December 1971

Design of the Data Description Language Processor

Andrew French
University of Pennsylvania

Jesus A. Ramirez
University of Pennsylvania

Harold Solow
University of Pennsylvania

Noah S. Prywes
University of Pennsylvania, nsp@seas.upenn.edu

Recommended Citation

Andrew French, Jesus A. Ramirez, Harold Solow, and Noah S. Prywes, "Design of the Data Description Language Processor", . December 1971.

University of Pennsylvania Department of Computer and Information Science Technical Report No. MS-CIS-72-19.

Design of the Data Description Language Processor

Abstract

The Data Description Language (DDL) is a language for describing the structure of data, and expressing transformations that are to be performed on that data. The DDL Processor is a set of computer programs which interprets DDL statements and generates a computer program to perform the specified transformations. Together the DDL and its Processor provide a utility which can be used to perform jobs such as creating new data bases, reorganizing or extracting data from existing data bases, moving data to different storage devices, interfacing files between different programming languages, or between different operating systems.

This report documents the design of the DDL Processor. Special features of the design include the use of special purpose internal languages, compiler-compiler techniques, bootstrapping methods, and a descriptor tree which aids in the parsing of input data.

Comments

University of Pennsylvania Department of Computer and Information Science Technical Report No. MS-CIS-72-19.

University of Pennsylvania
THE MOORE SCHOOL OF ELECTRICAL ENGINEERING

ANNUAL REPORT

DESIGN OF THE DATA DESCRIPTION LANGUAGE PROCESSOR

by
A. French
J. Ramirez
H. Solow
N. S. Prywes

Project Supervisor
Noah S. Prywes

December 1971

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

under

Contract N00014-67-A-0216-0007
Project No. 049-272

DISTRIBUTION STATEMENT

Moore School Report No. 72-19

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, Pa. 19104

2a. REPORT SECURITY CLASSIFICATION

UNCLASSIFIED

2b. GROUP

3. REPORT TITLE

DESIGN OF THE DATA DESCRIPTION LANGUAGE PROCESSOR

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

Annual Report

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

Andrew French, Jesus A. Ramirez, Harold Solow

6. REPORT DATE

December, 1971

7a. TOTAL NO. OF PAGES

211

7b. NO. OF REFS

9

8a. CONTRACT OR GRANT NO.

N00014-67-A-0216-0007

b. PROJECT NO.

NR 049-272

c.

d.

9a. ORIGINATOR'S REPORT NUMBER(S)

Moore School Report No. 72-19

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 Data Description Language (DDL) is a language for describing the structure of data, and expressing transformations that are to be performed on that data. The DDL Processor is a set of computer programs which interprets DDL statements and generates a computer program to perform the specified transformations. Together the DDL and its Processor provide a utility which can be used to perform jobs such as creating new data bases, reorganizing or extracting data from existing data bases, moving data to different storage devices, interfacing files between different programming languages, or between different operating systems.

This report documents the design of the DDL Processor. Special features of the design include the use of special purpose internal languages, compiler-compiler techniques, bootstrapping methods, and a descriptor tree which aids in the parsing of input data.

Security Classification

TABLE OF CONTENTS

TABLE OF CONTENTS (continued)

TABLE OF CONTENTS (continued)

LIST OF FIGURES

1. INTRODUCTION

1.1 Background of Project

This report documents the design of a Data Description Language (DDL) Processor. It has been prepared with support by the Office of Naval Research, Information Systems Program, under Contract N00014-67-A-0216-0007. This report concludes Phase 1 of a two phase project. Phase 2 of this project will consist of the implementation of the DDL Processor in accordance with the design in this report.

The need for an efficient method of converting data for use with different programs or different computers has been long recognized by the Navy as well as the larger community of EDP users. Presently, a user can 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. Data organized in this way 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 some cases, the only way to interchange data is to write a special conversion program. This can require considerable effort, due to such problems as different word and character organization and machine word sizes.

There are two main approaches to the solution of the Data 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 where it has been incorporated. The other approach which is considered feasible is to develop a new "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. Our design is based on this second approach.

The Data Definition Language Processor reported here is a separate and distinct development from the Data Management System. While it can communicate and prepare information for the latter, it performs such functions as a separate processor. The existence of a Data Description Language Processor in respective computer systems will facilitate communications and exchange of data between computer systems and computer programs, and across computer language barriers.

The first step in the development of DDL was to consider how it would be used and what facilities should be included in it. The research toward this end started under the current contract early in 1970 and culminated in two reports. The first - "A Manual with Examples for the Data Description Language" by Diane P. Smith, April 1971 [SM1] specifies and describes the usage of DDL. The second report, also by Diane P. Smith - "An Approach to Data Description and Conversion," November 1971 [SM2] presents extensive research that constituted the basis for the DDL described in the April 1971 report. These two reports present and justify a highly comprehensive language into which are incorporated all the useful capabilities discovered by our research to date.

The next step was to implement a processor for DDL. The recognition of the need for such a processor was made at approximately the same time in a Tentative Specific Operational Requirement 31-47 issued by the Navy. A proposal for a two-phase design and implementation project was submitted to the Office of Naval Research on March 31, 1971 and work on Phase 1 started in July 1, 1971. This report represents the culmination of Phase 1.

The design of the DDL processor as reported here, and the subsequent implementation, have research and experimental aspects. They employ state of the art techniques and several new approaches to achieve machine and operating system independence and to provide for the adaptability of the Data Description Language Processor to new user environments.

1.2 Summary of Capabilities of the Data Description Language Processor

The DDL processor is designed to satisfy two requirements of data interchange: (1) data (organization) definition and (2) 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 Language [SM1, SM2] provides the descriptive language. The DDL processor described in this report is a set of computer programs which will perform the interpretation and translation. It interprets 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 are summarized below.

1.2.1 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.

1.2.2* Interfacing Files with Different Operating Systems and Different Data Management Systems

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

1.2.3* Interfacing Files with New Computers

Increased requirements and new technology require the phasing out of old computers and their replacement by new computers. DDL would enable files to be prepared for transfer from the old computer to the new.

1.2,4 Integration of Files

Fragmentation of information in numerous files frequently causes great difficulties in use and considerable inefficiency in processing. A DDL will provide the capability to integrate many files into one.

* These facilities are not part of the subset being implemented now. See Table 1-1 for details.

1.2.5 Extraction of Data From Files

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

1.2.6 Creation of New Data Bases

The combination of the above two capabilities allows the translating of one set of files into another set of files.

1.2.7* Interfacing Files to Use New Devices

Advances of technology introduce new input/output devices which enhance the total cost effectiveness of the entire computer system. The change of such input/output devices can be facilitated by the DDL processing of the files from the old devices to the new ones.

1.2 8* Improving Utilization of Computers or Storage

A further application is in the design and operation of data and data management systems. For example, a DDL can be used to create new data structures which can then improve computer or storage utilization.

1.2.9 A Language for Communication Between Humans About Data

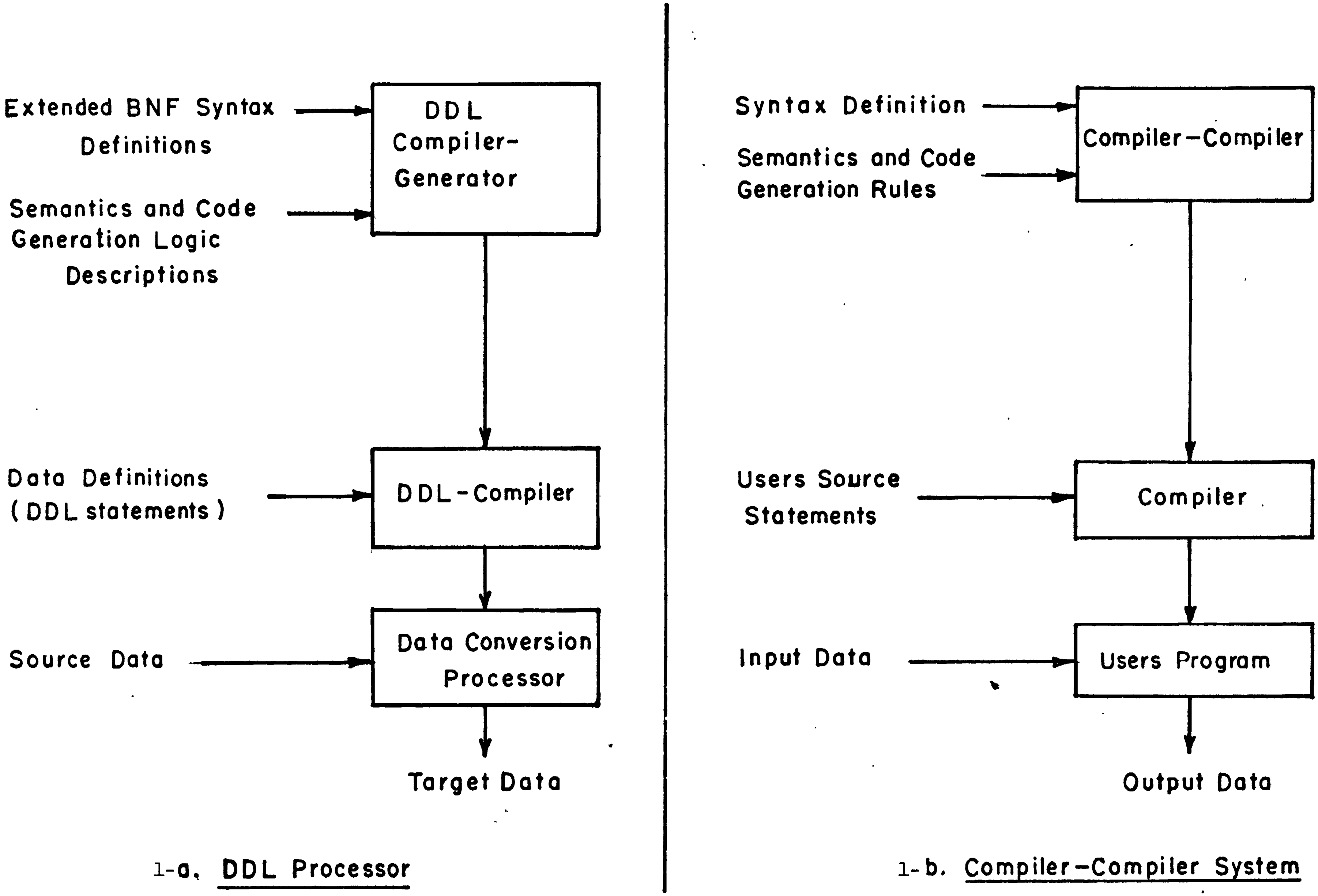
One important application of a DDL is as means of communication between humans. For example, using a DDL a designer of a data base can describe precisely to an applications programmer the exact structure of the data the programmer wants to use. Just as BNF [NA1] is now used to describe the syntax of a language so can a DDL be used to describe data structures.

1.3 Important Features of the Design

The DDL Processor system consists of three major parts. The first part uses syntax and semantic definitions to generate the second part - the DDL Compiler. The DDL Compiler formats and translates DDL Data Definition Statements and produces a computer program which will do the data base conversion. The third part of the processor actually processes the source data base and produces as an output the target data base. This three part structure is illustrated in Fig. 1-1a.

Changes in the definition of DDL will occur only infrequently after an initial language development phase and the compiler generator facility is primarily intended for DDL language development (although it is very helpful in transferring the DDL system to a new computer). Because one of the DDL Processor Implementation design goals is maximum early feedback about design decisions, a subset of the language was selected, and the remaining components will constitute later additions to DDL. These will cause additions to syntax and structure tables. 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 data management systems, or 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, the previously created Data Conversion Processor may be used again.


Extended BNF Syntax Definitions
Semantics and Code Generation Logic Descriptions
DDL Compiler-Generator
Data Definitions (DDL statements)
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-a. DDL Processor

1-b. Compiler-Compiler System

Figure 1-1

COMPARISON OF DDL PROCESSOR AND COMPILER-COMPILER SYSTEM DESIGNS

There are two features in the Design which are of special interest.

(1) The application of the Concept of a Compiler-Compiler to the DDL Processor.

This is illustrated in Fig. 1-1 where the design of the DDL Processor is shown on the left (Fig. 1-1a) as compared to a design of a Compiler Compiler system as shown on the right (Fig. 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 users' object program, output from the compiler.

(2) Machine Independence Through the Use of a Macro Language Design

Machine independence is achieved by implementing much of the DDL Processor in a macro language J-Lang and the translation of J-Lang to a machine language-like processor - K-Lang. Only the interpretation from the machine like processor, K-Lang, to the specific machine (in our case PL/1 and the 360 system) are machine dependent. Thereby, the dependence on the specific programming language and computer are drastically reduced. The macro structure successively being interpreted is illustrated in Figure 1-2, showing the sequence of interpretation from the highest level J-Lang macro language to the eventual execution on the IBM 360. First, the J-Lang is translated into a K-Lang **program.** The K-Lang is easiest and simplest then to interpret and this is where dependence on a specific language and specific computer system is made; K-Lang is interpreted using a PL/1 program (which has been translated into assembly language code) and thus the K-Lang machine is simulated on the IBM 360.

1.4 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 360/75 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, LISP, GPSS and GASP. 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 360 computer system was selected because of PL/1 availability.

1.5 Selection of a Subset of DDL for Implementation

The reports by Diane P. Smith [SM1,2] showed that a model for data description can be divided into three largely independent levels, namely:

1) the record,

2) file and

3) storage levels.

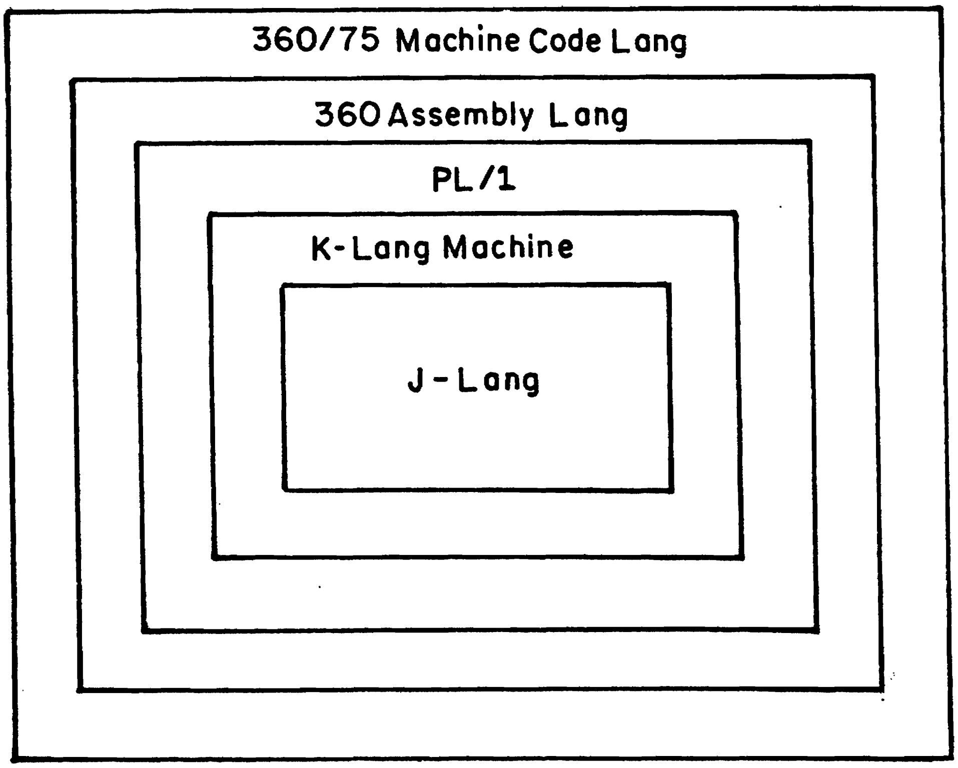


Figure 1-2

SUCCESSIVE INTERPRETATION OF THE MACRO LANGUAGE I-LANG

Using these statements it will be possible to

(a) Accurately describe the structure of data. (This is valuable during system design, and implementation, as well as for later documentation.)

(b) Interface files with different programs and programming languages. (Using the DDL to reformat files before program execution.)

(c) Integrate and extract data in existing files, producing new files. (The "existing file" might be just a deck of cards, or unformatted tape records.)

The subset selected for this implementation will support the record and part of the file level definitions. It will not include the storage structure description facilities. Instead the standard data access methods of the implementation operating system will be used to store data on devices. The statements which are supported by the first implementation and those statements which will be implemented later, are shown in Table 1-1.

1.6 Organization of the Report

This section (Section 1) has responded to a number of questions that were left open, and resolved during the design phase. These consisted of selection of a computer system for the initial implementation - the IBM 360 Mod 75 (soon to be exchanged for IBM 370 Mod 165) at the University of Pennsylvania Computer Center was selected; selection of a programming language for implementation of the DDL Processor - PL/1

Table 1-1
DDL Statements For Initial and Later Implementation

TYPE		INITIAL IMPLEMENTATION STATEMENT	LATER
DESCRIBE	RECORD	CHAR FIELD GROUP RECORD SET CRITEX CRITERION STORRCD RPLVAL CONSTANT CONCODE (DELIM and FILLER options only)	CRITMESSAGE LOGRCD CONCODE
	DEVICE	BLOCK BBLOCK CARDIN CARDOUT TAPEIN (No association list option, no header or trailer and only "SPEC" order) TAPEOUT (same restrictions as TAPEIN)	TAPEIN TAPEOUT DISKIN DISKOUT TTYIN TTYOUT
	PARAMETER	LENGTH CNT EXIST	

Table 1-1 (continued)

TYPE		INITIAL IMPLEMENTATION STATEMENT	LATER
DESCRIBE	FILE	LINK (only sequential links)	OCC
		SEQUEN	ALLOCC
		STORFILE	SOMEOCC
			LINK
			INVLINK
			EMBED
			DIREC
			LOGFILE
EXECUTE	TASK	ASSOCIATE	PROGDATA
		COMBINE	INTERFACE
		CREATE	PARAMPROG
		DISJOIN	EXTEND

was selected; and finally selection of a subset of the commands described in the DDL manual (April 1971) [SM1], that will be implemented in the initial versions of the system. In the selections made and as well as in the design, the experimental nature of the project as well as the need to provide a system as independent as possible of a specific environment were stressed.

Section 2 provides an overview of the design of the processor. Section 3 describes the Compiler-Generator. Section 4 describes the DDL compiler. Section 5 describes the Data Conversion Processor. The remainder of the report (Sections 6, 7, 8) are a specification of the three internal languages used in the system (I-Lang, J-Lang, and K-Lang). A list of references and appendices conclude this report.

2. OVERALL DESCRIPTION OF THE DESIGN OF THE DDL PROCESSOR

The "DDL Processor System" is actually three processors (i.e. sets of computer programs). The first is the DDL Compiler Generator, the second is the DDL Compiler, and the third is the Data Conversion Processor (see Fig. 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 systems was made in Fig. 1-1; the Data Conversion Processor corresponds to an executable user program, the DDL compiler corresponds to a Cobol, Fortran or PL/1 compiler, and the DDL Compiler Generator corresponds to a Compiler-Compiler which is used to produce the Cobol, Fortran or PL/1 compiler. 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 files in format A to files in format B). To aid in the generation of Data Conversion Processors, there is the DDL Compiler.

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. These statements are read by the DDL Compiler, which produces a new Data Conversion processor. It is important to note that, just as it is not necessary to compile a Cobol program each time it is used, it is not necessary to create a new Data Conversion Processor each time it is used. Only when the functions of the processor change is that required.

The DDL Compiler, and the Data Conversion Processors produced by it, are the components of the DDL Processor System that most users will need. The other component, the DDL Compiler Generator, is the set of programs used to create 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. But just as the average user does not often modify his Cobol compiler, he would not often modify his DDL Compiler.

An overview of the DDL Processor System is shown in Figure 2-1. Each processor is surrounded by a broken line. Programs are shown in rectangles, input/output are shown in trapezoids. When the output is a program double lines are used to show the destination of the output. The following paragraphs describe each of the three components of the DDL Processor System.

2.1 The DDL Compiler-Generator

The DDL Compiler-Generator is composed of three parts - the Syntactic Analysis Program Generator, the I-J Translator (see Fig. 2-2), and the J-K Translator. The relationship of these three is shown in Figure 2-3.

The first part of the Compiler Generator is the Syntactic Analysis Program Generator. The input to the Syntactic Analysis Program Generator is a syntax description coded in Extended Backus-Naur Form (EBNF).

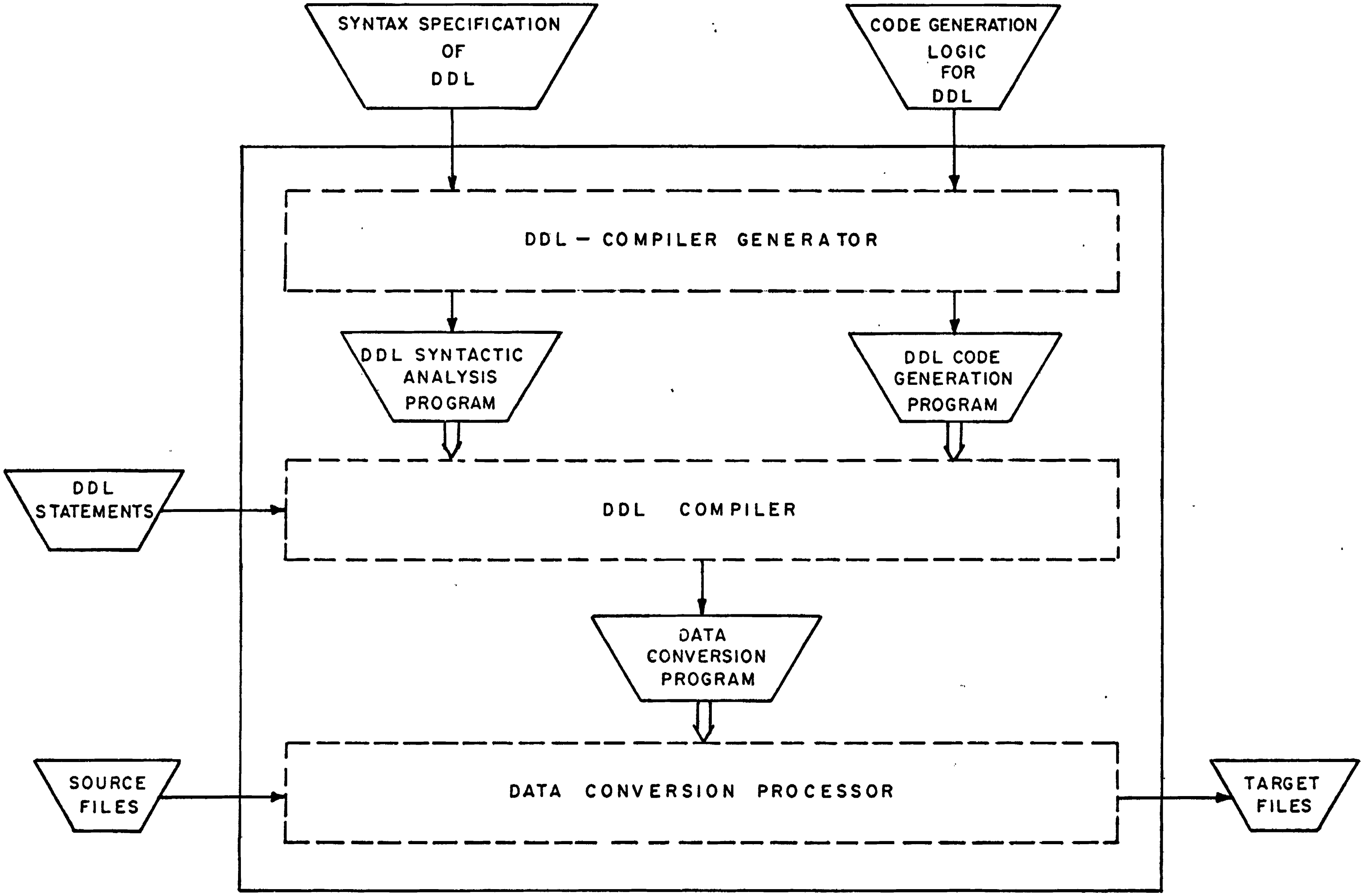

SYNTAX SPECIFICATION OF DDL
CODE GENERATION LOGIC FOR DDL
DDL – COMPILER GENERATOR
DDL SYNTACTIC ANALYSIS PROGRAM
DDL CODE GENERATION PROGRAM
DDL STATEMENTS
DDL COMPILER
DATA CONVERSION PROGRAM
SOURCE FILES
DATA CONVERSION PROCESSOR
TARGET FILES

Figure 2-1
DDL – PROCESSOR SYSTEM

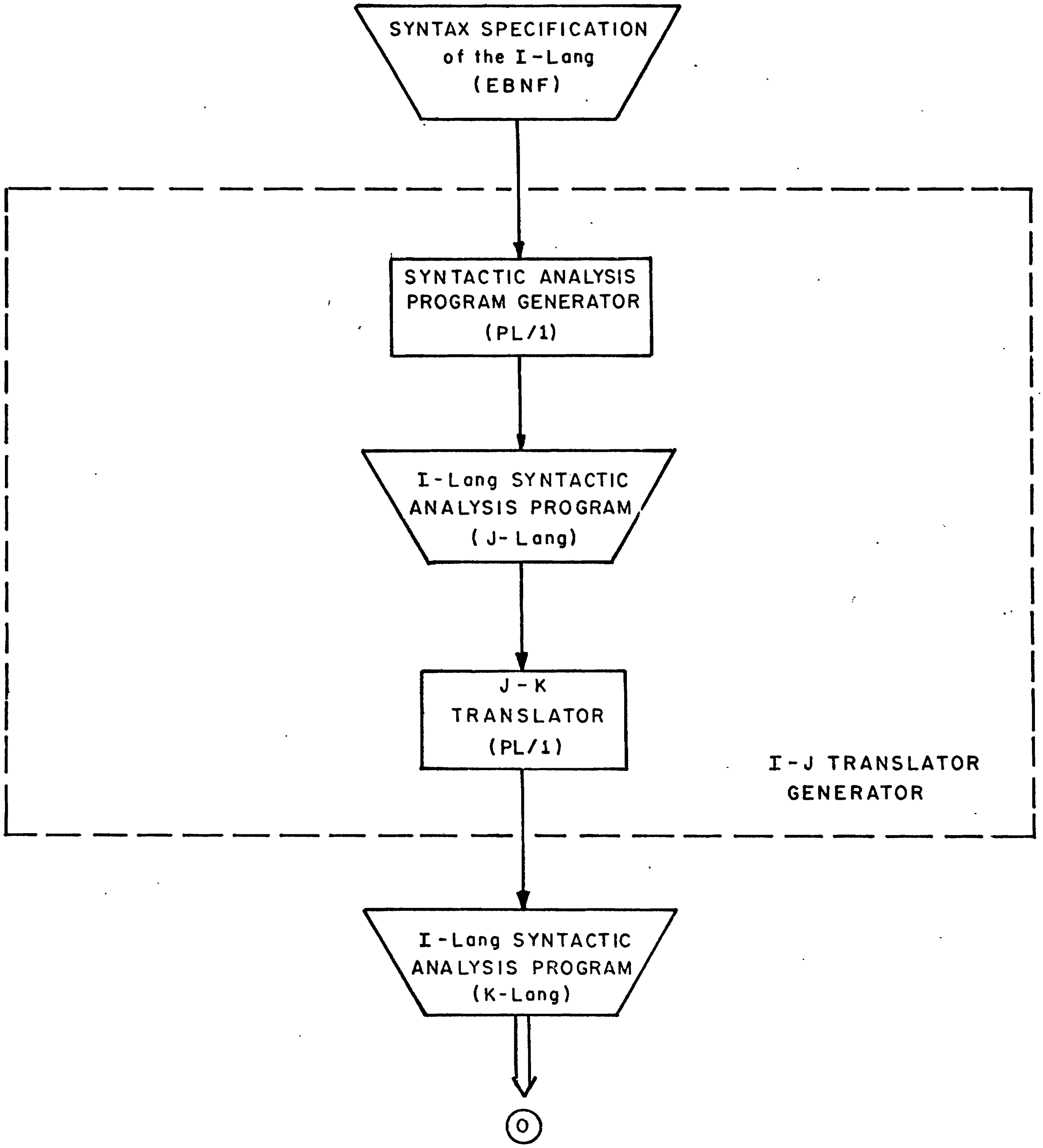


Figure 2-2

Generating a Translator
From I-Lang to J-Lang

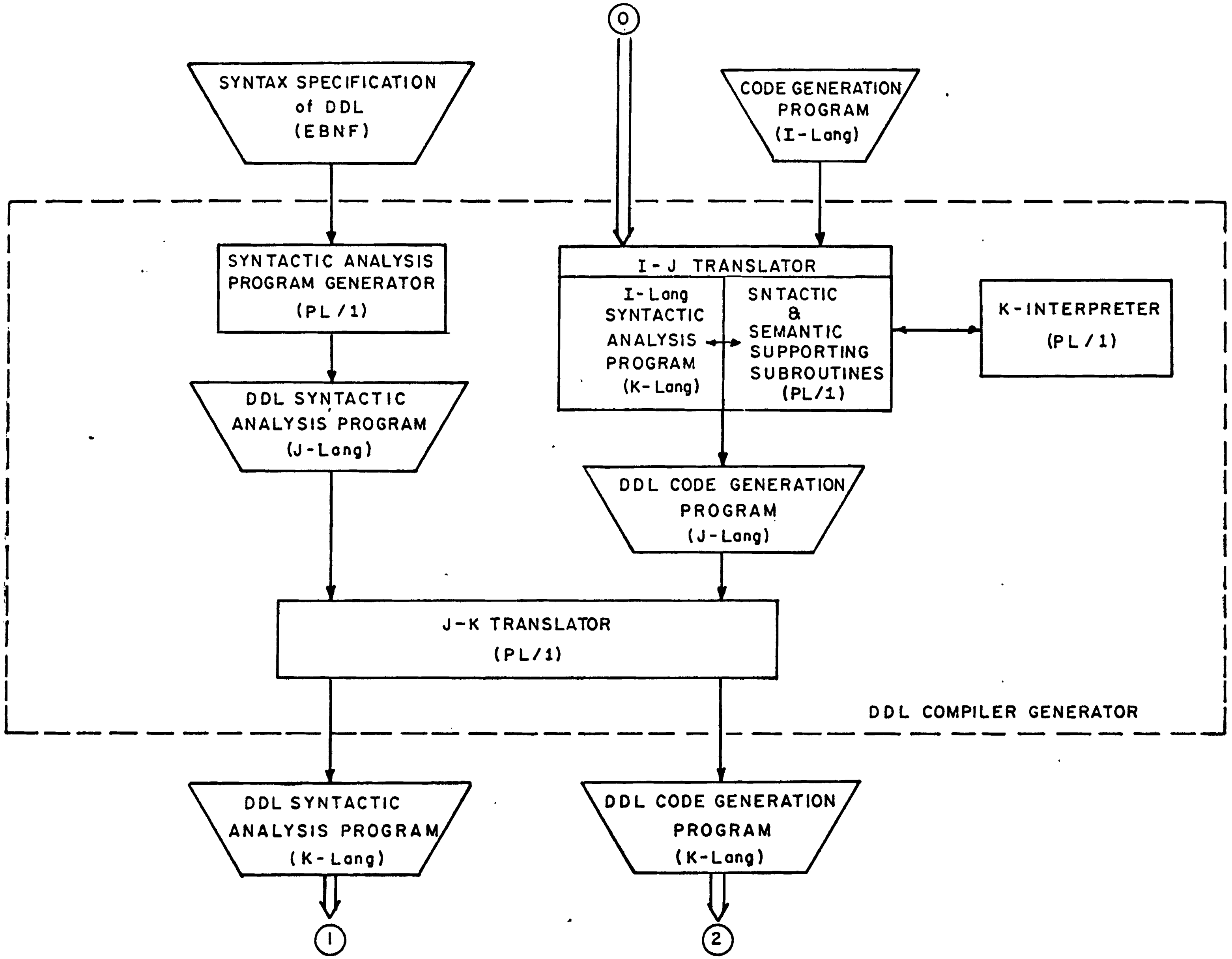

0
SYNTAX SPECIFICATION of DDL (EBNF)
CODE GENERATION PROGRAM (I-Lang)
SYNTACTIC ANALYSIS PROGRAM GENERATOR (PL/1)
I-J TRANSLATOR
I-Lang SYNTACTIC ANALYSIS PROGRAM (K-Lang)
SNTACTIC & SEMANTIC SUPPORTING SUBROUTINES (PL/1)
K-INTERPRETER (PL/1)
DDL SYNTACTIC ANALYSIS PROGRAM (J-Lang)
DDL CODE GENERATION PROGRAM (J-Lang)
J-K TRANSLATOR (PL/1)
DDL COMPILER GENERATOR
DDL SYNTACTIC ANALYSIS PROGRAM (K-Lang)
DDL CODE GENERATION PROGRAM (K-Lang)
1
2

Figure 2-3
Generation of the DDL Compiler

(EBNF notation will be described in Section 3.2.) The Syntactic Analysis Program Generator produces a program which is the Syntax Analysis phase of the DDL Compiler. The second part of the Compiler Generator is the I-J Translator. Its input is the Code Generation logic, written in I-Lang. The I-J Translator then produces a program which contains the basic code generation logic used in the DDL Compiler.

The output of the Syntactic Analysis Program Generator and the I-J Translator are J-Lang programs. The third major component of the DDL Compiler Generator, the J-K Translator, is used to translate these J-Lang programs to K-Lang programs, which are then suitable for interpretive execution.

2.2 The DDL Compiler

An overview of the DDL Compiler is shown in Figure 2-4. The DDL Compiler contains five major parts: (1) a Syntactic Analysis Program (SAP), (2) a Code Generation Program (CGP), (3) a series of supporting subroutines, (4) an interpreter, (5) a J-K Translator.

The function of the SAP is to perform the syntax analysis on the DDL statements. The function of CGP is to generate the Data Conversion Program (in J-Lang). Since both SAP and CGP are in K-Lang, the K-Interpreter is used to execute them (the K-Interpreter logically converts the IBM 360 machine into the K-machine). The J-K Translator accepts as input the J-Lang code produced by CGP and produces the Data Conversion Program in K-Lang.

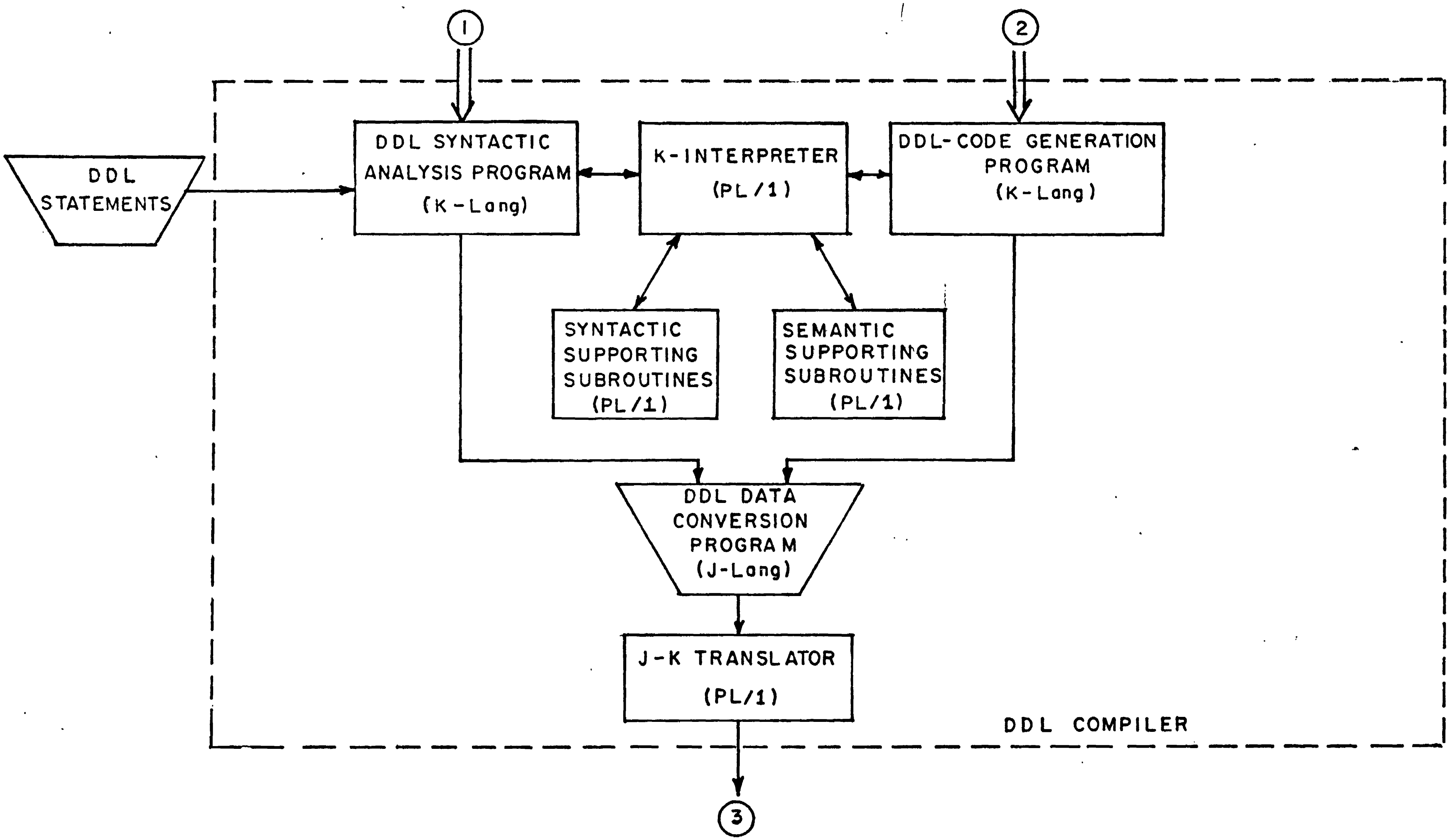

1
2
DDL STATEMENTS
DDL SYNTACTIC ANALYSIS PROGRAM (K-Lang)
K-INTERPRETER (PL/1)
DDL-CODE GENERATION PROGRAM (K-Lang)
SYNTACTIC SUPPORTING SUBROUTINES (PL/1)
SEMANTIC SUPPORTING SUBROUTINES (PL/1)
DDL DATA CONVERSION PROGRAM (J-Lang)
J-K TRANSLATOR (PL/1)
DDL COMPILER
3

Figure 2-4

Compilation of a DDL Program

The supporting subroutines are a set of functions used during Syntax Analysis and in Code Generation; these subroutines, as well as the K-Interpteter, and the J-K Translator are written in PL/1.

