A FORTRAN CROSS COMPILER FOR A TMS 9900 MICROCOMPUTER SYSTEM

By

STEVEN ROGER HEARD

Bachelor of Science

Oklahoma State University

Stillwater, Oklahoma

1977

Submitted to the Faculty of the
Graduate College of the
Oklahoma State University
in partial fulfillment of the
requirements for the Degree of
MASTER OF SCIENCE
July, 1979

A FORTRAN CROSS COMPILER FOR A TMS

9900 MICROCOMPUTER SYSTEM

Thesis Approved:

J. E. Hedrick
Thesis Adviser

J P Chandler

J. K. Phillips

Norman N Durham
Dean of Graduate College

ACKNOWLEDGMENT

I wish to express my sincere appreciation to Dr. George E. Hedrick for spending many hours in valuable guidance on this thesis research. I also wish to thank the other members of my committee, Dr. John R. Phillips and Dr. John P. Chandler.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER I

INTRODUCTION

Background

Cross compilers can be a valuable aid in the development of software for typical microcomputer applications. Microcomputers generally lack sufficient storage to support a high-level language processor, yet it is often desirable to write programs for process control, small business, automation, and home computer purposes in a language less cumbersome than assembler or machine code (18) (8). Programs usually require less time and money to develop and are easier to maintain if they are written in a high-level language. Through the use of a FORTRAN cross compiler it is possible for a programmer to make use of existing FORTRAN subroutines and to test for proper execution of programs before transferring the object code from the host computer to the object computer. A cross compiler executes on one machine and produces object code for another machine. The host computer is the machine on which the cross compiler actually runs. The object computer is the machine for which machine language (object code) is generated from the source code.

The hardware of the host and object computers need only be compatible to a point that will permit data transfer.

Methods used for transferring compiled programs include read only memory (ROM) chips, direct communication links, automatic send and receive (ASR) terminals, paper tape, flexible disks (floppies) and magnetic tape. A diagram of a system using a direct communication link is in Figure 1.

A cross compiler that is written in a commonly used language such as FORTRAN or BASIC can be portable from one host computer to another. If the parsing of source code and the generation of object code are divided into separate steps, it is possible to modify the cross compiler for use with different object computers. This may be accomplished by altering the segments of object code (templates) that are stored for use in code generation.

Objectives and Motivation

The primary objective of the compiler described in this thesis is to provide a means of implementing programs written in a subset of Basic Standard FORTRAN on a TMS 9900 microcomputer through the use of an IBM 370. Existing programs written in FORTRAN or in other languages may be implemented after hand translation to the supported FORTRAN subset. A secondary objective is to provide parsing and code generation software that can be adapted to other machines. The compiler may be modified to generate code for a different object computer at a fraction of the time and cost involved in developing a new compiler.

