTER_TAB
α
FILE_STMT
ST_PTR, STMT_NAME(I) <- FILE.SYM
DCL_NO(I),CURR_STMT <- STMT_NUM
STMT_TYPE(I) <- 'FILE'
ST_PTR <- FILE.RCD_NAME
CALL
ENTER_TAB
ST_PTR <- FILE.STORAGE
CALL
ENTER_TAB
α

FIGURE 5-4B

where the statement number is kept for the rest of the DDL statements.

When control is passed to the entry, CURR_STMT (which is an internal variable where the statement number of the statement currently being processed is stored) and DCL_NO(I) are set to the statement number in the CONVERT Data Table entry. STMT_TYPE (I) is set to the character string 'CONVERT'. And a Symbol Table entry for the CONVERT statement is allocated. Then ST_PTR (which is a pointer variable on which the Symbol Table entries are based) is set to CONVERT TARGET. And ENTER_TAB (see Section 5.4.3) is called to enter this reference in the cross-reference table. Next ST_PTR is set to CONVERT_SOURCE and ENTER_TAB is called. Then control is passed to the loop where the next Data Table is to be processed.

5.4.1.2 FILE_STMT Entry (see Figure 5-4B)

As we pointed out in the preceeding Section, the statement numbers for all the DDL Statements but the CONVERT statement are kept in the corresponding Symbol Table entry where the name of the DDL Statement is stored. Now when control is passed to this entry point, ST_PTR and STMT_NAME(I) are set to File SYM (pointer to Symbol Table where name of File stmt is stored). And DCL_NO(I) and CURR_STMT are set to STMT_NUM. The variable STMT_TYPE (I) is set to the character string 'FILE'. The pointer to the record name is stored in ST_PTR and ENTER_TAB is called (see Section 5.4.3) to enter reference to Record statement into cross-reference table. Next the pointer to storage name is stored in ST_PTR and ENTER_TAB is called to enter reference to STORAGE stmt (CARD, TAPE or DISK) into the cross-reference table. Finally control is returned to the loop were the next Data Table is to be processed.

5.4.1.3 CARD_STMT entry. (See Figure 5-4C)

First ST_PTR and STMT_NAME(I) are set to CARD.SYM (where the pointer to the Symbol table entry for the name of the CARD statement is stored). Next DCL_NO(I) is set to STMT_NUM and STMT_TYPE (I) is set to the character string 'CARD'. Then control is passed to the loop were the next Data Table is to be processed.

5.4.1.4 TAPE_STMT Entry. (See Figure 5-4C)

The actions taken by this entry are similar to those of the CARD_STMT entry the difference is that STMT_TYPE (I) is set to the character string 'TAPE'.

5.4.1.5 DISK_STMT Entry (See Figure 5-4C)

In this case STMT_TYPE (I) is set to the character string 'DISK' and the rest of variables are set in a similar manner as those in the above two entries.

5.4.1.6 RECORD_STMT Entry (See Figure 5-4C)

In this entry ST_PTR and STMT_NAME(I) are set to RECORD.SYM

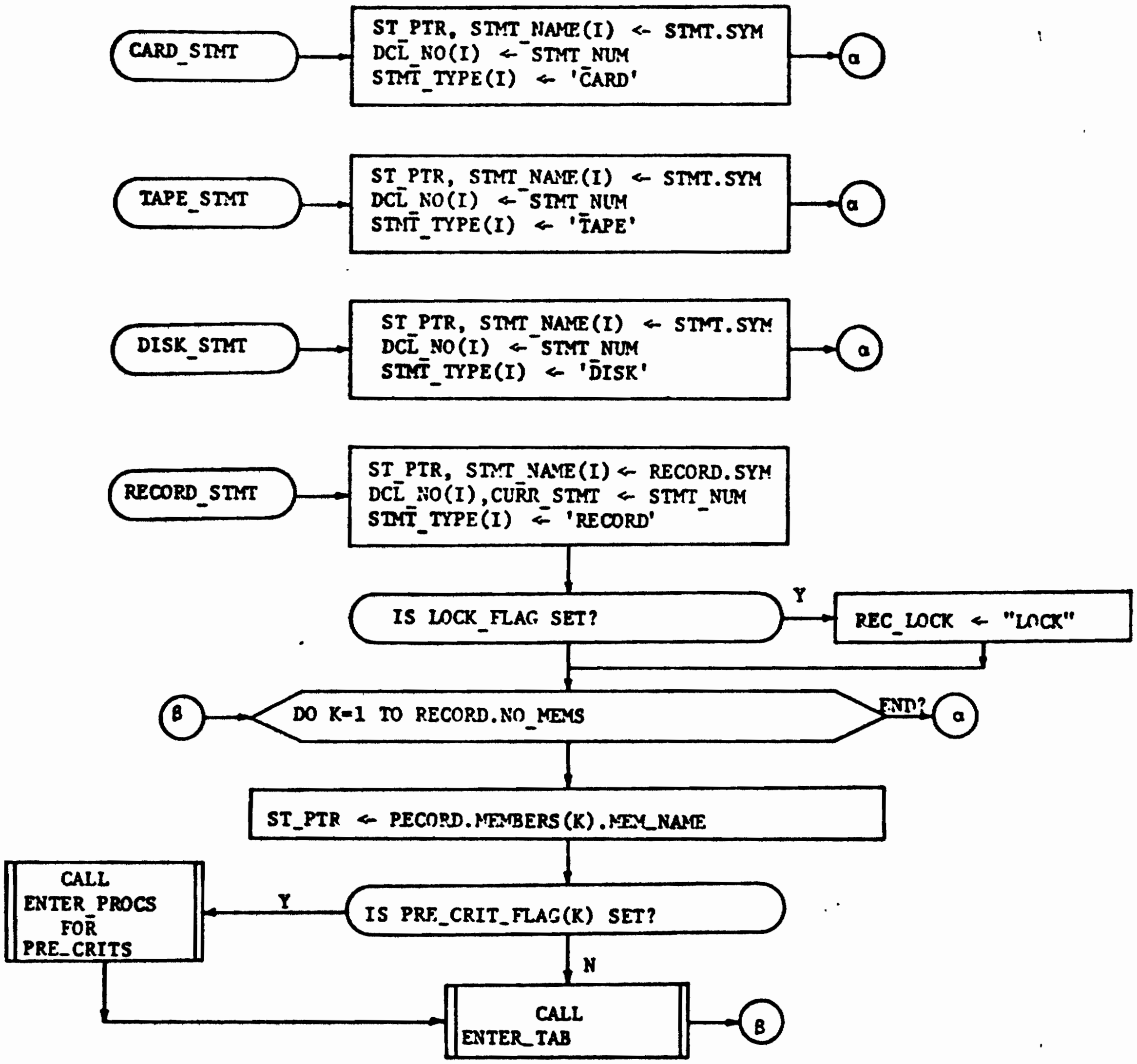


FIGURE 5-4C
CARD_STMT, TAPE_STMT,
DISK_STMT, RECORD_STMT

(pointer to Symbol Table entry where the name of RECORD stmt is stored), DCL_NO(I) and CURR_STMT are set to STMT_NUM, and STMT_TYPE(I) is set to the character string 'RECORD'.

Next if the LOCK_FLAG is set the name of the DML routine is saved in REC_LOCK. Then a loop to process all the members in the RECORD statement is initiated. ST_PTR is set to point to Symbol Table entry of current member of RECORD and if the PRE_CRIT_FLAG for this member is set ENTER_PROCS (see Section 5.4.4) for PRE_CRITS is called. Next ENTER_TAB is called to store reference to GROUP and/or FIELD into cross-reference table. After all the members of the RECORD statement have been processed control is returned to loop where the next Data Table is to be processed.

5.4.1.7 GROUP_STMT Entry (See Figure 5-4D)

In this entry ST_PTR and STMT_NAME(I) are set to GROUP.SYM (pointer to Symbol Table where the name of GROUP statement is kept), DCL_NO(I) and CURR_STMT are set to STMT_NUM and STMT_TYPE(I) is set to the character string 'GROUP'. Next a loop to process all the members of current GROUP is initiated. ST_PTR is set to point to Symbol Table entry of current member of GROUP, and if the PRE_CRIT_FLAG is set then ENTER_PROCS for PRE_CRITS is called. To store the reference to GROUP and/or FIELD, ENTER_TAB is called. Finally after all the members of GROUP have been processed control is transferred to loop where next Data Table entry is to be processed.

5.4.1.8 FIELD_STMT Entry (See Figure 5-4D)

When control is passed to the FIELD_STMT entry, DTPTR1 is set to a continuation of the FIELD Data Table where the FIELD description is stored. Next ST_PTR and STMT_NAME(I) are set to FIELD.SYM (pointer to Symbol Table entry where the name of the FIELD statement is stored), DCL_NO(I) and CURR_STMT are set to STMT_NUM, and STMT_TYPE(I) is set to the character string 'FIELD'.

If the conversion flag is set then ENTER_PROCS (see Section 5.4.3) for DML conversion routines is called. Next if the FIELD statement has been specified with the NUM_PICTURE attribute and no mapping has been specified control is passed to the loop where the next Data Table is to be processed. If a mapping has been specified DTPTR3 is set to point to the description of such mapping and control is passed to the TEST_SOURCE_NAME entry (see Figure 5-4D).

If the attribute of the FIELD statement is CHAR_PICTURE, CHAR or BIT a test is made to determine if the length of the FIELD is given as an integer, *, parameter statement (DDL_LENGTH or DDL_COUNT), reference name or a DML procedure. If length is an integer or * control is passed to the GET_ASSG entry (see Section 5.4.1.8.1). If length is specified via a DML procedure ENTER_PROCS (see Section 5.4.3) for FLD_LENGTHS is called and control then passed to the GET_ASSG entry. If the length is given as a parameter statement DTPTR is set to point to the argument of the parameter statement i.e. reference Name

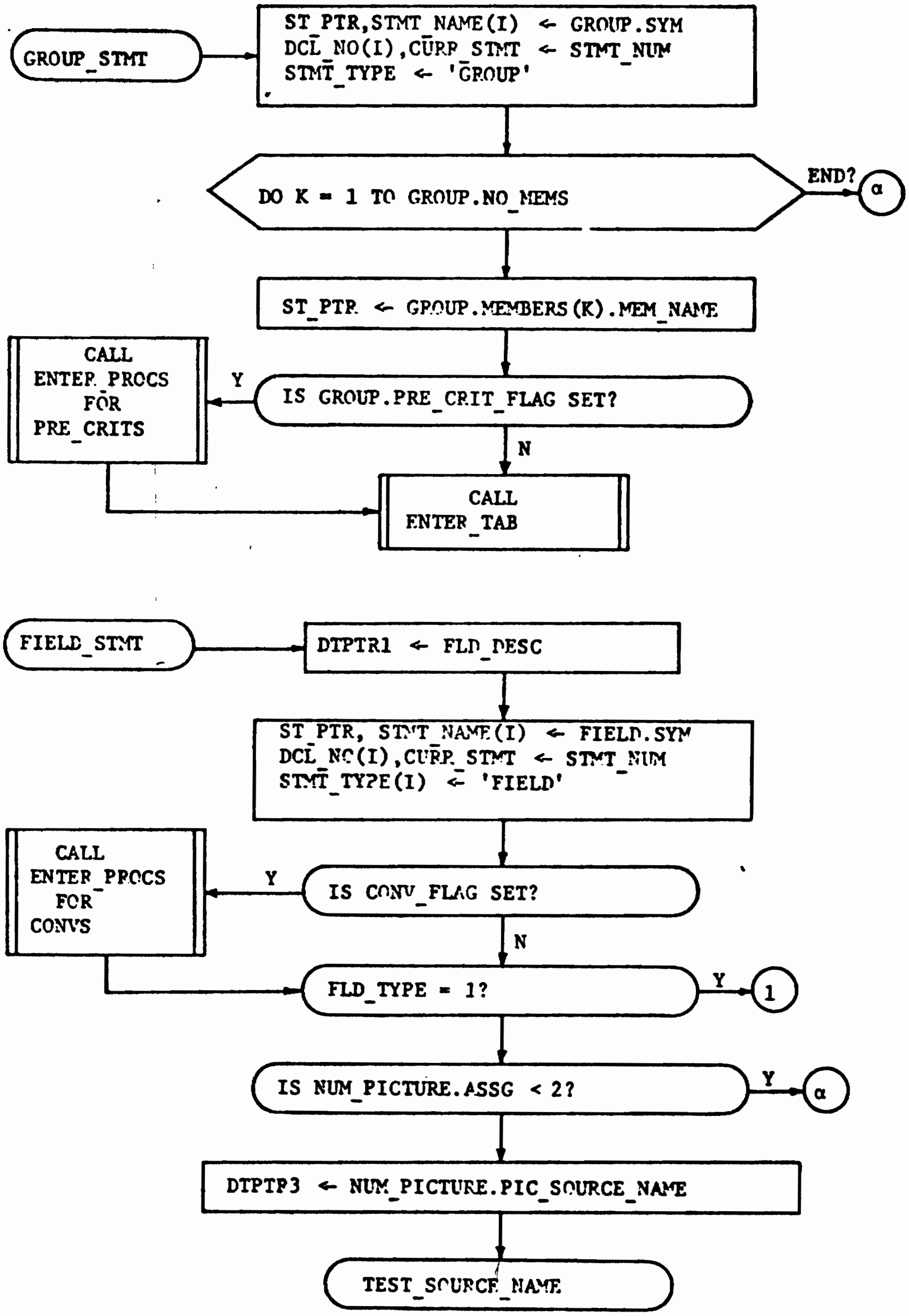

GROUP_STMT
ST_PTR,STMT_NAME(I) ← GROUP.SYM
DCL_NO(I),CURR_STMT ← STMT_NUM
STMT_TYPE ← 'GROUP'
DO K = 1 TO GROUP.NO_MEMS
END?
α
ST_PTR ← GROUP.MEMBERS(K).MEM_NAME
IS GROUP.PRE_CRIT_FLAG SET?
Y
CALL ENTER_PROCS FOR PRE_CRITS
N
CALL ENTER_TAB
FIELD_STMT
DTPTR1 ← FLD_DESC
ST_PTR, STMT_NAME(I) ← FIELD.SYM
DCL_NO(I),CURR_STMT ← STMT_NUM
STMT_TYPE(I) ← 'FIELD'
IS CONV_FLAG SET?
Y
CALL ENTER_PROCS FOR CONVS
N
FLD_TYPE = 1?
Y
1
IS NUM_PICTURE.ASSG < 2?
Y
α
DTPTR3 ← NUM_PICTURE.PIC_SOURCE_NAME
TEST_SOURCE_NAME

FIGURE 5-4D
GROUP_STMT, FIELD_STMT

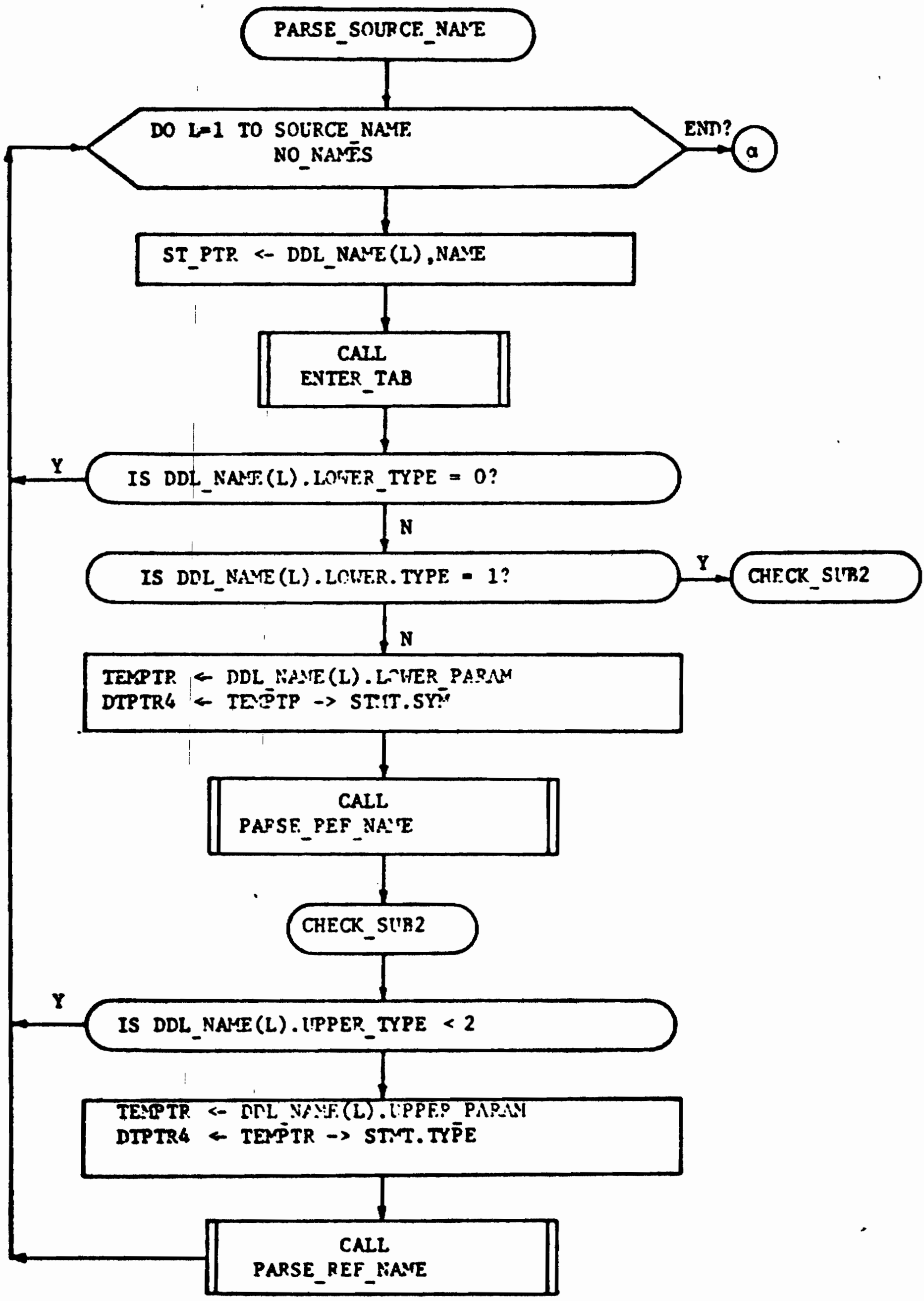

PARSE_SOURCE_NAME
DO L=1 TO SOURCE_NAME
NO_NAMES
END?
a
ST_PTR <- DDL_NAME(L),NAME
CALL
ENTER_TAB
Y
IS DDL_NAME(L).LOWER_TYPE = 0?
N
IS DDL_NAME(L).LOWER.TYPE = 1?
Y
CHECK_SUB2
N
TEMPTR <- DDL_NAME(L).LOWER_PARAM
DTPTR4 <- TEMPTP -> STMT.SYM
CALL
PARSE_REF_NAME
CHECK_SUB2
Y
IS DDL_NAME(L).UPPER_TYPE < 2
TEMPTR <- DDL_NAME(L).UPPER_PARAM
DTPTR4 <- TEMPTR -> STMT.TYPE
CALL
PARSE_REF_NAME

FIGURE 5-4D (continued)
GROUP_STMT, FIELD_STMT

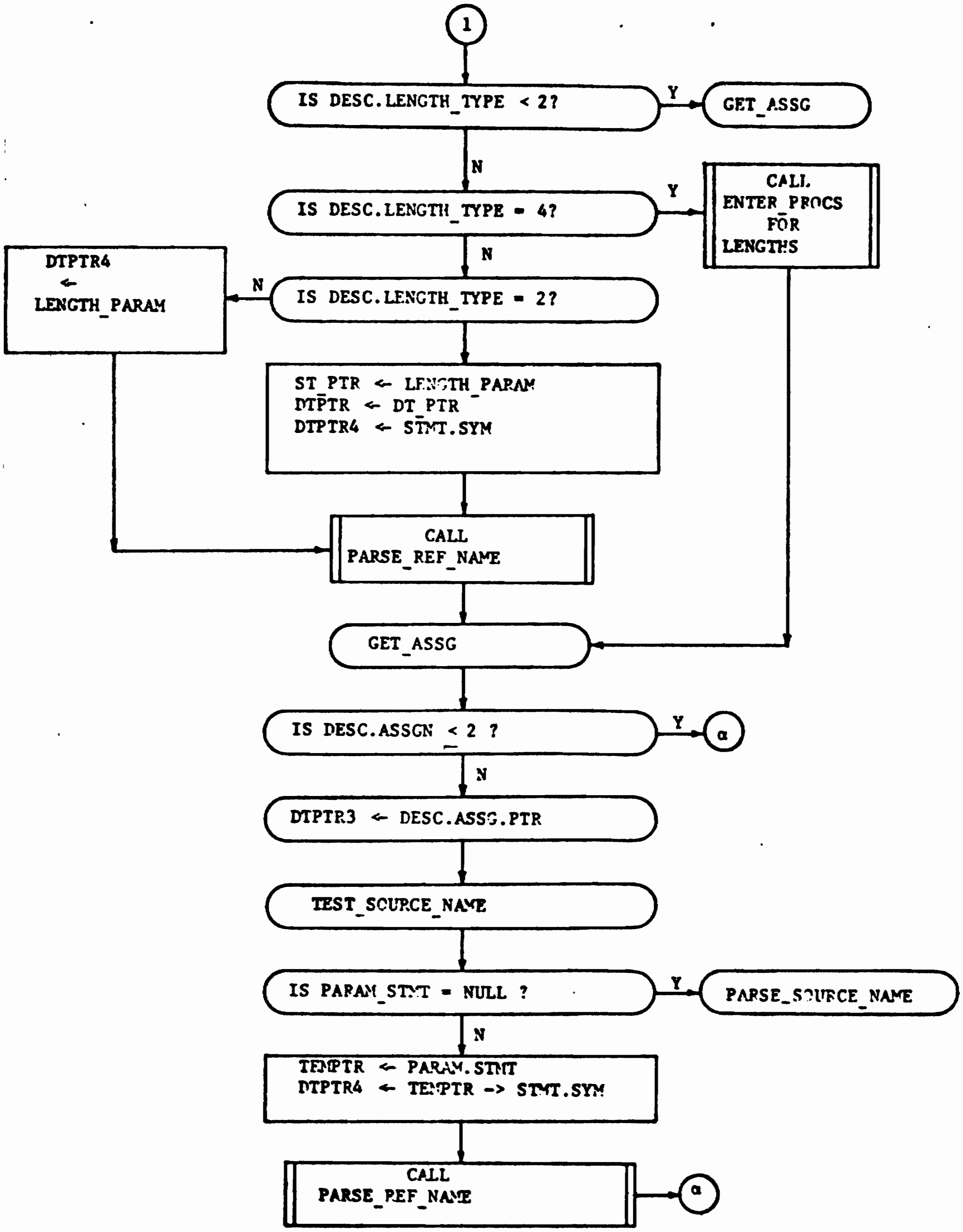


FIGURE 5-4D (continued)
GROUP_STMT, FIELD_STMT

and PARSE_REF_NAME (see Section 5.4.1.8) is called to get all statement references in reference Name. Then control is passed to GET_ASSG (see Section 5.4.1.7.1) entry. If the length is given as a reference name PARSE_REF_NAME is called and then control is passed to the entry GET_ASSG.

5.4.1.8.1 GET_ASSG entry (See Figure 5-4D)

Here a check is made to see if a mapping of the form " <= 'SOURCE_NAME" has been specified, if not control is returned to the loop were the next Data Table is to be processed. If an assignment has been specified DTPTR3 is set to point to Data Table where the assignment specification is stored. Next if the assignment is a parameter statement DTPTR4 is set to point to where the argument (i.e., reference name is stored) and PARSE_REF_NAME (see Section 5.4.1.9) is called. After returning from PARSE_REF_NAME control is passed to the loop were the next Data Table is to be processed. If the argument is via a Source Name then PARSE_SOURCE_NAME is called (see Section 5.4.1.9).

5.4.1.9 PARSE_REF_NAME (See Figure 5-4E)

This routine uses the structure based on DTPTR4 where the names that form a DDL reference name are stored. A loop is initiated to get all the names in reference name and for every name ENTER_TAB (see Section 5.4.3) is called.

5.4.2 PARSE_SOURCE_NAME (see Figure 5-4D)

This routine uses the structure based on DTPTR3 where the name and its subscripts that form a Source Name are stored. Then a loop is initiated to get all names in Source Name and ENTER_TAB (see Section 5.4.3) is called to store references into the cross-reference table for every name in Source Name. If the first subscript of name (I) is variable i.e., a parameter statement DTPTR4 is set to point to where the argument is stored (i.e., a reference name) and PARSE_REF_NAME (see Section 5.4.1.8) is called. If the first subscript is an integer and the second subscript do not appear control is transferred to loop where the next name in Source Name is to be processed.

If the second subscript in name (I) appears but it is an integer,control is passed to loop where the next name in Source Name is to be processed. If second subscript is variable i.e., a parameter statement DTPTR4 is set to point to where the argument is stored and PARSE_REF_NAME is called.

After all the names have been processed control is transferred to loop where next Data Table is to be processed.

5.4.3 ENTER_TAB (see Figure 5-4E)

This procedure enters the cross-reference information into the cross-reference table XREFTAB. This table is indexed by the DDL_STMT number. Upon entry to this routine, ST_PTR must point to the Symbol Table entry for the referenced statement. CURR_STMT

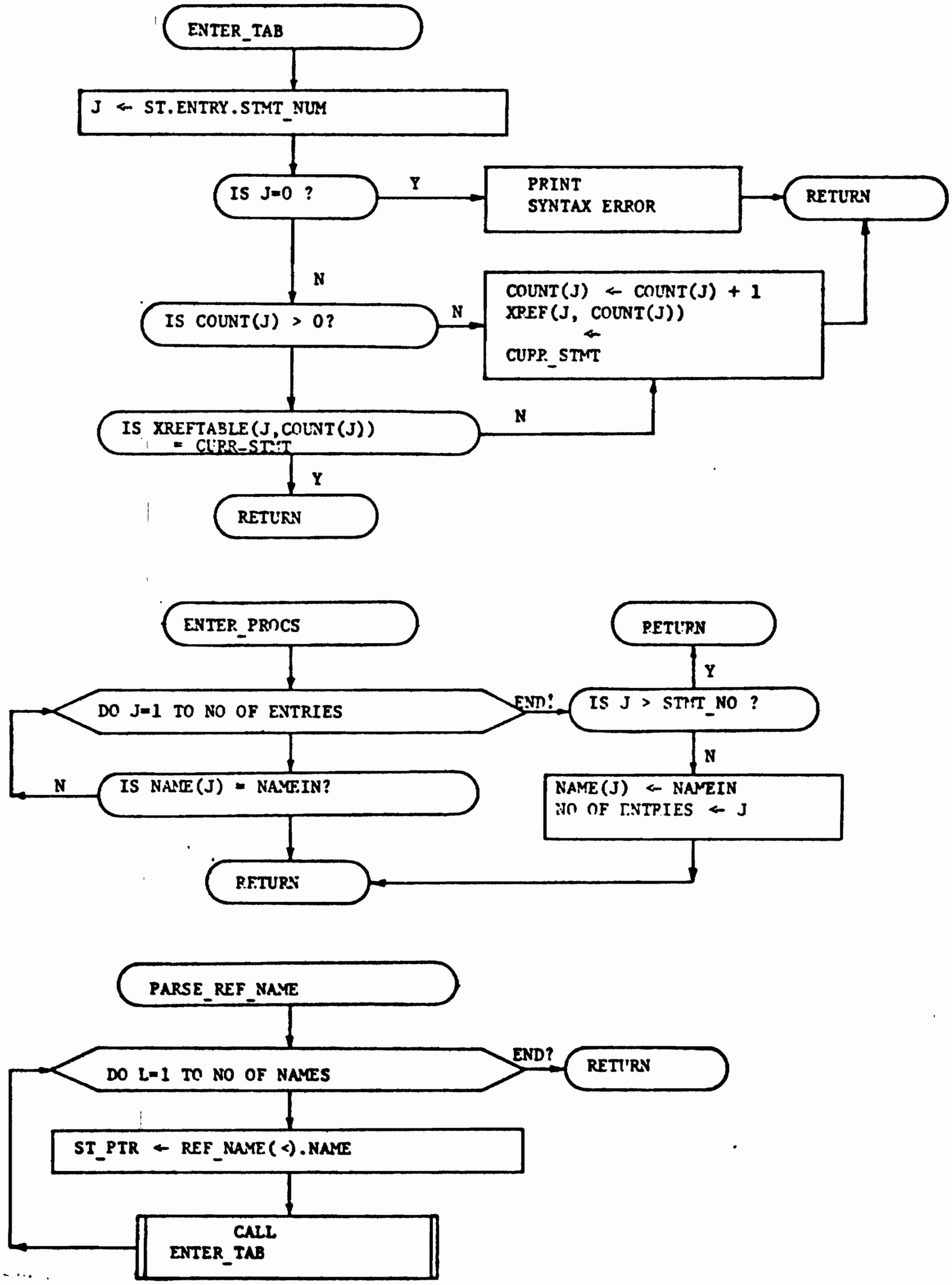


FIGURE 5-4E
ENTER_TAB, ENTER_PROCS

must contain the DDL_STMT number for the referencing statement. ENTER_TAB also checks if the statement number for the referenced statement is zero, if true, an error message is output.

First ENTER_TAB set J to the DDL_STMT number of the reference statement, if it is equal to zero the following error message is output:
"SYNTAX ERROR < STMT_NAME > USED IN STMT_NUM < CURR_STMT > HAS NOT BEEN DEFINED". and control is transferred to calling routine.

If J is different of zero a check is made to determine if the reference has already been made, if so, control is passed to calling routine. If the references have not been made COUNT (J) is increased by 1 and XREFTABLE (J, COUNT(J)) is set to CURR_STMT and control is then passed to calling program.

5.4.4 ENTER_PROCS (see Figure 5-4E)

This procedure is used to store the name of the DML procedures referenced in the DDL statements. There are three structures where the DML names are to be stored; FLD_LENGTHS based on FLD_LENGTHS_PTR, PRE_CRITS based on PRE_CRITS_PTR and CONVS based on CONVS_PTR. ENTER_PROGS uses two input parameters, the first one PTR which is a pointer and is set by the calling routines to either FLD_LENGTHS, PTR, PRE_CRITS_ PTR or CONVS_PTR, the second parameter is DML_NAME and is set by the calling routines to the name of the DML procedure referenced in the DDL statement.

The names are stored by ENTER_PROCS in the appropriate structure in a sequential form, i.e., every time a DML name is to be stored a check is made starting with the first DML name stored to determine if such DML name has not been already stored, if the DML name is not found the number of entries in that structure is increased by one and the DML name is stored. In either of the above two cases control is transferred to calling routine.

5.4.5 PRINT_TAB (see Figure 5-4F)

This procedure is used to first sort the DDL_NAMES in alphabetic order and then output the XREFTABLE. Also PRINT_TAB output the names of the DML procedures referenced in the DDL statements.

The arrays STMT_NAME and STMT_TYPE are indexed by the stmt number given by the sequential scan of the Data Table. The arrays XREFTABLE and COUNT are indexed by the DDL_STMT numbers. These are contained in the DCL_NO array which is indexed in the same way as STMT_NAME and STMT_TYPE.

After the XREFTAB and DML names have been output control returns to the DDL COMP procedure.

