A BASIC LIST-ORIENTED INFORMATION STRUCTURES SYSTEM (BLISS)

by

Charles Scott Thorell

United States Naval Postgraduate School

THESIS

A BASIC LIST-ORIENTED INFORMATION STRUCTURES SYSTEM (BLISS)

by

Charles Scott Thorell

and

William Otto Poteat, Jr.

June 1970

A Basic List-Oriented Information Structures System (BLISS)

by

Charles Scott Thorell
Lieutenant Commander, United States Navy
B. S., United States Naval Academy, 1961

and

William Otto Poteat, Jr.
Lieutenant, United States Navy
B. S., University of North Carolina, 1964

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
June 1970

ABSTRACT

The design and implementation of the Basic List-Oriented Information Structures System is described. Manipulation of list structures in an efficient and cogent manner is the system function. The language, which is patterned after Bell Telephone Laboratories' L^6, is generated from a precedence grammar for rapid syntax analysis. A compiler produces code for a pseudo-machine that is designed to effectively carry out list-oriented functions. Dynamic storage allocation and structure definition are significant execution-time features. The implementation, written in PL/I, is for operation under the CP/CMS time-sharing system on the IBM 360/67 computer.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

Rio

Paris

I. THE PROBLEM AND DEFINITION OF TERMS USED

There is an important subset of computer programming problems in which the conventional general-purpose languages such as FORTRAN, ALGOL and PL/I are of limited value. These problems are characterized by complex data structures in which both the information content and the interrelationships within the structures are of importance. Such situations arise in the fields of artifical intelligence, information retrieval, network simulation, graph manipulation, and many other similar areas. The family of programming languages referred to as list processing languages has been developed to simplify the solution of these problems.

A. STATEMENT OF THE PROBLEM

The primary objective in the definition and implementation of the Basic List-Oriented Information Structures System (BLISS) was to provide an on-line list processing language which would stress conceptual learning.

BLISS is based upon Bell Telephone Laboratories' Low-Level Linked List Language, L^6 [Ref. 1]. The language has been modified to parallel the vocabulary of the list processing field. The additional features of on-line syntax analysis and subsequent program correction have been added.

The basic goals of BLISS are essentially the same as those of L^6. BLISS allows a flexible means of defining complex data structures in a manner which stresses easy visualization of structural interrelationships. In addition, it allows efficient use of available storage. Since BLISS operations are primitive

list processing functions, the programmer is forced to work at a level where a complete understanding of the underlying concepts is a prerequisite.

B. DEFINITIONS OF TERMS USED

Throughout the discussion which follows, a general familiarity with basic terms in Computer Science is assumed. The terms defined in this section are essential for comprehension of the remainder of the paper.

1. Compiler

In general terms, a compiler may be defined as a computer program which accepts as input a machine independent source language program and produces a translated version of it as machine dependent object code.

The BLISS compiler consists of three parts. There are two parts devoted to analysis of the BLISS source language statements and one part which generates object code.

The first part of the BLISS compiler is the lexical analyzer. In this phase of compilation, the entities in the source language are classified and recorded.

The syntax analyzer is the second part, and its function is to check the sequence of entities passed by the lexical analyzer to insure that they constitute an allowable statement in the language.

The last portion of the BLISS compiler is the code generator, which generates the object code for the statements passed by the analysis phases.

2. Hash Coding

Hash coding, also referred to as scatter storage, scramble coding and address calculation, is a term used to describe the class of methods used to enter data into a table by using some feature of the data to construct the table address. The reason for entering data in this fashion is to simplify the job of locating the data when it must be retrieved.

When hash coding produces the same address for two (or more) different data items, a collision is said to exist, and some secondary procedure must be devised for entering the additional data into the table.

Of the numerous methods for hash coding and resolving collisions, many result in an even distribution of data in the table and allow efficient retrieval. An excellent review of hash coding techniques is presented by Morriss in Ref. 2.

3. Stack

A stack, or pushdown store, is one of the fundamental concepts used in list processing. It may be thought of as an array or string of elements in which only the most recently added element is accessible. An element is added to the stack by "pushing down" the stack, and elements are removed by "popping up" the stack.

A stack is sometimes referred to as a LIFO (last-in, first-out) queue.

4. Backus-Naur Form

Backus-Naur or Backus-Normal Form (BNF) is one of the principle notational methods used in the formal definition of programming languages. It is presented in the "Revised Report on the Algorithmic Language ALGOL 60," Ref. 3.

BNF is designed to reduce the confusion between the metalanguage (the language used to describe the programming language, e.g. the English language) and the language being described. In the BNF metalanguage, the brackets < > enclose metalinguistic variables, but are omitted for basic elements of the language being described. The marks := and |, which mean "is defined as" and "or," are metalinguistic connectives used in the formulae.

As an example,

<Conditional> := IFANY|IFALL|IFNONE|IFNALL

is the notation for: "a conditional is defined as IFANY or IFALL or IFNONE or IFNALL."

5. Interpreter

There are many and varied definitions of what constitutes an interpreter. For the purposes of this discussion it is considered to be a computer program which accepts as input some form of source language (e.g. assembly code) and causes the appropriate commands to be carried out through the used of pre-compiled procedures.

In the BLISS system, the interpreter may be thought of as a software "machine" which causes the output of the

BLISS compiler to be executed on the actual machine which is executing the interpreter.

C. ORGANIZATION OF THE REMAINDER OF THE THESIS

Background information is presented in Chapter II, which contains a brief history of several list processing languages, including L^6 and related systems. A discussion of the basic features and proposed uses of BLISS is contained in Chapter III. The implementation is described in the next two chapters, IV and V, which explain the compiler and interpreter, respectively. Conclusions, recommendations, and possible extensions are presented in Chapter VI. Appendix A, the BLISS Programmer's Manual, contains a complete language description and programming examples. Next, the BLISS interpreter instructions are included as Appendix B. Program listings of the BLISS compiler and interpreter comprise Appendices C and D.

II. PREVIOUS WORK OF A SIMILAR NATURE

The first portion of this section is devoted to a brief description of the evolution of some major list processing languages. This is followed by a description of L^6 and several other programming systems that are derived from it.

A. IPL-V, LISP AND SNOBOL

Three well-known list processing languages have been chosen for a brief description of their features. Among them, they exhibit many of the principle functions that are valuable in list processing applications.

1. IPL-V

The first major work on list processing languages began with Newell, Shaw and Simon in 1954, and resulted in "Information Processing Language 5" (IPL-V). The description of IPL-V appeared in 1960 [Ref. 4].

The IPL series of programming languages is oriented toward work in heuristic problem-solving. The last of the series, IPL-V, is a procedure language essentially at the assembly language level. The instructions are the assembly code of a hypothetical machine which is oriented toward list processing applications.

The significant contributions of IPL include: (1) the introduction of the concept of a list and the demonstration of its practical applications in solving a variety of problems, (2) the development of pushdown and popup operations, and (3) the concept of redefinition of data structures during program execution.

2. LISP

Work on the LISP programming system was originated at M.I.T. in 1959 in conjunction with the Advice Taker Project, under the direction of Professor John McCarthy. The goal of the programming system was to allow manipulation of expressions representing formalized declarative and imperative sentences so that the Advice Taker system could make deductions [Ref. 5]. The LISP 1.5 manual was published in 1960 [Ref. 6].

The concepts and notation used in LISP are unique among programming languages. The basic data element is the atom, which is either an identifier or a number. The data and programs in LISP are described in parenthesized expressions (S-expressions) which define the interrelationships between the atoms. The list notation results in a binary tree structure built from atoms.

The language for defining operations on S-expressions is the metalanguage, consisting of M-expressions. The M-expressions are parenthesized structures of elementary functions, with arguments consisting of S-expressions. Some of the functions which are available allow defining structures, logical tests on atomic symbols and on structures, conditional expressions, and arithmetic operations.

The complex notation used LISP makes it quite difficult for the average programmer to read and comprehend. Recursion and primitive list processing functions make LISP a powerful language, but unfortunately it is understood and used

properly by a small number of programmers. In some circles, LISP is considered to be more an "art form" than a programming language.

Some of the significant contributions of the LISP system are: (1) formalism of the concept of the conditional expression, (2) demonstration of the value of recursive function definition, and (3) the technique of "garbage collection," which consists of automatically returning unused storage to available space.

3. SNOBOL

In 1962 at Bell Telephone Laboratories, Farber, Griswold and Polonsky developed the SNOBOL system [Ref. 7]. While SNOBOL is defined as a string manipulation language, it is included in this discussion because it can be readily used to define list processing functions. It contains many features of an earlier string manipulation language, COMIT, which is described in Ref. 8. One of the principle applications of both SNOBOL and COMIT is in natural language translation research.

The basic data element in SNOBOL is the string, which consists of a finite sequence of symbols from the character set. Strings and substrings may be assigned names.

A SNOBOL program is a set of statements, each of which involves a rule. The set of rules allows manipulation of strings in a variety of ways.

The facility for the user to define functions of a general nature and the extensive pattern-matching capabilities are important features of SNOBOL.

B. L^6 AND RELATED LANGUAGES

L^6 was designed by Kenneth C. Knowlton at the Bell Telephone Laboratories and presented in 1966 [Ref. 1]. In 1967, Newell, Haney and Earley implemented a language based on L^6 called *1 (pronounced star-one) at Carnegie-Mellon University [Ref. 9]. An adaption of *1, the Data Structures Programming System (DSPS), was developed by Evans and Van Dam in 1969 [Ref. 10].

1. L^6

As Knowlton states in his description of the language, L^6

> ...permits the user to get much closer to machine code in order to write faster-running programs, to use storage more efficiently and to build a wider variety of linked data structures.

As this description implies, the format of L^6 is similar to that of an assembly language program. It provides primitive facilities for defining and operating upon data structures. The L^6 program structure consists of a sequence of individual statements, each of which appears on a separate line.

The basic data structure element in L^6 is the block, which is a contiguous set of 2^n words in memory, where n can take on the values zero through seven. Within the block the programmer can define up to 36 fields, named with a single

character or digit. Fields may be defined as small as one bit and as large as 36 bits (in the IBM 7094 L^6 versions), and overlapped fields are permitted.

The L^6 system also contains 26 base fields or registers, called BUGS. These consist of permanently assigned fullword fields used to hold pointers or other data. They are primarily used to point to blocks in storage, and are distinguished from defined fields by the context in which they are used.

Data is accessed by concatenation of a bug with defined fields containing pointers, with the field containing the desired data as the last field in the sequence. For example, WAB represents the sequence: bug W points to a block with an A field which points to a block with a B field which contains the desired data.

The instructions in L^6 are specified using somewhat cryptic abbreviations. For example,

```
IFALL (XA,E,0)(YA,G,4) THEN (YA,S,1)
```

results in subtracting one from field YA if field XA equals zero and field YA is greater than four. Operations and tests are performed from left to right in each statement.

Instructions in L^6 fall into one of two categories: tests or operations. Tests allow comparison of a bug or field with another bug, field or literal. Tests are available to determine if the first argument is equal to, not equal to, greater than, or less than the second argument. L^6 operations

provide for data structure manipulation, arithmetic, logical, shifting, input/output, conversion and pushdown/popup operations. Tests and operations in BLISS are basically the same as in L^6, and will be explained in detail in the description of BLISS.

The general format for an L^6 statement is

[label] conditional test(s) THEN operation(s) [label]

The only restriction on the number of tests and operations in a statement is that they must all fit on one card image. The first label in the statement is optional, and allows branching to the statement, either from another statement or recursively. The conditional instructions allowed are "IFONE," "IFANY," "IFALL" and "IFNALL." They are satisfied by the truth of none, any, all, or not all of the tests which follow. If the conditional is satisfied, then the operations are performed. If a branch label follows, control is transferred to the specified labelled statement, otherwise to the next line. If the operations are missing the delimiter "THEN" may be omitted, but in that case the branch label is obligatory. If the reserved word "THEN" appears first in a statement, the operations or branch (or both) which follow are unconditionally executed.

Subroutines consist of a sequence of statements which have the subroutine name as the label of the first line. Subroutines are called by using the elementary operation "(DO, label)." Exit from the subroutine is usually via the special

branch "DONE," which returns control to the point in the program just following the "DO" operation which invoked the subroutine. Care must be taken in placing the subroutine in the program so it is not inadvertently executed.

L^6 was originally implemented on the IBM 7094. Since that time, implementations have been adapted to the IBM 7040, IBM 360, MOBIDIC B, PDP 6, and SDS 940 [Ref. 11]. A version has also been developed on the London University ATLAS Computer, with the syntactical definition and implementation written in BCL, a general purpose programming language with special emphasis on data structures [Ref. 12].

As can be readily seen, L^6 is a primitive list processing system. The facilities available in L^6, however, allow the programmer to perform virtually any operation imbedded in higher level languages at the cost of some detailed programming.

2. *1

The *1 system was implemented on Carnegie-Mellon's IBM 360 in 1967 by Professors Newell, Earl and Haney. It is based on L^6, and consists of a set of system/360 assembly language macros designed to perform the list processing functions defined in L^6. *1 has added features which include operations on blocks, dynamic bounds on fields and blocks, and a meta-language.

Block operations are accomplished by allowing the programmer to name blocks as well as fields. Accessing of

data is done as in L[6]. Single letter names in the sequence are concatenated with no punctuation, but other symbolic names may be assigned to fields or blocks, in which case they require brackets around them in the sequence.

The function of the meta-language is to allow the programmer to have some control over the code that is produced. Before using this feature, the programmer must be familiar with the *1 macros used in the implementation.

3. DSPS

DSPS, which is implemented on Brown University's IBM 360/50, is an adaptation of *1, with added features making it particularly useful for graphical and file mangagement applications.

A facility for retrieving items from a 2314 disk file using Boolean and relational functions of keywords has been added. A method of "paging" items onto the disk file (for either temporary or permanent use) gives the programmer a great deal of flexibility in creating and manipulating data structures.

III. A DESCRIPTION OF BLISS

BLISS is not intended to solve all the problems in the list processing field. Rather, it is designed to allow straightforward programming of all of the principle list processing concepts. Because of its basic nature, BLISS is well-suited for instructional use or experimentation, but its use is not limited to these areas. As in L^6, virtually any list processing operation can be performed at the expense of detailed programming.

This chapter includes a description of the key features of BLISS, along with a presentation of proposed system uses and a discussion of the BLISS implementation.

At this point in the discussion, it is suggested that the reader familiarize himself with the BLISS language by reading the BLISS Programmer's Manual [Appendix A].

A. KEY FEATURES OF BLISS

The basic instructions and concepts of BLISS are essentially the same as those of L^6, but the language and program structure have been redefined to make it appear as a higher-level language to the programmer.

In this section, the key features of BLISS are presented.

1. Data Structures

One of the most significant features of L^6 is the flexible method of defining data structures. The programmer is free to define his data structure in the manner which best suits his particular problem. BLISS allows virtually the same flexibility with several added features.

Blocks, defined fields, and bugs are used in BLISS in the same manner as in L^6. In BLISS, however, the names of defined fields and bugs are not restricted to a single character. Any non-reserved sequence of letters and digits (beginning with a letter) may be used. Data is accessed by concatenation of pointers to the desired field, using a period as a delimiter. In complex data structures, the ability to use meaningful names for fields and bugs makes visualization of the problem simpler, and also makes the entire program more readable.

BLISS also allows multiple-word character field definition, thus simplifying the job of storing and retrieving large blocks of character data. As an example, the five-word character field

T H I S
I S
A N
E X A M
P L E

can be stored or retrieved in one operation, whereas in L^6 it would have to be done one fullword at a time.

2. The BLISS Instruction Set

While the instruction set of BLISS is similar to that of L^6, the formats and operators have been changed to make statements more readable. Most BLISS operators parallel the terminology used in the list processing field.

As an example, the L^6 statement

```
BRANCH IF(X,L,10)THEN(X,A,1)(S,FC,Y)(Y,GT,4)BRANCH
```

is equivalent to the BLISS statement

```
BRANCH: IF X < 10, THEN X+1, STACK Y, GET Y 4, GOTO BRANCH;
```

Sophisticated programmers may get more satisfaction out of generating compact (albeit somewhat cryptic) strings of symbols, but the notation of BLISS should prove more satisfying to most users.

3. BLISS Program Structure

In the formal definition of BLISS, found in Addendum I to Appendix A, one of the primary goals was to achieve a logical and flexible program structure. A BLISS program starts with a BEGIN statement and ends with a STOP statement. BEGIN and STOP statements are paired within the program to delimit subroutines. For example,

```
BEGIN:
  ________________
  ________________
  BEGIN;
  SUB1:_____
  ___________
  DONE;
  STOP;
  ________________
  DO SUB1;
STOP;
```

is a sample program format that demonstrates the use of a subroutine, SUB1. Note that the subroutine name is defined as the label of the first line within the subroutine body.

The BLISS statement format has been modified from that used in L^6. BLISS statements are free-field and not

restricted to one line as in L^6. The basic elements of a BLISS statement are

<LABEL><CONDITIONAL> <TEST SET> THEN <OPERATION SET><BRANCH> ;

The basic statement elements and their order of appearance are the same as in L^6. However, a BLISS statement is more flexible and readable than an equivalent L^6 statement due to the modified instruction set and the free-field features of BLISS.

B. THE BLISS IMPLEMENTATION

BLISS is implemented on the IBM 360/67 Cambridge Monitor System at the Naval Postgraduate School Computer Center. The entire system is programmed in PL/I.

PL/I was selected as the programming language for the system for several reasons. First, it was recognized that programming in a general-purpose language would make subsequent modifications to the system much easier than would be the case if assembly language coding was used. Further, PL/I achieves a degree of machine independence not possible with assembly language. PL/I was chosen over other high-level languages due to its superior bit-string and character-string manipulation features.

While currently implemented for on-line use only, the system could be easily modified for batch processing. This would be particularly useful if long production runs were envisioned.

One of the particularly useful system features is the facility for on-line program correction during compilation. This is particularly useful for rapid debugging. If a syntax error is detected in a particular statement, the compilation process is interrupted and a diagnostic message informs the programmer that he has made an error in the statement. The programmer then has the option of correcting the statement or stopping the compilation process. Since most of the detectable errors either involve punctuation or instruction format, these should be readily detectable.

C. POSSIBLE SYSTEM USES

The inherent simplicity and flexibility of BLISS make it quite suitable for demonstrating the concepts associated with information structures and basic information retrieval. In this section, some of the possible applications of BLISS as an instructional system are presented.

1. Information Structures

The most logical application area for BLISS would be in an introductory course in Information Structures. The system lends itself nicely to the concepts presented in Volume I, Chapter 2 of Knuth [Ref. 13], which is one of the standard textbooks in the field.

Example A in the Programmer's Manual, the postorder traversal of a binary tree, is a BLISS program which executes Knuth's Algorithm T [page 317 of Ref. 13].

The following BLISS subroutine, presented as an L^6 example by Knowlton in Ref. 1, returns the binary tree in Figure 1 to free storage:

```
BEGIN:
  RETURNTREE: IF W=0, THEN DONE;
  STACK W.R, FREE W W.L,
  DO RETURNTREE;
  POP W, GOTO RETURNTREE;
STOP;
```

For comparison, the equivalent L^6 subroutine is:

```
RETURNTREE: IF (W,E,0)DONE
  THEN(S,FC,W.R)(W,FR,W.L)(DO,RETURNTREE).
  (R,FC,W)RETURNTREE
```

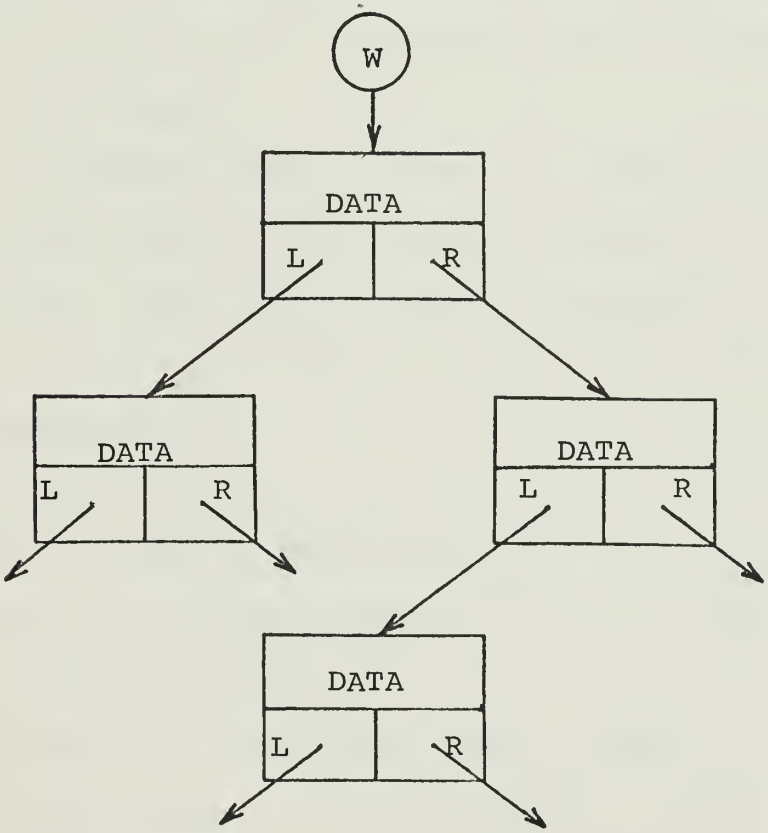


Figure 1. Binary Tree Data Structure.

2. Information Retrieval

Because of the flexibility for defining data structures and establishing interrelationships among various elements, BLISS should prove useful in basic information retrieval experiments.

One disadvantage which would preclude using BLISS for large information retrieval systems is the inability to save data structures for future runs. It is hoped that this feature will be added in a revision to the system.

3. Other Application Areas

There are several other application areas where the BLISS system can be used to advantage. Among these are automata theory, graphics data structures, and some areas of artificial intelligence.

In automata theory, BLISS can readily be used to program problems dealing with pushdown automata and basic Turing machines. Most of the concepts presented in Chapters 5 and 6 of Hopcroft and Ullman's textbook [Ref. 14] can be demonstrated with BLISS.

BLISS can also be used profitably to investigate manipulation of graphics data structures. For actual use of the structures in graphics applications, it is best to program the system using L^6 or DSPS because of timing considerations.

Certain problems which appear in the field of artificial intelligence can be programmed in BLISS. Game playing, pattern recognition and other areas where data

structures are complex and where operations such as pattern matching are important are cases in point.

IV. THE BLISS COMPILER

Compilation of a BLISS program is accomplished in three phases: (1) lexical analysis, (2) syntax analysis, and (3) code generation. Figure 2 shows the interrelationships among the phases.

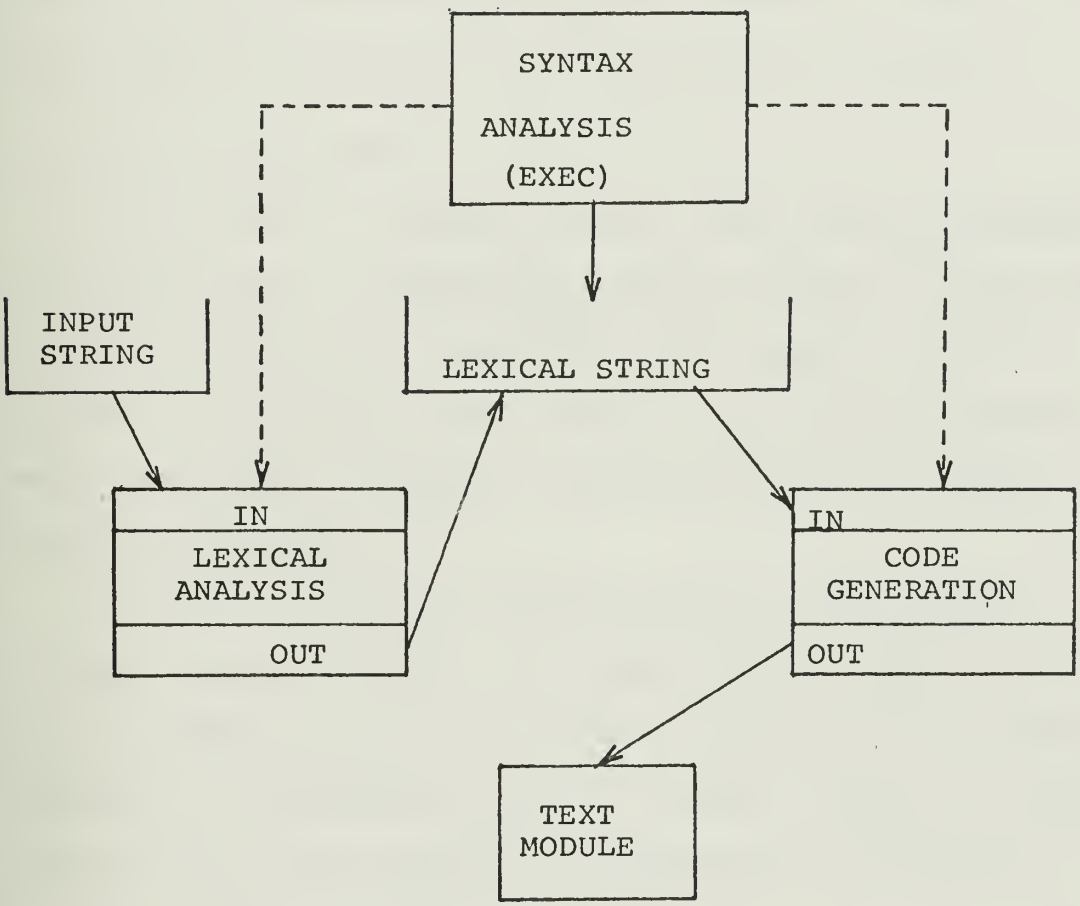


Figure 2. The BLISS Compiler.

The executive portion of the compiler is imbedded in the syntax analyzer. It calls the lexical analyzer, which returns a complete BLISS statement in the form of a lexical string. The syntax analyzer then checks the statement to insure that it is syntactically correct. If the statement is correct, the code generator is called to generate the code which will be executed by the BLISS "machine." Each of the three major sections will now be described in detail.

A. LEXICAL ANALYSIS

In the lexical analysis phase, BLISS source statements are scanned and the meaningful program elements in the statement are recorded and classified. The lexical analyzer performs its functions when called by the syntax analyzer, and passes a string of elements corresponding to the BLISS statement entities.

Lexical analysis is accomplished in three phases: (1) the scanner routine, (2) symbol table construction routines, and (3) construction of the lexical string for the syntax analyzer. The procedure used for lexical analysis (LEXPHAZ) can be found in the compiler program listing, Appendix C.

1. The Scanner Routine

The scanner routine (SCAN) of the lexical analyzer scans the BLISS input file from left to right, placing one program element at a time into a buffer (ACCUM). The variable RESULT is set to one, two or three, depending on whether the

item in the buffer is an identifier, integer or special character. The variable COUNT is set to the number of characters in the element. One program element is returned on each call on SCAN. When the end of a card image is reached, SCAN calls the read-in procedure (READER) which reads the next card image.

2. Construction of the Symbol Table

a. Elements of the Symbol Table

The symbol table used in the BLISS compiler consists of two principal parts: (1) the information array (INFO) and (2) the array used to store the character representation of program elements (CHRSTOR). Figure 3 shows the elements of the symbol table and their relationships. The symbol table is used to classify program elements passed by the SCAN routine, and to record information used in the subsequent syntax analysis and code generation phases.

Data describing program entities is placed in the information array using a hash coding scheme. This simplifies both retrieval and checking for the presence of an item. A computer word (32 bits) is constructed with the first three characters (or less, if the item is less than three characters in length) of the entity in the rightmost portion of the word. The length of the element is placed in the high order eight bits of the word. The word is then divided by the prime number 127, and the remainder becomes the hash code for the element. If subsequent elements have the same hash

code address (i.e., a collision occurs), their information words are entered into the table and linked to the first entry by a chain of pointers.

Two words are constructed for entry into the information array to completely describe each entity. The first is called the scramword ("scram" is derived from "scramble code"), which is placed in the table using the item's hash code address. The first eight bits of the scramword specify the length of the entity (number of characters). The next 12 bits point to the position of the first character of the item in the CHRSTOR array. The last 12 bits of the scramword point to the location of the element's second descriptive word. The second descriptive word is called the elbatword ("elbat" is "table" spelled backwards), and is entered into the table in the area below the space reserved for scramwords. The first eight bits of the elbatword define the class of the program element, which is defined in the next section. The next 12 bits are used to store additional information for entities of the same class. The last 12 bits of the elbatword (when used) point to the scramword of a program element which has the same hash code address as the present entity.

As shown in Figure 3, the information array is segmented into several sections. The first 126 words are reserved for scramword entries. Word 127 points to the next available word in the lower section of the array, and word 128 points to the next available character location in the

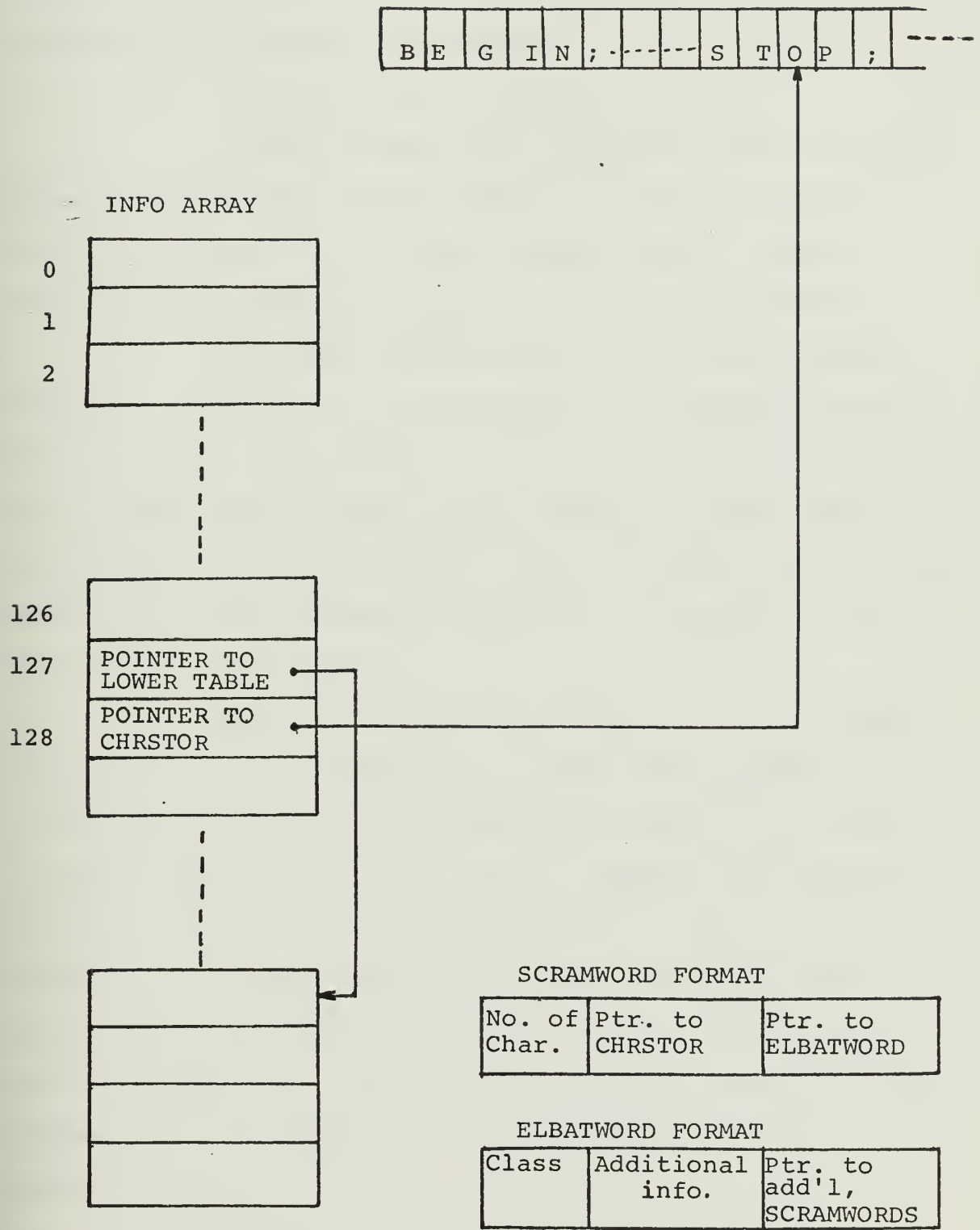


Figure 3. The Symbol Table.

CHRSTOR buffer. The remaining words are used for storing elbatwords and collision scramwords.

b. Classification of Program Elements

Program elements are assigned a specific class number in the lexical analysis phase in order to simplify identification during the syntax analysis phase. Table I lists the class numbers assigned to BLISS program elements.

The class numbers may be divided into several categories. The first 36 class numbers are assigned to BLISS reserved words. The numbering begins with four, since numbers zero through three are used by the syntax analyzer. The 11 allowable special characters are assigned numbers 40-50. Class numbers 51-54 are assigned to identifiers, integers, labels, and branches, respectively.

The additional information field in an item's elbatword is used for identifiers, labels and branches. For an identifier, it is used to specify the number of the bug or defined field, as the case may be. Numbers are sequentially assigned to bugs as they are encountered in the program, and to defined fields when they are defined. For labels and branches, the additional information field is used to store code block addresses. The method of inserting the code block addresses will be explained in the description of code generation.

c. Initialization of the Symbol Table

The symbol table is initialized with the first call on the lexical analyzer. The BLISS reserved words and

TABLE I. BLISS Language Elements.

Class Number	Element	Class Number	Element
4	ALLDUMP	30	HRSHIFT
5	DEFINE	31	DONE
6	GET	32	FAIL
7	FREE	33	BEGIN
8	COPY	34	STOP
9	DUPLICATE	35	IFANY
10	INTERCHANGE	36	IFALL
11	POINTER	37	IFNONE
12	INPCHAR	38	IFNALL
13	INPBIT	39	THEN
14	INPDEC	40	,
15	PRINTCHAR	41	+
16	PRINTBIT	42	-
17	PRINTDEC	43	*
18	STACK	44	/
19	POP	45	=
20	DFSTACK	46	¬
21	DFPOP	47	>
22	DO	48	<
23	OR	49	.
24	AND	50	;
25	XOR	51	Identifier
26	COMPLEMENT	52	Integer
27	LSHIFT	53	Label
28	RSHIFT	54	Branch
29	HLSHIFT		

special characters are specified by entering their descriptive words and character representations into the symbol table.