Phase 1 of the compilation process is the execution of the Syntactic Analysis Program, with its supporting subroutines. Phase 1A includes most of the Semantic Interpretation logic; it completes the internal tables which will be used in Phase 2 (Code Generation). Phase 3 of compilation is the execution of the J-K Translator; this produces the machine-level language (K-Lang) which is the output of the DDL compiler - namely, the Data Conversion Processor.

2.3 The Data Conversion Processor

The Data Conversion Processor is a set of K-Lang programs and data which was produced by the code generation logic of the DDL compiler. It is composed of

(a) a Data Conversion Program

(b) a Data Structure to perform the data conversion,

(c) an interpreter, and

(d) a set of Run Time Supporting Subroutines.

The Data Structure is a network of threaded lists. The nodes are data descriptor entries; they are used by the Data Conversion Program to aid in parsing the source data, and to format the output data. Because the Data Conversion Program is encoded in K-Lang, it must be interpretively executed by a K-Interpteter. The other components of the Data Conversion Processor, the Run Time Supporting Subroutines, perform functions such as record level input-output, main memory allocation, character set conversion, extension or truncation of fields, and data type conversion. An overview of the Data Conversion Processor is shown in Figure 2-5.

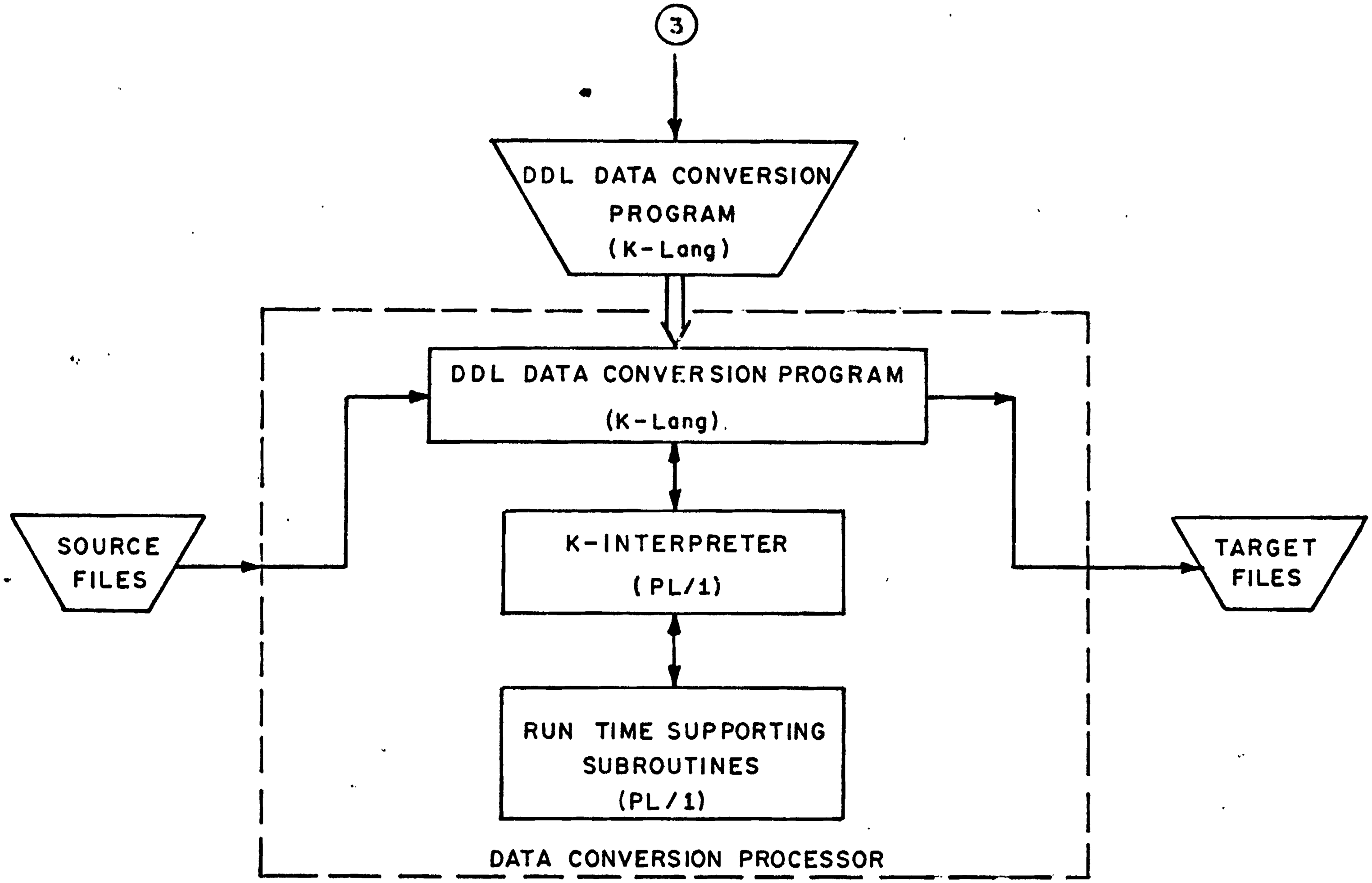

3
DDL DATA CONVERSION PROGRAM (K-Lang)
DDL DATA CONVERSION PROGRAM (K-Lang)
SOURCE FILES
K-INTERPRETER (PL/1)
RUN TIME SUPPORTING SUBROUTINES (PL/1)
TARGET FILES
DATA CONVERSION PROCESSOR

Figure 2-5

Execution of a Data Conversion Program

Figure 2-6 shows a simplified view of the flow in the DDL compiler and Data Conversion Processor. The K-Interpreter and supporting subroutines have been omitted to allow the logical flow of control and data to be seen more easily. Note that the Code Generation Program of the DDL Compiler produces both a network of data descriptors and a Data Movement Program. These two become the Data Conversion Processor. During the execution of the Data Conversion Processor the data descriptor network is used to parse the input data and act as a framework which the Data Movement Program uses to construct the output data files.

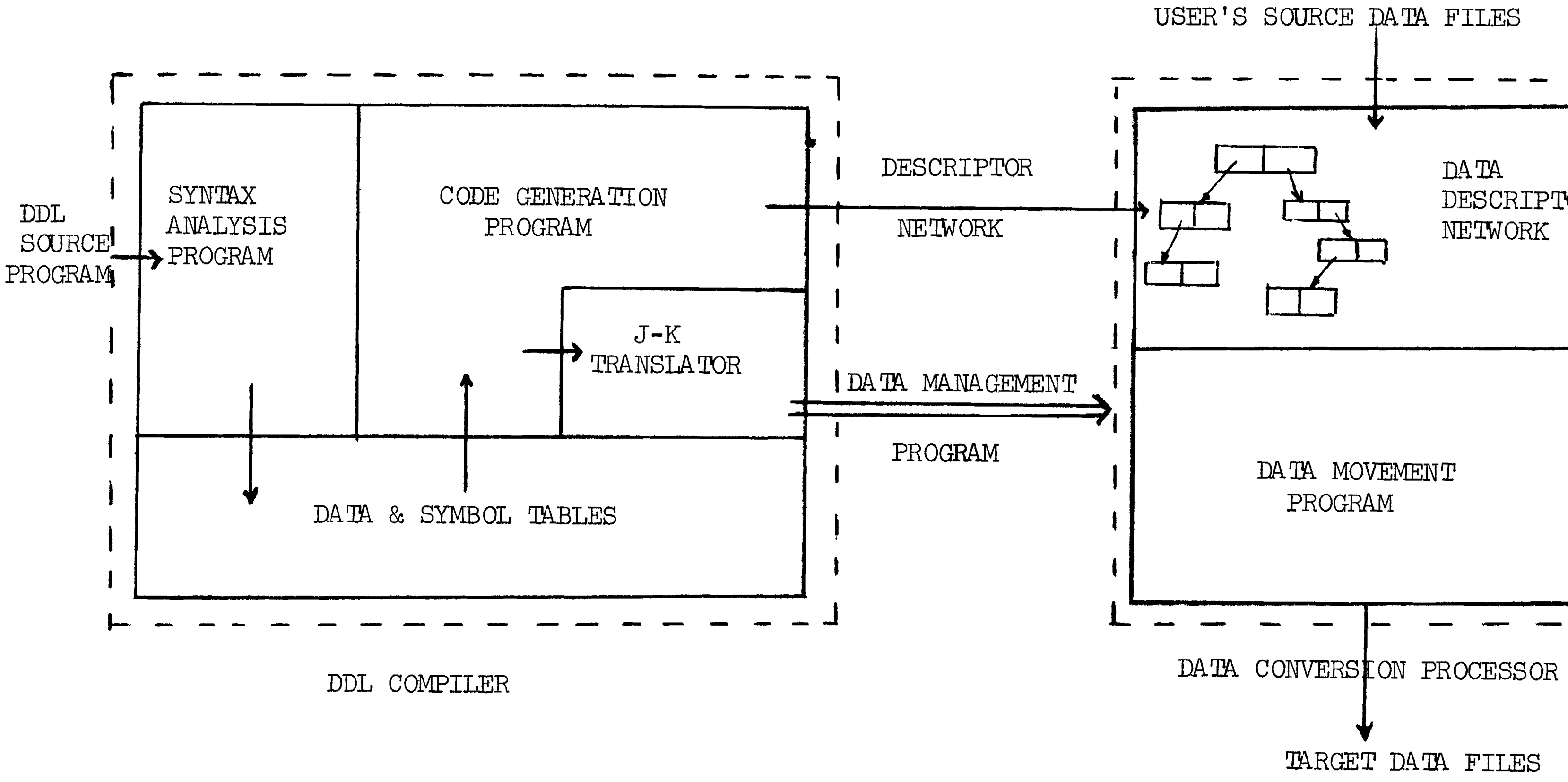


Figure 2-6

Overview of DDL Compiler and Data Conversion Processor

2.4 Summary of Languages Used in the DDL Processor System

The DDL processor system relies heavily on the I, J, K and EBNF languages; therefore the translators and interpreters for these will be used to illustrate the design techniques used in DDL Processor.

2.4.1 The Syntactic Analysis Program Generator

This program is used in two places in the system: (1) to generate the syntactic analysis phase of the DDL Compiler and (2) to generate the main procedures of the I-J Translator. Its implementation cost is thus divided between two subcomponents. The Syntactic Analysis Program Generator is valuable during initial implementation as it allows the syntax of DDL, and of I-Lang to change without requiring manual rewriting of the syntax analysis programs (only the EBNF descriptions must be changed).

In addition, the EBNF source statements provide excellent documentation of the rules of grammar for the language and of the logic of the syntactic analysis program. The Syntactic Analysis Program Generator also centralizes the generation of syntax-checking programs.

2.4.2 The I-J Translator

This program is used (in conjunction with the J-K Translator) to generate the Semantic Analysis/Code Generation phase of the DDL Compiler. Most of its logic is generated using the Syntactic Analysis Program Generator, thereby keeping implementation cost low. Other benefits of the I-J Translator lie in the fact that it provides a method for translating the self-documenting, human readable I-Lang into the encoded J-Lang form used internally by the DDL processor. Thus encoded, the logic can be translated by the same program which is used in the DDL compiler (the J-K Translator).

2.4.3 The J-K Translator

It is used in three places in the system: (1) it forms the third phase of the DDL Compiler, (2) it is used at the end of the Compiler Generator program which produces the Semantic Analysis/Code Generation logic, (3) it is used at the end of the Compiler Generator program which produces the Syntactic Analysis Program. Although it is not a simple program, its cost is distributed over three important areas of DDL implementation. The J-K Translator performs all the K-Lang generation done by the DDL Processor. Any extensions to the K-Lang affect only it and the K-Interpreter. Together with the K-Interptеter, it provides the basis for the machine independence of the DDL Processor.

2.4.4 The K-Interptеter

This program simulates the K-machine. It is used to "execute" the DDL compiler, and the program which the compiler produces - the Data Conversion Processor. It decodes machine-level instructions and performs the elementary commands indicated by them. It is the primary method used to achieve machine independence. Because it operates interpretatively it provides an excellent place in which to place debugging tools, program performance monitors, and user data validity tests.

3. THE DDL COMPILER GENERATOR

3.1 Compiler-Compilers

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 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 [NA1] 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.

The following excerpt from a paper by J. A. Feldman on FSL (Formal Semantic Language) [Fe 1] outlines the basic method and should serve as an adequate introduction to our discussion of compiler-compilers.

> When a compiler for some language, $\mathcal{L}$, is required, the following steps are taken. First the formal syntax of $\mathcal{L}$, expressed in a syntactic metalanguage, is fed into the syntax loader. This program builds tables which will control the recognition and parsing of programs in the language $\mathcal{L}$. Then the semantics of $\mathcal{L}$, written in a semantic metalanguage, is fed into the semantic loader. This program builds another table, this one containing a description of the meaning of statements in $\mathcal{L}$. Finally, everything

to the left of the double line [in Fig. 3-1] is discarded leaving a compiler for $\mathscr{L}$.

The resulting compiler 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 P. Ingerman [IN 1]; W. M. McKeeman, J. J. Horning, D. B. Wortman [MC 1]; Feldman [Fe 1]; Brooker and Morris [Br 1].

3.2 The DDL Compiler Generator

The correspondence between the compiler-compiler described by Feldman [Fe 1] and the DDL Compiler Generator is given in the following sentences:

a) The metalanguage used to express the formal syntax (of DDL) is EBNF (see Section 3.3).

b) The *Syntax Loader* is the (SAPG) Syntactic Analysis Program Generator, i.e., the EBNF of DDL is fed into the SAPG. This program produces the SAP (Syntactic Analysis Program) which will control the recognition and parsing of programs in the language DDL.

c) The semantic metalanguage is the I-Lang (see Section 6).

d) The *Semantic Loader* corresponds to the (CGPG) Code Generation Program Generator (or I-J Translator). The Code Generation logic (in I-Lang) is fed into the CGPG. This translator produces the CGP (Code Generation Program) which will control code generation.

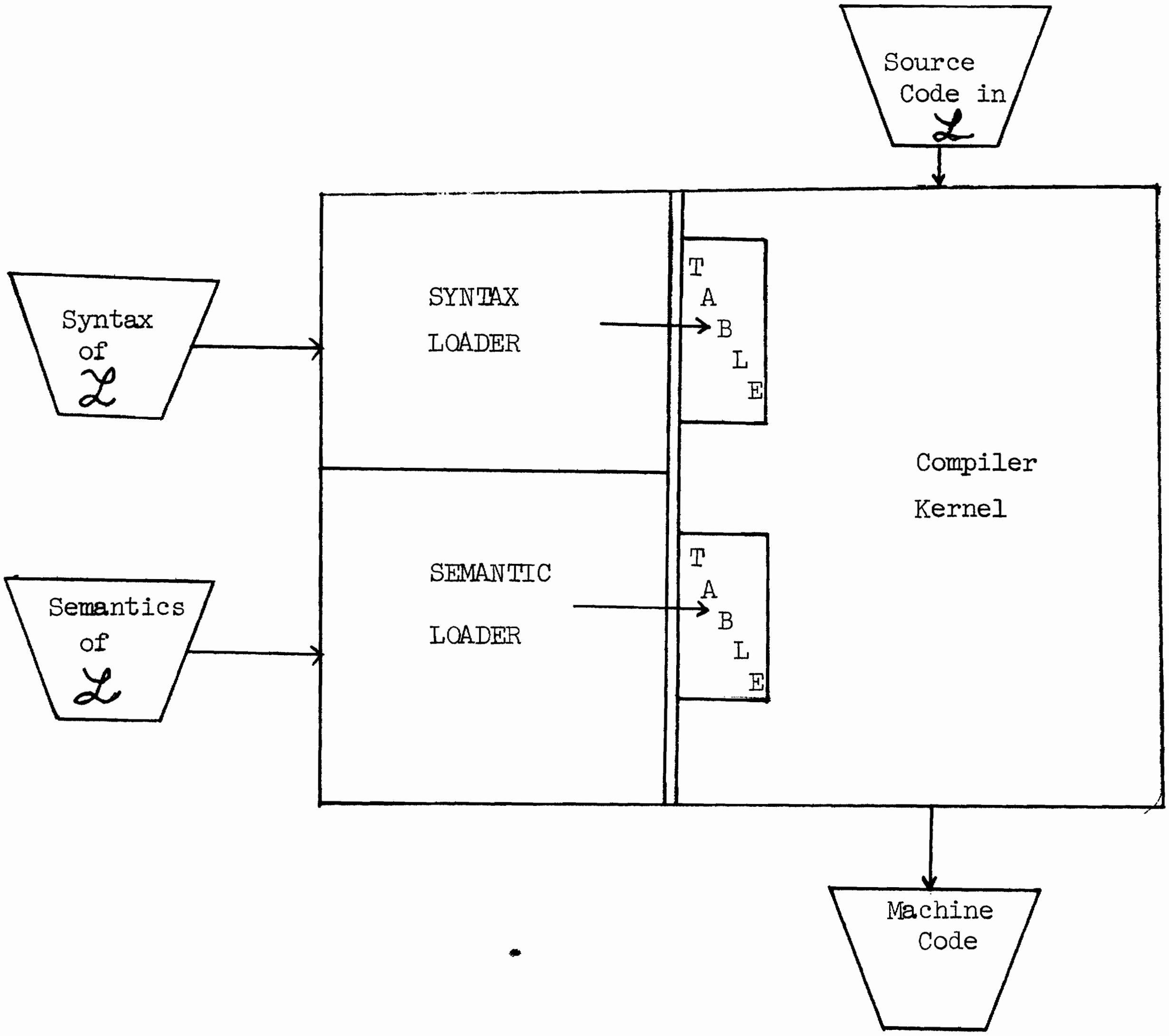


Figure 3-1

A Compiler-Compiler

This figure appears in [Fe 1].

e) The compiler kernel corresponds to the set of Syntactic and Semantic supporting subroutines.

The corresponding diagram for the DDL Compiler Generator is shown in Figure 3-2. Everything to the left of the double line in Figure 3-2 is discarded leaving a compiler for DDL.

In the next section a description of the EBNF metalanguage is presented. In the following ones the description of each of the components of the DDL Compiler Generator is presented.

3.3 Description of the Metalanguage (EBNF)

The formal syntax of the DDL language will be described with the aid of a metalinguistic formulae Extended BNF (EBNF).[†] The interpretation of the basic EBNF is best explained by an example.

```
1    < CRITERION STMT > ::= CRITERION (< CRITERION EXP NAME >";
                                 < RPLVAL STMT >")

2    < CRITERION EXP NAME > ::= < UUDN >

3    < RPLVAL STMT > ::= RPLVAL (< DATA NAME >, < REP VAL >)

4    < DATA NAME > ::= < REF NAME >

5    < REP VAL > ::= < REF NAME > | < CONSTANT STMT >

6    < REF NAME > ::= < UUDN > "OF < REF NAME >"
                    | < IUDN > "OF < REF NAME >"

7    < UUDN > ::= < ALPHA CHAR > "< SINGLE STRING >"*

8    < SINGLE STRING > ::= < ALPHA CHAR > | < DECIMAL DIGIT >
```

[†] A complete description of the Syntax of DDL is presented in Appendix A.

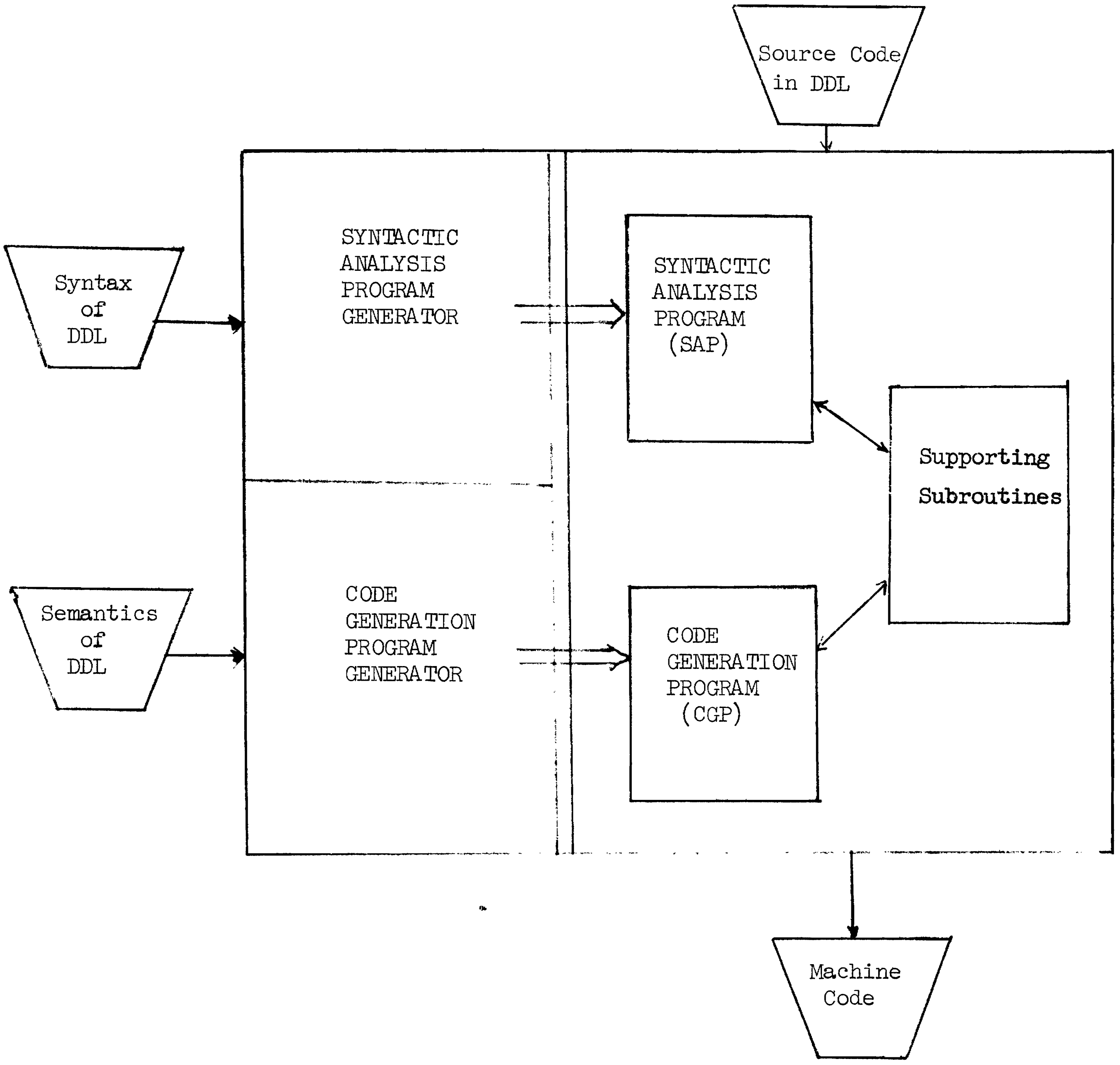

Source Code in DDL
Syntax of DDL
SYNTACTIC ANALYSIS PROGRAM GENERATOR
SYNTACTIC ANALYSIS PROGRAM (SAP)
Supporting Subroutines
Semantics of DDL
CODE GENERATION PROGRAM GENERATOR
CODE GENERATION PROGRAM (CGP)
Machine Code

Figure 3-2

The DDL Compiler Generator

Sequences of characters enclosed in the bracket < > represent BNF metalinguistic variables whose values are sequences of symbols. The marks ::= and | (the latter meaning OR) are BNF metalinguistic symbols. The extension of BNF (EBNF) is through use of double quotes " and * as also metalinguistic symbols. Double quotes are used to indicate that the item enclosed by these symbols may appear zero or one times in the "object" formulae. If the close quotes are followed by *, the symbols may appear zero or more times.[†] Any mark in a formula, which is not a variable or a connective, denotes itself (or the class of marks which are similar to it). Juxtaposition of marks and/or variables in a formula signifies juxtaposition of the sequences denoted.

Therefore the above formulae (1 through 8) gives a rule for the formation of:

< CRITERION STMT >, in line 1. It indicates that the < CRITERION STMT > begins with the keyword CRITERION, followed by "(", followed by some value of the variable < CRITERION EXP NAME >, and that ;< RPLVAL STMT > may appear zero or one times, finally ")" must appear.

< CRITERION EXP NAME >, in line 2. It indicates that the variable < CRITERION EXP NAME > will take some value of the variable < UUDN >.

< RPLVAL STMT >, in line 3. It indicates that the < RPLVAL STMT > begins with the keyword RPLVAL, followed by "(", followed by some value of < DATA NAME >, followed by the separator "," followed by some value of the variable < REP VAL >, followed by ")".

< DATA NAME >, in line 4. It indicates that < DATA NAME > will take some value of the variable < REF NAME >.

[†] To allow nested quotes, the following conventions for preparation of source input cards were adopted:

punch a $\binom{12}{8}$ in the column preceding an open quote;
punch a $\binom{12}{0}$ in the column following a close quote.

< REP VAL >, in line 5. It indicates that < REP VAL > will either be (1) some value of the variable < REF NAME > or (2) some value of the variable < CONSTANT STMT >.

< REF NAME >, in line 6. It indicates that < REF NAME > will either be (1) some value of the variable < UUDN >, and that the keyword OF followed by some value of the variable < REF NAME > (note here the right recursion of the production) may occur zero or one times, or (2) some value of the variable < IUND >, and that the keyword OF followed by some value of the variable < REF NAME > may occur zero or more times.

< UUDN >, in line 7. It indicates that < UUDN > will take some value of the variable < ALPHA CHAR > followed by the occurrence of zero or more times of the value of the variable < SINGLE STRING >. (Note the use of the metalinguistic symbol * to achieve right repetition.)

< SINGLE STRING > in line 8. It indicates that < SINGLE STRING > will either be (1) some value of the variable < ALPHA CHAR > or (2) some value of the variable < DECIMAL DIGIT >.

The above description of EBNF is sufficient for the syntax description of DDL (see Appendix A), but to allow the subroutine call specification permitted by the SAPG, EBNF must be further extended. This is described in the next paragraph.

3.3.1 Extended BNF with Subroutine Calls

The extensions to EBNF allow semantics to be added to the syntactic description. They permit the specification of subroutines which will be executed during syntactic analysis, and thus simplify the machine-assisted program generation of the SAP. The programmer provides a non-procedural description of the language as input and then receives procedural logic (a program) which will perform syntax-checking of the language he described. The program (SAP) will also contain calls to the subroutines which produce the Symbol Table and Data Tables.[*] The SAP is produced in J-Lang by the SAPG.

An example of an EBNF statement with subroutine calls is:

< item 1 > ::= < name > / .SUB1/ : (< part 1 >//",/.SETP2/ < part 2 >//"*)

This statement means:

To test for (recognize) a syntactic unit (or token) called "item 1", first see if the current pointer in the input string points to the start of a string which satisfies the syntactic definition of "name". (Do this by calling a recognizer routine.) If the recognizer exits false go to a system routine called .FAIL (see Section 4.3.3 for a discussion of this routine). If < name > was recognized, then call .SUB1. Upon return from this routine check that the next token in the input is a ":". If not, call .FAIL. Similarly check the next two tokens in the input string, i.e., for "(" and < part 1 >. If they are recognized then call .VCALL (this is indicated by //), .VCALL pops the "subroutine vector stack" (see Sect. 4.3.2 for the discussion of this routine) and gives control to the routine whose address was on top of the stack. After returning from that subroutine the syntactic analyzer tests for the next token, but failure to find a match does not cause a call to .FAIL, because the token is the first element in an optional group as indicated by the " symbol. (Note that quotes surround the optional group.) In this case failure causes the scan to skip to the first token after the " or "* indicating end of optional group. This token is ")" in this case[‡]. If the ","

[*] See Section 4. and 4. for a description of the Symbol Table and Data Table respectively.

[‡] When references are made to punctuation marks in the text they will be enclosed in double quotes.

was found a call is made to .SETP2, then a test is made for the token < part 2 >. If this fails, then .FAIL is called. (Note that only if the first item of an optional group is present then all other items in the group must be present.) After successfully recognizing < part 2 >, a call is made to the vector subroutine .VCALL. After return, the syntax analyzer repeats the check for the "," before /.SETP2/. This loop of testing for the optional part is signified by the * following the " (close quotes). If no * was present the scan would have proceeded to the next token, ")". Eventually the loop will terminate when a "," is not found.

There are three other points which should be mentioned with respect to EBNF syntax.

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

(2) an EBNF statement line may be nothing more than subroutine calls.

(3) optional parts may be nested without limit.

3.4 The Syntactic Analysis Program Generator (SAPG)

The SAPG will be hand coded in PL/1; it consists of three phases or passes. An overview of the creation of the SAPG is shown in Figure 3-3.

3.4.1 Pass 1 (The algorithm is given in Appendix E, Part I.)

Pass 1 of the SAPG reads the EBNF source statements (productions) and encodes them into a table called the Encoded Table.

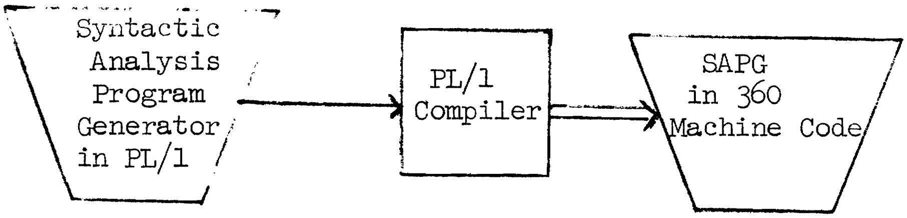


Figure 3-3

Creation of the Syntactic Analysis Program Generator (SAPG)

Lexical units of each production are encoded in the Encoded Table which is organized by production. The units are typed as: (a) terminal symbols, (b) subroutine calls or (c) non-terminal symbols. The entries for each lexical unit are actually pointers to tables containing the types listed above, one table for each type. The non-terminal symbols are further divided into two tables. Non-terminals appearing on the left side of a production are kept in the Symbol Table. Non-terminals appearing on the right side of a production are placed in a work table for use in Pass 2. All the symbols within a table are unique. (This is done to conserve storage in the Encoded Table, since the same entry in tables may be referred by different entries in the Encoded Table.) Each production thus far found acceptable is saved in the Encoded Table for processing by Pass 2 of the SAPG.

The symbol table consists of all non-terminal symbols appearing as the left part of a production, i.e., appearing to the left of the metalinguistic symbol "::=". These symbols are the syntactic units of the language being defined. A subroutine (.ENTSYM) searches the symbol table for the current symbol. If it already appears in the table, an ambiguity is present in the language, and the entire production is flagged and rejected. If an entry is not found, a new entry is created for the symbol. (Rejected productions are noted, but not analyzed during the second pass of the SAPG.)

The production is also checked to see if it is "singular". A singular production is a production of the form < item 1 > ::= < item 2 >. This type of production effectively equates < item 1 > and < item 2 >. < item 2 > may be substituted for < item 1 > in any production which contains < item 1 >. Doing so eliminates unnecessary intermediate levels of the language specification. (This substitution will be done internally by the SAPG. The listing of the source statements produced by the SAPG will be an exact copy of the source statements, along with diagnostic information.) Singular productions are flagged for Pass 2 of the SAPG, so that all possible intermediate levels may be eliminated.

Detailed scanning of the EBNF source statements is accomplished by a lexical routine (.LEXEBNF). See Appendix E, Part IA. This routine returns to the SAPG the next lexical unit in the input stream as well as an indication of the beginning of a new production. (A lexical unit is a metalinguistic symbol, a non-terminal symbol (i.e., a string of characters beginning with "<" and ending with ">"), a separator (i.e., a character such as ",", "(", ")", ";"), or a terminal symbol (i.e., any string of characters which does not include a metalinguistic symbol, a non-terminal symbol, a separator, or embedded blanks).) The lexical routine inputs source records as necessary to fulfill the requests of the SAPG for a new lexical unit.

3.4.2 Pass 2 (The algorithm is given in Appendix E, Part II.)

Pass 2 of the SAPG scans the outputs of the first pass to resolve all symbolic references. As many as possible intermediate levels of singular productions are removed so that there will be no singular production references remaining. Each singular production is resolved either the first time it is referenced by another production or as it is encountered during the course of scanning the output from the first pass, whichever occurs first. The resolution of singular productions need be performed only once. If any non-terminal symbol on the right side of a production is not found in the symbol table, the production is flagged. If there are no flagged productions, the output from Pass 2 is saved for Pass 3. Otherwise, the generation of SAP is discontinued.

3.4.3 Pass 3 (The algorithm appears in Appendix E, Part III.)

Pass 3 of the SAPG outputs the J-Lang SAP. The SAP depends heavily on the PROCEDURE- and DO-group and IF-THEN-ELSE-clause features of the J-Lang. Each production is encoded as a PROCEDURE which will return a true or false value depending upon whether or not the syntactic unit is recognized. The scan for syntactic units is accomplished by nexted IF-THEN-ELSE clauses using DO groups. (Optionally repetitive items also use the GOTO statement to cause scanning for another syntactic unit of the same type previously sought for.) This type of construction also eliminates the necessity of look-ahead scanning of the EBNF source statements. The exclusive nature of the EBNF descriptions is accomplished by the exclusive nature of the THEN and ELSE clauses.

Pass 3 of the SAPG operates in an error free environment since it is not called by Pass 2 unless no errors have been detected. Thus no code will be generated for detectable erroneous language specifications.

3.5 The Code Generation Program Generator (I-J Translator)

The I-J Translator is composed of the SAP for I-Lang and a set of syntactic and semantic supporting routines. The SAP for I-Lang is produced using the SAPG described above (Section 3.4). The syntax description of I-Lang expressed in EBNF is fed to the SAPG; the result is the SAP for I-Lang. The I-Lang SAP performs syntax-checking of the I-Lang and, because of the subroutine calls embedded in it, will produce the internal symbol and data tables. The syntactic and semantic supporting subroutines are also used to help perform the syntax analysis and to transform the I-Lang statements to J-Lang statements. For a description of the I-Lang see Section 6 and for J-Lang see Section 7.

Because the output of the SAPG (the SAP for I-Lang) is in J-Lang, it must be translated from J-Lang into K-Lang (for a description of K-Lang see Section 8). To perform this translation the J-K Translator will be used (the description of the J-K translation is given in Section 4.4). The J-K translator will not perform any syntactic analysis here since J-Lang was produced by the SAPG and it need not be checked. An overview of the creation of the I-J Translator is shown in Figure 3-4.

Because the J-Lang is in a one-to-one correspondence with the I-Lang, little semantic and code generation is required and the majority of the logic in the I-J Translator is for J-Lang syntax checking.

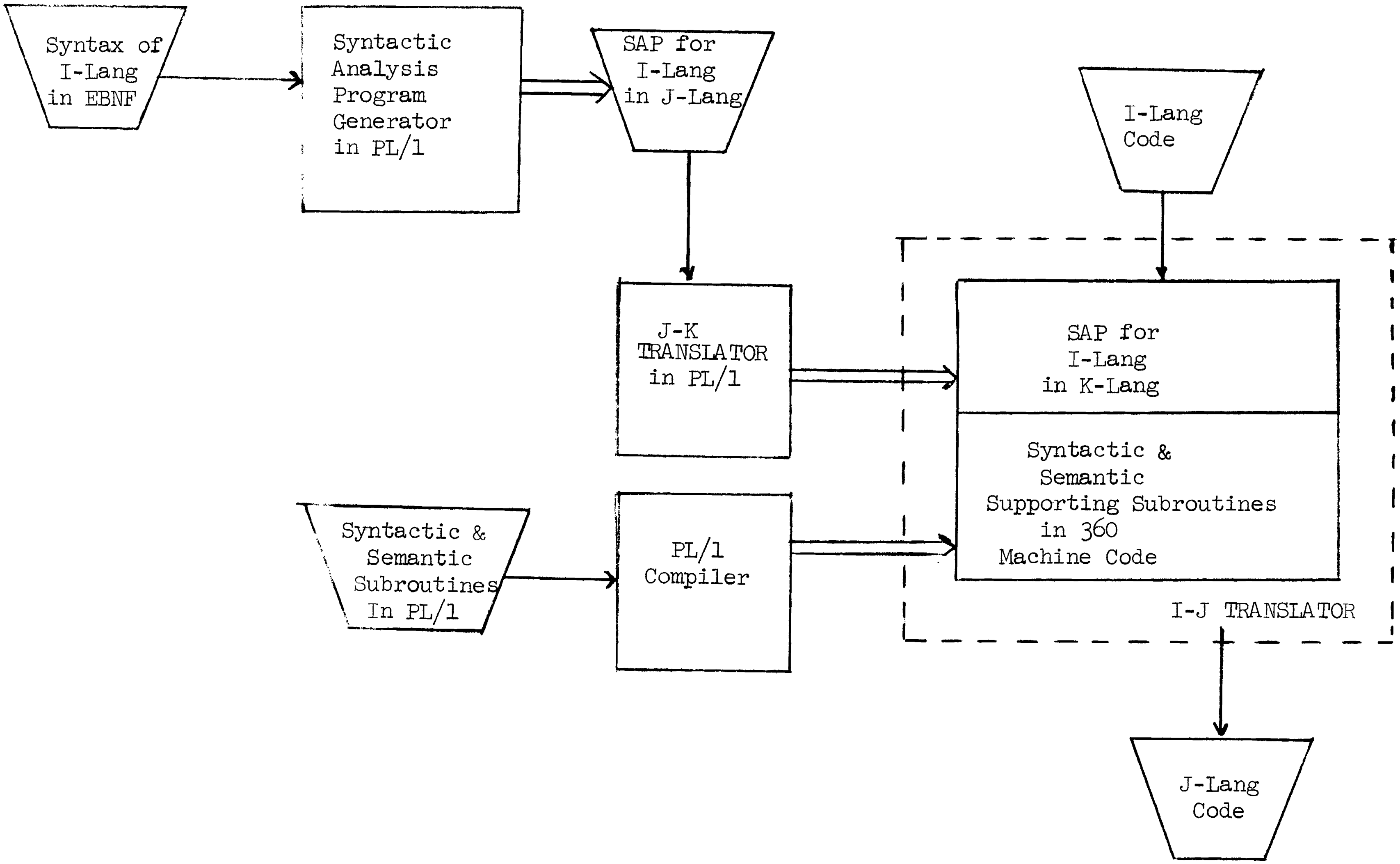

Syntax of I-Lang in EBNF
Syntactic Analysis Program Generator in PL/1
SAP for I-Lang in J-Lang
I-Lang Code
J-K TRANSLATOR in PL/1
SAP for I-Lang in K-Lang
Syntactic & Semantic Supporting Subroutines in 360 Machine Code
I-J TRANSLATOR
Syntactic & Semantic Subroutines In PL/1
PL/1 Compiler
J-Lang Code

Figure 3-4

Creation of the I-J Translator

This completes the description of the SAPG and the I-J Translator. In the section that follows all the components of the DDL Compiler Generator are joined and the description of the overall logic of the DDL Compiler Generator is given.