5.5 Phase 2A of The DDL Compiler - Code Generation (Data Parsing)

After the Symbol and Data Tables are created for the DDL

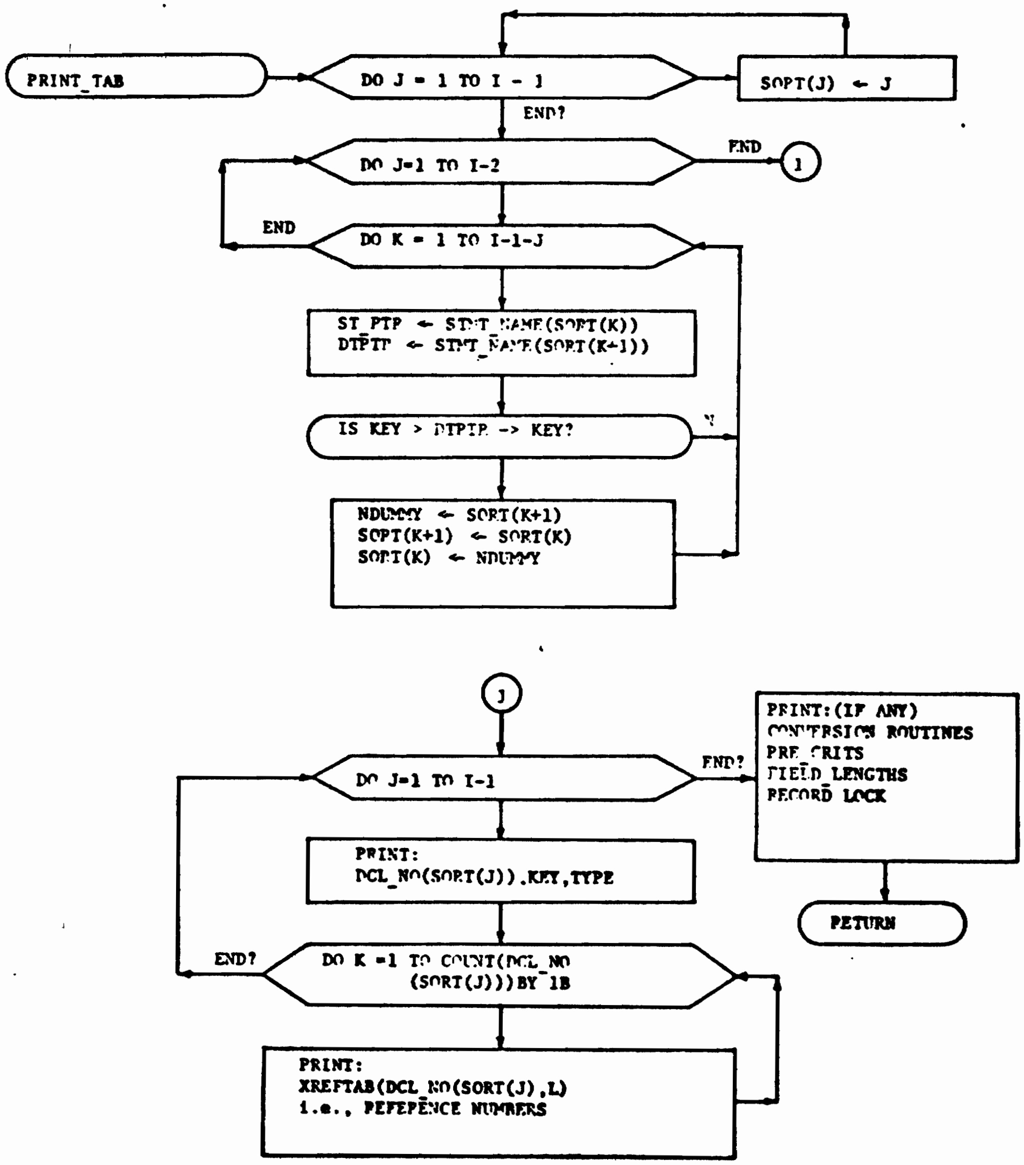


FIGURE 5-4F
PRINT_TAB

statements the next step is the generation of the object language. The DDL compiler as we already mention generates PL/1 as the object language; in other words it translates DDL statements into PL/1 statements. This phase of DDL compiler is called phase 2. It is composed of phase 2a - Code Generation for data parsing and phase 2b - Code Generation for data movement. Phase 2a is presented in the following paragraphs and Phase 2b is given in Section 5.6.

The main task of Phase 2a is to generate a PL/1 structure corresponding to the structure for the Source file described in DDL, this is done via PL/1 declare statements. In order to achieve a better explanation in this topic let us again give an example. Assume that the user has not specified the INLEN and INCNT parameters to the compiler and that the description of its Source file in DDL is the following:

```
     FL1 IS FILE (RCD1 ,STORAGE = CARDIN);
         RCD1 IS RECORD (ELEM1(4), ELEM2(0:20)
                         ,PRE_CRIT = 'FUN1' ,SIZE =  VARIABLE
                                                     (1680));
              ELEM1 IS FIELD (CHAR(20));
              ELEM2 IS GROUP (FLD2, FLD2, FLD3(7));
                    FLD1 IS FIELD (BIT(16));
                    FLD2 IS FIELD (CHAR(8));
                    FLD3 IS FIELD (CHAR(10));
CARDIN IS CARD;
```

The above example specified that FL1 is a file which is stored in CARDS, and the RECORD which forms the file is variable in size, its maximum value is 1680 characters. RCD1 is composed of two members, the first one, ELEM1 is a repeating field which occurrs 4 times, and its second member, ELEM2, is a optional group which can occur up to a maximum of 20 times depending on the criteria given by the user via a DML routine called "FUN1".

The PL/1 declare statements that the DDL compiler will generate for this example are: (keep in mind that the INLEN and INCNT options were not specified).

```
DCL 1 FL1,
     2 RCD1,
      3 ELEM1 (4),
       4 PTR POINTER,
       4 LEN FIXED BIN INIT ((4)0B),
      3 ELEM2 (20),
       4 FLD1,
        5 PTR POINTER,
        5 LEN FIXED BIN INIT ((20)0B),
       4  FLD 2,
        5 PTR POINTER,
        5 LEN FIXED BIN INIT ((20)0B),
       4 FLD3 (7),
        5 PTR POINTER,
        5 LEN FIXED BIN INIT ((140)0B);
```

From the above example, it is easy to see that for every member of its repeating the DDL compiler generates an array whose name is the DDL name and its dimensions is the maximum number of times such member can occur. For the DDL FIELD two PL/1 statements are generated; the first one is a pointer (this pointer will be set at run time with the core address of the Data associated to this field is kept) and the second is a fixed binary field (this data element in the PL/1 structure will be set at run time with the length of the DDL FIELD).

If the INCNT and INLEN options were specified to the DDL COMP the PL/1 structure generated by the DDL compiler is the following:

```
DCL 1 FL1,
     2 RCD1,
      3 LEN  FIXED BIN,
      3 ELEM1,
       4 COUNT FIXED BIN,
       4 #ELEM1 (4)
        5 PTR POINTER,
        5 LEN FIXED BIN INIT ((4)0B),
     3 ELEM2,
      4 COUNT FIXED BIN,
      4 #ELEM2 (20),
       5 LEN FIXED BIN INIT (0B),
       5 FLD1,
        6 PTR POINTER,
 6 LEN FIXED BIN INIT ((20)0B),
5 FLD2,
 6 PTR POINTER,
 6 LEN FIXED BIN INIT (120)0B),
5 FLD3
 6 COUNT FIXED BIN,
 6 #FLD 3(7),
  7 PTR POINTER,
  7 LEN FIXED BIN INIT ((140)0B);
```

Thus, only a new data item has been added to the PL/1 structure, i.e., "COUNT" Data items (which is fixed binary and at run time this field will contain the number of times a repeating member actually occurred in the Record being parsed).

Concurrently with the generation of the PL/1 declare statements, is the generation of PL/1 statements which when executed will perform the parsing of the input record.

5.5.1 CODE_GEN_PARSE

CODE_GEN_PARSE is called from the DDLCOMP, after the Symbol Table and Data Table have been created and if there were no errors detected. GEN_CODE_PARSE walks in these tables and calls on the appropriate routines to achieve the declaration of the PL/1 structure and the generation of PL/1 code to carry on the parsing of the input records. CODE_GEN_PARSE is the monitor for code generation to parse the input record.

After getting the entry for the CONVERT statement in the Data Table CODE_GEN_PARSE generates the PL/1 procedure statement. Next it generates the PL/1 statements for END OF FILE condition and the initialization of run-time variables used in data parsing. It then generates code to declare CDP (i.e., current data pointer) which is a pointer to the Internal Input buffer where the Input Record is kept, and to declare #DDL_MAXCDP which is used to hold the maximum value reached by CDP during the parsing process when the SCAN statement is used.

After getting the entry in the Data Table for the Source File CODE_GEN_PARSE saves the character code used to record the input record. Then it gets the entry in the Data Table corresponding to the Record statement and it saves the information concerning the size of the Record. Next it gets the entry in the Data Table for the Storage statement and sets the value of TRANS_FLAG to:

0 if character code is EBCDIC or if BCD or ASCII and recording is mode ALL_BIN

1 if character code is ASCII and recording mode ALL_CHAR

2 if character code is BCD and recording mode ALL_CHAR

3 if character code is ASCII and recording mode MIXED

4 if character code is BCD and recording mode MIXED

Then if Storage Type is TAPE,CODE_GEN_PARSE calls on GEN_DD_TAPE (see Section 5.5.15), if TYPE is DISK then

GEN_DD_DISK (see Section 5.5.15) is called, otherwise the input is via CARDS and the appropriate DD CARD is generated. Then CODE_GEN_PARSE calls on GEN_READ (see Section 5.5.16) which is a procedure used to generate PL/1 statements to declare internal input buffers and to generate PL/1 read statements to read the Source File. If translation between character codes BCD or ASCII and EBCDIC is required then CODE_GEN_PARSE generates the appropriate PL/1 translate statements.

The next step taken by CODE_GEN_PARSE is to generate the PL/1 declare statements and the PL/1 statements that will achieve the parsing of the input record. This is done by taking the members of the RECORD statement. The order in which the members are processed is given by the SCAN VECTOR, such vector is set by either the SCAN statement if specified or to 1, 2, to numbers of members if SCAN was not given, such members can be GROUPS or FIELDS. Then the procedure, to set "CDP" for current member, SET_POS is called (see Section 5.5.14). If a member is a FIELD then the procedure to generate PL/1 declare statements for the FIELD is called, this procedure is GEN_FLD_DCL (see Section 5.5.5). Next the procedure to generate PL/1 code to parse FIELD is called, the name of this procedure is GEN_PARSE_FLD (see Section 5.5.7). After this CODE_GEN_PARSE gets next member in RECORD statement and the process repeats. If member is being declared as a GROUP then the procedure to

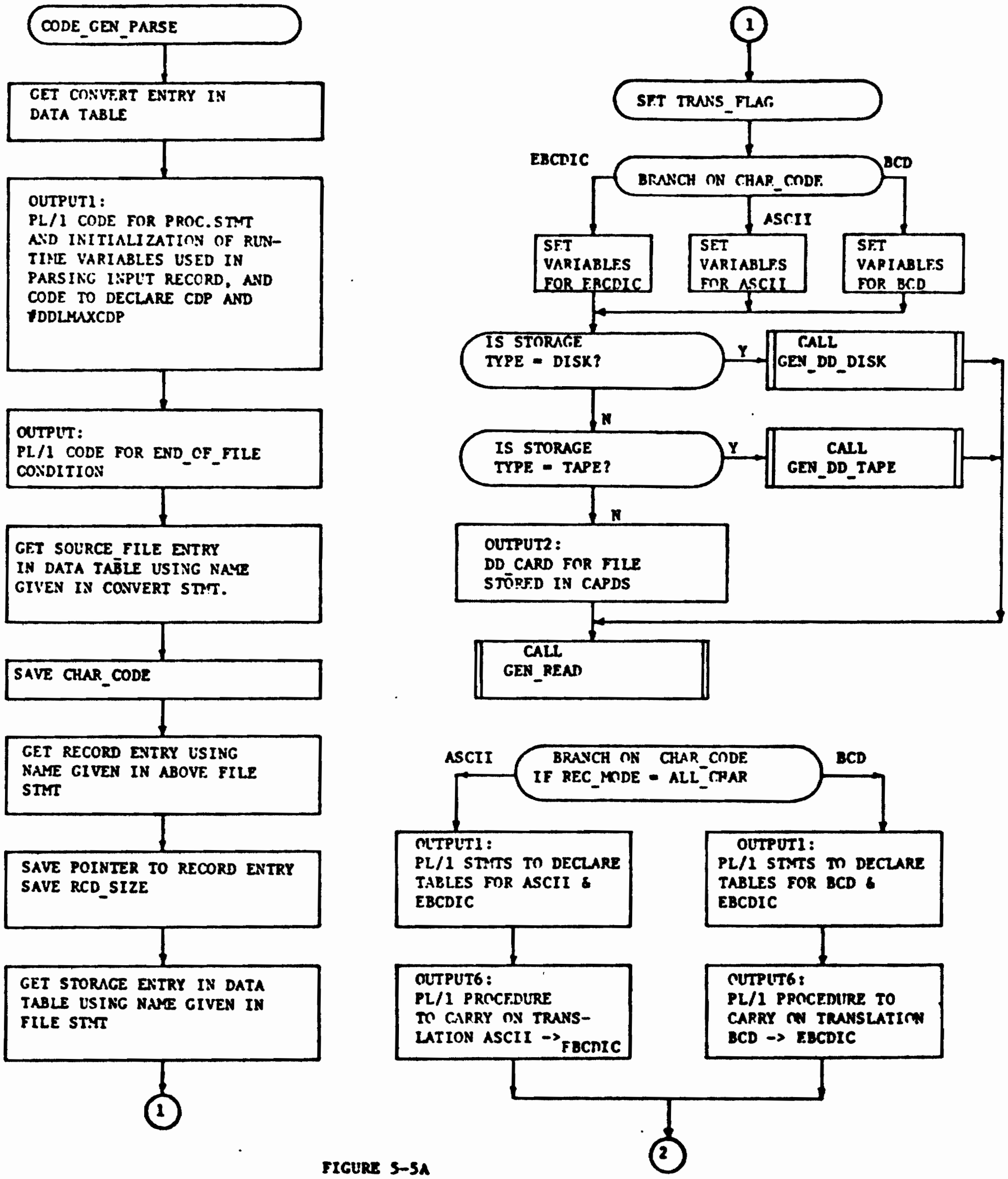

CODE_GEN_PARSE
GET CONVERT ENTRY IN DATA TABLE
OUTPUT1: PL/1 CODE FOR PROC.STMT AND INITIALIZATION OF RUN-TIME VARIABLES USED IN PARSING INPUT RECORD, AND CODE TO DECLARE CDP AND #DDLMAXCDP
OUTPUT: PL/1 CODE FOR END_OF_FILE CONDITION
GET SOURCE_FILE ENTRY IN DATA TABLE USING NAME GIVEN IN CONVERT STMT.
SAVE CHAR_CODE
GET RECORD ENTRY USING NAME GIVEN IN ABOVE FILE STMT
SAVE POINTER TO RECORD ENTRY SAVE RCD_SIZE
GET STORAGE ENTRY IN DATA TABLE USING NAME GIVEN IN FILE STMT
1
1
SET TRANS_FLAG
EBCDIC
BRANCH ON CHAR_CODE
BCD
ASCII
SET VARIABLES FOR EBCDIC
SET VARIABLES FOR ASCII
SET VARIABLES FOR BCD
IS STORAGE TYPE = DISK?
Y
CALL GEN_DD_DISK
N
IS STORAGE TYPE = TAPE?
Y
CALL GEN_DD_TAPE
N
OUTPUT2: DD_CARD FOR FILE STORED IN CAPDS
CALL GEN_READ
ASCII
BRANCH ON CHAR_CODE IF REC_MODE = ALL_CHAR
BCD
OUTPUT1: PL/1 STMTS TO DECLARE TABLES FOR ASCII & EBCDIC
OUTPUT1: PL/1 STMTS TO DECLARE TABLES FOR BCD & EBCDIC
OUTPUT6: PL/1 PROCEDURE TO CARRY ON TRANS-LATION ASCII -> EBCDIC
OUTPUT6: PL/1 PROCEDURE TO CARRY ON TRANSLATION BCD -> EBCDIC
2

FIGURE 5-5A

CODE_GEN_PARSE

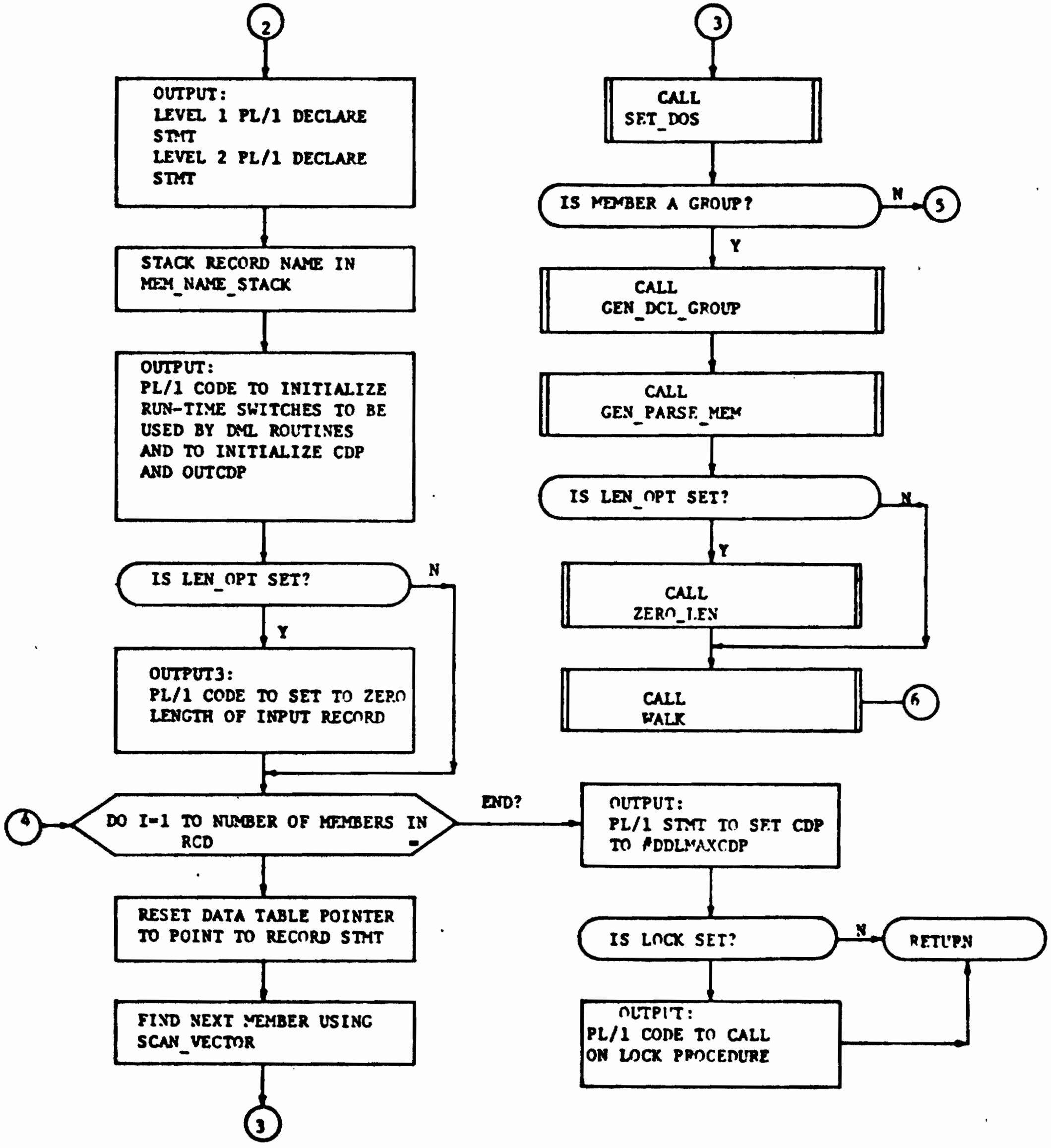

2
OUTPUT:
LEVEL 1 PL/1 DECLARE STMT
LEVEL 2 PL/1 DECLARE STMT
STACK RECORD NAME IN MEM_NAME_STACK
OUTPUT:
PL/1 CODE TO INITIALIZE RUN-TIME SWITCHES TO BE USED BY DML ROUTINES AND TO INITIALIZE CDP AND OUTCDP
IS LEN_OPT SET?
N
Y
OUTPUT3:
PL/1 CODE TO SET TO ZERO LENGTH OF INPUT RECORD
4
DO I=1 TO NUMBER OF MEMBERS IN RCD
END?
RESET DATA TABLE POINTER TO POINT TO RECORD STMT
FIND NEXT MEMBER USING SCAN_VECTOR
3
3
CALL SET_DOS
IS MEMBER A GROUP?
N
5
Y
CALL GEN_DCL_GROUP
CALL GEN_PARSE_MEM
IS LEN_OPT SET?
N
Y
CALL ZERO_LEN
CALL WALK
6
OUTPUT:
PL/1 STMT TO SET CDP TO #DDLMAXCDP
IS LOCK SET?
N
RETURN
OUTPUT:
PL/1 CODE TO CALL ON LOCK PROCEDURE

FIGURE 5-5A (CONTINUED)

CODE_GEN_PARSE

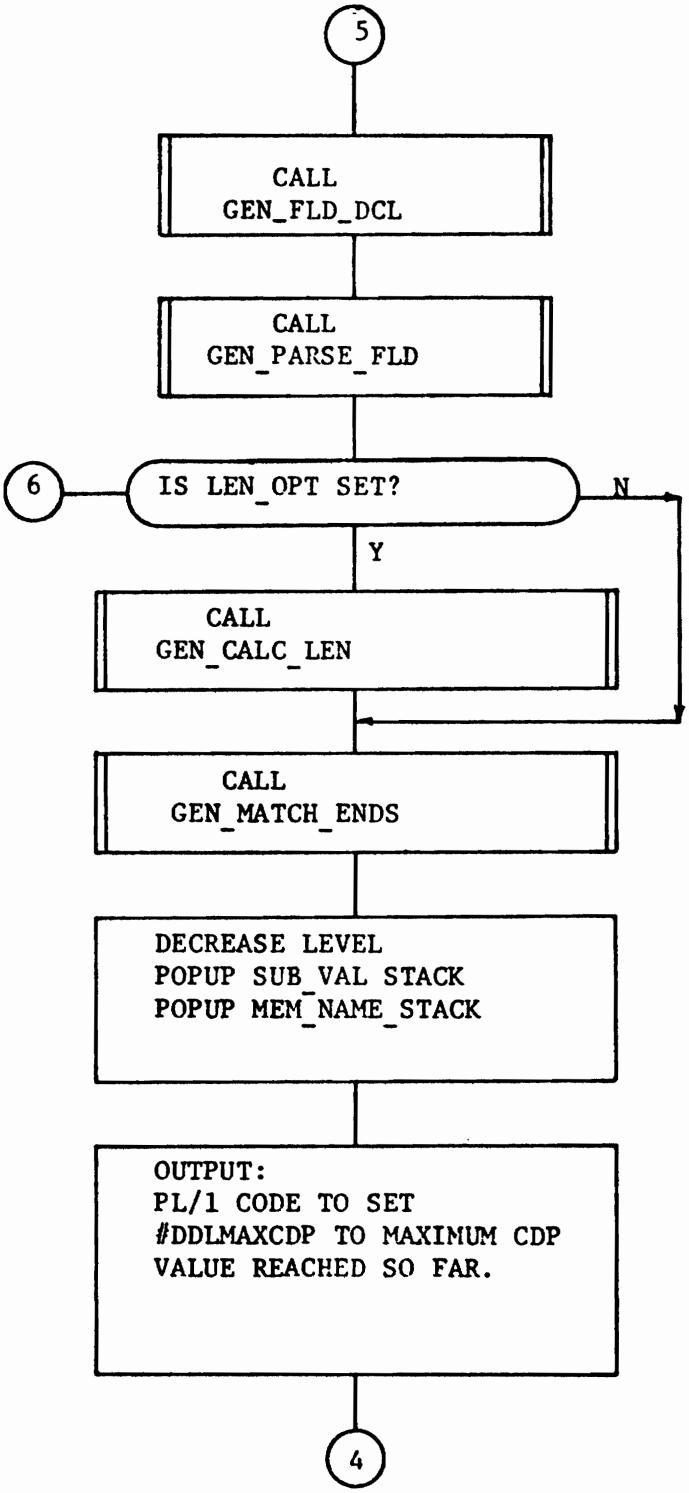


FIGURE 5-5A (continued)
CODE_GEN_PARSE

generate PL/1 declare statements for the GROUP is called, this procedure is GEN_DCL_GROUP (see Section 5.5.4). Similarly a procedure to generate PL/1 code to parse Group is called, the name of this procedure is GEN_PARSE_MEM. Since the GROUP can have members of its own a procedure to parse such members is called, this procedure is recursive since some or all the members of a GROUP can themselves be GROUPS. This procedure is WALK (see Section 5.5.2). After all members of the GROUPs and SUB-GROUPs have been processed WALK returns to CODE_GEN_PARSE and then the next member in the RECORD is processed and the process repeats until all members in the RECORD statement have been processed.

After each member has been processed the procedure GEN_MATCH_ENDS (see Section 5.5.9) is called, this procedure generates the PL/1 END statement corresponding to the DO loop generated by GEN_PARSE_MEM, if it was a repeating member.

When the INLEN option was specified to the DDL compiler then after returning from GEN_PARSE_MEM the procedure ZERO_LEN is called (see Section 5.5.8), this procedure generates PL/1 code to set to zero the length of the current member being processed after a new input has been read. Also, after returning from GEN_PARSE_FLD the procedure GEN_CALC_LEN is called (see Section 5.5.8), this procedure generates PL/1 code to accumulate the length of the current field into the length of its ancestors.

If the LOCK option was specified in the RECORD statement the CODE_GEN_PARSE generates PL/1 code to call on the procedure to perfcrm security checking before the translation of the input record is attempted.

5.5.2 WALK

WALK is called from CODE_GEN_PARSE (see Section 5.5.1), it is used to process the members of the GROUP statement specified in the definition of the Source file. WALK is a recursive procedure since the members of a GROUP can also be GROUPS, and thus WALK will call on itself.

Upon entry to the procedure, the pointers to the members of the current GROUP being processed are saved. Then the first member is processed, if it is a FIELD, GEN_FLD_DCL (see Section 5.5.5) is call, this procedure generates the PL/1 statements for the FIELD. Next, the procedure GEN_PARSE_FLD is called (see Section 5.5.7) to generate PL/1 code to parse the FIELD. If the member is a GROUP then the procedure GEN_DCL_GROUP is called (see Section 5.5.4) to generate PL/1 declare statements for this GROUP. Next, GEN_PARSE_MEM is called (see Section 5.5.3) to generate PL/1 code to parse Group. Since current member is a GROUP then call WALK to process its submembers.

Similarly, to GEN_CODE_PARSE, the procedure WALK after

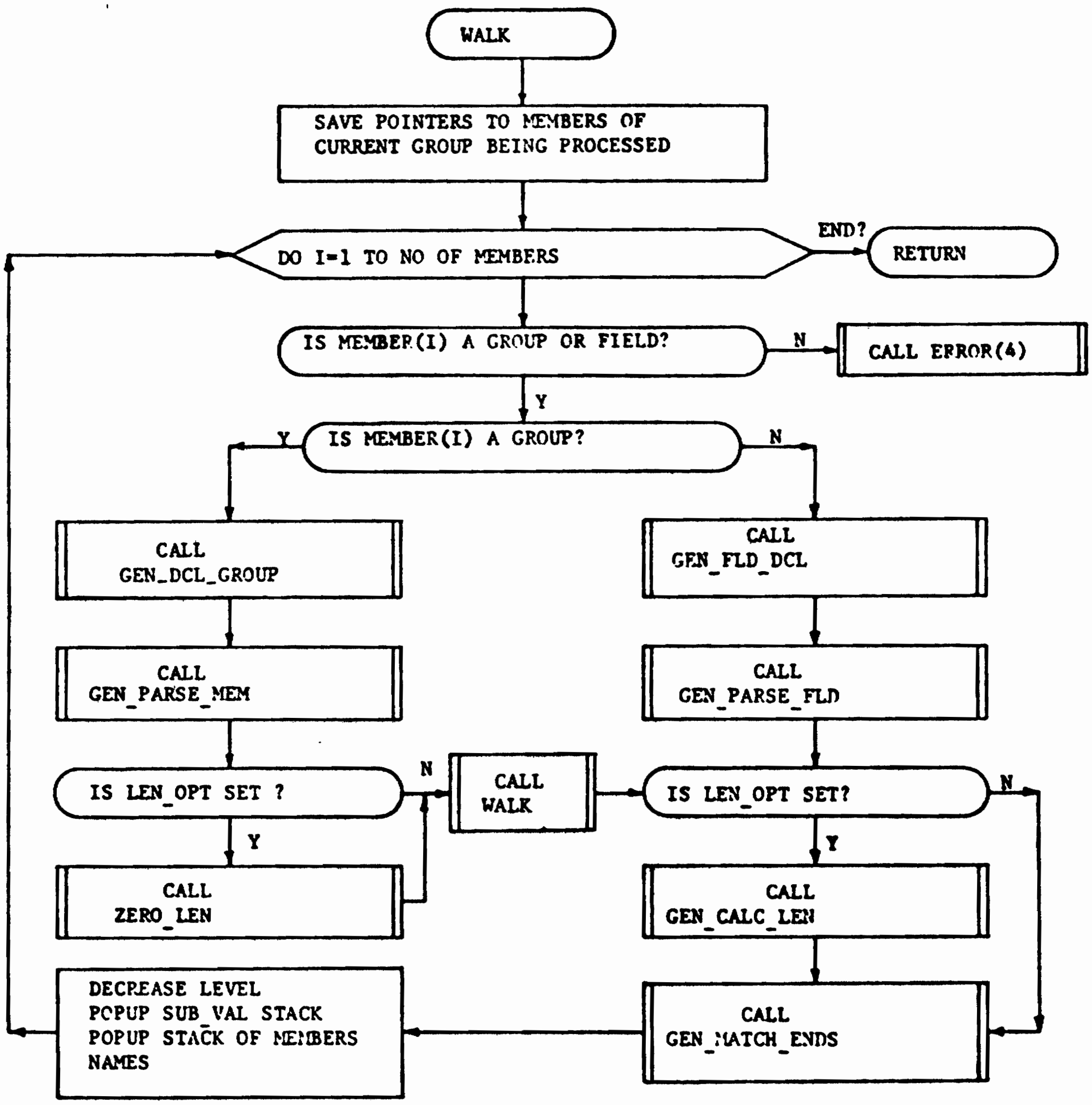


FIGURE 5-5B
WALK

processing each member calls on GEN_MATCH_END (see Section 5.5.9).

If the INLEN option was specified to the DDL compiler then after returning from GEN_PARSE_MEM, the procedure ZERO_LEN is called (see Section 5.5.8) and also after returning from GEN_PARSE_FLD the procedure GEN_CALC_LEN is called (see Section 5.5.8).

5.5.3 GEN_PARSE_MEM

This procedure when called generates PL/1 code to parse the member which can be a FIELD or a GROUP. The member can occur a fixed number of times or can be optional and occur a variable number of times. For internal purposes it stacks the iteration variable (if repeating member) in ITER_STACK, the pointer to the member name in the symbol table in the MEM_NAME_STACK, and a unique label number (LABEL #) in LABEL #_STACK.

ITER_STACK is a stack which contains the iteration variable in parenthesis (e.g. (I01)), if the member repeats, or the null string if not. MEM_NAME_STACK is a stack where the pointer to the member name is kept. LABEL #_STACK is a stack which contains an unique label number used to generate the corresponding PL/1 END statement for the DO loops if the members repeats, or a null string otherwise.

If the member occurrs a variable number of times then GEN_PARSE_MEM generates PL/1 code to call the procedure specified by the ",PRE_CRIT =" given by the user.