Initialization is done with precomputed values which were obtained using the same table entry routines that are presently in the compiler. Class identifiers were placed in the leftmost eight bits of the element's elbatwords. The values obtained are entered into the information table by the initialization routine (INIT), so that no computation is necessary.

Subsequent program elements are entered into the symbol table the first time they are encountered by the lexical analyzer.

3. Building the Lexical String

When called by the syntax analyzer, the lexical analyzer builds the lexical string for a complete BLISS statement. The elements of the lexical string are placed in an array. Figure 4 is a diagram of the sequence of events that take place in the lexical analyzer.

The TABLE procedure is the executive routine for building the lexical string. TABLE builds the lexical string one element at a time until a semicolon is encountered. Program control is then passed to the syntax analyzer, which checks the newly constructed lexical string.

The TAB array consists of a sequence of modified elbatwords, one for each element in the BLISS statement. The

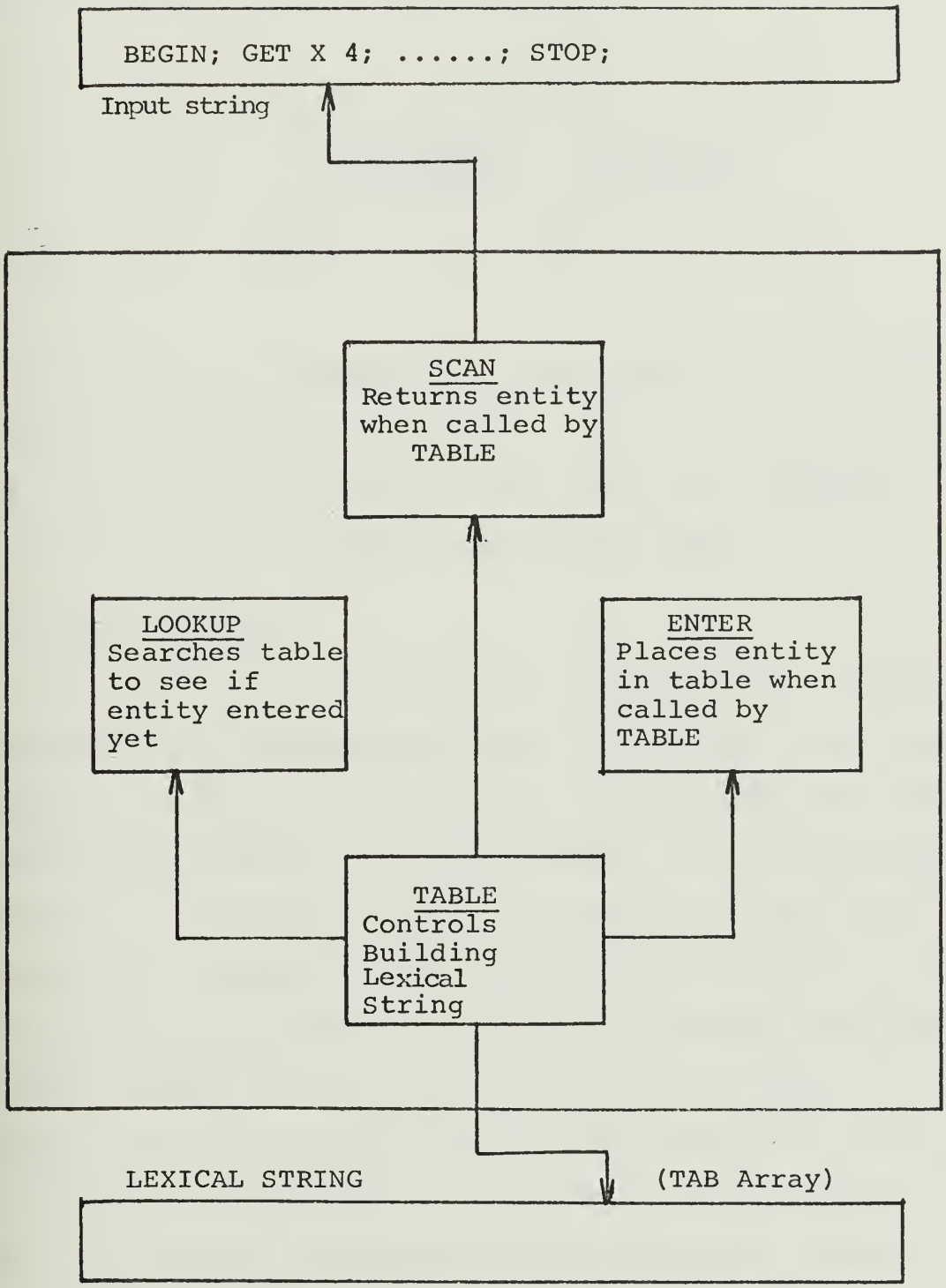
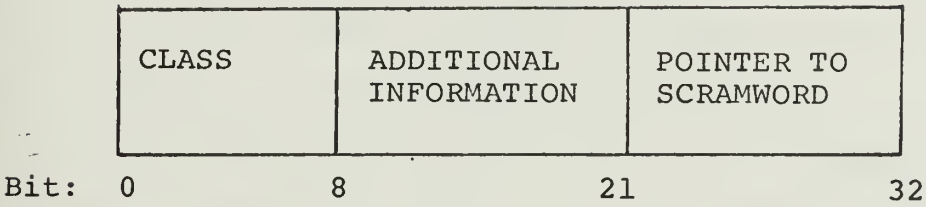


Figure 4. Lexical Analysis.

format of the modified elbatword is:



Note that the only modification to the elbatword is that the rightmost field contains a pointer to the element's scramword. Thus, the modified elbatword allows access to a complete description of the element in the symbol table.

B. SYNTAX ANALYSIS

The grammatical structure of a program must be checked mechanically for correctness before the production of workable machine code can be completed. This checking procedure is called syntax analysis. Conventionally, such analyzers are tailored to the language they are to analyze. This "hand construction" process is both complicated and tedious from the standpoint of the compiler programmer. Subsequent additions or modifications to the source language, even though readily incorporated in the formal grammar, may necessitate extensive alterations to the analyzer. The syntax analyzer may also be very time consuming, depending on the number and complexity of the productions of the grammar and the parsing algorithm used.

The syntax analyzer for BLISS is built to minimize the problems listed above. The grammar is preprocessed to allow bottom-up analysis based on unique precedence relationships between symbols. If the syntax of the program is correct this feature reduces the search for matching productions in the grammar to those that are guaranteed. The structure of the syntax analyzer need not be altered to accomodate language modification. This is due to the fact that the preprocessing is largely mechanical with automatic production of a parsing data structure for the compiler.

There are three precedence relationships possible between symbols in a grammar. In this discussion they are represented by $\doteq$, $\lessdot$, and $\gtrdot$. The well known rules for discovering these relationships are outlined below. The $\doteq$ relater is applicable when the two symbols appear adjacent to each other in a production. Considering these adjacent symbols, the rightmost symbol of any generation from the lefthand symbol will be related to the righthand symbol by the $\lessdot$ relater. The leftmost symbol of any generation from the righthand symbol will be related to the lefthand symbol by the $\gtrdot$ relater. The rightmost and leftmost symbols of productions just described are always related by the $\lessdot$ relater by convention. For example, if S:= <A> <B> , then the relationship between <A> and <B> is $\doteq$. If <A> := a, then the relationship between a and <B> is $\lessdot$. Likewise, if <B> := b then <A> $\gtrdot$ b

is applicable, and finally, the relater between a and b is ·>. The theory of precedence grammars, along with their advantages and limitations, is well covered by Floyd [Ref. 15], and more formally by Wirth and Weber [Ref. 16].

Before the discussion of grammar preprocessing and the compiler data structure, it is valuable to briefly look at the parsing algorithm and its associated working structure.

1. Parsing Algorithm

The parsing algorithm, Figure 5, is used to check the syntactic correctness of individual BLISS statements. A statement is analyzed for syntax after its representation is placed into the lexical string (TAB array) by the lexical analyzer. The parse of the statement is done in a working stack to leave the lexical string unaltered for later use in code generation. The parse requires access to the compiler data structure where the precedence relationships and the productions of the grammar are stored. In Figure 5, the symbol a_{K_T} represents the K_Tth element of the lexical string, while the symbol W_{K_W} represents the K_Wth element of the working stack. The symbol n_a stands for the length of the lexical string.

At point (1) in Figure 5, the lack of a precedence relationship between adjacent symbols in the lexical string indicates a syntax error. Not shown in the diagram, but discussed in the Programmer's Manual (Appendix A), is the recovery option available to the user. The recovery can be

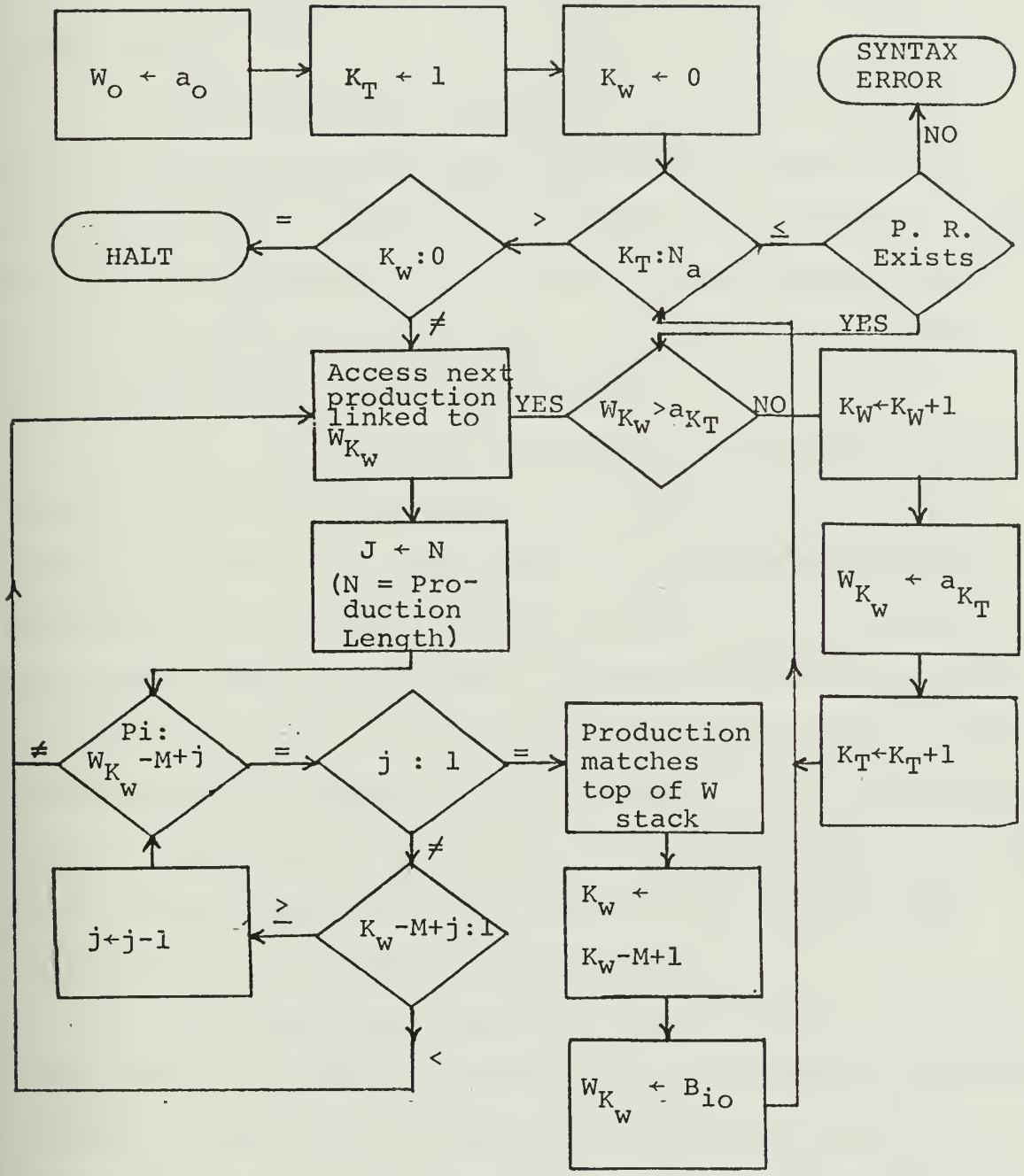


Figure 5. Precedence Parsing Algorithm

accomplished by typing in a corrected statement when a syntax error occurs.

The algorithm continues to check the precedence relationship between the top of the working stack and the next member of the lexical string until a ·> relater is found. The last tested member of the lexical string becomes the top element in the working stack when any other relater exists, as is shown at point (3) in the figure.

A ·> relationship guarantees a production substitution for an undetermined number of members of the working stack. At point (2) of the figure, the possible matching productions are accessed from the longest to the shortest. This access order is required to prevent premature acceptance of a partial string in the parsing operation. The P_i of the parsing algorithm refers to the ith member of the production which is being tested for a match. The symbol m is the length of the tested production. The length is stored with each production in the compiler data structure.

The substituted symbol becomes the top of the working stack when the production matching is successfully accomplished, as shown at point (4) of Figure 5. The replaced symbols are effectively removed from the stack and the process continues until the parse is terminated.

2. Preprocessing the Grammar

A certain amount of preprocessing must be done to put the BLISS grammar into a useful form for precedence syntax

analysis. The formal definition of the grammar is found in Addendum I of Appendix A and should be referred to for a better understanding of this section.

If a count of the number of BEGIN and STOP statements is kept the problem of syntax analysis can be reduced to that of analyzing a single statement at a time. The logical end of the program occurs when an equal number of BEGIN and STOP statements have been encountered.

The precedence relationships are discovered mechanically by using an auxiliary program that produces a precedence matrix of relationships using the grammar productions as input. This program also reveals any precedence conflicts that exist. A "conflict" refers to the case in which more than one precedence relationship exists between two symbols in the grammar. All precedence conflicts are resolved by appropriately altering the productions of the grammar.

The matrix of valid and unique relationships is available for use in the compiler after all the precedence conflicts are resolved. This matrix is, however, too large for economic storage in the conventional manner, hence sparse matrix representation is used.

3. Data Structure For Compiler Syntax Analysis

A linked list is built for each symbol in order to efficiently store the precedence matrix. The list is accessed by the class identifier number of the symbol. Each precedence relationship is allocated one node on the list if the

relationship exists between the base symbol and any other. The class number of a single related symbol is placed in that node. If more than one symbol is related then the smallest class number is placed in the node. The remaining symbols are placed in a separate storage area, ordered numerically by class number, and linked to the precedence relater node. Storage in numerical order reduces symbol search time. Precedence relationships are numbered two, three, and four, corresponding to $\cdot>$, $\doteq$ and $<\cdot$.

As is seen in the parsing algorithm, the syntax analyzer must have access to the productions of the grammar as well as the precedence relationships. The use of a production is governed by the symbol that appears on its extreme right, since the analyzer only attempts a parse when a $\cdot>$ relationship is found. The productions of the grammar are, therefore, placed in a separate storage area along with their lengths. The address of each applicable production is placed in the list linked to the right-most symbol.

Figure 6 shows a typical data structure for one symbol along with the corresponding pointers used to access the common store of productions and precedence relationships. The example shows an occurrence of the $\cdot>$ and $<\cdot$ relationships.

4. Program Description

A short outline of the program that accomplishes the syntax analysis is now presented to complete the discussion of that part of the compiler.

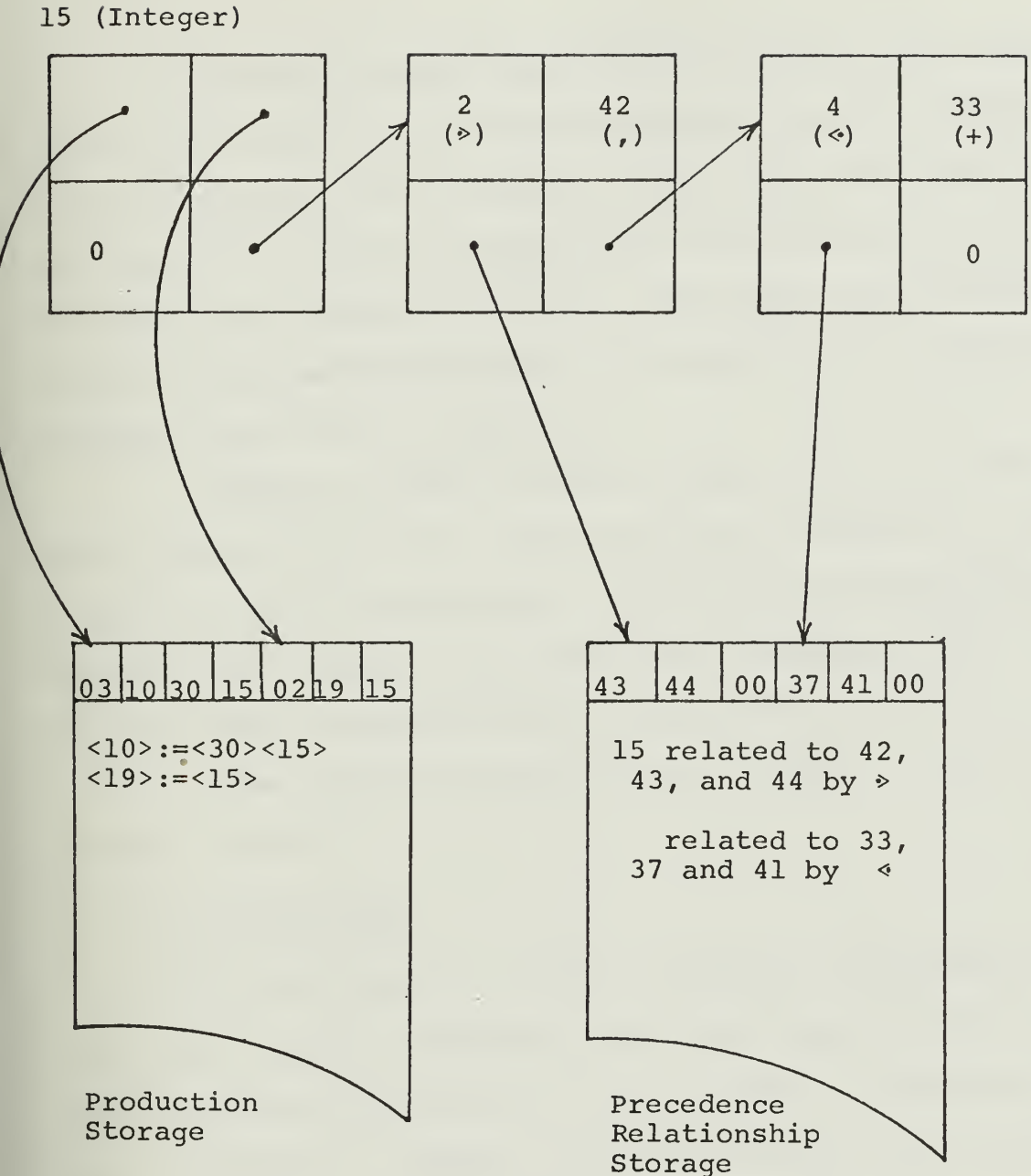


Figure 6. Compiler Precedence and Production Storage

The COMP program, Appendix C, performs the syntax analysis and acts as the executive routine for the compiler. The list structure for the symbol class identifiers and the two sequential stores is initialized from program-contained structures and character strings. The input for initialization, except the production storage, is the product of a routine, PAMBLDR, also found in Appendix C.

PAMBLDR accepts a block of data for each of the symbol class identifiers. This block may include: (1) addresses of productions in storage and (2) precedence relationships, keyed by their symbol class number and including a list of related symbols extracted from the precedence matrix.

The character string for the storage of productions is built by hand.

C. CODE GENERATION

Code generation for a BLISS statement is performed after the syntax analyzer determines the correctness of the statement. The code generation portion of the program is contained in three procedures: CODEGEN, INPROC and BRANCH, which appear in the compiler program listing, Appendix C.

The codeword for the BLISS machine (described in Chapter V) consists of a six-bit operation code field plus two 13-bit fields used to specify values or locations in the machine. Appendix B contains a listing of the 43 machine instructions used, with a description of each.

As codewords are generated, they are placed in a 2000-word array (CODE). Upon reaching the logical end of the program, the non-empty portion of the CODE array is built into a text module by the executive routine.

The basic features of the code generator and some of the more interesting methods used in the code generation will now be described.

1. Code Generator Program Structure

The code generator builds the code segment for a BLISS statement by using the lexical string which has been checked by the syntax analyzer. The class descriptor field of the elements in the lexical string is used to key the operation of the code generator.

There is a program segment within the code generator for each reserved word and special character. Branching to the appropriate code generation segment is accomplished by allowing the class descriptor numbers of the elements to correspond to an array of labels which is defined within the code generator program.

Since the lexical string is known to be syntactically correct, the code generator can anticipate the possible sequences of parameters which must accompany an operator. For example, in the COPY operation, it is known that the elements which follow COPY must either be identifier identifier or identifier integer. Since the possible parameters can be anticipated, code for a complete operation or test can be generated quite easily.

Internal procedures within the code generator allow retrieval of data from fields and retrieval of integer values from the symbol table, since these operations are conducted frequently.

2. Method of Filling Branch Addresses

The additional information field (bits 9-20) of a label's elbatword in the symbol table is used to hold the address of the label entry in the code array. If a GOTO label is encountered before the label is reached, the label is entered into the symbol table and the first bit of the additional information field is set to "1." The remaining eleven bits contain the code block address of the unfilled branch statement. If additional branch statements are encountered before the label, a chain of pointers is maintained in the rightmost 13-bit field of the codeword for the branch statement. Figure 7 shows the chaining from the symbol table to the branch statements in the code block.

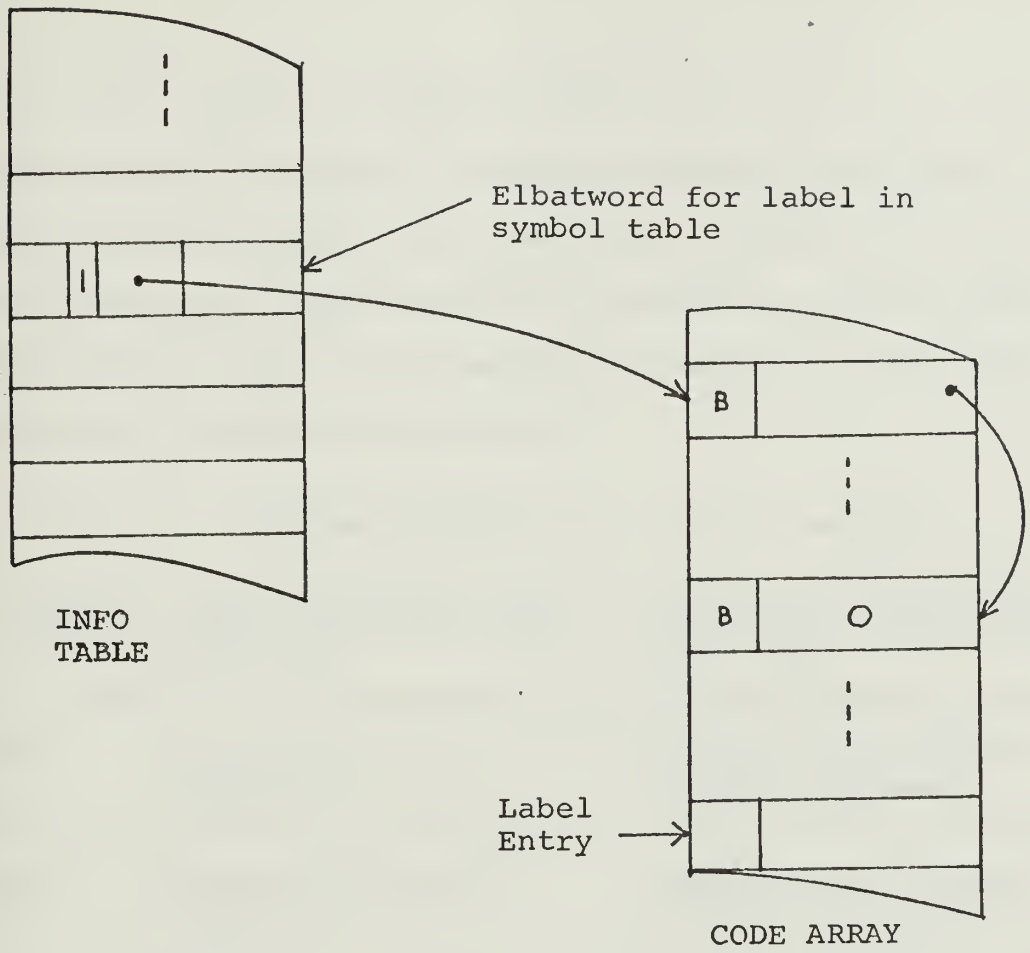


Figure 7. Branch Statement Linking Method

When the label in question is encountered by the code generator, bit nine in the elbatword is set to "0," and the label address is filled in the elbatword and in all unfilled branch statements in the chain.

3. Method of Handling Conditional Expressions

The code segment for a series of tests preceded by a conditional is constructed to allow immediate branching to either the operations or to the next statement, depending upon whether the given condition is satisfied. For example, in the statement

IFALL X=0, Y.DATA>7, Z.DATA<1, THEN.....;

the code generator inserts a "branch on false to the next statement" after each of the three tests. Thus, if all of the tests are true, the code for the operations which follow the "THEN" is executed. The method for handling each of the conditionals is presented in TABLE II.

TABLE II. Conditional Expression Branch Methods.

CONDITIONAL	BRANCH INSERTED AFTER EACH TEST	ADDITIONAL BRANCH AFTER LAST TEST
IFANY	Branch if true to operations	Unconditional branch to next statement
IFNALL	Branch if false to operations	Unconditional branch to next statement
IFALL	Branch if false to next statement	NONE
IFNONE	Branch if true to next statement	NONE

The code block addresses of the operations and of the next statement are inserted into the appropriate branch statements in a manner similar to that used with labels. A chain of back-pointers to the unfilled branches is maintained, and when the operations and end of statement are reached, the appropriate branch addresses are filled.

V. THE BLISS MACHINE AND PROGRAM EXECUTION

A simulated machine designed as a dedicated computer which is a part of BLISS seems more valuable than using an existing machine. The BLISS machine is therefore designed to efficiently accomplish the list processing tasks. The fictitious hardware is specified to complement primitive list processing operators. The machine is simulated by an interpreter that is written in a higher level language.

The BLISS interpreter, written in PL/I, has advantages over other possible implementation methods. First, a certain amount of machine independence is achieved. Secondly, primitive list processing operators may be specified as machine functions. This reduces the complexity of the code generation and allows efficient machine execution. Finally, the alteration of the machine structure and machine operations is attainable with a minimum of programming effort. This last point was important in the experimentation carried out during implementation. It will be of even greater importance to experimenters who desire to alter the system at a later date. The single disadvantage of using an interpeter is that it may not achieve the speed possible by using the macro-type implementation of Knowlton. The effect of this limitation has not been investigated.

Certain methods proved most successful in the investigation of machine implemented list processing operators. These methods are discussed below and include: (1) the ability to use a pushdown stack with virtually no concern for size of the stack

or depth of recursion (accomplished without allocating extensive amounts of storage to the process), (2) the use of an efficient free storage allocator that automatically maintains free storage in the largest blocks of contiguous words possible, consistent with the system maximum block size, and (3) the representation of the machine in a single storage block with dedicated storage locations for machine hardware. The third method allows monitoring of machine execution and provides the option of saving the entire machine for continued execution at a later time.

The sections that follow describe the basic machine operations, the structure of the machine, the PL/I programs which comprise the machine, and program execution.

A. BASIC MACHINE OPERATIONS

The basic BLISS machine operations are described in order to provide insight into the elements of the machine structure. The first operation is storage allocation which is basic to almost all procedures. Next, the design of a pushdown stack is considered, followed by a discussion of the operations possible on a pushdown. Finally, typical register operations are mentioned.

1. Storage Allocation

The BLISS storage allocator is patterned after work done by Knowlton [Ref. 17]. Other than the difference in word size and location of the block size indicator (both required by the characteristics of the machine), there is little

difference in the two allocation systems. Knowlton's system allows the user to specify the size of the largest block. Unlike L[6], BLISS fixes the maximum block size at 128 words.

Blocks are available from free storage in eight sizes, corresponding to the powers of two from zero to seven. A request for a block returns the next largest size, if the size requested is not a power of two. The first bit of the zeroth word of each block is used by the system to indicate whether or not the block is in use. The next three bits of the zeroth word indicate the size of the block, converted to a logarithm to the base two. For example, if the first four bits of the zeroth word of a block are "1011" then the block is in free space and is eight words long.

Upon machine activation, the available free space is broken into blocks of 128 words and double-linked to form a list. This list is attached to a header cell of one word that contains a pointer to the list and a count of the number of cells on that list. Header cells are available for each of the other possible block sizes, but are initially zeroed.

When a block is requested from free storage, it is supplied by taking the first block on the list that is attached to the appropriate header cell. If no block of the requested size is available, it is provided by a block-splitting process. A block from the first available list with free blocks is recursively split in half until a block of the correct size is produced. Blocks are supplied with all usable space zeroed.

Figure 8 shows the configuration of free storage during execution of a program in which some space has already been used.

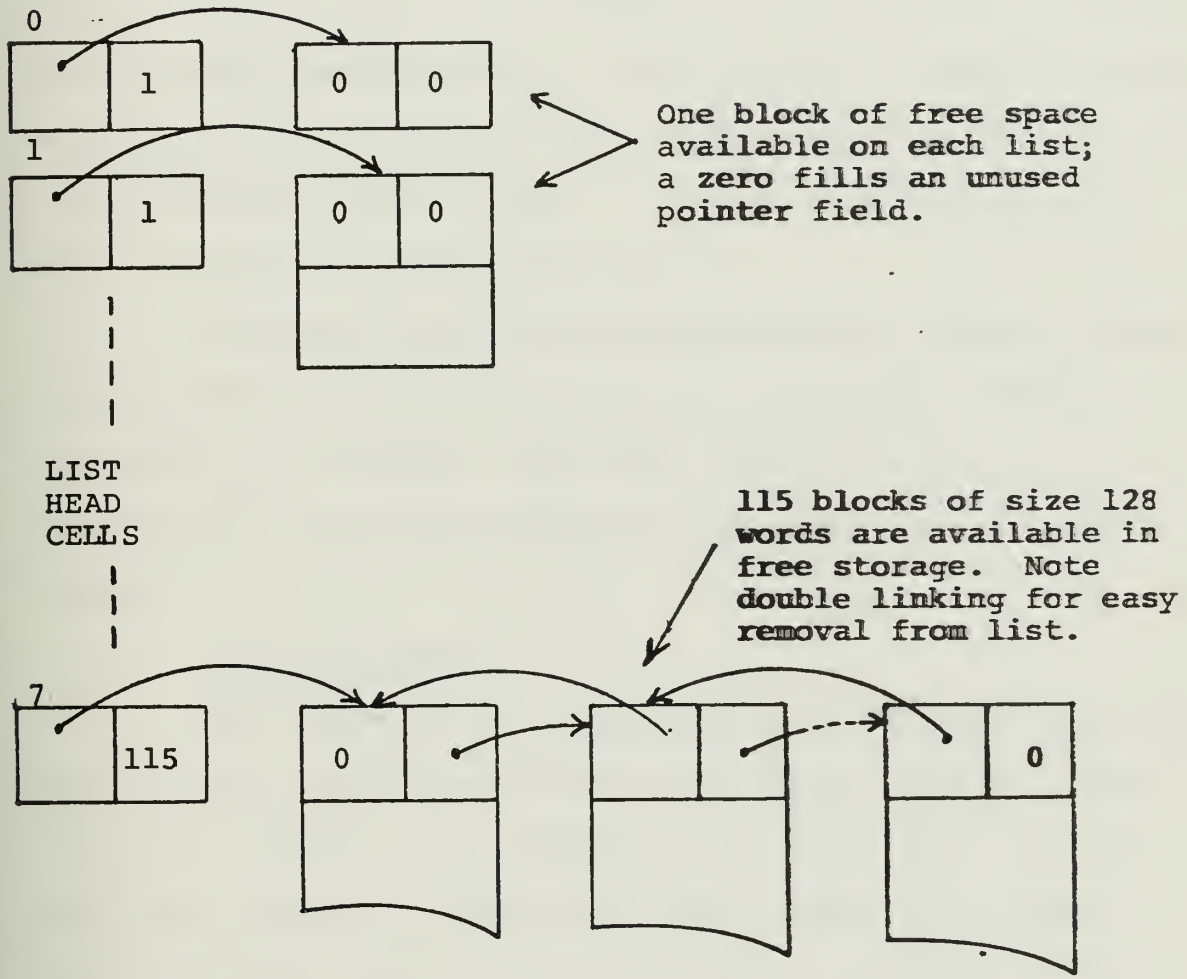


Figure 8. "Free Space Memory Map"

When a block is returned to free storage, the usable space is zeroed and the status of its mate block is considered. A "mate" to a block is a block of storage of the same size Mates are contiguously located in memory. The result of combining the mate blocks reverses the splitting which originally

separated them into smaller blocks. The status of a mate block can be determined, without actually locating the block, by inspecting the "use bit." If the mate block is not in use, it is removed from its free space list and combined with the newly returned block. The freeing procedure is then reentered with the newly combined block. The process of combining blocks continues until the largest system block is recovered or until a mate is found to be "in use." At that time the block is added at the head of the proper list.