3.6 DDL Compiler-Generator Logic Flow

An overall description of the components of the DDL Compiler-Generator is given in Figure 3-5. The DDL Compiler Generator accepts as input the EBNF description of DDL and the Code Generation logic in I-Lang and produces the DDL-SAP and the DDL-CGP respectively.

3.6.1 Creation of the SAP for DDL

To produce the SAP for DDL a syntax specification of DDL described in EBNF is fed to the SAPG, the output is the SAP for DDL in J-Lang. This output is then fed to the J-K Translator and the result is the SAP for DDL in K-Lang. The flow is shown in the left part of Figure 3-5.

3.6.2 Creation of the CGP for DDL

The Code Generation Program for DDL will be produced in the following way: The Code Generation logic coded in I-Lang is fed to the I-J Translator, which will output the CGP for DDL in J-Lang. Because the SAP (a component of the I-J Translator) is in K-Lang the program will be interpretively executed using the K-Interpreter. (For a description of the K-Interpreter see Section 4.5.) After the CGP in J-Lang has been produced it will be fed to the J-K Translator, which translates the CGP into K-Lang. The flow is shown in the right part of Figure 3-5.

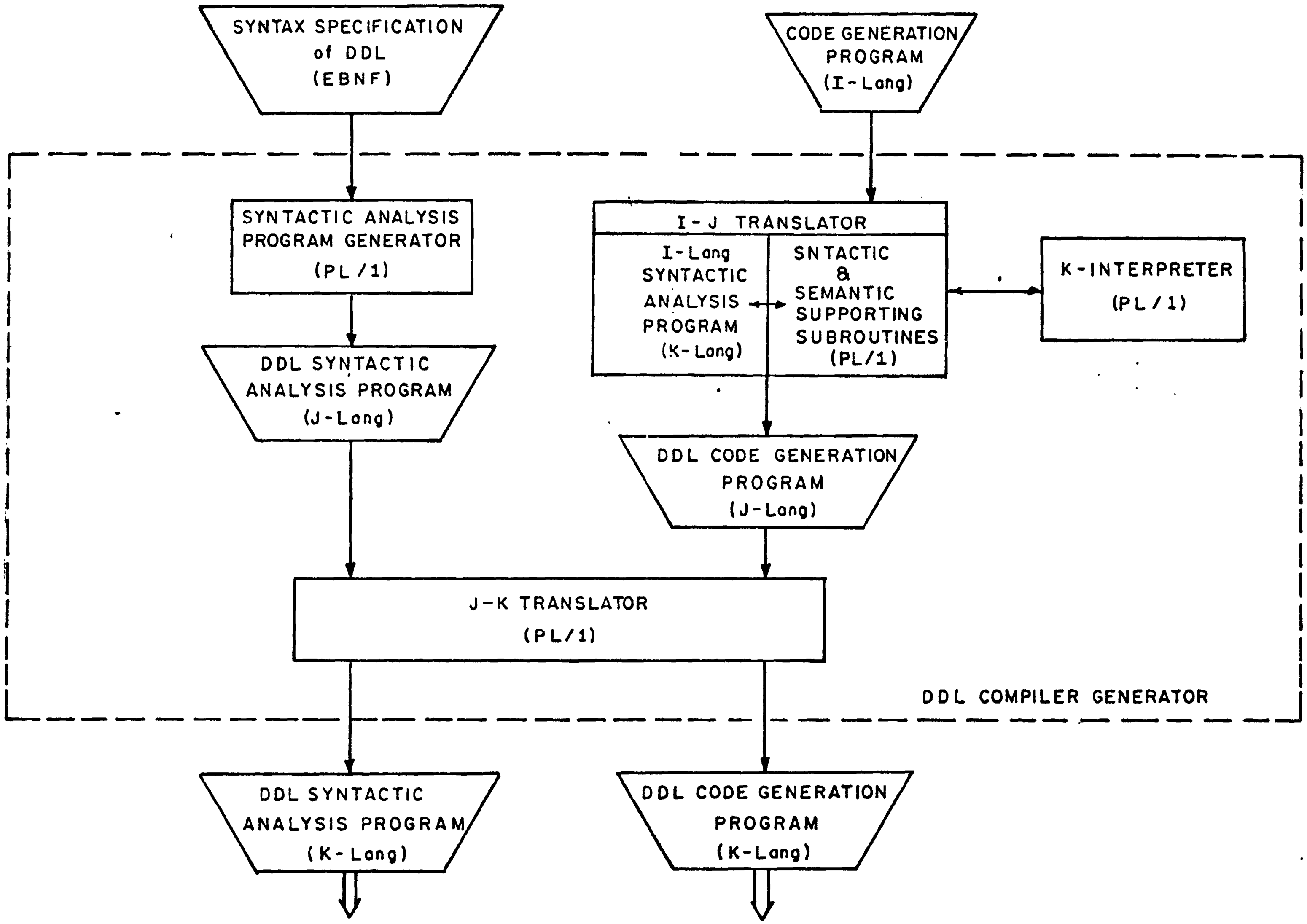

SYNTAX SPECIFICATION of DDL (EBNF)
CODE GENERATION PROGRAM (I-Lang)
SYNTACTIC ANALYSIS PROGRAM GENERATOR (PL/1)
I-J TRANSLATOR
I-Lang SYNTACTIC ANALYSIS PROGRAM (K-Lang)
SNTACTIC & SEMANTIC SUPPORTING SUBROUTINES (PL/1)
K-INTERPRETER (PL/1)
DDL SYNTACTIC ANALYSIS PROGRAM (J-Lang)
DDL CODE GENERATION PROGRAM (J-Lang)
J-K TRANSLATOR (PL/1)
DDL COMPILER GENERATOR
DDL SYNTACTIC ANALYSIS PROGRAM (K-Lang)
DDL CODE GENERATION PROGRAM (K-Lang)

Figure 3-5

DDL Compiler-Generator Logic Flow

4. THE DDL COMPILER

The DDL Compiler is generated using the DDL Compiler Generator. It is used to translate DDL source statements into a Data Conversion Program in K-Lang (the object language of the K-Machine). It does the translation using the Syntactic Analysis Program and the Code Generation Program. Since both the SAP and CGP are in K-Lang, we need a K-Interpreter to execute them. And since the code produced by CGP is in J-Lang we need to translate it to K-Lang. Therefore, the DDL compiler as a whole will consist of:

(a) The Syntactic Analysis Program (SAP)

(b) The Code Generation Program (CGP)

(c) A set of Syntactic and Semantic supporting subroutines

(d) The J-K Translator

(e) The K-Interpreter

An overview of the DDL Compiler is shown in Figure 4-1.

4.1 The Syntactic Analysis Program (SAP)

The Syntactic Analysis Program is created using the SAPG and J-K Translators. The main function of this program is to perform syntax analysis of the DDL-source statements, and the creation of the symbol and Data Tables. See Figure 4-2.

The Syntax Analysis, Phase 1 of the DDL Compiler, is controlled by the SAP. A statement will be scanned, using the subroutine LEX, which returns the next syntactic unit present in the source statement; each syntactic unit is then examined according to the logic in the SAP.

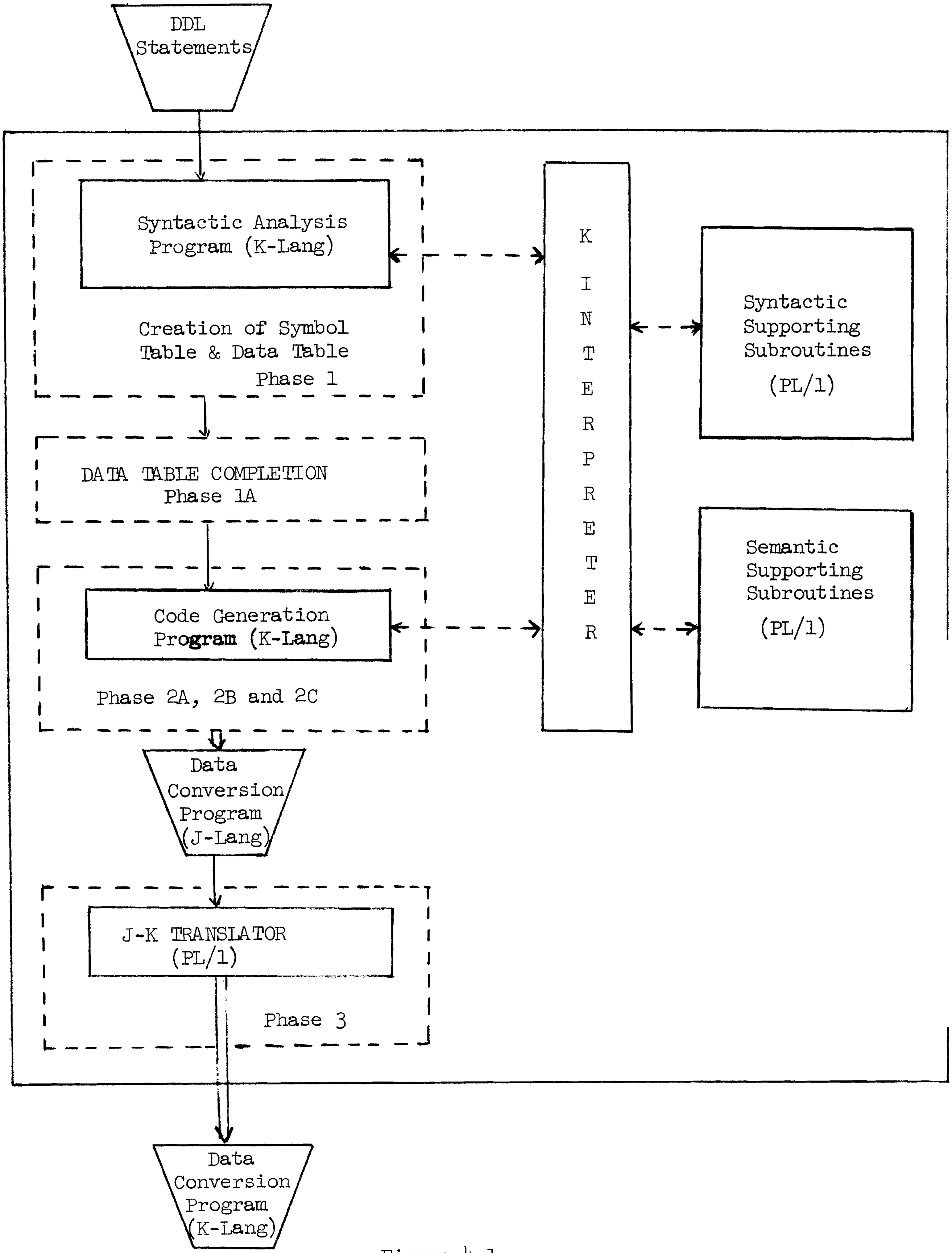

DDL Statements
Syntactic Analysis Program (K-Lang)
Creation of Symbol Table & Data Table
Phase 1
DATA TABLE COMPLETION
Phase 1A
Code Generation Program (K-Lang)
Phase 2A, 2B and 2C
Data Conversion Program (J-Lang)
J-K TRANSLATOR (PL/1)
Phase 3
Data Conversion Program (K-Lang)
K INTERPRETER
Syntactic Supporting Subroutines (PL/1)
Semantic Supporting Subroutines (PL/1)

Figure 4-1

The DDL Compiler

During this process syntactic supporting routines will be called to capture syntactic information, and to produce error diagnostics (if necessary). If a correct syntactic unit is recognized, a call to semantic supporting subroutines is made, resulting in one or more entries being made in the Symbol Table (see Appendix C) and/or Data Table (see Appendix D). Analysis of CRITEX statements will cause the output of encoded strings which are used in Phase 2 (Code Generation). Discovery of a syntax error will cause control to be returned to calling subroutines with a code describing the error and a pointer to the last Symbol Table entry used (these items will be used in producing error messages).

Section 4.1.1 and 4.1.2 describe the two main data structures which are created by the syntactic supporting subroutines during Phase 1 of compilation.

4.1.1 Symbol Table

The Symbol Table is created during the first phase of compilation. The symbol table is a doubly-chained balanced tree with filial set size ranging between 3 and 6 (see Appendix C). Memory is allocated as it is required. Each node is composed of three pointers:

1) a pointer to the key (the keys are in separate storage area);
2) a pointer to the first member of the filial set of which this node is a parent;
3) a pointer to the next member of the filial set of which this node is a member.

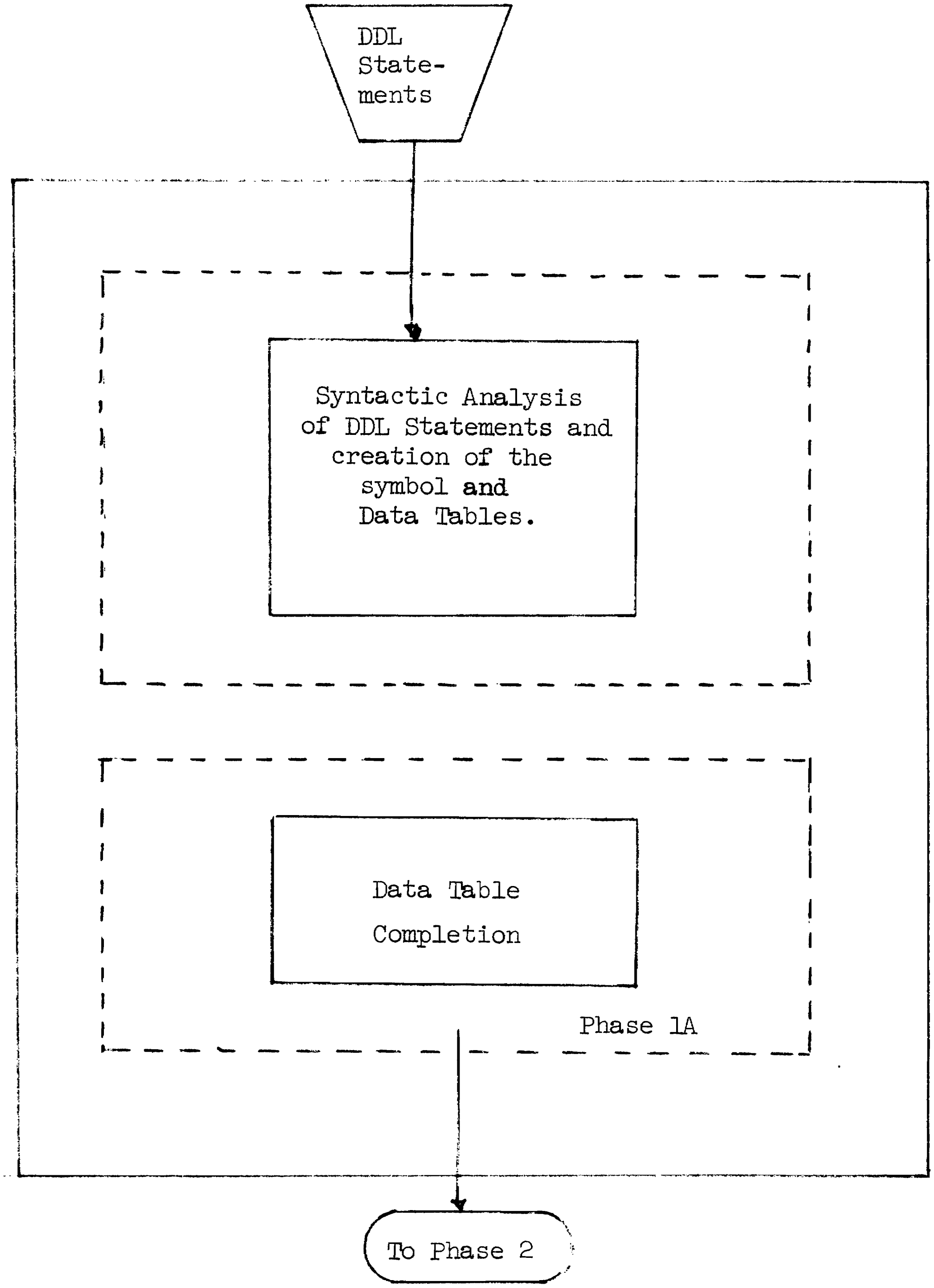


Figure 4-2

Phase 1 of the DDL Compiler

4.1.2 Data Table

The Data Table contains information which describes the "data" associated with each symbol in the Symbol Table. This "data" may be part of the source or target files, may describe one of the files, or may be a procedure which uses the file data to enhance the file descriptions (e.g., a linkage criterion statement).

Many Data Table entries point to other entries to which they are related. Some relationships are "subordinate member of a group", and "supporting description". These relationships play a very important role in determining which of several items with identical simple names is the correct reference.

4.1.3 Phase 1A - Data Table Completion

When the supporting subroutines of phase 1 create the Data Table entries (and the encoded text of the CRITEX statement), all symbolic names are converted to symbol table entry numbers. One function of phase 1A is to change the symbol table entry numbers into data table entry numbers. A second function is to insert the "father" pointers into field and group entries. A third function is validity checking on the references - for example checking to see that all members of a group are groups or fields. These three functions are done concurrently. At the end of phase 1A the symbol table (with its two index structures) is no longer needed and the space it occupies is made available for use in phase 2. The processing of phase 1A is described in more detail below.

4.1.3.1 Processing Logic

Using the Data Table entry index, the data table entries are processed sequentially. First the type of entry is determined (see Appendix D1 for the description of the encoding used). Using logic specific to the entry type, each pointer parameter is located and, using the Symbol Table (inverted) index, the symbol table data pointer entry is obtained. The Data Table entry number contained in this entry replaces the symbol table entry number as the pointer parameter. If the entry is a Field or a Group the "most recent father" entry is moved from the Symbol Table entry for this Field or Group, to the "sup" (superior) pointer of the entry. The reason for the delay in filling this field is that there may be more than one Group to which this Field or Group belongs. If this is the case and if then there is the possibility that one or more groups, which declare this Field to be a member of them, may be discovered after the Field is defined. The best strategy is to wait until all groups are known before filling in the "most recent father" pointer.

After the pointer parameters are changed they are used to access the associated data table entry and the "type" field of that entry is tested for acceptability as a parameter. For example, the pointer found as a criterion name in a Record entry must point to a data table entry which describes a criterion statement. If the pointer parameter is now pointing at a function description it is assumed that the function will return an integer, and a test is made to see if an integer is permitted. (Some typical functions are LENGTH and CNT.)

4.2 Code Generation Program (CGP)

The second phase (see Figure 4-3) of the DDL compiler has three important functions:

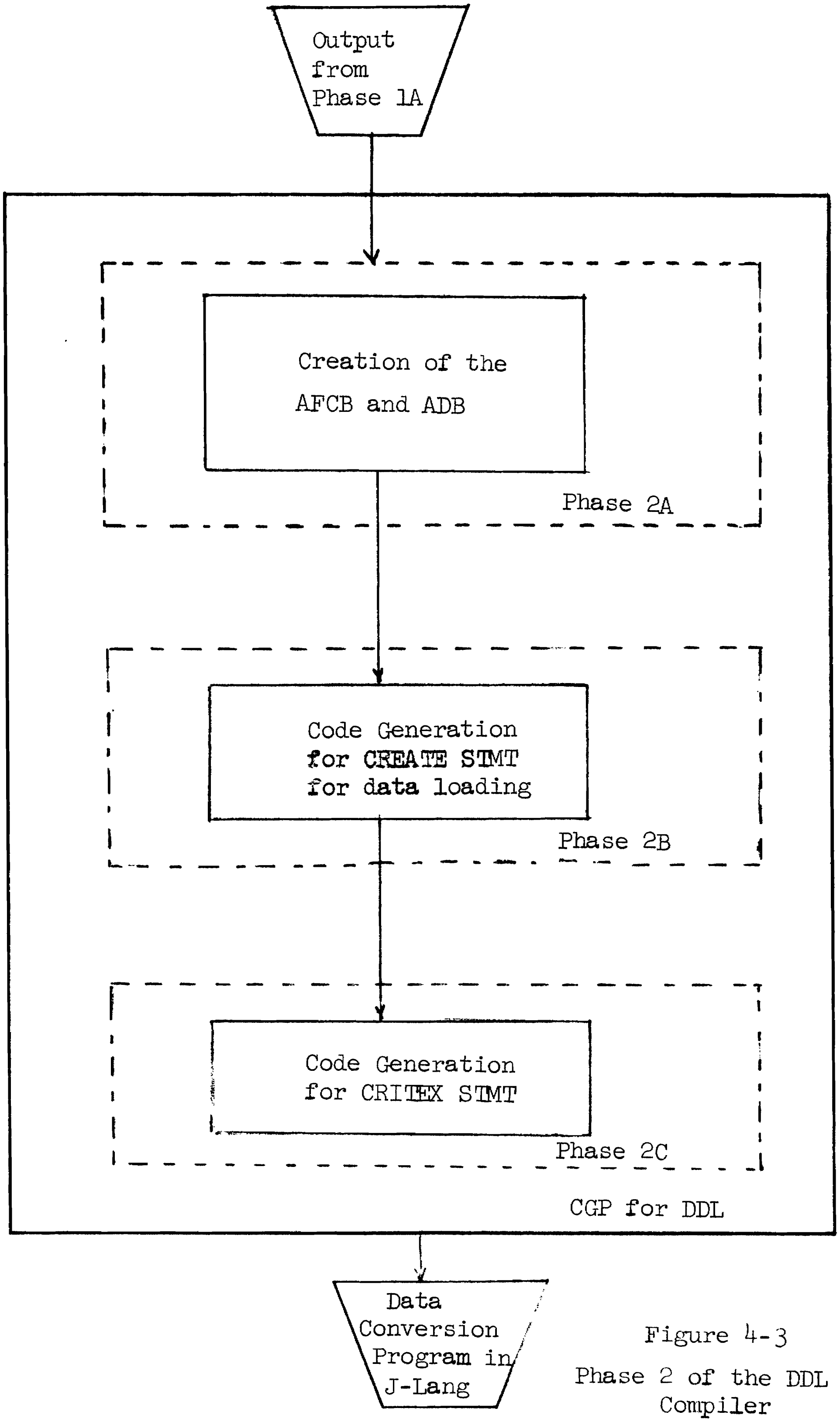


Figure 4-3
Phase 2 of the DDL
Compiler

(1) it creates the Active File Control Blocks (AFCB) and the Auxiliary Descriptor Blocks (ADB) required for execution of the CREATE statement. (Phase 2A)

(2) using the data structure created in (1) it generates J-Lang code to load data into main memory and execute the CREATE statement. (Phase 2B)

(3) it generates J-Lang code for the procedural statements of DDL (CRITEX, RPVAL). (Phase 2C)

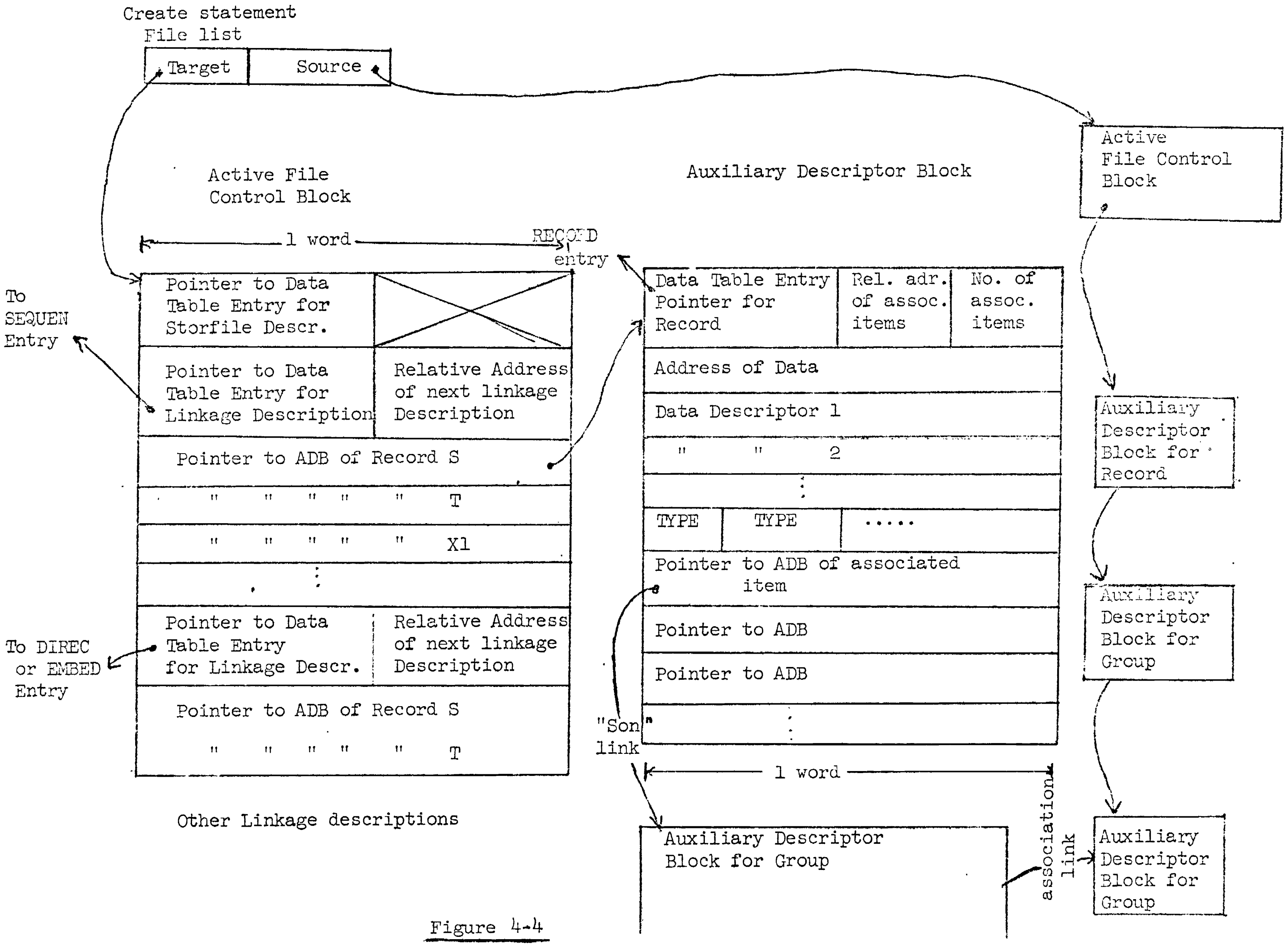
The next sections describe the implementation of these three functions.

4.2.1 The Active File Control Block (AFCB)

The format of the AFCB is shown in Figure 4-4. There is one AFCB for each file which will be used in the execution of a CREATE statement. The primary function of the AFCB is to point to records from the file which are currently in core. The number of these records is quite small - usually just the number of different record occurrences concurrently required in the evaluation of the criterion statements for the linkage specified for the file. The following paragraphs describe the fields of the AFCB.

The "pointer to Data Table Entry for the STORFILE description" is the data table entry number of the entry which contains the description of the file structure. The first "pointer to the Data Table Entry for Linkage description" contains the data table entry number for the SEQUEN statement which governs the physical order of the records in this file. Through the pointers in these linkage statements one finds the criterion expressions that are used in the creation of the linkage.

AFCB and ADB's for Create Statement



Create statement
File list
Target
Source
Active File Control Block
Auxiliary Descriptor Block
Active File Control Block
1 word
RECORD entry
Pointer to Data Table Entry for Storfile Descr.
To SEQUEN Entry
Pointer to Data Table Entry for Linkage Description
Relative Address of next linkage Description
Pointer to ADB of Record S
" " " " " T
" " " " " X1
To DIREC or EMBED Entry
Pointer to Data Table Entry for Linkage Descr.
Relative Address of next linkage Description
Pointer to ADB of Record S
" " " " " T
Other Linkage descriptions
Data Table Entry Pointer for Record
Rel. adr. of assoc. items
No. of assoc. items
Address of Data
Data Descriptor 1
" " 2
TYPE
TYPE
.....
Pointer to ADB of associated item
Pointer to ADB
Pointer to ADB
"Son" link
1 word
Auxiliary Descriptor Block for Group
association link
Auxiliary Descriptor Block for Record
Auxiliary Descriptor Block for Group
Auxiliary Descriptor Block for Group

Figure 4-4

The "relative address of next linkage description" is the address of the next "pointer to data table entry for linkage description". The address is relative to the start of the AFCB. It is necessary because the ADB pointers which follow (and will be explained next) are variable in number. The ADB pointers point to the ADB's associated with the core resident records, symbolically called S (source), T (target), X1, X2, etc. These symbolic names are those used in the CRITEX statements. They refer to different record occurrences at different times during processing.

The next group of fields of the AFCB is a repetition of the format just described. These fields point to the DIREC or EMBED statements specified by the STORFILE statement. The ADB pointers for the S, T, X, etc. records are for use in processing the CRITEX statements associated with these linkages.

4.2.2 The Auxiliary Descriptor Block (ADB)

The ADB's (see Figure 4-5) are used to hold core address of the record and the data descriptive information (e.g., length, repetition count, etc.) which varies with each occurrence of the record. This information cannot be stored in the Data Table entry because there may be two or more occurrences of the record in core concurrently, for example, source and target of records used in linkage, or multiple occurrences of a repeating group.

The "data table entry pointer" is used to find the symbolic name of the record (or group or field) and the fixed descriptive data (e.g., CONCODE statements, criterion statements, and possibly repetition number, order, etc.). For internal use, the data table entry number is the "name" of a record, group or field. Therefore when searches are being made for names

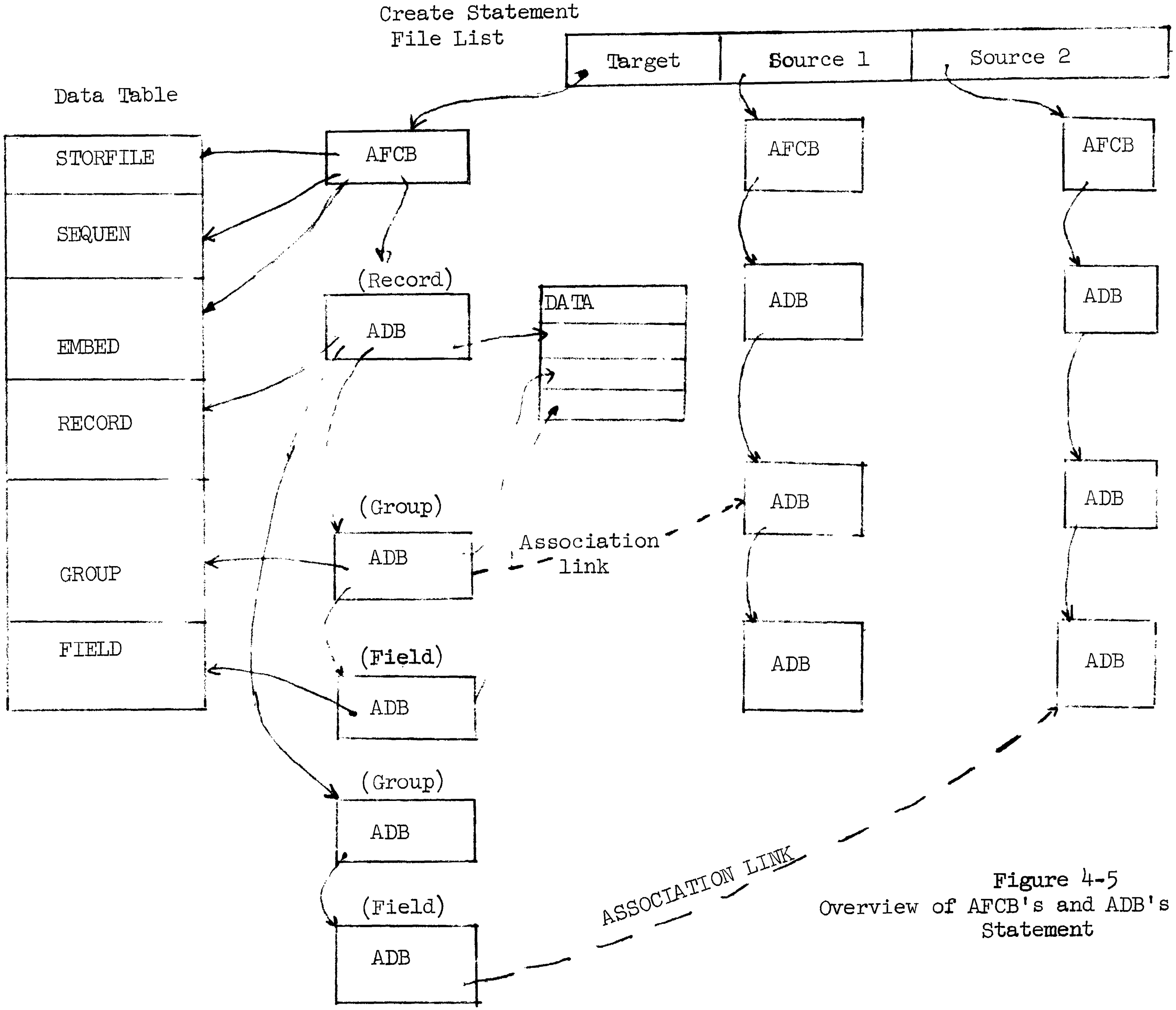


Figure 4-5
Overview of AFCB's and ADB's for Create
Statement

(for example, in the creation of the associate links described below) the "name" is present in the ADB.

The "Rel. address of Assoc. items" is the ADB relative address of the start of the "TYPE" bytes. This is necessary because the number of data descriptors is variable. The "address of data" is the core address of the start of the data described by the ADB. (Note - this is not filled in until execution time.) The data descriptors are the items which would be in the data table entry except that they vary with each occurrence of the record. In these cases the data table entry contains a pointer to the procedure which obtains the value to be placed in the data descriptor. After code-generation for the data loading code the data table will contain the number of the descriptor in the ADB where the value will be stored.

Each "TYPE" field describes the sequentially corresponding "pointer to ADB of associated item". The meaning of type is shown in the following table:

Type	Meaning
1	Pointer to "father" (the group which contains this item)
2	Pointer to "son" (subdivision of this group)
3	Pointer to next core resident member of this repeating group (or field)
4	Association link

The pointers are to ADB's.

The association link corresponds to the associate pairs in the ASSOCIATE statement specified by the CREATE statement. Using these links the code generation logic (to be described in 4.3.3) will create a procedure which will construct the target record from the source records (see Figure 4-5).

4.2.2.1 The ADB Hierarchy

This paragraph describes in more detail the procedure for determining the hierarchical structure of the ADB's. It explains when an ADB is created and how it relates to other ADB's.

The hierarchy is basically a replica of the structure of the data table structure (that is, records, groups, and fields, in descending order of scope). Beyond the basic structure are (1) ADB's for the "pseudo-variables" (data fields which result from the statements LEN, CNT and EXIST), and (2) ADB's for repeating occurrences of a group or field.

The pseudo-variables are constructed by the code generation logic; they act as normal user data fields except that the data is inserted by the data loading procedure (see Sections 4.2.3 and 5). The ADB's for pseudo-variables are linked only as a result of an associate statement. There is no hierarchical linking. Of course, there may be any number of references to the ADB, through the data table, for use in CRITEX expression evaluation.

When a field or group may repeat ADB's are created for the number of occurrences which will be resident in main memory concurrently. If the maximum number of occurrences is fixed, the ADB's are generated (and linked sequentially) at code generation time. If the number is variable, the ADB's are set up dynamically by logic in the data loading procedure.

4.2.2.2 Phase 2A Completion Logic

After the AFCB's and ADB's have been created, and the association links from the ASSOCIATE statement have been made, the phase 2 logic searches down the "tree" of ADB's for the target file, to find the fields which have no association links. These exist because some groups are associated with other groups, and the first pass of associate linkage doesn't enter the subordinate field association. Once the tree has been completed, code generation can start.

4.2.3 Phase 2B - Code Generation for CREATE Statement

Code generation consists of creating an ordered list of basic J-Lang operations and subroutine calls which will load data into main memory, move each of the source fields to the associated target field, and then link the newly created record into the file which is being created.

Code generation starts by creating a data loading procedure for each file to be used. Data Loading is the process of linking an input record to the ADB and AFCB, and then executing whatever functions are necessary to complete the ADB descriptors. The generation of this code requires getting the pointers to functions which are parameters in the Data Table Entry and generating code to call these functions and place the returned values in the ADB. Now the ADB descriptor offset replaces the function pointer for the rest of code generation. In cases of variable length, or repeating groups, the information in the CONCODE statements is used to generate code to recognize the end of data fields. The Data Loading procedure is further described in Section 5.

The code generation logic then walks the tree of ADB's of the target file, generating code to move each field from the source to the target. (To find the source it uses the association links.) The operand addresses used in the generated code are indirect through the ADB, not the direct data addresses, which are assigned at execution time. The data descriptions in the Data table entries indicate where to generate code conversion calls, length truncations, etc. In cases where the data table indicates that information will be in the ADB, the code generation logic generates code to obtain the values from the ADB descriptors at execution time. In some cases, code to move data will not be necessary because the source data is in the proper form for inclusion in the target record; all that is necessary is to set the core address pointer of the target field ADB.

4.2.4 Phase 2C - Code Generation for CRITEX

CRITEX statements will be translated using the technique of stacking operators and operands on two different stacks, and generating code when newly input operator has a precedence lower than the operator on the operator stack. To generate code the required number of operands are taken off the top of the operand stack and the operator is taken off the operator stack. The results of the operator are assigned a symbolic location and the symbol used is placed on the operand stack in place of the operands removed. Whenever code has been generated, the new top of the operator stack is tested against the recent input which caused the code generation to begin. If the operator precedence of the stack is lower than the input the code generation process is repeated. If not the input operator is stacked. Whenever the top of the operator stack contains ")" immediately followed by "(", both of these are deleted. The operator precedence is, from highest to lowest:

Operator	Comment
(	when it is the newly input operator or it is on the stack and being tested against ")"
)	
*/	
+-	
NE,LT,GT,EQ,LE,GE MFM	
NOT	
OR AND	
(	when on the stack and not being compared with ")"

4.3 Syntactic and Semantic Supporting Subroutines

As was stated before, Syntactic Analysis of Phase 1 of the compiler is basically the interpretive execution of the code generated by the SAPG (the SAP). This code requires many supporting subroutines. The most important ones are discussed below:

a) Lexical Subroutine

b) The System Vector Subroutine

c) The Syntax Recognition Failure Subroutine

d) The Semantic Supporting Routines

4.3.1 Lexical Subroutine

This subroutine reads DDL source statements and buffers them. It returns single tokens, when called. Tokens are of 4 types:

1. names (strings of characters within single quotes)
2. keywords (strings of characters not surrounded by quotes)
3. punctuation marks
4. numbers (strings of decimal digits)

Each call to .LEX returns a pointer to the next token, and the token type.