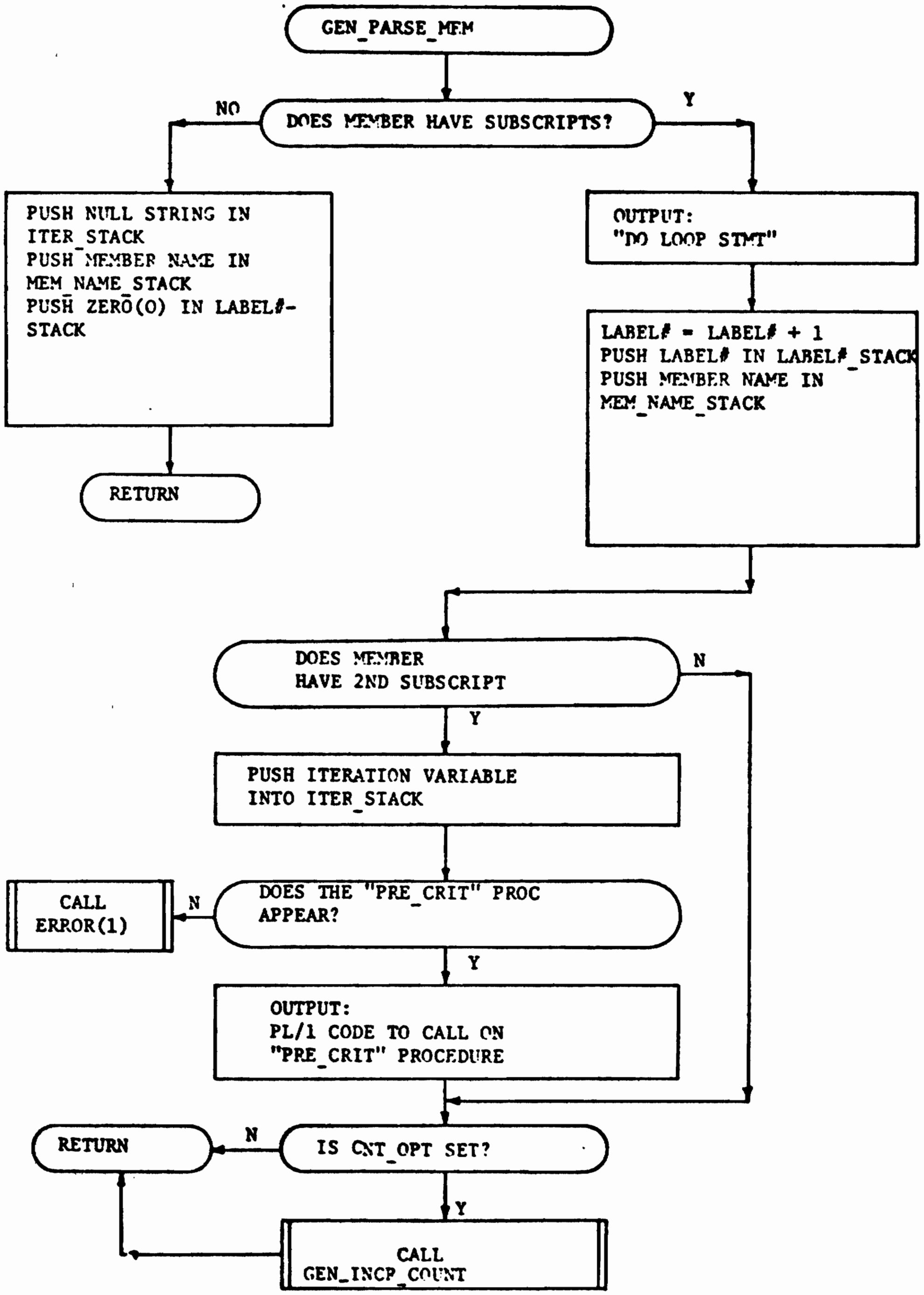

GEN_PARSE_MEM
DOES MEMBER HAVE SUBSCRIPTS?
NO
Y
PUSH NULL STRING IN ITER_STACK
PUSH MEMBER NAME IN MEM_NAME_STACK
PUSH ZERO(0) IN LABEL#-STACK
RETURN
OUTPUT: "DO LOOP STMT"
LABEL# = LABEL# + 1
PUSH LABEL# IN LABEL#_STACK
PUSH MEMBER NAME IN MEM_NAME_STACK
DOES MEMBER HAVE 2ND SUBSCRIPT
N
Y
PUSH ITERATION VARIABLE INTO ITER_STACK
DOES THE "PRE_CRIT" PROC APPEAR?
N
CALL ERROR(1)
Y
OUTPUT: PL/1 CODE TO CALL ON "PRE_CRIT" PROCEDURE
IS CNT_OPT SET?
N
RETURN
Y
CALL GEN_INCP_COUNT

FIGURE 5-5C
GEN_PARSE_MEM

Finally, if CNT_OPT was specified then the procedure GEN_INCR_COUNT (see Section 5.5.13) is called to generate PL/1 code to accumulate the number of times the member actually occurred in the current input record.

5.5.4 GEN_DCL_GROUP

This procedure generates the PL/1 declare statements for the current GROUP being processed. Upon entry to the procedure CALL_SUB (see Section 5.5.6) is called. CALL_SUB is a procedure that returns the value of the subscript if the GROUP repeats, it returns the maximum value if the GROUP repeats a variable number of times. Then the PL/1 statement of the type:

"LEVEL < GROUP_NAME > [(SUBSCRIPT)] is generated. If the CNT_OPT was specified then a PL/1 declare statement is generated to declare a field to keep a COUNT of the number of times the GROUP actually occurred in the current input record. Also if the LEN_OPT was specified then GEN_DCL_GROUP generates a PL/1 declare statement to declare a field where the length of this repeating Group is to be stored.

When repeating GROUPS are being declared in PL/1, in order to initialize the COUNT field to zero, GEN_DCL_GROUP calls on INIT_CNT (see Section 5.5.12). In order to initialize the LEN field to zero. The procedure INIT_LEN (see Section 5.5.12) is called.

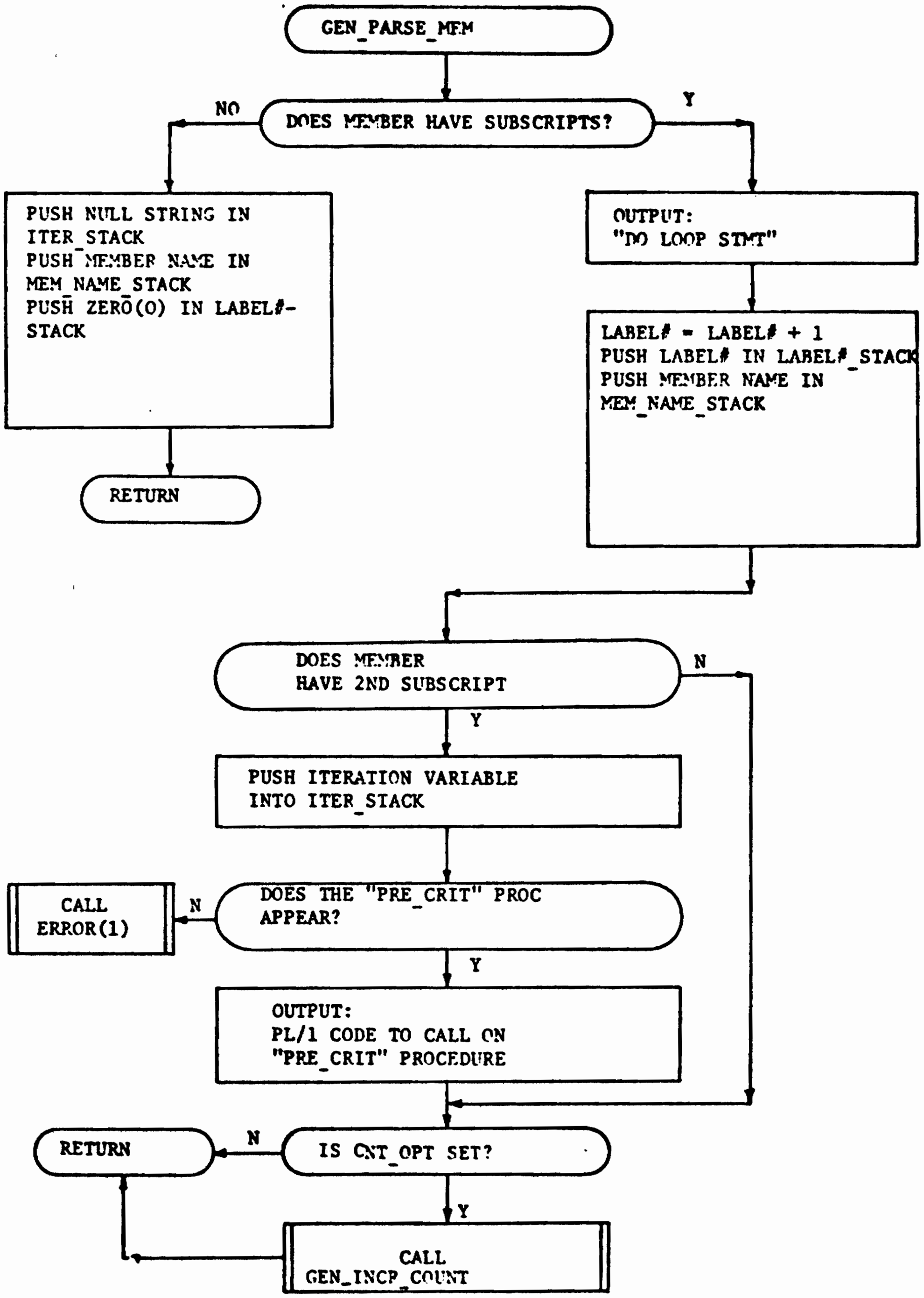

GEN_PARSE_MEM
DOES MEMBER HAVE SUBSCRIPTS?
NO
Y
PUSH NULL STRING IN ITER_STACK
PUSH MEMBER NAME IN MEM_NAME_STACK
PUSH ZERO(0) IN LABEL#-STACK
RETURN
OUTPUT: "DO LOOP STMT"
LABEL# = LABEL# + 1
PUSH LABEL# IN LABEL#_STACK
PUSH MEMBER NAME IN MEM_NAME_STACK
DOES MEMBER HAVE 2ND SUBSCRIPT
N
Y
PUSH ITERATION VARIABLE INTO ITER_STACK
DOES THE "PRE_CRIT" PROC APPEAR?
N
CALL ERROR(1)
Y
OUTPUT: PL/1 CODE TO CALL ON "PRE_CRIT" PROCEDURE
IS CNT_OPT SET?
N
RETURN
Y
CALL GEN_INCP_COUNT

FIGURE 5-5C
GEN_PARSE_MEM

Finally, if CNT_OPT was specified then the procedure GEN_INCR_COUNT (see Section 5.5.13) is called to generate PL/1 code to accumulate the number of times the member actually occurred in the current input record.

5.5.4 GEN_DCL_GROUP

This procedure generates the PL/1 declare statements for the current GROUP being processed. Upon entry to the procedure CALL_SUB (see Section 5.5.6) is called. CALL_SUB is a procedure that returns the value of the subscript if the GROUP repeats, it returns the maximum value if the GROUP repeats a variable number of times. Then the PL/1 statement of the type:

"LEVEL < GROUP_NAME > [(SUBSCRIPT)] is generated. If the CNT_OPT was specified then a PL/1 declare statement is generated to declare a field to keep a COUNT of the number of times the GROUP actually occurred in the current input record. Also if the LEN_OPT was specified then GEN_DCL_GROUP generates a PL/1 declare statement to declare a field where the length of this repeating Group is to be stored.

When repeating GROUPS are being declared in PL/1, in order to initialize the COUNT field to zero, GEN_DCL_GROUP calls on INIT_CNT (see Section 5.5.12). In order to initialize the LEN field to zero. The procedure INIT_LEN (see Section 5.5.12) is called.

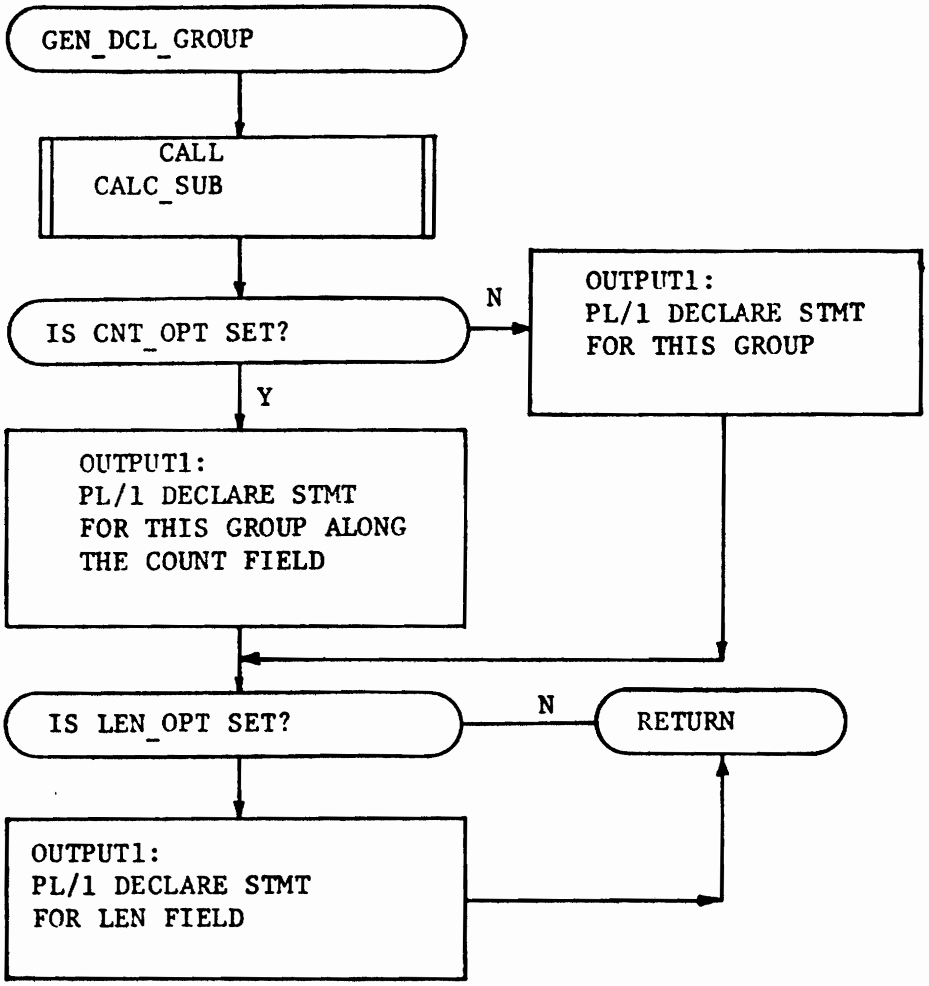

GEN_DCL_GROUP
CALL CALC_SUB
IS CNT_OPT SET?
N
OUTPUT1: PL/1 DECLARE STMT FOR THIS GROUP
Y
OUTPUT1: PL/1 DECLARE STMT FOR THIS GROUP ALONG THE COUNT FIELD
IS LEN_OPT SET?
N
RETURN
OUTPUT1: PL/1 DECLARE STMT FOR LEN FIELD

FIGURE 5-5D

GEN_DCL_GROUP

5.5.5 GEN_FLD_DCL

This procedure forms the PL/1 declare statements for the FIELD statements. Upon entry to GEN_FLD_DCL, CALC_SUB (see Section 5.5.6) is called, this procedure gets the value of the first and second subscripts of the FIELD (if any). Then the PL/1 declare statements for this FIELD are generated. Also if the CNT_OPT was specified then the PL/1 declare statement for the field count is generated. INIT_LEN (see Section 5.5.12) and INIT_CNT are also called from GEN_FLD_DCL in order to initialize the length and count fields to zero.

5.5.6 CALL_SUB

CALL_SUB is used to get from the corresponding entry in the Data Table the value of the subscripts (if any) for the current member being processed. It stacks the subscript in numeric form in VAL_SUB and saves the subscripts enclosed in parentheses in SUB. The stack VAL_SUB is used by INIT_LEN and INIT_CNT (see Section 5.5.12). SUB is used in the generation of the PL/1 declare statement for the current member. If the member does not have subscripts then SUB is set to the Null string and the value 1 (one) is stacked in VAL_SUB.

5.5.7 GEN_PARSE_FLD

This procedure is used to generate PL/1 code, which when executed will set the field PTR in the PL/1 structure with the core address in the input buffer corresponding to the current field being processed. Also it generates PL/1 code to

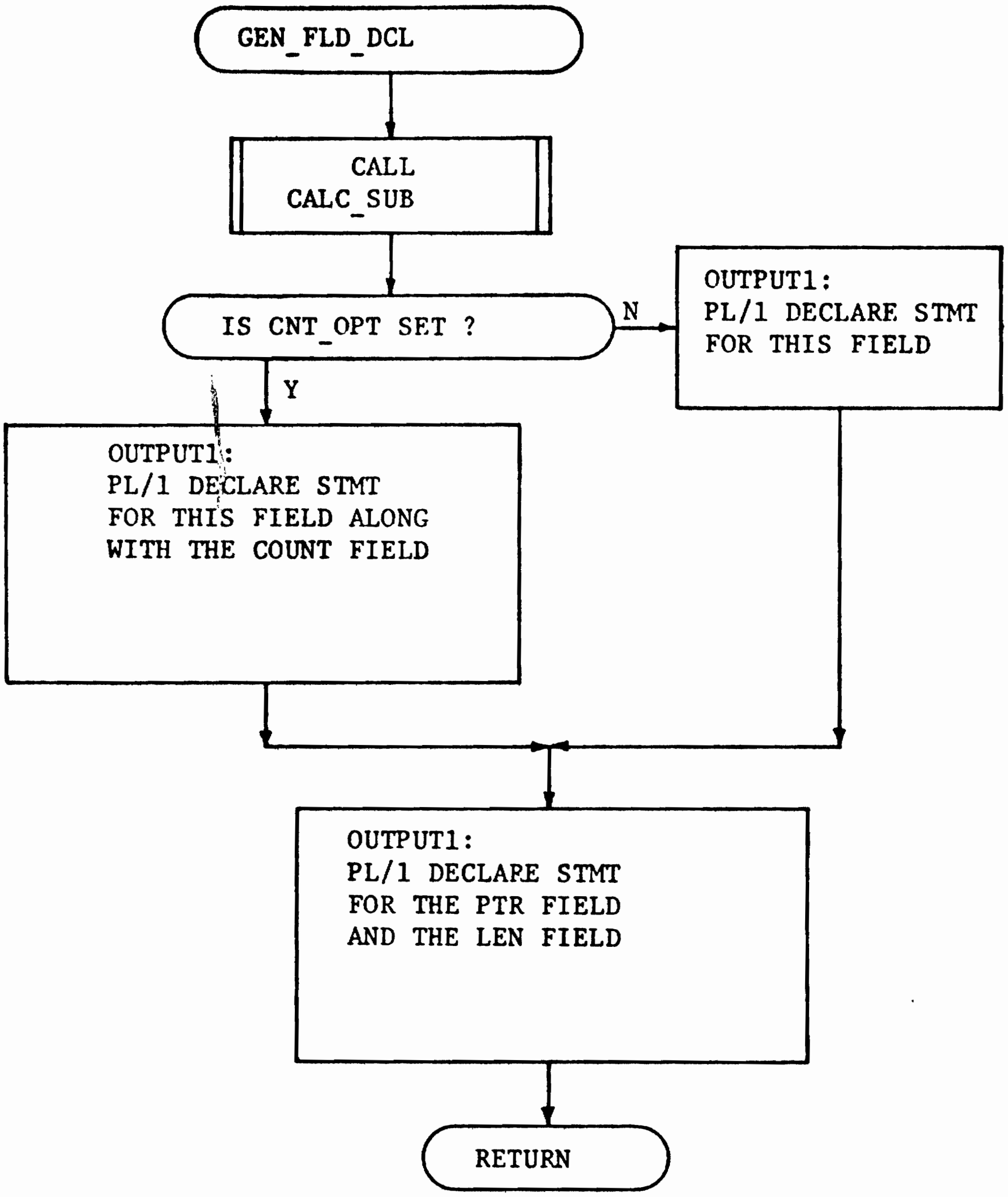


FIGURE 5-5E
GEN_FLD_DCL

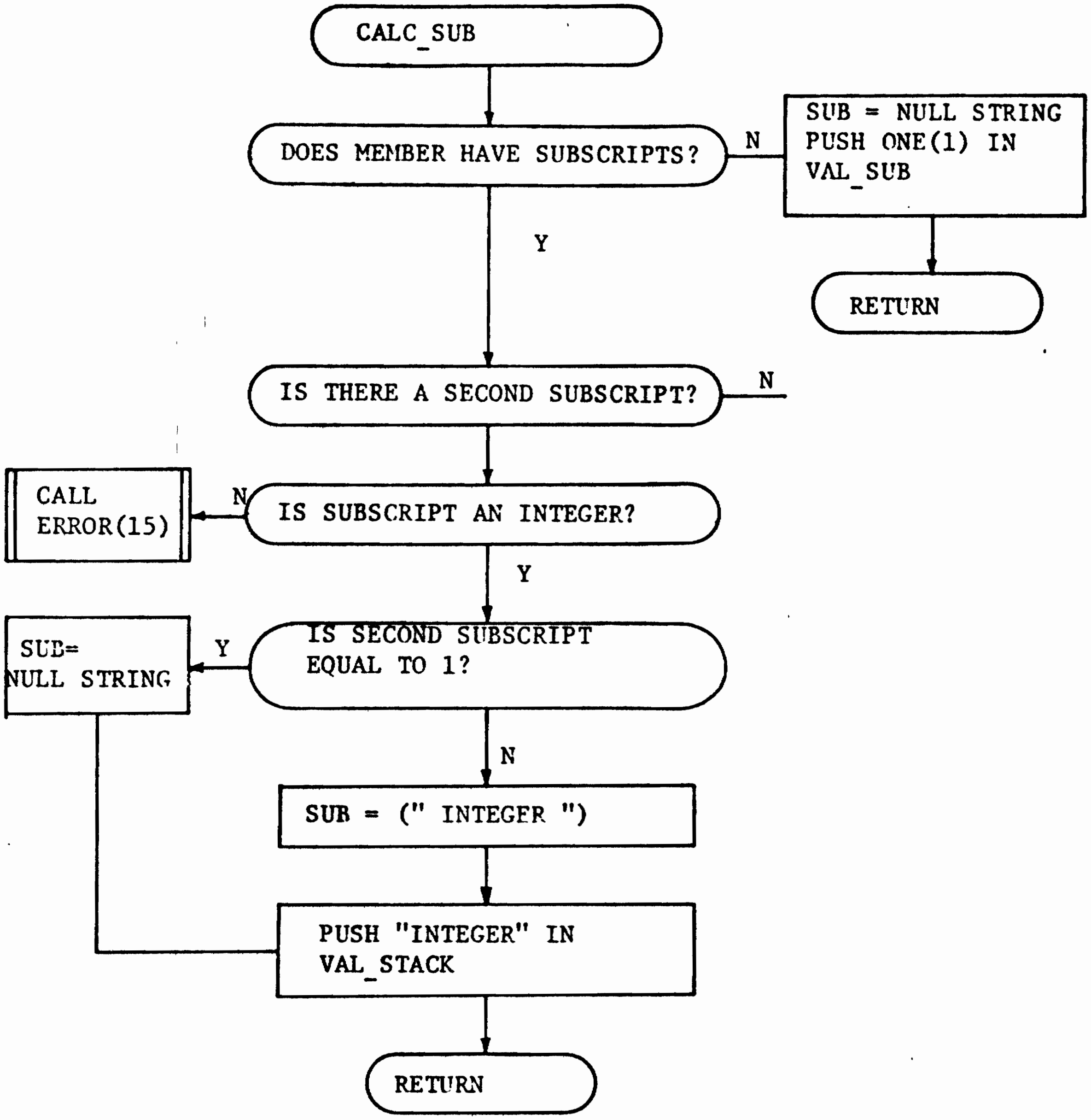


FIGURE 5-5F
CALC_SUB

set the field LEN in the PL/1 structure to the length of the current field being processed.

Upon entry to GEN_PARSE_FLD, the procedure GEN_PARSE_ MEM is called (see Section 5.5.3). Then the procedure which forms the fully qualified PL/1 name for the current FIELD is called, the name of this procedure is GEN_FLD_NAME (see Section 5.5.10). Then GEN_PARSE_FLD generates PL/1 code to set the field PTR. If the CHECK option was specified to the DDL compiler then GEN_PARSE_FLD generates PL/1 code to initialize to NULL the field PTR corresponding to current member. Next CALL_FLD_LEN (see Section 5.5.11) is called, this is a procedure which calculates the length of the current field, the length can be a constant and if so its value is returned in FLD_LEN, if variable, i.e., length to be given by a reference_name or a parameter statement or a user procedure written in DML or when the length of the current FIELD is to be delimited by a separator then CALC_FLD_LEN returns the name of a function to be called at execution time in FLD_PROC. After returning from CALC_FLD_LEN, GEN_PARSE_FLD generates the PL/1 code to set the field LEN in the PL/1 structure to the value returned in FLD_LEN or FLD_PROC.

If the TYPE of the current FIELD is CHAR, the CHAR_CODE is ASCII and the REC_MODE is MIXED then GEN_PARSE_FLD generates PL/1 code to call on the procedure that will perform the translations from ASCII to EBCDIC. If the CHAR_CODE is BCD then the PL/1 generated code will call on the procedure to translate

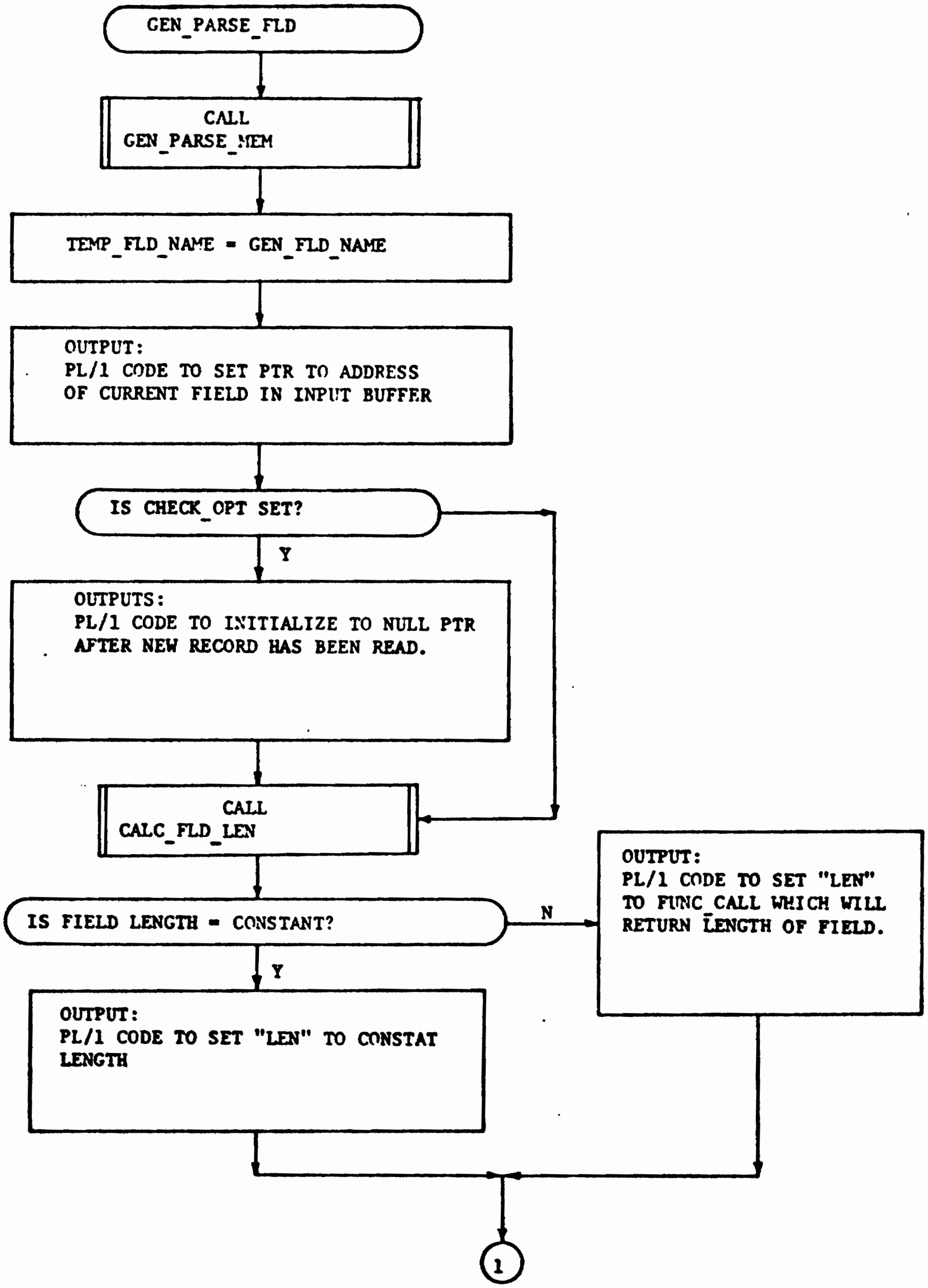

GEN_PARSE_FLD
CALL
GEN_PARSE_MEM
TEMP_FLD_NAME = GEN_FLD_NAME
OUTPUT:
PL/1 CODE TO SET PTR TO ADDRESS
OF CURRENT FIELD IN INPUT BUFFER
IS CHECK_OPT SET?
Y
OUTPUTS:
PL/1 CODE TO INITIALIZE TO NULL PTR
AFTER NEW RECORD HAS BEEN READ.
CALL
CALC_FLD_LEN
IS FIELD LENGTH = CONSTANT?
N
OUTPUT:
PL/1 CODE TO SET "LEN"
TO FUNC_CALL WHICH WILL
RETURN LENGTH OF FIELD.
Y
OUTPUT:
PL/1 CODE TO SET "LEN" TO CONSTAT
LENGTH
1

FIGURE 5-5G
GEN_PARSE_FLD

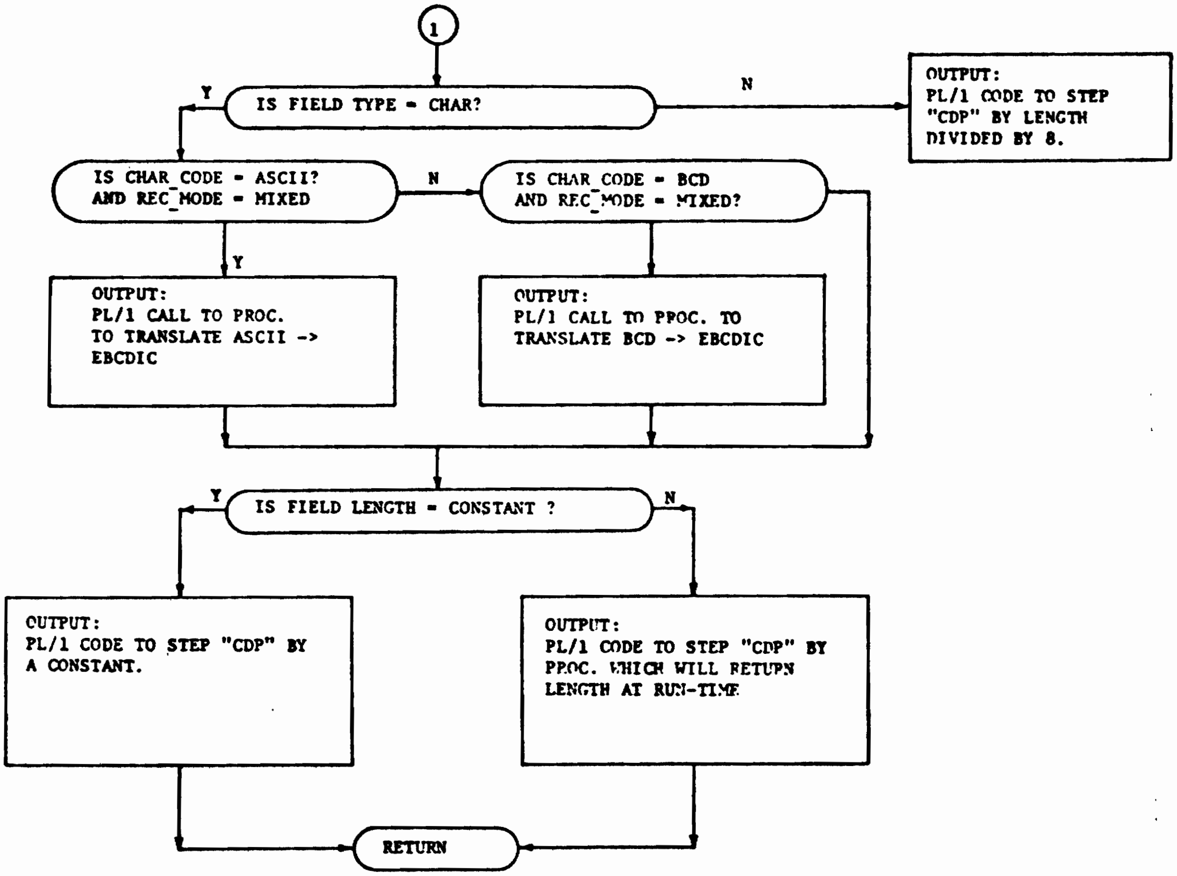

1
IS FIELD TYPE = CHAR?
Y
N
OUTPUT:
PL/1 CODE TO STEP "CDP" BY LENGTH DIVIDED BY 8.
IS CHAR_CODE = ASCII? AND REC_MODE = MIXED
N
IS CHAR_CODE = BCD AND REC_MODE = MIXED?
Y
OUTPUT:
PL/1 CALL TO PROC. TO TRANSLATE ASCII -> EBCDIC
OUTPUT:
PL/1 CALL TO PROC. TO TRANSLATE BCD -> EBCDIC
IS FIELD LENGTH = CONSTANT ?
Y
N
OUTPUT:
PL/1 CODE TO STEP "CDP" BY A CONSTANT.
OUTPUT:
PL/1 CODE TO STEP "CDP" BY PROC. WHICH WILL RETURN LENGTH AT RUN-TIME
RETURN

FIGURE 5-5G (continued)
GEN_PARSE_FLD

BCD to EBCDIC.