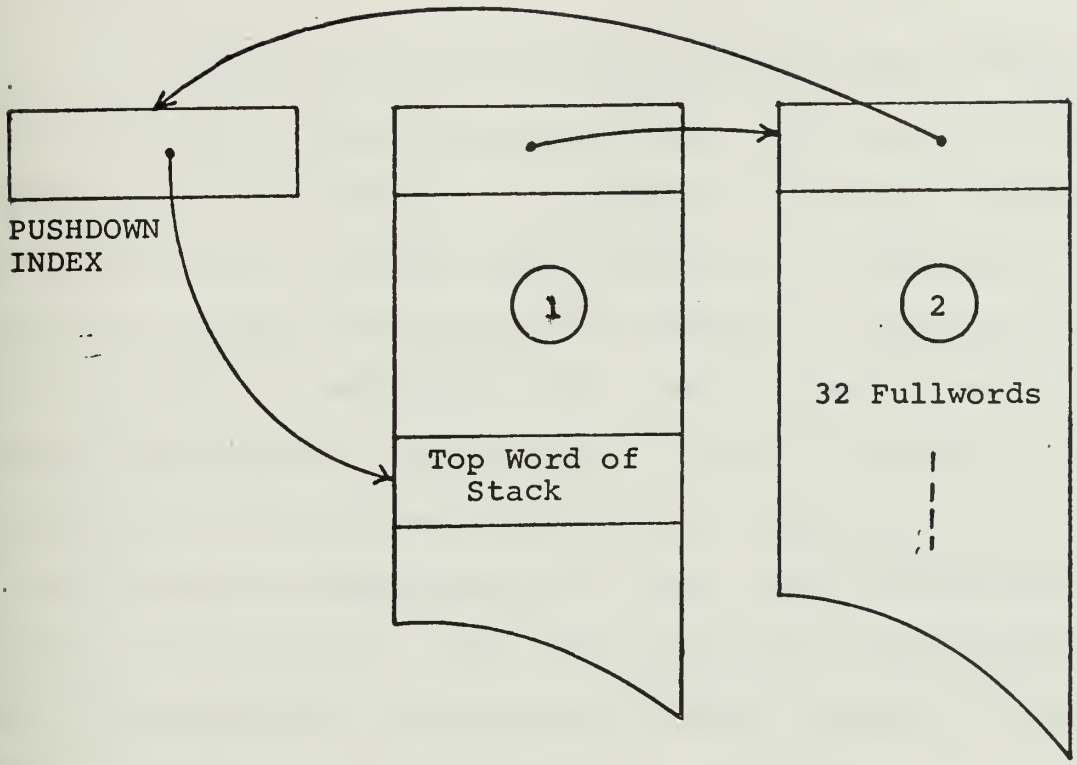
Storage allocation requires minimal dedicated storage and processing time for splitting and combining. The user can balance the need for free space against the cost of returning it, since the freeing of storage is controlled by the user.

2. Pushdown Stacks

The structure and operation of a pushdown stack is basic to most of the operations of the BLISS machine. When allocating storage for a pushdown stack, most systems must accept the trade-off between efficient stack size and the possibility that the stack will overflow. The BLISS stacks are designed to dynamically change their size as storage requirements change. Therefore, they will effectively never overflow, unless all free storage is used up.

The following is a discussion of BLISS machine stack operation. A 32 word block is obtained from storage when a stack is first accessed. The top of the stack is located by

a special pushdown index word. This word contains a zero before a pushdown is activated. A new block of 32 words is obtained and linked to the original block when a stacking call would overflow the pushdown. This "back" linking is all that is needed to operate the single pushdown system. If the stack is "popped" so that a block is no longer part of the pushdown, the unused block is returned to free storage, unless it is the only remaining block in the pushdown. This exception saves time in most applications since most trivial stacking operations empty the pushdown often. An attempt to "pop" an empty or unactivated stack will result in a BLISS machine interrupt with a corresponding error diagnostic. Figure 9 is a diagram of an active stack showing the index pointer and the block-linking pointers.



The block labeled 1 contains the most recent stack elements. Block 2 contains older elements of the same stack.

Figure 9. BLISS Stack Configuration

The procedure for handling system and user stacks represents a machine where the most recent portion of the stack is in hardware registers. When this space is filled, it is swapped to core storage and placed in a block for later retrieval. Only that portion of the stack that would be in the hardware is displayed in the machine dump (Section C - Chapter V).

Of the six pushdown stacks in the BLISS machine, two are dedicated to input/output handling. The extent of their operation as I/O buffers is to receive and return information, but they are not available for access by the programmer. The other four stacks are used by the system to accomplish arithmetic, logical, subroutine return, data transfer, test, and branch operations. Two of them, the Field Contents and Field Definition pushdowns, are available for programmer use. For a more complete understanding of these stack-oriented operations see Appendix B, The BLISS Interpreter Instructions, and the appropriate sections of the INTEP program in Appendix D.

3. Register Operations

The system registers are used only for temporary storage, shifting and pointer-chaining operations since most machine functions are carried out on the pushdown stacks. The registers reserved for bugs are dedicated storage, but are not used to perform system functions.

B. STRUCTURE OF THE MACHINE

The BLISS machine is shown in outline form in Figure 10. The BLISS machine occupies an array of 16,384, 32-bit words. The first 128 words are reserved for system use. Each of the machine elements is discussed below so the reader may gain insight into the basic machine operations.

BLISS MACHINE

0 . . . 7	AVAILABLE SPACE LIST - HEADER CELLS.
8	ACTIVE INSTRUCTION REGISTER
9	SYSTEM
10	REGISTERS
11	SYSTEM PUSHDOWN INDEX
12	FIELD DEFINITION PUSHDOWN INDEX
13	FIELD CONTENTS PUSHDOWN INDEX
14	SUBROUTINE RETURN PUSHDOWN INDEX
15	INSTRUCTION COUNTER
16	INPUT BUFFER INDEX
17	OUTPUT BUFFER INDEX
18 . . . 40	BUG REGISTERS (23)
41 . . . 127	FIELD DEFINITIONS (87)
128 . . . 16,383	FREE SPACE

Figure 10. BLISS Machine.

1. Available Space List Header Cells

The available space list header cells occupy eight full-words. Each word contains two pieces of information. The first halfword contains a pointer to the first cell on the list of free space cells of a particular size. The last halfword holds a count of cells on the list. A header cell equal to zero denotes and empty list.

2. Active Instruction Register

A fullword register contains a pointer to the memory location of the presently executing instruction. This is the Active Instruction Register.

3. System Registers

Two registers are assigned for system use. One use of these two registers is to perform shift operations which are generally used to extract field data or perform bit manipulations. The registers are also used by the system to chain along a pointer path to a block during a load or store operation.

4. Pushdown Indices

The four pushdown indices hold pointers to the memory locations of the top of their respective stacks. An index register value equal to zero indicates that the pushdown has never been activated.

5. Instruction Counter

Each instruction in the user program text file is given a sequential number. This number is placed in the Instruction Counter register when the instruction is executed.

6. Input/Output Buffer Indices

The input/output buffer index registers hold pointers to the present buffer index location. As in the other stacks, a zero index indicates that the buffer has not been activated.

7. Bug Registers

Twenty-three registers hold the contents of the programmatically defined bugs. A register is assigned to a bug sequentially as the bug name appears in the program. This assignment process is discussed in Chapter IV.

8. Field Definitions

Each of 87 registers is designed to contain the definition of a single field. The field definition is an ordered triplet of integers that is determined from the user program and completely describes the field. Field definitions are discussed in Chapter IV.

C. MACHINE PROGRAMS - INTERCONNECTION AND EXECUTION

This section provides a brief explanation of the operation and interconnection of the programs that make up the BLISS machine. For a detailed look at program execution, see the program comments in The BLISS Interpeter, Appendix D.

1. INTEP

The INTEP program is the executive routine for the machine. As mentioned previously, available space is initialized upon machine execution. The text module of the user program is then read into the machine memory using as many 128-word blocks as necessary. The active instruction pointers are set to the

beginning of the program. The program execution loop is initiated and sequentially extracts the instruction code of the active program text word, branching to the INTEP program block for that instruction. Instruction indices are advanced on return from each program block. A termination instruction code causes exit from the program execution loop.

The individual machine instructions consist primarily of primitive list processing operations. They are designed to be executed by using the machine elements. Thus an accurate trace of program execution is possible by observing the change in BLISS machine registers and stacks.

2. OUTDIAG

The OUTDIAG routine is called when a BLISS program error causes the machine to interrupt. The machine diagnostic is output at the terminal and a full machine dump is executed by the DUMP routine. At present, all possible errors are not covered by BLISS diagnostics. No memory protection is provided to prevent alteration of the machine elements by a program addressing error. As a substitute, the INTEP program calls for a full BLISS machine dump on a PL/I error condition. The PL/I error diagnostic is written into a "SYSPRINT" file in CMS. The BLISS machine is terminated on occurrence of a PL/I error condition.

3. DUMP

The DUMP routine may be called both by the BLISS programmer through the ALLDUMPS instruction and by the machine

as previously noted. The machine dump resulting from either call is written into a file, "LISTING," in the current implementation.

There are six elements produced on a full machine dump. The number of elements produced may be varied by the user. The dump elements are: (1) the free space list header cells, (2) the block that contains the active user program instruction, (3) the contents of the system registers, (4) the contents of the bug registers, (5) the field definitions, and (6) the contents of the most recent block in each of the four pushdown stacks.

4. BRANCH

Program branches are handled by the BRANCH routine. BRANCH locates the containing block, then the machine address of the required instruction.

5. COUNTBT

In the present implementation, the COUNTBT routine locates the first "1" bit of a field or register.

6. GET_CEL and FRE_CEL

The GET_CEL and FRE_CEL procedures handle the free storage manipulation described in Section A of this chapter. A "MEMORY EXHAUSTED" diagnostic and machine termination occur if free storage is used up.

7. PUSH and POP

The PUSH and POP routines handle the stack operations for the four machine pushdowns and the input/output buffers.

The input routine is initiated from POP when a request for input is received and the input buffer is empty. The output buffer is automatically dumped to the terminal when it is full. This operation is initiated in the PUSH routine.

8. INP and OUTP

The INP and OUTP procedures form the interface between the BLISS machine and the computer on which it is running. In the current implementation, input can be taken from the terminal or from a disk file. Output is available only at the terminal.

VI. POSSIBLE EXTENSIONS AND CONCLUSIONS

Only through use and experimentation can any new programming system be optimized. At the time of this writing, BLISS is still going through this proofing stage.

BLISS in its present form should prove to be invaluable for its intended purpose, but there are several extensions and improvements which would be beneficial. This chapter contains a discussion of possible extensions to BLISS and presents concluding remarks.

A. POSSIBLE EXTENSIONS

The possible extensions to BLISS fall into two categories. The first category of improvements is considered to be those that directly enhance the primary goal of the system, or those that are of direct benefit to the user. The second category consists of those improvements that are of academic interest (i.e., machine optimization and optimization of the compiler), although they are of benefit to the production user.

1. Direct System Improvements

Some possible extensions to the system which would directly benefit the user are presented in this section.

a. Saving Data Structures

As noted earlier, one highly desirable extension to the system would be a facility allowing the data structures created in a run to be saved for future use. This could be easily accomplished by saving the whole machine, but this would be inefficient if only a small portion of the machine's free memory space were in use. A more desirable method of

saving the structures would be to extract them from the machine in the form of a module which could be reconstructed at a later time.

b. Conversion to Batch Processing

Conversion of BLISS to allow batch processing in addition to on-line use would be invaluable for two primary reasons. First, on-line execution of long programs with a great deal of input/output is very time consuming. On-line compilation could be utilized to check the program, but execution in a batch environment would be beneficial. The second instance in which batch processing would be particularly useful occurs when on-line terminal time is at a premium.

2. Improvements of an Academic Nature

Design optimization was of secondary consideration during system construction. Now that the system is a working reality, there are several areas in which interesting sub-problems can be developed.

a. Machine Design

The BLISS interpreter is built to perform the basic system tasks in an efficient manner. Hence, the BLISS machine was conceived after the language operations were defined. The present configuration has not been sufficiently measured to conclude that the machine design is optimum. A thorough study of both the BLISS machine design and the software which represents the machine would be beneficial. The interpreter programs allow easy modification of the machine

structure, and the modularity of the individual operations lends itself to experimentation.

b. Code Generation

There are several areas within the code generation section of the compiler which could be optimized. Within the generating program (CODEGEN) there are many sections where similar operations are performed which could be combined into internal procedures. By varying the calling parameters the appropriate code for several similar operations could be produced by the same procedure.

A detailed study of the code generation should be closely linked with machine design optimization.

c. Improved Features for Language Structural Changes

In the present implementation some of the language preprocessing requires hand manipulation of the data between execution of program parts. In addition, there is no programmatic connection between the preprocessor and the compiler which uses the preprocessor output to do the precedence-directed syntax analysis. As a result, language structural changes are time-consuming, even though the methods are straightforward.

A user interested in machine and language experimentation could add the following improvements to the BLISS compiler. First, the precedence analyzer program should have access to an initialization program to construct the initial

compiler data structure. The combined program would then require the productions of the language as input data, and would construct the associative memory for the parsing algorithm. This combined program could possibly be connected to the compiler routines and be called at the user's option for the initialization process.

B. CONCLUSIONS

The primary value of BLISS is that it provides a language and a system that are dedicated to minimizing confusion in basic list processing applications. The concepts used in the implementation are of secondary value, but do provide some original ideas.

Since the simplicity of BLISS is the feature which makes it useful as a basic list processing language, any proposed changes should be carefully weighed, lest they detract from the primary objective of the system.

APPENDIX A

BASIC LIST-ORIENTED INFORMATION STRUCTURES SYSTEM

(BLISS)

PROGRAMMERS MANUAL

FROM A THESIS BY:

LCDR C. S. THORELL, USN

and

LT W. O. POTEAT, JR., USN

NAVAL POSTGRADUATE SCHOOL
1970

TABLE OF CONTENTS

I. INTRODUCTION

The introduction to basic abstract computing structures is generally a straightforward learning process. There is nothing conceptually overpowering about "linked lists" or "binary trees." The student can readily visualize the solutions to many of the problems dealing with such entities.

The situation immediately deterioriates when one tries to write a program that handles the problem the way he visualizes it. There are several choices available to the programmer. At one end of the spectrum, he can learn a difficult language with complex notation, such as LISP 1.5. Alternatively, he can build his own processing system out of a basically non-list-oriented language, such as FORTRAN IV. Either way the conceptual learning process must slow down until the student can hurdle the programming problems.

BLISS is a computer system that is designed to handle information structures problems at the tutorial level. It is based upon L^6, a programming language designed by Kenneth C. Knowlton at the Bell Telephone Laboratories. L^6 is cryptic in nature and just above the assembly code level. However, entities of BLISS parallel the vocabulary of list processing, so that the user may express himself with a minimum of translation.

The BLISS machine (interpreter) is configured to execute programs by utilizing hardware (simulated) that is designed for list processing applications.

II. DATA STRUCTURES

In BLISS, as in other list processing languages, the basic entity is the node or block. A node is simply a set of contiguous words in storage. The program variables are specified by overlaying these nodes with a template of defined fields. In many other languages, either the node is size-limited, or the template is fixed, or both. In BLISS the restrictions are much less severe. The size of the nodes and the configuration of the templates may vary widely. There is no requirement to completely define structures at compile time, as this may be done programmatically.

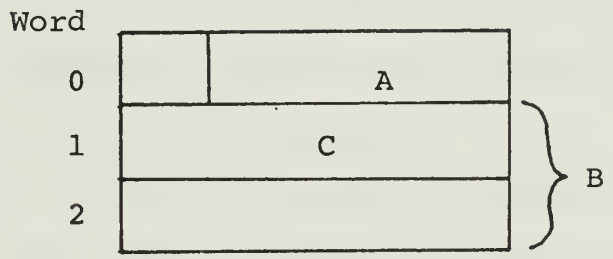
A. BLOCKS

Blocks, or nodes, may be specified as a contiguous set of 2^n fullwords, where n can take on values from zero through seven. Over 16,000 fullwords are available in the BLISS machine's free storage, but the amount available to the user varies with program size and complexity.

B. DEFINED FIELDS

A BLISS program may have a maximum of 87 fields that are defined at any one time. Fields are named by the concatenation of alphanumeric symbols, the first of which must be a letter. By defining fields, the programmer describes the template he desires for overlaying blocks in the structure. Fields may be specified as all or part of a single word, or as a concatenation of several fullwords in the node. The latter option is restricted to character data, but any other defined field may contain

character, bit, or decimal information. Fields may overlap and contain other fields. The following illustrates a three-field template overlaying a three-word block:



The words in a block are numbered consecutively, beginning with zero. The first four bits of word zero are restricted for system use and should not be specified within a user field. (See the "DEFINE" operation for specification requirements.)

C. POINTERS

Any field long enough to hold a memory address may contain a pointer to a block. Blocks are addressed by the memory location of their first word (word zero). Fields defined as large as a fullword and as small as 14 bits in length may be used as pointer fields.

D. BASE FIELDS (BUGS)

There are twenty-three permanently allocated fullword registers, called bugs, reserved for programmer use. They are assigned names sequentially at compile time. If more than 23 bugs are used in the program, they overlay previously assigned

names. For example, if the first bug used is name ALPHA and the twenty-fourth is name USE, then both names reference the same register. The names used for these bugs can be any concatenation of alphanumerics (first symbol a letter). Bugs are recognized by the compiler from the context in which they are used, and consequently need no user definition. A field and a bug may not have the same name.

A bug may be used to store any information that will fit into 32 bits. When they are used as pointer fields, bugs may "crawl" about the data structure, hence the name. Access to lists and structures is achieved through a pointer bug, as will be seen in the next section. In Figure 1, as in other illustrations, a bug is depicted by a circle with the name inside.

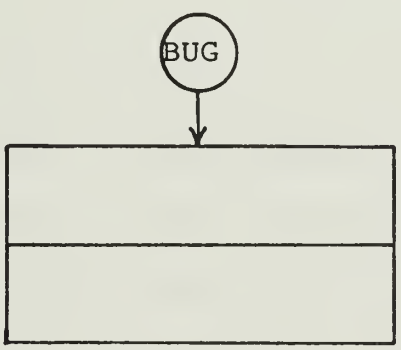


Figure 1

E. REFERENCING FIELDS AND LINKING NODES

The content of a bug is referenced by stating its name. If a bug contains a pointer to a node, the contents of any field in that node may be referenced by concatenating the

field name to the name of the bug, using a period as a delimiter. Remote fields are referenced by concatenating the linking pointers until the node containing the field is accessed.

In Figure 2, the contents of field A1 are accessed by BUG.A1, while A3 could be referenced by BUG.B.B.A3. Bug.B.C... might be the link to branch to an associated structure.

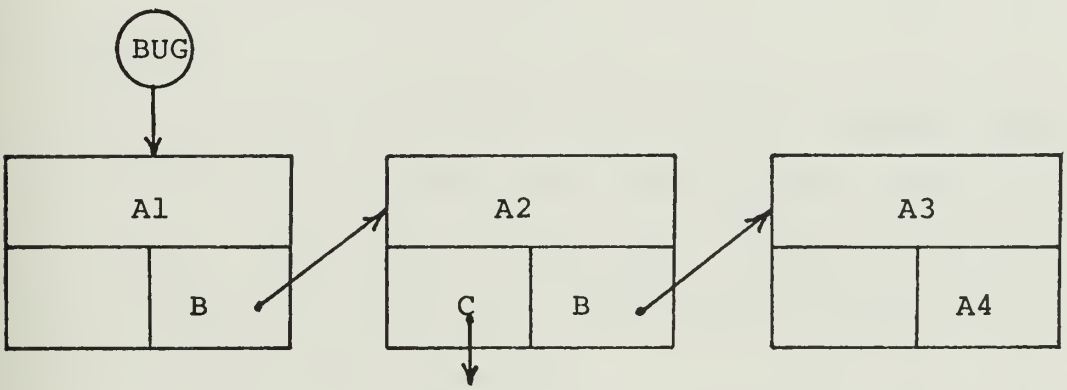


Figure 2

Normally a pointer will contain the address of the first word of a node. However, the programmer may increment or decrement such a pointer to move the template. As long as he is conscious of the node size and the present location of the field in that node, he can conduct a detailed search of the data. Figure 3 shows an example of an operation which moves the template by incrementing the pointer X1.

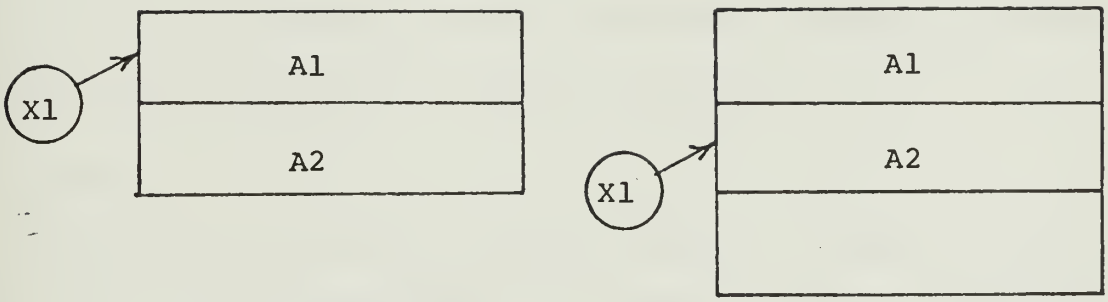


Figure 3

F. LITERALS

Decimal integers are the only allowable compile time literals. Bit and character data must be read into the program with standard input/output operations. No special flag for literal integers is required. Their use is explained further in the discussion of operations.

III. PROGRAM STRUCTURE AND SYNTAX

Addendum I to this manual gives the complete BNF syntax for BLISS. This section explains the overall program structure and the syntactic elements within a program.

A. SYNTAX

A program has a block structure similar to that of PL/I or ALGOL, but is less complicated. The main program is contained between the initial BEGIN and final STOP statements. The instructions of the main program are executed sequentially, with a branching option available.

From the syntax, one can see that the main program may contain any number of imbedded programs. There is no machine restriction on the number of contained programs or on access to them from the main program or other contained programs. Program flow is basically up to the programmer. Backing out of imbedded program blocks is his responsibility.

B. SUBROUTINES

All imbedded program blocks are labeled subroutines. The subroutine name is the label appended to the first statement following the "BEGIN;" that separates the subroutine from the rest of the program. The following shows a sample program structure with one subroutine, "SUB1."

```
BEGIN;
        ----------
        ----------
        ----------
        BEGIN;
        SUB1:
              -------
              -------
              ------- DONE;
        STOP;
        ----------
        ----------
        DO SUB1;
STOP; -----
```

Without a label defining its name, a subroutine block can never be entered. Without a termination statement, program control cannot be transferred correctly. In this case, "DONE" returns control to the calling location. Termination is discussed in greater detail below.

The physical location of the subroutine is not important. It may be placed at any point in the program where it is logically pleasing to the programmer.

BLISS subroutines allow complete recursive entry to a level which is bounded only by the free storage available. The system only stacks the return locations associated with each recursive call. If other variable values are to be saved, the programmer must do it using system stacks which are available for this purpose.

Logical termination of a subroutine is accomplished through the operations "DONE" and "FAIL." "DONE" returns program control to the statement following the calling statement.

"FAIL" transfers control to a label designated in the calling statement. Since a "FAIL" termination may bypass some or all levels in a recursive routine, the programmer must be aware of the consequences of this method of exit. The following illustration is a sample program structure that demonstrates the termination alternatives:

```
BEGIN;

LAB:

    BEGIN;
    SUB1:

        FAIL;
        DONE;
    STOP;
DO SUB1 LAB;

STOP;
```

The main program block may be called from within itself, but no system termination is provided for endless loops. The user is required to consider the program flow and to insure that a logical termination occurs (the program reaches the STOP statement in the main block).

IV. THE STATEMENT: THE BASIC ELEMENT IN THE PROGRAM STRUCTURE

As can be seen from the BNF definition of BLISS, the individual program blocks are made up of groups of statements. The statements are built of allowable combinations of syntactic components. The six syntactic components of a general BLISS statement are:

<LABEL> <CONDITIONAL> <TEST SET> THEN <OPERATION SET><BRANCH>

Each of these components are discussed separately in the sections that follow. Amplifying statement examples accompany each section. Sample programs are presented in Chapter V. The allowable combinations of syntactic components are defined in the formal definition of BLISS.

Statement length is limited by implementation to approximately 25 tests and operations. A statement may be placed free-field on one or more card images (columns one to 72), or there may be more than one statement per card image.

A. LABELS

A label is an alphanumeric string with a letter as the first symbol. It is located at the beginning of a statement and is followed by a colon. Labels are used to mark an entry point for branch statements or to name subroutines. Only one label per statement is allowed.

B. CONDITIONALS

There are four conditional keywords that are used with tests to produce an "if.....then...else" type result. "IFALL" is satisfied if all of the specified tests in the statement

are true. The abbreviation "IF" may be used in place of "IFALL." "IFANY" requires that at least one of the tests be true. "IFNONE" is satisfied if none of the tests are true. Finally, "IFNALL" (if not all) requires that at least one of the specified tests be false. A conditional is not required in a statement, but if it is included at least one test and the "THEN" are required.

C. TEST SET

A <TEST SET> consists of one or more elementary tests, separated by commas. The tests are infix expressions, and as one would expect, result in a bit by bit comparison. The operators are: = , ¬ , > and < (equal, not equal, greater than and less than). The quantities tested may be either field or bug contents for the first operand, and may be field contents, bug contents or an integer for the second operand. For example, in the following statement:

IFALL X = 7, X.A1.C >W.B2, THEN...

if the contents of the bug X equal 7 and the contents of field X.A1.C is greater than the contents of field W.B2, then the conditional is satisfied and the operations following the "THEN" are executed. If the conditional is not satisfied, control transfers to the next program statement.

D. OPERATIONS

An <OPERATION SET>, consisting of one or more operations, can either stand alone as a statement or be used as the "action

part" of a statement which contains a conditional. All operations (except arithmetic) are prefix-type instructions with up to four operands following the keyword. Commas separate strings of operations unless the last element in a statement is an operation, in which case the end of statement semicolon suffices. Individual operands are separated by one or more spaces.

The user will find that improper field use within an operation will lead to program errors that the machine will often fail to detect. For instance, field overflow may not be detected until the contents are used again or printed out. It must be remembered that multiple word fields are for character data only.

In the following sections, each operation is described in sufficient detail for programmer use.

1. Arithmetic Operations

Arithmetic operations are of the form:

a (operator) cd

The allowable operators are +,-,* and / (add, subtract, multiply and divide). The result of the operation is stored in the first operand. Operand a may be a field or a bug, while cd may be a field, bug, or integer. For example,

W.A + X.B1.C

adds the contents of the fields and stores the result in field W.A.

2. Machine Dump Operation

The machine dump operation:

ALLDUMPS i i

is a debugging tool that selectively displays parts of the BLISS interpreter. The six parts may be called by inserting the correct octal number in the operand locations. For example,

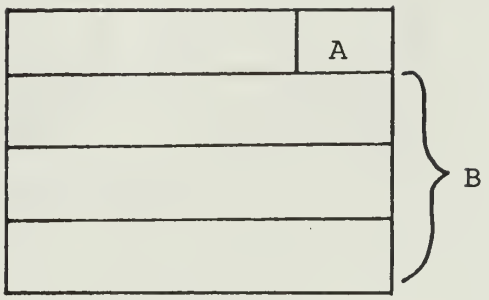
ALLDUMPS 7 6,

would call for parts one through five of the option. Consider $(76)_8 = (111110)_2$ for insight. Further use of this tool is discussed in the section on debugging in Chapter VI.

3. Define Operation

The operation:

DEFINE f cd_1 cd_2 cd_3 is used to specify the fields in which program data is placed. f is the field name. The cd_i, where i = 1,2,3, are either integers or the contents of bugs or fields. cd_1 is the number of the word in the node in which the field is located or begins. cd_2 may specify the leftmost bit of the field, or if zero, indicates that field cd_3 specifies the number of fullwords desired (character data only). Otherwise, cd_3 specifies the last bit in the field. For example, in Figure 4, field A is defined in word zero as an eight-bit field, beginning with bit 25 and ending with bit 32. Field B is defined as a three word field beginning at word one in the node.




```
DEFINE A 0 25 32,
DEFINE B 1 0  3;
```

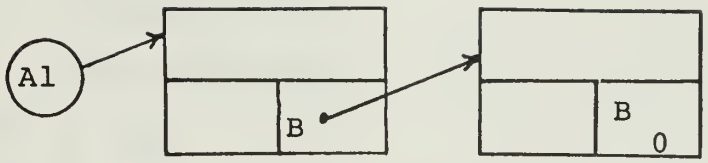
Figure 4

4. Get Operation

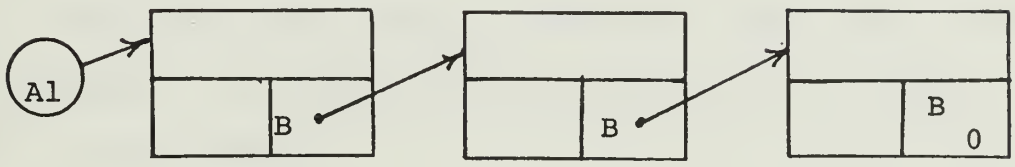
The get operation is used to fetch nodes from free storage. All the words of a new node are zeroed, except bits one through four of word zero, which are not available to the programmer. The operation requires two or three operands, depending upon the action desired. In the operation

GET $\underline{a}_1$ $\underline{cd}$ $\underline{a}_2$,

$\underline{a}_1$ is the field or bug which will point to the new block. $\underline{cd}$ is either an integer or the contents of a field or bug which specifies the size of the node required. $\underline{a}_2$ is an optional operand which, if specified, indicates the field or bug into which the original contents of $\underline{a}_1$ will be placed. In Figure 5 a new node is added to the list which is accessed by bug A1. The pointer to the new node is placed in bug A1 and the original pointer from bug A1 is put in field A1.B.



"Get A1 2 A1.8;"



(New mode)

Figure 5

5. Free Operation

The operation

FREE a c,

is used to return to free storage those parts of the data structure no longer needed in the program. No other "garbage collection" is provided in BLISS. It is up to the programmer to maintain maximum free storage when space is critical. Operand a specifies the field or bug which points to the node to be returned. Operand c is an integer or the contents of a field or bug which will replace the pointer in a after the node is freed. For example,

FREE A1 A1.B,

would restore the structure in Figure 4 to its initial form.

6. Copy Operation

The operation

COPY a cd,

assigns the integer or the contents of the bug or field specified in operand cd to the field or bug a.

7. Logical Operations

There are four logical operators: "OR," "AND," "XOR" and "COMPLEMENT." The operation performed is as the name implies. In the format

operator a cd,

a is a field or bug contents, while cd may be a field, bug or integer. In "OR," "AND," and "XOR" the specified operation is performed on the two operands and the result is stored in a. The COMPLEMENT operation complements the second operand and stores the result in the first.

8. Shift Operations

Using the two available shift operations "LSHIFT" or "RSHIFT," the user may shift the contents of a field of bug left or right, and optionally replace the vacated positions with either zeroes or the contents of a field or bug. The format for the shift operation is:

<Shift Operator> a cd_1 cd_2,

Operand a is the bug or field to be shifted. cd_1 is an integer, bug, or field which specifies the number of bit positions of the shift. cd_2 is an optional operand that specifies the integer, bug, or field to be shifted into the vacated bit

positions. If cd_2 is not specified, the vacated positions become zero. There is no program interrupt for a field overflow.

An example of a shifting operation is shown in Figure 6. The left three bits of field BUG.B are left shifted into the right of field BUG.A. The left three bits of BUG.A are lost. This operation also provides an "end-around" shift by allowing a field to be shifted into itself.

```
                "LSHIFT  BUG.A  3  BUG.B,"

        1 0 1 0 1    1 1 1 0 0 0   Before Shift

        0 1 1 1 1    1 1 1 0 0 0   After Shift
             From B
          A              B
```

Figure 5

9. Stack Operations

The BLISS machine has two pushdown stacks available to the user. The Field Contents Pushdown (FC) is accessible by using operations "STACK" and "POP." The operation

STACK a,

puts the contents of the field or bug a on the FC pushdown. The operation

POP a,

takes the top entity of the FC pushdown and stores it in field or bug a.

The other available pushdown is the Field Definition Pushdown (FD). The operation

DFSTACK f,

stacks the triplet that defines the field f on the FD pushdown.

The field f may then be redefined. The operation

DFPOP f,

takes the top triplet from the FD pushdown and uses it to define or redefine the field f.

WARNING. An attempt to pop either pushdown when it is empty results in a terminal error.

10. Do Operation

The "DO" operation is used to execute subroutines. There are two operands in the format:

DO <subentry> <failexit> ,

The subentry operand is the label appended to the first line of a subroutine. The second operand, failexit, is the label name optionally provided to specify the program branch if subroutine execution encounters the operator "FAIL." A normal subroutine termination occurs when the operator "DONE" is encountered, and control is returned to the point immediately following the "DO" operation in the invoking procedure.

11. Input Operations

Program input is accomplished via an input buffer that is 128 characters in length. This buffer may be filled from a file of card images or directly from the terminal. The input source is designated as a second parameter in the "execution command," which is discussed in a later section. The parameter keywords for specifying the input source are "CARDS" and "TERM."

When an input operation is executed, the input buffer is accessed to retrieve the number of characters specified. If

the buffer is emptied before the input is completed a new input is required from the user-designated source.

Regardless of the source, the input format requirements are the same. Input strings are accessed in groups of four characters (32 bits) due to the stack construction of the buffer. Each input string must be specified as a multiple of four characters. There is no requirement to completely fill the buffer on each input.

The following discussion of the three possible input operations references the card or terminal image in Figure 7. Note the "*", which is an "end-of-input" marker for each card or terminal line. The symbol "_" represents a blank space in Figure 7.

12____49BOOKS___1011011_*

Figure 7

The operation:

INPDEC a 2,

is used for integer input and will take the first buffer group "12__," convert it to binary, and store it in the field or bug a. The same operation will input "49" from the next group.

INPCHAR a 8,

is a similar input operation for characters. It will take the next two groups (8 characters) from the buffer and store them unconverted in field a (a bug will only hold 4 characters). Characters must be left justified in the buffer group that begins the string for input into a fullword field. The

operation

INPBIT a 7,

is an example of bit string input. The final two buffer groups of Figure 7 are converted to bits and stored in the field or bug a.

The examples shown neglect the case where a character string might be stored in a field smaller than one fullword. In that instance the one to three characters required must be right justified in the buffer group.

Generally, every input operation utilizes at least one buffer group. Decimal input or bit input is free-field within the group or groups, but character strings are not.

12. Output Operations

BLISS output is accomplished via a 128 character buffer that prints exclusively at the terminal. When the buffer is full it dumps to the terminal. There is no available method to dump a partially full buffer but this feature can be circumvented by padding the buffer with blanks.