In order to take advantage of existing software and pro-

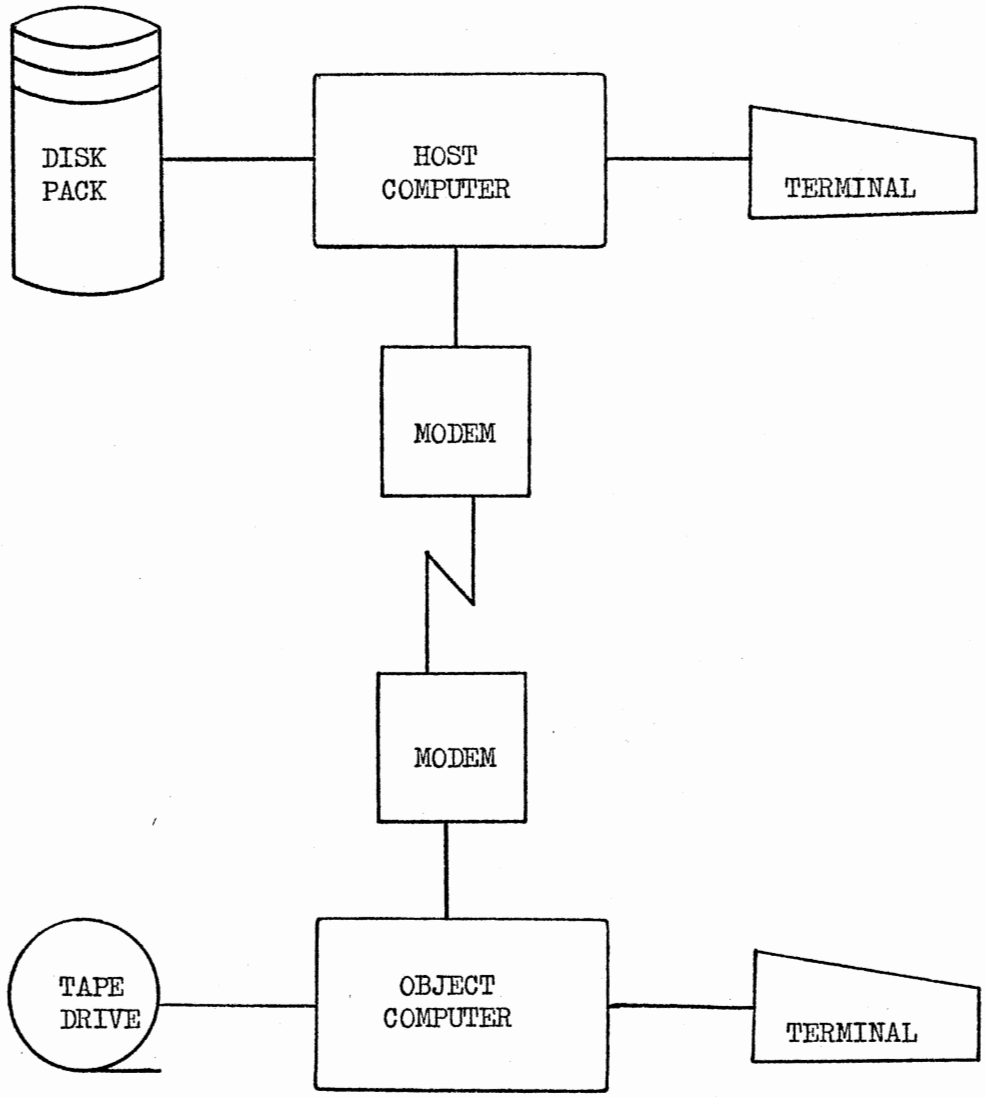

Figure 1. Diagram of System Using Direct Communication Link

gramming skills, it is desirable for the compiler to support a commonly used language. BASIC, FORTRAN and COBOL are all common programming languages. BASIC interpreters exist for most common microprocessors. This reduces the practicality of a BASIC cross compiler. COBOL is more suitable for file processing on large machines than for typical microcomputer applications (18). The major drawback to FORTRAN is the time and cost involved in developing a compiler to support the full language. For these reasons, a FORTRAN subset was selected to be supported by the compiler.

The choice of the TMS 9900 microprocessor was influenced by the lack of existing software for that particular machine. To maximize the practicality of the compiler, the object computer should be one for which a FORTRAN cross compiler is not available. Since such software is offered for most popular microcomputer systems, a logical second choice is a microcomputer for which the cost of this support is beyond the resources of some users. A FORTRAN cross compiler is available from Texas Instruments for the TMS 9900 at a cost of over two thousand dollars.

The IBM 370 is the primary machine for general use at Oklahoma State University with facilities for data transfer via telephone lines. This is a useful feature for transferring object code since it eliminates the need for actually transporting some medium such as cards or tape.

Literature Review

A FORTRAN subset as a language to be supported by the compiler has several advantages. Sammet (18) presents several of the advantages and disadvantages of FORTRAN as a programming language along with a brief history of FORTRAN. Higman (8) discusses the evolution of programming languages and makes comparisons of several modern languages including FORTRAN. Barron (3) explores several of the general principles involved in the design of programming languages and gives a number of FORTRAN examples.

The Backus-Naur form (BNF) grammar used in Appendix D was originally defined by Naur (14) and is discussed by Gries (6) and by Aho and Ullman (1). Several other grammars used for the definition of programming languages are presented by Weingarten (20). Ghandour (5) examines the use of canonic systems for the recursive definition of programming languages.

A concise presentation of many of the aspects involved in the design of compilers is given by Hopgood (9). Lee (11) provides a more detailed treatment of these areas. Halstead (7) discusses implementation considerations for compilers as well as design considerations.

To add to its portability, the compiler is written in FORTRAN. Cocke (4) suggests that the portability and ease of implementation of a compiler may be greatly increased if it is written in the language it supports and allowed to compile itself.

CHAPTER II

IMPLEMENTATION CONSIDERATIONS

The compiler has been constructed so that it is reasonably portable. Certain modifications may be required prior to implementation because of differences in software and hardware between systems. Reasons for such changes include the version of FORTRAN supported by the host computer, communication facilities between the host and object computers, and software for loading code onto the object computer. This chapter outlines the considerations made for the current implementation on an IBM 370 using a TMS 9900 as an object computer.