Finally GEN_PARSE_FLD generates PL/1 code to step CDP which is the pointer to the input record stored in the input buffer. If the TYPE of the FIELD is CHAR,CDP is stepped adding the length of the FIELD,if it is BIT the length is divided by 8.

5.5.8 GEN_CALC_LEN : ZERO_LEN

This procedure forms the PL/1 statements to accumulate the length of the current member into its ancestors.

ZERO_LEN is an entry point in GEN_CALC_LEN, when called it generates the PL/1 statements to set to zero the length of the ancestors of current members after a new input record has been read.

5.5.9 GEN_MATCH_ENDS

This procedure when called from WALK or CODE_GEN_MOVE generates the PL/1 END statement corresponding to a DO loop (if any). To perform its task GEN_MATCH_ENDS uses LABEL#_STACK which is a stack where the unique labels corresponding to the Do loop are stored and VAL_SUB which is a stack where the value of the subscripts is stored.

5.5.10 GEN_FLD_NAME

GEN_FLD_NAME is a procedure used to form the fully qualified PL/1 name for the name of current FIELD. When one of the ancestors or the current field itself have subscript the fully qualified name is form using a subscript the name of the iteration variable

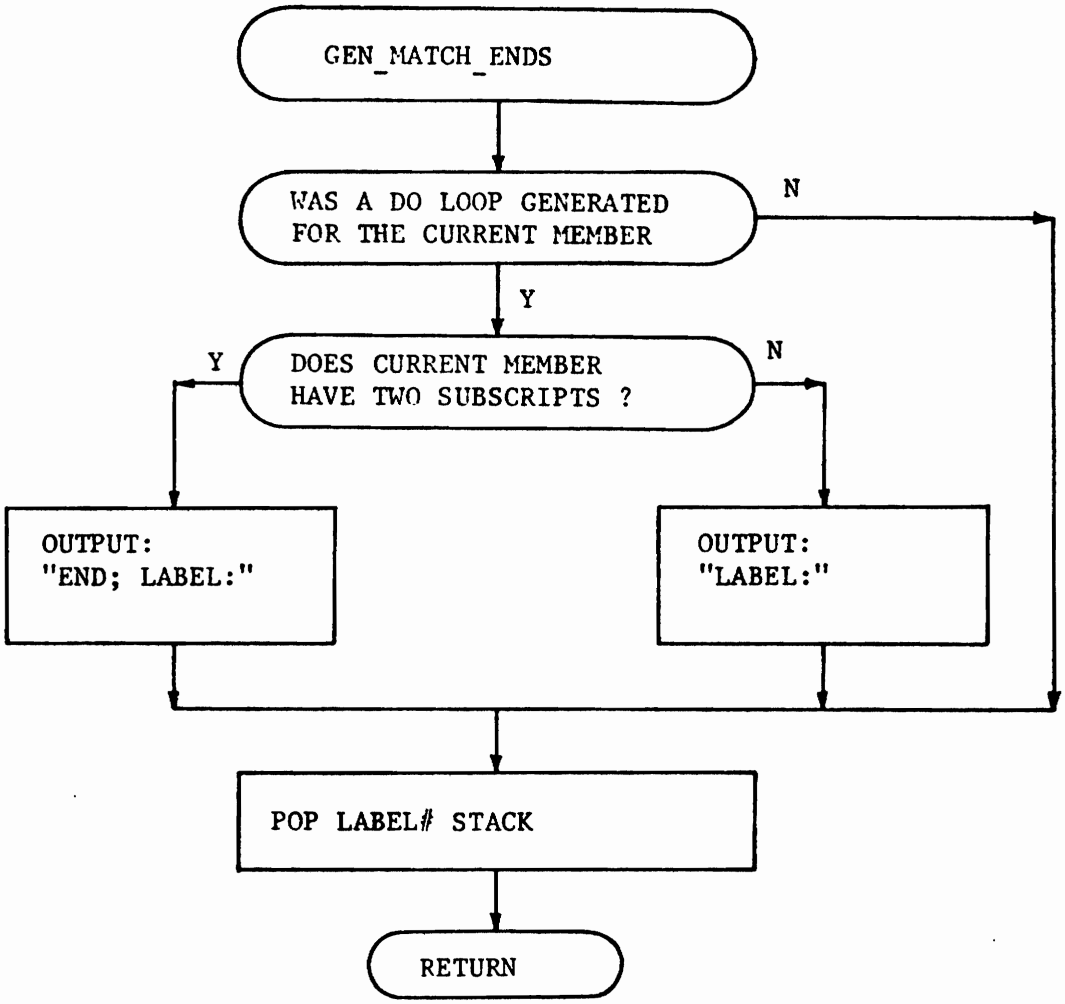

GEN_MATCH_ENDS
WAS A DO LOOP GENERATED FOR THE CURRENT MEMBER
N
Y
DOES CURRENT MEMBER HAVE TWO SUBSCRIPTS ?
Y
N
OUTPUT: "END; LABEL:"
OUTPUT: "LABEL:"
POP LABEL# STACK
RETURN

FIGURE 5-5I

GEN_MATCH_ENDS

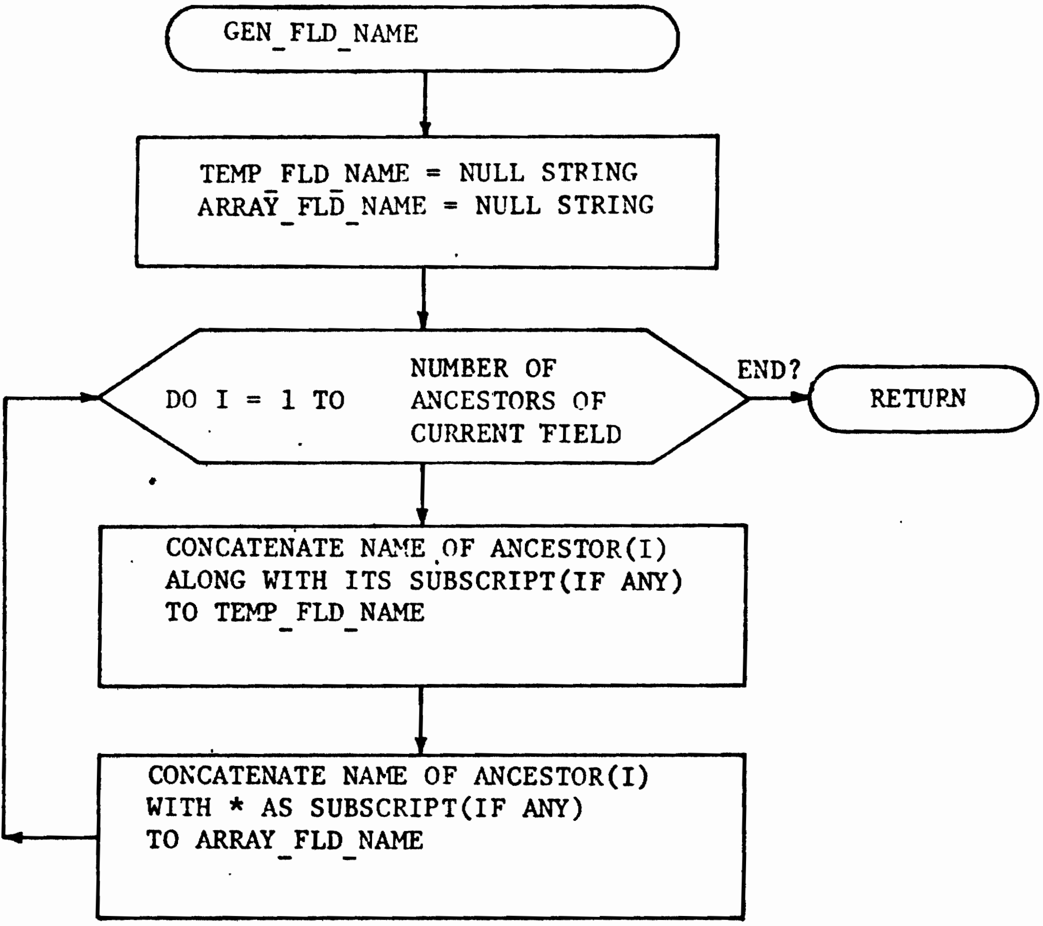

GEN_FLD_NAME
TEMP_FLD_NAME = NULL STRING
ARRAY_FLD_NAME = NULL STRING
DO I = 1 TO
NUMBER OF
ANCESTORS OF
CURRENT FIELD
END?
RETURN
CONCATENATE NAME OF ANCESTOR(I)
ALONG WITH ITS SUBSCRIPT(IF ANY)
TO TEMP_FLD_NAME
CONCATENATE NAME OF ANCESTOR(I)
WITH * AS SUBSCRIPT(IF ANY)
TO ARRAY_FLD_NAME

FIGURE 5-5J
GEN_FLD_NAME

(which at execution time will take the appropriate value). The fully qualified name is stored in TEMP_FLD_NAME, (e.g. A(I01).B.C, B.C(I03).D(I04)) .GEN_FLD_NAME also forms ARRAY_FLD_NAME, the difference between ARRAY_FLD_NAME and TEMP_FLD_NAME is that the name stored in the former contains * in place of the iteration variable. (e.g.,A(*).B.C , B.C(4). D(*))

5.5.11 CALC_FLD_LEN

This procedure returns the length of the current FIELD being processed in:

1) FLD_LEN if length of FIELD is constant
2) FLD_PROC if length of FIELD is given by
   a) Reference Name, or
   b) Parameter statement (DDL_COUNT or DDL_LENGTH)
   c) DML user supplied procedure, or
   d) DELIMETER

5.5.12 INIT_LEN
INIT_CNT

This procedure finds the value of the subscripts of all the ancestors of the current member and return its product. To carry on its task INIT_LEN or INIT_CNT uses VAL_SUB which is a stack containing the values of the subscripts of the current member and its ancestors. SUB_IND is a variable which contains the number of subscripts, so far stacked into VAL_SUB, up to the point of the invocation.

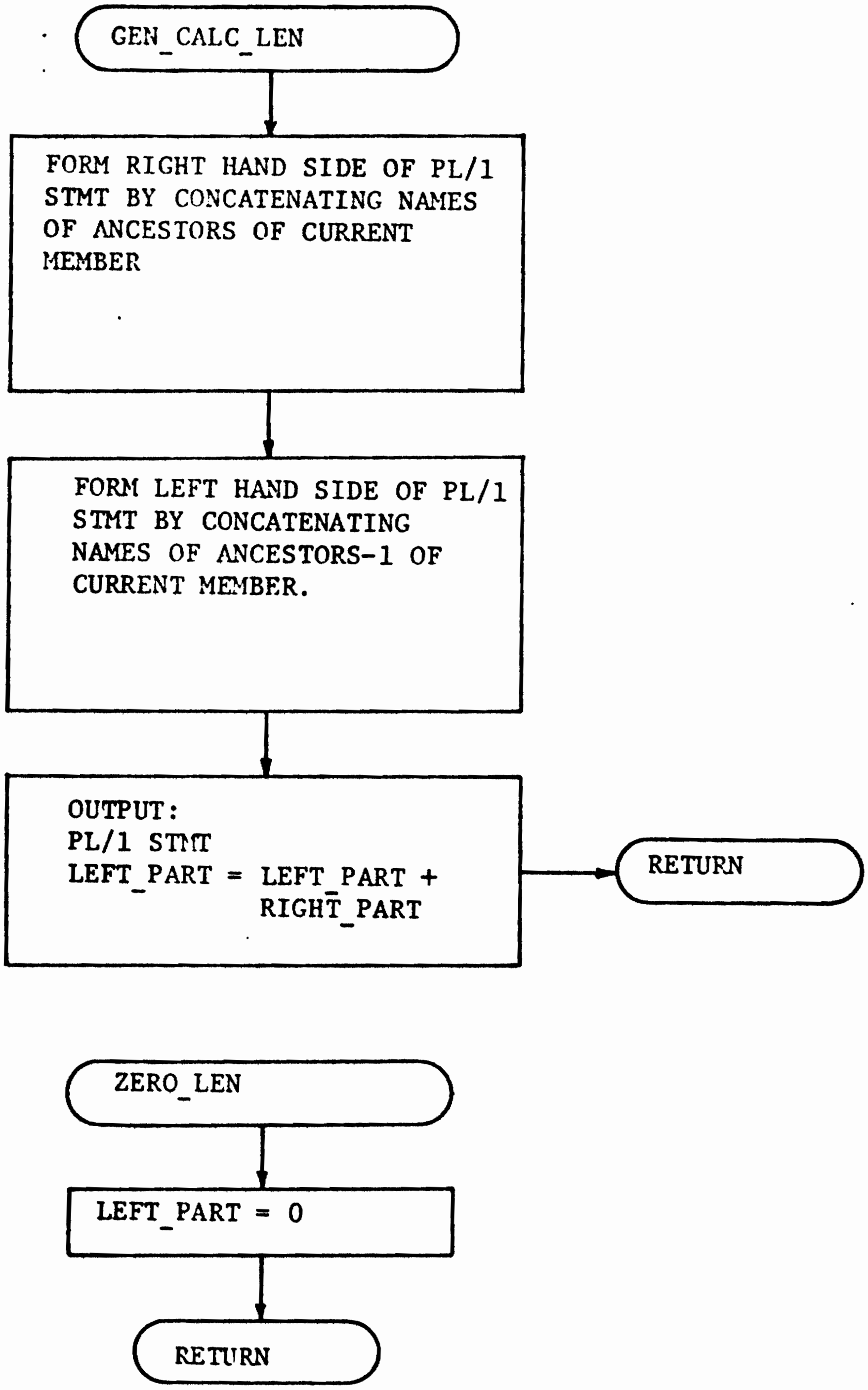

GEN_CALC_LEN
FORM RIGHT HAND SIDE OF PL/1 STMT BY CONCATENATING NAMES OF ANCESTORS OF CURRENT MEMBER
FORM LEFT HAND SIDE OF PL/1 STMT BY CONCATENATING NAMES OF ANCESTORS-1 OF CURRENT MEMBER.
OUTPUT:
PL/1 STMT
LEFT_PART = LEFT_PART +
RIGHT_PART
RETURN
ZERO_LEN
LEFT_PART = 0
RETURN

FIGURE 5-5H
GEN_CALC_LEN, ZERO_LEN

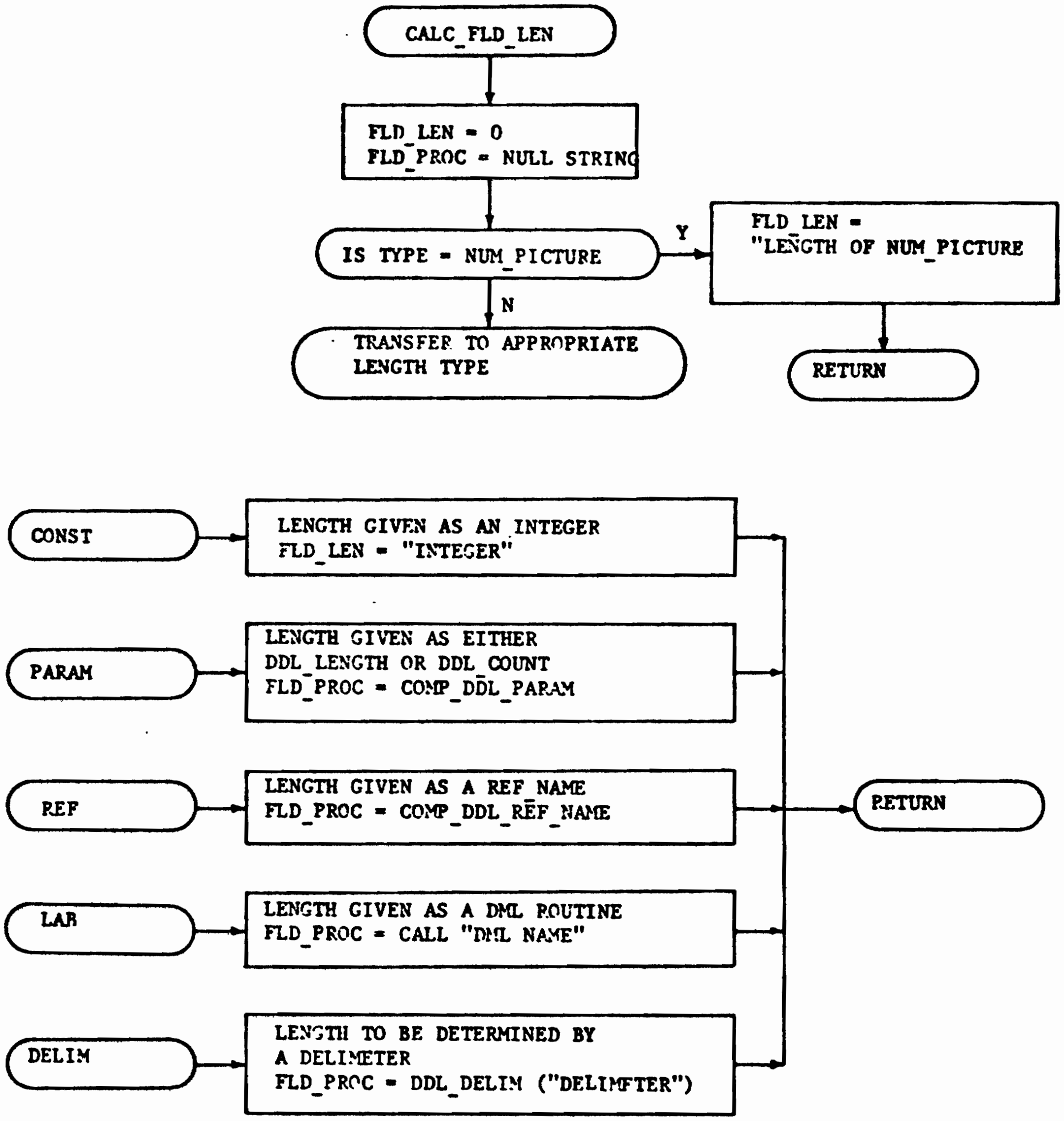


FIGURE 5-5K
CALC_FLD_LEN

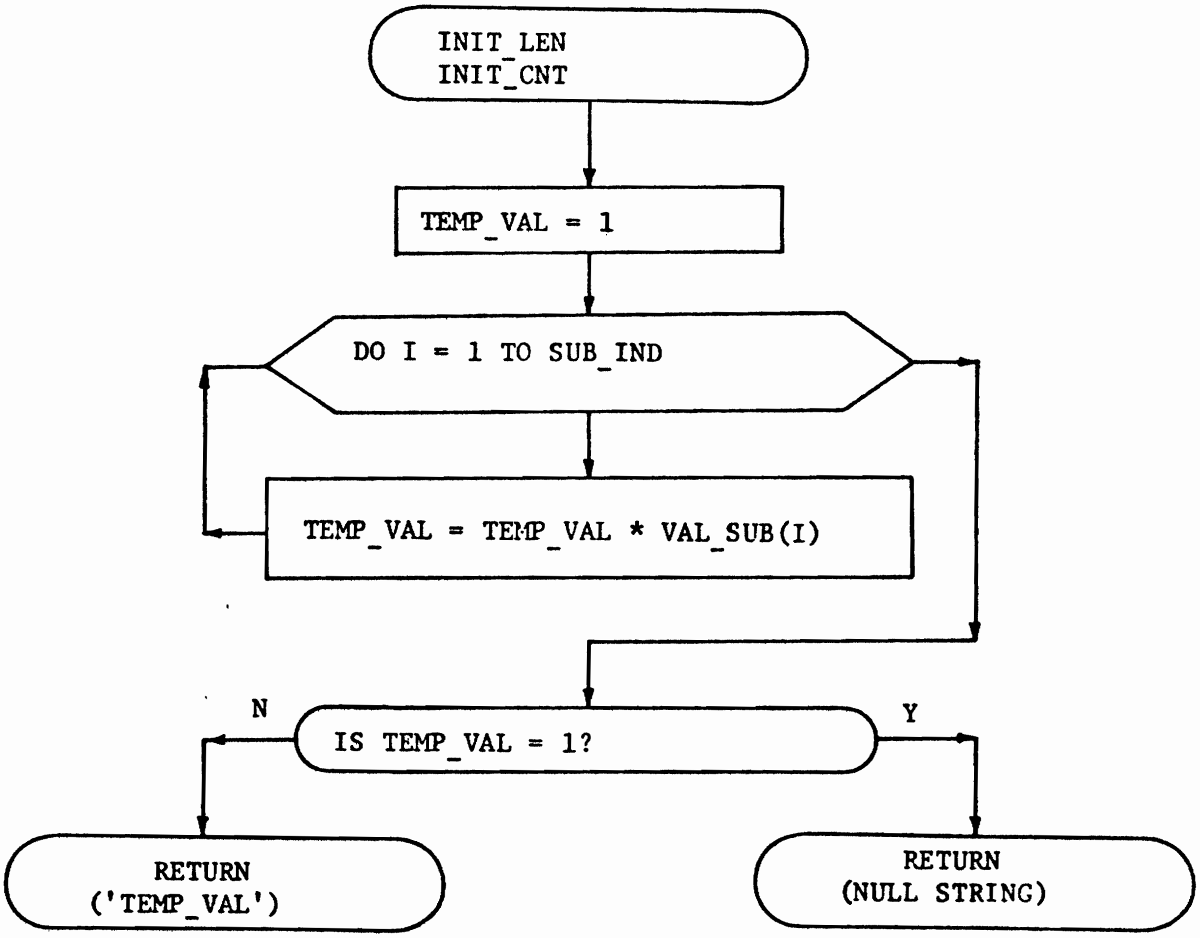


FIGURE 5-5L
INIT_LEN, INIT_CNT

5.5.13 GEN_INCR_COUNT

This procedure is called if the CNT_OPT was specified. It generates the PL/1 statements to accumulate the number of times the current repeating member have actually occurred in the input record. When one of the ancestors or the current member itself contains subscripts the fully qualified name is form using as subscript the name of the iteration variable (which at execution time will take the appropriate value). The fully qualified name is stored in TEMP_FLD_NAME. TEMP_FLD1 is similar to TEMP_FLD_NAME, the difference is that subscripts are formed with *.

5.5.14 SET_POS

This procedure is called if the SCAN statement appears in the DDL program. Its job is to generate PL/1 code to set CDP to the value given in the SCAN statement for the current member being processed. CDP can be set to an integer or to the value returned at run-time by the DML routine specified by the user.

5.5.15 GEN_DD

This procedure generates the DD card for tapes or disks from the information in the DISK or TAPE Data Table entry. GEN_DD has two entry points GEN_DD_TAPE and GEN_DD_DISK. The DD cards in JCL language are written to the OUT2 file. If GEN_DD_TAPE was called TAPE_DISK_FLAG is set to one, and if

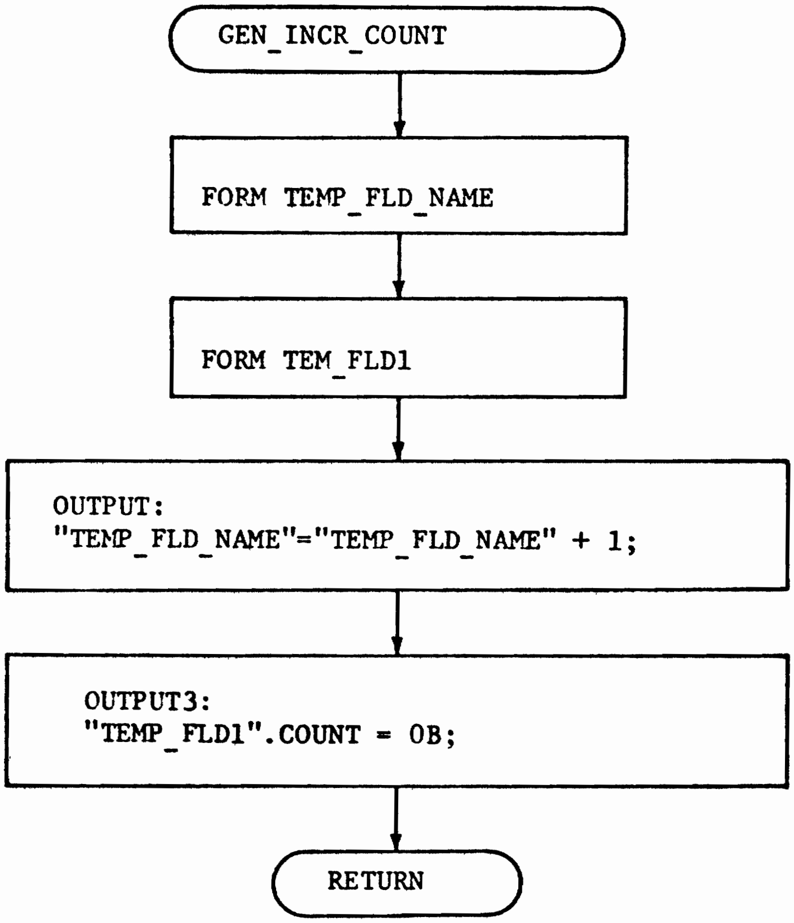


FIGURE 5-5M

GEN_INCR_COUNT

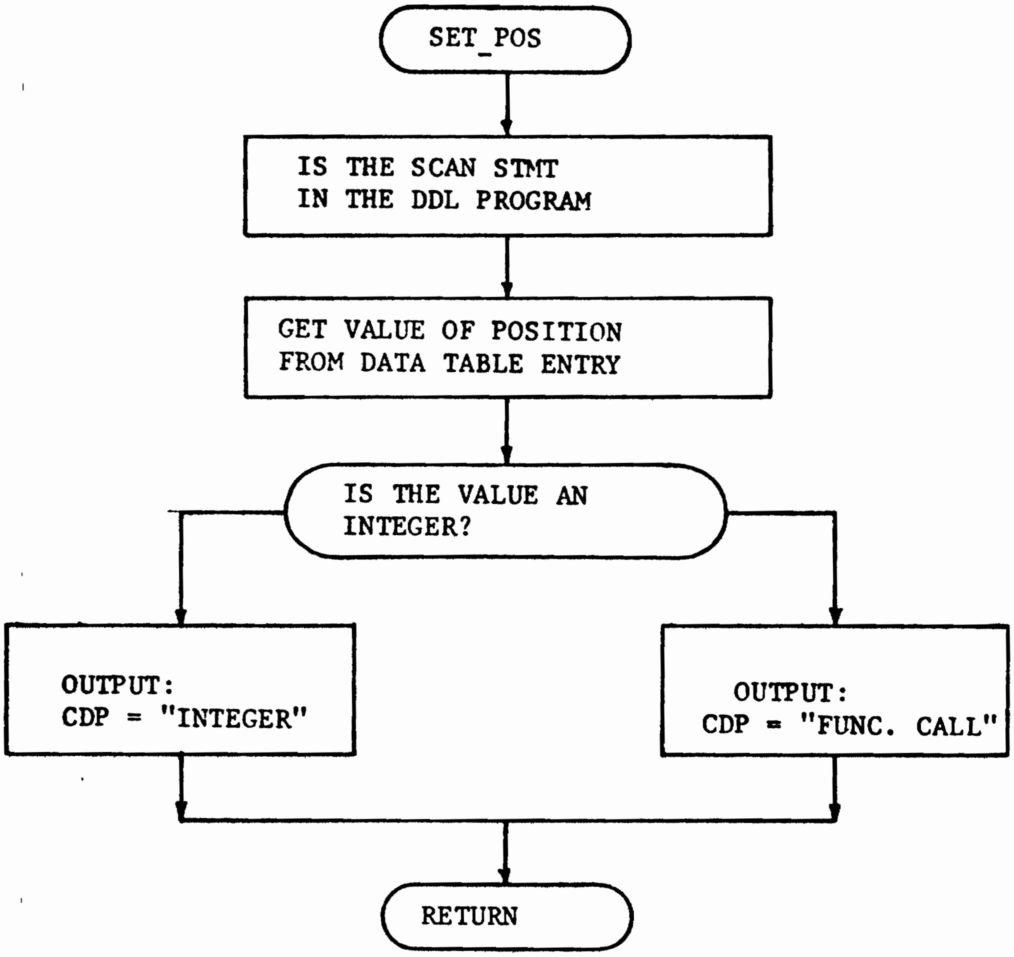

SET_POS
IS THE SCAN STMT IN THE DDL PROGRAM
GET VALUE OF POSITION FROM DATA TABLE ENTRY
IS THE VALUE AN INTEGER?
OUTPUT: CDP = "INTEGER"
OUTPUT: CDP = "FUNC. CALL"
RETURN

FIGURE 5-5N
SET_POS

GEN_DD_DISK was called TAPE_DISK_FLAG is set to zero.

Then GEN_DD sets #RECFM according to the following table

DISK OR TAPE RCD_FORMAT	#RECFM	
0	F	Fixed
1	FB	Fixed blocked
2	V	Variable
3	VB	Variable blocked
4	U	Undefined
5	VS	Variable spanned
6	VBS	Variable blocked spanned

Next, the BLKSIZE and LRECL information is saved in #BLKSIZE and #LRECL, deleting leading blanks. Then GEN_DD generates the first portion of the DD_CARD common to both TAPE and DISK (namely, DSN ,VOL = SER and DISP). CTL_CHAR information is saved in #CTL_CHAR.

If TAPE_DISK_FLAG is equal to one the UNIT and LABEL DD parameters are generated, and the density is saved in #DEN. Otherwise #DEN is set to null and the SPACE parameter along with its subparameters UNITS, QUANTITY and INCREMENT are generated.

Finally the last portion of the DD_CARD is generated i.e., the DCB parameter with subparameters of #RECFM, #LRECL, #BLKSIZE, #TRTCH and #DEN, and those not used in the current DDL program are already set to null.

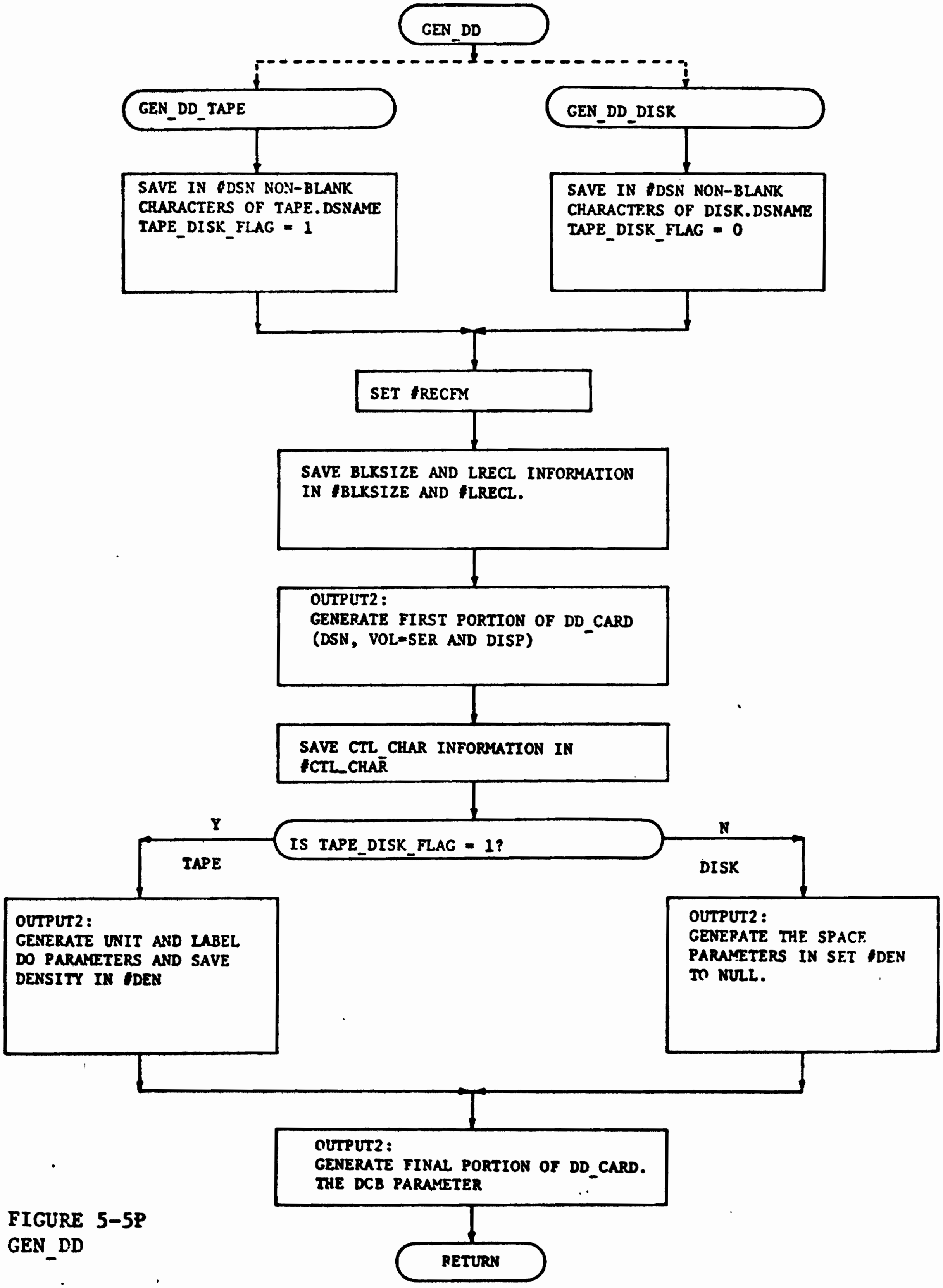

GEN_DD
GEN_DD_TAPE
GEN_DD_DISK
SAVE IN #DSN NON-BLANK CHARACTERS OF TAPE.DSNAME TAPE_DISK_FLAG = 1
SAVE IN #DSN NON-BLANK CHARACTERS OF DISK.DSNAME TAPE_DISK_FLAG = 0
SET #RECFM
SAVE BLKSIZE AND LRECL INFORMATION IN #BLKSIZE AND #LRECL.
OUTPUT2: GENERATE FIRST PORTION OF DD_CARD (DSN, VOL=SER AND DISP)
SAVE CTL_CHAR INFORMATION IN #CTL_CHAR
Y
IS TAPE_DISK_FLAG = 1?
N
TAPE
DISK
OUTPUT2: GENERATE UNIT AND LABEL DD PARAMETERS AND SAVE DENSITY IN #DEN
OUTPUT2: GENERATE THE SPACE PARAMETERS IN SET #DEN TO NULL.
OUTPUT2: GENERATE FINAL PORTION OF DD_CARD. THE DCB PARAMETER
RETURN

FIGURE 5-5P
GEN_DD

5.5.16 GEN_READ

This procedure generates PL/1 code to declare internal input buffers and to read the Source File. "User Record" refers to the logical record from the user's point of view. "System Record" refers to the record retrieved by the record statement. The PL/1 code that GEN_READ generates goes to OUT4 file.