The format for the decimal output operation is

PRINTDEC a #,

The bit string found in field or bug a is converted to decimal characters and output to the buffer. The string is left justified and padded to the right with blanks to total # characters (# must be an integer multiple of four for all output operations). The operation

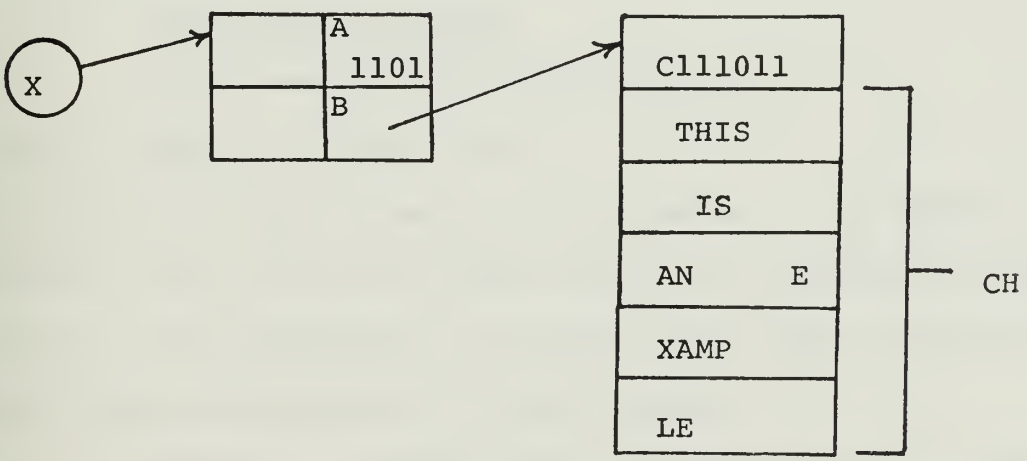
PRINTCHAR a #,

outputs the character strings from field or bug a padded with blanks to total a length of # characters. The following operation:

PRINTBIT a #,

outputs a bit string from field or bug a. The string is converted to character symbols and padded with blanks to a length of # characters.

As an example of output, consider the structure in Figure 8. The code string shown in statement (1) produces the output in line (2). Notice that the entities are left justified in the first group of each output operation. Buffer dump is accomplished by adding the required 96 blanks to the last operation. Output could have been achieved in any of the operations by padding to 128 total characters.



(1) PRINTBIT X.A 8, PRINTDEC X.B.C 4, PRINTCHAR X.B.CH 116;

(2) 1101___59__THIS_IS_AN_EXAMPLE__....
(96 blanks)

Figure 8

13. Duplicate Operation

The operation

DUPLICATE a c,

makes an exact copy of the node pointed to by the field or bug c and puts a pointer to the new node in field or bug a. The contents of the node as well as the size are duplicated. This operation should be used sparingly as it requires several machine instructions to accomplish the task.

14. Interchange Operation

The operation

INTERCHANGE a_1 a_2,

exchanges the contents of field or bug a_1 with field or bug a_2. Field sizes must be identical, as either truncation or a terminal error will occur if the user attempts to interchange fields of different sizes. Fields larger than one fullword are not allowed in this operation.

15. Branching Operations

The branching operations are:

GOTO <label> , DONE, FAIL

The latter two have already been discussed. The "GOTO" instruction must be the last operation in a statement (it may also be a legal statement in itself) and <label> must be a label name used somewhere in the program.

CAUTION. A branch into a subroutine via a "GOTO" will cause a terminal error when the "DONE" or "FAIL" instruction is executed.

V. PROGRAM EXAMPLES

The following two sections present sample BLISS programs. The program listings are followed by explanatory comments.

A. POSTORDER TRAVERSAL OF A BINARY TREE

```
BEGIN;
     DEFINE NAME 0 9 32, DEFINE LLINK 1 1 16,
     DEFINE RLINK 1 17 32;
     BEGIN;
          LEFT:  GET BUG.LLINK 2,  INPCHAR BUG.LLINK.NAME 3,
          DONE;
     STOP;
     BEGIN;
          RIGHT:  GET BUG.RLINK 2, INPCHAR BUG.RLINK.NAME 3,
          DONE;
     STOP;
     GET BUG 2, COPY HEAD BUG, INPCHAR BUG.NAME 3;
     DO LEFT, COPY BUG BUG.LLINK, DO LEFT, DO RIGHT;
     COPY BUG HEAD, DO RIGHT, COPY BUG BUG.RLINK, DO RIGHT;
     COPY BUG HEAD, COPY ZERO 0, STACK ZERO;
  T1:  IFALL BUG = 0, THEN GOTO T3;
  T2:  IFALL BUG.LLINK = 0, THEN GOTO T4;
          STACK BUG, COPY BUG BUG.LLINK, GOTO T2;
  T3:  POP BUG;  IFALL BUG = 0, THEN GOTO END;
  T4:  PRINTCHAR BUG.NAME 4;
 END:     COPY BUG BUG.RLINK, GOTO T1;
STOP;
```

This first example builds a binary tree with a node size of two words and traverses the tree in postorder. As each node is visited its name is printed at the terminal. The node template is shown in Figure 9.

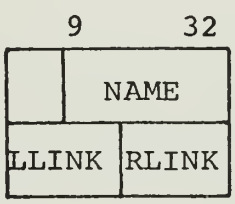


Figure 9

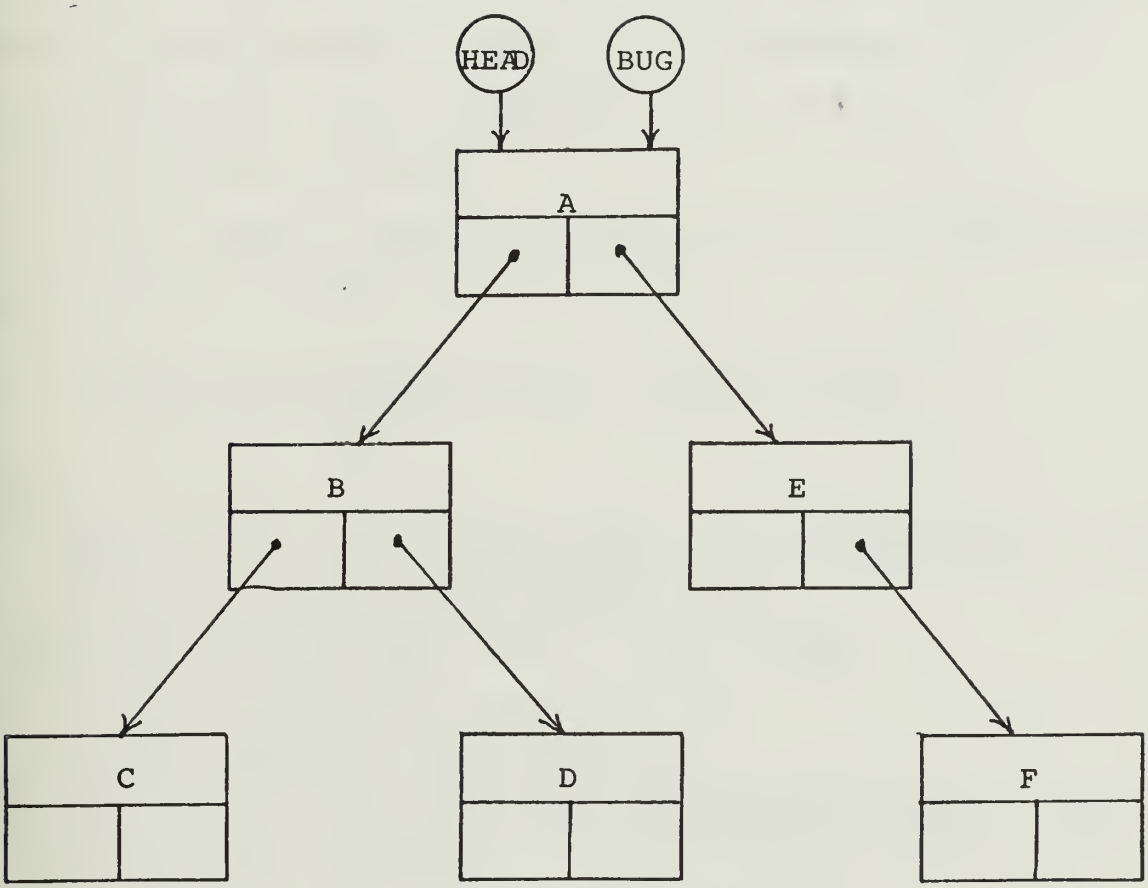


Figure 9

The input card image for the program is:

...a...b...c...d...e...f*

where "." denotes a space. Figure 10 shows the tree structure and the associated bugs as they are configured before traversal commences at label T1. Note that a zero is used as a pushdown marker, since an attempt to pop an empty pushdown results in a terminal error. The program outputs the string:

...C...B...D...A...E...F.0

The zero is output to dump the print buffer, but a blank field could have been created for this purpose.

B. FINDING THE MINIMUM PATH IN A DIRECTED GRAPH

```
BEGIN;
    DEFINE DIST 0 5 18, DEFINE LINK 0 19 32, DEFINE FLAG 1 1 1,
    DEFINE CPTR 1 19 32, DEFINE CITY 2 0 2, DEFINE NEXT 1 2 18,
    DEFINE CIT1 2 0 1, DEFINE CIT2 3 0 1;
    GET DEST 4, GET ORIG 4, INPCHAR ORIG.CITY 8,
  DUPLICATE ANCH ORIG,GET MILES 1, COPY W ANCH;
IP2:  INPDEC MILES.DIST 4;
    IFALL MILES.DIST = 0, THEN GOTO WORK;
    INPCHAR DEST.CITY 8, COPY TAG 0;
IP1:  IFALL W.LINK ¬ 0, THEN COPY W W.LINK, GOTO IP1;
    GET W.LINK 2, COPY W.LINK.DIST MILES.DIST, COPY Q W.LINK;
    DO LOOK, COPY Q.CPTR W;
    IFALL TAG = 0, THEN INTERCHANGE ORIG DEST, COPY TAG 1,
    GOTO IP1;
    INPCHAR DEST.CITY 8,  DO LOOK, INTERCHANGE ORIG DEST,
    GOTO IP2;
    BEGIN;
    LOOK: COPY W ANCH;
       L1:   IFALL W.CIT1 = DEST.CIT1,
         W.CIT2 = DEST.CIT2, THEN DONE;
         IFALL W.NEXT = 0, THEN GET W.NEXT 4, COPY W W.NEXT,
                    COPY W.CITY DEST.CITY,DONE;
         COPY W W.NEXT, GOTO L1;
    STOP;
WORK:  INPCHAR DEST.CITY 8, COPY X W,
    DO MEASURE;
    GET R 8, DEFINE RA 1 0 3, DEFINE RB 4 0 4;
    INPCHAR R.RA 12, INPCHAR R.RB 16;
```

```
     PRINTCHAR R.RA 16, PRINTDEC T.DIST 112;
    PRINTCHAR R.RB 128;
  B1:  COPY T T.LINK;
    IFALL T ¬ 0, THEN PRINTCHAR T.CPTR.CITY 12,
         PRINTDEC T.DIST 116, GOTO B1;
    BEGIN;
    MEASURE: IFALL X.CIT1 = DEST.CIT1, X.CIT2 = DEST.CIT2,
              THEN GET T 1, DONE;
    IFALL X.FLAG = 1, THEN GET T 1, COPY T.DIST 999, DONE;
    COPY X.FLAG 1, GET B 1, COPY B.DIST 999
 JP1:  IFALL X.LINK = 0, THEN COPY FLG 1, DONE;
    COPY X X.LINK, STACK X, STACK B,COPY X X.CPTR,
    DO MEASURE, POP B, POP X;
    IFALL FLG = 1, THEN COPY X.CPTR.FLAG 0, COPY FLG 0;
    T.DIST = X.DIST, DO ADD;
    IFALL T.DIST < B.DIST, THEN COPY Y B, COPY B T, DO RET,
         GOTO JP1;
    COPY Y T, COPY T B, DO RET, GOTO JP1;
    BEGIN;
    ADD: IFALL T.LINK = 0, THEN DUPLICATE T.LINK X,
         COPY T.LINK.LINK 0, DONE;
         COPY Q T.LINK;
     AD1:  IFALL Q.LINK ¬ 0, THEN COPY Q Q.LINK GOTO AD1;
         DUPLICATE Q.LINK X, COPY Q.LINK.LINK 0, DONE;
    STOP;
    BEGIN;
     RET:  IFALL Y = 0, THEN DONE;
        COPY Q Y.LINK, FREE Y Q, DO RET, DONE;
    STOP;
    STOP;
STOP;
```

For this example the nodes of a directed graph correspond to the names of cities and the connective links correspond to the distance between the cities. The BLISS program accomplishes the following:

(1) Reads in the city pairs:

(city 1) (dist) (city 2)*

| 8 | 4 | 8 | spaces

(2) Builds an associative structure shown in Figure 15.

(3) Upon encountering a card with distance equal to zero, computes the minimum distance between the two cities on that card.

(4) Outputs the minimum distance, the minimum path, and the length of each minimum path leg.

The template defined in the program, Figure 11, is used for all program nodes. The nodes vary in size from one to four words.

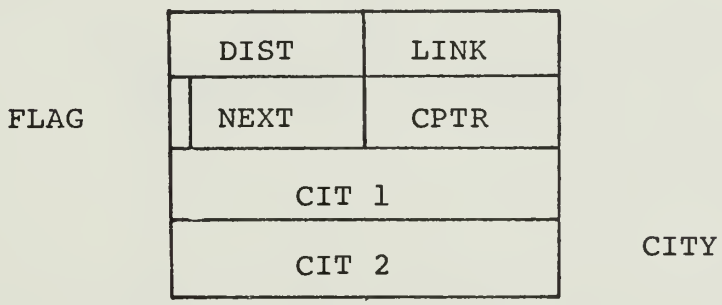
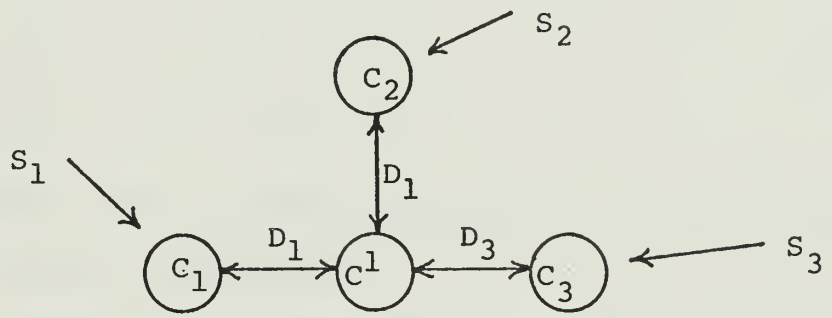


Figure 10

The display of template independence from a specific node or class of node is an important feature of this example and should be studied carefully.

The subroutine MEASURE is a recursive route-finding procedure. To find the shortest route from a city C' to a city C one need only consider the neighboring cities about C'. Let the neighboring cities (directly connected cities) be C_1, C_2, and C_3, with corresponding distances D_1, D_2, and D_3 from C'. If the shortest distance from each of the neighboring cities to the destination C is known then the solution is

straightforward: let S_1, S_2, and S_3 be the shortest distance from C_1, C_2, C_3 (respectively) to C. This is shown in the following diagram:



The shortest route from C' to C is then the smallest of

$D_1 + S_1$

$D_2 + S_2$

$D_3 + S_3$

To take a concrete example, suppose one wishes to fine the shortest route from KINGSTON to BRISTOL (See Figure 16). In this case.

C' = KINGSTON

C_1 = DETROIT

C_2 = BRISTOL = C

C_3 = JONES

By examining the graph of Figure 12, the shortest route from BRISTOL to BRISTOL is 0 miles ($S_2 = 0$), the shortest route from DETROIT to BRISTOL is 37 miles ($S_1 = 37$) (not considering going

back through KINGSTON), and the shortest route from JONES to BRISTOL is 40 miles ($S_3 = 40$)(again not back through KINGSTON). The values for D_1, D_2, and D_3 can also be determined by examining the graph of Figure 12. Thus:

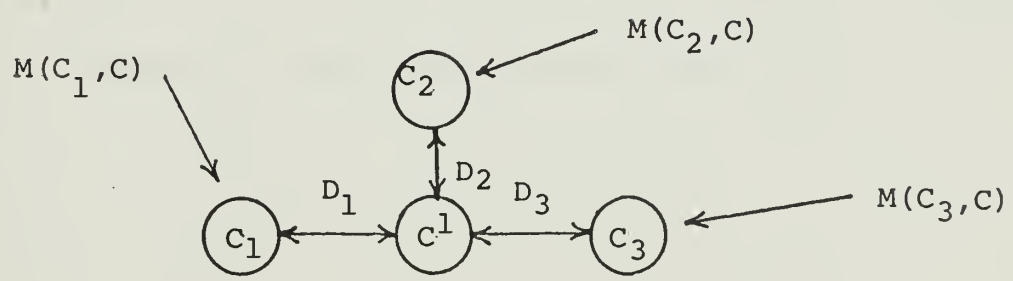
$$D_1 + S_1 = 15 + 37 = 52$$

$$D_2 + S_2 = 10 + 0 = 10$$

$$D_3 + S_3 = 20 + 40 = 60$$

and it follows that the shortest route (of length 10 miles) is KINGSTON directly to BRISTOL.

The obvious problem is that the shortest route from each of the neighboring cities is unknown. Here is where the recursive procedure is useful. MEASURE finds the shortest route from a city P to a city Q (denote this result by M(P,Q)). It can equally be applied to find S_1, S_2, S_3. Suppose, as above, the problem is to find the shortest route from a city C' to a city C. In the place of S_1, S_2, and S_3 compute $M(C_1,C)$, $M(C_2,C)$, and $M(C_3,C)$. This changes the previous diagram to:



Then M(C',C) is the smallest of:

$D_1 + M(C_1,C)$

$D_2 + M(C_2,C)$

$D_3 + M(C_3,C)$

There is one additional problem. While currently examining a city C' for the shortest route to a city C, one does not want a route which winds around and eventually comes back through C'. In Figure 12, for instance, it would do no good to go from KINGSTON to JONES to CARMEL, and back through KINGSTON. By running up a flag at city C' all routes which come through C' from any other city are ignored when M(C',C) is computed. After computing M(C',C) the flag is taken down.

Note that in MEASURE a best route to date is kept in a list pointed to by bug B and stacked, then popped and compared as the recursion backs out. On finally returning to the point of original call the bug T will contain a pointer to the list of cities in the shortest path. Scanning this list provides the program output.

Figure 13 contains the complete input for the example with a search for the shortest distance from DETROIT to CARMEL specified. Figure 14 is the program execution result.

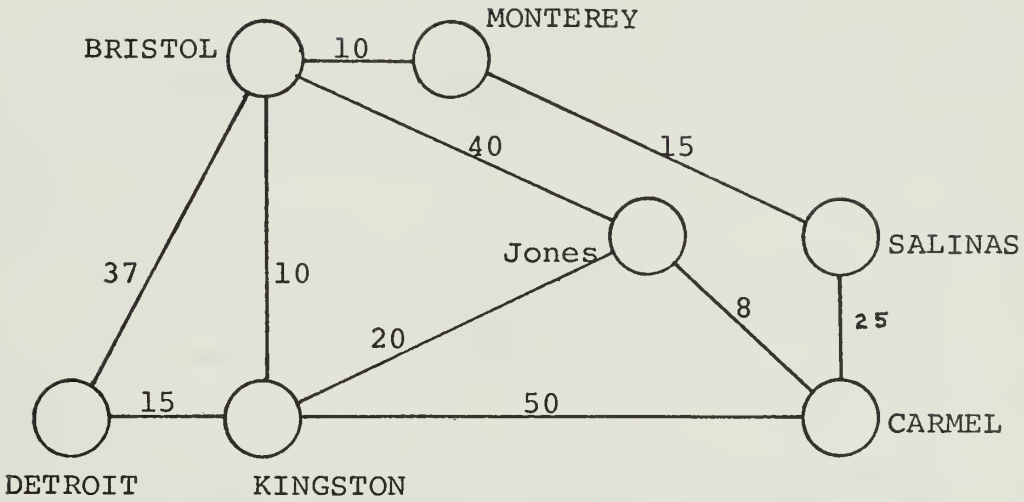


Figure 12

```
DETROIT    37BRISTOL *
DETROIT    15KINGSTON*
BRISTOL    10KINGSTON*
BRISTOL    40JONES   *
KINGSTON   20JONES   *
BRISTOL    10MONTEREY*
MONTEREY   15SALINAS *
SALINAS    25CARMEL  *
KINGSTON   50CARMEL  *
JONES       8CARMEL  *
DETROIT     0CARMEL  *
MINIMUM DIST*
CITY         DIST*
```

Figure 13

```
ss cities cards

MINIMUM DIST    43
CITY         DIST
CARMEL       8
JONES        20
KINGSTON     15
R;  T.............
```

Figure 14

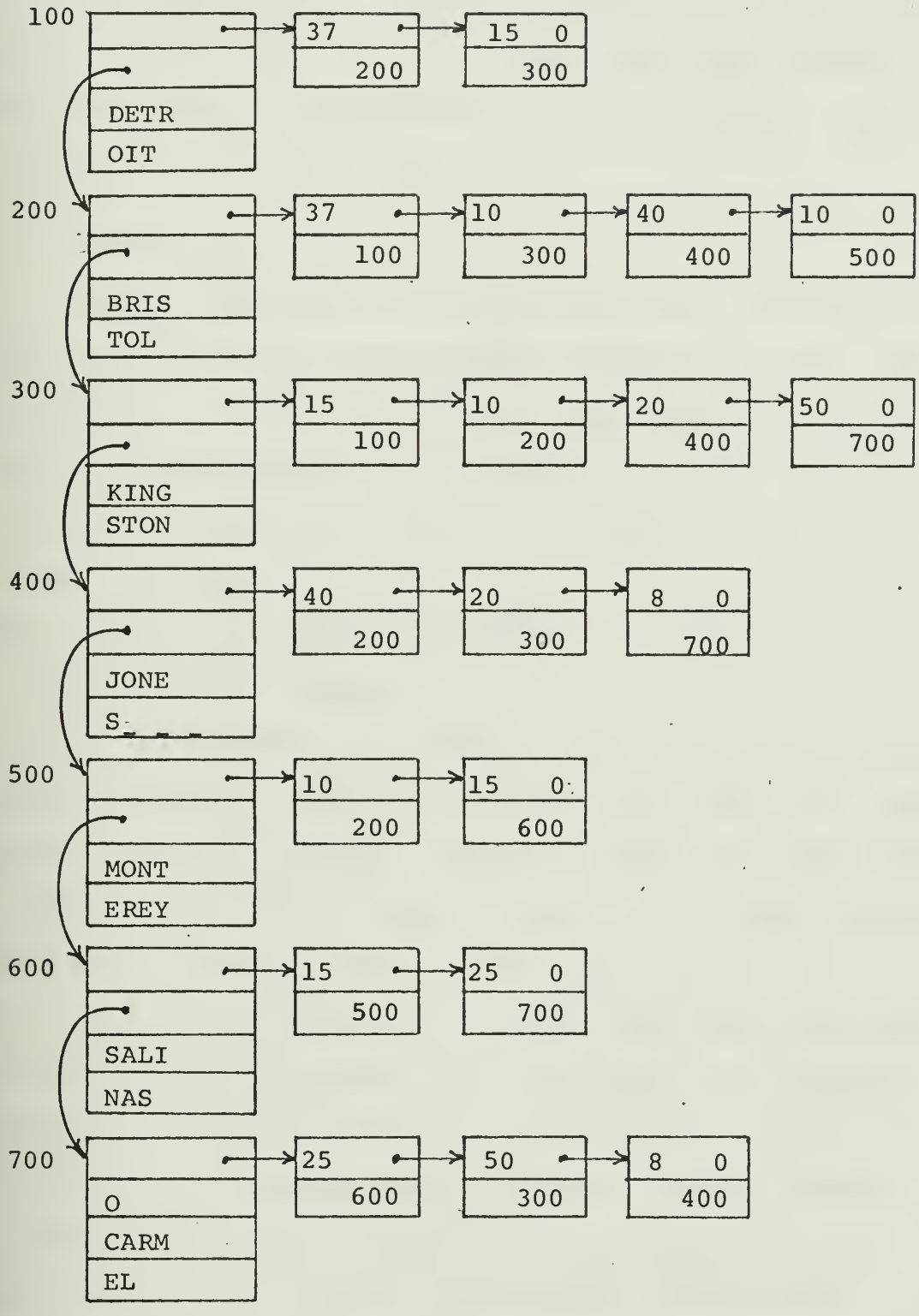

100
37
200
15 0
300
DETR
OIT
200
37
100
10
300
40
400
10 0
500
BRIS
TOL
300
15
100
10
200
20
400
50 0
700
KING
STON
400
40
200
20
300
8 0
700
JONE
S_ _ _
500
10
200
15 0
600
MONT
EREY
600
15
500
25 0
700
SALI
NAS
700
25
600
50
300
8 0
400
O
CARM
EL

Figure 15

VI. SYSTEM USE

The BLISS SYSTEM, presently implemented on the CP/CMS time sharing system, is easily used. Both the compiler and the interpreter are maintained as system modules. They are, therefore, accessed by name.

A. COMPILATION

Before compilation the user must create a program file. The file may be free-field within columns one to 72. The "filename" is created by the user, and restricted to 8 characters on the CP/CMS system. The "filetype" is "BLI." After creating the file, compilation is begun by typing:

```
BLI  <filename>
```

The compiler will cause the terminal to output:

```
PROGRAM LISTING DESIRED?
```

A "YES" answer will produce a line for line program listing at the terminal. Any other response will result in a compilation without the listing. Compiler output is in the form of a "TEXT" file with the same "filename" as the input program. This file is used as input to the interpreter.

If a syntax error occurs during compilation, the last line of the erroneous statement will be output at the terminal. The compiler will then request the line number of the desired correction. If termination is desired, the user returns a zero, otherwise the number of the line containing the error. Next, the compiler will request the corrected line be input. After the corrected line is input, the compiler requests the number of the first line in the erroneous statement. After the user

responds, compilation will recommence at the beginning of the corrected statement. The user's program file is automatically corrected when he inputs the replacement line.

B. EXECUTION

To execute the program "TEXT" module the user must issue the command:

ss <program name> <parameter>

The parameter is either "CARDS" or "TERM" and designates the source of input for the program. If a card file is desired, it is built using the same "filename" as the associated program with a "filetype" of "DATA." If terminal input is chosen, the statements are input when the interpreter displays:

***
_

All output is at the terminal and no user action is necessary.

C. DEBUGGING

The BLISS system provides an automatic dump when most terminal program errors occur. Also, a user-called dump (see ALLDUMPS) is available. If the error results in a PL/I diagnostic a "FILE SYSPRINT" is created. None of these tools are particularly helpful unless the user is familar with the BLISS machine and the associated psuedo-assembly language. For sophisticated work in BLISS the user is advised to study the explanation of these diagnostic tools in the reference

thesis. Most problems should be solvable without reference to these system functions.

ADDENDUM I. A FORMAL DEFINITION OF BLISS

<PROGRAM> := <BEGIN SET> <STATEMENT SET> <END SET>

<BEGIN SET> := BEGIN ;

<END SET> := STOP ;

<STATEMENT SET> := <STATEMENT> | <STATEMENTS SET> <STATEMENT>

<STATEMENT> := <PROGRAM> | <1><2><3><4><5><6> ; |
<1><2><3><4><5> ; | <1><2><3><4><6>
<1><5><6> ; | <1><5> ; | <2><3><4><5><6> ; |
<2><3><4><5> ; | <2><3><4><6> ; | <5><6> ; |
<5> ; | <6> ;

<1> := <LABEL>;

<2> := <CONDITIONAL>

<3> := <TEST SET>

<4> := THEN

<5> := <OPERATION SET>

<6> := <BRANCH>

<CONDITIONAL> := IFANY | IFALL | IFNONE | IFNALL

<TEST SET> := <TEST> | <TEST SET><TEST>

<TEST> := <IDENTIFIER> <TEST TYPE><FIELD TYPE>

<OPERATION SET> := <OPERATION>, | <OPERATION SET><OPERATION>,

<OPERATION> := <ARITHMETIC OPERATION> | <ALLDUMPS OPERATION> |
<DEFINE OPERATION> | <GET OPERATION> |
<FREE OPERATION> | <COPY OPERATION> |
<LOGICAL OPERATION> | <SHIFT OPERATION> |
<STACK OPERATION> | <DO OPERATION> |
<INPUT/OUTPUT OPERATION>
<DUPLICATE OPERATION> | <INTERCHANGE OPERATION>

```
<BRANCH> := GOTO<LABEL> | DONE | FAIL

<LABEL> := <FIELD>

<ARITHMETIC OPERATION> := <IDENTIFIER><ARITHMETIC OPERATOR>
                                                    <FIELD TYPES>
<ARITHMETIC OPERATOR> := + | - | * | /

<FIELD TYPE> := <IDENTIFIER> | INTEGER

<TEST TYPE> := = | ¬ | > | <

<ALLDUMPS OPERATION> := ALLDUMPS INTEGER INTEGER

<DEFINE OPERATION> := DEFINE<IDENTIFIER><E><F><G>

<E>&<F>&<G> := <IDENTIFIER> | INTEGER

<GET OPERATION> := GET <IDENT><IDENT> | GET <IDENT> INTEGER |
                   GET <IDENT><IDENT><IDENT> |
                   GET <IDENT> INTEGER <IDENT>

<FREE OPERATION> := FREE <IDENT><IDENT> | FREE <IDENT> INTEGER

<COPY OPERATION> := COPY <IDENT><IDENT> |
                    COPY <IDENT> INTEGER

<LOGICAL OPERATION> := <LOGICAL OPERATOR><IDENT><IDENT>
                       <LOGICAL OPERATOR><IDENT> INTEGER

<LOGICAL OPERATOR> := OR | AND | XOR | COMPLEMENT

<SHIFT OPERATION> := <SHIFT OPERATOR><IDENT><H><I>

<H> := <IDENT> | INTEGER

<I> := <IDENT> | INTEGER | NULL

<SHIFT OPERATOR> := LSHIFT | RSHIFT

<STACK OPERATION> := <STACK OPERATOR><IDENT>

<STACK OPERATOR> := STACK | POP | DFSTACK | DFPOP

<DO OPERATION> := DO <LABEL> | DO <LABEL><LABEL>

<INPUT/OUTPUT OPERATION> := <I/O OPERATOR><IDENT> INTEGER
```

<I/O OPERATOR> := INPCHAR | INPBIT | INPDEC | PRINTCHAR |
PRINTBIT | PRINTDEC

<IDENT> := <FIELD IDENTIFIED>

<FIELD IDENTIFIER> := <FIELD IDENTIFIER> . <FIELD> | <FIELD>

<FIELD> := CHARACTER STRING

APPENDIX B. BLISS INTERPETER INSTRUCTIONS

The following table is a listing of the code instructions used to perform operations on the BLISS "machine."

In the table, the symbols used as operands (OP1 and OP2) are defined as follows:

R = register. (C,D, bug or field definition registers)

P = pushdown. (System, field contents, field definition, input/output or return pushdowns)

= integer. (Can refer to any unsigned integer less than 8,192)

SYS = system pushdown.

CR = C register.

DR = D register.

OP CODE	MNEMONIC	OP1	OP2	DESCRIPTION
1	LR	R1	R2	Load Contents of R2 into R1
2	LRI	R1	#	Load # into R1
3	LRID	R1	R2	Load contents of field R2 pointed to by R1 into R1.
4	L	P1	R1	Load value in R1 into P1
5	LI	P1	#	Load # into P1
6	LID	P1	R1	Load contents of field R1 in block pointed to by P1 into P1.
7	S	R1	P1	Store top of P1 into R1 (pop P1)
8	SI	R1	P1	Store value in P1 into field R1 in block pointed to by CR
9	SRID	R1	R2	Store value in R2 into field R1 in block pointed to by CR
10	B	-	#	Unconditional branch to code address #.
11	BT	-	#	Branch to address # if value of SYS is TRUE (1).
12	BF	-	#	Branch to address # if value on SYS is FALSE (0).
13	GTO	-	-	Get block of size on top of SYS. Put pointer to it on SYS.
14	SHIFTL	R1	#	Shift R1 left # bits.
15	SHIFTR	R1	#	Shift R1 right # bits.
16	AD	-	P1	Add top two items on P1. Put results on P1.
17	E	-	P1	Test top two items on P1 for equality. If true, put '1' on P1, else '0'.
18	NE	-	P1	Same as 17, except for not equal.
19	GT	-	P1	Same as 17, except for greater than.
20	LT	-	P1	Same as 17, except less than.

OP CODE	MNEMONIC	OP1	OP2	DESCRIPTION
21	EOP	-	-	End of program marker.
22	WLD	R1	P1	Load fullword pointed to by P1 into R1.
23	SUB	-	P1	Subtract top item on P1 from next to top, put result on P1.
24	MULT	-	P1	Multiply top two items on P1. Put results on P1.
25	DV	-	P1	Divide top item on P1 into next to top item. Put results on P1.
26	OR	-	P1	Take the logical OR of the top two items on P1. Put results on P1.
27	AND	-	P1	Take the logical AND of the top two items on P1. Put results on P1.
28	XOR	-	P1	Take logical exclusive OR of the top two items on P1. Put results on P1.
29	COMP	-	P1	Complement the item on the top of P1. Put results on top of P1.
30	FRE	-	-	Free the block pointed to by top of SYS.
31	ALOG	-	-	Take antilog (base 2) of item on top of SYS. Place results on top of SYS.
32	WSD	P1	R1	Put value in R1 into fullword pointed to by P1.
33	LP	P1	P2	Load top item on P2 onto top of P1.
34	PUSHDUP	-	P1	Duplicate the item on top of P1 and push it down on top of P1.
35	SCB	R1	P1	Convert the character string on P1 to bit string and put in R1.
36	SCD	R1	P1	Convert the decimal number on P1 to bit string and put in R1.
37	BP	-	P1	Branch to the code address on top of P1.