The lexical processing routine calls a .PRINT routine to print the source line, if this option has been specified. (Error messages are produced by the various syntactic processing "failure" routines. These routines call .PRINT directly.)

4.3.2 The System Vector Subroutine

There are many places in the logic of syntactic analysis where one routine needs to have control over the succeeding sequence of routines. In particular, this occurs when the succeeding routines will process a list of parameters, which must occur in fixed order. The subroutines described below help provide this sequencing facility.

A "subroutine vector stack" (a stack of subroutine addresses) is used. By calling .SETV, addresses are pushed on the stack. (The parameters of the .SETV call are the addresses, in the order in which the subroutines are to execute.) Each call to .VCALL causes control to be given to a subroutine whose starting address is popped off the vector stack. It is a CALL .VCALL which is generated by the SAPG when it detects // in the EBNF source statements.

4.3.3 The Syntax Recognition Failure Subroutine

The logic for syntax recognizer "failure" exits also requires the type of sequence control described in the preceding paragraph. The stack used here is called the "fail vector stack". It is set by a call to .SETF, the parameters, as before, are the addresses to be stacked; the order is the order in which they are to be executed. When .FAIL is called, it gives control to a subroutine whose address is popped off the fail vector stack. To pop the stack without executing any subroutine, .POPF is called. The SAPG automatically generates calls to .POPF; they are executed after each syntactic item is correctly recognized. The execution of the .POPF after each syntactic item means that when a list of failure exits is set up, it specifies an address for each syntactic item including terminal symbols like ",", ")", etc. For example for the string

< name >: (< part 1 >)

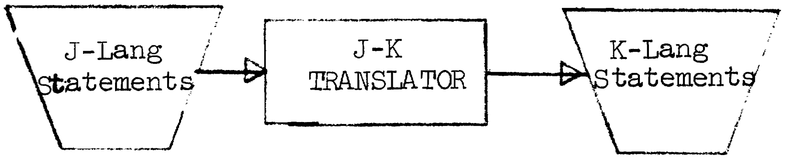
there would be 5 failure exits specified.

4.3.4 The Semantic Supporting Routines

The Semantic supporting routines are called by CGP and are used to create the AFCB's, ADB's which help in the code generation phase.

4.4 The J-K Translator

As mentioned in the introduction of Section 4, the Code Generation Program (CGP) produces J-Lang statements; to translate the J-Lang into K-Lang (the object language of the K-Machine), a J-K Translator is needed. A complete description of it is given in Section 8, after the descriptions of the J- and K-Languages. The overview of the J-K Translator is shown in the figure below.



4.5 The K-Interpreter

A machine language interpreter is a computer program that performs the instructions of another program, where the other program is written in some machine-like language. A machine-like language is a method of representing instructions using operation codes, address, etc.

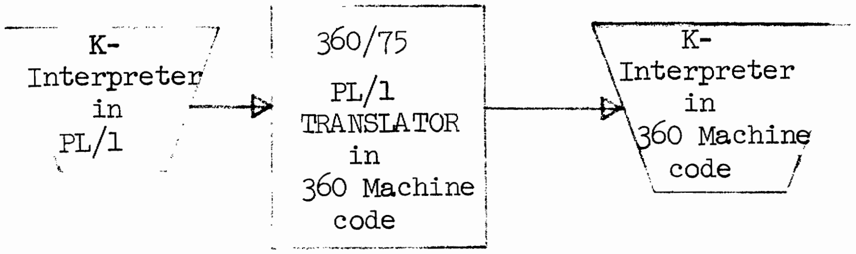
Machine language interpreters are used chiefly for the following purposes:

1) to allow the representation of a fairly complicated sequence of decisions and actions in a compact, efficient manner without the need to construct the physical machine (hardware) which executes these instructions.

2) to communicate between passes of a multiple pass program (i.e., in a multipass program, the earlier passes must transmit information to the latter passes). This information is often transmitted most efficiently in a machine-like language, as a set of instructions for the later pass. This philosophy of multipass operation may be characterized as "telling" the later pass what to do, rather than simply presenting it with facts and asking it to "figure out" what to do.

The interpretive technique has the further advantage of being relatively machine-independent - only the interpreter must be rewritten when changing machines. Furthermore, helpful debugging aids can readily be built into an interpretive system. Finally, an interpreter can usually be written so that the amount of time spent in interpretation of the code itself and branching to the appropriate routine is negligible.

In this case the machine-like language is the K-Lang, and the K-Interpreter is a computer program written in PL/1, and then translated (using the PL/1 compiler) into 360 machine code. An overview of the translation is shown in the figure below.



5. THE DATA CONVERSION PROCESSOR

The Data Conversion Processor is a set of programs (in K-Lang) and data produced by the DDL compiler. At execution time the programs use the data (a network of descriptor blocks*) to read and parse the input data and construct the output data in the desired format.

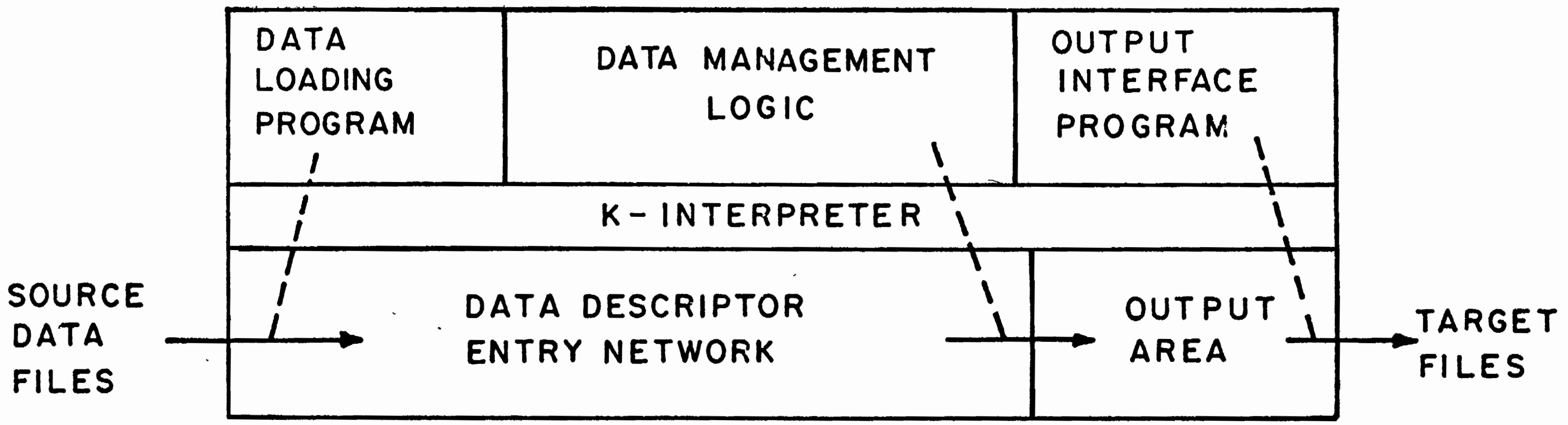
The K-lang programs include: (1) a "data loading" subroutine which brings data into main memory; (2) a data movement subroutine which reformats the data; (3) an output subroutine to write the data to the auxiliary storage device. These programs are described below. The use of supporting subroutines (written in PL/1) is indicated in the program descriptions. Section 5.4 completes the description of the supporting subroutines. An overview of the Data Conversion Process is shown in Figure 5-1.

5.1 Loading Data

Whenever a new record from one of the input files is required, two operations must be performed. First the record must be brought into core from secondary storage. This is done by calling the "Read" Supporting Subroutine. Second the "Data Loading Procedure" is executed. This procedure is generated during phase 2 (see Section 4.2.3). The execution of it causes the core addresses of the various data elements (fields, etc.) to be placed in the ADB's associated with them. To do this may require the execution of subroutines which locate the end of a field, using the information supplied in the CONCODE statement. As the addresses of data items are resolved, code to complete the descriptor items is executed. This code may move the data from one field to the descriptor of another

* A detailed description of these descriptor blocks is given in Section 4.2.

THE DATA CONVERSION PROCESSOR



Solid lines indicate data flow; broken lines indicate Control.

Figure 5-1

(e.g., in the case where a symbolic field name had been specified as the length of another field); the count of a repeating field (which was calculated by the 1st part of data loading logic) might be used to fill-in a descriptor entry (for example, if the function CNT had been used in the DDL source).

If the ADB indicates that an item repeats a variable number of times, the data loading code will dynamically set up new ADB's if the existing number is not sufficient. Item lengths, association links, and the memory addresses of the data are placed in the new ADB's. When the loading of an item with a variable repetition factor has been completed, the ADB's in the list which were not used are released through a function call. (The list of ADB's for a repeating item contains only one ADB at the start of execution. It grows and shrinks then to fit the number of repetitions for the various occurrences of the group.)

5.2 Target Data Space Allocation and Data Movement

After the required source records have been loaded (as described in the preceding paragraph), space is allocated for those variable length fields of the target record which, because of length, or code conversions, cannot be copied directly from the source field. In cases where space is allocated, the address is entered in the ADB.

At this point execution can start in the data movement routine, which was generated by phase 2B of the compiler. Recall that the Data Movement program was generated by walking across the lowest level of the ADB tree (the fields), generating code to move the address of the source field from the ADB of the source, into the ADB of the target (when no data conversion is required) or generating code to call the proper field length or character conversion function, specifying that the results are to be placed in the newly allocated target data area.

Where conditional selection of source data was specified, through the use of CRITERION statements, the decision logic is also generated by Phase 2B. This decision logic may be a simple test coded in-line before the data movement, or it may be a closed subroutine, with the call to it and a test for the outcome being placed ahead of the data movement code.

5.3 Data Output

After execution of the Data Movement program, the ADB's of the fields in the target record are pointing to the data which is ready for output.

Now a call is made to the "Record Linkage" routine associated with the file(s) to which this record belongs. In the current implementation, where we are supporting only sequential files, the "Linkage" routine will be a standard one, used to output sequential files via the data access methods of the 360.

(In later versions, where intra-file, and inter-file record linkages are supported, a Linkage routine will be generated specifically for each storage record type, based on information specified by the LINK statements in the DDL source program.)

The Record Linkage routine creates the proper links, and then assembles the fields of the record (remember, until now the only association of the fields was through the pointers in the ADB.) In most cases assembling the record will just mean concatenation of the fields; but, in cases where the record is longer than the host operating system permits (this may mean longer than the physical records permitted on the auxiliary storage device) the Record Linkage routine will segment the record.

In any case the final action is the calling of the "Submonitor" I/O routine for the auxiliary storage device specified by the DDL source statements. These I/O routines are described in the next section.

5.4 Data Conversion Processor - Supporting Subroutines

The Supporting subroutines of the Data Conversion Processor are divided into two groups: (1) the Submonitor, which interfaces with the operating system of the host machine, and (2) the utility functions used in data movement.

5.4.1 The Submonitor

The Submonitor is designed to maintain operating system independence, and to provide facilities often not available from host operating systems. Among the many subroutines which form the submonitor there are the following:

(1) Main Memory Management - gets (from the host operating system) and manages main memory space. "GET" and "FREE" main memory commands supported.

(2) Auxiliary Storage I/O - writes to and reads from auxiliary storage devices. READ, WRITE commands are supported.

(3) Print Function - Prints main memory, or auxiliary storage areas, in binary, character, hexidecimal, octal, etc. formats.

The utility functions used in data movement include:

(1) Character Set Conversion - will convert a field from one character set to any other. Both character sets are specified as variable length, one dimensional arrays. The result will be placed in the location specified as a parameter. The length of the source and target field need not be the same; a parameter specifies the pad character to be used if necessary.

(2) Field Movement - a special case of the character conversion routine, where the character set is the same for both source and destination fields.

(3) Data Type Conversion - the source and destination data types are specified as one of the following: FIXED POINT (a one parameter gives the number of digits, another gives the radix) FLOATING POINT (one parameter gives the number of digits in the characteristic, another gives the number of digits in the mantissa, a third gives the radix of the numbers) CHARACTER - same specification as used in character set conversion, including length.

6. I-Lang

This chapter and the next two describe the logical machine which is used in the implementation of the Data Conversion Processor, and much of the DDL compiler. The machine (we shall call it the K-Machine) is expressible in any of three languages (I, J or K-Lang). It is used in the following ways within the DDL Processor System:

(1) to perform the syntactic analysis of DDL during the first phase of compilation,

(2) to drive the code generation process (mainly at the subroutine call level),

(3) to execute almost all of the Data Conversion Processor,

(4) to perform the syntactic analysis during conversion of I to J language*.

Each language is designed for a particular use. I-Lang is a high-level programming language, in many ways similar to PL/1. In the present implementation it is used by programmers to encode the code generation logic of the DDL compiler, but its use can be expanded to include most of the hand-coded programs of the DDL Processor System.

J-Lang is an encoded form of I-Lang. It is a language designed to be produced and used by computer programs, rather than by humans. For this reason, numbers, not symbols, are used to represent the operation codes and operands of the J-Lang instructions. The Data Conversion processor as produced by the DDL compiler is represented in J-Lang. The Syntactic Analysis Program is in J-Lang, in the original form produced by the Syntactic Analysis Program Generator.

* It can do this bootstrapping operation because the syntax analysis is generated directly in J-Lang, by a special purpose program generator called the Syntactic Analysis Program Generator.

K-Lang is the lowest level language - it is the only one which is directly interpreted and executed. The others are interpreted and translated into the next lower level language; thus all I and J-Lang is eventually translated into K-Lang for execution. Because the K-Lang is to be executed, there is a "K-Machine" (a simulated computer) within each component of the DDL Processor. The K-machine, like actual hardware-implemented machines, has an op-code interpreter, address calculation logic, diagnostic trace facilities, and a provision for automatic transfer of control on special ("interrupt") conditions. The K-machine, and the languages used to express the logic to be executed by it are described in this and the following two chapters.

6.1 Where I-Lang Is Used

I-LANG is the language in which the semantic/code generating subroutines will be written. I-Lang statements are input to the I-J translator, which produces the equivalent J-Lang code. (The J-Lang code is then translated into K-Lang, the form in which it can be executed.)

6.2 Statements

6.2.1 PROCEDURE Statement

The format is:

label: PROCEDURE [(parameter[,parameter]...)];

The optional list of parameters serves to equate names used in the procedure with names defined outside the procedure. (The passing of parameters is discussed in the Section 8.4.) A procedure block is started by a PROCEDURE statement and terminated by a matching END statement. The block is treated as a subroutine during execution. Nested procedure definitions are not permitted, but nested subroutine calls are.

6.2.2 IF Statement

The format is:

[label:] IF conditional expression THEN part 1
    [ELSE part 2]

The conditional expression must be a simple logical expression (one logical operator*, two operands), optionally followed by a "data limit" expression; or a function call followed by one of the logical operators (or a symbol equated to one of the logical operators). A data limit expression is either ",LEN-B (symbolic address expression)" or ",LEN-C (symbolic address expression)" or ",DELIM 'punctuation mark'". LEN-B is binary length. LEN-C is character length. DELIM is field delimiter. They determine the length of both operands. If the data limit field is absent, it is assumed to be that given in the DECLARE statement for the first operand.

"part 1" and "part 2" may be IF statements, GOTO statements, assignment statements, or DO groups. If the conditional expression is true, "part 1" is executed. If it is not "part 2" is executed (if it is present).

6.2.3 Assignment Statement

The format is:

[label:] A=B [,data limit expression];†

or [label:] A=B arithmetic operator C [,data limit expression];

A, B, C are names of data items (i.e. constants, symbolic names, functions, etc. See Section 6.2.11). The arithmetic operators are +, -, /, *. The execution of this statement causes the value of the expression on the right side of the equal sign to replace the value of the data item on the left side. See 6.2.2 for an explanation of the data limit expression.

* These are EQ, NE, GT, LT, LE, GE. See Section 6.3 for more details.

† Square brackets surround optional parts.

6.2.4 GOTO Statement

The format is:

```
[label 1:]  GOTO label;
```

"Label" may be a statement label or an address expression (i.e. may contain indexed names and function calls). See Section 6.2.11 for a description of address expressions.

6.2.5 DO Statement

The format is:

```
DO;
```

The group formed by DO and its matching END statement can be used as one of the "parts" within an IF statement. Within the DO group any statement (except a PROCEDURE statement) is permitted.

6.2.6 END Statement

The format is:

```
[label:]  END;
```

The END statement closes a Procedure group or a DO group. The END statements will be counted by the I-J translator and must balance.

6.2.7 CALL Statement

The format is:

```
[label:]  CALL label_1[(parameter[,parameter]...)];
```

"Label_1" must be the label of a Procedure group. Execution of the statement causes a subroutine call to be made to the procedure "label_1". The optional list of parameters is, if present, converted to effective addresses; using the algorithm for address expression evaluation (section 6.2.11). The addresses are then passed to the called routine.

6.2.8 RETURN Statement

The format is:

RETURN [condition] [,value];

"Value" and "condition" may be any data item names. Execution of the statement causes a return from subroutine. If the subroutine was invoked as a function call, the "value" address expression is evaluated and the data at the effective address is returned. The address expression "condition" is also evaluated and the data at its effective address determines the setting of the condition codes (GT, LT, etc.) which are used in the IF statement logic (see Section 6.2.2).

6.2.9 DECLARE Statement

The format is:

DECLARE label $\left\{\begin{array}{l}\text{BINARY}\\ \text{CHARACTER}\end{array}\right\}$ (length) [BASED];‡

or DECLARE label DELIM 'punctuation mark';

This statement is used to define the length of internal storage areas. "length" is a simple name (no indexing, no function parameters). A literal specifies an integer-constant length. A symbolic name specifies that the length of the variable will be the contents of the location (word) specified by this symbolic name. It indicates the length of the field in bits (if BINARY) or bytes (if CHARACTER). "DELIM" indicates the field is CHARACTER and of varying length, always ending with the occurrence of the "punctuation mark" - a single character. The optional "BASED" indicates no space is to be allocated for this variable at compilation time. (The variable must then always be written with a "base" i.e. index field which will contain the base address.) Variable length fields must be and are assumed to be BASED whether or not so specified. The user must allocate variable length fields himself. (See Section 7.4

‡ Braces indicate exactly 1 of the items must appear.

for a description of memory allocation.) The purpose of a BASED declaration is to specify the length attribute of the symbol. Since during code generation, the length attribute of the first symbol in an address expression is taken to be the length attribute of the whole operand, the BASED declaration can be used to obtain the desired length attribute for operands expressed in base-displacement form. For example the operand "LABEL (POINTER)" could be made to have a length attribute of 17 characters if LABEL were declared with the line "DECLARE LABEL CHARACTER ('17') BASED;". Length attributes need not be restricted to constants, the DELIM or symbolic name option may be used.

6.2.10 Function Calls

Function calls (and pseudo-variables, used synonomously here) are expressions of the form:

```
.name [(parameter[,parameter]....)]
```

The symbolic "name" must begin with a period, and must be the label of a procedure. The parameters are treated the same as those of the CALL statement (section 6.2.7). The return of values and conditions is discussed in the section on RETURN (section 6.2.8).

Because parameters may be function calls, function calls can be nested.

6.2.11 Data Items and Naming Conventions and Definitions

Data Items are the data that are obtained using the effective address of address expressions. Symbolic names are one kind of address which may be used in address expressions. They must start with a non-numeric character, and be less than 18 characters long. The other kind of name is a Literal - these names represent themselves. Data types are the forms in which the data items are stored. Because the number of data types is small, naming conventions identify the type of all data items. The following table lists the naming conventions.

Name	Data Type	Data Type Code Number
.name	Function	1
$Iname	Internal storage*	2
$Sname	Pushdown stack	3
$Lname	List (FIFO list)	4
'data'	Literal	5
decimal number**	immediate address	Not used

* This is the default type - used if no special prefix characters are used on a symbol. Only internal names must be declared (using DECLARE statement).

** Maximum value 64K.

Internal Storage is the work area used by the program. Stacks and lists are managed by the run time system. The names $Lname and $Sname are like built-in function calls to the run-time system subroutines. There are an unlimited number of these; each with a unique user defined "name" portion. Literals and Immediate Addresses are an exception to the rule about names being addresses of data. Literals are the data, the address of the literal is substituted for the name during I-J translation. From then on the literal is treated as any other symbolic operand. Literals are assumed to be character, unless preceded by "B" which indicates they are decimal values to be converted to binary and stored as binary words.

Immediate addresses are the decimal addresses (or parts of addresses) and are passed unmodified to the address generation logic of the translators.

6.2.12 Address Expressions

An address expression is used wherever a data item is required. Address expressions are names (any type) optionally indexed by another name or address expression. The evaluation of address expressions proceeds left to right; a value within parenthesis (an "index") is an indirect address. All values are summed to form the effective address. Plus signs (+) may be inserted between names. Examples are:

A(B(C)(D))(E) which is equivalent to

A + contents of [B + contents of C + contents of D] + contents of E

4 ((A)(B))(C) is equivalent to

4 + contents of [contents of A + contents of B] + contents of C

.LEN(A)+.TYPE($S1) is equivalent to

Length of A + value of data type of item on top of stack 1

6.2.13 Equating Values and Names

The I-Lang translator will perform simple substitution for names. To request this one writes

```
name::=name_1;
```

"Name_1" may be any type (including a literal); it may also be blank, in which case "name_1" is deleted from the source. Deletion can be used to remove "noise" words.

Example:

```
FOUND::=GT;;
EXITS::= ;;
IF.LOOKTAB($S1) EXITS FOUND THEN GOTO LABEL1;
```

The above statements are equivalent to:

```
IF.LOOKTAB($S1)  GT THEN GOTO LABEL1;
```

6.2.14 Procedures in Other Languages

It is expected that functions will not be written in I-Lang. To facilitate linkage from I-Lang procedures to procedures written in another language (PL/1 for our system) the following statements must be coded for each I-Lang procedure.

```
DECLARE  label EXTERNAL PROCEDURE [(param[,param]....)] PL/1;
```

6.2.15 Comments in I-Lang Statements

Comments are permitted anywhere, blanks are permitted in I-Lang statements (that is, between syntactic units). Comments start with "/*" and end with the first "*/" following the start of the comment.

Note: The source statement listing produced, contains the comments and all the original text, before any substitutions caused by the Equate Pseudo-op.

6.3 EBNF Description of I-Lang

```
< i-lang-prog > ::= < proc-stmt > < prog-body >

< prog-body > ::= < stmt > [ < stmt > ]* < end-stmt >

< stmt > ::= < assgn-stmt > | < goto-stmt > | < call-stmt > | < ret-stmt >
             | < if-stmt > | < declare-stmt >

< proc-stmt > ::= < label > PROC [ < param-list > ];

< param-list > ::= ( < adr-exp > [, < adr-exp > ]* )

< if-stmt > ::= [ < label > ] IF < cond-exp > THEN < phrase >
                [ ELSE < phrase > ]

< phrase > ::= < stmt > | < do-group >

< do-group > ::= DO; < prog-body >

< cond-exp > ::= < log-exp > | < f-call > < rel-op >

< log-exp > ::= < adr-exp > < log-op > < adr-exp > [ < data limit > ]
```

test the results, the following I-Lang statements might be used:

```
        FOUND ::= 'EQ'
        IF .LOOKUP(NAME,TABLE) FOUND THEN
             PTR = $STACK; /*GET ENTRY POINTER OFF STACK*/
        ELSE GOTO ERROR;
```

The $STACK is the "main processor stack".

(See Section 8.4 for a description of the passing of parameters and returning of values.)

An alternative method of using the LOOKUP program is:

```
PTR = .LOOKUP(NAME,TABLE); /*GET ENTRY PTR*/
IF PTR=NULL THEN GOTO ERROR;
```

The LOOKUP program is shown below.

```
     LOOKUP:  PROCEDURE(NAMEPTR,TABLEPTR); /*THIS PROGRAM SEARCHES A TABLE
                                             OF NAMES AND RETURNS A POINTER
                                             TO THE FOUND ENTRY OR A 'NOT FOUND'
                                             INDICATOR*/
(1)*                         TYPE1 ::='80'; /*TABLE TYPES*/
(2)                          TYPE2 ::='81';
                             /*FORMAT OF TABLES --- ENTRY OFFSETS SHOWN*/
(3)                          TYPE ::= 0;
(4)                          T1NEXTPTR ::= 4; /*NEXT ENTRY POINTER*/
(5)                          T1NAMELEN ::= 6; /*LENGTH OF NAME*/
(6)                          T1NAME ::= 8; /*NAME*/
(7)                          T2NEXTPTR ::= 6; /*NEXT ENTRY POINTER*/
(8)                          T2NAMEPTR ::= 8; /*POINTER TO NAME*/
                             /*FORMAT OF T2 NAME FIELD*/
(9)                          T2NAMELEN ::= 0;
```

* The line numbers shown here are not part of the I-Lang; they are used to simplify reference in the text below.

```
< data limit > ::= ,LEN-B (< symb >) | ,LEN-C (< symb >) | ,DELIM
                   '< alpha-numx >'
< adr-exp > ::= < symb > [ + < symb > ]* [(adr-exp)]
< log-op > ::= EQ | NE | GT | LT | LE | GE
< symb > ::= < nliteral > | < ident > | < f-call > | < literal >
< ident > ::= < alpha > [< alpha-num >]*
< nliteral > ::= < num > [< num >]*
< literal > ::= '< alpha-numx > [< alpha-numx > ]*'
< assgn-stmt > ::= [ < label > ] < adr-exp > = < adr-exp > [< arith-op >
                   < adr-exp >] [< data limit >];
< arith-op > ::= + | - | * | /
< goto-stmt > ::= [< label >] GOTO < adr-exp >;
< f-call > ::= . < ident > [< param-list >]
< call-stmt > ::= [ < label > ] CALL < f-call >;
< declare-stmt > ::= DECLARE < ident > BINARY (< num >
                     [< num >] [< num > ]);
                     | DECLARE < ident > CHARACTER (< num >
                     [< num >] [< num >]);
                     | DECLARE < ident > DELIM '< alpha-numx >';
< ret-stmt > ::= [ < label > ] RETURN [ < adr-exp > ] [, < adr-exp >];
< end-stmt > ::= [ < label > ] END;
< label > ::= < ident >:
< alpha > ::= A | B | C | .... | Z | _ | # | & | $
< alpha-num > ::= < alpha > | < num >
< num > ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
< alpha-numx > ::= < alpha-num > | , | ; | : | . | ? | ! | ' | ) | (
                   | space | * | @ | ¢
```

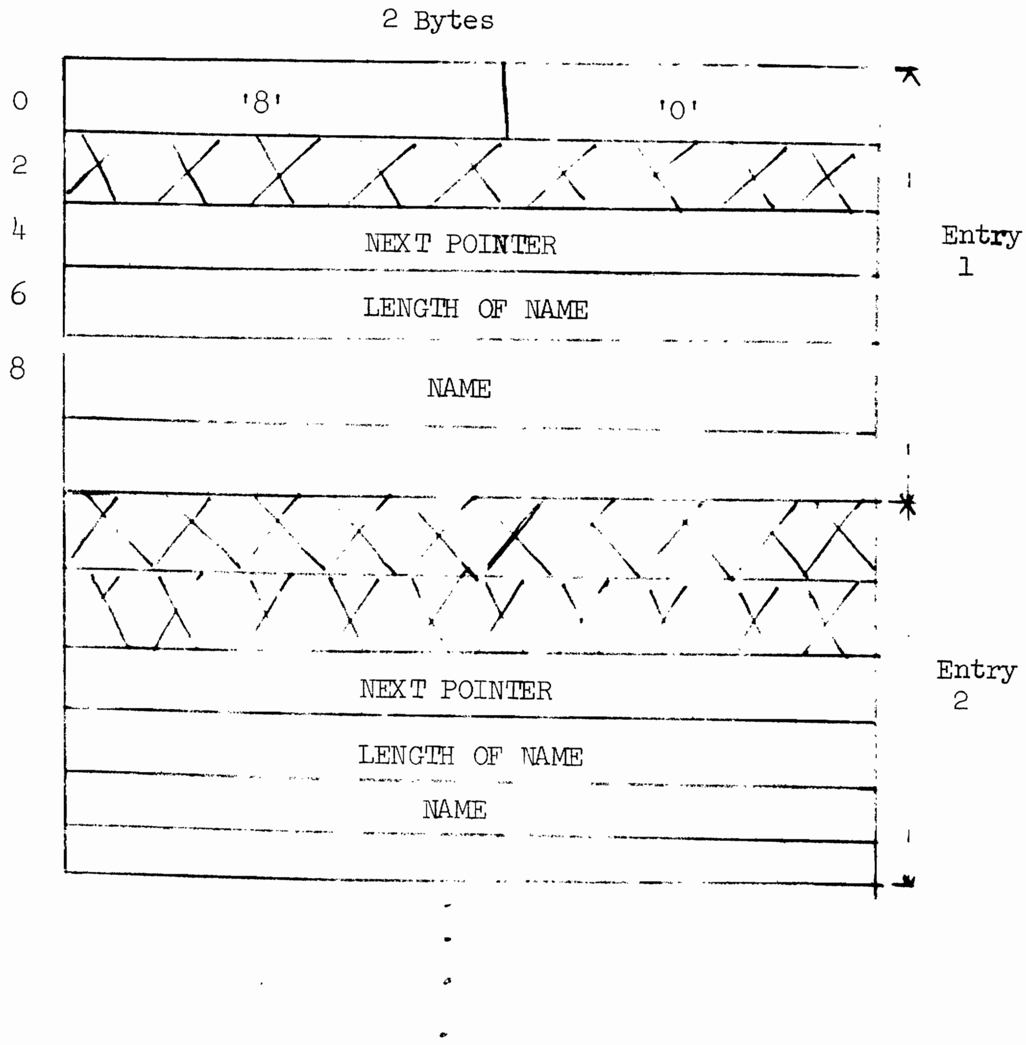
Notes: 1. The square brackets "[" "]" have been used in the above description instead of the double quotes. This was done to enhance readability.

2. No Subroutine calls are shown above, but they would be included in the EBNF used in I-J translator generation.

6.4 I-Lang Sample Program

The I-Lang program shown below searches either of two types of name tables. The format of the tables are as follows:

Type 1 Table



```
(10)                            T2NAME ::= 2; /*NAME*/
                                         /* GENERAL SUBSTITUTIONS*/
(11)                               = ::= EQ; /*EQUAL SIGN*/
(12)                            NULL ::= '0';
(13)                            FOUND ::= EQ;
(14)                            NOT FOUND ::= NE;
             /*DECLARATIONS*/
(15)                            DECLARE $ILEN BINARY ('16');
(16)                            DECLARE $IPTR BINARY ('16');
(17)                            DECLARE TYPE CHARACTER ('2') BASED;
                                   /*THE LAST STMT DECLARED
                                     THE LENGTH OF TYPE, NO SPACE WILL
                                     BE RESERVED*/
(18)                            DECLARE LOCATION BINARY ('16');
(19)                            DECLARE LOOPX BINARY ('16');
                                             /*SUBROUTINE LINKAGE WILL
                                               SET TABLEPTR AND NAME*/
(20)          $STEMP=TABLEPTR;  /*STACK TABLEPTR*/
(21)          IF TYPE(TABLEPTR) = TYPE2 THEN GOTO T2;
              /* OTHERWISE DO TYPE1 - SEQUENTIAL TABLE*/
(22)                         LOOPX = .ADR(LOOP1);
(23)  LOOP1:  $ILEN = T1NAMELEN(TABLEPTR); /* GET LENGTH*/
(24)          IF (NAMEPTR), LEN-C ($ILEN) = T1NAME (TABLEPTR) THEN
(25)      GOOD:  DO; LOCATION = TABLEPTR; /*SETUP FOR RETURN*/
(26)             RETURN 'FOUND', LOCATION;
(27)             END;
(28)             ELSE
(29)      TABLEPTR, LENB('16') = T1NEXTPTR (TABLEPTR); /*STEP TO NEXT ENTRY*/
```

```
(30)  TEST:  IF TABLEPTR, LEN-B('16') = NULL THEN
(31)           RETURN 'NOT FOUND';
(32)         ELSE GOTO (LOOPX); /*LOOPX CONTAINS AN ADDRESS*/
(33)     T2: LOOPX = .ADR(LOOP2); /*SET UP RESTART ADDRESS*/
(34) LOOP2:  $ILEN = T2NAMELEN (T2NAMEPTR(TABLEPTR)); /*GET LENGTH OF NAME*/
(35)         IF (NAMEPTR), LEN-C($ILEN) = T2NAME(T2NAMEPTR (TABLEPTR)) THEN
(36)              GOTO GOOD;
(37)         ELSE
(38)              TABLEPTR,LEN-B('16') = T2NEXTPTR (TABLEPTR); /*STEP TO NEXT
                       ENTRY*/
(39)              GOTO TEST;
             END;  /*END OF PROGRAM*/
```

Note: 1. Line 20 shows the use of a stack; note that the stack need not be declared.

2. No data limit need be specified when using the symbolic name TYPE, because its length was declared (see lines 17 and 21).

3. LOCATION is by default an internal storage variable (see line 25).

4. The RETURN (line 26) sets the condition code to 'EQ' and pushes the address of the Data Item LOCATION onto the system stack (see the Subroutine linkage description in 8.4). The RETURN on line 31 does not return any value.

5. The statement on line 22 moves the address of the instruction LOOP1 to the location LOOPX.

6. Lines 34 and 35 show the use of multilevel indexed and indirect addressing. For line 35, the pointer TABLEPTR points to the base of an area, in which at offset T2NAMEPTR, there is a pointer to an area. In this latter area, at offset T2NAME there is the data to be tested.

6.5 I-J Translator Implementation

The I-J translator is used to translate I-Lang into J-Lang. Its implementation is composed of three parts:

(1) syntactic processing

(2) text encoding and

(3) identifier table creation.

Syntactic processing parses the text, and generates any error messages. Text substitutions specified by the Equate pseudo-op are made by the Syntactic analysis routine before the analysis is attempted; substitution is based on a comparison of each lexical unit* with all the items in a "replacement pair list". The replacement pair list is actually a set of threaded, ordered lists - one list for each symbol length. The location of an entry in the lists is based on the length and collating value of the characters of the symbol to be replaced. An exception to this placement rule is made for function names (symbols starting with a period) - these are placed on a separate, single list - the use of which will be explained shortly. Text encoding is the set of subroutines called when a syntactic unit has been identified. Symbols are then encoded using the algorithm described in Appendix C. Literals are not encoded because the allowable character set is larger than the set which can be encoded. Symbols are placed in a tree structured symbol table.

* A lexical unit is a string of characters, ending in a blank, or punctuation mark. The punctuation marks are ":", ";", ",", ")", "(", "+".

This symbol table has the same structure as the one used in phase 1 of the DDL compiler, except that the "pointer to data table" (see Appendix D) is replaced by an index into an "identifier table". (See Section 7.1 for a description of the identifier table.) The index into the identifier table replaces the actual symbol during the remainder of the I-J and J-K translation process.

Literals are also placed in the identifier table; the address field of the entry is filled with a pointer to the actual constant (literal). The index into the identifier table is used to represent the literal, in the encoded text. Function names (symbols starting with a period) as other symbols, are placed in the identifier table, the address field is set with a value taken from the list of function names. This list is part of the replacement pair list, and is ordered on the collating value of the function name (excluding the leading period). The "replacement name" of the entry is the 3 byte integer function call number.

Declare statements are parsed and the length or delimiter information is encoded in the identifier table. The value of the address field is set to "Null" if the declare statement contains the keyword "BASED" (see Section 7.1 for more details). Equate statements are also parsed and the pair of symbols is entered in the replacement list.

Text encoding subroutines output the encoded tokens through calls to a standard interface routine (.CODEG) which writes them to an intermediate file. The identifier table and the literals are also written to intermediate files. Now the symbol table is no longer needed and is dropped.

The implementation of I-J syntactic processing is very similar to Phase 1 of the DDL compiler. An EBNF description is written; calls to test encoding subroutines are imbedded in the EBNF. The Syntactic Analysis Program Generator is then used to generate the syntactic processing program. This program is in J-Lang and must be translated into K-Lang before it can be "executed" i.e., interpreted. The encoding subroutines are written in PL/1.

The execution of the syntactic processing program, with the calls to test encoding routines does the I-J translation.

The System Vector Subroutine and the Syntax Failure Subroutine, as described in 4.3.2 and 4.3.3 are used during I-J translation. Their functions are the same here as they are in Phase 1 of the DDL compiler - they allow presetting of the subroutines to be called when syntactic items are recognized, and they facilitate production of error diagnostics.

7. J-Lang

J-Lang code is generated by the SAP Generator and by the DDL Compiler (code generation phase). It is translated into K-Lang by the J-K translator.

7.1 Structure of the Language

The J-Lang is a string of "tokens" which are output individually by calling a routine (.CODEG). Each token is in the form:

1 byte	1 or 2 bytes
type	item number

The type is one of the following codes:

Code	Type	Length of item number
1	Operation	1
2	Relational operator	1
3	Arithmetic operator	1
4	Label	2
5	Symbolic operand	2
6	Pseudo-op	1
7	Punctuation	1
8	Immediate Address	2

"Item Number" is a unique identifier within the group of symbols of the same type. The following table explains how the item number is assigned within the group.

	ITEM NUMBER	SYMBOL
Relational operators	0	EQ
	1	NE
	2	GT
	3	LT
	4	LE
	5	GE
Arithmetic operators	0	+
	1	-
	2	*
	3	/
Pseudo-ops	0	DO
	1	END
	2	::=(equate)
	3	PROCEDURE
	4	LEN-B
	5	LEN-C
	6	DELIM
Operations	0	IF
	1	ELSE
	2	ASSIGNMENT
	3	GOTO
	4	CALL
	5	RETURN
Punctuation	1	THEN
	2	;
	3	(
	4	)

The item number for immediate addresses is the binary equivalent of the decimal number which appeared in the I-Lang statement.

The item number for labels, symbolic operands, literals (constants) is the index into an identifier table. Thisindex is the entry number . The format of the table entries for labels and symbolic operands is:

1 byte	1 byte	2 bytes	3 bytes
DATA TYPE CODE*	LENGTH TYPE	LENGTH	RESERVED FOR MEMORY ADDRESS (Set by J-K Translator)

* See Section 6.2.10 for a list of these codes. Code 0 is used for labels.

The format of the entry for literals is:

1 byte	1 byte	2 bytes	3 bytes
DATA TYPE CODE= 5	LENGTH TYPE	LENGTH	POINTER TO THE CONSTANT

The coding of LENGTH TYPE is as follows:

Bit 0 = 0 - Data is binary
1 - Data is character
1 & 2 = 0 - LENGTH is in binary
1 - LENGTH is the Identifier Table entry index for the symbolic name of the length field (or function)
2 - LENGTH is delimiter punctuation mark.