Upon entry to this procedure STORAGE.RCD_SIZE is set to BLKSIZE if it is zero. If the STORAGE_RECORD_FORMAT is Variable (V or VS) then eight (8) is substracted from the RCD_SIZE to ignore the block count and record count. If the STORAGE_RECORD_FORMAT is variable blocked (VB or VBS) then four (4) is substracted from the RCD_SIZE to ignore the record count.

If USER_RCD_FLAG is zero i.e., length of system record is equal to length of user record, then the information is taken from the STORAGE Data Table for the Source File, then the generation of the PL/1 statements to declare INBUFS and INBUF takes place (as a character string and an overlaid array of characters, respectively, used as an input buffer for the Source File), and generates the PL/1 read statement to read a record from Source File (DDLSRC) into the input buffers.

If USER_RCD_FLAG is equal to one i.e., the user record is fixed in size but different to the length of the system record given in the STORAGE statement. Then if the RCD_FORMAT_

TYPE is less than two i.e., the system record is also fixed size, a check is made to test if the user input record size is a multiple of storage input record size, call ERROR(16) if not. If the user record size is equal to storage record size GEN_READ generates PL/1 code to declare INBUFS and INBUF and generates PL/1 code to read input record into input buffers. Generate PL/1 code to call on procedure to translate the input user record (INBUFS) from ASCII or BCD to EBCDIC if TRANS_FLAG have been set. Next GEN_READ returns control to calling procedure. If user input record size is different from system input record size then GEN_READ generates PL/1 code to declare: SYS_INBUF - input buffer to hold system input record. INBUFS - input buffer to hold user input record (its size is a multiple of SYS_INBUF). INBUF an array of character (1) overlaid on INBUFS. And finally INBUF2 an array of character strings (each has the length of SYS_INBUF) overlaid on INBUF. Next, GEN_READ generates PL/1 code to read system records into SYS_INBUF and to move them to INBUF. If last user input record is not filled, then pad it with blanks, and generate PL/1 GO TO statement to #DDL_PARSE, before finishing. Generate PL/1 code to call on procedure to translate the input user record (INBUFS) from ASCII or BCD to EBCDIC if TRANS_FLAG have been set. Next GEN_READ returns control to calling procedure.

If USER_RCD_FLAG is equal to two, i.e., the user input

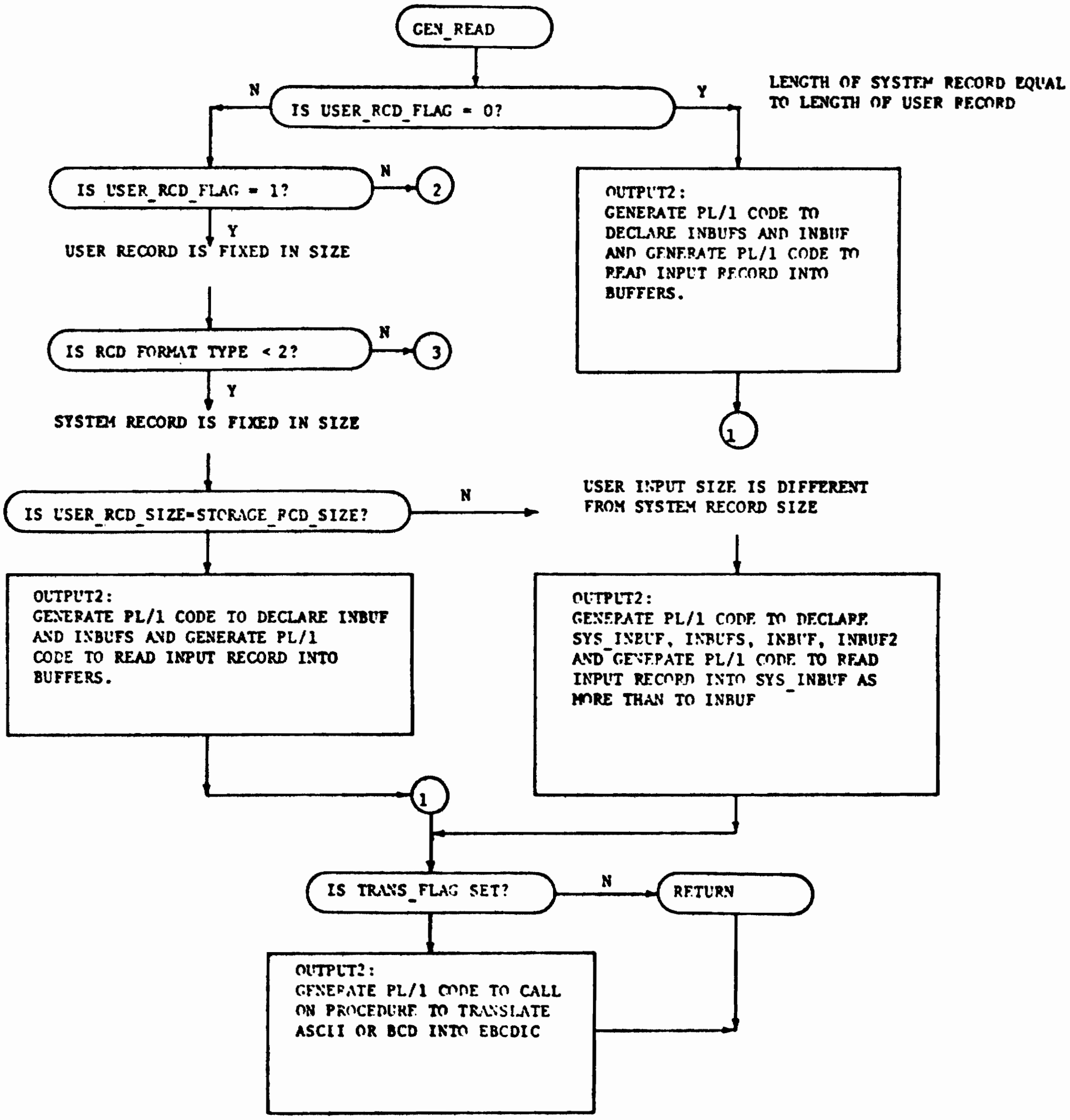

GEN_READ
IS USER_RCD_FLAG = 0?
N
Y
LENGTH OF SYSTEM RECORD EQUAL TO LENGTH OF USER RECORD
IS USER_RCD_FLAG = 1?
N
2
Y
USER RECORD IS FIXED IN SIZE
OUTPUT2:
GENERATE PL/1 CODE TO DECLARE INBUFS AND INBUF AND GENERATE PL/1 CODE TO READ INPUT RECORD INTO BUFFERS.
IS RCD FORMAT TYPE < 2?
N
3
Y
SYSTEM RECORD IS FIXED IN SIZE
1
IS USER_RCD_SIZE=STORAGE_RCD_SIZE?
N
USER INPUT SIZE IS DIFFERENT FROM SYSTEM RECORD SIZE
OUTPUT2:
GENERATE PL/1 CODE TO DECLARE INBUF AND INBUFS AND GENERATE PL/1 CODE TO READ INPUT RECORD INTO BUFFERS.
OUTPUT2:
GENERATE PL/1 CODE TO DECLARE SYS_INBUF, INBUFS, INBUF, INBUF2 AND GENERATE PL/1 CODE TO READ INPUT RECORD INTO SYS_INBUF AS MORE THAN TO INBUF
1
IS TRANS_FLAG SET?
N
RETURN
OUTPUT2:
GENERATE PL/1 CODE TO CALL ON PROCEDURE TO TRANSLATE ASCII OR BCD INTO EBCDIC

FIGURE 5-5Q
GEN_READ

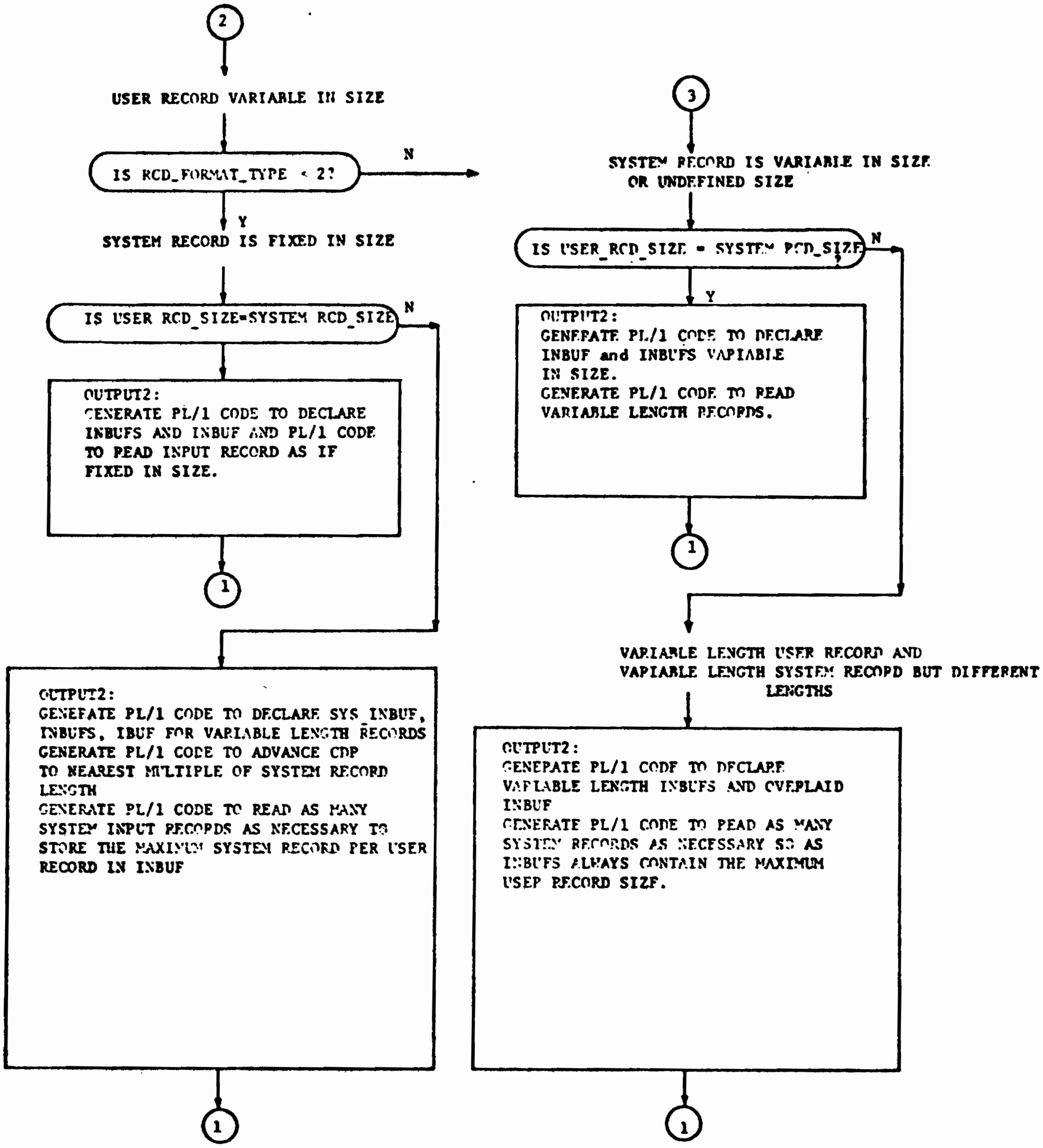

2
USER RECORD VARIABLE IN SIZE
IS RCD_FORMAT_TYPE < 2?
N
Y
SYSTEM RECORD IS FIXED IN SIZE
IS USER RCD_SIZE=SYSTEM RCD_SIZE
N
OUTPUT2:
GENERATE PL/1 CODE TO DECLARE INBUFS AND INBUF AND PL/1 CODE TO READ INPUT RECORD AS IF FIXED IN SIZE.
1
OUTPUT2:
GENERATE PL/1 CODE TO DECLARE SYS_INBUF, INBUFS, IBUF FOR VARIABLE LENGTH RECORDS
GENERATE PL/1 CODE TO ADVANCE CDP TO NEAREST MULTIPLE OF SYSTEM RECORD LENGTH
GENERATE PL/1 CODE TO READ AS MANY SYSTEM INPUT RECORDS AS NECESSARY TO STORE THE MAXIMUM SYSTEM RECORD PER USER RECORD IN INBUF
1
3
SYSTEM RECORD IS VARIABLE IN SIZE OR UNDEFINED SIZE
IS USER_RCD_SIZE = SYSTEM RCD_SIZE
N
Y
OUTPUT2:
GENERATE PL/1 CODE TO DECLARE INBUF and INBUFS VARIABLE IN SIZE.
GENERATE PL/1 CODE TO READ VARIABLE LENGTH RECORDS.
1
VARIABLE LENGTH USER RECORD AND VARIABLE LENGTH SYSTEM RECORD BUT DIFFERENT LENGTHS
OUTPUT2:
GENERATE PL/1 CODE TO DECLARE VARIABLE LENGTH INBUFS AND OVERLAID INBUF
GENERATE PL/1 CODE TO READ AS MANY SYSTEM RECORDS AS NECESSARY SO AS INBUFS ALWAYS CONTAIN THE MAXIMUM USER RECORD SIZE.
1

FIGURE 5-5Q (continued)
GEN_READ

record is variable in size. Then if STORAGE.RCD_FORMAT_TYPE is less than two (i.e., fixed length system records but a variable number of them form a user record). Then if user record size is equal to storage record size GEN_READ generates PL/1 code to declare INBUF and INBUFS and generates PL/1 code to read input record as if fixed length. Then if TRANS_FLAG is set GEN_READ generates PL/1 code to call on procedure to translate input user record (INBUFS) from ASCII or BCD to EBCDIC, and returns control to calling procedure. If user record size is different from storage record size GEN_READ generates PL/1 code to declare: SYS_INBUF - buffer for system record, INBUF, INBUFS - buffers for variable length user record. Then GEN_READ generates PL/1 code to advance "CDP" (the current data pointer in the input buffer) to the nearest multiple of system record length. Next GEN_READ generates PL/1 code to read as many system input records as necessary to store the maximum system records per user record in INBUF. Finally, if TRANS_FLAG is set GEN_READ generates PL/1 code to call on procedure to translate input user record (INBUF) from ASCII or BCD to EBCDIC and then GEN_READ returns control to procedure which call on GEN_READ.

If the USER_RCD_FLAG is equal to two and STORAGE.RCD_FORMAT_TYPE is not less than two - i.e., variable or undefined system record length specified with a variable length user record length. And if user record size is equal to storage record size - i.e., system variable length record size is the same as the user record. GEN_READ will generate PL/1 to declare INBUF, INBUFS and generate the appropriate PL/1 read statement. If user record size is different from storage record size - i.e., variable user record and variable system record of different size each. Then GEN_READ generates. PL/1 code to declare INBUFS with variable length and INBUF to be overlaid on INBUFS. Then GEN_READ generates PL/1 code to read as many system records as necessary so as INBUFS always contain the maximum user record size.

After either of the two cases above GEN_READ if TRANS_FLAG is set generates PL/1 code to call on procedure to translate (INBUFS) from ASCII or BCD to EBCDIC, then returning control to procedure which call on GEN_READ.

5.6 Phase 2B of the DDL Compiler - Code Generation (Data Movement)

Phase 2B of the DDL compiler is a set of programs which uses the Data Tables for the Target file and the mappings specified in the Target Fields to generate PL/1 code for data movement. After Phase 2B is completed the DML statements are read by the DDL compiler and they are merged with the code produced during code generation phase.

5.6.1 CODE_GEN_MOVE

CODE_GEN_MOVE is called from DDLCOMP after the code generation for parsing the input record has been completed. CODE_GEN_MOVE uses the Symbol and Data Table for the Target File.

CODE_GEN_MOVE is the monitor for the generation of PL/1 statements to move data from the Source Record into the Target Record using the information given in the FIELD statement.

Upon entry to CODE_GEN_MOVE the generation of PL/1 declare statements for the run-time variables used in Data Movement takes place. Next CODE_GEN_MOVE generates the PL/1 declare statements for the DML routines specified in the Target File as conversion routines. The pointer to the Symbol Table were the Target File name is stored is taken from the Data Table entry for the CONVERT statement. Then CODE_GEN_MOVE generates PL/1 code to declare OUTCDP (current data pointer for output buffer were the output record is being formed). The character code in which the output file is to be output is saved, and the pointer to the target record in the Symbol Table is taken from the Data Table entry of the FILE statement, and the pointer to the Target Data Table entry is saved.

Using the pointer to the target record name in the Symbol Table, the Data Table entry of the Target RECORD statement is found. And the Target record size (if specified) is saved, and the pointer to the Target Record Data Table entry is saved.

Next CODE_GEN_MOVE gets the Data Table entry for the STORAGE statement using the pointer in the File Statement to the Symbol Table entry where the name of the STORAGE Statement is kept.

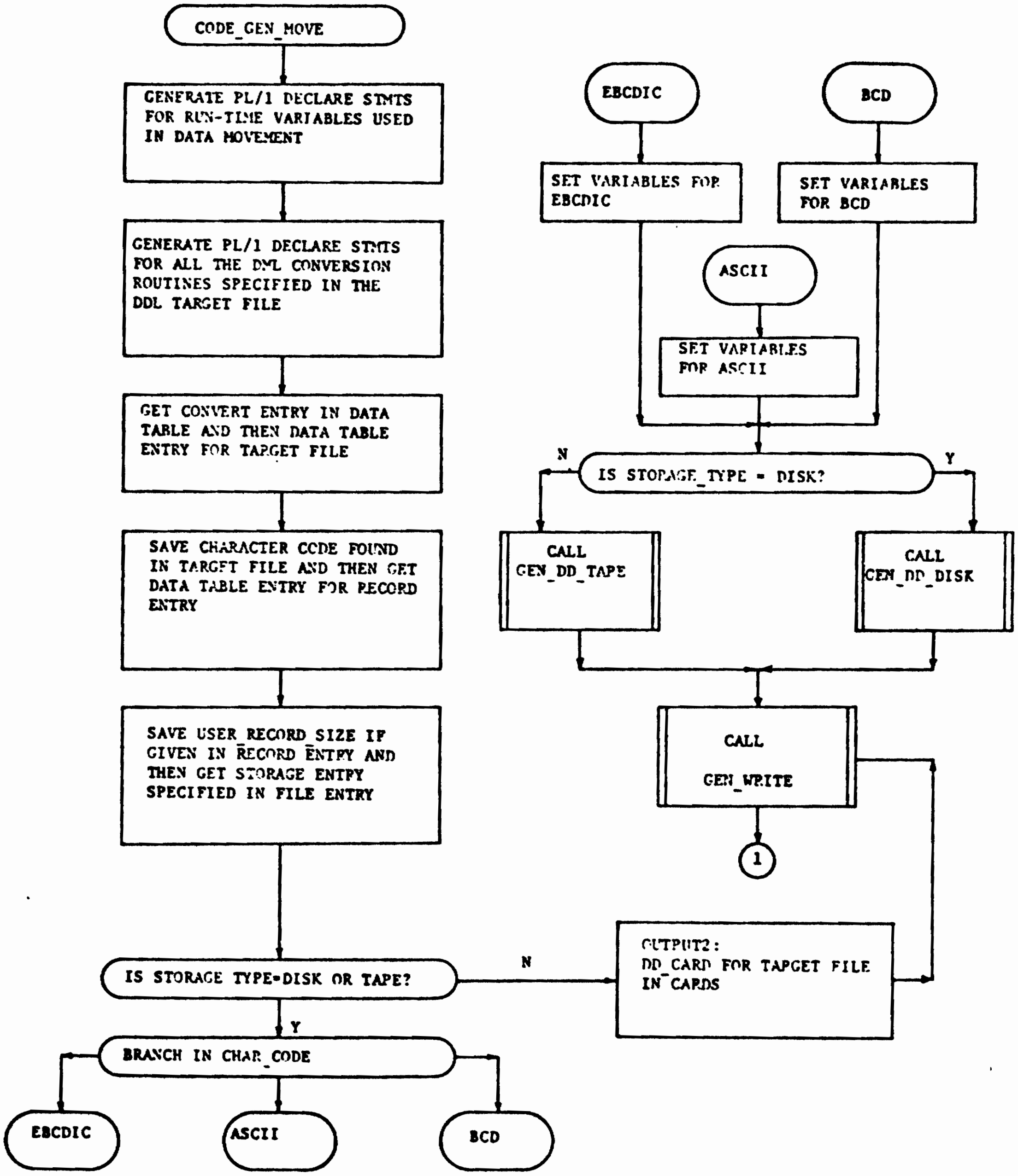

CODE_GEN_MOVE
GENERATE PL/1 DECLARE STMTS FOR RUN-TIME VARIABLES USED IN DATA MOVEMENT
GENERATE PL/1 DECLARE STMTS FOR ALL THE DML CONVERSION ROUTINES SPECIFIED IN THE DDL TARGET FILE
GET CONVERT ENTRY IN DATA TABLE AND THEN DATA TABLE ENTRY FOR TARGET FILE
SAVE CHARACTER CODE FOUND IN TARGET FILE AND THEN GET DATA TABLE ENTRY FOR RECORD ENTRY
SAVE USER RECORD SIZE IF GIVEN IN RECORD ENTRY AND THEN GET STORAGE ENTRY SPECIFIED IN FILE ENTRY
IS STORAGE TYPE=DISK OR TAPE?
N
OUTPUT2: DD_CARD FOR TARGET FILE IN CARDS
Y
BRANCH IN CHAR_CODE
EBCDIC
ASCII
BCD
EBCDIC
SET VARIABLES FOR EBCDIC
BCD
SET VARIABLES FOR BCD
ASCII
SET VARIABLES FOR ASCII
N
IS STORAGE_TYPE = DISK?
Y
CALL GEN_DD_TAPE
CALL GEN_DD_DISK
CALL GEN_WRITE
1

FIGURE 5-6A
CODE_GEN_MOVE

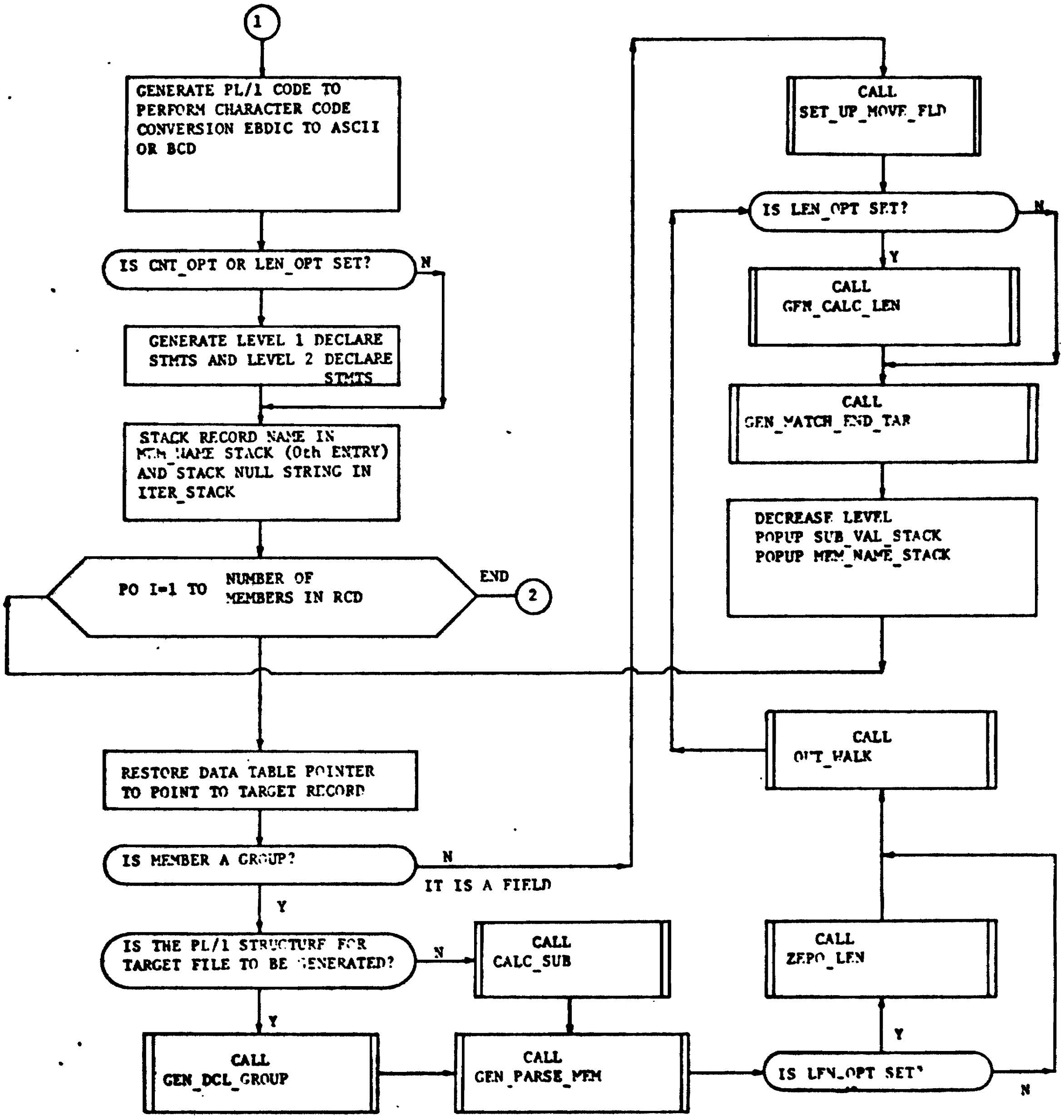

1
GENERATE PL/1 CODE TO PERFORM CHARACTER CODE CONVERSION EBDIC TO ASCII OR BCD
IS CNT_OPT OR LEN_OPT SET?
N
GENERATE LEVEL 1 DECLARE STMTS AND LEVEL 2 DECLARE STMTS
STACK RECORD NAME IN MEM_NAME STACK (0th ENTRY) AND STACK NULL STRING IN ITER_STACK
DO I=1 TO NUMBER OF MEMBERS IN RCD
END
2
RESTORE DATA TABLE POINTER TO POINT TO TARGET RECORD
IS MEMBER A GROUP?
N
IT IS A FIELD
Y
IS THE PL/1 STRUCTURE FOR TARGET FILE TO BE GENERATED?
N
Y
CALL CALC_SUB
CALL GEN_DCL_GROUP
CALL GEN_PARSE_MEM
IS LEN_OPT SET?
Y
N
CALL ZERO_LEN
CALL OUT_WALK
CALL SET_UP_MOVE_FLD
IS LEN_OPT SET?
N
Y
CALL GEN_CALC_LEN
CALL GEN_MATCH_END_TAR
DECREASE LEVEL
POPUP SUB_VAL_STACK
POPUP MEM_NAME_STACK

FIGURE 5-6A (continued)
CODE_GEN_MOVE

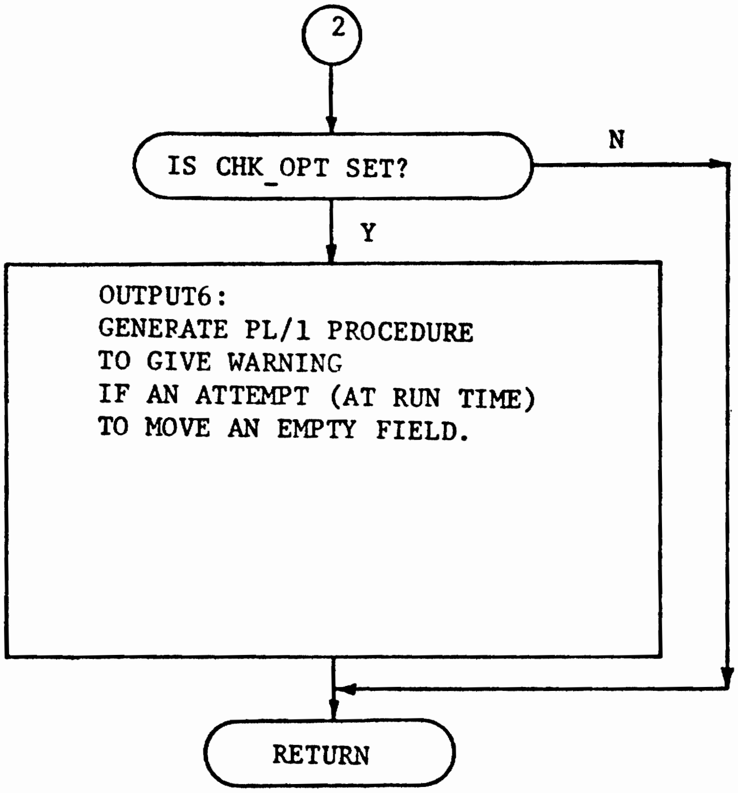


FIGURE 5-6A (continued)
CODE_GEN_MOVE

Next CODE_GEN_MOVE test the TYPE of storage which can be TAPE, DISK or CARD. If TAPE or DISK it uses the character code to set appropriate variables for EBCDIC, ASCII or BCD whichever the case may be. Then if storage TYPE is TAPE, GEN_DD_TAPE is called (see Section 5.5.15) otherwise GEN_DD_DISK is called (see Section 5.5.15). If storage TYPE is CARD the DD_CARD for the Target file is generated in it is output to file OUT2.

After the DD_CARDS have been generated CODE_GEN_MOVE calls on GEN_WRITE, (see Section 5.6.12) which is a procedure used to generate PL/1 code to declare internal output buffers and to generate PL/1 code to WRITE the Target record into the output file. If the character code is ASCII or BCD and the recording mode is ALL_CHAR then CODE_GEN_MOVE generates PL/1 code to form tables to translate EBCDIC into ASCII or BCD.

Next CODE_GEN_MOVE checks if the OUTLEN or OUTCNT were specified, if they were then the PL/1 structure corresponding to the Target record is to be generated since information regarding the LENGTH and/or COUNT of Target fields is to be used in the DDL specification of the Target record. If OUTLEN nor OUTCNT were specified then the PL/1 structure is not to be generated since there is no need for it.

The main task of CODE_GEN_MOVE is to process the members (and its submembers) of the Target RECORD statement. To do this the pointer to the Data Table for the RECORD statement is reset and the pointer to the name of the first member in the Symbol Table is used to get its corresponding Data Table entry. If the TYPE of member is GROUP then if the PL/1 structure is to be generated GEN_DCL_GROUP is called (see Section 5.5.4) otherwise CALL_SUB is called (see Section 5.5.6). Now, since the current member can be a repeating member GEN_PARSE_MEM (see Section 5.5.3) is called to generate PL/1 DO_LOOP for this member. If the OUTLEN option was specified then ZERO_LEN is called (see Section 5.5.8). Then since member is a GROUP its submembers must be processed, so OUT_WALK (see Section 5.6.2) is called.