OP CODE	MNEMONIC	OP1	OP2	DESCRIPTION
38	PC	R1	R2	Print contents of R1 in character form. Value in R2 is number of characters desired.
39	PB	R1	R2	Same as 38, except bit string vice character data.
40	PD	R1	R2	Sames as 38, except decimal vice character data.
41	SHL	R1	R2	Shift quantity in R1 left the number of bits designated in R2.
42	SHR	R1	R2	Shift quantity in R1 right the number of bits designated in R2.
43	DUMP	#1	#2	Dump system as specified by code numbers #1 and #2.

```
COMP: PROC(PARMS) OPTIONS(MAIN):
        DCL PARMS CHAR(40) VAR;
        DISPLAY(PARMS);
        DCL
              (BUGMK,IDCTR) BIN FIXED(12) EXT,
             PRINT BIT(1) EXT,
             ANS CHAR(3) VAR,
              DO_BIT BIT(1) EXT,
             CODEGEN ENTRY EXT,
             (J,K) FIXED BIN EXT,
             CODE(2000) FIXED BIN(31)  EXT,
             LEXPHAZ ENTRY EXTERNAL,
             CORRECT ENTRY EXTERNAL,
             LINECT BIN FIXED EXT,
             TAB(200) FIXED BIN(31)  EXT,
             INED(0:4095) FIXED BIN  EXT,
             INITBIT BIT(1) EXT,
             CHRSTOR CHAR(4095)  EXT,
             1 AM(4:195),
                2 INF BIN FIXED(31) INIT (
      196660,     196659,     196659,     196659,  191430656,  193003520
,  194576384,  196149248,  221315072,  222887936,  224460800,  226033664
,  227606528,  229179392,  210305024,  211877888,  213450752,  215023616
,     196661,  197722112,  199294976,  200867840,  202440704,  204013568
,  205586432,  207159296,  208732160,   90767360,   92340224,     196658
,     196658,   47775744,   49348608,   50921472,   52494336,     196662
,   55640953,   93913088,   95485952,   97058816,   98631680,  185139200
,  186712064,  188284928,  189857792,     196698,    7930017,  100206129
,  144246993,  216599825,   18415961,     131072,     131072,     196696
,   16842752,     196696,     196658,     196694,   54067200,   70320128
,     196648,   65078281,     196673,     196648,   71892992,   73465856
,   75038720,   76611584,   78184448,   79757312,   81330176,   82903040
,  844759C4,   86048768,   87621632,     196673,     196659,     196659
,     196659,     196659,     196659,     196696,     196647,  393877932
,      65577,  232849408,  234425889,          0,          0,          0
,     131112,     196657,     131112,     262148,     196665,     262175
,     196662,     131122,   46202880,     262233,     262233,     262233
,     262233,     262233,     262195,     131112,     131112,     131112
,     131112,     131112,     131112,     131112,     131112,     131112
,     131112,     131112,     262196,     131112,     262148,     196675
,     131103,     262195,     196672,     131111,     262195,     262148
,     196668,     131160,     131122,   37814272,   29950457,     262148
```

```
,      196661,      131112,      262233,      196659,      131112,   180947697
,   173083257,   165218817,   155257209,      262185,      196659,      131112
,   137955425,   130090985,   121177961,   111216345,      131076,    14745600
,      131123,      131123,      131123,      131123,      131123,      131123
,      131123,      131123,      131076,    62980096,      262148,      131123
,      131123,      131123,      131123,      131122,      131122,      131123
,      131123,      131123,      131123,      131123,      131123,      131123
,      131123,      131123,      131123,      131123,      131123,      131123
,      131123,      131123,      131123,      131123,      131123,      131123
,      131123,      131123,      131123,      262233,      262233,      262233
),                    2 PTR BIN FIXED(31) INIT (
             0,           195,           194,           193,           192,           191
,          190,           189,           188,           187,           186,           185
,          184,           183,           182,           181,           180,           179
,            0,           178,           177,           176,           175,           174
,          173,           172,           171,           170,           169,             0
,            0,           168,           167,           166,           165,    50397348
,     60883107,           161,           160,           159,           158,           157
,          156,           155,           154,             0,    12648601,   107544727
,    151584912,    89194633,    26280070,             0,             0,             0
,          131,             0,             0,           128,           127,           124
,            0,           121,           120,             0,           119,           118
,          117,           116,           115,           114,           113,           112
,          111,           110,           109,           108,           107,           106
,          105,           104,           103,             0,             0,   230752358
,      5308514,            96,            94,             0,             0,             0
,    312541184,             0,   304676959,   272171008,   270598241,   269025280
,           99,           100,           101,   267976704,   266928128,   265879552
,    264830976,   263782400,   261685248,             0,             0,             0
,            0,             0,             0,             0,             0,             0
,            0,             0,   259588096,             0,   235470848,           122
,    232849531,   231276544,           125,           126,   228655104,   198246400
,          129,           130,             0,           132,    35717253,   170983424
,    169410695,   168362120,   167313408,   166264970,           134,           140
,    178323597,   170459278,   162070671,   161021952,   158924045,           146
,    142672019,   135331988,   127467569,   118030486,   127467520,           152
,    124846080,   122224640,   119603200,   116981760,   114360320,   111738880
,    109117440,   106496000,    78184448,           162,    51445760,    47251456
,     44105728,    40960000,    37814272,             0,             0,    36241408
,     34668544,    33095680,    31522816,    29949952,    28377088,    26804224
,     25231360,    23658496,    22085632,    20512768,    18939904,    17367040
,     15794176,    14221312,    12648448,    11075584,     9502720,     7929856
,      6356992,     4784128,     3211264,     2162688,     1114112,       65536
),
                  RELSTOR BIT(5000) VAR,
                  PSTRNG BIT(5000) VAR,
                      1 FCB EXT,
                          2 COMMAND CHAR(8),
```

```
                        2 FILENAME CHAR(8),
                        2 FILETYPE CHAR(8),
                        2 CARD_NO FIXED BIN(31,0),
                        2 STATUS FIXED BIN(31,0),
                        2 CARD_BF CHAR(80),
              WKSTK(100) FIXED BIN(8),
              (KT,KW,PREC,ADDI,ADD,PSTO,ADDJ,I) BIN FIXED,
              TABLEN BIN FIXED(31) EXT,
              PDNUM BIN FIXED(8),
              PDA BIN FIXED INIT(0),
TESTB BIT(8),
              NSET BIN FIXED INIT(1),
              BUFF1 CHAR(130) VAR INIT(' '):
         CALL SETUP(RELSTOP):
         NSET = NSET + 1:
         CALL SETUP(PSTRNG):
         INITBIT = '1'B:
         CODE = 0:
         FILENAME = PARMS:
         J = 0:
         K = 1:
         BUGMK = 18:
         LINECT = 0:
         IDCTR = 41:
         DO_BIT = '0'B:
         CARD_NO = 1:

/*       IF "YES" THEN PRINTS HEADER AND PASSES LISTING
         DESIGNATOR TO "LEXPHAZ".                          */

         DISPLAY('  PROGRAM LISTING DESIRED?')REPLY(ANS):
         IF ANS = 'YES' THEN DO:
          DISPLAY('            LINE        STATEMENT        TEXT'):
             PRINT = '1'B:
         END:
         ELSE PRINT = '0'B:

/*       STARTS COMPILATION LOOP - IF THE "BEGIN:"/"STOP:" COUNTER
         EMPTIES, TERMINATION OCCURS.  IF PROGRAM IS INCORRECTLY
         COMMENCED, BRANCHES TO SYNTAX ERROR SECTION.      */

COM:     IF PDA = 0 & CARD_NO > 2 THEN GOTO COM7:
         LINECT = LINECT + 1:
         CALL LEXPHAZ:
         IF CARD_NO = 2 & PDA = 0 THEN GOTO COM5:

/*       PARSING ALGORITHM FORM THIS POINT TO "COM5". SEE SYNTAX
         ANALYSIS SECTION OF THESIS.                       */
```

```
          KT = 2;
          KW = 1;
          WKSTK(1) = SUBSTR(UNSPEC(TAB(1)),1,8);
COM1:     IF KT > TABLEN THEN
               IF KW = 1 THEN GOTO COM6;
               ELSE GOTO COM2;
          CALL CKPREC;
          IF PREC = 1 THEN GOTO COM5;
          IF PREC ¬= 2 THEN DO;
               KW = KW + 1;
               WKSTK(KW) = SUBSTR(UNSPEC(TAB(KT)),1,8);
               KT = KT + 1;
               GOTO COM1;
               END;
COM2:     ADDI = 0;
COM3:     CALL GET_ADD;
          IF ADD = 0 | (ADD>1 & ADD<5) THEN GOTO COM5;
          PDNUM = SUBSTR(PSTRNG,ADD,8);
          ADDJ = PDNUM;
          PSTQ = ADD + ((ADDJ + 1) * 8);
COM35:    IF WKSTK(KW - PDNUM + ADDJ) = SUBSTR(PSTRNG,PSTQ,8) THEN DO;
          IF ADDJ = 1 THEN GOTO COM4;
          IF (KW - PDNUM + ADDJ) < 1 THEN GOTO COM3;
          ADDJ = ADDJ - 1;
          PSTQ = PSTQ - 8;
          GOTO COM35;
          END;
          ADDI = ADDI + 1;
          GOTO COM3;
COM4:     KW = KW - PDNUM + 1;
          WKSTK(KW) = SUBSTR(PSTRNG,(PSTQ-8),8);
          IF WKSTK(KW) = 57 THEN PDA = PDA + 1;
          IF WKSTK(KW) = 59 THEN PDA = PDA - 1;
          GOTO COM1;

/*        ON SYNTAX ERROR, BRANCH TO CORRECTION ROUTINE.  IF ZERO
          RETURNED THEN TERMINATE.                                  */

COM5:      CALL CORRECT;
          IF CARD_NO = 0 THEN GOTO COM8;
          LINECT = LINECT - 1;
          GOTO COM;

/*        SYNTAX CORRECT.  CALL FOR CODE GENERATION.          */

COM6:     CALL CODEGEN;
          GOTO COM;
```

```
SETUP: PROC(CSTR):
   DCL
  CSTR BIT(5000) VAR,BT BIT(8), I BIN FIXED,BA BIN FIXED(8),
                   A CHAR(1240) VAR INIT(                          '
90090090008990008990008990008990008990008990008990008990008990008990008990008990
89900089900089900089900089900089900089900089900089900089900089900089900089900089
62648689900062648689900062648689900062648689900087000506070809101112131 4
151617181920212223242526272829303132515363676869707172737475767778808182
8384999000506070809101112131415161718192021222324252627282930313239 5051
535464676869707172737475767778808182838489900052658990005265899000526589
900052658990005265899000526589900052658990000506070809101112 13
141516171819202122232425262728293031323536373851535460616367686970717273
747576777880818283848788999000526679004243444546474889900052009000500061
870005060708091011121314151617181920212223242526272829303536373851636768
697071727374757677788081828384899000050607080910111213141516171819202122
232425262728293031323536373851535461636768697071727374757677788081828384
878990006264899000899000325053540005060708091011121314151617181920212223
242526272829305168697071727374757677788081828384899000528990005289000090
009000900009000000325300598500050607080910111213141516171819202122232425
262728293031323334353637385153545860616367686970717273747576777880818283
848789900041424344454647485152667989900041424344454647484951526679899000
99'),      B CHAR(1200) VAR INIT(                          '
035688578803568885880356885988035858605002586050025733500259345001855806
605361863987540560536186395405606186398754046061863954036053875402608754
016054056053618639870460618639870160870161350161360161370161380186620362
626440036363674002626440026367400364516665036851796501876301676801676901
677001677101677201677301677401677501677601677701677801545301543101543201
794101794201794301794405700551515151057005515152510570055152525105700551
525151047106515151047106515251047582515151047582515251037106515103720751
510373805151037481515103758251510276835101655105700551515152057005515152
520570055152525205700551525152047582515152047582515252035904525203710651
520372075152037380515203748151520375825152037884515201655201664501664601
664701664801800801900901801001801101812301812401812501812601822701822801
822901823001831801831901832001832103772253530277225301841201841301841401
84150184160184170260538701518903398949900189909 9'):
        I = 1:  CSTR = '';
        IF NSET = 1 THEN DO;
      SEP1: BA = SUBSTR(A,I,2):
            IF BA = 99 THEN GOTO SEP3:
            BT = BA:
            CSTR = CSTR||BT:
            I = I + 2:
            GOTO SEP1:
            END:
        ELSE DO:
      SEP2: BA = SUBSTR(B,I,2):
            IF BA = 99 THEN GOTO SEP3:
            BT = BA:
```

```
                    CSTR = CSTR||BT:
                    I = I + 2:
                    GOTO SEP2:
                    END:
SEP3:     END SETUP:
          CKPREC: PROC:
               DCL
                    (RTAB,RSTG) BIN FIXED(8),
                    (REL,RPTR,QPTR) BIN FIXED:

/*        CHECKS FOR PRECEDENCE RELATIONSHIP BETWEEN SYMBOLS.
          RETURNS: 1 - NO RELATIONSHIP, 2 - >, 3 - =, 4 - <.  */

               RTAB = SUBSTR(UNSPEC(TAB(KT)),1,8);
               RPTR = WKSTK(KW):
               REL = SUBSTR(UNSPEC(INF(RPTR)),1,16):
               DO WHILE(REL<2 | REL>4):
CK1:                RPTR = SUBSTR(UNSPEC(PTR(RPTR)),17,16):
                    REL = SUBSTR(UNSPEC(INF(RPTR)),1,16):
               END:
               IF RTAB <= SUBSTR(UNSPEC(INF(RPTR)),17,16) THEN
                    IF RTAB = SUBSTR(UNSPEC(INF(RPTR)),17,16) THEN DO:
                         PREC = REL:
                         RETURN:
                    END:
                    ELSE GOTO CK2:
               ELSE DO:
                    QPTR = SUBSTR(UNSPEC(PTR(RPTR)),1,16):
                    IF QPTR = 0 THEN GOTO CK2:
                    RSTG = SUBSTR(RELSTOR,QPTR,8):
                    DO WHILE(RSTG ¬= 0):
                         IF RSTG >= RTAB THEN DO:
                              IF RSTG = RTAB THEN DO:
                                   PREC = REL:
                                   RETURN:
                              END:
                              ELSE GOTO CK2:
                         END:
                         QPTR = QPTR + 8:
                         RSTG = SUBSTR(RELSTOR,QPTR,8):
                    END:
CK2:                IF SUBSTR(UNSPEC(PTR(RPTR)),17,16) = 0 THEN DO:
                         PREC = 1:
                         RETURN:
                    END:
                    GOTO CK1:
               END:
          END CKPREC:
```

```
          GET_ADD: PROC:
               DCL (GADDI,GBLK,GI) BIN FIXED, GPTR BIN FIXED STATIC:
               IF ADDI = 0 THEN DO:
                    GPTR = WKSTK(KW):
                    GADDI = 0:
                    GOTO GET1:
                    END:
               GADDI = MOD(ADDI,3):
               IF GADDI = 0 THEN GPTR = SUBSTR(UNSPEC(PTR(GPTR)),17,16):
GET1:          IF GADDI = 0 THEN
                    ADD = SUBSTR(UNSPEC(INF(GPTR)),1,16):
               IF GADDI = 1 THEN
                    ADD = SUBSTR(UNSPEC(INF(GPTR)),17,16):
               IF GADDI = 2 THEN
                    ADD = SUBSTR(UNSPEC(PTR(GPTR)),1,16):
          END GET_ADD:
COM7:     DISPLAY('LOGICAL END OF PROGRAM - LINE NUMBER '||CARD_NO):

/*        OUTPUTS TO A "TEXT" FILE THE PRODUCT OF THE CODE GENERATOR
          AFTER ALL COMPILATION IS COMPLETED.                          */

          FILETYPE = 'TEXT':
          COMMAND = 'ERASE':
          CALL IHEFILE(ECB):
          COMMAND = 'WRBUF':
          CARD_NO = 1:
               DO I = 1 TO 2000 BY 6:
                    CARD_BF = ' ':
                    PUT STRING(CARD_BF)EDIT((CODE(J+I) DO J = 0 TO 5))
                         (F(12)):
                    CALL IHEFILE(ECB):
                    CARD_NO = CARD_NO + 1:
                    IF CODE(I+5) = 0 THEN GOTO COM8:
               END:
COM8:  END COMP:



  LEXPHAZ: PROC OPTIONS :
     DCL 1 ECB EXTERNAL,
           2 COMMAND CHAR(8),
           2 FILENAME CHAR(8),
           2 FILETYPE CHAR(8),
           2 CARD_NO FIXED BIN(31),
           2 STATUS   FIXED BIN(31),
           2 CARD_BF CHAR(80),
         INITBIT BIT(1) EXT  ,
```

```
    PRINT BIT(1) EXT,
    DO_BIT BIT(1)EXT,
    BP_BIT(1),
    (BUGMK,IDCTR) FIXED BIN(12) EXTERNAL,
    DEF BIT(1),
    Y FIXED BIN(31),
    X FIXED BIN(31),
     ACCUM CHAR(80),
     BPTR FIXED BIN(31)EXT,
     (RESULT, ALPHABASE,DIGITBASE,COUNT
       ) FIXED BINARY(31) STATIC,
     CHARSET CHAR(50) VARYING INIT
     (' <+*;¬-/,>:=.|ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789'),
     BUFFER CHAR(130) VARYING,
     LINECT FIXED BIN(31) EXT,
     INFO(0:4095) FIXED BIN(31)EXT,
     SCAN ENTRY,
     READER ENTRY,
     LOOKUP RETURNS(FIXED BIN(12)),
     ENTER RETURNS(FIXED BIN (12)),
     (LASTELBAT,SCRAMCODE)FIXED BIN(12),
     ELBAT BIT(32), ELCLASS BIT (8),
     TAB(200) FIXED BIN (31) EXT ,
     TABLEN FIXED BIN(31) EXT,
     CHRSTOR CHAR(4095) EXT:
READER:PROC:
          BPTR=1;
          FILETYPE='BLI';
          COMMAND='RDBUF';
          CARD_NO=CARD_NO+1;
          CALL IHEFILE(FCB);
          IF PRINT THEN
          DISPLAY(CARD_NO||LINECT||'           '||CARD_BF):
          IF STATUS = 0 THEN RETURN:
          ELSE RESULT=0:
END READER;


    /* SCANS INPUT FILE AND RETURNS MEANINGFUL ENTITIES */
SCAN: PROC:
     DCL TYPE(0:2) LABEL (DEBLANK,IDENT,DIGIT)
           INIT (DEBLANK,IDENT,DIGIT),
         K FIXED BIN(31)STATIC,
       T CHAR(1) STATIC:
     BRET: RESULT,COUNT=0:
     ACCUM=' ':
       DO WHILE ('1'B);
```

```
          BPTR=BPTR+1:
          IF BPTR > 72 THEN CALL READER:
          IF STATUS¬= 0 THEN RETURN:
          T=SUBSTR(CARD_BF,BPTR,1):
          K= INDEX(CHARSET,T):
          GO TO TYPE(RESULT):
       DEBLANK:
         IF K>1 THEN  /* NOT BLANK */
           IF K>ALPHABASE THEN  /* NOT SPEC CHAR */
            IF K>DIGITBASE THEN /* NOT ALPHA CHAR */
              RESULT=2:
            ELSE RESULT = 1:
           ELSE DO: RESULT = 3; GO TO SPEC_CHAR:
                END:
         ELSE GO TO XIT:
         GO TO STORET:
       SPEC_CHAR:
         SUBSTR(ACCUM,1,1)=T:
         COUNT=1:
         GO TO RETURNL:
       IDENT: IF K > ALPHABASE THEN GO TO STORET:
       IF K=11 THEN DO: RESULT = 4; GO TO RETURNL: END:
         GO TO BACKUP:
       DIGIT: IF K> DIGITBASE THEN GO TO STORET:
         GO TO BACKUP:
       STORET: IF COUNT>79 THEN GO TO RETURNL:
                  COUNT=COUNT + 1:
                  SUBSTR(ACCUM,COUNT,1) = T:
       XIT:
         END:
       BACKUP: BPTR = BPTR - 1:
       RETURNL: IF ACCUM = 'GOTO' THEN DO: BR='1'B; GO TO BRET: END:
          IF ACCUM='DEFINE' THEN DEF='1'B:
         ELSE IF ACCUM='DO' THEN DO_BIT='1'B:
         ELSE IF ACCUM=',' THEN DO_BIT='0'B:
      RETURN:
END SCAN:


  /* CONTROLS ENTRY INTO AND CHECKING OF ITEMS IN INFO TABLE */
TABLE: PROC:
      DCL J FIXED BIN(12) STATIC,
          X FIXED BIN(7) STATIC INIT(4),
        C BIT(8) STATIC,
      SW(0:4) LABEL (EOF,ID,DIG, SPEC,LABL)
      INITIAL(EOF,ID,DIG,SPEC,LABL):
      J=-1:
```

```
   ACCUM='    ';
     DO WHILE (ACCUM¬= ';');
        CALL SCAN;
        J=LOOKUP;
        IF J<0 THEN
            DO: J=ENTER;      IF J<0 THEN
                NULLSTR:
                DO: ELBAT='0'B: ELCLASS='0'B: RETURN;
                END;
            IF BP='1'B | DO_BIT='1'B THEN RESULT = 4;
            GO TO SW(RESULT);
            EOF: GO TO NULLSTR;
    ID: C='01011010'B; GO TO XIT;
    DIG: C='00110100'B; GO TO XIT;
    SPEC: GO TO XIT;
    LABL:   DO: C='00110101'B:BR='0'B;END;
          XIT:
        IF RESULT=4 THEN ELBAT=C||'1'B||(23)'0'B; ELSE
            ELBAT=C||(24)'0'B;
            IF RESULT = 1  & DEF = '0'B THEN
               DO: SUBSTR(ELBAT,9,12) =BIT(BUGMK,12);
                   BUGMK=BUGMK+1;
                   IF BUGMK>40 THEN BUGMK=18;
               END;
               ELSE IF DEF='1'B & ACCUM¬='DEFINE' THEN
                 DO: SUBSTR(ELBAT,9,12)=BIT(IDCTR,12);
                  IDCTR=IDCTR+1:DEF='0'B;
                 END;
            UNSPEC(INFO(LASTELBAT))=ELBAT;
            END;
            ELSE
        ELBAT=UNSPEC(INFO(SUBSTR(UNSPEC(INFO(J)),21,12)));
        SUBSTR(ELBAT,21,12)=BIT(J,12);
        IF ACCUM='IF' THEN
         UNSPEC(TAB(TABLEN))='00100100'||REPEAT('0'B,15)||
         '10101001';
        ELSE
        UNSPEC(TAB(TABLEN))=UNSPEC(ELBAT);
        TABLEN=TABLEN+1;
        ELCLASS=SUBSTR(ELBAT,1,8);
        IF ACCUM ¬= 'DEFINE' THEN DEF='0'B;
        BR='0'B;
        END;
END TABLE;


  /* COMPUTES SCRAMWORD FOR ITEMS AND DETERMINES IF TABLED YET */
```

```
LOOKUP: PROC FIXED BIN(12);
    DCL(K BIT(32), L FIXED BIN(12))STATIC:
    LASTELBAT=0:
    L,SCRAMCODE=MOD(UNSPEC(SUBSTR(ACCUM,1,3))+
                MOD(COUNT,128)*16777216,127):
    K=UNSPEC(INFO(SCRAMCODE)):
    /* K CONTAINS THE ORIGINAL SCRAMWORD */
        DO WHILE(K):
        IF COUNT=SUBSTR(K,1,8) THEN
        IF SUBSTR(ACCUM,1,COUNT)=
        SUBSTR(CHRSTOR,SUBSTR(K,9,12),COUNT) THEN
            RETURN(L):
        LASTELBAT=SUBSTR(K,21,12):
        L=SUBSTR(UNSPEC(INFO(LASTELBAT)),21,12):
        /* L IS LOCATION OF NEXT SCRAMWORD IN CHAIN */
        IF L=0 THEN K=0: ELSE
        K=UNSPEC(INFO(L)): /* K IS NEXT SCRAMWORD */
        END:
    RETURN(-1):
    END LOOKUP:



  /* ENTERS ITEMS IN INFO TABLE */
ENTER: PROC FIXED BIN(12):
    DCL((K,M,N) FIXED BIN(12), C FIXED BIN(8),
    T BIT(32)) STATIC:
    M=INFO(127): /* M IS LOC OF NEW ELBAT WORD */
    IF M > 4095 THEN GO TO ERR: INFO(127)=M+1:
    IF LASTELBAT=0 THEN K=SCRAMCODE: ELSE
        DO: /* GET SPACE FOR NEW SCRAMWORD */
        K=INFO(127):
        IF K > 4095 THEN GO TO ERR: INFO(127)=K+1:
        END:
    /* BUILD CHRSTOR AND SCRAMWORD */
    N=INFO(128):
    IF N+COUNT > 4095 THEN GO TO ERR:
    INFO(128)=N+COUNT:
    SUBSTR(CHRSTOR,N,COUNT)=SUBSTR(ACCUM,1,COUNT):
    C=COUNT:
    UNSPEC(INFO(K))=BIT(C,8)||BIT(N,12)||BIT(M,12):
    IF LASTELBAT¬=0 THEN /* FIX CHAIN OF SW/EW PAIRS */
        DO: T=UNSPEC(INFO(LASTELBAT)): SUBSTR(T,21,12)=BIT(K,12):
        UNSPEC(INFO(LASTELBAT))=T:
        END:
    LASTELBAT=M:
    RETURN(K): /* K IS THE LOCATION OF SCRAM WORD */
```

```
      ERR: RETURN(-1);
      END ENTER;


      /* TABLE INITIALIZATION ROUTINE. CALLED ON FIRST LEXPHA7 CALL */
      INIT: PROC;
      INFO=0;
      DCL CRSTR CHAR(223) INIT
 ('ALLDUMPSDEFINEGETFREECOPYDUPLICATEINTERCHANGEPOINTERINPCHARINPBITINPC
HARPRINTCHARPRINTBITPRINTDECSTACKPOPDESTACKDEPOPDOORANDXORCOMPLEMENTLSHIF
TARITHLSHIFTHRSHIFTDONEFAILBEGINSTOPIFANYIFALLIFNONEIFNALLTHEN,+-*/=¬>
<.(BUFFER;'),
      INF(0:191) FIXED BIN(31) INIT((2)0,151101575,17625262,
  17649849,(2)0,17662141,(3)0,84586661,67936429,67702060,(5)0,
  101224606,184692872,34033816,(4)0,100700290,(7)0,134549648,
  84223527,0,84344083,(4)0,151289999,50302210,(13)0,67182724,0,
  67776675,(3)0,101441706,117870742,0,84283539,17600815,50830708,
  2)0,134221953,(3)0,17629360,(3)0,67829926,101552318,(14)0,
  100909196,(3)0,34042009,(2)0,168292509,117657729,(2)0,
  118050977,(11)0,118079650,17621166,(2)0,17653946,(3)0,
  182.224,67109048,83886080,100663456,117440646,134217728,
  67109100,150995093,167772298,184549531,117623065,201326502,
  218103950,234881024,100933773,251658240,268435602,285212672,134582417,
  301989888,318767104,50749588,335544320,352321536,369098752,385875968,
  402653184,50827418,419430582,436207616,452984832,469762048,101249183,
  486539264,503316668,520093874,536870912,553648128,570425524,
  587202729,603979776,84644008,620757164,637534208,101466283,
  654311424,671088640,687865956,704642072,721420288,17633457,
  738197504,17637555,754974720,17641653,771751936,17645751,
  788529152,805306368,822083584,17658043,1476395008,822746880,
  838860800);
      BPTR=72;
      CAPD_NO=0;
      BR='0'B;
      ALPHABASE = INDEX(CHARSET,'A')-1;
      DIGITBASE = INDEX(CHARSET,'0')-1;
      DO I=0 TO 191; INFO(I) = INF(I); END;
      CHRSTOR = CRSTR;
      END INIT;



      /* LEXPHA7 CONTROL SECTION */
      DO WHILE(INITBIT); CALL INIT; INITBIT='0'B; END;
      TAB=0;
      DEF='0'B;
```

```
      TABLEN=1;
      UNSPEC(TAB(TABLEN))= '01011000000000000000000000000000'B;
      TABLEN=2;
       CALL TABLE;
       Y=TABLEN-2;
       IF SUBSTR(UNSPEC(TAB(Y)),1,8)='00110101'B & DO_BIT='0'B THEN
          UNSPEC(TAB(Y))='00110110'B||SUBSTR(UNSPEC(TAB(Y)),9,24);
          IF SUBSTR(UNSPEC(TAB(Y)),1,8)='01011010'B |
             SUBSTR(UNSPEC(TAB(Y)),1,8)='00110100'B
           | SUBSTR(UNSPEC(TAB(Y)),1,8)='00110101'B THEN
        DO; X=TABLEN-1; UNSPEC(TAB(TABLEN))=UNSPEC(TAB(X));
          UNSPEC(TAB(X))='00101'B||REPEAT('0'B,19)||'1111000'B;
            TABLEN=TABLEN+1;
          END;
      UNSPEC(TAB(TABLEN))= '01011000000000000000000000000000'B;
      DO_BIT='0'B;
   END LEXPHAZ;


CODEGEN: PROC;
   DCL INFO(0:4095)FIXED BIN(31)EXT,
    TAB(200)FIXED BIN(31)EXT,
    TABLEN FIXED BIN(31)EXT,
    CHRSTOR CHAR(4095) EXT,
    GENBLOK(4:48) LABEL(DUMP,DF,GT,FR,COPY,DUP,INT,POINT,INPC,INPB,INPD,
          PRTC,PRTB,PRTD,STACK,POP,DFST,DFPOP,DOO,OR,AND,XOR,COMP,LSH,
          RSH,HLSH,HRSH,DON,FAIL,REG,STOP,IFANY,IFALL,IFNONE,IFNALL,
          THN,COMMA,ADD,SUBT,MULT,DIV,EQ,NOTEQ,GRT,LESS) INITIAL
       (DUMP,DF,GT,FR,COPY,DUP,INT,POINT,INPC,INPB,INPD,PRTC,PRTB,PRTD,
          STACK,POP,DFST,DFPOP,DOO,OR,AND,XOR,COMP,LSH,RSH,HLSH,HRSH,
          DON,FAIL,REG,STOP,IFANY,IFALL,IFNONE,IFNALL,THN,COMMA,ADD,
          SUBT,MULT,DIV,EQ,NOTEQ,GRT,LESS),
        (TS,VAL,VAL_1) FIXED BIN(31) EXT,
        ( IDSTK(15),ID,X,Y,Z, SCRAMWORD,LOC,COUNT,CC INIT
        (0)) FIXED BIN(31) STATIC,
       CH_VAL CHARACTER(15) VARYING,
       BOP BIT(6) EXT, FILL BIT(13) EXT,
       IO_OP FIXED BIN(13)EXT,
        BRANCH ENTRY EXTERNAL,
        INPROC ENTRY EXTERNAL,
       (B2_MK,B1_TOP,B1_MK,FP,HOLD,MK_1,MK_2, C2)FIXED BIN(13)STATIC,
       C1 FIXED BIN(13) EXT,
       (J,K,FD_PTR)FIXED BIN(13) EXT,
       (LAST,FB,DEF,BUG_1)BIT(1) INIT ('0'B ) STATIC,
       BUG BIT(1) EXT,
       SR_BP(8)FIXED BIN(31) STATIC,
       CODE(2000)FIXED BIN(31) EXT,
```

```
      COND(35:38) LABEL(IFANY,IFALL,IFNONE,IFNALL)
         INITIAL(IFANY,IFALL,IFNONE,IFNALL),
      (CR INIT ('0000000001001'B),DR INIT ('0000000001010'B),
       SYS INIT ('0000000001011'B),FD INIT ('0000000001100'B),
       IP INIT ('0C0C0CCC10000'B),
       EC INIT ('0000000001101'B),RET INIT ('0000000001110'B))
        BIT(13) STATIC,
      (LR INIT ('000001'B),LRI INIT ('000010'B),LRID INIT ('000011'B),
       L INIT ('C00100'B),LI INIT ('000101'B),LID INIT ('000110'B),
       S INIT ('000111'B),SI INIT ('001000'B),SRID INIT ('001001'B),
       B INIT ('001010'B),BT INIT ('001011'B),BF INIT ('001100'B),
       GTO INIT ('001101'B),SHIFTL INIT ('001110'B),
       E INIT ('010001'B), PUSHDUP INIT ('100010'B),
       SHIFTR INIT ('001111'B),AD INIT ('010000'B),
       SUB INIT ('010111'B),MUL INIT ('011000'B),DV INIT ('011001'B),
       LOP INIT ('000000'B),EOP INIT('010101'B), ERR INIT ('011110'B),
       CB1 INIT ('000000'B), LP INIT ('100001'B),
       WLD INIT('010110'B), ALOG INIT('011111'B),WSD INIT('100000'B),
        BP INIT('100101'B),SCB INIT('100011'B),SCD INIT('100100'B),
        PC INIT ('100110'B),PB INIT('100111'B),PD INIT('101000'B),
        SHOP INIT('000000'B), SH_OP INIT('000000'B),
        DP INIT ('101011'B), NE INIT ('010010'B),
        SHL INIT ('101001'B), SHR INIT ('101010'B),
       TEST INIT ('0C0000'B),AOP INIT ('000000'B)) BIT(6) STATIC;


   ST=K; ID=1;TS=2;
   BT_MK,BF_MK=0; CC=0;
   B1_MK,B2_MK,B1_TOP=0;


ENT: X=SUBSTR(UNSPEC(TAB(TS)),1,8);    /* GET CLASS */
   IF X<40 THEN GO TO GENBLOK(X);
   IF X=50 THEN GO TO XIT;
   IF X=0C THEN  /* IDENTIFIER */
     DO; CALL FLD_STK; GO TO ENT; END;
   IF X=53 THEN  /* LABEL */
     DO;
       Y=SUBSTR(UNSPEC(TAB(TS)),21,12);
       Z=SUBSTR(UNSPEC(INFO(Y)),21,12);
       IF SUBSTR(UNSPEC(INFO(Z)),9,12)= '100000000000'B THEN
         DO;
          UNSPEC(INFO(Z))=SUBSTR(UNSPEC(INFO(Z)),1,8)||
          SUBSTR(UNSPEC(K),21,12)||SUBSTR(UNSPEC(INFO(Z)),21,12);
          TS=TS+1; GO TO ENT;
```

```
        END:
 ELSE
   DO:
     Y=SUBSTR(UNSPEC(INFO(Z)),10,11):
    UNSPEC(INFO(7))=SUBSTR(UNSPEC(INFO(7)),1,8)||
         SUBSTR(UNSPEC(K),21,12)||SUBSTR(UNSPEC(INFO(7)),21,12):
     DO WHILE (Y¬=0):
       Z=Y:
       Y=SUBSTR(UNSPEC(CODE(Y)),7,26):
      UNSPEC(CODE(Z))=SUBSTR(UNSPEC(CODE(Z)),1,19)||BIT(K,13):