The Memory address is set to "NULL"* if space is not to be allocated at compilation time for this variable. This means the variable is a label or has been specified as "BASED" (see Section 6.2.9 for a description of BASED). All other entries have the memory address set to zero. (A zero in the memory address does not guarantee that space will be allocated at compilation time, because a further screening is made by the J-K translator - only fixed length fields are allocated.

* "Null" will mean a storage field with all bits set to 1.

7.2 Example of Coding

The instructions

```
DECLARE ALPHA BINARY '16' BASED;

DECLARE B DELIM ';';

DECLARE A CHAR (ALPHA);

TAG:    IF ALPHA LT '1234' THEN DO; A=B; GOTO LABEL1; END;

             ELSE GOTO LABEL2+4;
```

would be encoded -

4	0	1	0	5	1	2	3	7	2	7	1
	TAG		IF		ALPHA		LT		'1234'		THEN
6	0	5	3	1	2	5	4	7	2	6	1
	DO		A		Assignment		B		;		EN
7	2	1	1	1	3	5	5	3	0	5	6
	;		ELSE		GOTO		LABEL2		+		4
7	2										
	;										

The Identifier table would be:

Entry Number	Data Type Code	LENGTH TYPE	LENGTH	Memory Address
0	0			NULL
1	2	(Bits) 00000000	6	NULL
2	5	(Bits) 10000000	4	pointer to constant `1 2 3 4´
3	2	(Bits) 10100000	8	0
4	2	(Bits) 11000000	;	0
5	0			NULL
6	6			4

Note that the separators ":" after label and ";" after DO are not encoded.

7.3 The J-K Translator

J-Lang is generated either by the I-J translator or by the SAP Generator. Although numerically encoded, it is still a high-level, procedural programming language, and the J-K translator must convert this code into a machine level program.

There are two data structures input to the J-K translator: (1) the threaded list of encoded tokens, and (2) the identifier table.

The first step in the translation (code generation) process is assigning segment and offset addresses to all items in the identifier table. This is done by processing sequentially through the table; only entries whose address field contains the value zero are processed.* The length is obtained from LENGTH field of the entry. Only fixed length fields are allocated. Binary items are assigned byte aligned storage; all assignments are in multiples of a byte. The memory address (segment and offset) is placed in the identifier table entry. All data variables are placed in segment 1. All object code will be in segment 2, literals will be segment 3.

The translation process then proceeds by reading tokens from the input stream and stacking them, each time comparing them to certain prototypes in a "reduction table", and, if a match is found, calling a code generation subroutine which produces K-Lang code and pops the proper items off the stack. The only instance where the memory address of an identifier will not be present in the identifier table at the time of code generation, is when the identifier is a statement label that has

* Because the entries for "BASED" variables are set to NULL, and the entries for literals contain non-zero values, this zero-test specifies whether or not allocation was requested.

not yet been defined. This case is handled by generating a special K-Lang address descriptor (code 9), and placing the identifier table entry number in the address bytes where the memory address of the operand would normally go. At the end of code generation these identifier table addresses are replaced by actual memory addresses.

The next section describes the stacking/reduction process, the following section outlines the code generation procedures.

7.3.1 The Translation Process - Stacking/Reduction*

To explain the logic of stacking/reduction we will use a special meta-language. (The implementation will be coded in PL/1. The use of the meta-language is for ease of description.) Each line written in the meta-language contains 4 parts, separated by slashes. Numeric prefix labels (optional) identify the reduction lines. The format is:

label:old top of stack/actions to perform/new top of stack/
next line to execute

The "old top of stack" describes a top of stack configuration to be tested. The items in capital letters are terminal symbols; the items in lower case letters are non-terminal symbols; all punctuation marks are terminal symbols. The order in which they are listed is "top of stack on the right." The symbol "any" indicates any token type is acceptable in this position. For example,

label:PROC any;/__/__/__

would indicate a test is to be made for S_1 (top of the stack) equal to ";", S_3 (3rd entry on stack) equal to "PROC", S_4 equal to ":", and S_5 equal to any label. If the "old top of stack" is not matched, the next sequential line of reduction logic is processed.

* The compilation technique and metalanguage described here are taken from lecture notes used in a course given at M.I.T. in the summer of 1970. See [JD 1].

The "actions to perform" contains a list of subroutines to be called when a match against the stack is found. The subroutine names are separated by blanks.

The "new top of stack" field describes what items should be on the stack after the actions are performed. If this field is null, no change to the stack will be made. If it contains a double quote, the items which were matched are to be deleted. The symbol S_i stands for the item that was matched in stack location i. All other symbols are terminal symbols.

The "next line to execute" is the label which appears on the line which will be executed next - after the appropriate actions have been taken and the new stack set up. Blank means next sequential line will be executed.

The reductions are listed on the next page.

7.3.2 Supporting Subroutines

7.3.2.1 lex

This subroutine reads a single token from the source input and places it on the stack.

7.3.2.2 start-proc

After reading the parameter names from the source, it generates the code necessary to pop any parameters off the processor stack, and store them in the locations addressed by the parameter names. The addresses of parameters will already be in the identifier table in this phase of J-K translation as a result of the storage allocation procedure described above.

Reduction Table

```
    //lex lex/
    label PROC/start-proc lex lex/PROC 2-new-tokens/1
    /error//
1:  DO any/lex/DO S1 new-token/
    label any/label lex/S1  new-token/
    GOTO any/tra/new token/2
    CALL any/gen-call/new token/2
    RETURN any/ret/new token/2
    IF fcall/gen-call-1/if-code/3
    IF any/rel-gen/if code/3
    DO END any//S1/2
    PROC END any/end-proc/" /
    /assign/new token/2
2:  ;/lex/new-token/4
    /error/
3:  any THEN/if-gen lex lex/new-label  THEN  2-new-tokens/1
    /error//
4:  any THEN ELSE/then lex lex/new-label  ELSE  2-new-tokens/1
    any THEN any/force-else lex/S1 2-new-tokens/1
    any ELSE any/else/S1/4
    /lex/new-token/1
```

7.3.2.3 label

The current value of the object code location counter is placed in the identifier table entry for this label. This "defines" the label; from now on, any references to this label can be generated directly, without the need to temporarily use an address in the identifier table entry, as described in 7.3.

7.3.2.4 tra

This generates a "jump" instruction. The method of determining the operand address for symbolic names has already been explained in Sections 7.3 and 7.3.2.3. The remainder of the process is described in the section on address expression evaluation (section 7.3.2.15).

7.3.2.5 gen-call-1 and gen-call

Reads the parameters and the condition code for the function call (fcall) from the source input stream. Generates the "call" instruction in K-Lang, using the parameter names just read. If this is a gen-call-1 it changes S2 to indicate the condition code specified in the source. Before existing it calls lex to read and stack the next token.

7.3.2.6 ret

Reads two address expressions from the source input stream (if they are present); uses the first to generate code to set the condition code, and the second to generate code to place the value to be returned on the processor stack. Exits with the stack holding the next token after the address expressions.

7.3.2.7 rel-gen

Reads "address expression < rel op > address expression" and the data limit expression, if present, from the input stream. Uses the two address expressions to generate the operands of a K-Lang compare instruction. If present, the data limit determines the length of the operands. If not, the length in the identifier table entry for the first operand is used. Uses the relational operator to leave the proper condition code on the stack - to be used for generating the branch instruction.

7.3.2.8 end-proc

This does the end of source processing, including the identifier table address replacement described in 7.3.

7.3.2.9 if-gen

Generates a conditional branch instruction whose condition is the negative of the condition code on the compiler's **stack**. (This branch is the "false jump" out of the If-statement.) The operand is obtained by calling the label-gen routine, which returns a new label. This new label is also saved on the stack (in S2).

7.3.2.10 then

Calls label-gen and generates a jump to the label returned by this routine. "Defines" the label saved in S3 (i.e. moves the current value of the object code location counter to the identifier table entry for the label). Deletes S2 and S3 and replaces them with the new label just obtained.

7.3.2.11 force-else

This is used when it is found that there is no "else-clause" in an If statement. It just "defines" the label saved in S3 (see 7.3.2.11 for the method used). S1 is kept on the stack, S2 and S3 are dropped.

7.3.2.12 else

Same processing as described under force-else.

7.3.2.13 label-gen

This routine generates unique labels by adding 1 to the last identifier assigned. (The value is initialized with the last **label** number which appeared in the source input.) At the time a label is generated, a new entry is created for it in the identifier table.

7.3.2.14 Address Expression Generation

This is a routine which is called to read and generate code for evaluation of address expressions. The output is a K-Lang variable length operand (including Data Limit, descriptors and addresses) as described in Section 8.1. The code generation logic is shown in Figure 7-1.

Special note should be made of the address code generation for pushdown stacks and lists. The identifier table entry for these indicates the address of the control words for the list (or stack). To generate the "address of the list", the address expression code generation routine generates a call to a general purpose list (or stack) handling routine - the parameter is the address of the control words. At execution, this function call will return the address of the data item.

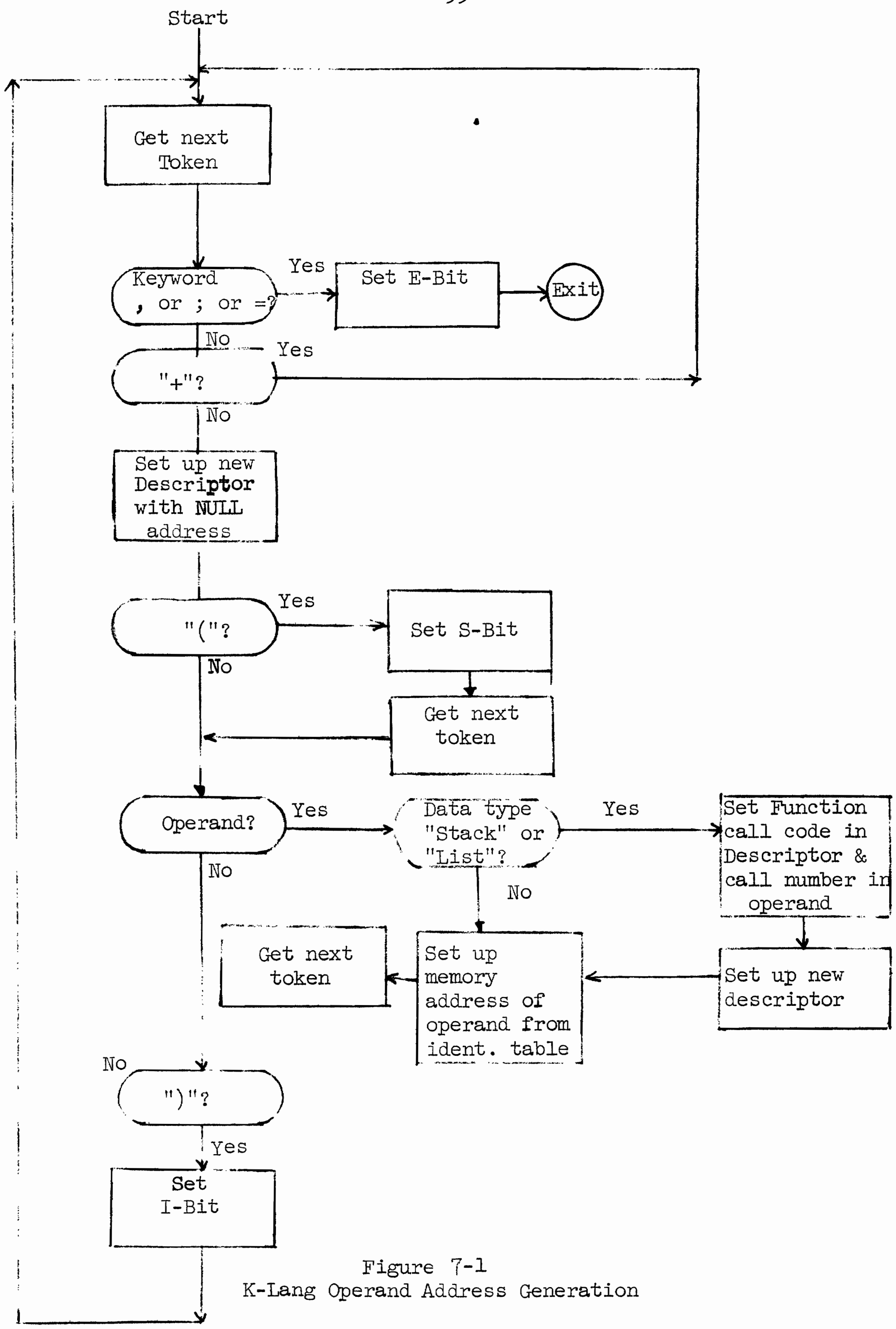

Start
Get next Token
Keyword , or ; or =?
Yes
Set E-Bit
Exit
No
Yes
"+"?
No
Set up new Descriptor with NULL address
"("?
Yes
Set S-Bit
No
Get next token
Operand?
Yes
Data type "Stack" or "List"?
Yes
Set Function call code in Descriptor & call number in operand
No
No
Get next token
Set up memory address of operand from ident. table
Set up new descriptor
No
")"?
Yes
Set I-Bit

Figure 7-1
K-Lang Operand Address Generation

The length to be encoded in the Data Limit field is obtained from the identifier table entry of the first symbolic name used in the address expression, unless the statement in which the operand appears is terminated with a Data Limit expression (LEN-B, LEN-C, or DELIM). In this case, the Data Limit expression provides the length of the first operand used in the statement (the length of other operands are not overridden by the Data Limit).*

7.3.2.15 Assign

If no match against the stack is found, this routine is called. It first calls the address generation routine (ADDGEN) which uses S2, and possibly S1 or more input tokens to generate the first operand of a K-Lang move or arithmetic operation. If ADDGEN does not exit with code "=", a syntax error is assumed and the scan continues till the next semi-colon, 2 tokens are read, and a return is made to the main processing routine. If ADDGEN exits with a "=", the next address expression is fetched. If ADDGEN exits with an arithmetic operator, the third address expression is fetched and a K-Lang arithmetic operation is generated; otherwise a K-Lang move of the second operand to the first is generated. The length of the move is determined by the data limit expression in the source input or, if it is not present, the length is determined by the identifier entry for the first operand. The routine exits with the new token (which must be a semi-colon) on top of the stack.

* One use of the Data Limit expression is to specify the desired length of the results of an arithmetic operation. Another use is to limit the length of a compare, or move operation - both of which are governed by the length of the first operand, and extend or truncate the other operands to suit the first.

8. K-Lang

K-Lang is the lowest level language generated by the DDL compiler. Although K-Lang is a machine level language, it is interpretively executed because there is no hardware implementation of the "K-Machine".

8.1 The K-Machine

The K-Machine is a variable-length instruction processor with a pushdown stack (4 bytes wide) and up to 8 variable length segments of main memory (each up to 64K bytes long and pointed to by a segment table entry). Data items moved into memory location zero are pushed on to a stack. An item can be popped off this stack by using location zero as the source address. To read or alter any of the top 16 elements of the stack (without pushing or popping), memory addresses 1-16 are used. The stack is used for arithmetic operations (including intermediate address calculations), and for passing parameters to subroutines.

The K-Machine has a "control storage" memory, accessible using the special op-codes Read and Write Control Storage (see below for a description of these op-codes). The Control Storage is unsegmented - it contains the pushdown stack, the segment table, the program counter, and an internal status word. It also contains a 64 word cyclic buffer, which contains the contents of the program counter before each of the last 64 branch instructions. In addition, there is a list of "interrupt condition" addresses, where control is given should any special condition occur. Special conditions include illegal op-code, illegal addressing (non-existent segment or outside bounds of segment), stack overflow, stack underflow, and a match on the program counter or effective operand address check. (Two words of control store hold the values to be tested against and, each time the program counter, or operand address changes, a test is made.)

Condition codes resulting from comparison operations are saved as binary values in an internal status word.

The format of the segment table entries is:

Starting Address	Length of Segment

Both bit and Character data are processed - the type is determined by a "data limit" field in the machine instruction. (See Section 8.2.) Instructions may have mixed-type operands.

8.2 Instruction Format

The format of the K-Lang instruction is shown in Figure 8-1.

8.2.1 Descriptors and Data Limits

Each "adr. descr i_j" (address descriptor) refers to a corresponding "address i_j" and describes the way in which address i_j is to be used. The format of the descriptor is:

S	I	E	code

$$\underbrace{\boxed{S}\boxed{I}\boxed{E}}_{\text{Bit flags}}$$

The meanings of the flags are:

S = Stack intermediate address

I = Do indirect addressing and unstack the saved address

E = End of address components for this operand

A more complete explanation of the flags appears in the section on address calculation. The meaning of "code" is presented in the next section.

Machine Instruction Format

Op-Code	Data Limit$_1$	adr.descr.1$_1$	address1$_1$	adr.descr.1$_2$	address1$_2$ ....
1 byte	1 byte or 2 bytes	1 byte	2 bytes		

address1$_n$	Data Limit$_2$	adr.descr.2$_1$	address2$_1$	adr.descr.2$_2$	address2$_2$ .... address2$_n$

Figure 8-1

The encoding used in the "data limit" field is as follows:

Bit 0 = 1 - Data is Binary

= 0 - Data is Character

Bits 1-7 = length (in bits or characters, as indicated by Bit 0)

If the 1st byte of the Data limit field is zero, then the operand is of varying length. In this case the data limit field has a second byte; if this second byte is non-zero it contains the punctuation mark which marks the end of the operand. If the second byte is zero the first address following points to the length (i.e. yields the address of the length field).

8.2.2 Address Components

The meaning of "address i;" varies with the descriptor code as follows:

Code	Meaning of Address
0-7	Relative address within memory segment 0-7
8	Function call number
9	Identifier table relative address*
10	Constant (immediate operand)**

8.2.3 Operand Address Calculation

The algorithm used to calculate effective addresses is

Step 1 - [Initialize]

Clear intermediate address. Set pointer to first descriptor. Go to step 3.

Step 2 - [Get Next Descriptor]

Step pointer to next descriptor.

* Used only during I-K translation.

** If an immediate operand is specified it must be the only component of the address.

Step 3 - [Get address]

If S-flag is on stack intermediate address, then clear Intermediate Address. Using descriptor code, obtain the address component value. Constants and memory addresses are values; function calls return the value . (See Section 8.4 for a description of passing parameters to function.) Add the value of the address component to the intermediate address.

Step 4 - [Indirect Addresses Resolved]

If the I-Bit is on replace the intermediate address with the value pointed to by the intermediate address. Pop the stack and add the popped item to the intermediate address.

Step 5 - [Test for End of Operand]

If the E-Bit in the descriptor is not on, then go to step 2. Otherwise the algorithm terminates with the "intermediate address" field containing the effective address of this operand.

8.3 **Machine** Operations

The following table shows the operations of the K-Lang Machine":

Op-Code	Operands	Action
JUMP	a_1	The next instruction is fetched from location a_1.*

* The a_i are addresses in the format described in the preceding paragraphs.

Op-Code	Operands	Action
CALL	$a_1,a_2,....a_n$	A subroutine call is made to location a_1. The number of operands is determined scanning the descriptors in the instruction. A descriptor of all 1's indicates the end of the operands. The operands $a_2...a_n$ are passed as parameters (see Section 8.4 for further details).
COMPARE	a_1,a_2	The operand addressed by a_1 is compared with the one addressed by a_2. The internal condition code is set to "greater" if operand 1 is greater than operand 2, to "equal" if they are equal, and to "less" if operand 1 is less than operand 2. The number of bits, or characters compared is specified in the "data limit" field of the instruction. See Section 8.2.
CONDITIONAL BRANCH	a_1,a_2	If the condition code of the K-Machine matches the code specified by a_1, then execution continues at location a_2. If not, execution continues in line. The conditions which may be tested for are EQ, NE, GT, LT, GE, LE.

Op-Code	Operands	Action
MOVE	a_1,a_2	The data at location a_1 is moved to location a_2. The number of bits or characters moved is specified by the data limit field of the instruction.
ADD SUBTRACT MULTIPLY	a_1,a_2,a_3	The arithmetic operation is performed on the first two operands (a_1 and a_2) and the results placed in location a_3.
DIVIDE	a_1,a_2,a_3,a_4	The first operand (a_1) is divided by the second (a_2); the quotient is placed in a_3 and the remainder in a_4.
RETURN	a_1,a_2	The contents of a_1 are used as the condition code (Equal, High, Low) to be set in the "internal machine indicators". (It is these indicators which are tested by the IF instructions.) The contents of a_2 become the value of the operand if the RETURN ends the execution of a function. Otherwise this value is not used. Ref. Sect. 8.4.
WRITE CONTROL STORAGE	a_1,a_2	The contents of the location in main memory specified (any type of address) by a_1 is moved to the location in Control Storage specified by a_2. The segment specified by a_2 is ignored in the address calculation.

Op-Code	Operands	Action
		The length of the operands is specified by the data limit field as in other instructions.
READ CONTROL STORAGE	a_1,a_2	The contents of the location in control memory specified by a_1 are moved to the main memory location specified by a_2. The segment specified by a_1 is ignored in a_1 address calculation.

8.4 Subroutine Linkage

Execution of the K-Lang CALL instruction causes four actions to occur.

(1) The current contents of the location counter are pushed onto the processor stack.

(2) The address of the called subroutine is obtained. This is done in one of two ways, depending on the contents of the descriptor associated with the first address in the K-Lang call instruction. If the address descriptor code indicates "Function Call Number", the associated address component is used to index a table of subroutine addresses, from which the starting address of the subroutine is taken. If the first operand address of the CALL instruction is not a function call number, it must be a main memory address (this is the only other addressing mode permitted for subroutine addresses). This main memory address is the starting address of the called subroutine.

(3) The remaining operand addresses of the CALL instruction are resolved and the effective addresses obtained are pushed on the stack.*

(4) The subroutine linkage exits by placing the address of the called subroutine in the program counter.

An alternate method of invoking a subroutine is the function call. Any address component may be a function call. The value returned by the subroutine becomes the value of the address component. The subroutine linkage logic for a function call is identical to the logic of the CALL instruction, with function call addressing mode, except that the addresses of the parameters are taken to be the next sequential operand addresses of the K-Lang instruction in which the function call appears. (The end of the parameters is indicated, just as in the case of the CALL instruction parameters, by the occurrence of an address descriptor of all 1's.)

The first instructions of each subroutine are a "prologue" generated by the J-K translator. The prologue is a series of "pop" commands which move the addresses off the stack and into the working storage area of the subroutine (see Section 7.3.2.2).

Execution of a K-Lang RETURN instruction causes a return from subroutine. The following actions are taken:

(1) the condition code of the K-Machine is set with the data addressed by the first operand of the RETURN instruction.

(2) the 4 bytes pointed to by the second address of the RETURN instruction are placed on top of the processor stack, replacing the return address which had been placed there by the subroutine call logic.

* Resolution of addresses is described in Section 8.1.3. Note that in this process a recursive use of the subroutine linkage is possible, because operand addresses may themselves be function calls. Recursion presents no problem because the working storage for the subroutine linkage is the processor stack.

(3) the return address just removed from the stack becomes the new value of the program counter (i.e., control is returned to the point at which the subroutine call was made).

If the subroutine was called as a function during address expression evaluation the 4 byte address on top of the processor stack becomes the effective value of the address component and is added to the effective address being accumulated. If the subroutine was called with a CALL instruction the calling program can take the returned value off the top of the processor stack. If more than one value is to be returned values may be placed in positions below the top of the stack without changing the top of the stack, through the use of the stack addressing scheme described in Sect. 8.1.

BIBLIOGRAPHY

[BR1] "The Compiler-Compiler," Brooker & Morris, Annual Review in Automatic Programming, Vol. 3, 1963, p. 229.

[IN1] "A Syntactic-Oriented Translator," Ingerman, Academic Press, New York, 1966.

[JD1] Lecture Notes from M.I.T. Course 6.251 Systems Programming, Summer 1970. Author, J. Donovan.

[KN1] "Fundamental Algorithms," The Art of Computer Programming, Vol. 1, D. Knuth, Prentice Hall, 1968.

[MC1] "A Compiler Generator," W. N. McKeeman, J. J. Horning, D. B. Wortman, Prentice Hall, Inc., 1970.

[NA1] Naur, P. (Ed.), Report on the Algorithm Language ALGOL 60, Comm. ACM, 3, May 1960.

[RA1] "ALGOL 60 Implementation," B. Randall and L. J. Russell, Academic Press, New York, 1964.

[SM1] "A Manual with Examples for the Data Description Language," Diane P. Smith, April 1971, University of Pennsylvania, The Moore School of Electrical Engineering, April 1971.

[SM2] "An Approach to Data Description and Conversion," Diane P. Smith, Ph.D. Dissertation, The Moore School of Electrical Engineering, University of Pennsylvania, December 1971.

APPENDIX A

SYNTAX DESCRIPTION OF DDL

DESCRIBED IN EBNF

```
---------SYNTAX OF THE DDL PROGRAMMING LANGUAGE-----------

<DDL_PROGRAM> ::= DESCRIPTION(<TITLE>)#<PROGRAM_BODY>DESCRIPTIONEND

<TITLE> ::= <UDN>

<PROGRAM_BODY> ::= <DESCRIBE_PARAGRAPH_LIST><EXECUTE_PARAGRAPH_LIST>
                  "<PROGRAM BODY>"

<DESCRIPTION_PARAGRAPH_LIST> ::= <DESCRIBE_PARAGRAPH> "<DESCRIBE_PARAGRAPH>" *

<EXECUTE_PARAGRAPH_LIST> ::= <EXECUTE_PARAGRAPH> "<EXECUTE_PARAGRAPH>" *

<DESCRIBE_PARAGRAPH> ::= DESCRIBE(<TITLE>)#<DESCRIBE_STMT_LIST>

<EXECUTE_PARAGRAPH> ::= EXECUTE(<TITLE>)#<EXECUTE_STMT_LIST>

<DESCRIBE_STMT_LIST> ::= <DESCRIBE_STMT> "<DESCRIBE_STMT>" *

<EXECUTE_STMT LIST> ::= <EXECUTE_STMT>  <EXECUTE_STMT>" *

<DESCRIBE_STMT> ::= <NAME>:<RCD_SPEC_STMT_TYPE1>#
                  | <RCD_SPEC_STMT_TYPE2>#
                  | <NAME>:<FILE_SPEC_STMT>#

<EXECUTE_STMT> ::= <NAME>:<TASK_SPEC_STMT_TYPE1>#
                 | <TASK_SPEC_STMT_TYPE2>#

<REC_SPEC_STMT_TYPE1> ::= <CHAR_STMT>
                        | <FIELD_STMT>
                        | <GROUP_STMT>
                        | <SET_STMT>
                        | <DEVICE_STMT>
                        | <CRITEX_STMT>
                        | <CRITERION_STMT>
                        | <STORRCD_STMT>
```

```
<REQ_SPEC_STMT_TYPE2> ::= <PARAMETER_STMT>
                        | <RELVAL_STMT>
                        | <CONCODE_STMT>


<DEVICE_STMT> ::= <BLOCK_STMT>
                | <BBLOCK_STMT>
                | <CARDIN_STMT>
                | <CARDOUT_STMT>
                | <TAPEIN_STMT>
                | <TAPEOUT_STMT>


<PARAMETER_STMT> ::= <LENGTH_STMT>
                   | <CNT_STMT>
                   | <EXIST_STMT>


<FILE_SPEC_STMT> ::= <LINK_STMT>
                   | <SEQUEN_STMT>
                   | <STORFILE_STMT>


<TASK_SPEC_STMT_TYPE1> ::= <ASSOCIATE_STMT>
                         | <CREATE_STMT>


<TASK_SPEC_SMT_TYPE2> ::= <COMBINE_STMT>
                        | <DISJOIN_STMT>
```

```
----------                  SYNTAX


<ASSOCIATE_STMT> ::= ASSOCIATE(<ASSOCIATE_LIST> ";<CRITERION_NAME>" )


<ASSOCIATE_LIST> ::= (<ASSOCIATE_PAIR>) ",(<ASSOCIATE_PAIR>)" *


<ASSOCIATE_PAIR> ::= <TARGET_NAME>,<SOURCE_NAME>


<TARGET_NAME> ::= <REF_NAME>


<SOURCE_NAME> ::= <REF_NAME>
                | <COMBINE_STMT>
                | <DISJOIN_STMT>

----------                  SEMANTICS
```

THE ASSOCIATE STATEMENT IS A USED TO SPECIFY THE RELATIONSHIPS BETWEEN VALUES OCCURRING IN ONE STATE AND THE SAME VALUES OCCURRING IN A DIFFERENT STATE. THE CURRENT STATE OF A VALUE IS CALLED ITS SOURCE STATE. THE STATE INTO WHICH A VALUE IS TO BE TRANSLATED IS CALLED ITS TARGET STATE.

```
----------                    SYNTAX


<BBLOCK_STMT> ::= BBLOCK(<UNIFORMITY>,<LENGTH>,<RECORD_DISTRIBUTION>,
                  <RECORD_COUNT_UNIF>,<RECORD_COUNT>,<BASIC_BLOCK_COUNT>,
                  <RECORD_ORDER>,(<LIST>) ",(<LIST>)" *)


<UNIFORMITY> ::= F
               | V
               | FIXED
               | VARIABLE


<LENGTH> ::= <INTEGER>
           | NOLIM


<RECORD_DISTRIBUTION> ::= WHOLE
                        | SPLIT


<RECORD_COUNT_UNIF> ::= <UNIFORMITY>


<RECORD_COUNT> ::= <INTEGER>
                 | NOLIM


<BASIC_BLOCK_COUNT> ::= <INTEGER>
                      | NOLIM


<RECORD_ORDER> ::= SPEC
                 | NOORD


<LIST> ::= <RECORD_NAME>,<OPTIONALITY>,<REP_NUMBER> ",<CRITERION_NAME>"


<RECORD_NAME> ::= <UUDN>


<OPTIONALITY> ::= M
                | O
                | OPTIONAL
                | MANDATORY


<REP_NUMBER> ::= <INTEGER>
```

```
                    | NOLIM


-----------                       SEMANTICS

THE BBLOCK STATEMENT IS USED TO DEFINE THE BASIC PHYSICAL BLOCK OF STORAGE ON A
DEVICE MEDIUM, AND TO DESCRIBE THE LOCATION OF A RECORD OCCURRENCES IN THE
BASIC BLOCK
```

```
-----------                    SYNTAX


<BLOCK_STMT> ::= BLOCK(<ORDER>,(<LISTA>) ",(<LISTA>)" *)


<ORDER> ::= NOORD
          | SPEC


<LISTA> ::= <B/BLOCK_NAME>,<OPTIONALITY>,<REF_NUMBER> ",<CRITERION_NAME>"

-----------                    SEMANTICS
```

THE BLOCK STATEMENT IS USED TO SPECIFY A BLOCK. A "BLOCK" IS AN ORGANIZATION OF OCCURRENCES OF DIFFERENT BASIC BLOCKS OF STORAGE AND/OR OTHER BLOCKS.

----------- SYNTAX

<CARDIN_STMT> ::= CARDIN(<BBLOCK_NAME> ";<ASSOCIATION_LIST>")

<BBLOCK_NAME> ::= <UUDN>

<ASSOCIATION_LIST> ::= <UUDN>

----------- SYNTAX

<CARDOUT_STMT> ::= CARDOUT(<BBLOCK_NAME> ";<ASSOCIATION_LIST>")

----------- SEMANTICS

THE CARDIN STATEMENT IS USED TO SPECIFY THAT THE CARD READER IS TO BE USED FOR INPUTTING DATA

----------- SEMANTICS

THE CARDOUT STATEMENT IS USED TO SPECIFY THAT THE CARD PUNCH IS TO BE USED FOR OUTPUTTING DATA.

---------- SYNTAX

```
<CHAR_STMT> ::= CHAR(<CHAR_SIZE>,<CHAR_SET_DESC>)


<CHAR_SIZE> ::= <INTEGER>


<CHAR_SET_DESC> ::= <UDN>,<UDN>
                  | BCD
                  | EBCDIC
```

---------- SEMANTICS

THE CHAR STMT IS USED TO SPECIFY THE CHARACTER SET TO BE USED TO REPRESENT VALUES.

---------- SYNTAX

```
<CONCODE_STMT> ::= CONCODE( "<LABEL>" <CONTROL_CODE_INT_EXP>)


<LABEL> ::= <CONSTANT_STMT>


<CONTROL_CODE_INT_EXP> ::= DELIM,<POSITION_TYPE>
                         | FILLER,<INTEGER>,<INTEGER> ";<INTEGER>"  ":INX"


<POSITION_TYPE> ::= PRX
                  | PTX
                  | INX
```

---------- SEMANTICS

THE CONCODE STATEMENT IS USED AS A PARAMETER IN OTHER DDL STATEMENTS TO DEFINE CONTROL CODES IN THE FORM OF CHARACTER STRINGS AND/OR POSITIONS ON MEDIA. THESE CODES ARE USED TO LOCATE OR POSITION VALUES AND GROUPS OF VALUES.

---------- SYNTAX

```
<CONSTANT_STMT> ::= CONSTANT(<CHAR_STRING>,<DATA_TYPE>)


<CHAR_STRING> ::= <FULL_CHAR_SET> "<FUL_CHAR_SET>" *


<DATA_TYPE> ::= B
              | C
              | <UUDN>
```

---------- SEMANTICS

THE CONSTANT STATEMENT IS USED AS A PARAMETER WHEN ARBITRARY CHARACTER STRINGS ARE REQUIRED

---------- SYNTAX

```
<CRITERION_STMT> ::= CRITERION(<CRITERION_EXP_NAME> ";<RPLVAL_STMT>" )
```

---------- SEMANTICS

THE CRITERION STATEMENT IS USED TO DETERMINE WHETHER OR NOT VALUES AND SETS OF VALUES ARE TO BE ACCEPTED FOR INPUT, OUTPUT, OR STORAGE.

```
----------                    SYNTAX


<CREATE_STMT> ::= CREATE(<FILE_NAME>,<SOURCE_FILE_LIST>,<ASSOCIATE_LISTA>)


<FILE_NAME> ::= <UUDN>


<SOURCE_FILE_LIST> ::= <FILE_NAME> ",<FILE_NAME>" *


<ASSOCIATE_LISTA> ::= <UUDN> ",<UUDN>"

----------                    SEMANTICS
```

THE CREATE STATEMENT IS USED TO REQUEST THE CREATION OF A FILE. IT SPECIFIES THE SOURCES OF THE VALUES TO BE ORGANIZED AND THE ORGANIZATION OF THOSE VALUES

```
----------                    SYNTAX


<CRITEX_STMT> ::= CRITEX(<CRITERION_EXP>)


<CRITERION_EXP> ::= (<CRITERION_DATA_NAME>)<CRITERION_DATA_NAME_EXP>
                  | NOT(<CRITERION_EXP_FORM>)
                  | <CRITERION_TAYPE_1>(<CRITERION_EXP_FORM>)
                  | <CRITERION_TAYPE_2>(<CRITERION_EXP_FORM>)


<CRITERION_DATA_NAME_EXP> ::= <REL.OP>(<ARITH.OP_EXP>)
                            | MEM(<SET_NAME>)


<CRITERION_DATA_NAME> ::= <UUDN>
                        | <INTEGER_EXP>


<ARITH.OP_EXP> ::= <CRITERION_DATA_NAME>
                 | <CONSTANT_STMT>
                 | (<CRITERION_DATA_NAME>)<ARITH_OP>(<ARITH.OP_EXP>)
                 | (<CONSTANT_STMT>)<ARITH_OP>(<ARITH,OP_EXP>)


<SET_NAME> ::= <UUDN>


<CRITERION_EXP_FORM> ::= <UUDN>
                       | <CRITERION_EXP>


<CRITERION_TYPE_1> ::= (<CRITERION_EXP_FORM>)AND "<CRITERION_TYPE 1>"
                     | <CRITERION_EXP>


<CRITERION_TYPE_2> ::= (<CRITERION_EXP_FORM>)OR "<CRITERION_TYPE 2>"
                     | <CRITERION_EXP>

----------                    SEMANTICS
```

THE CRITEX STATEMENT IS USED TO SPECIFY CRITERIA THAT MUST BE SATISFY BY VALUES AND SETS OF VALUES IN ORDER FOR THE VALUES TO BE ACCEPTED FOR INPUT, OUTPUT,OR STORAGE

---------- SYNTAX

```
<COMBINE_STMT> ::= COMBINE(<SOURCE_NAME>,<RCD_OCC_COUNT_TYPE>,<RDC_OCC_COUNT>
                   ";<CRITERION_NAME>" )


<SOURCE_NAME> ::= <REF_NAME>


<RCD_OCC_COUNT_TYPE> ::= F
                       | V
                       | FIXED
                       | VARIABLE


<RCD_OCC_COUNT> ::= <INTEGER>
                  | NOLIM
```

---------- SEMANTICS

THE COMBINE STATEMENT IS USED WHEN TWO OR MORE OCCURRENCES OF A RECORD ARE TO BE THE SOURCES OF VALUES FOR A REPEATING FIELD OR GROUP. IT MAY BE USED IN CONJUNCTION WITH THE ASSOCIATE STATEMENT

---------- SYNTAX

```
<DISJOIN_STMT> ::= DISJOIN(<SOURCE_NAME>,<RCD_OCC_COUNT_TYPE>,<RCD_OCC_COUNT>)
```

---------- SEMANTICS

THE DISJOIN STATEMENT IS USED WHEN VALUES OF A REPEATING FIELD OR GROUP IN A SINGLE RECORD OCCURRENCE ARE TO BE USED TO FORM TWO OR MORE TARGET OCCURRENCES OF A RECORD. THE DISJOIN STATEMENT MAY BE USED IN CONJUNCTION WITH THE ASSOCIATE STATEMENT.

```
----------                    SYNTAX


<FIELD_STMT> ::= FIELD(<UNIFORMITY>,<LENGTH_TYPE>,<LENGTH>,<DATA_TYPE>
               ";<OPTION_PART>"  ":<OPTION_CONCODE" )


<DATA_TYPE> ::= B
              | C
              | <UDN>


<UNIFORMITY> ::= F
               | V
               | FIXED
               | VARIABLE


<LENGTH_TYPE> ::= B
                | C
                | <UDN>


<LENGTH> ::= <INTEGER>
           | NOLIM
           | <INTEGER_EXP>


<INTEGER_EXP> ::= <INTEGER>
                | <LENGTH_STMT>
                | <CNT_STMT>
                | <EXIST_STMT>


<OPTION_PART> ::= F,<ALIGNMENT_FACTOR>,<ALIGNMENT>
                | V,<ALIGNMENT>


<OPTION_CONCODE> ::= <CONCODE_STMT> ",<CONCODE_STMT>" *


<ALIGNMENT_FACTOR> ::= <INTEGER>


<ALIGNMENT> ::= <ORIENTATION>,<PAD_FACTOR>


<ORIENTATION> ::= L
                | R
```

```
<PAD_FACTOR> ::= <CONSTANT_STMT>
```

----------- SEMANTICS

THE FIELD STATEMENT IS USED TO SPECIFY A FIELD. A FIELD IS THE FORM OF A SET OF VALUES AND A VALUE IS A STRING OF CHARACTERS OR BINARY DIGITS.