If the member TYPE is FIELD then CODE_GEN_MOVE calls on SET_UP_MOVE_FLD (see Section 5.6.3) which is a procedure which will process current field and will call on appropriate procedures to generate PL/1 code to move the Source Fields into the current Field being processed. After returning from SET_UP_MOVE_FLD, CODE_GEN_MOVE checks if OUTLEN was specified, if so GEN_CALC_LEN is called (see Section 5.5.11).

After returning from processing a GROUP or a FIELD CODE_GEN_MOVE calls on GEN_MATCH_END_TAR and then returns to process the next member in the Target Record Statement.

After all the members of the RECORD statement has been processed GEN_CODE_MOVE,if the CHECK option was specified , generates PL/1 code to form a procedure which will give a warning to the DDL user if he attempts to move an empty source field into the target record.

5.6.2 OUT_WALK

OUT_WALK is called from CODE_GEN_MOVE (see Section 5.6.1), it is used to process the members of the GROUP statement specified in the definition of the Target File. OUT_WALK is a recursive procedure since the members of the current member can themselves be GROUPS.

Upon entry to the procedure the pointers to members of current groups being processed are saved, and then the first member is processed, if it is a FIELD then call SET_UP_MOVE_FLD (see Section 5.6.3). If the member is a GROUP and if the PL/1 structure is to be generated then OUT_WALK calls on GEN_DCL_GROUP (see Section 5.5.4), otherwise CALL_SUB (see Section 5.5.6) is called. Then since the current GROUP can be repeating the procedure to generate PL/1 DO_LOOPS is called, this procedure is GEN_PARSE_MEM (see Section 5.5.3). If the OUTLEN option was specified then OUT_WALK calls in ZERO_LEN (see Section 5.5.8). Finally since current member is a GROUP its submembers must be processed so OUT_WALK call on itself.

After processing the current member if the OUTLEN option was specified then GEN_CALL_LEN is called (see Section 5.5.8). Finally OUT_WALK calls on GEN_MATCH_END_TAR (see Section 5.6.9) and returns to process next member of current GROUP.

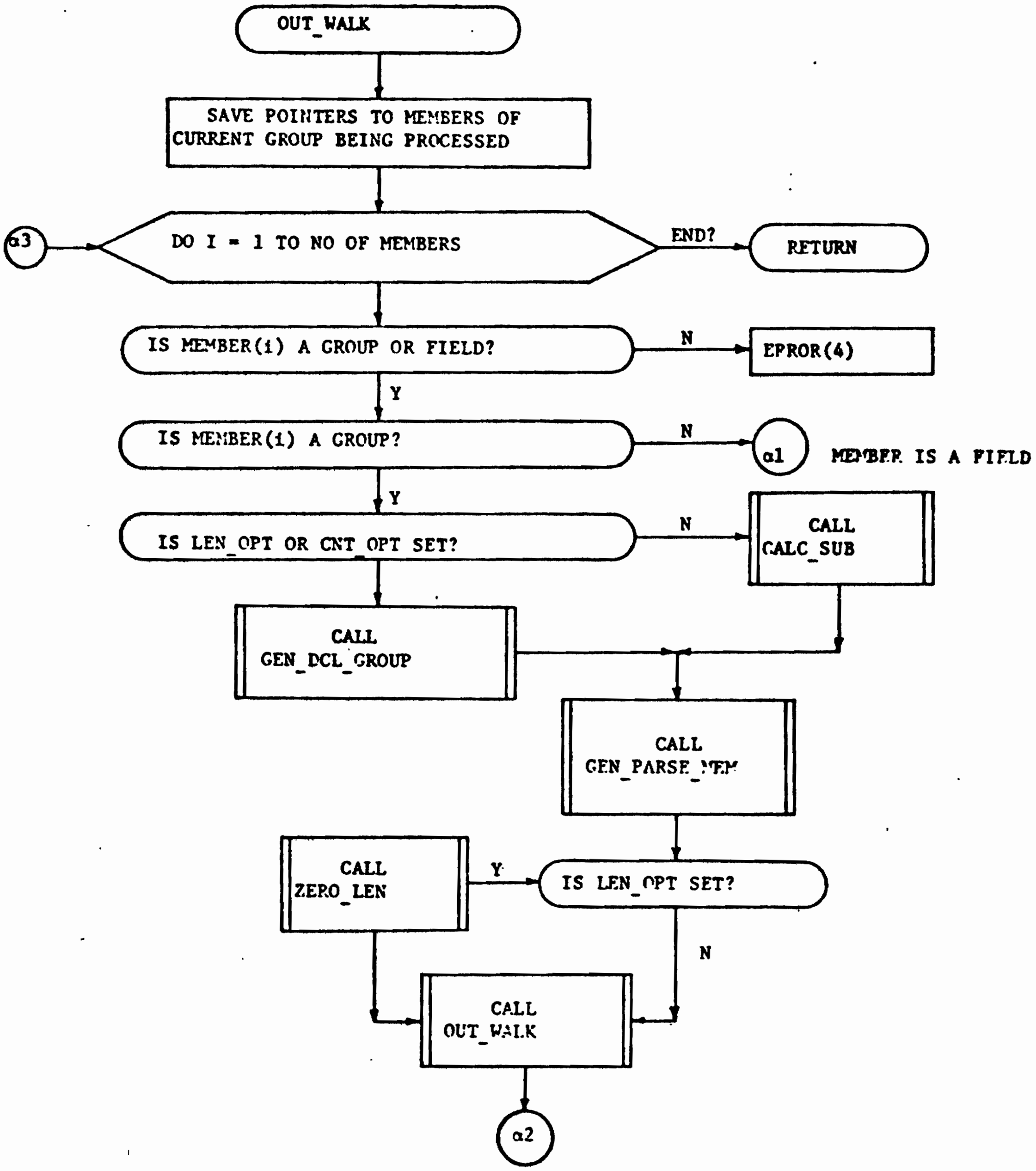


FIGURE 5-6B
OUT_WALK

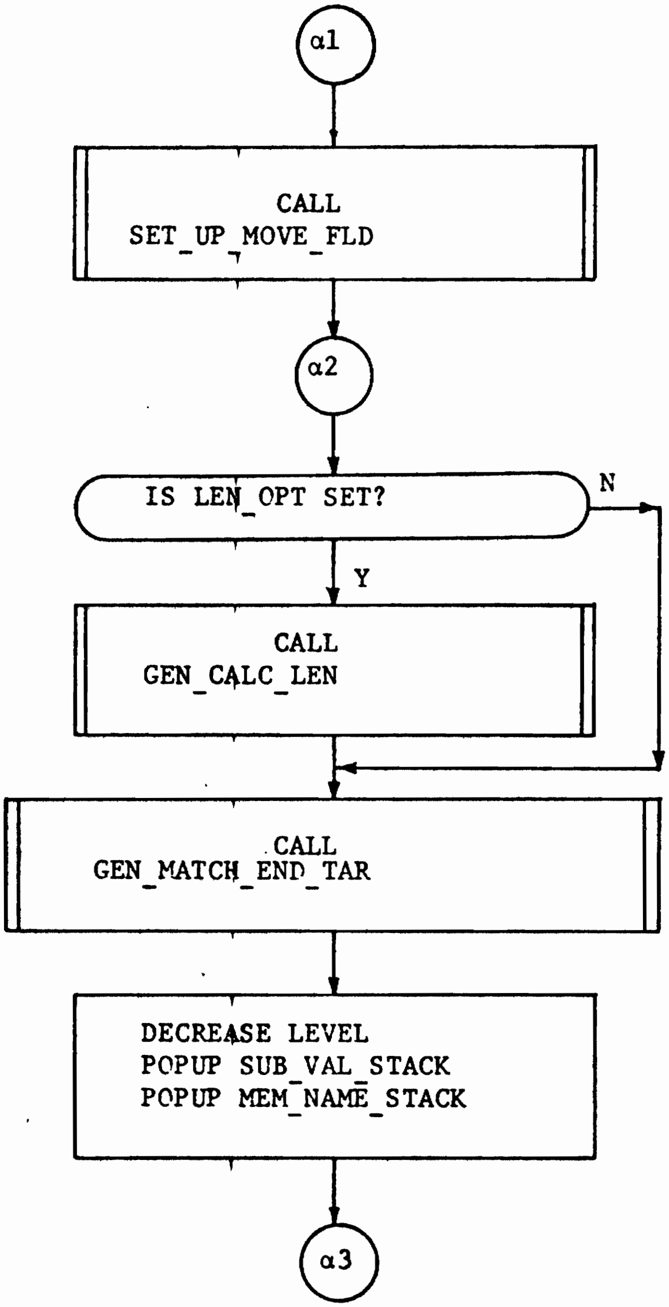


FIGURE 5-6B (continued)
OUT_WALK

When all the submembers of the member in the RECORD statement have been processed OUT_WALK returns control to CODE_GEN_MOVE.

5.6.3 SET_UP_MOVE_FLD

Upon entry to this procedure a check is made to find out if the PL/1 structure for the Target Record is to be generated, if so, GEN_FLD_DCL (see Section 5.5.5) is called, otherwise CALL_SUB (see Section 5.5.6) is called. Next SET_UP_MOVE_FLD calls on GET_SOURCE_NAME (see Section 5.6.4) this procedure forms the fully qualified Source Name specified in the current FIELD statement being processed. Next, since the current FIELD can be repeating GEN_PARSE_MEM (see Section 5.5.3) is called to form the PL/1 DO_loop. If any member in the Source Name just formed have two subscripts SET_UP_MOVE_FLD calls on SET_MIN_MAX (see Section 5.6.5), to generate PL/1 code to set the MINSUB and MAXSUB vectors with the appropriate values for as many elements as there are names in the Source Name with two subscripts. Then SET_MIN_MAX generates PL/1 code to call on the procedure SET_SOURCE_SUBS, this procedure when called at run-time returns the appropriate subscripts values for the Source Name.

Next SET_UP_MOVE_FLD determines the length of the Target field. This length may have been specified as *, meaning that the Target field is to have the same length of the Source field or if conversion was specified then the length of the Target

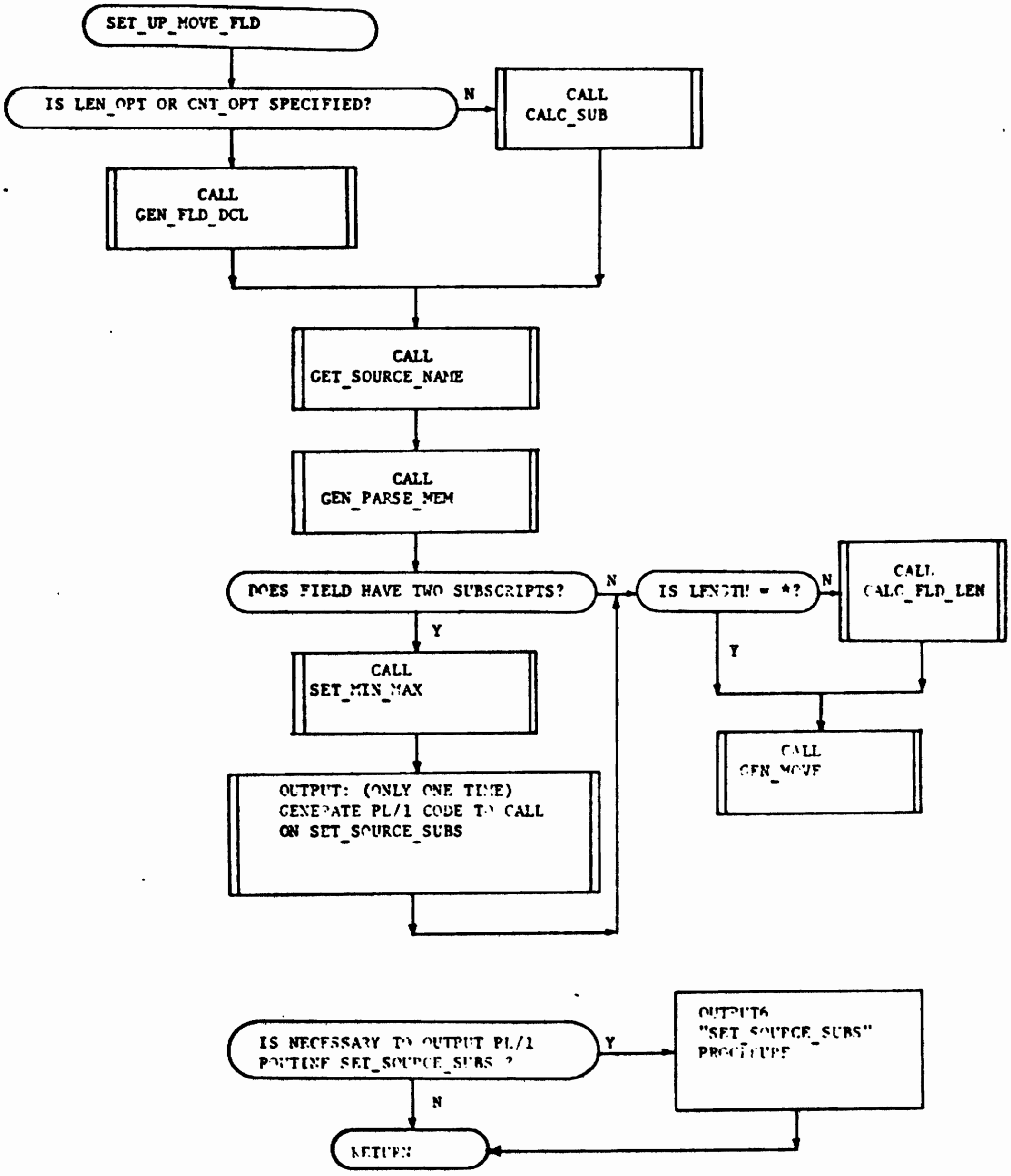


FIGURE 5-6C
SET_UP_MOVE_FLD

field is to be equal to the length of CONV_FLD_CHAR if Target field TYPE is CHAR or equal to the length of CONV_FLD_BIN if the TYPE is BIT. CONV_FLD_CHAR or CONV_FLD_BIN are global run-time variables which are set by the DML routine specified in the conversion parameter of the Target Field statement. If the length was not specified by * then SET_UP_MOVE_FLD calls on CALC_FLD_LEN (see Section 5.5.11) to determine the length of the Target field.

After the length of the Target field has been determined GEN_MOVE is called (see Section 5.6.6), this procedure generates PL/1 code to move the data in the Source field into the Target field.

Finally if any of the names in a Source Name had two subscripts and if the run-time procedure SET_SOURCE_SUBS have not been output into file OUT6, then SET_UP_MOVE_FLD outputs it into file OUT6 and returns control to calling program.

5.6.4 GET_SOURCE_NAME

This procedure forms the Full Source Name in PL/1 form, and if any name in Source Name does have two subscripts generates code to set IFIRST# and LAST# which the parameters of the run-time procedure SET_SOURCE_SUBS.

Upon entry to this procedure a check is made to find if a mapping has been specified, if not an error message is given and the process terminates. If a mapping was specified, then if TYPE of the Target field is NUM_PICTURE and if the

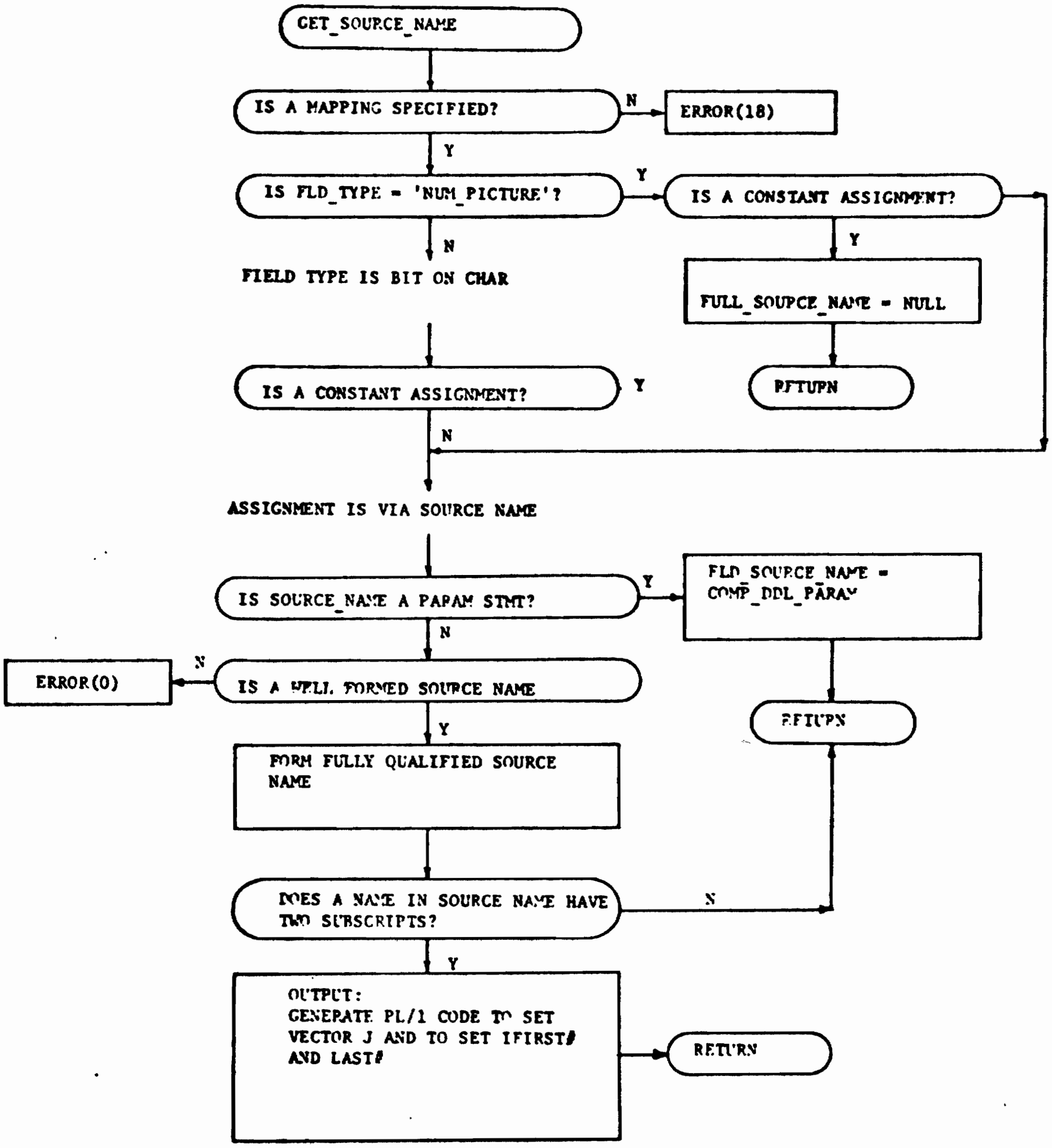

GET_SOURCE_NAME
IS A MAPPING SPECIFIED?
N
ERROR(18)
Y
IS FLD_TYPE = 'NUM_PICTURE'?
Y
IS A CONSTANT ASSIGNMENT?
Y
N
FIELD TYPE IS BIT ON CHAR
FULL_SOURCE_NAME = NULL
IS A CONSTANT ASSIGNMENT?
Y
RETURN
N
ASSIGNMENT IS VIA SOURCE NAME
IS SOURCE_NAME A PARAM STMT?
Y
FLD_SOURCE_NAME = COMP_DDL_PARAM
N
ERROR(0)
N
IS A WELL FORMED SOURCE NAME
RETURN
Y
FORM FULLY QUALIFIED SOURCE NAME
DOES A NAME IN SOURCE NAME HAVE TWO SUBSCRIPTS?
N
Y
OUTPUT:
GENERATE PL/1 CODE TO SET VECTOR J AND TO SET IFIRST# AND LAST#
RETURN

FIGURE 5-6D
GET_SOURCE_NAME

mapping of the form " <- "string"" then FULL_SOURCE_NAME is set to the null string and GET_SOURCE_NAME returns control to calling program.

If the TYPE of the Target field is CHAR_PICTURE, CHAR or BIT and if the mapping is of the form " <- 'string'" then FULL_SOURCE_NAME is set to the null string and control is given back to the calling program.

In either of the above two cases, if the mapping is of the form " <= 'SOURCE_NAME'" then if the Source Name is a Parameter statement (i.e., DDL_COUNT or DDL_LENGTH) then FULL_SOURCE_NAME is set to the result returned by COMP_DDL_PARAM (see Section 5.6.8). Otherwise a check is made to see if the Source Name is a valid one.

If the Source_Name passed the test then the Full PL/1 source name is formed and FULL_SOURCE_NAME is set to such name.

Next GET_SOURCE_NAME generates PL/1 to set the array J to the appropriate values (see Example below).

Finally, if any of the names in FULL_SOURCE_NAME had two subscripts GET_SOURCE_NAME generates PL/1 code to initialize IFIRST# and LAST# to the appropriate values. To better explain the above paragraph lets assume that the Source Name given by the DDL user was the following:

```
A(1:2) .B(2:3).C.D(3:4)
```

The corresponding FULL_SOURCE_NAME in PL/1 form is

A(J(1)). B(J(2)). C . D(J(3)) and IFIRST# is set to 1 and LAST# is set to 3. Then when at run-time SET_SOURCE_SUBS (IFIRST#, LAST#) is called it returns the following values

1st call J(1) = 1 ,J(2) = 2 ,J(3) = 3

2nd call J(1) = 1 ,J(2) = 2 J(3) = 4

3rd call J(1) = 1 J(2) = 3 J(3) = 3

4th call J(1) = 1 J(2) = 3 J(3) = 4

5th call J(1) = 2 J(2) = 2 J(3) = 3

6th call J(1) = 2 J(2) = 2 J(3) = 4

7th call J(1) = 2 J(2) = 3 J(3) = 3

8th call J(1) = 2 J(2) = 3 J(3) = 4

For the above Example GET_SOURCE_NAME generates the following PL/1 statements to initialize the array J

J(1) = 0 → value of first subscript of A minus 1

J(2) = 1 → value of first subscript of B minus 1

J(3) = 2 → value of first subscript of D minus 1, in order to understand better why the array J is initialized with the value of the first subscript minus one see Section 5.6.5 where SET_MIN_MAX and the run-time procedure SET_SOURCE_SUBS are given.

To achieve the generation of the appropriate subscript for the array J a count is kept (#SAVE_CNT), every time a name does have subscripts this count is increased by 1, and when a new member is taken from the Target_Record #SAVE_CNT is reset to zero.

5.6.5 SET_MIN_MAX

This procedure is used to generate the PL/1 code to set MINSUB vectors when at least one name in the Source Name have two subscripts.

Upon entry to this procedure a check is made in the names that form Source Name, if a name does have two subscripts #SAVE_CNT is increased by 1 and FORM_SUB (see Section 5.6.10) is called, to find value of 1st subscript. The subscript can be constant or variable, then the appropriate PL/1 code is generated to set MINSUB (#SAVE_CNT) to the value returned by FORM_SUB. Then FORM_SUB is called to find value of second subscript and MAXSUB (#SAVE_CNT) is set to the value returned by FORM_SUB.

Take for example the Source Name given in Section 5.6.4

"A(1:2). B(2:3). C . D(3:4)" and if #SAVE_CNT is equal to zero then SET_MIN_MAX will generate the following PL/1 code

```
MINSUB(1) = 1
MAXSUB(1) = 2
MINSUB(2) = 2
MAXSUB(2) = 3
MINSUB(3) = 3
MAXSUB(3) = 4,
```

the value assigned to MINSUB is equal to the value returned by FORM_SUB. Every time SET_SOURCE_SUBS is called J(i) is increased by 1 if J(i) is less than MAXSUB(i),

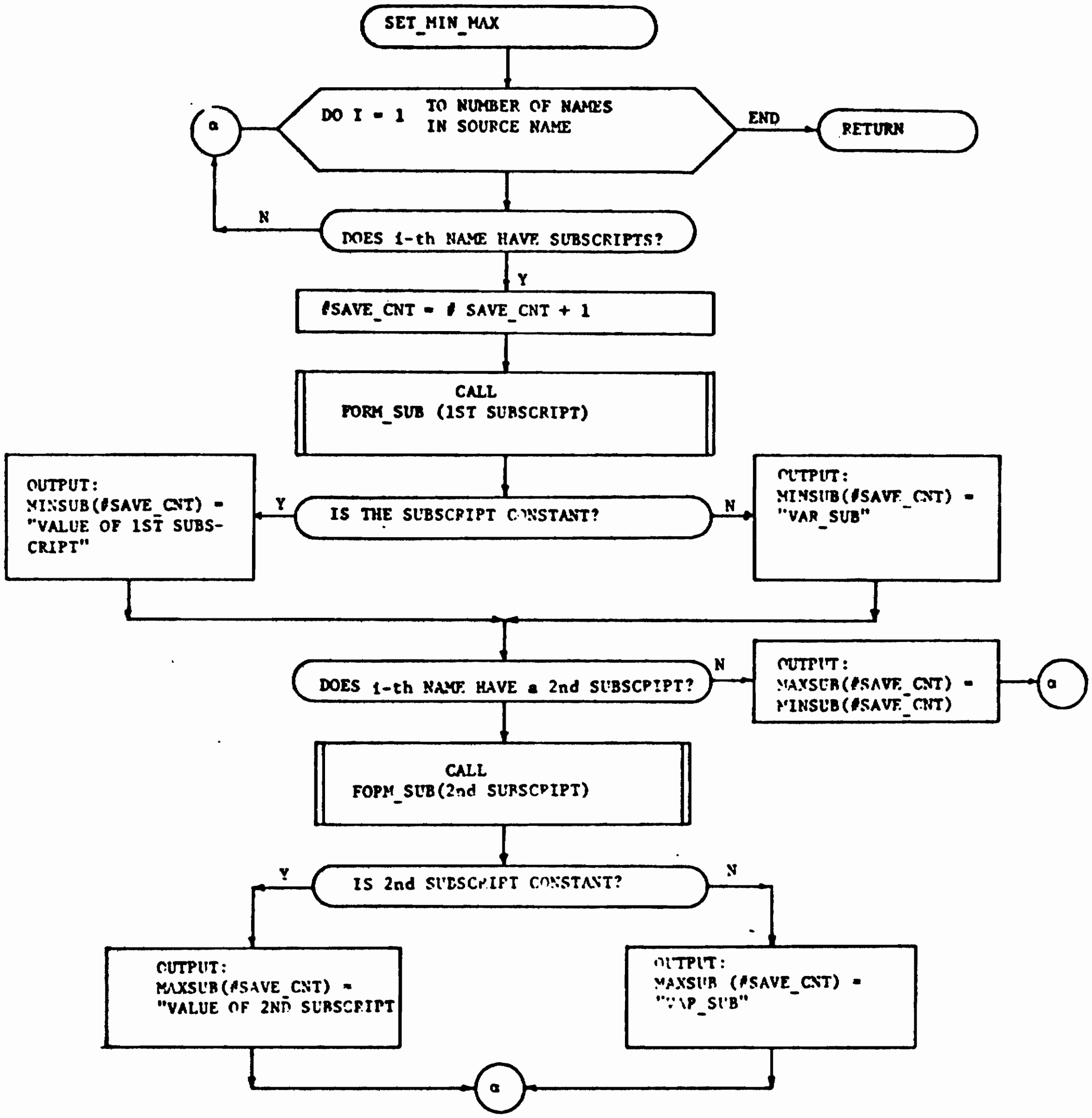

SET_MIN_MAX
DO I = 1 TO NUMBER OF NAMES IN SOURCE NAME
END
RETURN
α
N
DOES i-th NAME HAVE SUBSCRIPTS?
Y
#SAVE_CNT = # SAVE_CNT + 1
CALL FORM_SUB (1ST SUBSCRIPT)
Y
IS THE SUBSCRIPT CONSTANT?
N
OUTPUT: MINSUB(#SAVE_CNT) = "VALUE OF 1ST SUBSCRIPT"
OUTPUT: MINSUB(#SAVE_CNT) = "VAR_SUB"
DOES i-th NAME HAVE a 2nd SUBSCRIPT?
N
OUTPUT: MAXSUB(#SAVE_CNT) = MINSUB(#SAVE_CNT)
α
CALL FORM_SUB(2nd SUBSCRIPT)
Y
IS 2nd SUBSCRIPT CONSTANT?
N
OUTPUT: MAXSUB(#SAVE_CNT) = "VALUE OF 2ND SUBSCRIPT
OUTPUT: MAXSUB (#SAVE_CNT) = "VAR_SUB"
α

FIGURE 5-6E

SET_MIN_MAX

when it reaches the value of MAXSUB(i) J(i) is reset with the value of MINSUB(i). Remember that J(i) was initially set to the value of first subscript minus 1 (see Section 5.6.4).

The PL/1 code for SET_SOURCE_SUBS is the following:

```
SET_SOURCE_SUBS:  PROCEDURE (FIRST,LAST);
DCL (FIRST,LAST) FIXED BIN;
DCL IDDL FIXED BIN STATIC;
DO IDDL = LAST TO FIRST BY -1;
    IF J(IDDL) < MAXSUB (IDDL) THEN
      DO;
          J(IDDL) = J(IDDL) + 1B;
          RETURN;
      END;
    J(IDDL) = MINSUB (IDDL) - 1B;
END;
END SET_SOURCE_SUBS;
```

5.6.6 GEN_MOVE

This procedure processes the mappings " <- 'string'" or " <= 'SOURCE_NAME'", and generates PL/1 code to move the source data into the target field, generates PL/1 code to call on the conversion routine if specified. The Target record is formed in OUTBUF.

Upon entry to GEN_MOVE a test is made to find out the type of mapping, if it is of the form "<- 'string'". If the target field TYPE is CHAR_PICTURE, or CHAR then GEN_MOVE generates

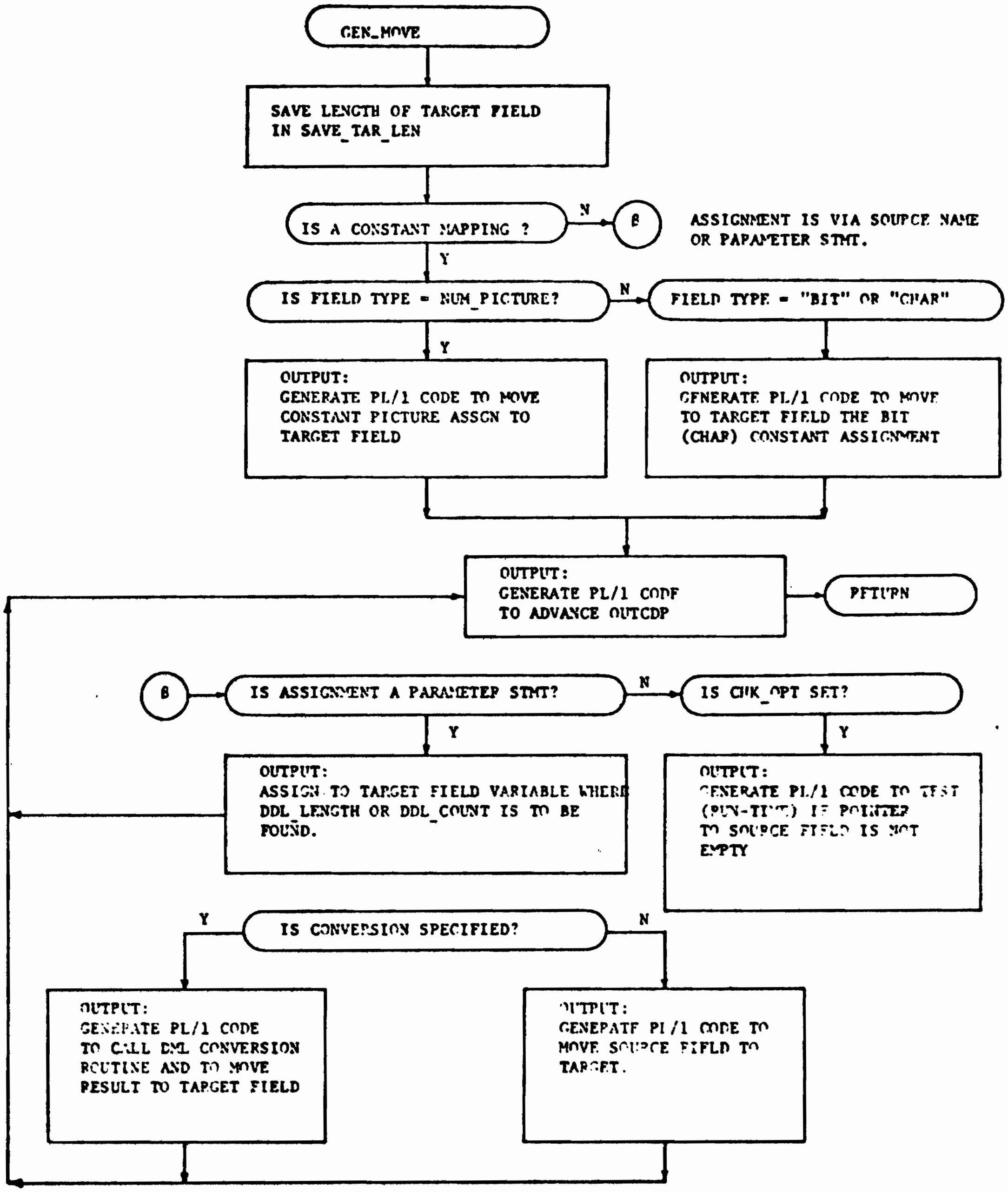

GEN_MOVE
SAVE LENGTH OF TARGET FIELD IN SAVE_TAR_LEN
IS A CONSTANT MAPPING ?
N
B
ASSIGNMENT IS VIA SOURCE NAME OR PARAMETER STMT.
Y
IS FIELD TYPE = NUM_PICTURE?
N
FIELD TYPE = "BIT" OR "CHAR"
Y
OUTPUT: GENERATE PL/1 CODE TO MOVE CONSTANT PICTURE ASSGN TO TARGET FIELD
OUTPUT: GENERATE PL/1 CODE TO MOVE TO TARGET FIELD THE BIT (CHAR) CONSTANT ASSIGNMENT
OUTPUT: GENERATE PL/1 CODE TO ADVANCE OUTCDP
RETURN
B
IS ASSIGNMENT A PARAMETER STMT?
N
IS CHK_OPT SET?
Y
OUTPUT: ASSIGN TO TARGET FIELD VARIABLE WHERE DDL_LENGTH OR DDL_COUNT IS TO BE FOUND.
Y
OUTPUT: GENERATE PL/1 CODE TO TEST (RUN-TIME) IF POINTER TO SOURCE FIELD IS NOT EMPTY
Y
IS CONVERSION SPECIFIED?
N
OUTPUT: GENERATE PL/1 CODE TO CALL DML CONVERSION ROUTINE AND TO MOVE RESULT TO TARGET FIELD
OUTPUT: GENERATE PL/1 CODE TO MOVE SOURCE FIELD TO TARGET.