     END:
     TS=TS+1:
     GO TO ENT:
   END:
   END:
 IF X=54 THEN    /* BRANCH */
  DO:
   BOP=8: FILL=(13)'0'B:
   CALL BRANCH:
   GO TO ENT:
  END:


/* ALLDUMPS CODE GENERATOR */
DUMP:TS=TS+1:CALL INT_GET:VAL_1=VAL:CALL INT_GET:
     UNSPEC(CODE(K))=OP||SUBSTR(UNSPEC(VAL_1),20,13)
      ||SUBSTR(UNSPEC(VAL),20,13): K=K+1:
     GO TO ENT:


 /* DEFINE OPERATION CODE GENERATOR */
 DF:TS=TS+1: FP=SUBSTR(UNSPEC(TAB(TS)),9,12):TS=TS+1:
     X=SUBSTR(UNSPEC(TAB(TS)),1,8):
     IF X=90 THEN DO: CALL FLD_STK: CALL ID_CONT:GO TO DEFSH: END:
      ELSE     CALL INT_GET:
 DEFSH: UNSPEC(CODE(K))=LRI||DR||'000000010000'B:K=K+1:
    UNSPEC(CODE(K))=SHIFTL||CR||DR:K=K+1:
          UNSPEC(CODE(K))=L||SYS||CR:K=K+1:
     DO I =1 TO 2:
      X = SUBSTR(UNSPEC(TAB(TS)),1,8):
      IF X=90 THEN DO: CALL FLD_STK: CALL ID_CONT:GO TO DEF2: END:
      ELSE CALL INT_GET:
     DEF2: IF I=2 THEN GO TO DEF3:
       UNSPEC(CODE(K))=LRI||DR||'000000001000'B;K=K+1:
        UNSPEC(CODE(K))=SHIFTL||CR||DR:K=K+1:
     DEF3:UNSPEC(CODE(K))=L||SYS||CR: K=K+1:
```

```
      END:
      UNSPEC(CODE(K))=AD||(13)'0'B||SYS;K=K+1:
      UNSPEC(CODE(K))=AD||(13)'0'B||SYS;K=K+1:
      UNSPEC(CODE(K))=S||BIT(FP,13)||SYS; K=K+1;
      GO TO ENT:


  /* GET OPERATION CODE GENERATOR */
GT:  TS=TS+1; CALL FLD_STK; CALL ID_FCON:
      UNSPEC(CODE(K))=LR||DR||CR; K=K+1:
     X= SUBSTR(UNSPEC(TAB(TS)),1,8):
     IF X=90 THEN
        DO; CALL FLD_STK; CALL ID_CONT; GO TO GET_ENT; END;
     ELSE CALL INT_GET;
GET_ENT: UNSPEC(CODE(K))=L||SYS||CR; K=K+1;
         UNSPEC(CODE(K))=GTO||(26)'0'B; K=K+1;
     IF BUG THEN DO; BUG='0'B; UNSPEC(CODE(K))=S||BIT(C1,13)||
                    SYS;K=K+1; GO TO GET_B;
                 END;
     UNSPEC(CODE(K))=S||CR||FD; K=K+1;
     UNSPEC(CODE(K))=SI||BIT(C1,13)||SYS; K=K+1:
 GET_B: X=SUBSTR(UNSPEC(TAB(TS)),1,8):
     IF X=90 THEN
        DO; CALL FLD_STK; CALL ID_FCON:
         UNSPEC(CODE(K))=L||SYS||DR; K=K+1:
         IF BUG THEN DO; BUG='0'B;UNSPEC(CODE(K))=S||BIT(C1,13)
           ||SYS; K=K+1; GO TO GET_OUT;
                    END;
          UNSPEC(CODE(K))=S||CR||FD; K=K+1:
          UNSPEC(CODE(K))=SI||BIT(C1,13)||SYS; K=K+1:
       GET_OUT: GO TO ENT:
        END;
     ELSE GO TO ENT:


  /* FREE OPERATION CODE GENERATOR */
 FR:  TS=TS+1;CALL FLD_STK;CALL ID_FCON:
     UNSPEC(CODE(K))=LT|SYS||CR; K=K+1:
     X=SUBSTR(UNSPEC(TAB(TS)),1,8):
     IF X=90 THEN
        DO; CALL FLD_STK;CALL ID_CONT;GO TO FR_1;END;
     ELSE CALL INT_GET;
     FR_1: UNSPEC(CODE(K))=L||SYS||CR; K=K+1:
     IF BUG THEN DO; BUG='0'B; UNSPEC(CODE(K))=S||BIT(C1,13)||
        SYS;K=K+1; GO TO FR_OUT;
                 END:
```

```
    ELSE UNSPEC(CODE(K))=S||CR||FD:K=K+1:
         UNSPEC(CODE(K))=SI||BIT(C1,13)||SYS:K=K+1:
   ER_OUT: UNSPEC(CODE(K))=ERE||(26)'0'B:K=K+1:GO TO ENT:


  /* COPY OPERATION CODE GENERATOR */
 COPY:  TS=TS+1:CALL FLD_STK:CALL ID_FCON:
        K=K-1:
          X=SUBSTR(UNSPEC(TAB(TS)),1,8):
          IF X=00 THEN
          DO: CALL FLD_STK: CALL ID_CONT: GO TO COPY1; END;
          ELSE CALL INT_GET;
    COPY1: UNSPEC(CODE(K))=L||SYS||CR: K=K+1:
          IF BUG THEN DO: BUG='0'B:
          UNSPEC(CODE(K))=S||BIT(C1,13)||SYS:K=K+1:
          GO TO ENT: END:
          ELSE UNSPEC(CODE(K))=S||CR||FD: K=K+1:
          UNSPEC(CODE(K))=SI||BIT(C1,13)||SYS:K=K+1:
       GO TO ENT:


/* DUPLICATE OPERATION CODE GENERATOR. DONE IN PROCEDURE INPROC */
DUP:  TS=TS+1:CALL FLD_STK:CALL ID_FCON:CALL FLD_STK:CALL ID_CONT:
     TO_OP=4:CALL INPROC: GO TO ENT:


  /* INTERCHANGE OPERATION CODE GENERATOR */
 INT:  TS=TS+1:CALL FLD_STK:CALL ID_FCON:FD_PTR=C1:
          UNSPEC(CODE(K))=L||SYS||CR: K=K+1:
       IF BUG THEN DO:BUG='0'B:BUG_1='1'B:END:
       ELSE DO: UNSPEC(CODE(K))=S||DR||FD:K=K+1: END:
          CALL FLD_STK: CALL ID_FCON:
          UNSPEC(CODE(K))=L||SYS||CR: K=K+1:
        IF BUG THEN DO: BUG='0'B:
     UNSPEC (CODE(K))=S||BIT(FD_PTR,13)||SYS:K=K+1:GO TO INT_1:END:
          UNSPEC(CODE(K))=S||CR||FD:K=K+1:
          UNSPEC(CODE(K))=SI||BIT(FD_PTR,13)||SYS:K=K+1:
          INT_1: UNSPEC(CODE(K))=LR||CR||DR:K=K+1:
           IF BUG_1 THEN DO:
           BUG_1='0'B:
           UNSPEC(CODE(K))=S||BIT(C1,13)||SYS:K=K+1:
           GO TO ENT:
           END;
          UNSPEC(CODE(K))=SI||BIT(C1,13)||SYS:K=K+1:
          GO TO ENT:
```

```
 PCINT:
   /* NOT IMPLEMENTED SINCE COPY OPERATION IS SAME */


 /* INPUT OPERATION CODE GENERATOR.  DONE IN PROCEDURE INPROC. */
  INPC: IO_OP=1;
     GO TO IO_1;
  INPB:
     IO_OP=2; GO TO IO_1;
  INPD:
     IO_OP=3;
   IO_1: TS=TS+1; CALL FLD_STK;CALL ID_FCON;
   K=K-1;
   CALL INT_GET;
   CALL INPROC;
   IF VAL>4 & IO_OP=1 THEN GO TO ENT;
    IF BUG THEN DO; BUG='0'B;
       UNSPEC(CODE(K))=SI||BIT(C1,13)||SYS:K=K+1; GO TO ENT;END;
    UNSPEC(CODE(K))=SI||CR||FD:K=K+1;
    UNSPEC(CODE(K))=SI||BIT(C1,13)||SYS:K=K+1; GO TO ENT;


/* OUTPUT OPERATION CODE GENERATOR */
   PRTC: BOP=PC;GO TO PRINT_1;
   PRTB: BOP=PB; GO TO PRINT_1;
   PRTD: BOP=PD;
   PRINT_1: TS=TS+1;CALL FLD_STK;CALL ID_FCON;
            K=K-1;
         CALL INT_GET;
        IF BUG THEN DO;BUG='0'B;GO TO PRINT_2;END;
       UNSPEC(CODE(K))=SI||CR||FD; K=K+1;
PRINT_2:UNSPEC(CODE(K))=BOP||BIT(C1,13)||SUBSTR(UNSPEC(VAL),20,13);
 K=K+1;
   GO TO ENT;


  /* STACK OPERATION CODE GENERATOR */
  STACK: TS=TS+1;CALL FLD_STK;CALL ID_CONT;
          UNSPEC(CODE(K))=LI||FC||CR;K=K+1; /*PUT ON FC PD */
          GO TO ENT;


 /* POP OPERATION CODE GENERATOR */
```

```
POP:  TS=TS+1;CALL FLD_STK;CALL ID_FCON:
  IF BUG THEN DO:BUG='0'B:
  UNSPEC(CODE(K))=SI||BIT(C1,13)||FC:K=K+1:GO TO ENT; END:
          UNSPEC(CODE(K))=SI||CR||FD; K=K+1:
          UNSPEC(CODE(K))=SI||BIT(C1,13)||FC: K=K+1:
          GO TO ENT:


/* DEFINED FIELD STACK AND POP CODE GENERATORS */
DFST: TS=TS+1; X=SUBSTR(UNSPEC(TAB(TS)),9,12);
          UNSPEC(CODE(K))=LI||FD||SUBSTR(UNSPEC(X),20,13):K=K+1:
          TS=TS+1; GO TO ENT:
DFPOP:  TS=TS+1:X=SUBSTR(UNSPEC(TAB(TS)),9,12):
          UNSPEC(CODE(K))=SI||SUBSTR(UNSPEC(X),20,13)||FD:K=K+1:
          TS=TS+1; GO TO ENT:


/* DO OPERATION CODE GENERATOR */
  DOO:  HOLD=K+3:
      UNSPEC(CODE(K))=LI||RET||BIT(HOLD,13):K=K+1;
      TS=TS+2:X=SUBSTR(UNSPEC(TAB(TS)),1,8):
        IF X=53 THEN
           DO:ROP=LI:FILL=RET:CALL BRANCH;FB='1'B;
            END:
        ELSE DO: UNSPEC(CODE(K))=LI||RET||(13)'0'B:
                 K=K+1:END:
        TS=TS-1:ROP=B:FILL=(13)'0'B:CALL BRANCH:
        IF FB THEN DO:FB='0'B;TS=TS+2:GO TO ENT: END:
        ELSE TS=TS+1:GO TO ENT:


/* LOGICAL OPERATIONS CODE GENERATOR */
OR:  LOP='011010'B:GO TO LOG:
AND: LOP='011011'B:GO TO LOG:
XOR: LOP='011100'B: GO TO LOG:
COMP:  LOP='011101'B:
    LOG:TS=TS+1:
    X=SUBSTR(UNSPEC(TAB(TS)),1,9):
    IF X=80 THEN
      DO:CALL FLD_STK: CALL ID_FCON: GO TO LOG1:END:
    ELSE CALL INT_GET:
    LOG1: UNSPEC(CODE(K))=LI||SYS||CR:K=K+1:
    X=SUBSTR(UNSPEC(TAB(TS)),1,9):
    IF X=80 THEN
       DO:CALL FLD_STK:CALL ID_CONT:GO TO LOG2:END:
```

```
    ELSE CALL INT_GET:
    LOG2: UNSPEC(CODE(K))=L||SYS||CR;K=K+1:
    UNSPEC(CODE(K))=LOP||(13)'0'B||SYS: K=K+1:
    IF BUG THEN DO:BUG='0'B:
    UNSPEC(CODE(K))=S||BIT(C1,13)||SYS;K=K+1;GO TO LOG3;END:
      UNSPEC(CODE(K))=S||CR||FD:K=K+1:
      UNSPEC(CODE(K))=SI||BIT(C1,13)||SYS:K=K+1:
 LOG3: IF LOP='011101'B THEN DO:
        UNSPEC(CODE(K))=S||CR||SYS:K=K+1;END:
      GO TO ENT:


/* SHIFT OPERATIONS CODE GENERATOR */
 LSH: SHOP=SHIFTL: GO TO SH_1:
 RSH:SHOP=SHIFTR:
   SH_1:TS=TS+1:CALL FLD_STK:CALL ID_FCON: FD_PTR=C1:
   UNSPEC(CODE(K))=L||SYS||CR:K=K+1:/*PUT FIRST FIELD ON SYS*/
   X=SUBSTR(UNSPEC(TAB(TS)),1,8):
   IF X='90'B THEN
    DO:CALL FLD_STK:CALL ID_CONT:END:
   ELSE CALL INT_GET:
   UNSPEC(CODE(K))=LR||DR||CR:K=K+1: /*PUT NUMBER OF SHIFTS IN DR*/
   UNSPEC(CODE(K))=S||CR||SYS:K=K+1:
   UNSPEC(CODE(K))=SHOP||CR||DR:K=K+1:
UNSPEC(CODE(K))=L||SYS||CR:K=K+1:
   X=SUBSTR(UNSPEC(TAB(TS)),1,8);
   IF X¬='90'B&X¬='52'B THEN GO TO SH_3:
   IF X='90'B THEN
  DO: CALL FLD_STK: CALL ID_FCON:
    IF SHOP=SHIFTL THEN
    DO:UNSPEC(CODE(K))=SHR||(13)'0'B||BIT(C1,13);K=K+1;GO TO SH_2:
    END:
    ELSE UNSPEC(CODE(K))=SHL||(13)'0'B||BIT(FD_PTR,13):K=K+1:
    GO TO SH_2;
 END:
    ELSE CALL INT_GET:
      IF SHOP=SHIFTL THEN
      DO:UNSPEC(CODE(K))=SHR||(26)'0'B:K=K+1:GO TO SH_2;END:
      ELSE UNSPEC(CODE(K))=SHL||(13)'0'B||BIT(FD_PTR,13):K=K+1:
   SH_2: UNSPEC(CODE(K))=L||SYS||CR:K=K+1:
   UNSPEC(CODE(K))=AD||(13)'0'B||SYS:K=K+1:
   SH_3: IF BUG THEN
     DO:BUG='0'B:UNSPEC(CODE(K))=S||BIT(FD_PTR,13)||SYS:
       K=K+1:GO TO ENT:
     END:
   ELSE UNSPEC(CODE(K))=S||CR||FD:K=K+1:
        UNSPEC(CODE(K))=SI||BIT(FD_PTR,13)||SYS:K=K+1:
```

```
    GO TO ENT:
  FLSH:
    /*NOT IMPLEMENTED */
  FRSH:
    /* NOT IMPLEMENTED */


 /* DONE AND FAIL OPERATIONS CODE GENERATOR */
  DON:
   UNSPEC(CODE(K))=SI||CR||RET; K=K+1:
    UNSPEC(CODE(K))=BP||(13)'0'B||RET: K=K+1;
   TS=TS+1: GO TO ENT:
  FAIL:
    UNSPEC(CODE(K))=BP||(13)'0'B||RET; K=K+1;
     TS=TS+1: GO TO ENT:


 /* BEGIN AND STOP STATEMENT CODE GENERATOR */
  BEG: J=J+1;SR_BR(J)=K:
       IF J>1 THEN DO:UNSPEC(CODE(K))=B||(26)'0'B;
                      K=K+1: RETURN:

                    END: RETURN:
  STOP:IF J>1 THEN
        DO:
  UNSPEC(CODE(SR_BR(J)))=SUBSTR(UNSPEC(CODE(SR_BR(J))),1,19)||
        BIT(K,13);
  J=J-1; RETURN:
        END;
       ELSE
        UNSPEC(CODE(K))= EOP||(26)'0'B; K=K+1;
  RETURN:


/* CONDITIONAL STATEMENT CODE GENERATOR */
  IFANY: CC=35; CB1=BT:GO TO COND_ENT:
  IF[illegible]:CC=36;CB1=BF; GO TO COND_ENT:
   IFNONE:CC=37:CB1=BT;GO TO COND_ENT:
  IFNALL:CC=38:CB1=BF:
COND_ENT:
     IF OPF THEN
      DO:
       IF B1_MK¬=0 THEN
        DO:
          UNSPEC(CODE(B1_MK))=SUBSTR(UNSPEC(CODE(B1_MK)),1,19)||
```

```
                    BIT(K,13);
               UNSPEC(CODE(K))=CB1||(26)'0'B: B1_MK=K:
               DEF='0'B:K=K+1;
               IF LAST THEN GO TO LASTCON:
               ELSE GO TO ENT;
             END:
           ELSE
             DO:
               UNSPEC(CODE(K))=CB1||(26)'0'B:
               B1_MK=K:B1_TOP=K:
               K=K+1:DEF='0'B:
               IF LAST THEN GO TO LASTCON:
               ELSE GO TO ENT;
             END:
           END:
         LASTCON: IF LAST THEN
                  DO: LAST='0'B:
                  UNSPEC(CODE(K))=B||(26)'0'B:
                  B2_MK=K:K=K+1:
                  IF CC=35|CC=38 THEN
                       DO:
                         Y=B1_TOP:
                        DO WHILE (Y¬=0):
                        Z=Y:Y=SUBSTR(UNSPEC(CODE(Z)),20,13):
                        UNSPEC(CODE(Z))=SUBSTR(UNSPEC(CODE(Z)),1,19)||
                          BIT(K,13):
                        END:
                        UNSPEC(CODE(B1_MK))=SUBSTR(UNSPEC(CODE(B1_MK)),1,19)
                          ||BIT(K,13):
                        TS=TS+1:GO TO ENT:
                          END;
                  ELSE UNSPEC(CODE(B2_MK))=SUBSTR(UNSPEC(CODE(B2_MK)),1,19)
                       ||BIT(K,13):
    END:
    TS=TS+1:GO TO ENT:

/* TH   STATEMENT AND COMMA HANDLING */
 TH   S=TS+1:GO TO ENT:
 COMMA:TS=TS+1: GO TO ENT:



/* ARITHMETIC OPERATIONS CODE GENERATOR */
   ADD: AOP=AD: GO TO ARITH;
   SUBT: AOP=SUB: GO TO ARITH:
   MULT: AOP=MUL: GO TO ARITH:
```

```
DIV: AOP=DV;
   ARITH:CALL ID_FCON;
      UNSPEC(CODE(K))=L||SYS||CR; K=K+1;
   TS=TS+1; X=SUBSTR(UNSPEC(TAB(TS)),1,8);
     IF X=90 THEN
       DO;CALL FLD_STK;CALL ID_CONT;GO TO ARITH1;END;
     ELSE CALL INT_GET;
   ARITH1: UNSPEC(CODE(K))=L||SYS||CR; K=K+1;
           UNSPEC(CODE(K))= AOP||(13)'0'B||SYS;; K=K+1;
     IF BUG THEN DO;BUG='0'B;
        UNSPEC(CODE(K))=S||BIT(C1,13)||SYS;K=K+1;GO TO ENT;END;
      UNSPEC(CODE(K))=S||CR||FD; K=K+1; /*GET POINTER*/
      UNSPEC (CODE(K))=SI||BIT(C1,13)||SYS;K=K+1;
   GO TO ENT;
   B_ENT:UNSPEC(CODE(K))=S||BIT(FD_PTR,13)||SYS;K=K+1;
   GO TO ENT;


/* TEST OPERATIONS CODE GENERATOR */
EQ: TEST='010001'B;
    GO TO TEST_OP;
NOTEQ: TEST ='010010'B;
    GO TO TEST_OP;
GRT: TEST ='010011'B;
    GO TO TEST_OP;
LESS: TEST='010100'B;
    TEST_OP:
     CALL ID_CONT;UNSPEC(CODE(K))=L||SYS||CR;K=K+1;
     TS=TS+1; X=SUBSTR(UNSPEC(TAB(TS)),1,8);
     IF X=90 THEN
        DO;
        CALL FLD_STK; CALL ID_CONT;UNSPEC(CODE(K))=L||SYS||CR;
        K=K+1;GO TO TEST_SET; END;
     IF X=52 THEN
       DO;
        CALL INT_GET; UNSPEC(CODE(K))= L||SYS||CR;
          K=K+1;
     END;
    TEST_SET:
     UNSPEC(CODE(K))=TEST||(13)'0'B||SYS; K=K+1;
     TS=TS+1;
      X=SUBSTR(UNSPEC(TAB(TS)),1,8);
     IF X=39 THEN     /* LAST TEST */
      DO;
        LAST='1'B; DEF='1'B;
       GO TO COND(CC);
     END;
```

```
      ELSE       /* MORE TESTS FOLLOW */
         DO:
          CALL FLD_STK:
          DEF='1'B;
          GO TO COND(CC):
        END:


/* PROCEDURE FLD_STK STACKS THE ELBAT WORDS OF CONCATENATED IDENTS*/
   FLD_STK: PROC:  ID=1:
      TSTK=0:
      ID_ENT: IDSTK(ID)=TAB(TS):
     TS=TS+1:
     X=SUBSTR(UNSPEC(TAB(TS)),1,8):
     IF X= 49 THEN    /* OTHERS FOLLOW*/
     DO:
      TS=TS+1: ID=ID+1:
       GO TO ID_ENT:
     END:
     ELSE  RETURN;
   END FLD_STK:


   INT_GET: PROC:
   /*RETURNS DESIRED VALUE TO THE C REGISTER AND IN VARIABLE VAL */
     SCRAMWORD=UNSPEC(INFO(SUBSTR(UNSPEC(TAB(TS)),21,12))):
     COUNT=SUBSTR(UNSPEC(SCRAMWORD),1,8):
      LOC=SUBSTR(UNSPEC(SCRAMWORD),9,12):
      VAL=SUBSTR(CHRSTOR,LOC,COUNT):
      UNSPEC(CODE(K))=LRI||CR||SUBSTR(UNSPEC(VAL),20,13):
      K=K+1: TS=TS+1:
   END INT_GET:


  ID_FCON:PROC:
   /*PUTS IDENTIFIER CONTENTS IN C REGISTER , BLOCK POINTER
     ON ED PUSHDOWN AND FIELD DEFINITION IN VARIABLE C1 */
    C1=SUBSTR(UNSPEC(IDSTK(1)),9,12):
    UNSPEC(CODE(K))=LR||CR||BIT(C1,13):
    K=K+1:
    IF ID=1 THEN DO: BUG='1'B:GO TO IDP:END:
    DO I=2 TO ID: IF I=ID THEN DO: UNSPEC(CODE(K))=LI||ED||CR:K=K+1:
       END:
       C1=SUBSTR(UNSPEC(IDSTK(I)),9,12):  /*C1 IS FIELD NUMBER */
       UNSPEC(CODE(K))=LRID||CR||BIT(C1,13):K=K+1:
```

```
          END;
          END;
          ID=1;
          IDSTK=0;
          RETURN;
        END ID_FCON;


        ID_CONT:PROC; /*PUTS CONTENTS IN C REGISTER */
           C2=SUBSTR(UNSPEC(IDSTK(1)),9,12);
           UNSPEC(CODE(K))=LR||GR||BIT(C2,13);K=K+1;
           IF ID=1 THEN GO TO IDQ;
           DO I =2 TO ID;
             C2=SUBSTR(UNSPEC(IDSTK(I)),9,12);
             UNSPEC(CODE(K))=LRID||GR||BIT(C2,13);K=K+1;
           END;
           IDQ: ID=1;
                IDSTK=0;
              RETURN;
        END ID_CONT;


      /* EXIT ROUTINE */
         XIT: IF CC=0 THEN RETURN;
           IF CC=35|CC=38 THEN
             DO;
             UNSPEC(CODE(B2_MK))=SUBSTR(UNSPEC(CODE(B2_MK)),1,19)
               ||BIT(K,13);
             RETURN;
             END;
           ELSE
             DO;
             Y=B1_TOP;
             IF B1_MK=B1_TOP THEN GO TO XIT_1;
                DO WHILE (Y¬=0);
                Z=Y;Y=SUBSTR(UNSPEC(CODE(Z)),20,13);
                UNSPEC(CODE(Z))=SUBSTR(UNSPEC(CODE(Z)),1,19)||BIT(K,13);
                END;
             XIT_1: UNSPEC(CODE(B1_MK))=SUBSTR(UNSPEC(CODE(B1_MK)),1,19)||
               BIT(K,13);
             RETURN;
             END;
      END CODEGEN;

     INPROC:PROC;
       DCL VAL FIXED BIN(31)EXT,
           CODE(2000)FIXED BIN(31)EXT,
```

```
   ''L_1 FIXED BIN(13),
      FIXED BIN(13) EXTERNAL,
      FIXED BIN(13) EXT, IO_OP FIXED BIN(13)EXT,
   MK_1 FIXED BIN(13) STATIC,
   (L  INIT('100001'B),MLD INIT('010110'B),SHIFTR INIT('001111'B),
   LR  INIT('000001'B),
    SUB INIT('010111'B),BF INIT('001100'B),E INIT('010001'B),
    L  INIT('000100'B),LI INIT('000101'B),AD INIT('010000'B),
    PUSHDUP INIT('100010'B),S INIT('000111'B),WSD INIT('100000'B),
    SCR INIT('100011'B),SHIFTL INIT('001110'B),SCD INIT('100100'B),
    LRI INIT('000010'B),ALOG INIT('011111'B),
    GTO INIT('001101'B),B INIT('001010'B),BT INIT('001011'B),
    MUL INIT('011000'B),SI INIT('001000'B))BIT(6) STATIC,
   FCTR BIT(13),
   (SYS INIT('0000000001011'B),IP INIT('0000000010000'B),
    OR INIT('0000000001010'B),CR INIT('0000000001001'B),
    FC INIT('0000000001101'B),
    FD INIT('0000000001100'B) ) BIT(13) STATIC,
   FD_PTR FIXED BIN(13) EXT,
   BUG BIT(1) EXT,
   MK_2 FIXED BIN(13)STATIC,
   INP_ENT(1:4) LABEL (INPC,INPB,INPD,DUPE) INIT(INPC,INPB,
      INPD,DUPE);
   GO TO INP_ENT(IO_OP);


  /* CHARACTER INPUT OPERATION CODE GENERATOR */
INPC: IF VAL <=4 THEN
      DO;
       UNSPEC(CODE(K))=LP||SYS||IP;K=K+1; RETURN;
      END;
      ELSE
       UNSPEC(CODE(K))=LR||DR||BIT(C1,13);K=K+1;
       UNSPEC(CODE(K))=LRI||CR||'0000000010000'B;K=K+1;
       UNSPEC(CODE(K))=SHIFTR||DR||CR; K=K+1;
       UNSPEC(CODE(K))=LI||SYS||DR;K=K+1;
       UNSPEC(CODE(K))=LP||SYS||FD;K=K+1;
       UNSPEC(CODE(K))=AD||(13)'0'B||SYS; K=K+1;
       VAL_1=((VAL-1)/4+1);
       UNSPEC(CODE(K))=LI||FC||BIT(VAL_1,13);K=K+1;
       MK_1=K;
       UNSPEC(CODE(K))=PUSHDUP||(13)'0'B||SYS; K=K+1;
       UNSPEC(CODE(K))=SI||DR||IP; K=K+1;
       UNSPEC(CODE(K))=WSD||SYS||DR; K=K+1;
       UNSPEC(CODE(K))=LI||SYS||'0000000000001'B; K=K+1;
       UNSPEC(CODE(K))=AD||(13)'0'B||SYS; K=K+1;
          UNSPEC(CODE(K))=LI||FC||'0000000000001'B;K=K+1;
```

```
            UNSPEC(CODE(K))=SUB||(13)'0'B||FC;K=K+1:
            UNSPEC(CODE(K))=PUSHDUP||(13)'0'B||FC;K=K+1:
            UNSPEC(CODE(K))=LI||FC||(13)'0'B:K=K+1:
            UNSPEC(CODE(K))=E||(13)'0'B||FC;K=K+1:
            UNSPEC(CODE(K))=LP||SYS||FC:K=K+1:
            UNSPEC(CODE(K))=BC||(13)'0'B||BIT(MK_1,13):K=K+1:
            UNSPEC(CODE(K))=S||DR||FC:K=K+1:
            UNSPEC(CODE(K))=S||DR||SYS:K=K+1:
          RETURN:


      /* BINARY INPUT OPERATION CODE GENERATOR */
  INPB: UNSPEC(CODE(K))=LI||SYS||(13)'0'B: K=K+1:
    INPB_1:UNSPEC(CODE(K))=SCB||DR||IP: K=K+1:
           VAL=VAL-4;
         IF VAL<=0 THEN GO TO INPB_2;
         UNSPEC(CODE(K))=LRI||CR||SUBSTR(UNSPEC(VAL),20,13):K=K+1:
          UNSPEC(CODE(K))=SHIFTL||DR||CR:
          K=K+1:
    INPB_2:UNSPEC(CODE(K))=LI||SYS||DR:K=K+1:
           UNSPEC(CODE(K))=AD||(13)'0'B||SYS: K=K+1:
           IF VAL>0 THEN GO TO INPB_1:
           ELSE RETURN:


/* DECIMAL INPUT OPERATION CODE GENERATOR */
  INPD: UNSPEC(CODE(K))=LI||SYS||(13)'0'B: K=K+1:
        IF VAL/4 = 2 THEN
         DO:
          UNSPEC(CODE(K))=SCD||DR||IP: K=K+1:
          UNSPEC(CODE(K))=LI||SYS||DR: K=K+1:
          UNSPEC(CODE(K))=LI||SYS||'0000000010000'B: K=K+1:
          UNSPEC(CODE(K))=MUL||(13)'0'B||SYS: K=K+1:
          VAL=VAL-4;
         END:
        IF VAL/4 = 1 THEN
         DO:
          UNSPEC(CODE(K))= SCD||DR||IP: K=K+1:
          UNSPEC(CODE(K))=LI||SYS||DR: K=K+1:
          UNSPEC(CODE(K))=AD||(13)'0'B||SYS: K=K+1:
          VAL=VAL-4:
         END:
        VAL = MOD(VAL,4):
        IF VAL=0 THEN RETURN;
        IF VAL=1 THEN
         DO: FCTR='0000000000010'B;GO TO INPD_1: END:
```

```
      IF VAL = 2 THEN
       DO:ECTR='000000000100'B: GO TO INPD_1; END:
      IF VAL=3 THEN
          ECTR='000000001000'B;
  INPD_1: UNSPEC(CODE(K))=LI||SYS||ECTR; K=K+1;
          UNSPEC(CODE(K))=MUL||(13)'0'B||SYS:K=K+1:
          UNSPEC(CODE(K))=SCD||DR||IP:K=K+1:
          UNSPEC(CODE(K))=L||SYS||DR; K=K+1;
          UNSPEC(CODE(K))=AD||(13)'0'B||SYS:K=K+1:
          RETURN:


   /* DUPLICATE OPERATION CODE GENERATOR */
DUPE:
          UNSPEC(CODE(K))=L||SYS||CR:K=K+1:
          UNSPEC(CODE(K))=L||EC||CR:K=K+1:
          UNSPEC(CODE(K))=WLD||CR||SYS:K=K+1:
          UNSPEC(CODE(K))=LRI||DR||'000000000001'B:K=K+1:
          UNSPEC(CODE(K))=SHIFTL||CR||DR:K=K+1:
          UNSPEC(CODE(K))=LRI||DR||'000000011101'B:K=K+1:
          UNSPEC(CODE(K))=SHIFTR||CR||DR:K=K+1:
          UNSPEC(CODE(K))=L||SYS||CR:K=K+1; /*ALOC SIZE ON SYS */
          UNSPEC(CODE(K))=ALOC||(13)'0'B||SYS:K=K+1:
          UNSPEC(CODE(K))=S||DR||SYS:K=K+1:
          UNSPEC(CODE(K))=L||SYS||DR:K=K+1:
          UNSPEC(CODE(K))=GTO||(24)'0'B;K=K+1:
          UNSPEC(CODE(K))=PUSHDUP||(13)'0'B||SYS:K=K+1:
       IF BUG THEN
         DO:
          BUG='0'B:
          UNSPEC(CODE(K))=S||BIT(C1,13)||SYS:
         K=K+1:
          GO TO DUP_1:
         END:
          UNSPEC(CODE(K))=S||CR||ED:K=K+1:
          UNSPEC(CODE(K))=SI||BIT(C1,13)||SYS:K=K+1:
          DUP_1: UNSPEC(CODE(K))=LPI||ED||SYS:K=K+1:
          UNSPEC(CODE(K))=L||SYS||DR:K=K+1:
          MK_1=K:
          UNSPEC(CODE(K))=LI||SYS||'000000000001'B:K=K+1:
          UNSPEC(CODE(K))=SUB||(13)'0'B||SYS:K=K+1:
          UNSPEC(CODE(K))=PUSHDUP||(13)'0'B||SYS:K=K+1:
          UNSPEC(CODE(K))=LI||SYS||(13)'0'B:K=K+1:
          UNSPEC(CODE(K))=E||(13)'0'B||SYS:K=K+1:
          MK_2=K:K=K+1:
          UNSPEC(CODE(K))=PUSHDUP||(13)'0'B||EC:K=K+1:
          UNSPEC(CODE(K))=PUSHDUP||(13)'0'B||ED:K=K+1:
```

```
                UNSPEC(CODE(K))=WLD||CR||EC:K=K+1:
                UNSPEC(CODE(K))=WSD||ED||CR:K=K+1:
                UNSPEC(CODE(K))=LI||EC||'000000000001'B:K=K+1:
                UNSPEC(CODE(K))=LI||ED||'000000000001'B:K=K+1:
                UNSPEC(CODE(K))=AD||(13)'0'B||ED:K=K+1:
                UNSPEC(CODE(K))=SD||(13)'0'B||EC:K=K+1:
                UNSPEC(CODE(K))=B||(18)'0'B||BIT(MK_1,13):K=K+1;
                UNSPEC(CODE(MK_2))=BT||(18)'0'B||BIT(K,13):
                UNSPEC(CODE(K))=WLD||CR||[illegible]=K+1:
                UNSPEC(CODE(K))=WSD||ED||CR:K=K+1:
                UNSPEC(CODE(K))=S||CP||SYS:K=K+1:
        RETURN;
    END INPROC:


BRANCH: PROC:
  /* THE BRANCH PROCEDURE INSERTS ADDRESSES FOR BRANCH AND DO OPS*/
    DCL TAB(200) FIXED BIN(31) EXT,
        INFO(0:4095)FIXED BIN(31)EXT,
        TS FIXED BIN(31)EXT,
        K FIXED BIN(13) EXT,
        CODE(2000)FIXED BIN(31)EXT,
        BOP BIT(6) EXT, FILL BIT(13)EXT,
        (Y,Z) FIXED BIN(31):
        Y=SUBSTR(UNSPEC(TAB(TS)),21,12):
        Z=SUBSTR(UNSPEC(INFO(Y)),21,12):
        IF SUBSTR(UNSPEC(INFO(Z)),9,1)='0'B THEN  /*LABEL FILLED*/
         DO:
          UNSPEC(CODE(K))=BOP||FILL||'0'B||
          SUBSTR(UNSPEC(INFO(Z)),9,12):K=K+1:TS=TS+1:RETURN;
         END:
        ELSE
         DO:
          Y=SUBSTR(UNSPEC(INFO(Z)),10,11):
          IF Y=0 THEN   /* FIRST TO FILL */
             DO:
              UNSPEC(INFO(Z))=SUBSTR(UNSPEC(INFO(Z)),1,9)||
                SUBSTR(UNSPEC(K),22,11)||SUBSTR(UNSPEC(INFO(Z)),21,12):
              UNSPEC(CODE(K))=BOP||
              FILL||(13)'0'B:
              K=K+1:TS=TS+1:RETURN;
             END:
          ELSE
            DO WHILE (Y¬=0);
              Z=Y:
              Y=SUBSTR(UNSPEC(CODE(Y)),7,26);
            END:
```

```
            UNSPEC(CODE(Z))=SUBSTR(UNSPEC(CODE(Z)),1,19)||BIT(K,13):
            UNSPEC(CODE(K))=BOP||
             FILL||(13)'0'B:
            K=K+1;TS=TS+1:
            RETURN:
END BRANCH;


   CORRECT:PROC; /*ALLOWS ON-LINE PROGRAM CORRECTION*/
     DCL 1 FCB EXT,
           2 COMMAND CHAR(8),
             2 FILENAME CHAR(8),
             2 FILETYPE CHAR(8),
             2 CARD_NO FIXED BIN(31),
             2 STATUS FIXED BIN(31),
             2 CARD_BF CHAR(80),
             LINECT FIXED BIN(31)EXT,
             LINE CHAR(8) VARYING,
             BPTR FIXED BIN(31) EXT,
             TRYAGIN CHAR(8) VARYING;


        DISPLAY('SYNTAX ERROR IN STATEMENT '||LINECT):
        DISPLAY('ENTER LINE NUMBER:') REPLY(LINE):
        CARD_NO=LINE;
        IF CARD_NO=0 THEN RETURN:
        ELSE COMMAND='FINIS';CALL IHEFILE(FCB);
        COMMAND='WRBUF':
        DISPLAY('ENTER CORRECT LINE:')REPLY(CARD_BF):
        CALL IHEFILE(FCB):
        DISPLAY('ENTER FIRST LINE NUMBER OF CORRECTED STATEMENT:')
        REPLY(TRYAGIN):
        CARD_NO=TRYAGIN:
        BPTR=2: CARD_NO=CARD_NO-1:
        COMMAND ='FINIS';CALL IHEFILE(FCB):
        RETURN:
   END CORRECT;



PAMBLDR: PROC OPTIONS (MAIN):
DCL
        PAM FILE OUTPUT ENVIRONMENT(CONSECUTIVE F(800,80)),
        1 AM(4:105),
             2 INF BIN FIXED(31),
```

```
          2 PTR BIN FIXED(31),
     INCR BIN FIXED INIT(3),
     RELSTR CHAR(1140) VAR,
     PREL BIN FIXED INIT(1),
     1 ECB EXT,
          2 COMMAND CHAR(8) INIT('RDBUF'),
          2 FILENAME CHAR(8) INIT('PAMS'),
          2 FILETYPE CHAR(8) INIT('FILE'),
          2 CARD_NO FIXED BIN(31,0) INIT(0),
          2 STATUS FIXED BIN(31,0),
          2 CARD_RE CHAR(80),
     (P,Q,R,S,I,J,T,U) BIN FIXED,
     BUFF CHAR(80) VAR:

/*   BUILDS ASSOCIATIVE LISTS WITH THE AM(ADDRESS)
     CORRESPONDING TO THE SYMBOL CLASS IDENTIFIER INTEGER.
     LISTS CONTAIN THE ADDRESSES, IF ANY, IN THE PRODUCTION
     SEQUENTIAL STORE AND PRECEDENCE HEADERS(>,=,<)(2,3,4).
     IF ANY. WITH A POINTER TO THE PRECEDENCE SEQUENTIAL
     STORE, IF NECESSARY.

                              ->
                             /
          TADDTADDT         /       T--2T-3?T
                           /
          TADDTADDT       /         TADDT--0T
          -T-------                 -T-------
            |                         |
            |                         |
            ->                        ->

     PRODUCTION SEQ STORE            PRECEDENCE SEQ STORE
     T-----------------------        T--------------------------

     BUILDS THE PRECEDENCE SEQUENTIAL STORE.  POINTER FIELDS
     ASSIGNED AS REQUIRED.                                      */

     AM = 0:
P = 195;
DO I = 4 TO 90:
     IF I = P THEN DO:
          DISPLAY('FREE AREA EXHAUSTED'):
          GOTO JMP1:
          END:
     J = I;
JMP2: CARD_NO = CARD_NO + 1:
```

```
          CARD_BF = ' ';
          CALL IHEFILE(FCB);
          GET STRING(CARD_BF) LIST (R,S,T,U);
          IF R > 1 & R < 5 THEN DO;
               CALL RELS;
               IF J = 0 THEN GOTO JMP3;
               GOTO JMP2;
               END;
          IF R = 0 THEN GOTO JMP1;
          INF(J) = (R * 65536) + S;
          IF U = 1 THEN DO;
               U = P;
               P = P - 1;
               PTR(J) = (T*65536) + U;
               J = U;
               GOTO JMP2;
               END;
JMP3:  END;
JMP1:     FILENAME = 'PAMOUT';
          FILETYPE = 'FILE';
          COMMAND = 'WRBUF';
          CARD_NO = 1;
JMP4:     CARD_BF = ' ';
          CARD_BF = 'INF WORDS';
          CALL IHEFILE(FCB);
          CARD_NO = CARD_NO + 1;
          DO I = 4 TO 196 BY 6;
          CARD_BF = ' ';
          PUT STRING(CARD_BF) EDIT ((INF(J+I) DO J = 0 TO 5))
               (F(12));
          CALL IHEFILE(FCB);
          CARD_NO = CARD_NO + 1;
          END;
          CARD_BF = ' ';
          CARD_BF = 'PTR WORDS';
          CALL IHEFILE(FCB);
          CARD_NO = CARD_NO + 1;
          DO I = 4 TO 196 BY 6;
               CARD_BF = ' ';
               PUT STRING(CARD_BF) EDIT ((PTR(J+I) DO J = 0 TO 5))
                    (F(12));
               CALL IHEFILE(FCB);
               CARD_NO = CARD_NO + 1;
          END;
          RELSTOR = RELSTOR||'//';
          CARD_BF = ' ';
          CARD_BF = 'RELSTOR';
          CALL IHEFILE(FCB);
```

```
          CARD_NO = CARD_NO + 1:
          CARD_RE = ' ';
          P = I;
          J = 1:
JMP45:    IF SUBSTR(RELSTOR,P,2) = '//' THEN GOTO JMP5:
          BUFF = BUFF||SUBSTR(RELSTOR,P,2):
          J = J + 2:
          P = P + 2:
          IF J/73 = 1 THEN DO:
               J = 1:
               CARD_RE = BUFF:
               BUFF = '':
               CALL IHEFILE(FCB):
               CARD_NO = CARD_NO + 1:
               CARD_RE = ' ':
               END:
          GOTO JMP45:
JMP5:     CARD_RE = BUFF:
          CALL IHEFILE(FCB):
          RELS: PROC:
               DCL
                    RE BIN FIXED,
                    NUM CHAR(2) INIT(' '):
               INF(J) = (P * 65536) + S:
               IF U = 1 THEN DO:
                    U = P:
                    P = P - 1:
                    END:
               IF T = 1 THEN T = PREL:
               PTR(J) = (T * 65536) + U:
               J = U:
               IF T ¬= 0 THEN DO:
REL1:               RE = 1:
                    CARD_NO = CARD_NO + 1:

                    CARD_RE = ' ':
                    CALL IHEFILE(FCB):
                    DO WHILE(NUM ¬= '00'):
                         NUM = SUBSTR(CARD_RE,RE,2):
                         RELSTOR = RELSTOR||NUM:
                         RE = RE + 2:
                    PREL = PREL + 8:
                         IF RE = 73 THEN GOTO REL1:
                    END:
          END RELS;
END PAMPLDR:
```