----------   SYNTAX

```
<GROUP_STMT> ::= GROUP(<GROUP_ORDER>,(LIST1>) ",(<LIST1>)" * ";<CONCODE>" )


<GROUP_ORDER> ::= NOORD
                | SPEC


<LIST1> ::= <NAME>,<OPTIONALITY>,<REP.UNIFORMITY>,<REP._NUMBER> ";O,<ORDER1>"
        ";V,<CRITERION_NAME>"


<NAME> ::= <UUDN>


<OPTIONALITY> ::= M
                | O
                | MANDATORY
                | OPTIONAL


<REP._UNIFORMITY> ::= F
                    | V
                    | FIXED
                    | VARIABLE


<REP._NUMBER> ::= <INTEGER>
                | NOLIM
                | <INTEGER_EXP>


<ORDER1> ::= ASCLEX
           | DSCLEX
           | <UDN>


<CRITERION_NAME> ::= <UUDN>
```

----------   SEMANTICS

THE GROUP STATEMENT IS USED TO SPECIFY A GROUP WHERE A GROUP IS DEFINED AS AN ORGANIZATION OF VALUES OF DIFFERENT FIELDS AND/OR OTHER GROUPS.

----------- SYNTAX

```
<LENGTH_STMT> ::= LENGTH(<DATA_NAME>,<LENGTH_TYPE>)
```

```
<LENGTH_TYPE> ::= B
                | C
```

----------- SYNTAX

```
<CNT_STMT> ::= CNT(<DATA_NAME>)
```

----------- SYNTAX

```
<EXIST_STMT> ::= EXIST(<DATA_NAME>)
```

----------- SEMANTICS

THE LENGTH STATEMENT CAN BE USED AS A PARAMETER WHEN THE PARAMETER CAN BE AN INTEGER AND THE PARAMETER IS TO BE ASSIGNED THE LENGTH OF FIELD, OR GROUP, ETC.

----------- SEMANTICS

THE CNT STATEMENT CAN BE USED AS A PARAMETER WHEN THE PARAMETER CAN BE AN INTEGER AND THE PARAMETER IS TO BE ASSIGNED THE NUMBER OF TIMES A FIELD, OR GROUP, ETC. OCCURS.

----------- SEMANTICS

THE EXIST STATEMENT CAN BE USED AS A PARAMETER WHEN THE PARAMETER CAN BE AN INTEGER AND THE PARAMETER IS TO BE THE INTEGER 1 IF THE VALUES OF ANOTHER FIELD OR GROUP OCCUR AND 0 IF THE VALUES OF THE OTHER FIELD OR GROUP DO NOT OCCUR.

---------- SYNTAX

```
<LINK_STMT> ::= LINK(<RCD_STATE_NAME>,<RCD_STATE_NAME>,<CRITRION_NAME>,
                <LINK_UNIFORMITY>,<LINK_NUMBER>)


<CRITERION_NAME> ::= NUCRD
                   | <UCRN>


<LINK_UNIFORMITY> ::= FIXED
                    | F
                    | VARIABLE
                    | V


<LINK_NUMBER> ::= <INTEGER>
                | NOLIM
```

---------- SEMANTICS

THE LINK STATEMENT IS USED TO SPECIFY THE MEANS FOR DETERMINING WHEN ONE OCCURRENCE OF A RECORD IS TO BE LINKED TO ANOTHER OCCURRENCE OF THE SAME OR OF A DIFFERENT RECORD, AND THE MAXIMUM NUMBER OF TARGET OCCURRENCES THAT MAY BE LINKED TO A SINGLE SOURCE OCCURRENCE.

```
----------                    SYNTAX


<RECORD_STMT> ::= RECORD (<GROUP_NAME> ".<CRITERION_NAME>" )


<GROUP_NAME> ::= <UIDN>
```

----------   SEMANTICS

THE RECORD STATEMENT IS USED TO SPECIFY A RECORD WHERE A RECORD IS DEFINED AS A GROUP WHOSE VALUES ARE TO BE THE BASIC UNIT OF STORAGE AND RETRIEVAL.

----------- SYNTAX

```
<RPLVAL_STMT> ::= RPLVAL(<DATA_NAME>,<REP_VAL>)


<DATA_NAME> ::= <REF_NAME>


<REP_VAL> ::= <REF_NAME>
            | <CONSTANT_STMT>
```

----------- SEMANTICS

THE RPLVAL STATEMENT IS USED AS A PARAMETER IN A CRITERION STATEMENT WHENEVER THE VALUES OF A FIELD OR GROUP ARE TO BE REPLACED BY OTHER VALUES UPON THE FAILURE OF A VALUE OR SET OF VALUES TO SATISFY THE CRITERION.

```
----------                    SYNTAX


<SEQUEN_STMT> ::= SEQUEN(<LINK_NAME> ",<LINK_NAME>" *)


<LINK_NAME_LIST> ::= <LINK_NAME> ",<LINK_NAME>" *


----------                    SEMANTICS
```

THE SEQUEN STATEMENT IS USED TO SPECIFY THAT THE OCCURRENCES OF A RECORD ARE TO BE ORGANIZED SEQUENTIALLY IN STORAGE ACCORDING TO SOME CRITERION.

```
----------                    SYNTAX


<SET_STMT> ::= SET(<MEMBER>)


<MEMBER> ::= <CONSTANT_STMT> ",<CONSTANT_STMT>" *

----------                    SEMANTICS

THE SET STATEMENT IS USED TO SPECIFY A SET OF STRINGS OVER THE CHARACTER SET .
THE SET IS SPECIFIED BY LISTING THE MEMBERS EXTENSIVELY AND ASSIGNING A NAME
TO THE COLLECTION OF MEMBERS LISTED.
```

----------- SYNTAX

```
<STORFILE_STMT> ::= STORFILE(<STRUCTURE_NAME_LIST>)


<STRUCTURE_NAME_LIST> ::= <UUDN> ",<UUDN>" *
```

----------- SEMANTICS

THE STOREFILE STATEMENT IS USED TO SPECIFY THE ORGANIZATION OF RECORD OCCURRENCES IN STORAGE.

```
----------                    SYNTAX


<STORRCD_STMT> ::= STORRCD(<RECORD_STRUCTURE_NAME> ",<UDON>" )


<RCD_STRUCTURE_NAME> ::= <UDON>


----------                    SEMANTICS
```

THE STORRCD STATEMENT IS USED TO DECLARE THAT THE OCCURRENCES OF A PARTICULAR RECORD ARE TO BE INPUT, OR OUTPUT ON A PARTICULAR DECVICE, OR THAT THEY ARE TO BE STORED PHYSICALLY, AND TO SPECIFY THE STORAGE DEVICE WHEN THIS IS TO DIFFER FROM THE CONVENTIONS OF THE DL IMPLEMENTATION.

```
-----------                    SYNTAX


<TAPEIN_STMT> ::= TAPEIN(<FILE_BLOCK_NAME>,<PHYSICAL_BLOCK_NAME>
                 ",<PHYSICAL_BLOCK_NAME>   * ";<ASSOC_LIST_NAME>" )


<PHYSICAL_BLOCK_NAME> ::= <UUDN>


<ASSOC_LIST_NAME> ::= <UUDN>

-----------                    SYNTAX


<TAPEOUT_STMT> ::= TAPEOUT(<FILE_BLOCK_NAME>,<PHYSICAL_BLOCK_NAME>
                  ",<PHYSICAL-BLOCK-NAME>" * ";<ASSOC-LIST_NAME>" )

-----------                    SEMANTICS

THE TAPFIN STATEMENT IS USED TO SPECIFY THAT THE TAPE DEVICE IS TO BE USED FO
INPUTTING THE DATA

-----------                    SEMANTICS

THE TAPEOUT STATEMENT IS USED TO SPECIFY THAT THE TAPE DEVICE IS TO BE USED F
OUTPUTTING DATA.
```

```
-----------                    SYNTAX


<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
                | _
                | #
                | $
                | &
```

```
-----------                    SEMANTICS
```

ALPHABETIC CHARACTERS DO NOT HAVE INDIVIDUAL MEANING.THEY ARE USED FOR FORM "NAMES".

```
----------                 SYNTAX


<STORRCD_STMT> ::= STORRCD(<RECORD_STRUCTURE_NAME> ",<UDDN>" )


<RCD_STRUCTURE_NAME> ::= <UDDN>


----------                 SEMANTICS
```

THE STORRCD STATEMENT IS USED TO DECLARE THAT THE OCCURRENCES OF A PARTICULAR RECORD ARE TO BE INPUT, OR OUTPUT ON A PARTICULAR DECVICE, OR THAT THEY ARE TO BE STORED PHYSICALLY, AND TO SPECIFY THE STORAGE DEVICE WHEN THIS IS TO DIFFER FROM THE CONVENTIONS OF THE DDL IMPLEMENTATION.

----------   SYNTAX

```
<TAPEIN_STMT> ::= TAPEIN(<FILE_BLOCK_NAME>,<PHYSICAL_BLOCK_NAME>
                   ",<PHYSICAL_BLOCK_NAME>" * ";<ASSOC_LIST_NAME>" )
```

```
<PHYSICAL_BLOCK_NAME> ::= <UUDN>
```

```
<ASSOC_LIST_NAME> ::= <UUDN>
```

----------   SYNTAX

```
<TAPEOUT_STMT> ::= TAPEOUT(<FILE_BLOCK_NAME>,<PHYSICAL_BLOCK_NAME>
                    ",<PHYSICAL-BLOCK-NAME>" * ";<ASSOC-LIST_NAME>" )
```

----------   SEMANTICS

THE TAPEIN STATEMENT IS USED TO SPECIFY THAT THE TAPE DEVICE IS TO BE USED FOR
INPUTTING THE DATA

----------   SEMANTICS

THE TAPEOUT STATEMENT IS USED TO SPECIFY THAT THE TAPE DEVICE IS TO BE USED FO
OUTPUTTING DATA.

```
----------                    SYNTAX


<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
                   | _
                   | #
                   | $
                   | &


----------                    SEMANTICS


ALPHABETIC CHARACTERS DO NOT HAVE INDIVIDUAL MEANING.THEY ARE USED FOR FORMING
"NAMES".
```

```
----------                    SYNTAX


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

----------                    SEMANTICS
```

DIGITS ARE USED FOR FORMING NUMBERS AND "NAMES".

```
----------                      SYNTAX


<RELATIONAL_OPERATOR> ::= LT
                        | LE
                        | GT
                        | GE
                        | EQ
                        | NE


<ARITHMETIC_OPERATOR> ::= +
                        | -
                        | *
                        | /


<LOGICAL_OPERATOR> ::= AND
                     | OR
                     | NOT


<SEPARATOR> ::= ,
              | ;
              | :


----------                      SEMANTICS
```

DELIMETERS HAVE A FIXED MEANING WHICH FOR THE MOST OF THE PART IS OBVIOUS OR ELSE WILL BE GIVEN AT THE APPROPIATE PLACE IN THE SEQUEL.
TYPOGRAPHICAL FEATURES SUCH AS BLANK SPACE OR CHANGE TO A NEW LINE HAVE NO SIGNIFICANCE IN THE REFERENCE LANGUAGE.THEY MAY,HOWEVER,BE USED FREELY FOR FACILITATING READING.

```
----------                    SYNTAX


<INTEGER> ::= <DECIMAL-DIGIT>
            | <INTEGER><DECIMAL-DIGIT>


<SIGNED_INTEGER> ::= +<INTEGER>
                   | -<INTEGER>

----------                    SEMANTICS
```

INTEGERS HAVE THEIR CONVENTIONAL MEANING.

---------- SYNTAX

USER DEFINED NAME WILL BE ABBREVIATED UDN.
UNINDEXED USER DEFINED NAME WILL BE ABBREVIATED UUDN.
INDEXED USER DEFINED NAME WILL BE ABBREVIATED IUDN.

```
<SINGLE_STRING> ::= <ALPHA_CHAR>
                  | <DECIMAL_DIGIT>


<UDN> ::= <ALPHA_CHAR>
        | <UDN><SINGLE_STRING>


<UUDN> ::= '<UDN>'


<INDEX> ::= (<INTEGER>)


<IUDN> ::= '<UDN><INDEX>'
```

---------- SEMANTICS

USER DEFINED NAMES,UNINDEXED USER DEFINED NAMES AND INDEXED USER DEFINED NAMES HAVE NO INHERENT MEANING,BUT SERVE FOR THE IDENTIFICATION OF FIELDS,CHARACTERS, RECORDS,GROUPS,LINKS,ETC.
THE SAME NAME CANNOT BE USED TO DENOTE TWO DIFERENT"QUANTITIES" EXCEPT WHEN THESE QUANTITIES HAVE DISJOINT SCOPES(SEE REFERENCE NAMES).

```
-----------                   SYNTAX


<REF_NAME> ::= <UUDN> "   <REF_NAME>"
             | <IUDN> "   <REF_NAME>"


-----------                   SEMANTICS
```

REFFRENCE NAMES MUST START WITH THE NAME OF A FIELD;SUCCEEDING NAMES MUST BE NAMES OF GROUPS,OR,IN THE CASE OF THE FINAL NAME,THE NAME OF A RECORD.THERE MAY BE NO "HOLE " IN THE SAME SPECIFICATION THAT IS,EACH NAME USED IMMEDIATELY PRECEDE THE NAME OF THE GROUP TO WHICH IT IS IMMEDIATLY INFFRIOR.FOR EXAMPLE,IF FIELD 'A' IS A MEMBER OF GROUP 'B' WHICH IS A MEMBER OF RECORD 'C' THEN: 'A'OF'C' IS NOT PERMITTED.

```
<DATA_NAME> ::= <REF_NAME>


<FULL_CHAR_SET> ::= ','
                  | ;
                  | :
                  | (
                  | )
                  | '
                  | .
                  | %
                  | |
                  | !
                  | ?
                  | ¬
                  | V
                  | ^
                  | <ALPHA_CHAR>
                  | <DECIMAL_DIGIT>


<CONCODE_STATEMENT> ::= ;<CONCODE_STMT>
                      | <CONCODE_STATEMENT>;<CONCODE_STMT>


<CRITERION_NAME> ::= <UUDN>


<LINK_NAME> ::= <UUDN>


<NAME> ::= <UUDN>
```

APPENDIX B

EBNF WITH SUBROUTINE CALLS

The following is an example of the Associate statement described in EBNF with subroutine calls.

```
< ASSOCIATE_STMT > ::= ASSOCIATE/.SASSO/(< ASSOCIATE_LIST >
                       ";/.S#ASSO/< CRITERION_NAME >//")
< ASSOCIATE_LIST > ::= /.SASSOLIST/(< ASSOCIATE_PAIR >)//
                       ",/.S#ASSO1/(< ASSOCIATE_PAIR >)//"*
< ASSOCIATE_PAIR > ::= /.SASSOPAIR/ < TARGET_NAME >//, < SOURCE_NAME >
< TARGET_NAME > ::= < REF_NAME >
< SOURCE_NAME > ::= /.SASSOSRCE/< REF_NAME >/.SASSREF/
                  | < COMBINE_STMT >/.SASSCOMB/
                  | < DISJOIN_STMT >/.SASSDISJ/
```

APPENDIX C

PART I

SYMBOL TABLE

C.1 Symbol Table

Symbols will be up to 12 characters long, and will be encoded uniquely in four 1/2 words. The encoding algorithm is presented in Part II of this appendix. Because of this encoding technique, the character set allowable in user defined names may only contain the 40 symbols described in Appendix

The symbol table is a doubly-chained balanced tree with filial set size varying between 3 and 6 (see Figure C-1).

Each node is composed of three pointers:

1) a pointer to the key (the keys are in separate storage area);
2) a pointer to the first member of the filial set of which this node is a parent;
3) a pointer to the next member of the filial set of which this node is a member.

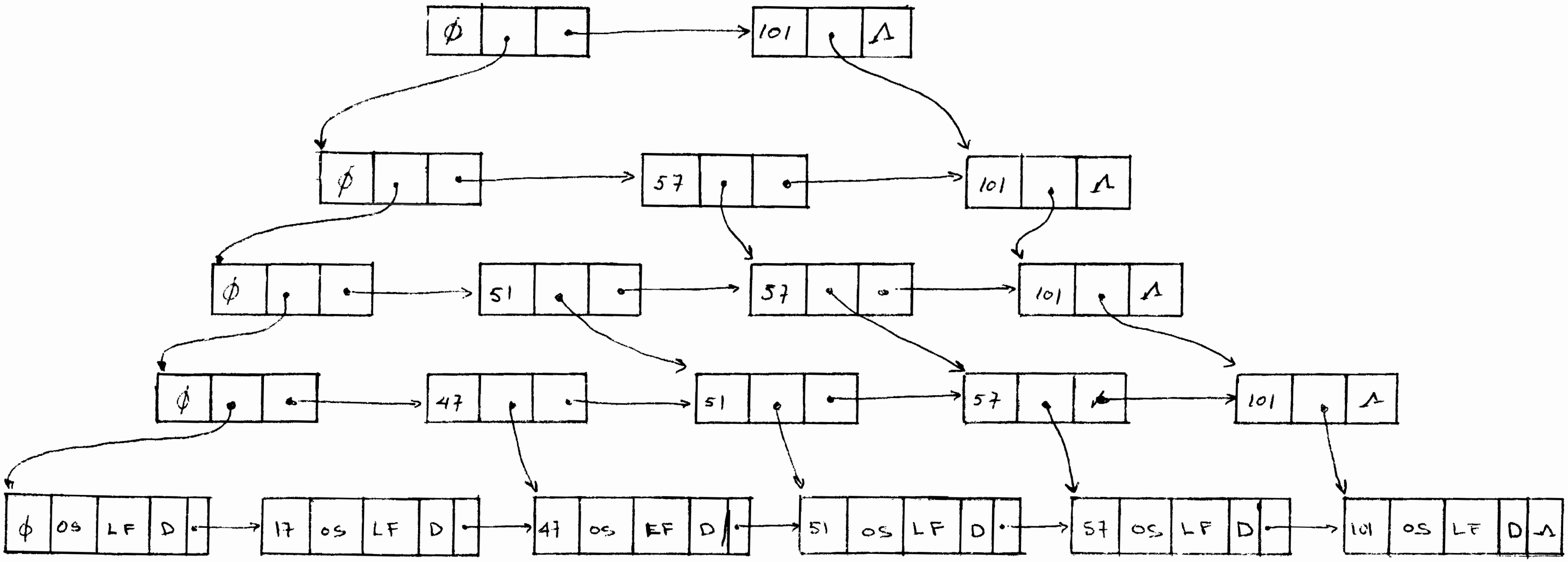
C.1.1 Format of Symbol Table Entries

The symbol table tree index entries (all those above the bottom level of the hierarchy shown in Figure C-1) have the following format:

pointer to symbol	pointer to next lower level	pointer to next at same level

The "pointer to symbol" points to the 3 word encoded symbol. The other two pointers point to other symbol table index entries.

Example of a "Symbol Tree"



Note: 1. The value of the key is shown where the pointer to the 'key' normally appear.

2. The fields shown as 'D' contain a pointer to the data table entry for that symbol (see Appendix D.1).

3. The fields shown as 'LF' contain a pointer to the last Father.

4. The fields shown as 'OS' contain an offset in the Inverted Symbol Table.

Figure C-1

The data pointer entries (those at the base of the hierarchy) have the following format:

pointer to symbol	offset	Last Father	Data Table Index	next to same level

The "offset" is the location in an inverter index for the Symbol Table where the address of this entry is stored.

The "Data Table Index" contains the relative address in the Data Table Index where the address of the Data Table entry for this symbol is located.

The "pointer to next at same level" points to the next data pointer entry.

The "pointer to last father" is used to hold the pointer to the data table entry for the most recently declared group to which the entry named by this symbol belongs. Only the last group need be pointed to because all previous groups which declared the present field or group to be a member, have been linked together in a LIFO list, i.e. one chained to the most recent group. In phase 1A of the compiler the "last father" pointer is moved to the "sup. pointer" of the data table entry associated with this symbol. (The reason for this somewhat complicated technique is that a field or group may be declared to be a member of one or more groups before it has been declared - therefore before it has a data table entry in which to place the sup. pointer. The symbol table is just a convenient place to save the pointer until it is certain that the member has been defined.)

C.1.2 Symbol Table Entry Addressing

To allow symbol table pointers to be changed to Data Table pointers (at the end of phase 1) without changing the size of the pointer, the symbol table entries are addressed in the same way as the Data Table entries - an inverted index of the data pointer entries is created as they are added to the symbol table. The "symbol table entry number" is then used as an offset in the inverted index to access the core address of the data pointer entry. Figure C-2 shows a symbol table with its inverted index.

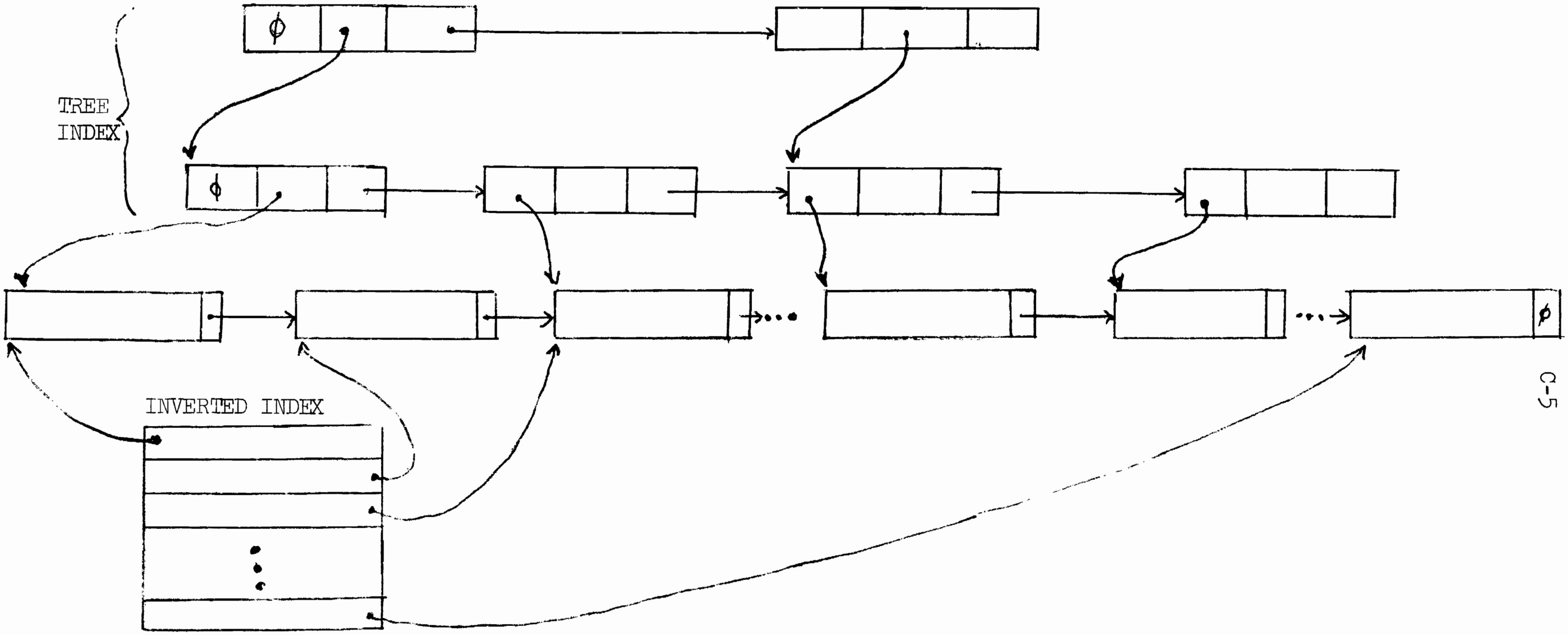


Figure C-2

Symbol Table Index

APPENDIX C

PART II

ALGORITHM TO ENCODE AND DECODE THREE 8-BIT CHARACTERS IN 16 BITS

(1) The allowable characters must be a subset of the full character set of the machine if the machine character set contains more than 40 characters.

(2) There must exist a 1 to 1 function from the selected character set to the set {0,1,2...39}. Call this function MAP The inverse function is IMAP.

(3) The Encoding algorithm is:

C_1, C_2, C_3 are the three characters to be encoded.

```
        $T_1 = MAP(C_1)$
        $T_2 = MAP(C_2)$
        $T_3 = MAP(C_3)$
        $SUM = T_1$
        $SUM = SUM*40$
        $SUM = SUM+T_2$
        $SUM = SUM*40$
        $SUM = SUM+T_3$
```

SUM contains the Encoded characters in 16 bits.

(4) The Decoding algorithm is:

```
        $T_1 = QUOTIENT\ [SUM/(40)^2]$
        $R_1 = REM\ [SUM/(40)^2]$
        $T_2 = QUOTIENT\ [R_1/40]$
        $T_3 = REM\ [R_1/40]$
        $C_1 = IMAP\ (T_1)$
        $C_2 = IMAP\ (T_2)$
        $C_3 = IMAP\ (T_3)$
```

APPENDIX C

PART III

SEARCH TREE ALGORITHM

(PTR is the current entry pointer. PPTR is the prior entry. IPTR points to the last index entry used.)

Step 1 - [Initialize Pointers]

Set IPTR=Ø. Set PPTR = top of tree. Set PTR = top of tree. Set CNT = Ø. Set LEVEL = highest tree level number.

Step 2 - [Compare Key]

If KEY of the entry pointed to by PTR is equal to the key being searched for, go to Step 4. If greater go to step 5. Else (it is less)

Step 3 - [Step to Next Entry]

Save PTR in PPTR. If the next pointer is zero go to step 5. Else, step PTR to next entry at this level. Increment CNT (the count of the number of entries searched at this level). If CNT=6 go to step 7. Else go to step 2.

Step 4 - [Get Lowest Level]

If LEVEL = 0 exit indicating KEY found. If not set PTR to the next lower level (using the downward pointer in the entry pointed to by PTR). Decrement LEVEL. Go to Step 4.

Step 5 - [Decrement Level Number]

Decrement level number by 1. If less than zero exit indicating not found. Else save PTR in IPTR.

Step 6 - [Set Pointer to Next Level]

Using the downward pointer of the entry pointed to by PPTR step down one level. Clear CNT. Go to Step 3. (Note: The key on the first entry need not be tested since it is the same as the one in the index entry above it.)

Create Index Entry Starts Here

Step 7 [Add New Level to Tree if Needed]

If IPTR≠0 go to step 8. Else increment the "highest level" number. Get memory for new index entry. Set its downward pointer with the old top of tree pointer. Set its next pointer to ∅. Set its key to ∅. Reset the top of tree pointer to point to this new entry. Set IPTR to the new entry.

Step 8 - [Find the 4th Entry]

Using IPTR to access the first of the entries at the level now being searched. Step to the 4th entry at this level. Save the pointer to this entry in TMP.

Step 9 - [Set up New Index Entry]

Set the next pointer of the new entry with the next pointer of the entry pointed to by IPTR. Set the next entry pointer of the entry pointed to by IPTR, to the new entry. Set the downward pointer of the new entry with TMP. Set the key of the new entry with the key pointed to by the TMP entry.

Step 10 - [Prepare to Return to Search]

Set IPTR to the new entry. Set CNT = 3. Go to Step 2.

APPENDIX C

PART III

INSERT NEW ENTRY

Step 1 - [Determine if Key Already in Tree]

Call Search. If found exit with error indicator.

Step 2 - [Link New Entry]

Use PTR, and PPTR as set by S arch. Set the next pointer of the entry pointed to by PPTR with the address of the new entry. Set the next pointer of the new entry with PTR. Exit.

APPENDIX D

DATA TABLE

The Data Table[*] is a core-resident file containing variable length entries and an inverted index (see Figure D-1). Each entry has a TYPE indicator as its first component. The types and the corresponding codes are shown below:

1	FIELD
2	CHAR
3	GROUP
4	RECORD
5	LENGTH
6	COUNT
7	EXIST
8	CONSTANT
9	BLOCK
10	RPLVAL
11	CONCODE
12	SET
13	CRITERION
14	CRITEX
15	TAPEIN
16	TAPEOUT
17	STORFILE
18	CARDIN
19	STORRCD
20	LINK
21	SEQUEN
22	ASSOC
23	CREATE
24	COMBINE
25	DISJOIN
26	BBLOCK
27	CARDOUT

Although each type has its own format certain conventions apply to all. These are:

1. There are 4 basic types of parameter fields, Binary (B), Integer (I), Short Integer (SI), Pointer (P). Binary parameters can take on two values (e.g. "F" and "V"). Binary parameters are grouped 8 to a byte. Integer parameters can have values in the range 0-32K. They occupy 2 bytes. Short Integers occupy one byte and can take a value in the range 0-255. Pointers are of three kinds. Pointers to

[*] The Data Table entry formats appear in Part I of this appendix.

auxiliary descriptor block entries, pointers to Data Table entries, and, during the 1st phase of compilation, pointers to Symbol Table entries.*

2. To distinguish the different kinds of parameters where more than one type may be used in a field, they are encoded in the following way:

Value	Meaning
0	Binary Value ∅
1-32K	Integer Value
(32K+1) to (32K+256)	Auxiliary descriptor entries 1-256
(32K-257) to (48K-1)	Data table entries 1 to (16K-257)
(48K) to (64K-1)	Symbol table entries 1 to (16K-1)
64K	Binary Value 1

* The need to distinguish data table pointers from symbol table pointers arises from the fact that not all pointers used in phase 1 are symbol table pointers. The exceptions are the "unlabeled" statements like CONCODE; the pointers to these are data table pointers, right from the time they are created.

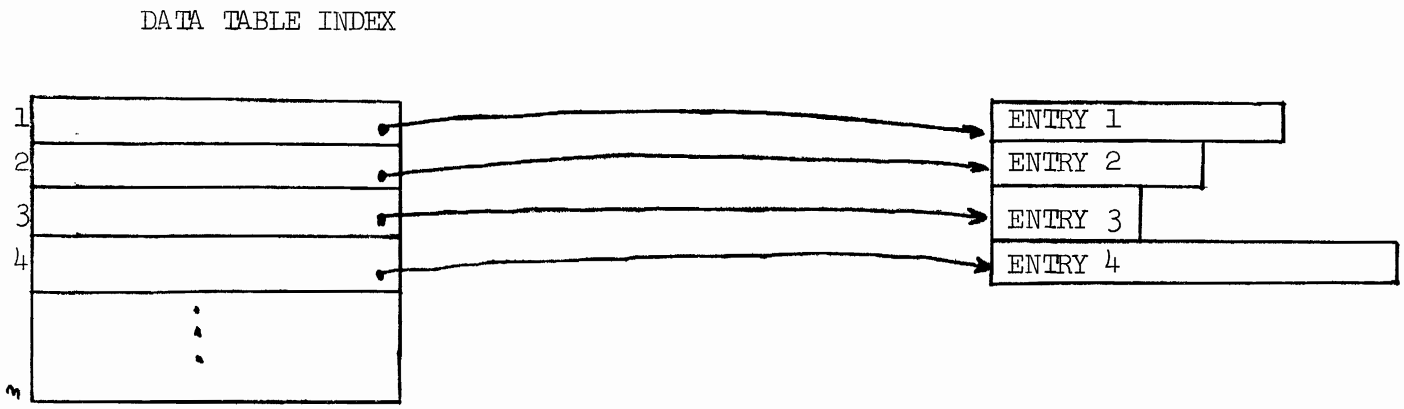


Figure D-1

Data Table Index

APPENDIX D

PART I

DATA TABLE FORMATS

DATA TABLE ENTRY FOR CREATE STMT

SI	P	SI	P		P	SI	P		P
TYPE	FILE NAME	COUNT	FILE LIST	...	FILE LIST	COUNT1	ASSO LIST	...	ASSO LIST

TYPE - (Short integer), field contains code associated to the CREATE stmt

FILE NAME - (Pointer), points to Data Table entry corresponding to a STORFILE stmt

COUNT - (Short integer), Number of File Lists in the entry

FILE LIST - (Pointer), points to Data Table corresponding to a STORFILE stmt

COUNT1 - (Short integer), number of Asso lists in the entry

Asso List - (pointer), points to Data Table entry corresponding to an ASSOCIATE stmt.

DATA TABLE ENTRY FOR BLOCK STMT

SI	B	SI	P	B	I	B	P		P	B	I	B	P
TYPE	ORDER	COUNT	RCD NAME	OPT	REP NUMBER	FLAG	CRIT	...	RCD NAME	OPT	REP NO	FLAG	CRIT
			← bblock list →						← bblock list →				

ORDER (Binary) 1 = SPEC, ∅ = NOORD

COUNT (Short integer), number of BBLOCK lists that follow

RCD NAME
OPT
REP NUMBER } same as for the BBLOCK entry
FLAG
CRIT

DATA TABLE ENTRY FOR BBLOCK STMT

									←	—	bblock list	—	→
SI	B	I	B	B	I	I	B	SI	P	B	I	B	P
TYPE	UNIF	LENGTH	RCD DIST	RCD COUNT UNIF	RCD COUNT	BASIC BLK COUNT	RCD ORDER	COUNT	RCD NAME	OPT	REP NUMBER	FLAG	CRIT

:

P	B	I	B	P
RCD NAME	OPT	REP NUMBER	FLAG	CRIT

|← bblock list →|

TYPE - (short integer), field contains the code for the BBLOCK stmt.

UNIF - (Binary), 1 = FIXED, Ø = variable

LENGTH - (integer), integer, 32K = NOLIM

RCD DIST - (Binary), 1 = WHOLE, Ø = SPLIT

RCD COUNT UNIF - (Binary), 1 = FIXED, Ø = variable

RCD COUNT - (integer), integer, 32K = NOLIM

BASIC BLK COUNT - (integer), integer, 32K = NOLIM

RCD ORDER - (Binary), 1 = SPEC, Ø = NOORD

COUNT - (short integer), number of bblock list in entry

RCD NAME - (pointer), points to Data Table entry for the RECORD stmt.

OPT - (binary), 1 = Mandatory, Ø = optional

REP NO - (I), integer, 32K = NOLIM

FLAG - (binary) if = 1 indicates field CRIT is present in the entry

CRIT - (pointer) points to Data Table entry for the CRITERION stmt.

DATA TABLE ENTRY FOR ASSOCIATE STMT

SI	SI	P	P		P	P	B	P
TYPE	COUNT	TARGET	SOURCE	° ° °	TARGET	SOURCE	FLAG	CRIT

TYPE - (short integer), field have the code for the associate stmt.

COUNT - (short integer), no. of associate pairs present in the stmt.

TARGET - (pointer), points to an entry in the Data Table.

SOURCE - (pointer), points to an entry in the Data Table.

FLAG - (Binary), if = 1 indicates that the CRIT field is present.

CRIT - (pointer), points to an entry in the Data Table, corresponding to a <u>CRITERION</u> <u>STMT</u> associated to this entry.

DATA TABLE ENTRY FOR RECORD STMT

SI	P	P	P	B	SI	P	P		P
TYPE	GROUP	SELF PTR	NEXT GROUP	FLAG	COUNT	CRIT	CONCODE	...	CONCODE

TYPE - (Short integer), field contains the code corresponding to the RECORD stmt.

GROUP - (Pointer), points to Data Table entry for the group which contains the data of this record.

SELF PTR - (Pointer), points to the beginning of this entry.

NEXT GROUP - (Pointer), same as for Group entry. This field links the records or groups to which the member of this record also belongs.

FLAG - (Binary), If = 1; indicates that the CRIT field is present.

COUNT - (Short integer), contains the number of CONCODE fields that follow.

CRIT - (Pointer), points to Data Table entry for the CRITERION stmt that determines whether this record is acceptable.

CONCODE - (Pointer), points to Data Table entries for the Concode stmt associated with this record.

DATA TABLE ENTRY FOR CARDIN STMT

SI	P	B	P
TYPE	BBLOCK	FLAG	ASSOC

DATA TABLE ENTRY FOR CARDOUT STMT

SI	P	B	P
TYPE	BBLOCK	FLAG	ASSOC

TYPE - (Short integer), field contains the code for the CARDIN STMT (CARDOUT STMT)

BBLOCK - (pointer), points to Data Table entry corresponding to the BBLOCK stmt

FLAG - (Binary), if = 1 indicates that assoc field is present

ASSOC - (pointer), points to Data Table entry for the association stmt.

DATA TABLE ENTRY FOR CRITERION STMT

SI	P	P
TYPE	CRIT	RPLVAL

TYPE - (Short integer), field contains code associated to the CRITERION stmt.

CRIT - (pointer), points to Data Table entry corresponding to a CRITEX stmt.

RPLVAL - (pointer), points to Data Table entry corresponding to a RPLVAL stmt.

DATA TABLE ENTRY FOR CHAR STMT

SI	SI	B P	P
TYPE	SIZE	DESC	DESC'

TYPE - (Short integer), field contains the code associated to the CARDIN stmt.

SIZE - (Short integer), integer

DESC - (Binary), if binary $\left\{\begin{array}{l}\text{0 - BCD}\\ \text{1 - EBCDIC}\end{array}\right.$

(Pointer), if pointer points to Data Table entry corresponding to a SET stmt.

DESC' - (Pointer), if pointer points to Data Table entry corresponding to a SET stmt.

DATA TABLE ENTRY FOR COMBINE STMT

SI	P	B	SI	B	P
TYPE	SOURCE NAME	COUNT TYPE	COUNT	FLAG	CRIT

TYPE - (Short integer), field contains the code for the COMBINE stmt.

SOURCE NAME - (Pointer), points to a Data Table entry.

COUNT TYPE - (Binary), $\begin{cases} 0 - F \text{ (FIXED)} \\ 1 - V \text{ (VARIABLE)} \end{cases}$

COUNT - (Short integer) - 32K = NOLIM, else integer

FLAG - (Binary), if = 1 indicates that CRIT field is present

CRIT - (pointer), points to Data Table entry corresponding to a CRITERION stmt.

DATA TABLE ENTRY FOR CONSTANT STMT

SI	SI	character	B P
TYPE	LENGTH	STRING	DATA TYPE

TYPE - (Short integer), field contains the code associated to the CONSTANT stmt.

LENGTH - (Short integer), length of the string of characters. 0 if it is an integer.