Requirements Imposed by FORTRAN

With the exception of the use of DATA statements in subroutines, the compiler is written in ANSI Basic Standard FORTRAN (2). It has been successfully tested with the FORTRAN G, FORTRAN H and WATFIV compilers. Excluding those variables whose values are set in DATA statements, subroutines and functions do not depend on the values of internal variables being preserved between calls.

Since Basic Standard FORTRAN is unable to detect the end of an input file, a delimiter record is used at the end of

the source code to be compiled (2). Because no FORTRAN statements can contain an asterisk in column 1, this record and all other control statements accepted by the compiler contain an asterisk in column 1. A complete description of all control statements is in the user's guide (Chapter V).

Requirements Imposed by the IBM 370/158

The internal code of the object computer is assumed to be ASCII. Since the internal code of the IBM 370 is EBCDIC, the compiler contains an EBCDIC to ASCII conversion routine for use with character strings encountered in formats and DATA statements. If implemented on an ASCII host computer, all such conversions should be eliminated.

The IBM 370 has a word length of 32 bits. This permits the addresses associated with generated object code to include the range from 0000 to FFFF (hexadecimal) since they can all be represented as positive numbers. If implemented on a 16-bit host computer, the range of addresses would have to be restricted to 0000 to 7FFF (hexadecimal). This is because a 16-bit computer uses the first bit of a word to designate the sign of an integer value.

Certain datasets or devices must be allocated to certain FORTRAN unit numbers for the compiler to operate. File allocations are as follows:

UNIT NUMBER	DESCRIPTION
5	Input dataset or device for FORTRAN source code.

6	Output dataset or device for source listings and diagnostics.
8	Dataset for intermediate code.
9	Dataset for preliminary object code.
10	Output dataset or device for processed object code.

Requirements Imposed by the TMS 9900

The primary requirement imposed on the compiler by the TMS 9900 is the compatibility of the generated object code with the TMS 9900 instruction set. The TMS 9900 is a 16-bit microcomputer featuring sixteen general purpose registers, hardware multiply and divide, and memory to memory arithmetic. A full description of the instruction set is available from Texas Instruments in the 990 Systems Handbook (19). The word length of sixteen bits permits the compiler to allocate exactly one word of storage for every integer variable. Because of the sixteen general purpose registers, the code generated by the compiler does not need to use much storage in memory for intermediate values. Hardware multiply and divide contribute to the efficiency of the algorithms used by the compiler for exponentiation, multiplication, and division.

The format of the object code is another requirement imposed by the TMS 9900. Each line of object code begins with a four-digit hexadecimal address followed by a colon. The remainder of the line contains two-digit hexadecimal

bytes of data separated by spaces. An example of the object code produced by the compiler is provided at the end of Appendix C.

CHAPTER III

LANGUAGE DESCRIPTION

The cross compiler supports a subset of Basic Standard FORTRAN (2). A Backus-Naur form (BNF) syntax description of the supported language is presented in Appendix D (1).

The compiler is capable of processing two types of FORTRAN routines. These are subroutines and main programs. Subroutines must start with a SUBROUTINE statement. Main programs, by definition, cannot contain a RETURN statement. Each routine is terminated with an END statement. DIMENSION, COMMON and DATA statements must precede any executable statements and appear in the order: DIMENSION, COMMON, DATA. The source code for each subroutine must precede any routines that invoke that subroutine. This is because the compiler requires the absolute address of a subroutine in order to generate the object code for a CALL statement.

The compiler accepts FORTRAN source routines from a file having a logical record length of eighty bytes. Each record of input is divided into several fields. If a 'C' appears in the first column, the compiler will not attempt to process the record as a FORTRAN statement. Columns one through five of a FORTRAN statement may either be blank or contain a

statement number. Statement numbers are optional for executable statements, required for formats and prohibited on all other non-executable statements. Duplicate statement numbers within any one subroutine or main program are not permitted. Column six, if not blank, designates a continuation of statement text from the previous record. Columns seven through seventy-two are reserved for the text of a FORTRAN statement. The BNF syntax in Appendix D is applicable to this field. Columns seventy-three through eighty are for record identification only and are ignored by the compiler.

FORTRAN statements may be divided into two categories, executable statements and non-executable statements. Executable statements cause the compiler to generate executable machine code. The READ, WRITE, assignment, GO TO, IF, DO, STOP, RETURN, CALL, and CONTINUE statements are all of the executable statements supported. Non-executable statements are used to define storage areas or to communicate other information about the program to the compiler. The SUBROUTINE, DIMENSION, COMMON, FORMAT and DATA statements are the supported non-executable statements. The REAL, INTEGER, EQUIVALENCE, FUNCTION, PAUSE, REWIND, BACKSPACE, ENDFILE and computed GO TO statements are not supported.