FIGURE 5-6F
GEN_MOVE

PL/1 code to move constant into Target field. If the TYPE is BIT then GEN_MOVE generates PL/1 code to move BIT constant into Target field. Then after any of the above two GEN_MOVE generates PL/1 code to advance OUTCDP.

If the mapping is of the form " <= 'SOURCE_NAME'", then a test is made to determine if Source Name is a parameter statement, if so, then GEN_MOVE generates PL/1 code to move the value returned by DDL_COUNT or DDL_LENGTH (at run-time) into the Target field. If Source Name is not a parameter statement then it refers to a Source Field, then if the CHECK option was specified then GEN_MOVE generates PL/1 code to test if pointer to Source field is not null.

If a conversion was specified, GEN_MOVE generates PL/1 code to call on DML procedure and to move the result into Target field. Otherwise, GEN_MOVE generates PL/1 code to move the data in the Source field into the Target field.

Finally, GEN_MOVE generates PL/1 code to advance OUTCDP, and returns control to calling program.

5.6.7 COMP_DDL_REF_NAME

This procedure returns the name of the run-time variable which will contain the length of the current field being processed.

Upon entry to this procedure the pointer to the Data table entry of current field is saved. Then FORM_REF and name (see Section 5.6.11) is called, this procedure returns the full reference in PL/1 form. Then using the last name of full reference name its symbol table is found and then its corresponding Data table entry, if the TYPE is not FIELD an error is given and the process terminates.

If the last name in full reference name is a FIELD then, if its TYPE is NUM_PICTURE the internal variable LENT is set to the length of the NUM_PICTURE and the internal variable ATT is set with the string "CHAR". If the FIELD TYPE is CHAR, ATT is set to "CHAR" and if it is BIT, ATT is set to "BIT".

Then a test is made to determine if the length of the reference name is constant or variable (i.e., if it is to be known until execution time). If the length is constant COMP_DDL_REF_NAME generates PL/1 code to declare a prototype field whose attribute is ATT whose length is equal to the length of reference name, then the pointer to the Data Table entry is reset to point to the Target field before the procedure was called. Finally the name of the prototype field generated is passed to calling program.

If the length of the reference Name field is variable, that is, it is not known at compile time, COMP_DDL_REF_NAME generates PL/1 code to declare a prototype field of length 104 whose attribute is ATT, and it is to be overlaid on the reference name. Then the pointer to Data Table entry is reset to the value

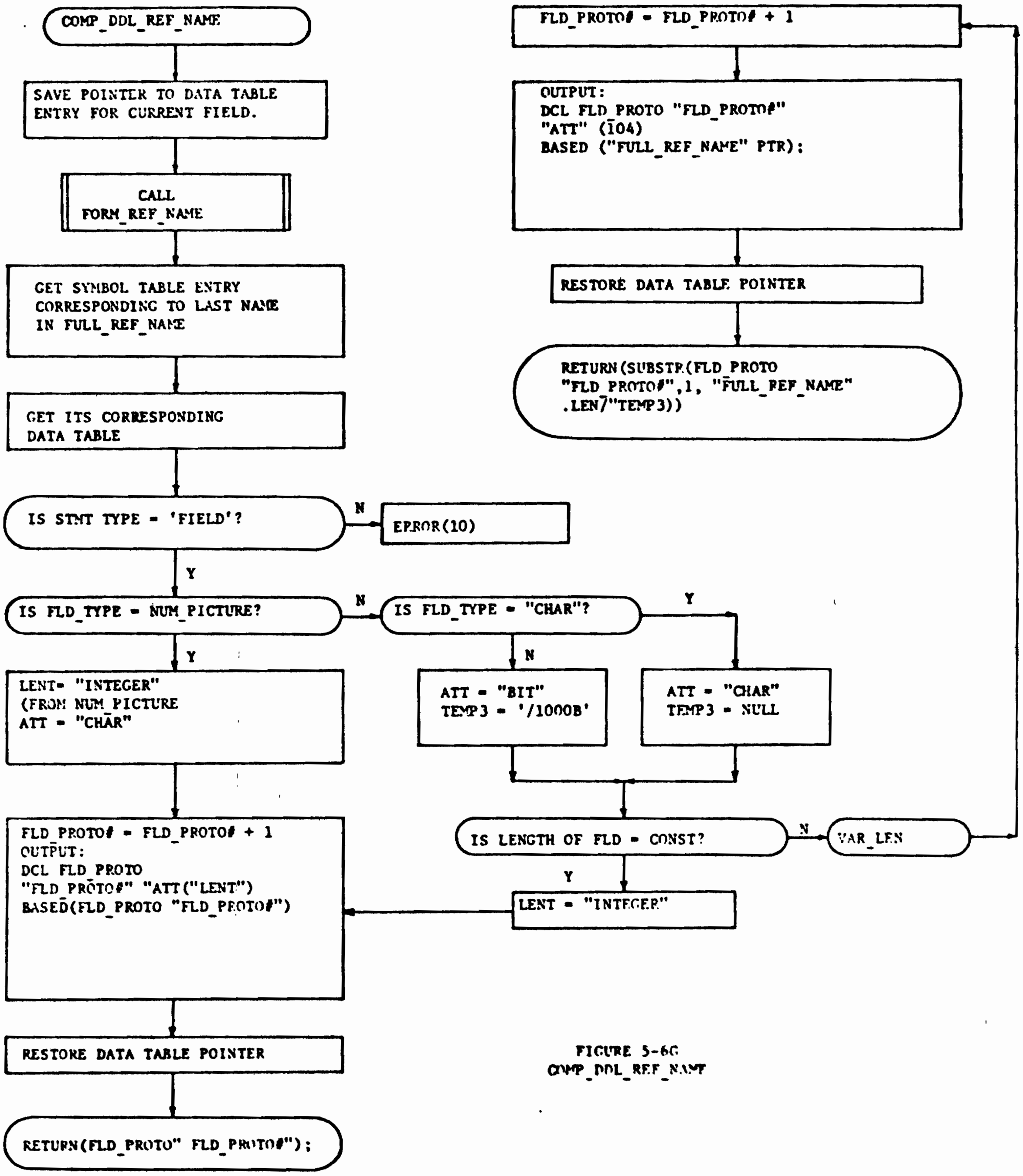

COMP_DDL_REF_NAME
SAVE POINTER TO DATA TABLE ENTRY FOR CURRENT FIELD.
CALL FORM_REF_NAME
GET SYMBOL TABLE ENTRY CORRESPONDING TO LAST NAME IN FULL_REF_NAME
GET ITS CORRESPONDING DATA TABLE
IS STMT TYPE = 'FIELD'?
N
ERROR(10)
Y
IS FLD_TYPE = NUM_PICTURE?
N
IS FLD_TYPE = "CHAR"?
Y
N
Y
LENT= "INTEGER" (FROM NUM_PICTURE
ATT = "CHAR"
ATT = "BIT"
TEMP3 = '/1000B'
ATT = "CHAR"
TEMP3 = NULL
IS LENGTH OF FLD = CONST?
N
VAR_LEN
Y
LENT = "INTEGER"
FLD_PROTO# = FLD_PROTO# + 1
OUTPUT:
DCL FLD_PROTO
"FLD_PROTO#" "ATT("LENT")
BASED(FLD_PROTO "FLD_PROTO#")
RESTORE DATA TABLE POINTER
RETURN(FLD_PROTO" FLD_PROTO#");
FLD_PROTO# = FLD_PROTO# + 1
OUTPUT:
DCL FLD_PROTO "FLD_PROTO#"
"ATT" (104)
BASED ("FULL_REF_NAME" PTR);
RESTORE DATA TABLE POINTER
RETURN(SUBSTR(FLD_PROTO "FLD_PROTO#",1, "FULL_REF_NAME" .LEN7"TEMP3))

FIGURE 5-6C
COMP_DDL_REF_NAME

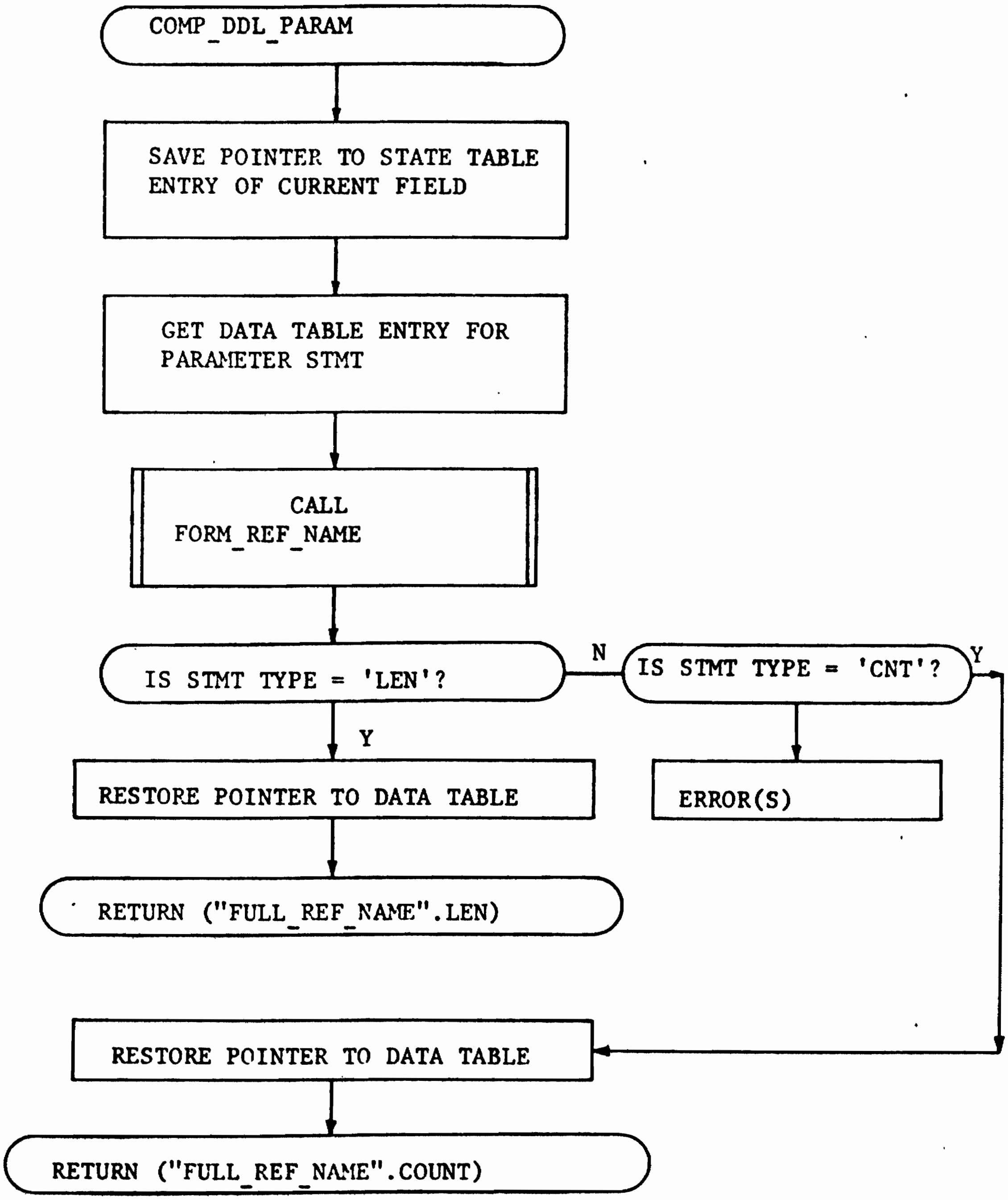

COMP_DDL_PARAM
SAVE POINTER TO STATE TABLE ENTRY OF CURRENT FIELD
GET DATA TABLE ENTRY FOR PARAMETER STMT
CALL FORM_REF_NAME
IS STMT TYPE = 'LEN'?
N
IS STMT TYPE = 'CNT'?
Y
Y
RESTORE POINTER TO DATA TABLE
ERROR(S)
RETURN ("FULL_REF_NAME".LEN)
RESTORE POINTER TO DATA TABLE
RETURN ("FULL_REF_NAME".COUNT)

FIGURE 5-6H
COMP_DDL_PARAM

it had before this procedure was called. Finally the name of the prototype field is passed to calling program.

5.6.8 COMP_DDL_PARAM

This procedure generates the PL/1 name of the run-time variable where the value of DDL_LENGTH or DDL_COUNT is to be found.

Upon entry to this procedure the pointer to the Data Table entry for the current field is saved. The argument of the parameter statement (DDL_COUNT or DDL_LENGTH) is then used to find the corresponding data table entry for that argument name, and COMP_DDL_PARAM calls on FORM_REF_NAME (see Section 5.6.11) to form the PL/1 reference name corresponding to the argument of the parameter statement, and stores it in FULL_REF_NAME.

If the Parameter statement is DDL_LENGTH then the name that COMP_DDL_PARAM returns is FULL_REF_NAME concatenated with ".LEN." And if the Parameter statement is DDL_COUNT COMP_DDL_PARAM returns FULL_REF_NAME concatenated with ".COUNT".

Before returning control to calling program COMP_DDL_PARAM restores the pointer to the Data Table of the Target field it was pointing to before the call on this procedure was made.

5.6.9 GEN_MATCH_END_TAR

This procedure when called from OUT_WALK or CODE_GEN_MOVE generates the PL/1 END statement corresponding to repeating members (if any).

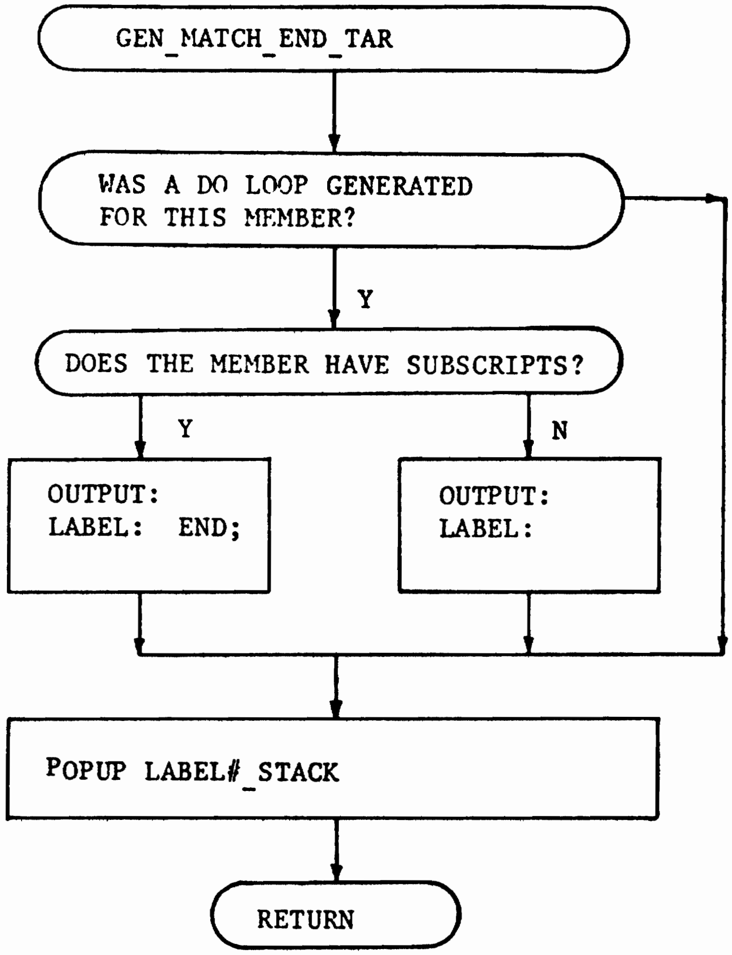


FIGURE 5-61
GEN_MATCH_END_TAR

5.6.10 FORM_SUB

This procedure determines the value of the first and/or second subscript for the i-th name in the Source Name being processed. If the subscript is constant the value is returned in CONST_SUB. If the value of subscript is variable, i.e., if it is to be known until execution time then if subscript is given by a Parameter statement VAR_SUB is set to the name returned by COMP_DDL_PARAM (see Section 5.6.8). And if the subscript is given as a reference name then VAR_SUB is set to the name returned by COMP_DDL_REF_NAME (see Section 5.6.7). After this FORM_SUB returns control to the calling program.

5.6.11 FORM_REF_NAME

This procedure forms the PL/1 reference name corresponding to the argument of the DDL Parameter statement. When this procedure is called it uses as parameter the pointer to a structure where the names that form the argument of the DDL_ PARAMETER statements are stored.

FULL_REF_NAME is formed by concatenating the name (i) and its subscript enclosed in parenthesis (if any) with the previous names already in FULL_REF_NAME which is set to the null string upon entry to this procedure.

5.6.12 GEN_WRITE

This procedure generates PL/1 code to declare internal output buffers and generates PL/1 code to WRITE or LOCATE the Target file into the output buffer. "User Record" refers to

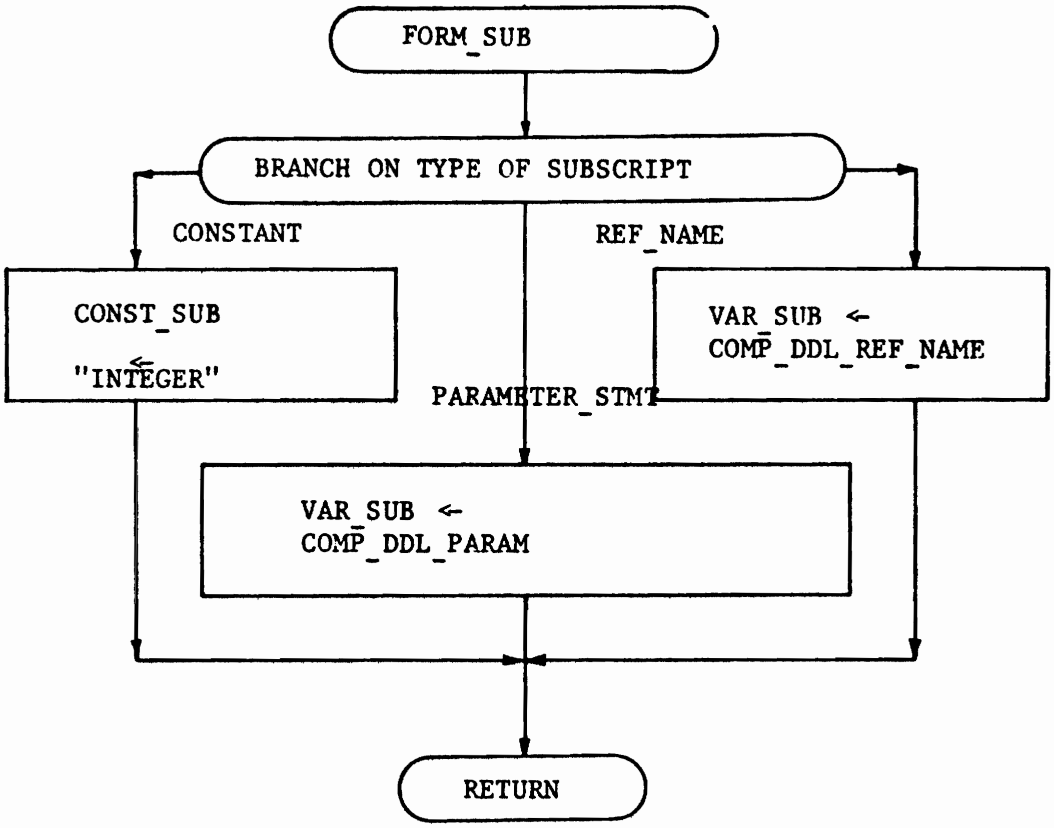


FIGURE 5-6J
FORM_SUB

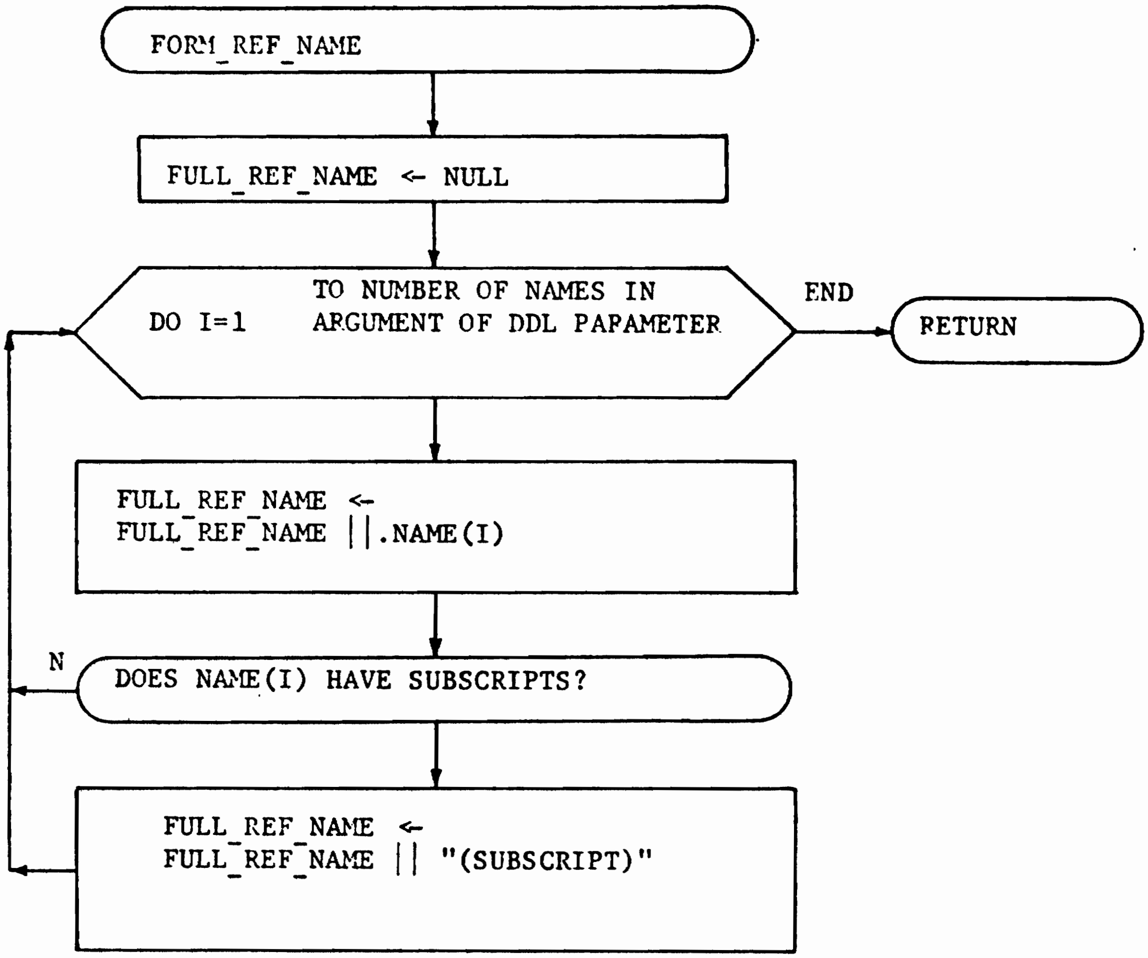

FORM_REF_NAME
FULL_REF_NAME ← NULL
DO I=1 TO NUMBER OF NAMES IN ARGUMENT OF DDL PARAMETER
END
RETURN
FULL_REF_NAME ←
FULL_REF_NAME ||.NAME(I)
N
DOES NAME(I) HAVE SUBSCRIPTS?
FULL_REF_NAME ←
FULL_REF_NAME || "(SUBSCRIPT)"

FIGURE 5-6K
FORM_REF_NAME

the logical target record from the user's point of view. "System Record" refers to the Target record stored by a WRITE or LOCATE statement.

Upon entry to this procedure GEN_WRITE sets STORAGE. RCD_SIZE to BLKSIZE if it is zero. If the STORAGE.RCD_FORMAT is variable or variable spanned (v or vs) then eigth is substracted from the record size to ignore the block count and record count. If the STORAGE.RCD_FORMAT is variable blocked or variable blocked spanned (VB or VBS) when substract four from the record size to ignore the record count.

If USER_RCD_FLAG is equal to zero i.e., length of system record is equal to length of user record, then the information is taken from the STORAGE statement for the Target file, then the PL/1 statements to declare OUTBUF takes place, (OUTBUF is declared as a character string whose size is given by the STORAGE stmt and it is used as an output buffer for DDLTAR file). Next GEN_WRITE generates PL/1 code to LOCATE the Target record to the Target file (DDLTAR), also GEN_WRITE generates PL/1 code to pad the last block of a fixed-blocked file if the last block on the file is not filled and if the user so requested via the PAD keyword.

If USER_RCD_FLAG is equal to one i.e., fixed number of bytes per user output record. Then if the RCD_FORMAT_TYPE is less than two i.e., the system record is also fixed size, a check is made to see if user output record size is a multiple of storage output record size, call ERROR(16) if not. If the user record size is equal to storage record size GEN_WRITE

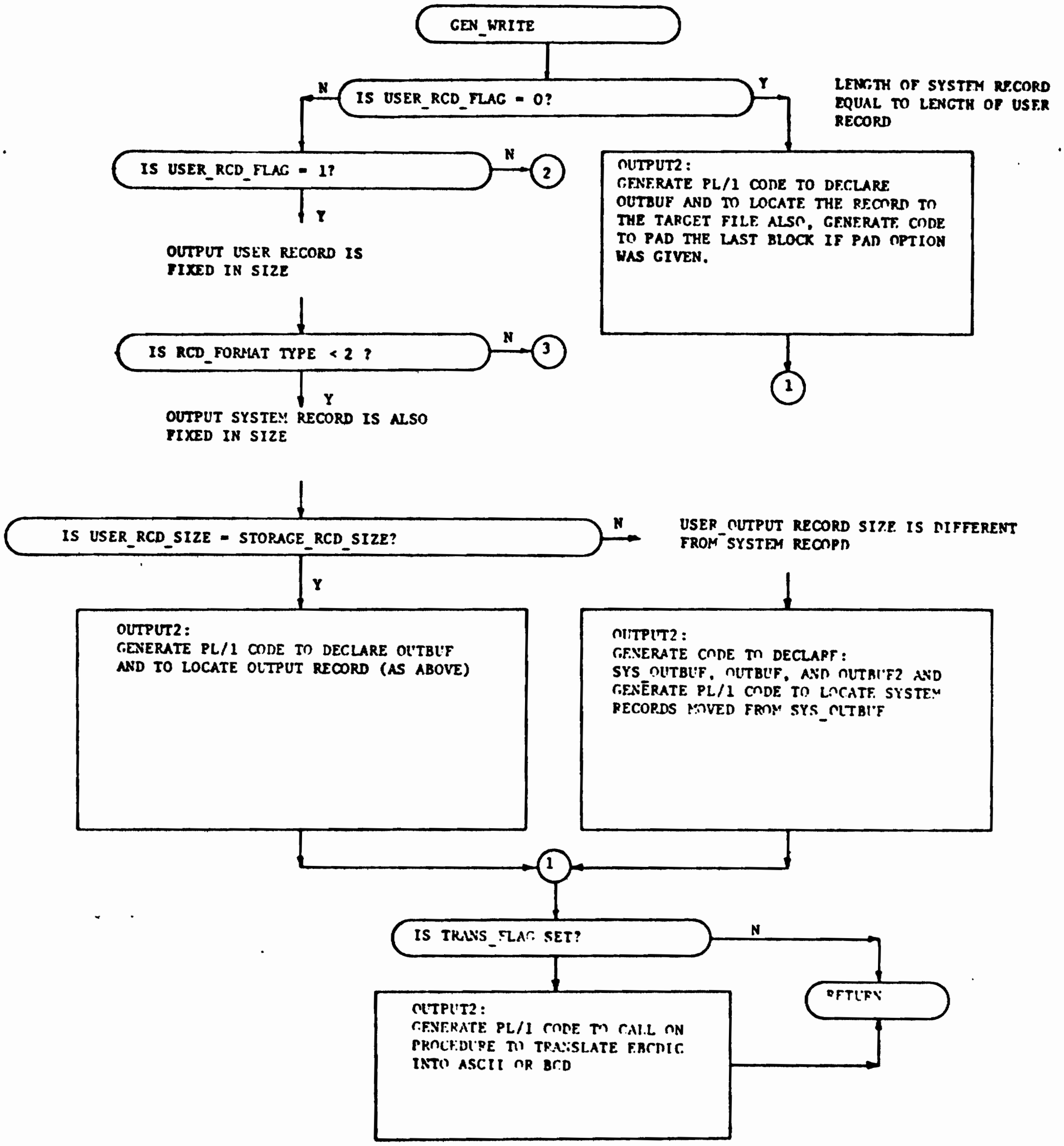

GEN_WRITE
IS USER_RCD_FLAG = 0?
N
Y
LENGTH OF SYSTEM RECORD EQUAL TO LENGTH OF USER RECORD
IS USER_RCD_FLAG = 1?
N
2
Y
OUTPUT2:
GENERATE PL/1 CODE TO DECLARE OUTBUF AND TO LOCATE THE RECORD TO THE TARGET FILE ALSO, GENERATE CODE TO PAD THE LAST BLOCK IF PAD OPTION WAS GIVEN.
1
OUTPUT USER RECORD IS FIXED IN SIZE
IS RCD_FORMAT TYPE < 2 ?
N
3
Y
OUTPUT SYSTEM RECORD IS ALSO FIXED IN SIZE
IS USER_RCD_SIZE = STORAGE_RCD_SIZE?
N
USER_OUTPUT RECORD SIZE IS DIFFERENT FROM SYSTEM RECORD
Y
OUTPUT2:
GENERATE PL/1 CODE TO DECLARE OUTBUF AND TO LOCATE OUTPUT RECORD (AS ABOVE)
OUTPUT2:
GENERATE CODE TO DECLARE:
SYS_OUTBUF, OUTBUF, AND OUTBUF2 AND GENERATE PL/1 CODE TO LOCATE SYSTEM RECORDS MOVED FROM SYS_OUTBUF
1
IS TRANS_FLAG SET?
N
RETURN
OUTPUT2:
GENERATE PL/1 CODE TO CALL ON PROCEDURE TO TRANSLATE EBCDIC INTO ASCII OR BCD