STRING - (Character), a tring of characters
(Integer), an integer

DATA TYPE - (Binary), if binary $\begin{cases} 0 - B \\ 1 - C \end{cases}$

(Pointer), if pointer; points to Data Table entry corresponding to a CHARACTER stmt.

DATA TABLE ENTRY OF DISJOIN STMT

SI	P	B	SI
TYPE	SOURCE NAME	COUNT TYPE	COUNT

TYPE - (Short integer), field contains the code for the DISJOIN stmt.

SOURCE NAME - (pointer), points to Data Table entry

COUNT TYPE - (Binary), {0 - F (Fixed)
1 - V (Variable)

COUNT - (Short integer) {32K - NOLIM
Integer

DATA TABLE ENTRY FOR FIELD STMT

SI	P	B	B	SI	B P	I P	B P	I P	P	P		P
TYPE	GROUP POINTER	FLAG1	FLAG2	COUNT	LENGTH TYPE	LENGTH	DATA TYPE	ALLIGN FACTOR	PAD	CONCODE	• • •	CONCODE

TYPE - (Short integer), field contains the code associated to the FIELD stmt.

GROUP POINTER - (pointer), points to "self-PTR' field of Data Table entry for the first group of which this field is a member

FLAG1 - (Binary), Uniformity (1 = FIXED, ∅ = Variable)

FLAG2 - (Binary), Orientation (1 = Right, ∅ = Left)

COUNT - (Short integer), number of CONCODE fields in the entry

LENGTH TYPE - {(Binary, if binary 1 = Bits, ∅ = character
Pointer, if pointer, it points to Data Table entry of a CHAR stmt

LENGTH - {Integer, it is the length of the field (32K = NOLIM)
Pointer, it points tc Data Table entry (or compound Data Table entry name)

DATA TYPE (same as length type)

ALLIGN FACTOR - {integer - is the number of lists between boundaries, ∅ = No Boundary
pointer - points to the field entry for the field containing the allignment factor

PAD - (pointer), points to Data Table entry for the constant stmt which specifies the pad character

CONCODE - (Pointer), points to Data Table entry for the CONCODE stmt associated with this entry.

DATA TABLE ENTRY FOR GROUP STMT

GROUP STMT

				GROUP-MEMBER									
SI	P	B	SI	P	P	P	B	B	B	I P	B P	B	P
TYPE	SUP POINTER	FLAG1	COUNT	MEMBER POINTER	SELF PTR	NEXT GROUP	IND1	IND2	IND3	NUMB	ORDER	IND4	CRIT

⋮

P	P	P	B	B	B	I P	P	B	P
MEMBER POINTER	SELF PTR	NEXT GROUP	IND1	IND2	IND3	NUMB	ORDER	IND4	CRIT
GROUP MEMBER									

SUP POINTER - (Pointer), points to the "field" SELF PTR of Data Table entry of the group or Record of which this group is a member.

FLAG1 - (Binary) 1 = SPEC, ∅ = NOORD

COUNT - (SI) The number of times the group member repeats

MEMBER POINTER (pointer), points to Data Table entry of the subordinate Group or Field.

SELF-PTR - (Pointer), points to beginning of this group entry

NEXT GROUP - (Pointer), points to the Data Table entry (SELF PTR of the member descriptor) of the next group of which this "group member" is also a member.

IND1 (binary), OPTIONALITY - (1 = MANDATORY, ∅ = OPTIONAL)

IND2 (binary), UNIF - (1 = FIXED, ∅ = VARIABLE)

IND3 (binary), indicates whether ORDER has been specified (1 = yes, ∅ = no)

NUMB - (pointer) - points to auxiliary descriptor entry which contains the value
(integer) - it is the repetition factor of the member. BLK = NOLIM

ORDER - (binary, 1 = ASCLEX, ∅ = DSCLEX
Pointer, points to Data Table entry for the LINK stmt.

IND4 - indicates if CRIT is present (1 = yes, ∅ = no)

CRIT - pointer to Data Table entry for the CRITERION stmt which determines whether a group element is acceptable.

DATA TABLE ENTRIES FOR LENGTH, CNT and EXIST STMT

SI	P	B
TYPE	NAME	TYPE1

TYPE - (Short integer), field contains the code corresponding to a {LENGTH stmt
or EXIST stmt
or CNT stmt

NAME - (pointer), pointer to a Data Table entry

TYPE1 - (Binary), {0 - B (only for the length stmt)
{1 - C

DATA TABLE ENTRY FOR LINK STMT

			B		
SI	P	P	P	B	SI
TYPE	RECORD SOURCE	RECORD TARGET	CRIT	UNIF	LINK NUMBER

TYPE - (Short integer), field contains the code corresponding to the <u>LINK</u> stmt

RECORD SOURCE - (Pointer), points to Data Table entry corresponding to a <u>RECORD</u> stmt.

TARGET - (pointer), points to Data Table entry corresponding to a <u>RECORD</u> stmt.

CRIT - {(Binary), if binary - NOORD
(Pointer), if pointer, it points to Data Table entry corresponding to a <u>CRITERION</u> stmt.

UNIF - (Binary), {0 - F (FIXED)
1 - V (VARIABLE)

LINK NUMBER - (Short integer), no. of target occurrences that may be linked to each same occurrence of a <u>source</u> record.

DATA TABLE ENTRY FOR STORRCD STMT

SI	P	B	P
TYPE	STRUCTURE NAME	FLAG	NAME

TYPE - (Short integer), field contains the code associated to the STORRCD stmt.

STRUCTURE NAME - (Pointer), points to Data Table entry corresponding to a RECORD stmt.

FLAG - (Binary), if = 1 indicates the field name is present.

NAME - (Pointer), points to Data Table entry corresponding to a Device stmt.

DATA TABLE ENTRY FOR RPLVAL STMT

SI	P	P
TYPE	DATA NAME	RPL VALUE

TYPE - (Short integer), field has the code associated to a RPLVAL stmt.

DATA NAME - (Pointer), points to Data Table entry
(it can be for a CONSTANT stmt or a FIELD or GROUP stmt)

DATA TABLE ENTRY FOR SEQUEN STMT

SI	SI	P		P
TYPE	COUNT	LINK NAME	...	LINK NAME

TYPE - (short integer), field contains <u>code</u> corresponding to the <u>SEQUEN</u> stmt.

COUNT - (short integer), no. of link names in the entry

LINK NAME - (pointer), points to Data Table entry corresponding to a <u>LINK</u> stmt.

DATA TABLE ENTRY FOR SET STMT

SI	SI	P		P
TYPE	COUNT	MEMBER	...	MEMBER

TYPE - (Short integer) field contains the <u>code</u> corresponding to the <u>SET</u> stmt.

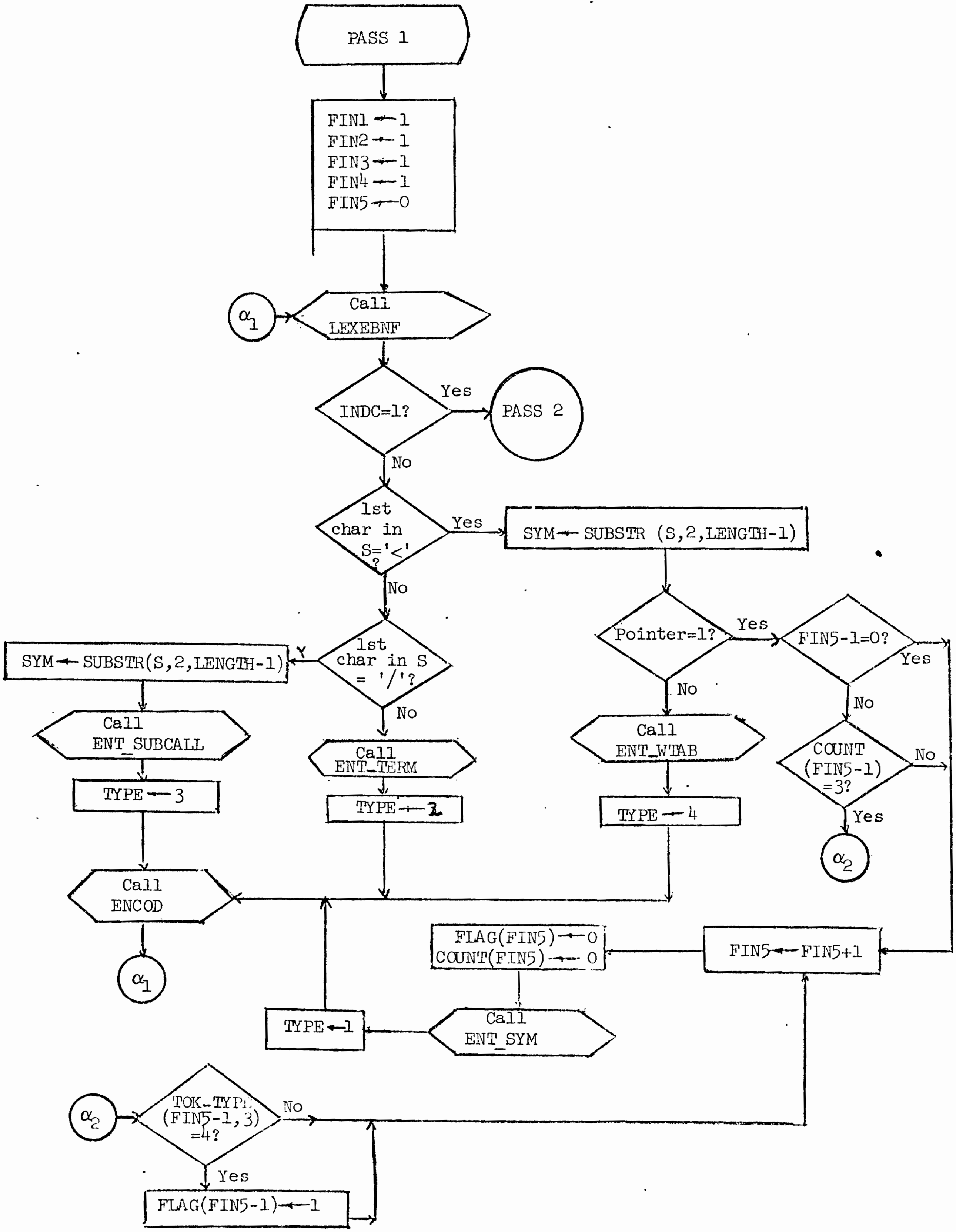
COUNT - (Short integer) number of members in the entry

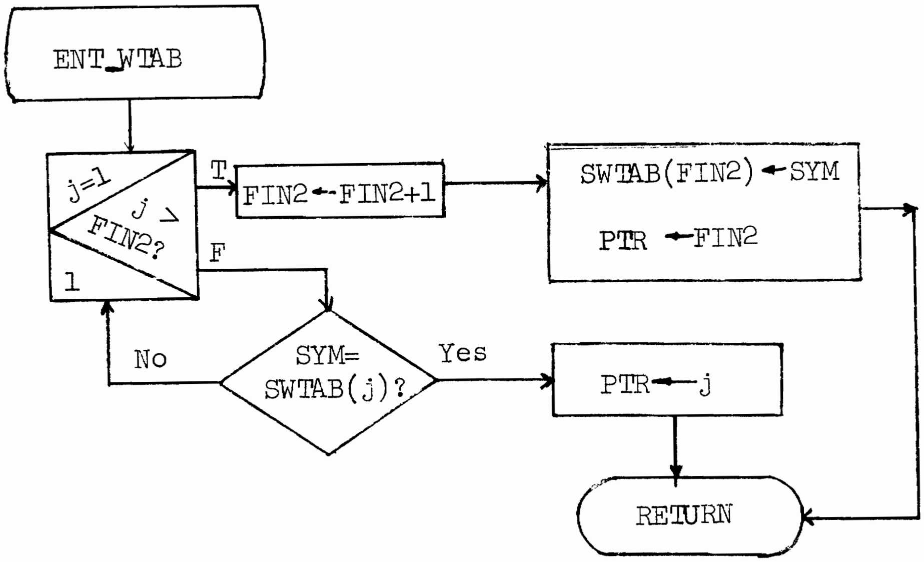
MEMBER - (Pointer), points to Data Table entry corresponding to a <u>CONSTANT</u> stmt.

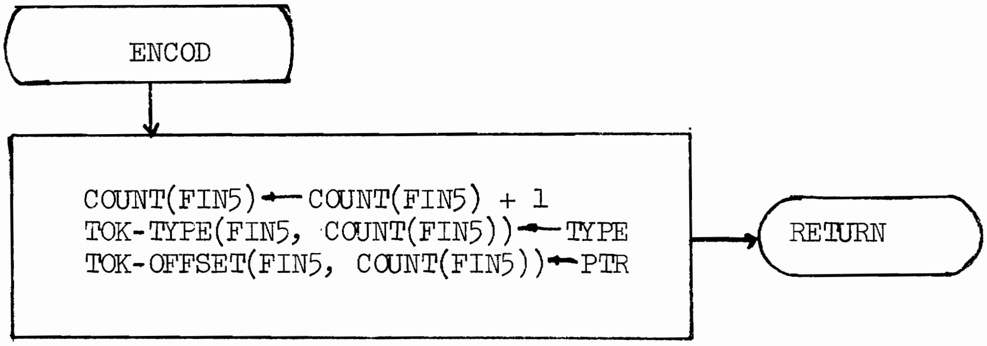
APPENDIX E

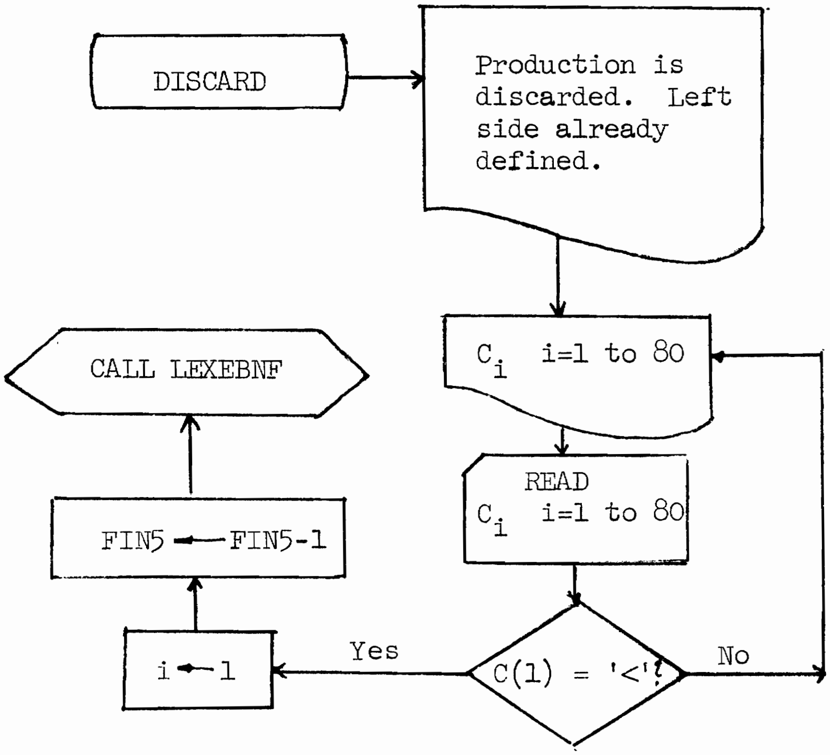
THE SAPG (Syntactic Analysis Program Generator)

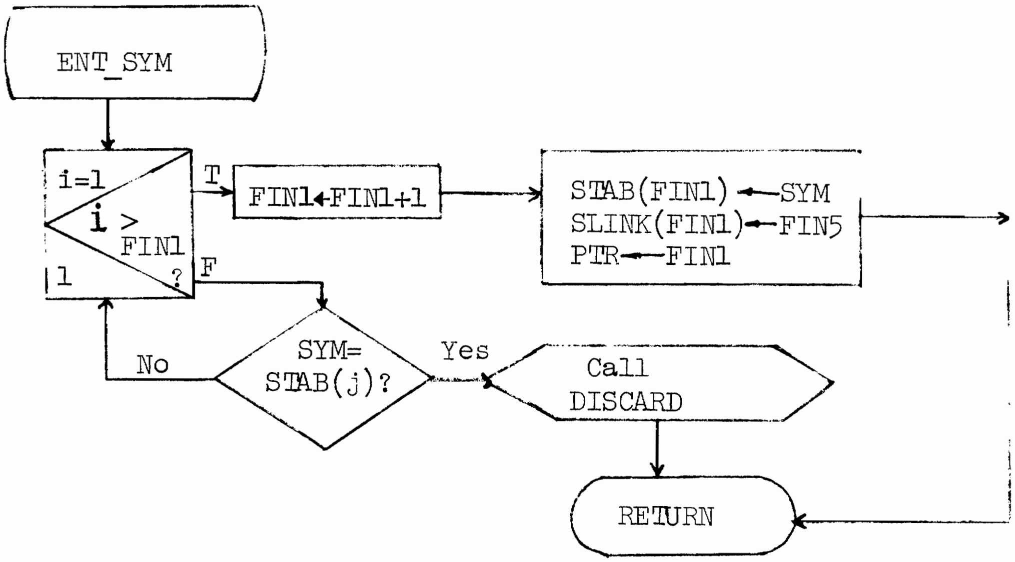
Part I - PASS 1

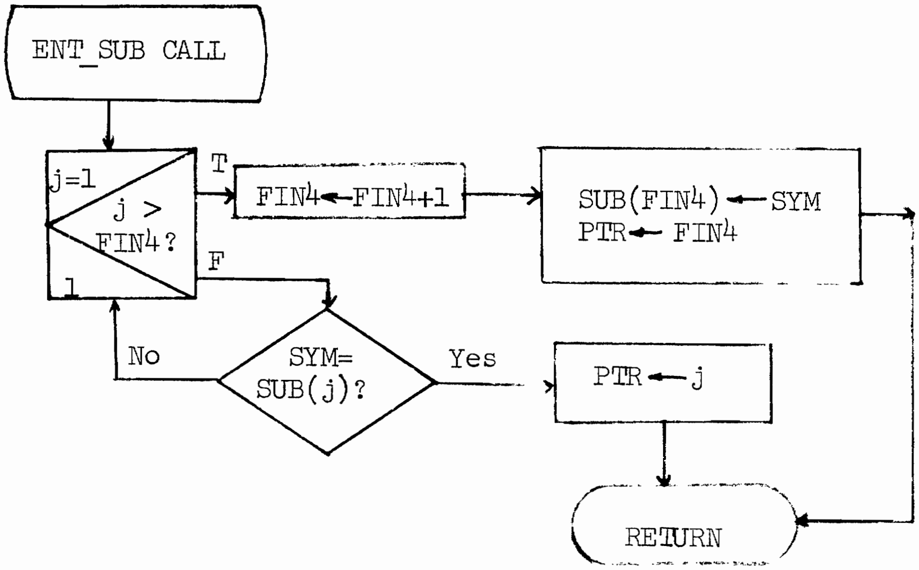


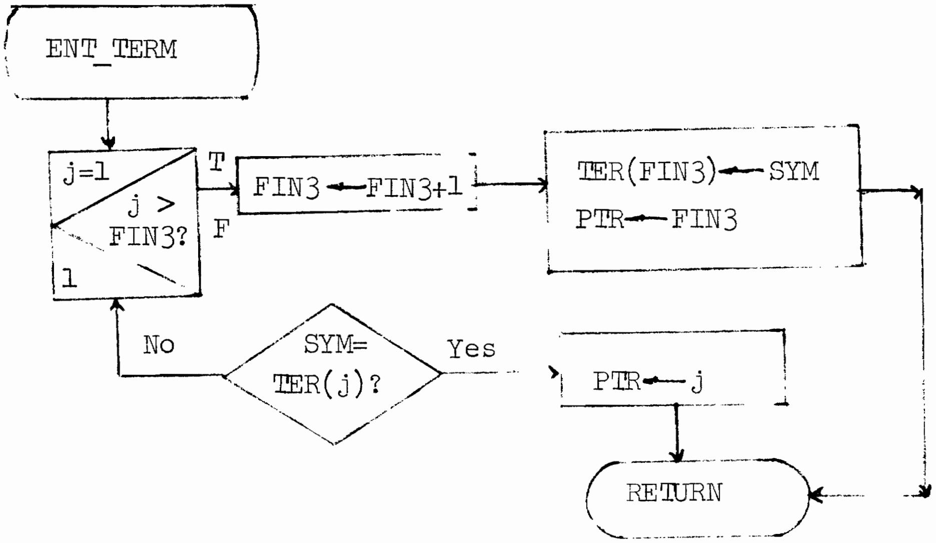

ENT_WTAB
j=1
j >
FIN2?
1
T
F
FIN2←FIN2+1
SWTAB(FIN2)←SYM
PTR ←FIN2
No
SYM=
SWTAB(j)?
Yes
PTR←j
RETURN

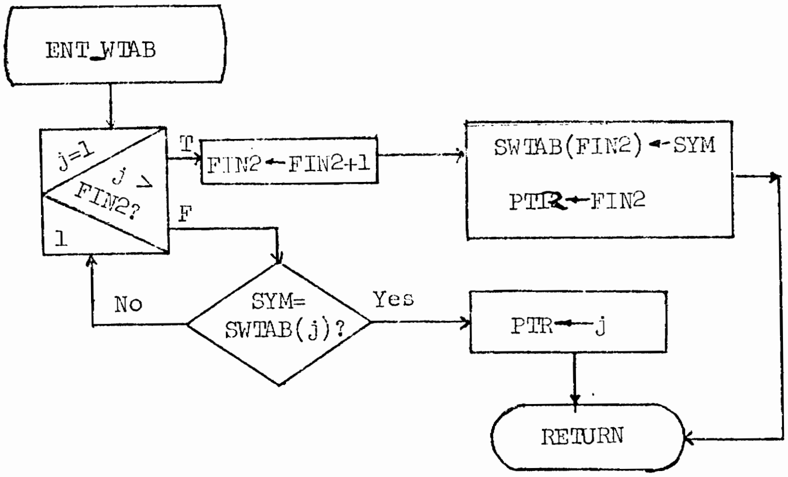

ENCOD
COUNT(FIN5)←COUNT(FIN5) + 1
TOK-TYPE(FIN5, COUNT(FIN5))←TYPE
TOK-OFFSET(FIN5, COUNT(FIN5))←PTR
RETURN

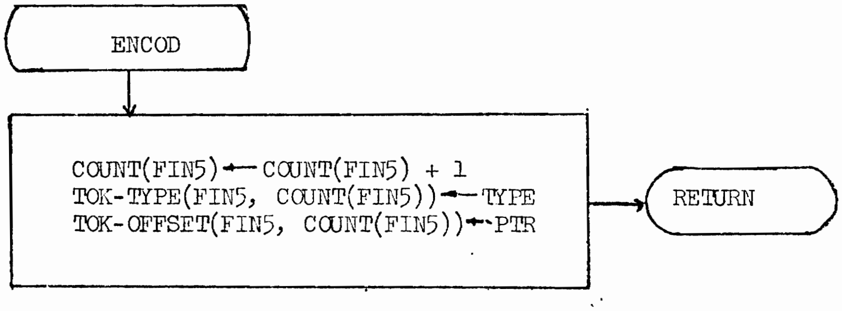

DISCARD
Production is
discarded. Left
side already
defined.
C_i i=1 to 80
READ
C_i i=1 to 80
C(1) = '<'?
Yes
No
i←1
FIN5←FIN5-1
CALL LEXEBNF

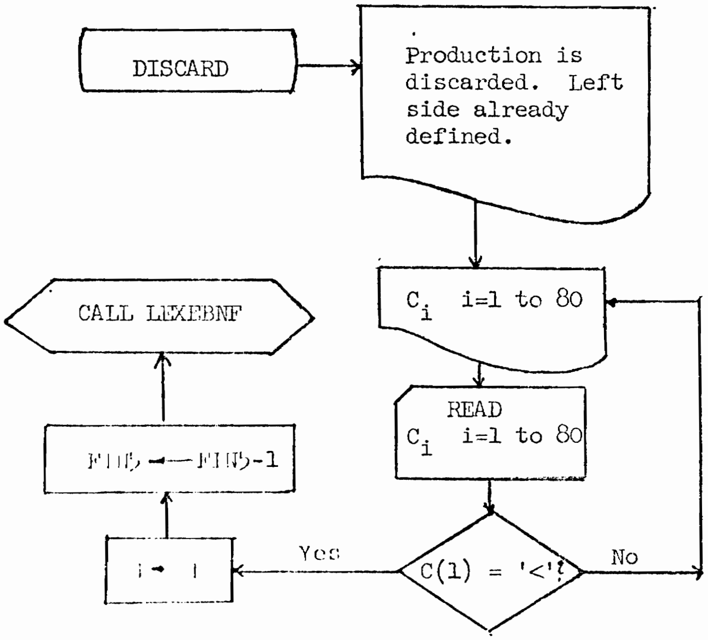

ENT_SYM
i=1
i > FIN1 ?
1
T
F
FIN1←FIN1+1
STAB(FIN1) ←SYM
SLINK(FIN1)←FIN5
PTR←FIN1
SYM= STAB(j)?
No
Yes
Call DISCARD
RETURN


ENT_SUB CALL
j=1
j > FIN4 ?
1
T
F
FIN4←FIN4+1
SUB(FIN4) ←SYM
PTR← FIN4
SYM= SUB(j)?
No
Yes
PTR←j
RETURN


ENT_TERM
j=1
j > FIN3?
1
T
F
FIN3 ←FIN3+1
TER(FIN3)←SYM
PTR←FIN3
SYM= TER(j)?
No
Yes
PTR←j
RETURN


ENT_WTAB
j=1
j >
FIN2?
1
T
F
FIN2←FIN2+1
SWTAB(FIN2)←SYM
PTR←FIN2
SYM=
SWTAB(j)?
No
Yes
PTR←j
RETURN


ENCOD
COUNT(FIN5)←COUNT(FIN5) + 1
TOK-TYPE(FIN5, COUNT(FIN5))←TYPE
TOK-OFFSET(FIN5, COUNT(FIN5))←PTR
RETURN


DISCARD
Production is
discarded. Left
side already
defined.
C_i i=1 to 80
READ
C_i i=1 to 80
C(1) = '<'?
Yes
No
i → 1
FIN5 ←— FIN5-1
CALL LEXEBNF

Part IA - LEXEBNF

LEXEBNF is a subroutine that will scan the EBNF statements and will return when called,the following items:

a) a syntactic unit

b) the length of the syntactic unit

c) the value of the pointer at the beginning of the scan

A syntactic unit can be:

1) a non-terminal symbol (i.e., name enclosed in < >)

2) a terminal symbol

a) Metalinguistic symbol, i.e., ::= | [ ][†] *

b) Keyword

c) Separator, i.e., : , ; ( )

3) a subroutine call (i.e., name enclosed between //)

To form a syntactic unit from single characters being scanned, a Decision Table (see Table E.1) has been constructed along with a set of functions to perform the formation of the syntactic units.

[†] Note that the [and] is in fact formed with two characters in the actual EBNF statements [≡ $1\overset{8}{2}$" and] ≡ "$1\overset{0}{2}$.

Table E.1 Decision Table

(This table is being realized as an array in PL/1 A(0:9,0:9))

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

* All blank boxes are invalid combination of characters (SE).

The mapping of characters is the following:

A B C ... **Z** 0 1 ... 9 - # & $. → 0

Space (blank) → 1

< → 2

> → 3

= → 4

" → 5

$\overset{0}{12}$ → 6

/ → 7

: → 8

( ) ; , * | → 9

The 8 functions are listed below, note that symbolic labels are being used. This is because the LEXEBNF will be coded in PL/1, therefore to achieve the appropriate branch to any of these functions a vector of labels is needed.

```
                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) = SE (syntactic error)
F1: S = S||LEFT;
    S = S||RIGHT;
    IF C(i+2) = '=' THEN DO;
                          S = S||C(i+2);
                          i = i+3;
                          LENGTH = LENGTH+3;
                          RETURN;
                          END;
                     ELSE GO TO ERROR 1;
    GO TO UNO;
```

```
F2: S = S||LEFT;
    i = i+1; LENGTH = LENGTH+1;
    GO TO UNO;
F3: S = S||LEFT;
    i = i+1; LENGTH = LENGTH+1;
    RETURN;
F4: S = S||LEFT;
    S = S||RIGHT;
    i = i+2; LENGTH = LENGTH+2;
    RETURN;
F5: SW1 = 1;
    S = S||LEFT;
    i = i+1; LENGTH = LENGTH+1;
    GO TO UNO;
F6: IF SW1 = 1 THEN DO; S = S||LEFT;
                        S = S||RIGHT;
                        i = i+2; LENGTH = LENGTH+2;
                        RETURN;
                        END;
                   ELSE DO; S = S||LEFT;
                        i = i+1; LENGTH = LENGTH+1;
                        RETURN;
                        END;
F7: i = i+1;
    GO TO UNO;
F8: i = i+2;
    GO TO UNO;
```

Note: The label UNO is the beginning of the loop in LEXEBNF in which a syntactic unit is being formed.

In order to speed up the process, the following translation technique is used: Given a single character, the corresponding value to look-up in the decision table is found using the mapping given in Table E.2. (To take advantage of the internal, EBCDIC, representation of characters in the system 360, a decimal equivalent is being used.)

The flowchart of the LEXEBNF is given in Figure E-1.

Character	Decimal Equivalent to EBCDIC	Decision Table Mapping
Space (blank)	64	1
.	75	0
<	76	2
(	77	9
|	79	9
&	80	0
$	91	0
*	92	9
)	93	9
;	94	9
/	97	7
,	107	9
-	109	0
>	110	3
:	122	8
#	123	0
=	126	4
"	127	5
$1\overset{0}{2}$	192	6
A → I	143 → 201	0
J → R	209 → 217	0
S → Z	226 → 233	0
0 - 9	240 - 249	0

Table E.2

Mapping of Character to Decimal Equivalent in Storage

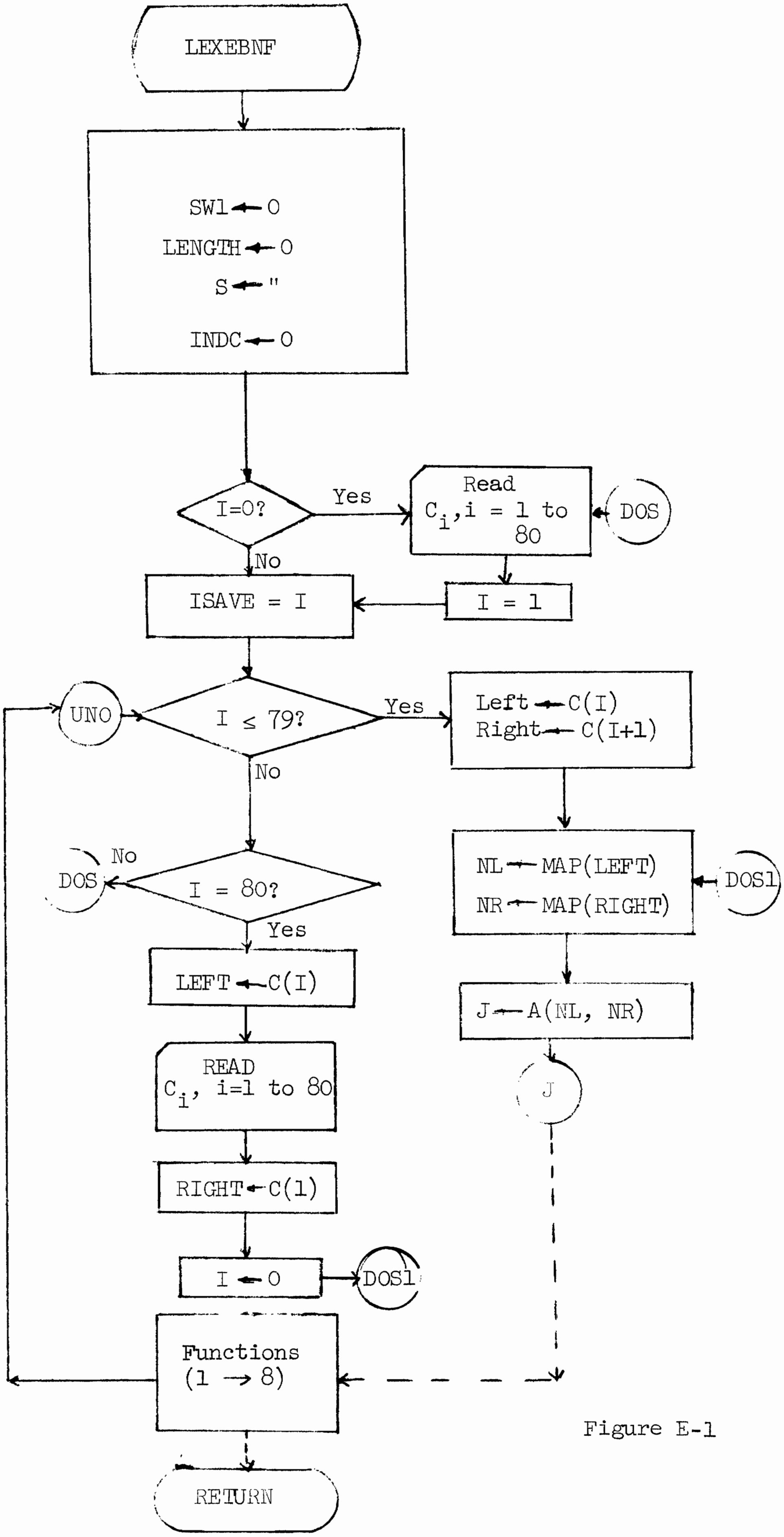
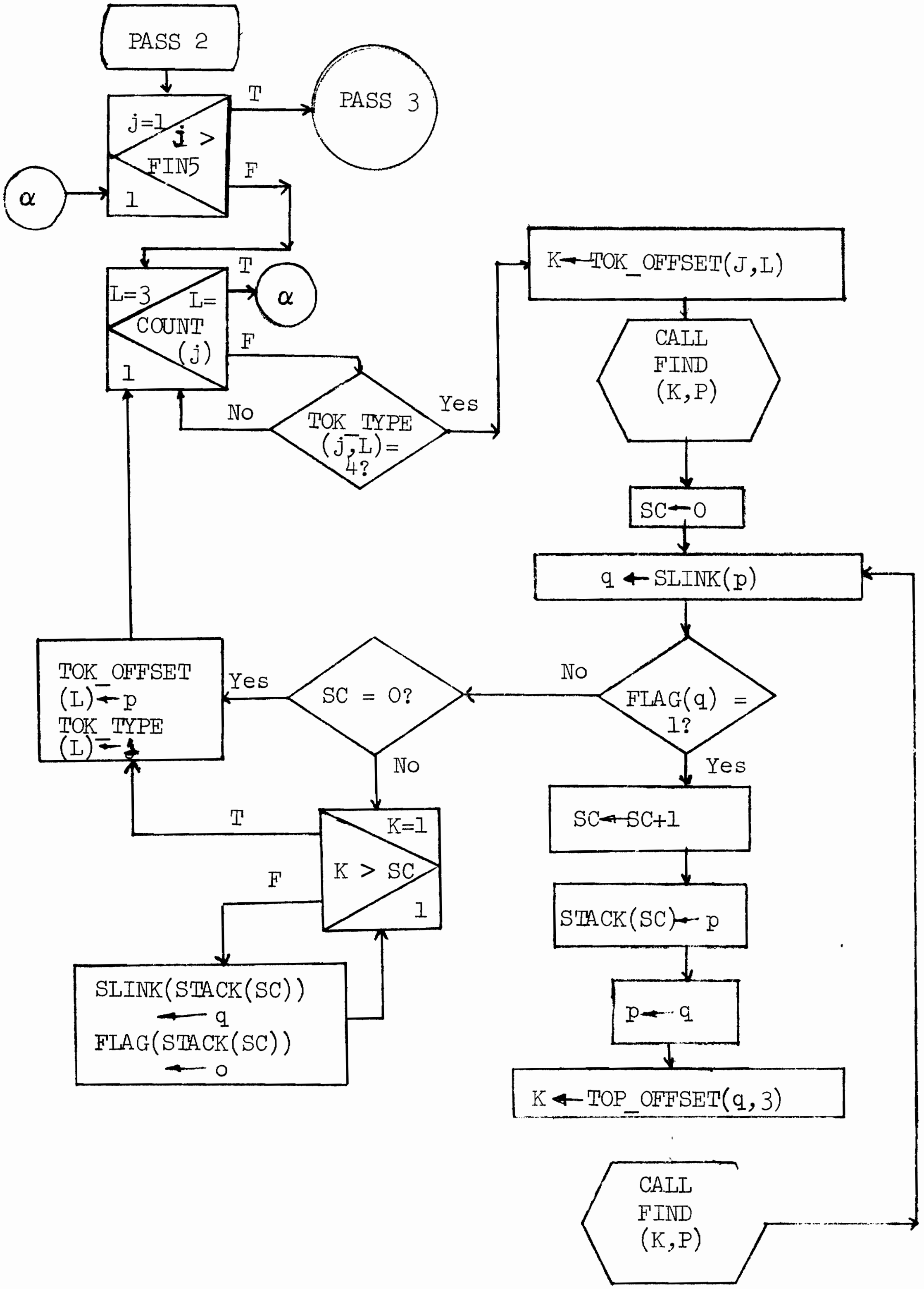
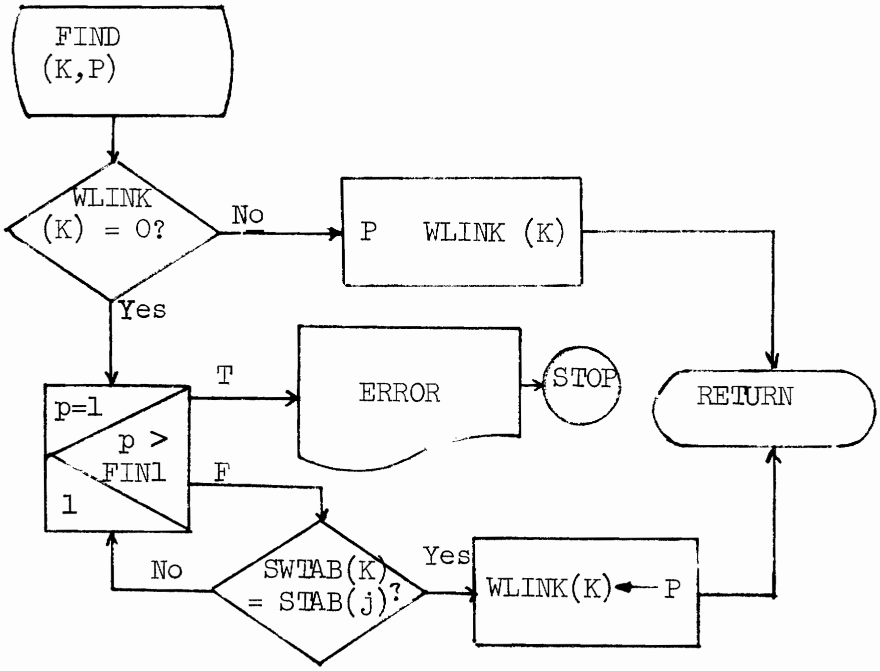
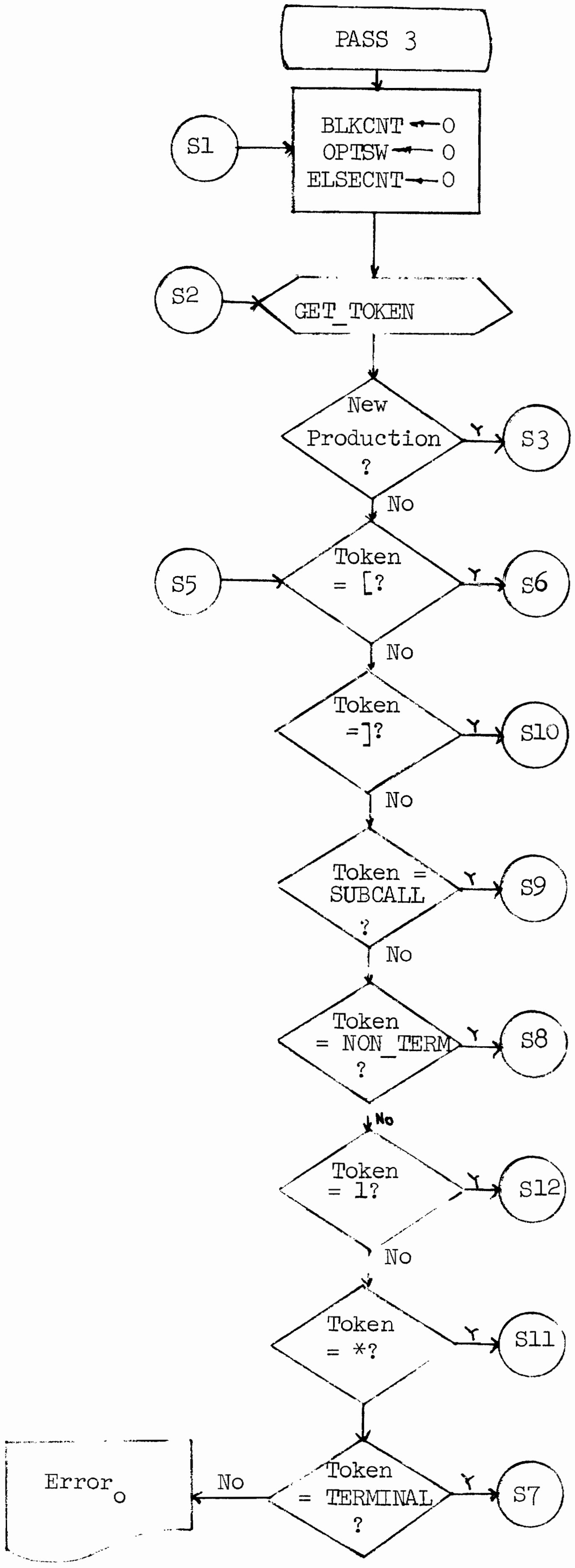