The only type of variable supported by the compiler is sixteen bit integer. The allowable range of values for both variables and constants is from -32768 to 32767. The seven basic external functions (EXP, ALOG, SIN, COS, TANH, SQRT

and ATAN) are not supported since the arguments of these functions must be real and no support is given to real variables.

The implementation of the STOP, END, RETURN, CONTINUE, DO, assignment, SUBROUTINE, CALL, and IF statements is standard. Certain restrictions exist for the use of the READ, WRITE, FORMAT, DIMENSION, DATA, and COMMON statements.

Arrays are restricted to a single dimension. No dimensioning of arrays is permitted in a COMMON statement. If an array is in common, the dimensions must be declared in a DIMENSION statement. Named common blocks are not permitted. Implied repetition of items within a DATA statement is not supported. For example:

```
      DIMENSION LINE(10)
      DATA LINE /10*'  '/
```

is not valid and should be coded as

```
      DIMENSION LINE(10)
      DATA LINE /'  ','  ','  ','  ','  ','  ','  ','  ',
     *           '  ','  '                                /
```

to be accepted by the compiler.

Implied do loops are not supported for use with READ or WRITE statements. Unformatted input or output is not allowed. Since real variables are not permitted, the E and F formats should not be used. Group repetition within a format is not supported (2). For example:

```
     60 FORMAT(2HX=,2(I2,','),I2)
```

should be coded as

```
     60 FORMAT(2HX=,I2,',',I2,',',I2)
```

Three extensions of Basic Standard FORTRAN are included

in the implementation. Quote marks may be used in FORMAT statements and DATA statements to define character strings. Variable names and subroutine names may contain six characters instead of five. In addition to the standard I, X, H and A formats, a U format field descriptor may be used within a FORMAT statement. When used with a WRITE statement, the U format functions as an I10 format. When used with a READ statement the U format permits integers to be input without having to be aligned with any specific columns of the input record. This is a useful feature when numbers must be entered manually from a terminal.

In summation, the language supported by the compiler is very close to Basic Standard FORTRAN. The major differences are that the supported language does not include floating point arithmetic or multidimensional arrays. The other differences detailed in this chapter are relatively minor.

CHAPTER IV

FEATURES OF THE IMPLEMENTATION

General Capabilities

The compiler possesses several other capabilities in addition to its capacity to compile FORTRAN programs. A FORTRAN program may reference subroutines that are not included in the source code to be compiled. Via the control language, a user may provide the compiler with subroutine names and associated absolute addresses to be used for subroutine entry. Hexadecimal machine code may be provided as input to the compiler for the purpose of being included in the object code output. This permits a user to interface a FORTRAN program with previously compiled and/or assembled routines. More extensive information on the use of the control language is in the user's guide (Chapter V).

For every routine compiled, the FORTRAN source code is listed and the execution address is printed. If syntax errors are present in a statement, an error message is included in the listing following the statement in which the error was detected. If errors such as invalid nesting of loops or references to non-existent statement numbers are present in a routine, an error message follows the listing of the routine in which the error was detected.

The compiler contains a preset address where the object code starts, but this address may be altered by the user through the control language. If a user wishes to make use of previously compiled or assembled routines, it may be necessary to modify this starting address to avoid conflicts for memory space.

Major Algorithms Used

The compiler consists of a control language processor and three separate steps of compilation, all of which are invoked from the main program. The main program (flowchart in Figure 2) establishes the input/output unit numbers and reads the first record from the input file. The control language processor is called if a control record is encountered. Any record containing an asterisk in column 1 is assumed to be a control record. The first pass of compilation is invoked for every FORTRAN routine to be compiled. The next two passes are called only if no errors are encountered during the first pass. Details concerning parsing, generation of code and manipulation of tables are presented in the programmer's guide (Chapter VI).