APPENDIX 2
THE BLISS INTERPRETER

```
INTERP: PROC(PARMS) OPTIONS(MAIN);
      DCL
          PARMS CHAR(16) VAR,
          (PARM1,PARM2) CHAR(8) EXT,
          MEM(0:16383) BIN FIXED(31) EXT,  /*BLISS MACHINE.*/
          LHEAD(0:7) BIN FIXED(31)    /*AVAILABLE SPACE LIST HEAD PTRS */
                     DEFINED MEM(1SUB),    /* SUBSCRIPT = LOG2 SIZE.*/
          (INT,R1,R2,B1,B2,D1,P,OPCODE,I,J,LPT,RPT) BIN FIXED,
          WORD BIN FIXED EXT,
          SCHAR CHAR(1),
          MCHAR CHAR(4) VAR,
          BSTG BIT(4) VAR,
          MACI BIT(32),
          LAB(43) LABEL(LR,LRI,LRID,L,LI,LID,S,SI,SRID,B,BT,BE,GET,
                        SHIFTL,SHIFTR,ADD,E,NE,GT,LT,EOP,WLD,SUB,MULT,
                        DIV,OR,AND,XCR,COMP,FREE,ALOG,WSD,LP,PDUP,SCB,
                        SCD,BP,PC,PB,PD,SHL,SHR,DP)
              INITIAL(LR,LRI,LRID,L,LI,LID,S,SI,SRID,B,BT,BE,GET,
                      SHIFTL,SHIFTR,ADD,E,NE,GT,LT,EOP,WLD,SUB,MULT,DIV,
                      OR,AND,XCR,COMP,FREE,ALOG,WSD,LP,PDUP,SCB,SCD,BP,
                      PC,PB,PD,SHL,SHR,DP),
        1 FCB EXT,
          2 COMMAND CHAR(8),
          2 FILENAME CHAR(8),
          2 FILETYPE CHAR(8),
          2 CARD_NO FIXED BIN(31),
          2 STATUS FIXED BIN(31),
          2 CARD_BF CHAR(80),
          FREG BIN FIXED(31),
          DUMPB BIT(6),
          A(6) BIN FIXED,
          COUNTBT ENTRY(BIN FIXED(31)) RETURNS(BIN FIXED) EXT,
           DUMP ENTRY(BIN FIXED,BIN FIXED,BIN FIXED,BIN FIXED,
                      BIN FIXED,BIN FIXED) EXT,
           POP ENTRY(BIN FIXED(31)) RETURNS(BIN FIXED(31)) EXT,
           PUSH ENTRY(BIN FIXED,BIN FIXED(31)) EXT,
           OUTDIAG ENTRY EXT,
           BRANCH ENTRY EXT,
           FRE_CEL ENTRY(BIN FIXED) EXT,
           INP ENTRY EXT,
           GETP ENTRY(CHAR(4)) EXT,
           TBUFF CHAR(20) VAR,
```

```
          CBUFF CHAR(82) VAR CONTROLLED,
          BBUFF BIT(81) CONTROLLED,
          BUFFER CHAR(130) INIT(' ') EXT,
          TWO BIN FIXED EXT INIT(2),
             (SW,TRACE) BIT(1) EXT INIT('0'B),
             (L1,L2) BIN FIXED,
             ANS CHAR(4) VAR,
          GET_CEL ENTRY(BIN FIXED,) EXT:
        PARM1 = SUBSTR(PARMS,1,8):
        PARM2 = SUBSTR(PARMS,9,8):

/*      MACHINE DUMP TO "FILE LISTING" ON STANDARD PL1 ERROR.  */

         ON ERROR BEGIN:
              CALL DUMP(1,1,1,1,1,1):
              GOTO JMP2:
              END:

/*      IF ANSWER IS "YES", MACHINE REGISTERS, OPCODE, INST. NUMBER,
         PUSH, AND POP VALUES ARE DISPLAYED FOR EACH INST. BETWEEN
         THOSE SPECIFIED.                                          */

        DISPLAY('PROGRAM TRACE?')REPLY(ANS):
         IF ANS = 'YES' THEN DO:
              SW = '1'B:
              DISPLAY('FROM CODE LINE = ')REPLY(ANS):
              L1 = ANS:
              DISPLAY('TO CODE LINE = ')REPLY(ANS):
              L2 = ANS:
         END:
      MEM = 0:
      LHEAD(7) = 8388735:

/* THE NEXT BLOCK BREAKS AVAILABLE SPACE INTO CELLS OF 128 WORDS
       EACH AND DOUBLE LINKS THEM TO LIST HEAD SEVEN.            */

      BEGIN:
          DCL
             (I,LPTR,RPTR) BIN FIXED:
          LPTR = 0:
          P = 128:
          DO I = 1 TO 126:
              RPTR = P + 128:
              MEM(P) = (LPTR*16384)+RPTR:
              UNSPEC(MEM(P)) = '1111'B||SUBSTR(UNSPEC(MEM(P)),5,28):
              LPTR = P:
              P = RPTR:
          END:
```

```
            MEM(P) = LPTR*16384;
            UNSPEC(MEM(P)) = '1111'B||SUBSTR(UNSPEC(MEM(P)),5,28);
        END;

/* READ TEXT CODE FROM FILENAME "PARMS" AND STORE IN DOUBLE LINKED
        128 WORD BLOCKS.  MEM(8) POINTS TO ACTIVE INSTRUCTION.      */

        FILENAME = PARM1;
        FILETYPE = 'TEXT';
        COMMAND = 'RDBUF';
        CARD_NO = 1;
        STATUS = 0;
        P = 129;
        LPT = 0;
        CALL GET_CEL(7,MEM(8));
        DO WHILE(STATUS = 0);
            CALL IHEFILE(FCB);
            J = 1;
            DO I = 1 TO 6;
                MEM(P) = SUBSTR(CARD_BF,J,12);
                J = J + 12;
                P = P + 1;
                IF MOD(P,128) = 0 THEN DO;
                        CALL GET_CEL(7,MEM(8));
                        P = P - 128;
                        UNSPEC(MEM(P)) = SUBSTR(UNSPEC(MEM(P)),1,18)||
                                         SUBSTR(UNSPEC(MEM(8)),19,14);
                        LPT,P = MEM(8);
                        UNSPEC(MEM(P))= SUBSTR(UNSPEC(MEM(P)),1,4)||
                            SUBSTR(UNSPEC(LPT),19,14)||REPEAT('0'B,13);
                        P = P + 1;
                END;
            END;
            CARD_NO = CARD_NO + 1;
        END;
            FILENAME = 'FILE';
            FILETYPE = 'LISTING';
            COMMAND = 'ERASE';
            CALL IHEFILE(FCB);
        CARD_NO = 0;

/* NEXT ELEVEN: DECODES OPCODE AND BRANCHES TO THE BLOCK THAT EXECUTES
        THE INSTRUCTION, ADVANCES THE MEM(8), AND HALTS ON STOP.    */

        MEM(8) = 129;
        MEM(15) = 1;
        DO WHILE(MEM(8) ¬= 0);
            MACI = UNSPEC(MEM(MEM(8)));
```

```
          OPCODE = SUBSTR(MACI,1,6);
IF SW THEN
          IF MEM(15)>=L1 & MEM(15)<=L2 THEN TRACE = '1'B;
          ELSE TRACE = '0'B;
IF TRACE THEN DISPLAY('NEXT INST '||MEM(15)||', OPCODE = '||OPCODE);
          GOTO LAB(OPCODE);
JMP1:     MEM(8) = MEM(8) +1;
          IF TRACE THEN DO;
               DISPLAY('C REG ='||UNSPEC(MEM(9)));
               DISPLAY('D REG ='||UNSPEC(MEM(10)));
          END;
          MEM(15) = MEM(15) + 1;
          IF MOD(MEM(8),128) = 0 THEN DO;
               MEM(8) = MEM(8) - 129;
               MEM(8) = SUBSTR(UNSPEC(MEM(MEM(8))),19,14);
               MEM(8) = MEM(8) + 1;
          END;
     END;
     GOTO JMP2;

/* LOADS CONTENTS OF R2 INTO R1 */
LR:  R1 = SUBSTR(MACI,7,13);
          R2 = SUBSTR(MACI,20,13);
          MEM(R1) = MEM(R2);
          GOTO JMP1;

/* LOADS THE SPECIFIED INTEGER INTO R1 */
LRI:      R1 = SUBSTR(MACI,7,13);
          INT = SUBSTR(MACI,20,13);
          MEM(R1) = INT;
          GOTO JMP1;

/* LOADS CONTENTS OF FIELD, DEFINED IN R2 AND LOCATED IN THE
          BLOCK POINTED TO BY R1.   INTO R1.                    */
LRID:     R1 = SUBSTR(MACI,7,13);
          R2 = SUBSTR(MACI,20,13);
          MACI = UNSPEC(MEM(R2));
          WORD = SUBSTR(MACI,5,12);
          WORD = WORD + MEM(R1);
          B1 = SUBSTR(MACI,17,8);
          B2 = SUBSTR(MACI,25,8);
          IF B1 = 0 THEN DO;
               B2 = B2 - 2;
               CALL STKFLD;
               MEM(R1) = MEM(WORD+B2+1);
          END;
     ELSE DO;
               IF (B2-B1+1) = 32 THEN
```

```
      UNSPEC(MEM(R1)) = SUBSTR(UNSPEC(MEM(WORD)),B1,(B2-B1+1)):
            ELSE
            MEM(R1) = SUBSTR(UNSPEC(MEM(WORD)),B1,(B2-B1+1)):
         END:
         GOTO JMP1:

/* PUSHES THE CONTENTS OF R1 ONTO PUSHDOWN P1. */
L:    R1 = SUBSTR(MACI,20,13):
         P1 = SUBSTR(MACI,7,13):
         CALL PUSH(P1,MEM(R1)):
         GOTO JMP1:

/* PUSHES THE SPECIFIED INTEGER ONTO PUSHDOWN P1. */
LI:    INT = SUBSTR(MACI,20,13):
         P1 = SUBSTR(MACI,7,13):
         CALL PUSH(P1,INT):
         GOTO JMP1:

/* PUSHES THE CONTENTS OF THE FIELD, DEFINED IN R1 AND LOCATED
      IN THE BLOCK POINTED TO BY THE TOP OF P1, ONTO PUSHDOWN P1. */
LID:     R1 = SUBSTR(MACI,20,13):
         P1 = SUBSTR(MACI,7,13):
         MACI = UNSPEC(MEM(R1)):
         WORD = SUBSTR(MACI,5,12):
         WORD = WORD + POP(P1):
         B1 = SUBSTR(MACI,17,8):
         B2 = SUBSTR(MACI,25,8):
         IF B1 = 0 THEN DO:
               B2 = B2 - 1:
               CALL STKFLD:
         END:
         ELSE DO:
         TREG = SUBSTR(UNSPEC(MEM(WORD)),B1,(B2-B1+1)):
         CALL PUSH(P1,TREG):
         END:
         GOTO JMP1:

/* STORES THE TOP OF PUSHDOWN P1 INTO REGISTER R1. */
S:     R1 = SUBSTR(MACI,7,13):
         P1 = SUBSTR(MACI,20,13):
         MEM(R1) = POP(P1):
         GOTO JMP1:

/* STORES THE TOP OF PUSHDOWN P1 INTO A FIELD WHICH IS
      DEFINED IN R1 IN THE BLOCK POINTED TO BY RC.        */
SI:      R1 = SUBSTR(MACI,7,13):
         P1 = SUBSTR(MACI,20,13):
         MACI = UNSPEC(MEM(P1)):
```

```
          WORD = SUBSTR(MACI,5,12):
          WORD = WORD + MEM(9):
          B1 = SUBSTR(MACI,17,8):
          B2 = SUBSTR(MACI,25,8):
          IF B1 = 0 THEN DO:
               B2 = B2 - 1:
               CALL UNSTKE:
          END:
          ELSE DO:
          TREG = POP(B1):
          MACI = UNSPEC(MEM(WORD)):
          SUBSTR(MACI,B1,(B2-B1+1)) = SUBSTR(UNSPEC(TREG),(32-(B2-B1)),
                                      (B2-B1+1)):
          UNSPEC(MEM(WORD)) = MACI:
          END:
          GOTO JMP1:

/* STORES CONTENTS OF R2 INTO A FIELD WHICH IS DEFINED IN
          R1 IN THE BLOCK POINTED TO BY RC.                    */
SRIC:     B1 = SUBSTR(MACI,7,13):
          R2 = SUBSTR(MACI,20,13):
          MACI = UNSPEC(MEM(B1)):
          WORD = SUBSTR(MACI,5,12):
          WORD = WORD + MEM(9):
          B1 = SUBSTR(MACI,17,8):
          B2 = SUBSTR(MACI,25,8):
          IF B1 = 0 THEN DO:
               MEM(WORD+B2-1) = MEM(R1):
               B2 = B2 - 2:
               CALL UNSTKE:
          END:
          ELSE DO:
          MACI = UNSPEC(MEM(WORD)):
        SUBSTR(MACI,B1,(B2-B1+1)) = SUBSTR(UNSPEC(MEM(R2)),(32-(B2-B1)),
                                    (B2-B1+1)):
          UNSPEC(MEM(WORD)) = MACI:
          END:
          GOTO JMP1:

/* UNCONDITIONAL BRANCH TO ADDRESS "WORD". */
B:        WORD = SUBSTR(MACI,20,13):
          CALL BRANCH:
          GOTO JMP1:

/* BRANCH IF TRUE (TOP OF SYSTEM STACK = 1).  */
BT:       IF POP(11) = 1 THEN DO:
               WORD = SUBSTR(MACI,20,13):
               CALL BRANCH:
```

```
          END;
          GOTO JMP1;

 /* BRANCH IF FALSE (TOP OF SYSTEM STACK = 0). */
 BF:      IF POP(11) = 0 THEN DO;
               WORD = SUBSTR(MACI,20,13);
               CALL BRANCH;
          END;
          GOTO JMP1;

 /* OBTAINS A BLOCK TO THE NEXT LARGEST POWER OF 2 COMPARED
          WITH THE VALUE ON THE TOP OF THE SYSTEM PUSHDOWN. */
 GET:     R1 = POP(11);
          IF R1 ¬= 1 THEN R1 = LOG2(R1-1) + 1;
          ELSE R1 = 0;
          CALL GET_CEL(R1,TREG);
          CALL PUSH(11,TREG);
          GOTO JMP1;

 /* SHIFTS REGISTER R1 LEFT BY R2 BITS. */
 SHIFTL:  R1 = SUBSTR(MACI,7,13);
          R2 = SUBSTR(MACI,20,13);
           UNSPEC(MEM(R1)) = SUBSTR(UNSPEC(MEM(R1)),MEM(R2)+1);
          GOTO JMP1;

 /* SHIFTS REGISTER R1 RIGHT BY R2 BITS. */
 SHIFTR:   R1 = SUBSTR(MACI,7,13);
          R2 = SUBSTR(MACI,20,13);
          MACI = (32) '0'B;
          UNSPEC(MEM(R1)) = SUBSTR(MACI,1,MEM(R2))||UNSPEC(MEM(R1));
          GOTO JMP1;

 /* ADDS THE TOP TWO ELEMENTS OF THE DESIGNATED PUSHDOWN AND
          STORES THE RESULT ON THE SAME PUSHDOWN.                    */
 ADD:     P1 = SUBSTR(MACI,20,13);
          CALL PUSH(P1,POP(P1) + POP(P1));
          GOTO JMP1;

 /* THE NEXT FOUR ENTRIES TEST THE TOP TWO ELEMENTS ON THE SYSTEM
          PUSHDOWN AND LEAVE A "1" ON TOP IF THE CONDITION IS TRUE -
          "0" IF FALSE.                                              */
 E:       P1 = SUBSTR(MACI,20,13);
           IF POP(P1) = POP(P1) THEN CALL PUSH(P1,1);
          ELSE CALL PUSH(P1,0);
          GOTO JMP1;
 NE:      P1 = SUBSTR(MACI,20,13);
          IF POP(P1) ¬= POP(P1) THEN CALL PUSH(P1,1);
          ELSE CALL PUSH(P1,0);
```

```
          GOTO JMP1:
GT:       P1 = SUBSTR(MACI,20,13):
          IF POP(P1) < POP(P1) THEN CALL PUSH(P1,1):
          ELSE CALL PUSH(P1,0):
          GOTO JMP1:
LT:       P1 = SUBSTR(MACI,20,13):
          IF POP(P1) > POP(P1) THEN CALL PUSH(P1,1):
          ELSE CALL PUSH(P1,0):
          GOTO JMP1:

/* BRANCH TO END OF PROGRAM  */
 EOP:      CALL DUMP(1,0,1,1,1,1):
           GOTO JMP2:

/* THE NEXT SIX ENTRIES OPERATE ON THE TOP TWO ELEMENTS OF THE
    DESIGNATED PUSHDOWN, TREATING THEM AS INVERTED REVERSE POLISH
          NOTATION.  THE RESULT REMAINS ON TOP.                    */
SUB:        P1 = SUBSTR(MACI,20,13):
            CALL PUSH(P1,-(POP(P1)) + POP(P1)):
          GOTO JMP1:
MULT:       P1 = SUBSTR(MACI,20,13):
            CALL PUSH(P1,POP(P1) * POP(P1)):
          GOTO JMP1:
DIV:        P1 = SUBSTR(MACI,20,13):
            TREG = POP(P1):
            CALL PUSH(P1,POP(P1)/TREG):
          GOTO JMP1:
OR:         P1 = SUBSTR(MACI,20,13);
  CALL PUSH(P1,(BOOL(UNSPEC(POP(P1)),UNSPEC(POP(P1)),'0111'))):
          GOTO JMP1:
AND:        P1 = SUBSTR(MACI,20,13):
  CALL PUSH(P1,(BOOL(UNSPEC(POP(P1)),UNSPEC(POP(P1)),'0001'))):
          GOTO JMP1:
XOR:        P1 = SUBSTR(MACI,20,13):
   CALL PUSH(P1,(BOOL(UNSPEC(POP(P1)),UNSPEC(POP(P1)),'0110'))):
          GOTO JMP1:

/* COMPLEMENTS THE TOP OF THE DESIGNATED PUSHDOWN  */
COMP:       P1 = SUBSTR(MACI,20,13):
          CALL PUSH(P1,(¬ UNSPEC(POP(P1)))):
          GOTO JMP1:

/* FREES THE BLOCK POINTED TO BY THE TOP OF THE SYSTEM PUSHDOWN. */
FREE: CALL FRE_CEL(POP(11)):
          GOTO JMP1:

/* PUTS THE FULL WORD POINTED TO BY PUSHDOWN P1 INTO REGISTER R1 */
WLD:       R1 = SUBSTR(MACI,7,13):
```

```
          P1 = SUBSTR(MACI,20,13);
          MEM(R1) = MEM(POP(P1));
          GOTO JMP1;

/* TAKES ANTI-LOG (BASE 2) OF TOP OF PUSHDOWN P1 AND PUTS THE
          RESULT ON TOP.                                          */
ALOG:     P1 = SUBSTR(MACI,20,13);
          CALL PUSH(P1,TWO**POP(P1));
          GOTO JMP1;

/* STORES THE FULL WORD IN REGISTER R1 INTO THE MEMORY WORD
          POINTED TO BY PUSHDOWN P1.                              */
WSD:      P1 = SUBSTR(MACI,7,13);
          P1 = SUBSTR(MACI,20,13);
          MEM(POP(P1)) = MEM(R1);
          GOTO JMP1;

/* PUSHES THE TOP OF PUSHDOWN P2 ONTO THE PUSHDOWN P1. */
LP:       P1 = SUBSTR(MACI,7,13);
          P2 = SUBSTR(MACI,20,13);
          CALL PUSH(P1,POP(P2));
          GOTO JMP1;

/* COPIES TOP OF PUSHDOWN P1 ONTO P1. */
PDUP:     P1 = SUBSTR(MACI,20,13);
          TREG = POP(P1);
          CALL PUSH(P1,TREG);
          CALL PUSH(P1,TREG);
          GOTO JMP1;

/* POPS PUSHDOWN P1, CONVERTS FROM CHARACTER STRING TO
          BIT STRING, AND STORES RESULT IN R1.                    */
SCB:      P1 = SUBSTR(MACI,20,13);
          R1 = SUBSTR(MACI,7,13);
          TREG = POP(P1);
          MCHAR = '';
          DO I = 1 TO 25 BY 8;
               UNSPEC(SCHAR) = SUBSTR(UNSPEC(TREG),I,8);
               IF SCHAR = ' ' THEN;
               ELSE MCHAR = MCHAR||SCHAR;
          END;
          BSTG = MCHAR;
          MEM(R1) = BSTG;
          GOTO JMP1;

/* POPS PUSHDOWN P1, CONVERTS FROM A CHARACTER STRING TO A
          BINARY FIXED INTEGER, AND STORES THE RESULT IN R1.      */
SCD:      R1 = SUBSTR(MACI,7,13);
```

```
          R1 = SUBSTR(MACI,20,13);
          UNSPEC(MCHAR) = UNSPEC(POP(R1));
          MEM(R1) = MCHAR;
          GOTO JMP1;

/* BRANCHES TO THE CODE ADDRESS ON TOP OF PUSHDOWN R1. */
BP:       R1 = SUBSTR(MACI,20,13);
          WORD = POP(R1);
          CALL BRANCH;
          GOTO JMP1;

/* LOCATES THE FIELD DEFINED IN R1, IN THE BLOCK POINTED TO
          BY REGISTER C, AND PADDING IT WITH BLANKS, MAKES
          THE OUTPUT EQUAL TO R2 CHARACTERS.                    */
BC:       R1 = SUBSTR(MACI,7,13);
          R2 = SUBSTR(MACI,20,13);
          IF R1 < 40 THEN DO;
               CALL PUSH(17,MEM(R1));
               R2 = 1;
               R2 = R2/4;
               IF R2 > B2 THEN DO;
                    UNSPEC(TREG) = UNSPEC('    ');
                    DO I = 1 TO (R2-B2);
                         CALL PUSH(17,TREG);
                    END;
               END;
               GOTO JMP1;
          END;
          MACI = UNSPEC(MEM(R1));
          WORD = SUBSTR(MACI,5,12);
          B1 = SUBSTR(MACI,17,8);
          B2 = SUBSTR(MACI,25,8);
          WORD = WORD + MEM(9);
          IF B1 = 0 THEN DO;
               R2 = R2/4;
               IF B2 > R2 THEN B1 = R2;
               ELSE B1 = B2;
               DO I = 1 TO B1;
                    CALL PUSH(17,MEM(WORD));
                    WORD = WORD + 1;
               END;
               IF R2 > B2 THEN DO;
                    UNSPEC(TREG) = UNSPEC('    ');
                    DO I = 1 TO (R2-B2);
                         CALL PUSH(17,TREG);
                    END;
               END;
          END;
```

```
          ELSE DO:
               ALLOCATE CBUFF:
               B2 = (B2-B1+1)/8:
               DO I = 1 TO B2:
                    UNSPEC(SCHAR) = SUBSTR(UNSPEC(MEM(WORD)),B1,8):
                    B1 = B1 + 8:
                    CBUFF = CBUFF||SCHAR:
               END:
               CBUFF = CBUFF||REPEAT(' ',128):
               DO I = 1 TO R2 BY 4:
                    MCHAR = SUBSTR(CBUFF,I,4):
                    UNSPEC(TREG) = UNSPEC(MCHAR):
                    CALL PUSH(17,TREG):
               END:
               FREE CBUFF:
          END:
          GOTO JMP1:

     /* LOCATES THE FIELD DEFINED IN R1, IN THE BLOCK POINTED TO BY
          REGISTER C, AND,AFTER CONVERTING, OUTPUTS THE CHARACTER
          VARIABLE, PADDING IT WITH BLANKS TO MAKE THE OUTPUT EQUAL TO
          R2 CHARACTERS.                                              */
PB:       R1 = SUBSTR(MACI,7,13):
          R2 = SUBSTR(MACI,20,13):
          IF R1 < 40 THEN DO:
               B1 = R1:
               R1 = 32:
               ALLOCATE BBUFF,CBUFF:
               BBUFF = UNSPEC(MEM(R1)):
               CBUFF = BBUFF||REPEAT(' ',128):
               GOTO PB1;
          END:
          MACI = UNSPEC(MEM(R1)):
          ALLOCATE CBUFF:
          WORD = SUBSTR(MACI,5,12):
          B1 = SUBSTR(MACI,17,8):
          B2 = SUBSTR(MACI,25,8):
          R1 = B2 - B1 + 1:
          WORD = WORD + MEM(9):
          ALLOCATE BBUFF:
          BBUFF = SUBSTR(UNSPEC(MEM(WORD)),B1,R1):
          CBUFF = BBUFF||REPEAT(' ',128):
PB1:      DO I = 1 TO R2 BY 4:
               MCHAR = SUBSTR(CBUFF,I,4):
               UNSPEC(TREG) = UNSPEC(MCHAR):
               CALL PUSH(17,TREG):
          END:
          FREE BBUFF,CBUFF:
```

```
          GOTO JMP1;

/* LOCATES THE FIELD DEFINED IN R1, IN THE BLOCK POINTED TO BY
          REGISTER C, AND, AFTER CONVERTING, OUTPUTS THE CHARACTER
          VARIABLE, PADDING IT WITH BLANKS TO MAKE THE OUTPUT EQUAL
          TO R2 CHARACTERS.                                          */
PD:       R1 = SUBSTR(MACI,7,13);
          R2 = SUBSTR(MACI,20,13);
          ALLOCATE CBUFF;
          IF R1 < 40 THEN DO;
               TBUFF = CHAR(MEM(R1));
               GOTO PD1;
          END;
          MACI = UNSPEC(MEM(R1));
          WORD = SUBSTR(MACI,5,12);
          B1 = SUBSTR(MACI,17,8);
          B2 = SUBSTR(MACI,25,8);
          WORD = WORD + MEM(9);
          R1 = B2 - B1 + 1;
          TREG = SUBSTR(UNSPEC(MEM(WORD)),R1,R1);
          TBUFF = CHAR(TREG);
     PD1: SCHAR = SUBSTR(TBUFF,1,1);
          IF SCHAR = ' ' THEN DO;
               TBUFF = SUBSTR(TBUFF,2);
               GOTO PD1;
          END;
          CBUFF = TBUFF||REPEAT(' ',128);
          DO I = 1 TO R2 BY 4;
               MCHAR = SUBSTR(CBUFF,I,4);
               UNSPEC(TREG) = UNSPEC(MCHAR);
               CALL PUSH(17,TREG);
          END;
          FREE CBUFF;
          GOTO JMP1;

/* SHIFTS RC LEFT X BITS. X = THE LENGTH OF FIELD R1 -
          THE NUMBER OF BITS IT WAS SHIFTED RIGHT.                   */
SHL:      R1 = SUBSTR(MACI,20,13);
          R2 = MEM(10);
          MACI = UNSPEC(MEM(R1));
          B1 = SUBSTR(MACI,17,8);
          B2 = SUBSTR(MACI,25,8);
          R2 = B2 - B1 + 1;
          B2 = B2 - R2;
          UNSPEC(MEM(9)) = SUBSTR(UNSPEC(MEM(9)),B2+1);
          GOTO JMP1;

/* LEFT JUSTIFIES RC, THEN RIGHT SHIFTS IT 32 - THE PREVIOUS
```

```
        LEFT SHIFT.                                                    */
SHR:     R1 = SUBSTR(MACI,20,13):
         IF R1 = 0 THEN DO:
              R1 = COUNTBT(MEM(9)):
              UNSPEC(MEM(9)) = SUBSTR(UNSPEC(MEM(9)),R1):
         END:
         ELSE DO:
         MACI = UNSPEC(MEM(R1)):
         B1 = SUBSTR(MACI,17,8):
         B2 = SUBSTR(MACI,25,8):
         B2 = B2 -B1 + 1:
         IF B2 -= 32 THEN
         UNSPEC(MEM(9)) = SUBSTR(UNSPEC(MEM(9)),(32-B2)):
           END:
         B2 = MEM(10):
         MACI = (32) '0'B:
         UNSPEC(MEM(9)) = SUBSTR(MACI,1,(32-B2))||UNSPEC(MEM(9)):
         GOTO JMP1;

/* INVOKES THE DUMP PROCEDURE.  THE TWO PARAMETERS IN OCTAL BREAK
        INTO THE DUMP PARAMETERS (X,X,X,X,X,X).                        */
DP:    DUMPB = SUBSTR(MACI,17,3)||SUBSTR(MACI,30,3):
         DO I = 1 TO 6:
              IF SUBSTR(DUMPB,I,1) = '1'B THEN A(I) = 1:
              ELSE A(I) = 0:
         END:
         CALL DUMP(A(1),A(2),A(3),A(4),A(5),A(6)):
         GOTO JMP1:

/*      STKFLD AND UNSTKF ARE USED TO MOVE THE CONTENTS OF FIELDS OF
         SIZE GREATER THAN ONE WORD.                                   */

         STKFLD: PROC:
              DO I = 0 TO B2:
                   CALL PUSH(11,MEM(WORD+I)):
              END;
         END STKFLD:
         UNSTKF: PROC:
              DO I = B2 TO 0 BY -1:
                   MEM(WORD+I) = POP(11):
              END:
         END UNSTKF:
JMP2: END INTER:


      OUTDIAG:PROC:
              DCL    BUFFER CHAR(130)EXT,
```

```
          DUMP ENTRY(BIN FIXED,BIN FIXED,
               BIN FIXED,BIN FIXED,BIN FIXED,BIN FIXED) EXT:

/*      PROCEDURE TO OUTPUT MACHINE ERROR DIAGNOSTICS.    */

        DISPLAY(BUFFER):
        BUFFER = ' ':
      CALL DUMP(1,1,1,1,1,1):
    END OUTDIAG:


      BRANCH: PROC;
          DCL
               (INDEX,BRAI,BJ,BRK,BK) BIN FIXED,
               MEM(0:16383) BIN FIXED(31) EXT, WORD BIN FIXED EXT:

/*      HANDLES ALL BRANCHES TO CODE ADDRESS DESIGNATED IN "WORD".
        STARTS AT THE FIRST INSTRUCTION BLOCK, CHAINS TO THE CORRECT
        BLOCK, AND THEN TO THE INSTRUCTION ONE SHORT OF THE ONE
        DESIRED.                                                  */

          INDEX = 128;
          WORD = WORD - 1;
          MEM(15) = WORD;
          BRAI = MOD(WORD,127):
          BJ = WORD/127:
          IF BJ = 0 THEN GOTO BRA1:
          DO BK = 1 TO BJ;
               INDEX = SUBSTR(UNSPEC(MEM(INDEX)),19,14):
          END:
BRA1:     MEM(8) = INDEX + BRAI:
      END BRANCH:


      COUNTBT: PROC(REGS) FIXED BIN;
          DCL
               REGS BIN FIXED(31), CBI BIN FIXED, CBT BIT(1):

/*      LOCATES THE FIRST "1" BIT IN A REGISTER AND RETURNS THE
        BIT NUMBER.  IF NO "1" BITS, RETURNS 32.                  */

          DO CBI = 1 TO 32:
               CBT = SUBSTR(UNSPEC(REGS),CBI,1):
               IF CBT = '1'B THEN GOTO CBT1:
          END:
          RETURN(32):
```

```
CBT1:         RETURN(CBI):
          END COUNTBT:


     GET_CEL: PROC(A,GP):
/* A = CELL SIZE DESIRED (LOG2), GP = POINTER TO CELL PROVIDED.      */
          DCL
                    TRACE BIT(1) EXT,
                    MEM(0:16383) BIN FIXED(31) EXT,
                    LHEAD(0:7) BIN FIXED(31) DEFINED MEM(1SUB),
               DUMP ENTRY(BIN FIXED, BIN FIXED, BIN FIXED, BIN FIXED,
                    BIN FIXED,BIN FIXED) EXT, TWO BIN FIXED EXT,
               (A,GP,GCT,GTP) BIN FIXED, GTEMP BIT(32):
/* THIS PROCEDURE GETS THE FIRST CELL TO THE RIGHT OF THE CORRECT
          SIZE LIST HEAD.  IF THE LIST IS EMPTY, CALLS "BREAK" TO
          PROVIDE PROPER SIZED CELL.                                  */
          IF LHEAD(A) = 0 THEN CALL BREAK:
          GP = SUBSTR(UNSPEC(LHEAD(A)),1,16):
          GTP = SUBSTR(UNSPEC(MEM(GP)),18,14):
          UNSPEC(MEM(GP)) = '0'B||SUBSTR(UNSPEC(MEM(GP)),2,3)||
                                                   REPEAT('0'B,27):
          IF GTP = 0 THEN LHEAD(A) = 0:
          ELSE DO:
               UNSPEC(LHEAD(A)) = SUBSTR(UNSPEC(GTP),17,16)||
                                  SUBSTR(UNSPEC(LHEAD(A)),17,16):
               LHEAD(A) = LHEAD(A) - 1:
               GTEMP = UNSPEC(MEM(GTP)):
               SUBSTR(GTEMP,5,14) = REPEAT('0'B,13):
               UNSPEC(MEM(GTP)) = GTEMP:
          END:
          BREAK: PROC:
/* THIS PROCEDURE SUPPLIES A CELL OF THE DESIRED SIZE BY:
          1. LOCATING THE NEXT LARGEST AVAILABLE CELL.
          2. SPLITTING THAT CELL RECURSIVELY UNTIL THE PROPER
               SIZE IS OBTAINED.
          3. INTERMEDIATE CELL LISTS ARE EACH LEFT WITH ONE CELL      */
               DCL
                    (B,Q,RPTR,LPTR)FIXED BIN,
                    SIZ BIT(3):
               B = A:
               DO WHILE(LHEAD(B) = 0):
```

```
                    B = B + 1;
                    IF B > 7 THEN DO;
                        PUT STRING(BUFFER)LIST(' MEMORY EXHAUSTED ');
                        CALL OUTDIAG;
                        GP = 16384;
                        RETURN;
                    END;
                END;
                DO WHILE(B ¬= A);
                    Q = SUBSTR(UNSPEC(LHEAD(B)),1,16);
                    GCT = SUBSTR(UNSPEC(LHEAD(B)),17,16);
                    GCT = GCT - 1;
                    IF GCT = 0 THEN LHEAD(B) = 0;
                    ELSE DO;
                        GTP = SUBSTR(UNSPEC(MEM(Q)),19,14);
                        UNSPEC(LHEAD(B)) = SUBSTR(UNSPEC(GTP),17,16)||
                                           SUBSTR(UNSPEC(GCT),17,16);
                        GTEMP = UNSPEC(MEM(GTP));
                        SUBSTR(GTEMP,5,14) = REPEAT('0'B,13);
                        UNSPEC(MEM(GTP)) = GTEMP;
                    END;
                    B = B - 1;
                    SIZ = SUBSTR(UNSPEC(B),30,3);
                    UNSPEC(LHEAD(B)) = SUBSTR(UNSPEC(Q),17,16)||
                                       REPEAT('0'B,15);
                    LHEAD(B) = LHEAD(B) + 2;
                    RPTR = Q + (TWO**B);
                    GTEMP = '1'B||SIZ||REPEAT('0'B,13)||
                                       SUBSTR(UNSPEC(RPTR),19,14);
                    UNSPEC(MEM(Q)) = GTEMP;
                    LPTR = Q;
                    Q = RPTR;
                    GTEMP = '1'B||SIZ||SUBSTR(UNSPEC(LPTR),19,14)||
                                      REPEAT('0'B,13);
                    UNSPEC(MEM(Q)) = GTEMP;
                END;
            END BREAK;
        IF TRACE THEN DISPLAY('GET CELL, SIZE = '||A||', PTR = '||GP);
        END GET_CEL;


        FRE_CEL: PROC(FP) RECURSIVE;

    /* ATTEMPTS, ON PROGRAMMER'S COMMAND, TO RECONNECT FREED CELLS INTO
            THE LARGEST POSSIBLE AVAILABLE SPACE BLOCK.                    */

            DCL     TWO BIN FIXED EXT, TRACE BIT(1) EXT,
```

```
          MEM(0:16383) BIN FIXED(31) EXT, LHEAD(0:7) BIN FIXED(31)
                                                DEFINED MEM(1SUB),
         (FSIZ,PWR,FP,CVAL) BIN FIXED, FTEMP BIT(32):
     FSIZ = SUBSTR(UNSPEC(MEM(FP)),2,3):
     IF FSIZ = 7 THEN DO:
         CALL ADLIST:
         RETURN:
         END:
     UNSPEC(MEM(FP)) = '1'B||SUBSTR(UNSPEC(MEM(FP)),2,3):
     PWR = TWO**FSIZ:

/* CHECK TO SEE IF MATE CELL IS LOCATED ABOVE OR BELOW THE CELL
     BEING RETURNED TO MEMORY.                                    */

     IF MOD(FP,TWO**(FSIZ+1)) ¬= 0 THEN GOTO FRE1:
     CVAL = FP + PWR:
     IF MEM(CVAL) < 0 & SUBSTR(UNSPEC(MEM(CVAL)),2,3) =
                        SUBSTR(UNSPEC(FSIZ),30,3)
         THEN DO:
         CALL ADJ:
         UNSPEC(MEM(CVAL)) = '0'B||SUBSTR(UNSPEC(MEM(CVAL)),2,3):
         FSIZ = FSIZ + 1:
         FTEMP = UNSPEC(MEM(FP)):
         SUBSTR(FTEMP,2,3) = SUBSTR(UNSPEC(FSIZ),30,3):
         UNSPEC(MEM(FP)) = FTEMP:
         CALL FRE_CEL(FP):
         RETURN:
     END:
     ELSE DO:
         CALL ADLIST:
         RETURN:
     END:
FRE1:  CVAL = FP - PWR:
     IF MEM(CVAL) < 0 & SUBSTR(UNSPEC(MEM(CVAL)),2,3) =
                        SUBSTR(UNSPEC(FSIZ),30,3)
         THEN DO:
         CALL ADJ:
         FSIZ = FSIZ + 1:
         FTEMP = UNSPEC(MEM(CVAL)):
     SUBSTR(FTEMP,2,3) = SUBSTR(UNSPEC(FSIZ),30,3):
         UNSPEC(MEM(CVAL)) = FTEMP:
         FP = CVAL:
         CALL FRE_CEL(FP):
         RETURN:
     END:
     ELSE DO:
         CALL ADLIST:
         RETURN;
```

```
          END;

/* THE ADJ PROCEDURE REMOVES A CELL FROM AN AVAILABLE SPACE LIST SO
          THAT IT MAY BE COMBINED WITH ITS FREED MATE.                    */

          ADJ: PROC;
               DCL
                    (ALPT,ARPT) BIN FIXED, ATEMP BIT(32);
               ALPT = SUBSTR(UNSPEC(MEM(CVAL)),5,14);
               ARPT = SUBSTR(UNSPEC(MEM(CVAL)),19,14);
               IF ALPT = 0 THEN DO;
                    ATEMP = UNSPEC(LHEAD(FSIZ));
                    SUBSTR(ATEMP,1,16) = SUBSTR(UNSPEC(ARPT),17,16);
                    UNSPEC(LHEAD(FSIZ)) = ATEMP;
                    ALPT = 0;
               END;
               ELSE DO;
                    ATEMP = UNSPEC(MEM(ALPT));
                    SUBSTR(ATEMP,19,14) = SUBSTR(UNSPEC(ARPT),19,14);
                    UNSPEC(MEM(ALPT)) = ATEMP;
               END;
               IF ARPT ¬= 0 THEN DO;
                    ATEMP = UNSPEC(MEM(ARPT));
                    SUBSTR(ATEMP,5,14) = SUBSTR(UNSPEC(ALPT),19,14);
                    UNSPEC(MEM(ARPT)) = ATEMP;
               END;
               LHEAD(FSIZ) = LHEAD(FSIZ) - 1;
          END ADJ;

/* ADLIST PROCEDURE ATTACHES A FREED AND POSSIBLY RECOMBINED CELL TO
          AN AVAILABLE SPACE LIST AFTER ALL POSSIBLE RECOMBINATION IS
          COMPLETED.  THE CELL IS ATTACHED NEXT TO THE LIST HEAD CELL.*/

          ADLIST: PROC;
               DCL
                    0 BIN FIXED, ADTEMP BIT(32);
               IF FSIZ ¬= 0 THEN DO;
                    DO I = 1 TO ((TWO**FSIZ)-1);
                       MEM(FP+I) = 0;
                    END;
               END;
               ADTEMP = UNSPEC(MEM(FP));
               Q = SUBSTR(UNSPEC(LHEAD(FSIZ)),1,16);
               SUBSTR(ADTEMP,5,28) = REPEAT('0'B,11)||
                                     SUBSTR(UNSPEC(Q),17,16);
               UNSPEC(MEM(FP)) = ADTEMP;
               IF Q ¬= 0 THEN DO;
                    ADTEMP = UNSPEC(MEM(Q));
```

```
                SUBSTR(ADTEMP,5,14) = SUBSTR(UNSPEC(FP),19,14):
                UNSPEC(MEM(0)) = ADTEMP:
            END:
            ADTEMP = UNSPEC(LHEAD(FSIZ));
            SUBSTR(ADTEMP,1,16) = SUBSTR(UNSPEC(FP),17,16):
            UNSPEC(LHEAD(FSIZ)) = ADTEMP:
            LHEAD(FSIZ) = LHEAD(FSIZ) + 1;
IF TRACE THEN DISPLAY('RETURN CELL, SIZ = '||FSIZ||' PTR = '||FP):
        END ADLIST:
    END FRE_CEL;


    POP:PROC(PDN) FIXED BIN(31) RECURSIVE:
        DCL    POP ENTRY(BIN FIXED(31)) RETURNS (BIN FIXED(31)),
            BUFFER CHAR(130)EXT, OUTDIAG ENTRY EXT,
                INP ENTRY EXT, FRE_CEL ENTRY(BIN FIXED) EXT,
                MEM(0:16383) BIN FIXED(31) EXT, TRACE BIT(1)EXT,
            (VAL,PDN) BIN FIXED(31), POPT BIN FIXED:
/*      RETURNS THE VALUE FROM THE TOP OF THE DESIGNATED PUSHDOWN
        STACK.  RETURNS TO STORAGE THOSE BLOCKS NO LONGER ACTIVE,
        UNLESS ONLY ONE BLOCK REMAINS.                              */
         IF MOD(MEM(PDN),32) = 0 THEN DO:
/*      CAUSES ERROR DIAGNOSTIC IF PD EMPTY.  IF INPUT BUFFER IS
        EMPTY, TRIGGERS INPUT REQUEST.                              */
            IF PDN = 16 THEN DO:
                CALL INP:
                VAL = POP(16):
                RETURN(VAL);
            END:
            IF PDN = 17 THEN DO:
                UNSPEC(VAL) = UNSPEC('    ');
                RETURN(VAL);
            END:
            ELSE DO:
                BUFFER = 'PUSHDOWN'||PDN||' EMPTY OR NEVER ACTIVATED'
                ||' - TERMINAL ERROR':
                CALL OUTDIAG:
            END:
        END:
        VAL = MEM(MEM(PDN)):
        MEM(PDN) = MEM(PDN) - 1;
        IF MOD(MEM(PDN),32) = 0 THEN DO:
            POPT = MEM(PDN):
```

```
          IF SUBSTR(UNSPEC(MEM(POPT)),17,16) ¬= PDN
               THEN DO:
               MEM(PDN) = SUBSTR(UNSPEC(MEM(POPT)),17,16) + 31:
               CALL FRE_CEL(POPT):
               END:
     END:
IF TRACE THEN DISPLAY('POP'||PDN||'VAL ='||UNSPEC(VAL)):
 RETURN(VAL):
 END POP:


 PUSH:PROC(PNUM,PUSHEE):
      DCL     MEM(0:16383) BIN FIXED(31) EXT,
              TRACE BIT(1) EXT,
              OUTP ENTRY(CHAR(4)) EXT, PCH CHAR(4),
             (PNUM,PUPT) BIN FIXED, PUSHEE BIN FIXED(31),
             GET_CEL ENTRY(BIN FIXED,) EXT:

/*       ON INITIAL CALL ACTIVATES PUSHDOWN STACKS.  INDEXES THE
         DESIGNATED PD AND LOADS THE VALUE.  ACTIVATES ADDITIONAL
         STORAGE AS NEEDED.                                      */

      IF MEM(PNUM) = 0 THEN DO:
               CALL GET_CEL(5,PUPT):


               MEM(PNUM) = PUPT:
               MEM(PUPT) = MEM(PUPT) + PNUM:
      END:
      MEM(PNUM) = MEM(PNUM) + 1:
      IF MOD(MEM(PNUM),32) = 0 THEN DO:

/*       CALLS FOR AUTODUMP OF PRINT BUFFER WHEN IT IS FULL.  */

            IF PNUM = 17 THEN DO:
                 MEM(PNUM) = MEM(PNUM) - 1:
                 UNSPEC(PCH) = UNSPEC(PUSHEE):
                 CALL OUTP(PCH):
                 RETURN:
            END:
             CALL GET_CEL(5,PUPT):
             MEM(PUPT) = MEM(PUPT) + MEM(PNUM) - 32:
             MEM(PNUM) = PUPT + 1:
      END:
      MEM(MEM(PNUM)) = PUSHEE:
IF TRACE THEN DISPLAY('PUSH'||PNUM||'VAL ='||UNSPEC(PUSHEE)):
 END PUSH:
```

```
INP: PROC:
        DCL
              1 FCB EXT,
                      2 COMMAND CHAR(8),
                      2 FILENAME CHAR(8),
                      2 FILETYPE CHAR(8),
                      2 CARD_NO FIXED BIN(31),
                      2 STATUS FIXED BIN(31),
                      2 CARD_BF CHAR(80),
                (PARM1,PARM2) CHAR(8) EXT,
              BUFF CHAR(120),(I,J) BIN FIXED,
              PUSH ENTRY(BIN FIXED,BIN FIXED) EXT,
              IACC CHAR(1), ICREG CHAR(4) VAR, BEREG BIN FIXED(31):

/*      READS 4 CHARACTER GROUPS INTO THE INPUT BUFFER FROM THE
        SOURCE DESIGNATED IN "PARM2".  SCANS THE INPUT LINE
        BACKWARDS DUE TO THE INPUT BUFFER PUSHDOWN CHARACTERISTICS. */

        IF PARM2 = 'CARDS' THEN DO:
                  COMMAND = 'RDBUF';
                  FILENAME = PARM1;
                  FILETYPE = 'DATA';
                  CARD_NO = CARD_NO + 1;
                  CARD_BF = ' ';
                  CALL IHEFILE(FCB);
                  BUFF = CARD_BF;
        END;
        ELSE DISPLAY('***')REPLY(BUFF);
        I = 1;
        IACC = ' ';
        DO WHILE(IACC ¬= '*');
              IACC = SUBSTR(BUFF,I,1);
              I = I + 1;
        END;
        J = 1;
        I = I - 1;
        ICREG = '';
INP1:   I = I - 1;
        IACC = SUBSTR(BUFF,I,1);
        ICREG = IACC||ICREG;
        IF J = 4 THEN DO:
              UNSPEC(BEREG) = UNSPEC(ICREG);
              CALL PUSH(16,BEREG);
              J = 1;
              ICREG = '';
```

```
            GOTO INP1;
        END;
        IF I <= 1 THEN GOTO INP2;
        J = J + I;
        GOTO INP1;
INP2:  END INP;


OUTP: PROC(OCREG);
        DCL
             BUFF CHAR(129) VAR, I BIN FIXED, OCREG CHAR(4),
             POP ENTRY(BIN FIXED) RETURNS(BIN FIXED) EXT;

/*      DUMPS THE OUTPUT BUFFER TO THE TERMINAL.    */

        BUFF = OCREG||BUFF;
        DO I = 1 TO 31;
             UNSPEC(OCREG) = UNSPEC(POP(17));
             BUFF = OCREG||BUFF;
        END;
        DISPLAY(BUFF);
END OUTP;


     DUMP: PROC(L,I,R,B,F,P);
         DCL  POP ENTRY(BIN FIXED(31))RETURNS(BIN FIXED(31)) EXT,
              MEM(0:16383) BIN FIXED(31) EXT,
              LHEAD(0:7) BIN FIXED(31) DEFINED MEM(1508),
             (DI,COUNT,DADD,DPT,DJ,L,I,R,B,F,P) BIN FIXED,
             LISTING FILE OUTPUT ENVIRONMENT(CONSECUTIVE F(3250,130)),
             DBUFF CHAR(127) INIT(' '),
             DBITS BIT(32),
             BUFOUT ENTRY(BIN FIXED), DPRINT ENTRY(BIN FIXED),
             (TPTPD,DPDC) BIN FIXED(31);
         OPEN FILE(LISTING);
         DBUFF = '  *** DUMP FOR BLISS INTERPRETER ***';
         CALL BUFOUT(1);
         IF L = 1 THEN GOTO DUM2;
DUM3:    IF I = 1 THEN GOTO DUM4;
DUM5:    IF R = 1 THEN GOTO DUM6;
DUM7:    IF B = 1 THEN GOTO DUM8;
DUM9:    IF F = 1 THEN GOTO DM10;
DM11:    IF P = 1 THEN GOTO DM12;
         GOTO DM1;

/*     DISPLAYS FREE STORAGE.  NUMBER OF FREE CELLS OF EACH SIZE AND
```

```
          THE ADDRESS OF THE FIRST CELL ON EACH SIZE LIST.              */

DUM2:      PUT STRING(DBUFF)EDIT('LIST HEAD NO','FIRST CELL',
                 'NUMBER ON LIST')(X(5),A(12),X(5),A(10),X(5),A(14)):
           CALL BUFOUT(2):
           CALL BUFOUT(0):
           DO DI = 0 TO 7:
               COUNT = SUBSTR(UNSPEC(LHEAD(DI)),17,16):
               DADD = SUBSTR(UNSPEC(LHEAD(DI)),1,16):
               PUT STRING(DBUFF)EDIT(DI,DADD,COUNT)(X(11),F(1),X(14),
                    F(5),X(12),F(3)):
               CALL BUFOUT(0):
           END:
           GOTO DUM3:

/*        LISTS THE NUMBER OF THE NEXT INSTRUCTION TO BE EXECUTED
          AT THE TIME OF THE DUMP, THE ACTIVE BLOCK OF 128 INSTS.,
          AND A MARKER IN THE BLOCK TO THE EXECUTION LOCATION.    */

DUM4:       DBUFF = ' ** NEXT INSTRUCTION NUMBER TO BE EXECUTED IS  '||
                    MEM(15):
           CALL BUFOUT(1):
           DBUFF = ' ** HARDWARE INSTRUCTION STACK **':
           CALL BUFOUT(1):
          PUT STRING(DBUFF)EDIT('INST NO','OPCODE','OPERAND 1',
               'OPERAND 2')(X(7),A(7),X(5),A(6),X(5),A(9),X(5),A(9)):
          CALL BUFOUT(1);
           DPT = MEM(8) - MOD(MEM(8),128):
           L = MEM(15) - MOD(MEM(15),127):
           DO DI = (DPT+1) TO (DPT+127):
               DBITS = UNSPEC(MEM(DI)):
               COUNT = SUBSTR(DBITS,1,6):
               DADD = SUBSTR(DBITS,7,13):
               DJ = SUBSTR(DBITS,20,13):
               IF DI = MEM(8) THEN DO:
                    DBUFF = '    ***EXECUTION LOCATION***':
                    CALL BUFOUT(0):
               END:
               L = L + 1:
               PUT STRING(DBUFF)EDIT(L,COUNT,DADD,DJ)
                    (X(9),F(4),X(8),F(2),X(10),F(4),X(10),F(4)):
               CALL BUFOUT(0):
           END:
           GOTO DUM5:

/*        DISPLAYS THE CONTENTS OF THE TWO SYSTEM REGISTERS.    */

DUM6:      DBUFF = '** CONTENTS OF THE CI REGISTER =  '||MEM(9):
```

```
          CALL BUFOUT(1);
          DBUFF = '** CONTENTS OF THE DI REGISTER =  '||MEM(10);
          CALL BUFOUT(0);
          GOTO DUM7;

/*      LISTS ALL BUG REGISTERS WITH THEIR CONTENTS.  */

DUM8:     DBUFF = '** BUG REGISTERS **  **CONTENTS:  DEC - BITS  **';
          CALL BUFOUT(2);
          DO DI = 18 TO 40;
              DBITS = UNSPEC(MEM(DI));
              PUT STRING(DBUFF)EDIT(DI,MEM(DI),DBITS)(X(8),F(2),X(12),
                  F(12),X(5),B(32));
              CALL BUFOUT(0);
          END;
          GOTO DUM9;

/*      LISTS ALL ACTIVE FIELD DEFINITIONS.   */

DM10:     DBUFF = '** FIELD NUMBER **  ** DEFINITION:  '||
                  'WORD - FROM - TO **';
              CALL BUFOUT(2);
          DO DI = 41 TO 127;
              IF MEM(DI) = 0 THEN GOTO DM11;
              DBITS = UNSPEC(MEM(DI));
              COUNT = SUBSTR(DBITS,5,12);
              DADD = SUBSTR(DBITS,17,8);
              DJ = SUBSTR(DBITS,25,8);
              PUT STRING(DBUFF)EDIT(DI,COUNT,DADD,DJ)(X(8),F(2),X(8),
                  F(3),X(5),F(3),X(5),F(3));
              CALL BUFOUT(0);
          END;
          GOTO DM11;

/*      DISPLAYS THE ACTIVE BLOCK CONTENTS OF THE FOUR PUSHDOWN
        STACKS.  DOES NOT ALTER THE STACK CONTENTS.          */

DM12:     DBUFF = '** CONTENTS OF THE SYSTEM PUSHDOWN **';
          CALL BUFOUT(2);
          CALL PDPRINT(11);
          DBUFF = '** CONTENTS OF THE FIELD DEFINITION PUSHDOWN **';
          CALL BUFOUT(2);
          CALL PDPRINT(12);
          DBUFF = '** CONTENTS OF THE FIELD CONTENTS PUSHDOWN **';
          CALL BUFOUT(2);
          CALL PDPRINT(13);
          DBUFF = '** CONTENTS OF THE SUBROUTINE RETURN PUSHDOWN **';
          CALL BUFOUT(2);
```

```
          CALL PDPRINT(14);

/*        PRINTS THE DESIGNATED PUSHDOWN STACK AND RETURNS THE
          INDEX TO ITS ORIGINAL POSITION.  IF A PD IS OVERLAYING
          MORE THAN ONE BLOCK, ONLY THE ACTIVE BLOCK IS PRINTED. */

          PDPRINT:PROC(PRPD);
               DCL
                     PRPD BIN FIXED;
               TPTPD = MEM(PRPD);
               IF TPTPD = 0 THEN GOTO PDP1;
               IF MOD(MEM(PRPD),32) = 0 THEN GOTO PDP1;
          PDP2:  IF MOD(MEM(PRPD),32) = 1 THEN GOTO PDP3;
               DPDC = POP(PRPD);
               DBUFF = UNSPEC(DPDC)||DPDC;
               CALL BUFOUT(0);
               GOTO PDP2;
          PDP3:  DPDC = MEM(MEM(PRPD));
               DBUFF = UNSPEC(DPDC);
               CALL BUFOUT(0);
         PDP1: MEM(PRPD) = TPTPD;
           END PDPRINT;

/*        WRITES OUTPUT INTO THE "FILE LISTING".     */

          BUFOUT:PROC(S);
               DCL
                     (SI,S)BIN FIXED,
                     BLANKS CHAR(127);
               IF S ¬= 0 THEN DO;
                    BLANKS = REPEAT(' ',126);
                     DO SI = 1 TO S;
                       PUT FILE(LISTING)LIST(BLANKS);
                     END;
               END;
               PUT FILE(LISTING)LIST(DBUFF);
               DBUFF = REPEAT(' ',126);
          END BUFOUT;
DM1  :END DUMP;
```