FIGURE 5-6L
GEN_WRITE

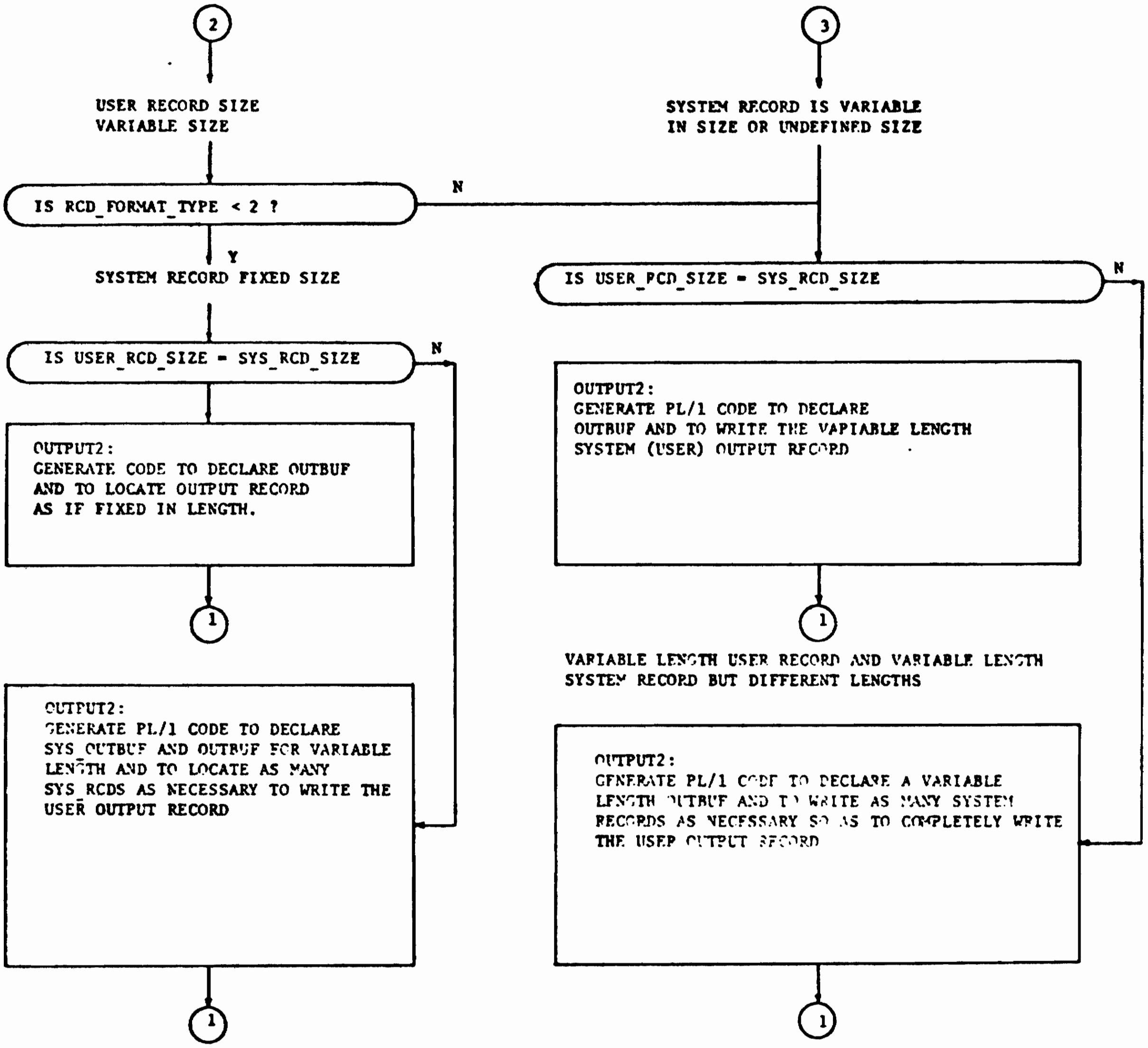

2
USER RECORD SIZE
VARIABLE SIZE
3
SYSTEM RECORD IS VARIABLE
IN SIZE OR UNDEFINED SIZE
IS RCD_FORMAT_TYPE < 2 ?
N
Y
SYSTEM RECORD FIXED SIZE
IS USER_RCD_SIZE = SYS_RCD_SIZE
N
IS USER_RCD_SIZE = SYS_RCD_SIZE
N
OUTPUT2:
GENERATE CODE TO DECLARE OUTBUF
AND TO LOCATE OUTPUT RECORD
AS IF FIXED IN LENGTH.
OUTPUT2:
GENERATE PL/1 CODE TO DECLARE
OUTBUF AND TO WRITE THE VARIABLE LENGTH
SYSTEM (USER) OUTPUT RECORD
1
1
VARIABLE LENGTH USER RECORD AND VARIABLE LENGTH
SYSTEM RECORD BUT DIFFERENT LENGTHS
OUTPUT2:
GENERATE PL/1 CODE TO DECLARE
SYS_OUTBUF AND OUTBUF FOR VARIABLE
LENGTH AND TO LOCATE AS MANY
SYS_RCDS AS NECESSARY TO WRITE THE
USER OUTPUT RECORD
OUTPUT2:
GENERATE PL/1 CODE TO DECLARE A VARIABLE
LENGTH OUTBUF AND TO WRITE AS MANY SYSTEM
RECORDS AS NECESSARY SO AS TO COMPLETELY WRITE
THE USER OUTPUT RECORD
1
1

FIGURE 5-6L (continued)
GEN_WRITE

generates PL/1 code to declare OUTBUF and to LOCATE the Target record into the output buffer. Finally GEN_WRITE returns control to the calling program.

If user output record size is different from system output record then GEN_WRITE generates PL/1 code to declare: SYS_OUTBUF - output buffer to hold system output record, OUTBUF - output buffer to hold user output buffer, and OUTBUF2 - an array of character strings (each has the length of SYS_OUTBUF and they are overlaid on OUTBUF). Next, GEN_WRITE generates PL/1 code to LOCATE system records moved from SYS_OUTBUF. Then GEN_WRITE returns control to calling program.

If USER_RCD_FLAG is equal to two, i.e., there are a variable number of bytes per user output record. Then if STORAGE.RCD_FORMAT_TYPE is less than two (i.e., fixed length output system records but a variable number of them form a user output record. Next a check is made to determine if the user output record size is equal to the storage output record size, if so, then GEN_WRITE generates PL/1 code to declare OUTBUF and generates PL/1 code to LOCATE output records as if fixed length. Finally GEN_WRITE returns control to calling program. If user output record size is different from system output record size GEN_WRITE generates PL/1 code to declare SYS_OUTBUF - buffer for system record, OUTBUF - buffer for variable-length user record, and also generates PL/1 code to LOCATE as many system output records as necessary to write the user output record. Finally GEN_WRITE returns control to callilg program.

If the USER_RCD_FLAG is equal to two and STORAGE.RCD_ FORMAT_TYPE is not less than two i.e., variable or undefined systems output records and variable length user output records. And if user output record size is equal to storage output record size i.e., system variable length record size is same as length user output record, then GEN_WRITE will generate PL/1 code to declare OUTBUF and appropriate PL/1 code to WRITE the variable-length system (and user) output record. Finally GEN_WRITE returns control to calling program. If user outp t record is different in length from storage output record i.e., variable user output record and variable output system record of different size each. Then GEN_WRITE generates PL/1 code to declare a variable length OUTBUF (buffer for user output record) and also generates PL/1 code to WRITE as many systems output records as necessary so as to completely WRITE the user output record. Finally GEN_WRITE returns control to calling program.

5.6.13 OUT_PROC

This procedure when called output to the corresponding OUT file (OUT → OUT7) the record given as the argument in the

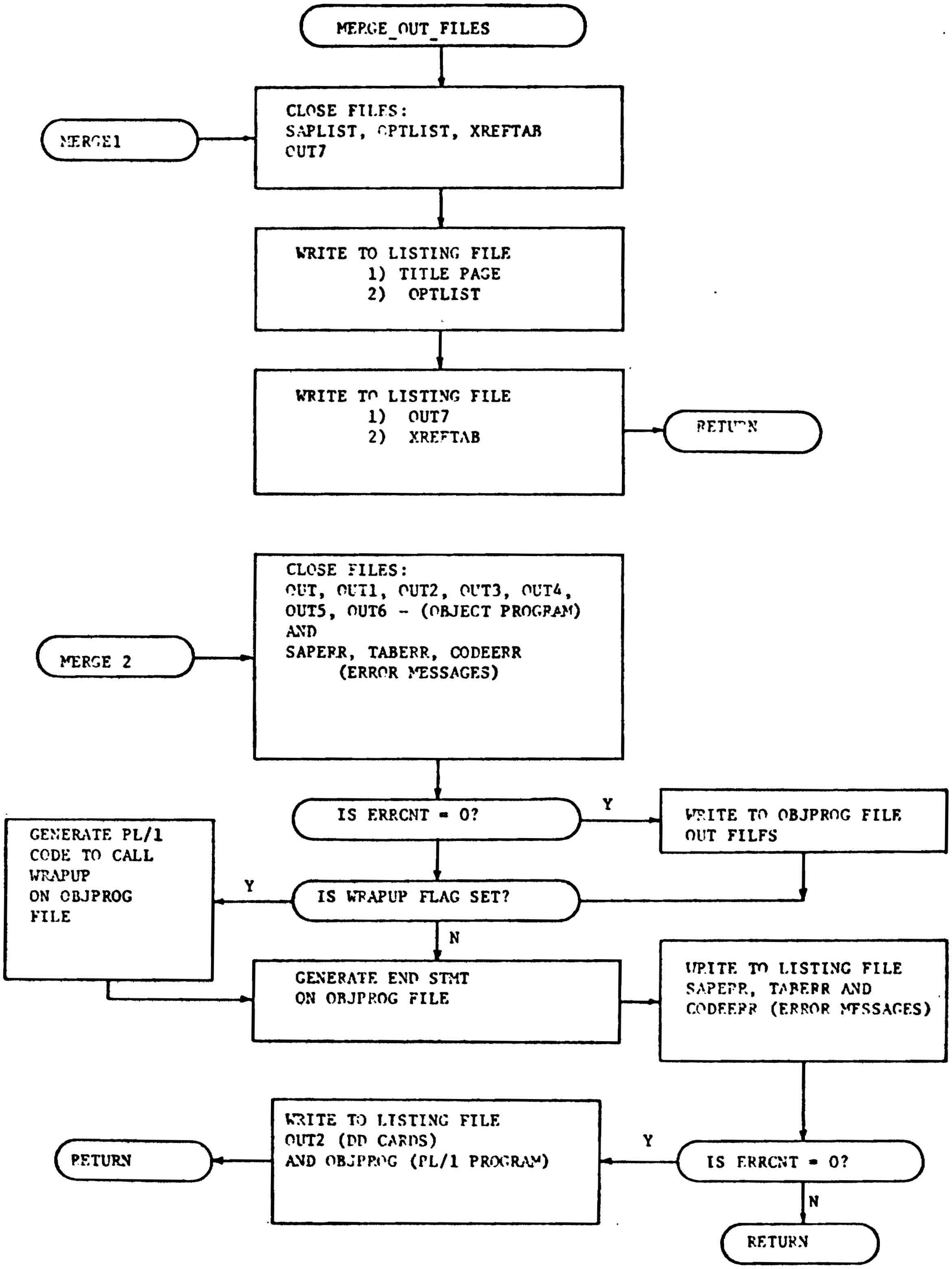


FIGURE 5-6M
MERGE_OUT_FILES

call.

If the input parameter record is larger than 80 characters OUT_PROC breaks the record in appropriate places in such a way that the record output is never larger than 72 characters, and it outputs as many records of such length as necessary.

5.6.14 MERGE_OUT_FILES

This procedure has 2 entry points; MERGE1, and MERGE2.

Upon entry to MERGE4 the output files SAPLIST, OPTLIST, XREFTAB and OUT7 are closed. Then the title page and the OPTLIST file are written into the LISTING file. Next OUT7 and XREFTAB are also written into the LISTING file.

When MERGE2 is called it closes all the output files. Then if the ERRCNT is equal to zero the OUT files are written into the OBJPROG file. If the WRAPUP option was specified to the DDL compiler then a call to the WRAPUP routine is generated and written into the OBJPROG file.

Next the END statement is generated and also written into the OBJPROG file. Then SAPERR, TABERR and CODEERR (error message files) are written into the LISTING file.

If the ERRCNT is equal to zero then the OUT2 (DD_CARDS) and OBJPROG files are written into the LISTING file.

When the compilation process terminates the LISTING file is printed.

CHAPTER 6

CONCLUSIONS & RECOMMENDATIONS

6.1 Background and Need For The DDL/DML Language and Processor

The need for an efficient method of converting data organization in view of new user needs or for use with different programs or different computers has long been recognized by the community of EDP users. Presently, a user can re-organize data by either writing his own special software or by using the data description facilities contained in the programming languages, operating systems and data management systems available for a particular computer. Converting file structures, however, can prove to be a very tedious task for programmers even for small organizational or format changes to the file. Furthermore, data organized on one computer system often cannot be directly used on a different computer installation due to incompatibilities of software and hardware. In many cases, the organization of this data cannot be communicated effectively to another user because the data organizations are implicit in the programs or software used. In all these cases, the only way to re-organize data has been to write a special conversion program, which can require considerable effort.

The DDL/DML processor described in this dissertation is a first step towards the solution of the general data conversion problem. While much work has been done towards the development of Data Description languages by such groups as the Codasyl Date Base Task

Group. Their basic approach was to enhance existing programming languages such as COBOL, FORTRAN, PL/1, etc. with "DDL/DML" statements and to modify the existing compilers to accept their DDL/DML descriptions. Our aims, on the other hand, has been to design and implement an utility capable of creating an ad-hoc program to perform a specific conversion based on a user-written DDL/DML language independent of existing compilers.

In order to develop this general-purpose conversion utility, our primary emphasis has furthermore been to use as far advanced state-of-the art techniques as possible in the compiler building process.

As Prywes/Smith have said [PRE 72].

> "Simply speaking, a DDL is a language which enables a person to describe every aspect of a data organization, from the interrelation among elements of the organization to its representation as a linear string and its positioning on a specific storage medium. Such descriptions can serve as a basis for organizing or converting the respective data bases automatically."

Such a DDL has been designed by Diane P. Smith [SM 71] but its primary deficiency was its difficulty of use and implementation. We later modified it and simplified it for ease of use and ease of implementation. The importance of developing such a DDL/DML independent of any Data Management System is easily seen in terms of its applications:

1) to communicate data organizations between humans,

2) to communicate the organization of data to computers to permit processing, and

3) to convert data from one organization to another.

The DDL processor described in this dissertation is a practical approach for using DDL/DML for data conversion. The DDL processor is actually three processors. The first is the Syntactic Analysis Program Generator (SAPG), the second is the DDL Compiler, and the third is the Data Conversion Processor.

6.2 Techniques Used To Design and Implement The DDL Processor

6.2.1 Syntax Specification and Analysis

Even though Backus-Naur Form (BNF) was introduced as a mechanism to generate the sets of strings occuring in natural and programming languages, we envision it as a tool for the following tasks: (a) as an aid in specifying the language, i.e., a tool for compiler writers; (b) as an aid to implement syntactic analysis program generators; i.e., for the purpose of recognizing whether or not a given string belongs to the language; (c) as an aid to the user in learning how to use the language correctly.

Extended Backus-Naur Form (EBNF) enhances BNF in a way which makes communication of syntax descriptions easier by providing a simple way to describe optionality and repetition. It also simplifies syntax checking, since the scan can be accomplished in a strictly left-to-right manner.

The Syntactic Analysis Program Generator, along with EBNF, is a useful tool for writing compilers and language study. Furthermore, it would be a valuable "stand alone" tool in writing any syntax analysis program. The SAPG produces a syntax analysis program which performs syntax checking. (The syntax is checked only locally and not globally; i.e., the syntax for each source statement is checked by automatically generated code, but relationships between statements in the program - must be checked by the compiler writer). The EBNF with subroutine calls (EBNF/WSC) enhances EBNF such a way to facilitate encoding of the source statements in internal tables. SAPG provides an overriding advantage over hand-coded SAP's in that it easily allows changes to the syntax of a language. In the case of DDL, a completely successful change from the DDL designed by D. Smith [SMI 71] to the DDL/DML version 1.0 presented in this dissertation - was accomplished in less than a week. Since the compiler writer needed only to describe the structure of the new language in EBNF/WSC and have the details of syntax analyses to the SAPG.

Local syntax checking is done in a single pass, this gives a "compilation advantage" to syntactically error-free programs, since internal table creation for a statement is performed until an error is detected. However, compilation of programs which contain errors will still invalue table creation overhead for all but unrecognizable statements.

Even though SAPG does not handle left recursion and multiple token look-ahead, these restrictions could be circumvented. Of course, this would probably necessitate changes in the code that SAPG generates and most likely the logic of SAPG.

The ideas of Floyd (FLO 69], Presser [PRE 69] and Conway [CON 63] (see Chapter 4) in regard to lexical analysis proved to be of great help in the design and implementation of the lexical analyzer for EBNF/WSC and for the lexical analyzer for DDL. The design of both lexical analyzers, however, could be improved in such a way as to allow the compiler writer to simply input the character-mapping and state-transition tables without rewriting the whole lexical routine. A general set of functions should be provided from which the compiler writer may chose those which he needs. The character-mapping and state-transition tables could be input at the start of compilation. In this way changes could very easily be made by simply inputing a new set of tables.

With the help of EBNF/WSC the encoding of the source statements is made possible. In our experience it proved to be beneficial in several areas. Encoding the input reduces the amount of work performed by code generation, permits the use of global syntax checking routines, and makes the building of the cross-reference table much easier. The separation of local and global syntax checking separate from code generation permits modifications to the compiler to be performed modularly, simplifying matters considerably. The internal tables have been designed to facilitate global syntax checking, and data preservation for code generation. To this end PL/1 structures are created to contain the encoded statements. Thus syntax checking routines, cross-reference routine and 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, therefore, seeing this qualified name in the code is enough of a clue to identify which structure is currently being dealt with.

Storage optimization in the encoding of the input statements was also considered. For this reason in the Data table formats (see Appendix D) a pointer to the DDL name rather than the name itself is kept, in this way only one word is used rather than 32 characters. Thus a substantial saving of space may be realized if DDL names are frequently referenced.

In many instances, Data table entries do not have a fixed structure. This means that they are allocated only after it has been determined just how much information is to be stored in them (this is done to achieve storage optimization). It is apparent that collection of this data must occur by way of temporary storage, these temporaries were chosen to be PL/1 controlled structures so that, after all information has been amassed, their storage allocation would be freed.

By designing the Symbol Table and Data Tables as doubly chained lists, the code necessary for walking through the structures was immensely simplified.

6.2.2 The Use of PL/I As A System Programming Language

For the selection of a language to implement the SAPG, the syntactic supporting routines, the cross reference routines and the code generation routines our views were almost identical to those of Charles A. Lang [LAN 70] and F.J. Corbato [COR 69] in regard to the selection of a Programming language to write systems programs.

The two main categories of programming language candidates were assembly language and higher level programming languages. The following criteria were taken into account when the decision was made to write the system in a higher level language:

1) Ease of Programming

The language should enable the programmer to state clearly what he wants rather than how he wants to do it; further, the program must be clear when read by himself and others. A high level language such as FORTRAN, ALGOL PL/1 meets these requirements better than Assembly Language.

2) Maintainability

Secondly there is the problem of technical management of programming projects: the problem of trying to maintain a system in the face of personnel turnover and in the face of varying standards of documentation. Personnel turnover is to be expected in the University environment. Students get their degrees and leave, carrying with them key know-how. Training of new personnel includes both learning a new programming language and learning about the system. It is well known that, it takes at least twice as long to learn assembly language, and to become familiar with system routines written in assembly language, as it takes to learn a high level language.

3) Efficiency of Execution

Finally, while code written in assembly language tends to be slightly more efficient than compiler-generated code, we felt that this was a small price to pay for the other two factors. Furthermore, there is more and more a trend toward good optimization of compiler-generated code.

The essential question was then, what high level language to use. We chose PL/1 for the following reasons:

1) Modularity

One can write a system in PL/1 modularly; one can compile each subsystem of the final processor separately, clean up the syntax, debug it and test it on an individual basis. (This also is true in Fortran, Algol and Assembly language).

2) Special Features

PL/1 is very good in providing data structures and data types which we considered to be very important features for writing the DDL compiler. These include controlled storage, based storage and pointer variables, dynamic storage allocation, string manipulation and capabilities, powerful built-in functions, block structures, recursive procedures, variable-length hierarchical structures, and varying strings. In other words, PL/1 is very powerful in the special language facilities which we needed.

3) Ease of Programming

Being a university research group we obviously could not compete with a software house in developing the most efficiently running system.

PL/1 allowed us to concentrate on the development of the system itself rather than worrying about a specific machine configuration and hardware. Thus, by viewing our machine as a "PL/1 machine," we were able to focus our attention on developing a reasonably efficient system with little manpower and time.

Two computer systems available at the University of Pennsylvania that could satisfy requirements were evaluated, the RCA Spectra 70/46 Time Sharing System, and the IBM 370/165 operating under OS/360. The RCA system includes support for ALGOL, COBOL, FORTRAN, SNOBOL, and Assembler languages. The IBM system supports these and also PL/1, APL and LISP. Since both systems meet the hardware and operating system requirements, the programming language was the determinant. The IBM/370 computer system was selected because of PL/1 availability.

Overall, the general result that we got from using PL/1 was a relatively small number of programming errors and rapid debugging and checkout. One of our very few major sources of residual trouble is of bugs caused by mismatched declaration in the structures. In fact, it is to be expected, since it is a defect in the PL/1 language in the sense that the independence of the separate compilation has left a gap in the checking of the structures and types.

A major positive effect of the use of PL/1 has been that we have been able to make major strategic changes without too much reprogramming.

6.2.3 PL/1 As An Intermediate Object Language

The technique of transforming DDL into the object program, IBM/370 machine code, by means of a sequence of simple transformations leads naturally to the idea of an "intermediate language." The need for an intermediate language which could act as a Universal Computer Oriented Language (UNCOL) has been proposed (Strong et.al. [STO 58], Steel [STE 61]) but the design of such a UNCOL has not been done. We decided to use PL/1 as an intermediate language in the translation of DDL to IBM/370 machine code, since we could more easily generate PL/1 code than machine code and thus take advantage of the existence of the PL/1 compiler. While one might argue that compilation time is greatly increased, we claim that we were able to achieve our goal of implementing a prototype DDL compiler much more quickly and producing a much more maintainable and expandable system than one which generated machine code directly. Another consideration and further benefit in deciding to use PL/1 as the object language was that the DML language being a subset of PL/1 allowed us to merge the PL/1 output of the DDL compiler with the DML routines supplied by the user and then to feed the result into the PL/1 compiler to obtain IBM/370 machine code. PL/1 as an intermediate language was also used by SAPG in producing the SAP for the DDL compiler for analogous reasons.

The code generation routines and the routine to build the cross-reference table were written using Ad-Hoc techniques due to the fact that there is no methodology available at this time to describe the semantics of a language in a Meta language as was possible to do with the syntax.

6.3 Experience With DDL/DML

The present version 1.0 of the DDL/DML compiler have been in operation at the University of Pennsylvania since September of 1972. The computer system in which we run the DDL/DML compiler is the IBM/370 model 165.

A User Guide for DDL/DML (see Appendix A) has been produced, in it we present how to use DDL and DML, and how to call the DDL/DML compiler under the OS of the IBM/370 model 165.

Three real life examples have been successfully run for the Naval Research Laboratory.

Example 1: We were given a source file, known as the internal data base (internal DB), stored in a Magnetic tape (7 trk), and coded in upper/lower case ASCII (8 bit code, where the highest order bit is always zero), odd parity. The output will be a file known as DSAM stored on magnetic tape (7 trk), and coded in BCD, even parity with block size of 720 characters except for header and trailer lables of 80 characters with end of file as specified in DSAM manual # 4185.5. The conversion process involves tests in security fields and validation of fields with the corresponding error messages when some security or validation was violated. The security and validation criteria are stated in a booklet produced by the Defense Supply Agency (December 1968).

To compile the DDL/DML program for Example 1 into PL/1 took 29 seconds, and from PL/1 into machine code (IBM/370 model 165) the compilation time was 32 seconds. The Conversion Program produced by the DDL compiler was tested with two different input tapes one with 10 records and the other with 11 records. The conversion process took in both cases 14 seconds.

Example 2: With the same input as in Example 1, the output is to be stored in a magnetic tape (7 trk) coded in "reverse" ASCII , odd parity, with variable record size, such tape will be used as input to one of the Navy computers to produce a "hard copy."

To compile the DDL/DML program for Example 2 into PL/1 took 15 seconds, and from PL/1 into machine code the compilation time was 20 seconds. The conversion program was tested with the same two tapes used in Example 1 in both tapes the conversion process took 2 seconds.

Example 3: In this case the input tape was already validated. The output was the same as in Example 1. The difference is that the conversion process is carried out without validation. The compilation times from DDL into PL/1 and from PL/1 to machine code were 20 and 22 seconds respectively. The conversion process took 2 seconds.

6.4 Future Trends and Developments

The present version of the DDL compiler allows the user to handle any type of sequential file to perform conversions, report generation, and if he provides DML routines any kind of validation.

It would be desirable for the DDL language and processor to be extended in such a way so that it processes more than one input file. Furthermore, other kinds of file organizations need to be permitted, i.e., random, index-sequential, etc.. Moreover, the validation of input data could be greatly improved by adding new statements to DDL where the validation criteria could be described, thus replacing the need for DML in validation.

It would also be desirable to extend DDL to encompass updating of existing Data bases. Thus, we envision that the DDL language could be a base for a versatile Data Base Management System.

In the area of report generation the present version 1.0 of the DDL/DML language does a good job, but it is the user who needs to worry about formats, headings, etc. Therefore, some work must be done to extend the DDL compiler to provide the user a better way to produce reports.

In order to achieve the above improvements it is necessary to change both the syntax and the semantics of DDL/DML. To change the syntax we have proved that it can be easily and more automatically done by using the SAPG, but it is in the area of semantics -Code Generation- where the problem can be more difficult. In order to facilitate the expansion of such a system, a more automated mechanism to implement code generation is needed, and we would therefore like to propose that the following research be done on the area of compiler generators.

In a paper presented at a Working Conference on Mechanical Language Structures, "On Context and Ambiguity" in Parsing [ROS 64] a very interesting general discussion in the area of semantics is reported, and it is highly important to our discussion on semantics here and I would like to quote some of the remarks reported in that discussion.

Iroms:

"In order to describe a language - any language - you have to have another language to use as some instrument by means of which to convey information. The only thing is, that you must have a language to use to describe the other language - the one you are trying to describe, and it should be simply enough so that it is easily understood by people who look at it."

Gorn: "However, I think that if you want a language to define the meaning of something it has to define it in terms of something else which already has meaning. So you have to have semantics to get semantics. In every case this means some machine in the background because that is where something happens which means something."

Bauer: "We never can get rid of this, and it means that we must really come to some level that we can easily agree upon."

Perlis: "We are interested in semantics so that we can mechanize the process of translation on computers."

Brooker: "In the case of ALGOL, the most useful concepts are scope of an identifier, block structure, substitution of expressions for names in the body of the text, and all the usual arithmetical concepts. The arithmetical parts are fairly easily understood by most people; it is the other concepts that the non-programmer is unfamiliar with. I personally cannot see how else these can be explained except by high quality English prose."

Backus: "I think one purpose that one could have in describing meaningful mechanisms, for describing the meaning of programs in arbitrary new languages, is so that people can publish a description of a newly proposed language and have it made clear to the readers in a fairly transparent way what interpretation he wishes to place on the statements of this language."

The above remarks apply to someone who wants to design a Meta language to describe the semantics of languages existing or new languages. I think that because of all the difficulties in this respect no one has been able to produce such a language. To the best of our knowledge the only Meta language to describe semantics available today is the Vienna Definition Language [WEG 72] and this language is a Meta language for defining interpreters rather than compilers.

The approach that we propose is one based in the remarks given above but specifically for the DDL compiler. That is, DDL statements are to be translated into an intermediate language (PL/1) and the Meta language will be based on a set of MACROS ad-hoc to DDL to produce code in PL/1. In this way we will use English prose to describe the semantics of the MACROS and then the compiler writer can call on them from the Meta language for describing DDL to generate the desired PL/1 code.

With this approach it is much easier to extend the DDL compiler in the areas we have mentioned since the compiler writer will use EBNF/WSC to describe the syntax of the new DDL and the proposed

Meta language for DDL to describe (by calling in the MACROS) the semantics of the new DDL.

In effect, we are proposing then, a compiler generator for DDL - type languages. As we see in Figure 6.1, such a compiler generator would consist of the SAPG which will produce the syntax analysis program and a Code Generation Program Generator (CGPG) which would assemble code generation routines. Such a system would greatly provide a useful vehicle for extending the current system.

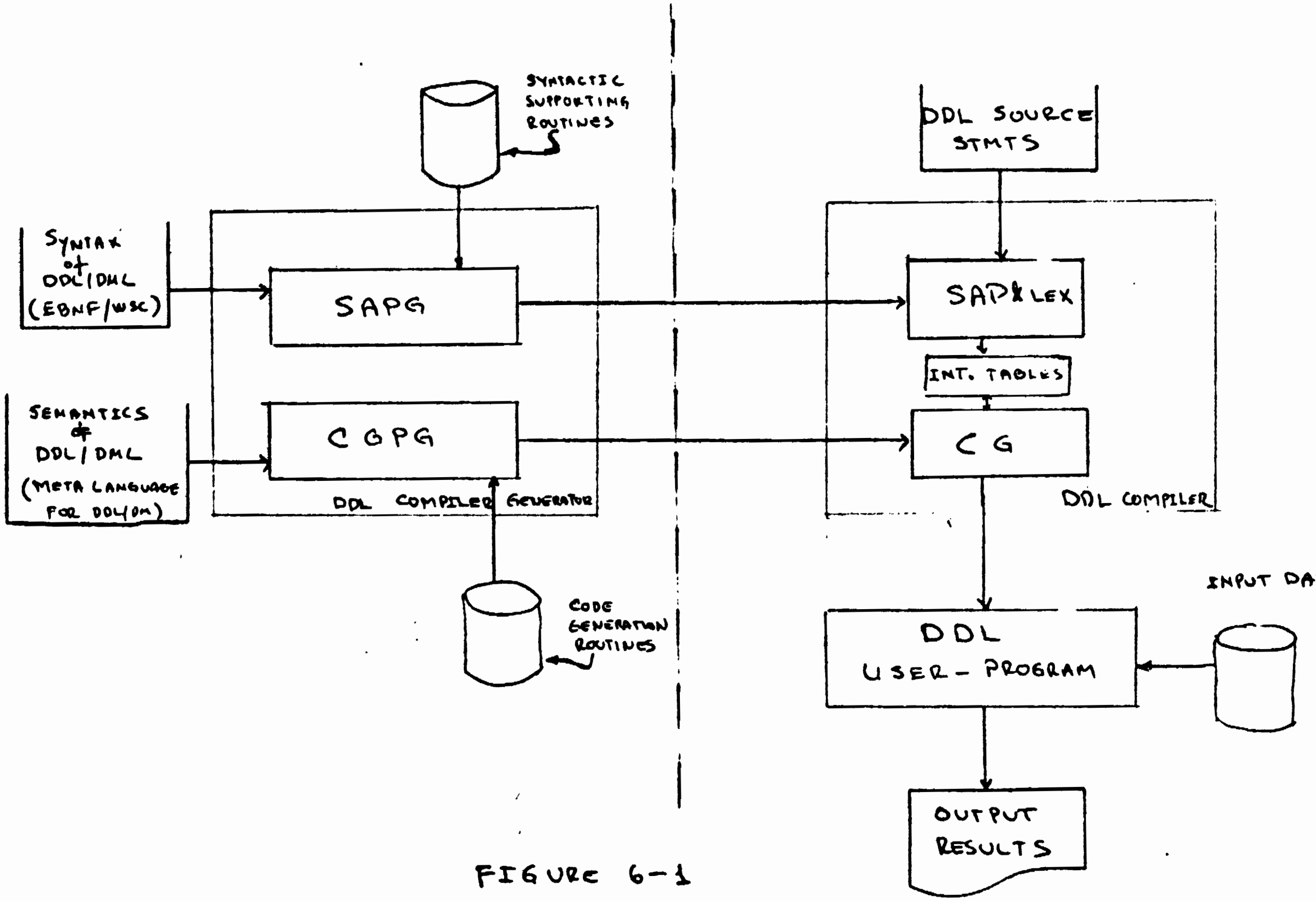


FIGURE 6-1

DDL COMPILER - GENERATOR