LEXEBNF
SW1 ← 0
LENGTH ← 0
S ← ''
INDC ← 0
I=0?
Yes
No
Read
C_i,i = 1 to 80
DOS
ISAVE = I
I = 1
UNO
I ≤ 79?
Yes
No
Left ← C(I)
Right ← C(I+1)
No
DOS
I = 80?
Yes
NL ← MAP(LEFT)
NR ← MAP(RIGHT)
DOS1
LEFT ← C(I)
J ← A(NL, NR)
READ
C_i, i=1 to 80
J
RIGHT ← C(1)
I ← 0
DOS1
Functions
(1 → 8)
RETURN

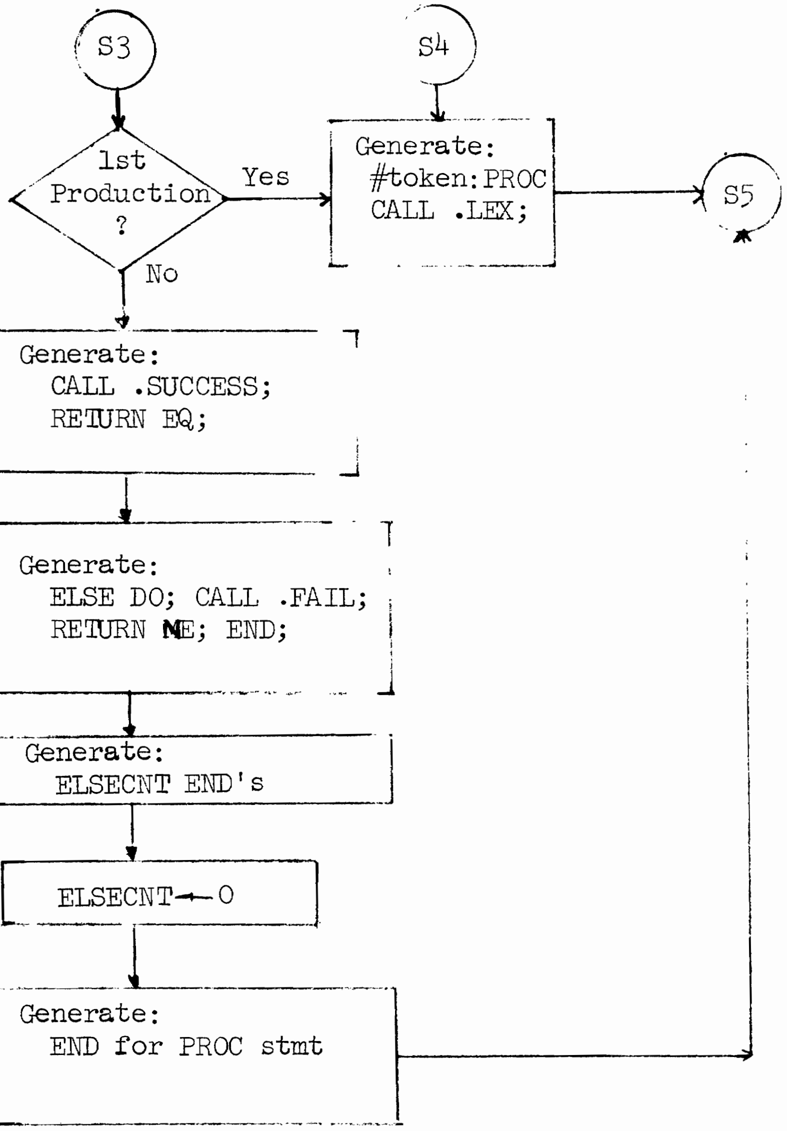
Figure E-1

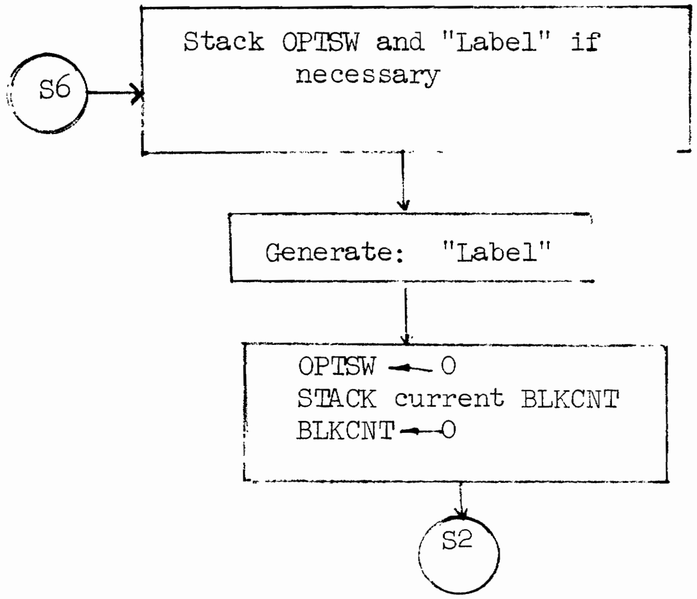
Part II - PASS 2

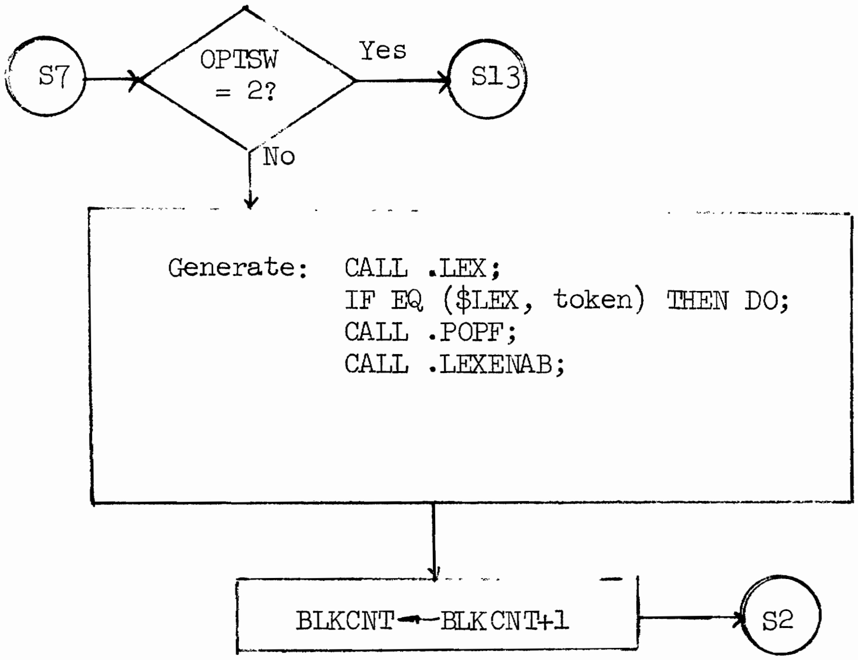


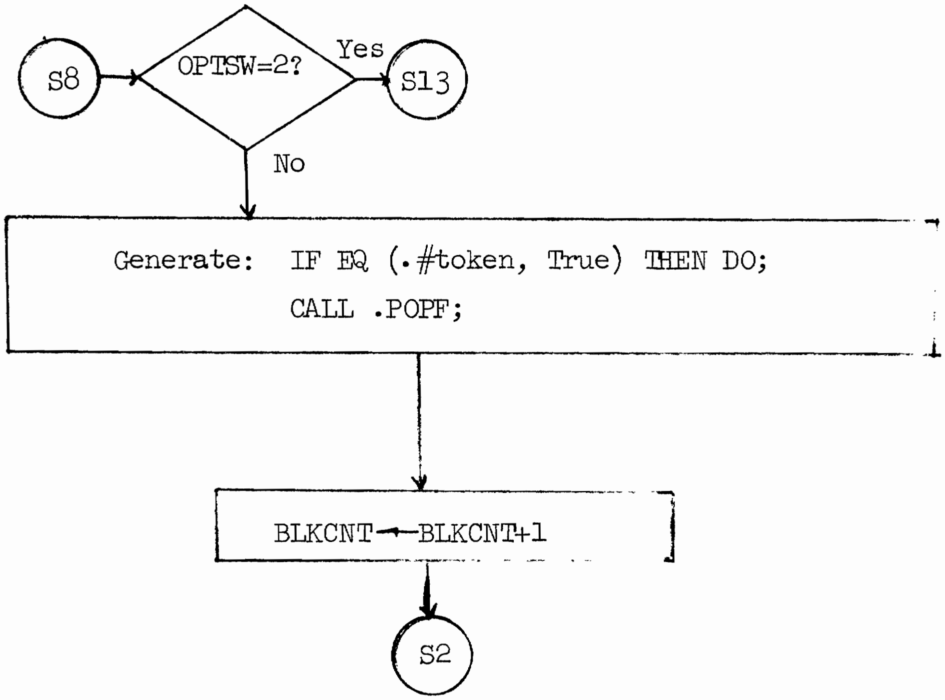

FIND
(K,P)
WLINK
(K) = 0?
No
P WLINK (K)
Yes
T
ERROR
STOP
RETURN
p=1
p >
FIN1
1
F
No
SWTAB(K)
= STAB(j)?
Yes
WLINK(K) ← P

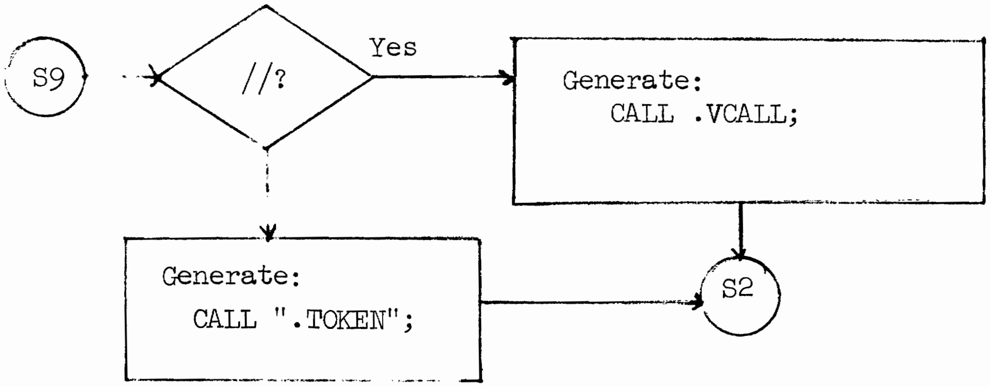

PASS 3
S1
BLKCNT ← 0
OPTSW ← 0
ELSECNT ← 0
S2
GET_TOKEN
New Production ?
Y
S3
No
S5
Token = [?
Y
S6
No
Token =]?
Y
S10
No
Token = SUBCALL ?
Y
S9
No
Token = NON_TERM ?
Y
S8
No
Token = 1?
Y
S12
No
Token = *?
Y
S11
Error₀
No
Token = TERMINAL ?
Y
S7

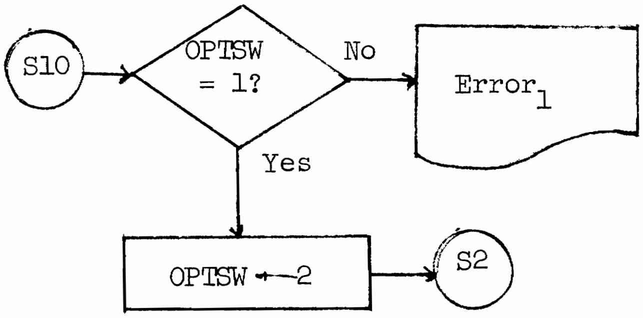

S3
1st Production ?
Yes
No
S4
Generate:
#token:PROC
CALL .LEX;
S5
Generate:
CALL .SUCCESS;
RETURN EQ;
Generate:
ELSE DO; CALL .FAIL;
RETURN NE; END;
Generate:
ELSECNT END's
ELSECNT←0
Generate:
END for PROC stmt

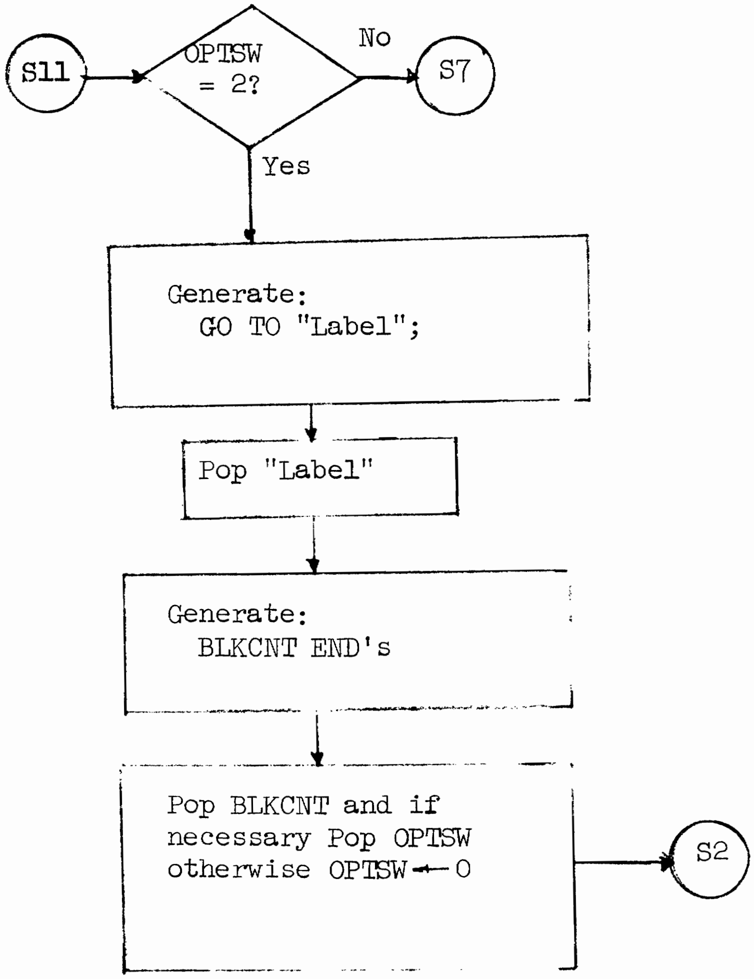

S6
Stack OPTSW and "Label" if necessary
Generate: "Label"
OPTSW ←0
STACK current BLKCNT
BLKCNT←0
S2

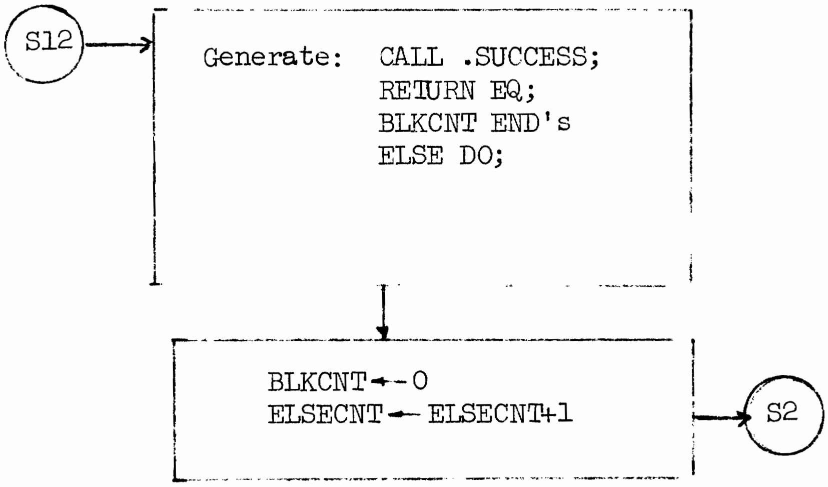

S7
OPTSW = 2?
Yes
S13
No
Generate: CALL .LEX;
IF EQ ($LEX, token) THEN DO;
CALL .POPF;
CALL .LEXENAB;
BLKCNT←BLKCNT+1
S2

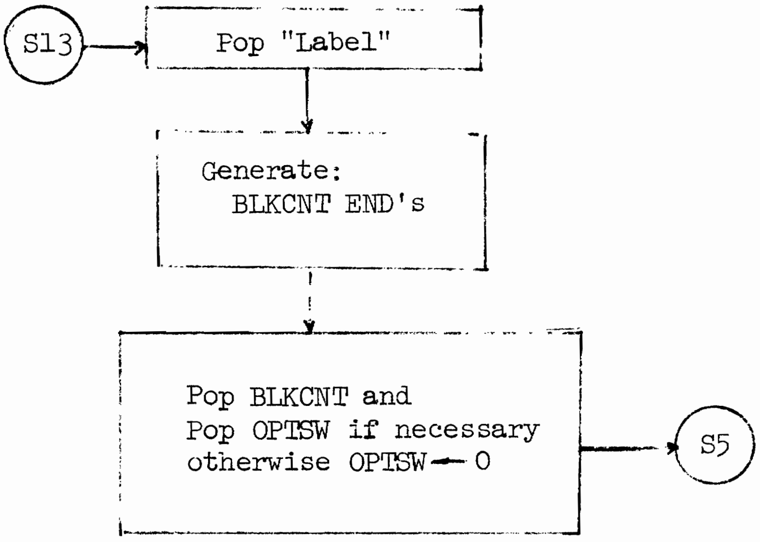

S8
OPTSW=2?
Yes
S13
No
Generate: IF EQ (.#token, True) THEN DO;
CALL .POPF;
BLKCNT←BLKCNT+1
S2


S9
//?
Yes
Generate:
CALL .VCALL;
Generate:
CALL ".TOKEN";
S2


S10
OPTSW
= 1?
No
Error$_1$
Yes
OPTSW ← 2
S2


S11
OPTSW
= 2?
No
S7
Yes
Generate:
GO TO "Label";
Pop "Label"
Generate:
BLKCNT END's
Pop BLKCNT and if
necessary Pop OPTSW
otherwise OPTSW ← 0
S2

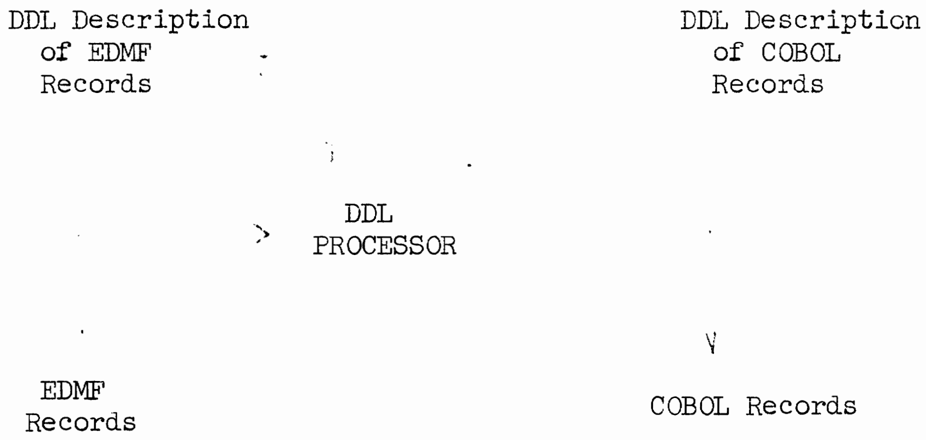

S12
Generate: CALL .SUCCESS;
RETURN EQ;
BLKCNT END's
ELSE DO;
BLKCNT ← 0
ELSECNT ← ELSECNT+1
S2


S13
Pop "Label"
Generate:
BLKCNT END's
Pop BLKCNT and
Pop OPTSW if necessary
otherwise OPTSW ← 0
S5

APPENDIX F

IMPLEMENTATION EXAMPLE

EDMF to COBOL Record Conversion


DDL Description of EDMF Records
DDL Description of COBOL Records
DDL PROCESSOR
EDMF Records
COBOL Records

This example demonstrates how the DDL can be used to describe the conversion of a file created by a program written in an Assembly Language to a file in the form which can be used by a COBOL program. The records to be converted are records produced by the Extended Data Management Facility (EDMF) of the University of Pennsylvania.

DDL PROGRAM

```
DESCRIPTION (EDMF_TO_COBOL)#

DESCRIBE (COBOL_FILE)#

'COBOL_FILE': STORFILE ('CF_STRUC')#

'CF_STRUC'" SEQUEN ('LINK_CRCD')#

'LINK_CRCD': LINK ('COBOL_RCD', 'COBOL_RCD', NOORD, F, 1)#

'COBOL_RCD': STORRC ('B_COBOL', 'OUT_CARDS')#

'B_COBOL': RECORD ('BOOK')#

'BOOK': GROUP (SPEC, ('X',M,F,1;'COUNT'),

                     ('Y',M,F,1),

                     ('Z',M,V,'X'))#

'X': FIELD (F,'E',Z,B)#

'Y': FIELD (F,'E',8,'E')#

'Z': GROUP (SPEC,('W',M,F,1))#

'W': FIELD (F,'E',W,'E')#

'COUNT': CRITERION ('COUNTEXP', 'X': RPLVAL (3: CONSTANT ('E')))#

'COUNTEXP': CRITEX ('X' LE (3" CONSTANT ('E')))#

'E': CHAR (8,EBCDIC)#

'OUT_CARDS': CARDOUT ('C_DECK_2')#

'C_DECK_2': BBLOCK (V,NOLIM,WHOLE,V,NOLIM,1;

              SPEC, ('B_COBOL',M,NOLIM)#

END#

DESCRIBE (EDMF_FILE)#

'EDMF_FILE': STORFILE ('EF_STRUC')#

'EF_STRUC': SEQUEN ('LINK_ERCD')#

'LINK_ERCD': LINK ('EDMF_RCD', 'EDMF_RCD', NOORD,F,1)#
```

```
'EDMF_RCD': STORRCD ('B_EDMF', IN_CARDS')#
'B_EDMF': RECORD ('G_EDMF')#
'G_EDMF': GROUP (SPEC, ('HDR',M,F,1),
                       ('D1',M,F,1),
                       ('D2',M,V,NOLIM))#
'HDR': GROUP (SPEC, ('SIZE',M,F,1),
                    ('REF',M,F,1),
                    ('INFO',M,F,1))#
'D1': GROUP (SPEC, ('A_V_HDR',M,F,1),
                   ('CENTRY',M,F,1))#
'D2': GROUP (SPEC, ('A_V_HDR',M,F,1),
                   ('AENTRY',M,F,1))#
'A_V_HDR': GROUP (SPEC,('A_V_LEN',M,F,1),
                 ('INFO',M,F,1),
                 ('D_NO',M,F,1))#
'CENTRY': GROUP (SPEC,('A_LEN',M,F,1),
                      ('C-NO_ATT',M,F,1),
                      ('V-LEN',M,F,1),
                      ('NUM',M,F,1))#
'SIZE': FIELD (F,'E1',3,B)#
'REF': FIELD (F,'E1',5,'E1')#
'INFO': FIELD (F,'E1',1,'B')#
'A_V_LEN': FIELD (F,'E1',3,'B')#
'D_NO': FIELD (F,'E1',1,'E1')#
'A_LEN': FIELD (F,'E1',2,'B')#
'C_NO_ATT': FIELD (F,'E1',11,'E1')#
```

```
'V_LEN': FIELD (F,'E1',3,'E1')#
'NUM': FIELD (F,'E1',8,'E1')#
'AENTRY': GROUP (SPEC,('A_LEN',M,F,1),
                      ('AUTH_ATT',M,F,1),
                      ('V_LEN',M,F,1),
                      ('AUTHOR',M,F,1))#
'AUTH_ATT': FIELD (F,'E1',6,'E1')#
'AUTHOR': FIELD (V,'E1','N_LEN' OF 'AENTRY','E1')#
'E1': CHAR (8,EBCDIC)#
'IN_CARDS': CARDIN ('C_DECK')#
'C_DECK': BBLOCK (V,NOLIM,WHOLE,V,NOLIM,I,SPEC;
                 ('B_EDMF',M,NOLIM))#
END#
EXECUTE (EDMF_FILE TO COBOL_FILE)#
CREATE ('COBOL_FILE','EDMF_FILE','ASSO_L')#
'ASSO_L': ASSOCIATION (('X', CNT ('D2')),
                       ('Y', 'NUM'),
                       ('Z', 'AUTHOR'))#
END#
DESCRIPTION END#
```

LOC	NAME*	Pointer to DATA TABLE	Pointer to FATHER
ST25	D2	D24	$D21_3$
ST26	SIZE	D27	$D22_1$
ST27	REF	D28	$D22_2$
ST28	INFO	D29	$D25_2$
ST29	A_V_HDR	D25	$D24_1$
ST30	CENTRY	D26	$D23_2$
ST31	AENTRY	D36	$D24_2$
ST32	A_V_LEN	D30	$D25_1$
ST33	D_NO	D31	$D25_3$
ST34	A_LEN	D32	$D36_1$
ST35	C_NO_ATT	D33	$D36_2$
ST36	V_LEN	D34	$D36_3$
ST37	NUM	D35	
ST38	E1	D39	
ST39	AUTH_ATT	D37	$D36_2$
ST40	AUTHOR	D38	$D36_4$
ST41	C-DECK	D41	
ST42	ASSO_L	D46	

SYMBOL TABLE FOR EDMF TO COBOL

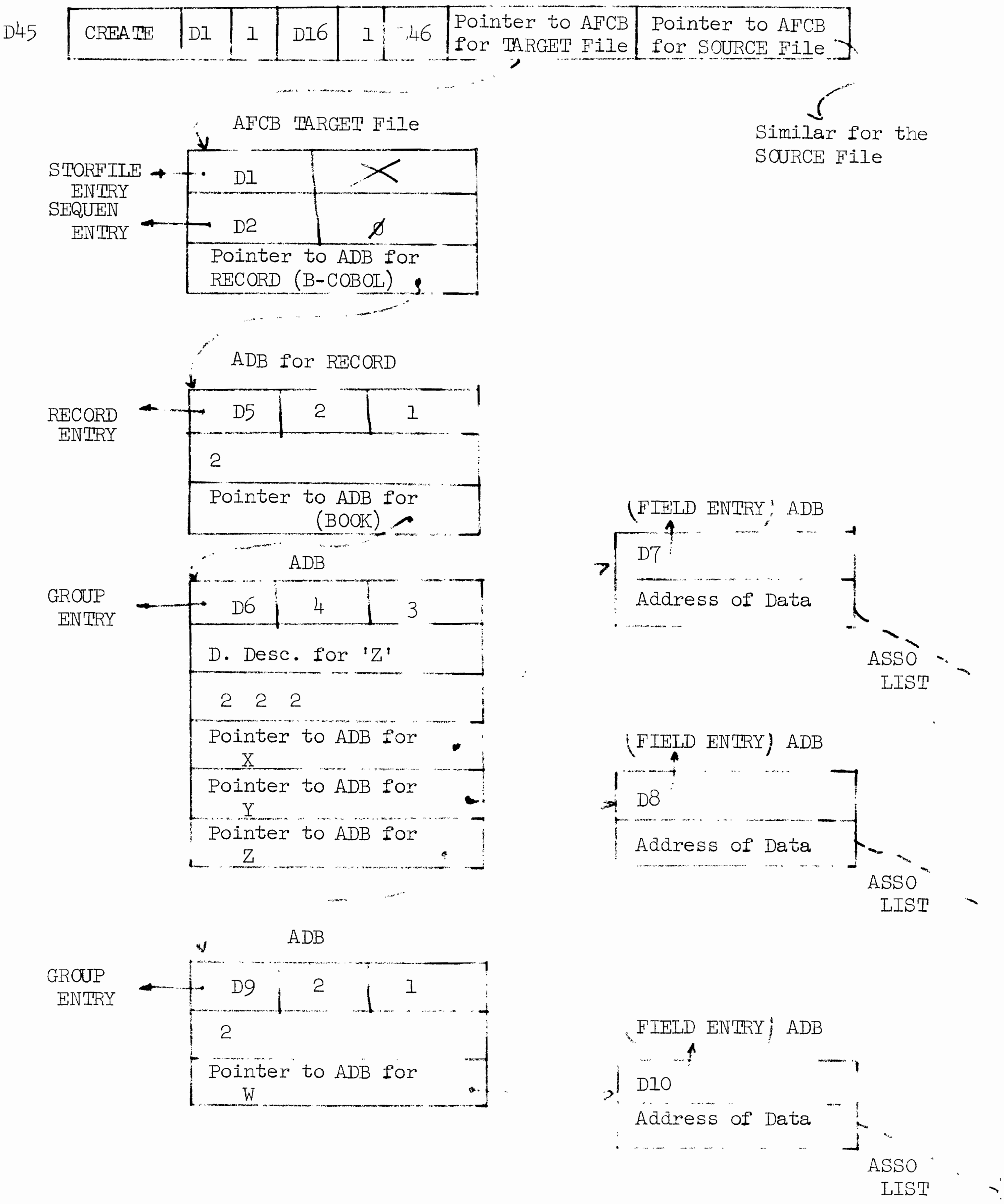
LOC	NAME*	Pointer to DATA TABLE	Pointer to FATHER
ST1	COBOL_FILE	D1	
ST2	CF_STRUC	D2	
ST3	LINK_CRCD	D3	
ST4	COBOL_RCD	D4	
ST5	B_COBOL	D5	
ST6	OUT_CARDS	D14	
ST7	BOOK	D6	$D5_0$
ST8	X	D7	$D6_1$
ST9	COUNT	D11	
ST10	Y	D8	$D6_2$
ST11	Z	D9	$D6_3$
ST12	E	D13	
ST13	W	D10	$D9_0$
ST14	COUNTEXP	D12	
ST15	C_DECK_2	D15	
ST16	EDMF_FILE	D16	
ST17	EF_STRUC	D17	
ST18	LINK_ERCD	D18	
ST19	EDMF_RCD	D19	
ST20	B_EDMF	D20	
ST21	IN_CARDS	D40	
ST22	G_EDMF	D21	$D20_0$
ST23	HDR	D22	$D21_1$
ST24	D1	D23	$D21_2$

DATA TABLE FOR THE EDMF TO COBOL EXAMPLE

	TYPE													
D1	STORFILE	1	ST2											
D2	SEQUEN	1	ST3											
D3	LINK	ST4	ST4	1	0	1								
D4	STORRCD	ST5	1	ST6										
D5	RECORD	ST6	$D5_0$	Φ										
D6	GROUP	$D5_0$	1	3	ST8	$D6_0$	Φ	1	1	0	1	Φ	1	ST9
					ST10	$D6_0$	Φ	1	1	0	1	Φ	0	Φ
					ST11	$D6_0$	Φ	1	0	0	ST8	Φ	0	Φ
D7	FIELD	$D6_1$	1	Φ	0	ST12	2	1						
D8	FIELD	$D6_2$	1	Φ	0	ST12	8	ST20						
D9	GROUP	$D6_3$	1	1	ST13	$D9_0$	Φ	1	1	0	1	Φ	0	Φ
D10	FIELD	$D9_0$	1	Φ	0	ST12	20	ST12						
D11	CRITERION	ST14	D42											
D12	CRITEX	ST8	LE	D44										
D13	CHAR	8	1											
D14	CARDOUT	ST5												
D15	BBLOCK	0	32K	1	0	32K	1	1	ST5	1	32K			
D16	STORFILE	1	ST17											
D17	SEQUEN	1	ST18											
D18	LINK	ST19	ST19	1	0	1								
D19	STORRCD	ST20	1	ST21										
D20	RECORD	ST22	$D20_0$	Φ										

D21	GROUP	$D20_0$	1	3	ST23	$D21_0$	Φ	1	1	0	1	Φ	0	Φ
					ST24	$D21_0$	Φ	1	1	0	1	Φ	0	Φ
					ST25	$D21_0$	Φ	1	0	0	32K	Φ	0	Φ
D22	GROUP	$D21_1$	1	3	ST26	$D22_0$	Φ	1	1	0	1	Φ	0	Φ
					ST27	$D22_0$	Φ	1	1	0	1	Φ	0	Φ
					ST28	$D22_0$	Φ	1	0	0	1	Φ	0	Φ
D23	GROUP	$D21_2$	1	2	ST29	$D23_0$	Φ	1	1	0	1	Φ	0	Φ
					ST30	$D23_0$	Φ	1	1	0	1	Φ	0	Φ
D24	GROUP	$D21_3$	1	2	ST29	$D24_0$	$D23_1$	1	1	0	1	Φ	0	Φ
					ST31	$D14_0$	Φ	1	1	0	1	Φ	0	Φ
D25	GROUP	$D24_1$	1	3	ST32	$D25_0$	Φ	1	1	0	1	Φ	0	Φ
					ST28	$D25_0$	$D22_3$	1	1	0	1	Φ	0	Φ
					ST33	$D25_0$	Φ	1	1	0	1	Φ	0	Φ
D26	GROUP	$D23_2$	1	4	ST34	$D26_0$	Φ	1	1	0	1	Φ	0	Φ
					ST35	$D26_0$	Φ	1	1	0	1	Φ	0	Φ
					ST36	$D26_0$	Φ	1	1	0	1	Φ	0	Φ
					ST37	$D26_0$	Φ	1	1	0	1	Φ	0	Φ
D27	FIELD	$D22_1$	1	Φ	0	ST38	3	1						
D28	FIELD	$D22_2$	1	Φ	0	ST38	5	ST38						
D29	FIELD	$D25_2$	1	Φ	0	ST38	1	1						
D30	FIELD	$D25_1$	1	Φ	0	ST38	3	1						
D31	FIELD	$D25_3$	1	Φ	0	ST38	1	ST38						
D32	FIELD	$D26_1$	1	Φ	0	ST38	2	1						
D33	FIELD	$D26_2$	1	Φ	0	ST38	11	ST38						
D34	FIELD	$D26_3$	1	Φ	0	ST38	3	ST38						
D35	FIELD	$D26_4$	1	Φ	0	ST38	8	ST38						

D36	GROUP	$D24_2$	1	4	ST34	$D36_0$	$D26_1$	1	1	0	1	Φ	0	Φ
					ST39	$D36_0$	Φ	1	1	0	1	Φ	0	Φ
					ST36	$D36_0$	$D26_3$	1	1	0	1	Φ	0	Φ
					ST40	$D36_0$	Φ	1	1	0	1	Φ	0	Φ
D37	FIELD	$D36_2$	1	Φ	0	ST38	6	ST38						
D38	FIELD	$D36_4$	0	Φ	0	ST38	'ST36 OF ST31'	ST38						
D39	CHAR	8	1											
D40	CARDIN	ST41												
D41	BBLOCK	0	32K	1	0	32K	1	1	ST20	1	32K			
D42	RPLVAL	ST8	D43											
D43	CONSTANT	0	3	ST38										
D44	CONSTANT	0	3	ST38										
D45	CREATE	ST1	1	ST16	1	ST42								
D46	ASSOCIATE	3	ST8	D47	ST10	ST37	ST11	ST40						
D47	CNT	ST8												


D45
CREATE
D1
1
D16
1
46
Pointer to AFCB for TARGET File
Pointer to AFCB for SOURCE File
Similar for the SOURCE File
AFCB TARGET File
STORFILE ENTRY
D1
SEQUEN ENTRY
D2
Ø
Pointer to ADB for RECORD (B-COBOL)
ADB for RECORD
RECORD ENTRY
D5
2
1
2
Pointer to ADB for (BOOK)
ADB
GROUP ENTRY
D6
4
3
D. Desc. for 'Z'
2 2 2
Pointer to ADB for X
Pointer to ADB for Y
Pointer to ADB for Z
(FIELD ENTRY) ADB
D7
Address of Data
ASSO LIST
(FIELD ENTRY) ADB
D8
Address of Data
ASSO LIST
ADB
GROUP ENTRY
D9
2
1
2
Pointer to ADB for W
FIELD ENTRY) ADB
D10
Address of Data
ASSO LIST