LIST OF REFERENCES

1. Knowlton, Kenneth C., "A Programmer's Description of L^6," Communications of the ACM, v. 9, no. 8, p.616-625, August 1966.

2. Morriss, Robert, "Scatter Storage Techniques," Communications of the ACM, v. 11, no. 1, p.38-43, January 1968.

3. Naur, Peter et al., "Revised Report on the Algorithmic Language, ALGOL 60," Communications of the ACM, v. 6, no. 1, p. 1-17, January 1963.

4. Newell, A. and Tonge, F. M., "An Introduction to Information Processing Language-V,", Communications of the ACM, v.3, no. 4, p.205-211, April 1960.

5. McCarthy, J., "Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I," Communications of the ACM, v.3, no.4, p.184-195, April 1960.

6. McCarthy, J. et al., LISP 1.5 Programmer's Manual, M.I.T. Computation Center and Research Lab. of Electronics, Cambridge, Mass., 1967.

7. Farber, D. J., Griswold, R. E., and Polonsky, I. P., "SNOBOL, A String Manipulation Language," Journal of the ACM, v. 11, no. 1, p.21-30, January 1964.

8. COMIT Programmer's Reference Manual, M.I.T. Research Lab. of Electronics and the Computation Center, Cambridge, Mass., November 1961.

9. Newell, A., Earley, J., and Haney, F., *1 Manual, Carnegie-Melton University, Department of Computer Science, 1967.

10. Evans, D., and Van Dam, A., "Data Structures Programming System," Information Processing 68, p.557-564, 1969.

11. Sammet, Jean E., Programming Languages: History and Fundamentals, p.400, Prentice-Hall, 1969.

12. Housden, R. J. W., "The Definition and Implementation of LSIX in BCL," Computer Journal, v. 12, no. 1, p.15-23, February 1969.

13. Knuth, Donald E., The Art of Computer Programming, Vol. I: Fundamental Algorithms, Addison-Wesley, 1968.

14. Hopcroft, John E. and Ullman, Jeffrey D., Formal Language and Their Relation to Automata, Addison-Wesley, 1969.

15. Floyd, R. W., "Syntactic Analysis and Operator Precedence," Journal of the ACM, v. 10, no. 3, p.316-333, 1963.

16. Wirth, Niklaus and Weber, Helmut, "EULER: A Generalization of ALGOL and its Formal Definition: Part I," Communications of the ACM, v. 9, no. 1, p.13-25, January 1966.

17. Knowlton, Kenneth C., "A Fast Storage Allocator," Communications of the ACM, v. 8, no. 10, p.623-625, October 1965.

INITIAL DISTRIBUTION LIST

		No. Copies
1.	Defense Documentation Center Cameron Station Alexandria, Virginia 22314	2
2.	Library, Code 0212 Naval Postgraduate School Monterey, California 93940	2
3.	Gary A. Kildall, Code 53Kd Department of Mathematics Naval Postgraduate School Monterey, California 93940	7
4.	LCDR C. Scott Thorell, USN Naval Ordnance Systems Support Office Pacific 4297 Pacific Highway P. O. Box 10103 San Diego, California 92110	1
5.	LT William O. Poteat, Jr., USN 502 East Tenth Avenue Johnson City, Tennessee 37601	1
6.	Assoc. Professor . B. Cowie, Code 62Ce Department of Bu . Admin. & Economics Naval Postgraduate School Monterey, California 93940	1

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)

ORIGINATING ACTIVITY (Corporate author) Naval Postgraduate School Monterey, California 93940	2a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED 2b. GROUP

3. REPORT TITLE

A Basic List-Oriented Information Structures System (BLISS)

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

Master's Thesis; June 1970

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

Charles Scott Thorell and William Otto Poteat, Jr.

6. REPORT DATE	7a. TOTAL NO. OF PAGES	7b. NO. OF REFS
June 1970	177	17

8a. CONTRACT OR GRANT NO.	9a. ORIGINATOR'S REPORT NUMBER(S)
b. PROJECT NO.	
c.	9b. OTHER REPORT NO(S) (Any other numbers that may be assigned this report)
d.	

10. DISTRIBUTION STATEMENT

This document has been approved for public release and sale; its distribution is unlimited.

11. SUPPLEMENTARY NOTES	12. SPONSORING MILITARY ACTIVITY
	Naval Postgraduate School Monterey, California 93940

13. ABSTRACT

The design and implementation of the Basic List-Oriented Information Structures System is described. Manipulation of list structures in an efficient and cogent manner is the system function. The language, which is patterned after Bell Telephone Laboratories' L^6, is generated from a precedence grammar for rapid syntax analysis. A compiler produces code for a pseudo-machine that is designed to effectively carry out list-oriented functions. Dynamic storage allocation and structure definition are significant execution-time features. The implementation, written in PL/I, is for operation under the CP/CMS time-sharing system on the IBM 360/67

DD FORM 1 NOV 65 1473 (PAGE 1)

S/N 0101-807-6811

Security Classification

A-31408

KEY WORDS	LINK A		LINK B		LINK C	
	ROLE	WT	ROLE	WT	ROLE	WT
ɔiler						
Processing						
rmation Retrieval						
rpreter						
ramming Language						

Thesis
T456
c.1

121544

Thorell
A Basic List-oriented Information Structures System (Bliss).