The control language processor is used to identify the various types of control statements and perform the associated control functions. If the control statement cannot be recognized an error message is printed and execution of the compiler is terminated. Functions performed by the control language processor include altering the initial machine code

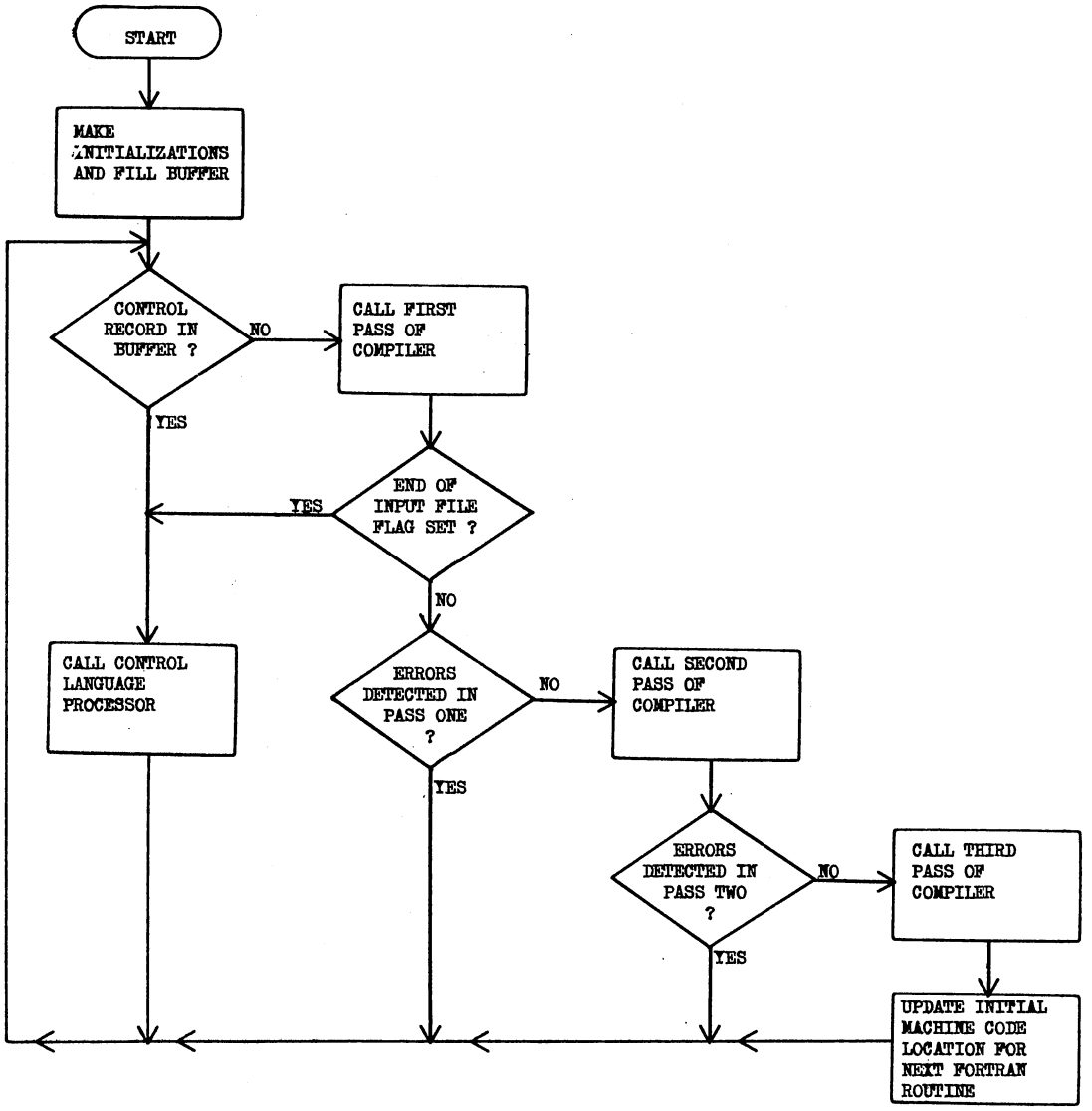

Figure 2. Flowchart of Main Program

address, adding a subroutine name and address to the subroutine table, invoking the routine to transfer hexadecimal machine code from the input file to the object code file and terminating execution. A complete description of the control language and control functions may be found in the user's guide (Chapter V).

The major functions of the first pass of the compiler (flowchart in Figure 3) are to parse the FORTRAN source code, to generate intermediate code, and to detect syntax errors. The intermediate code is a numerical representation of the FORTRAN source code to be used by the second pass. When invoked, the first pass rewinds (closes) temporary files and initializes tables used for storing variable names, information contained in formats and values established by DATA statements. For every FORTRAN statement, the routines to input, to print, and to parse the statement, and to generate intermediate code from the statement are called. Statements are checked for proper sequence as outlined in the language description (Chapter III). When either an END statement or a control statement is encountered, then the intermediate code is delimited; a check is made for undefined variables; the table containing values established in DATA statements is sorted into ascending addresses; and control is returned to the main program.

A table driven parser is used to generate intermediate code for all types of nontrivial executable statements that can be parsed by a finite state automaton without pushdown

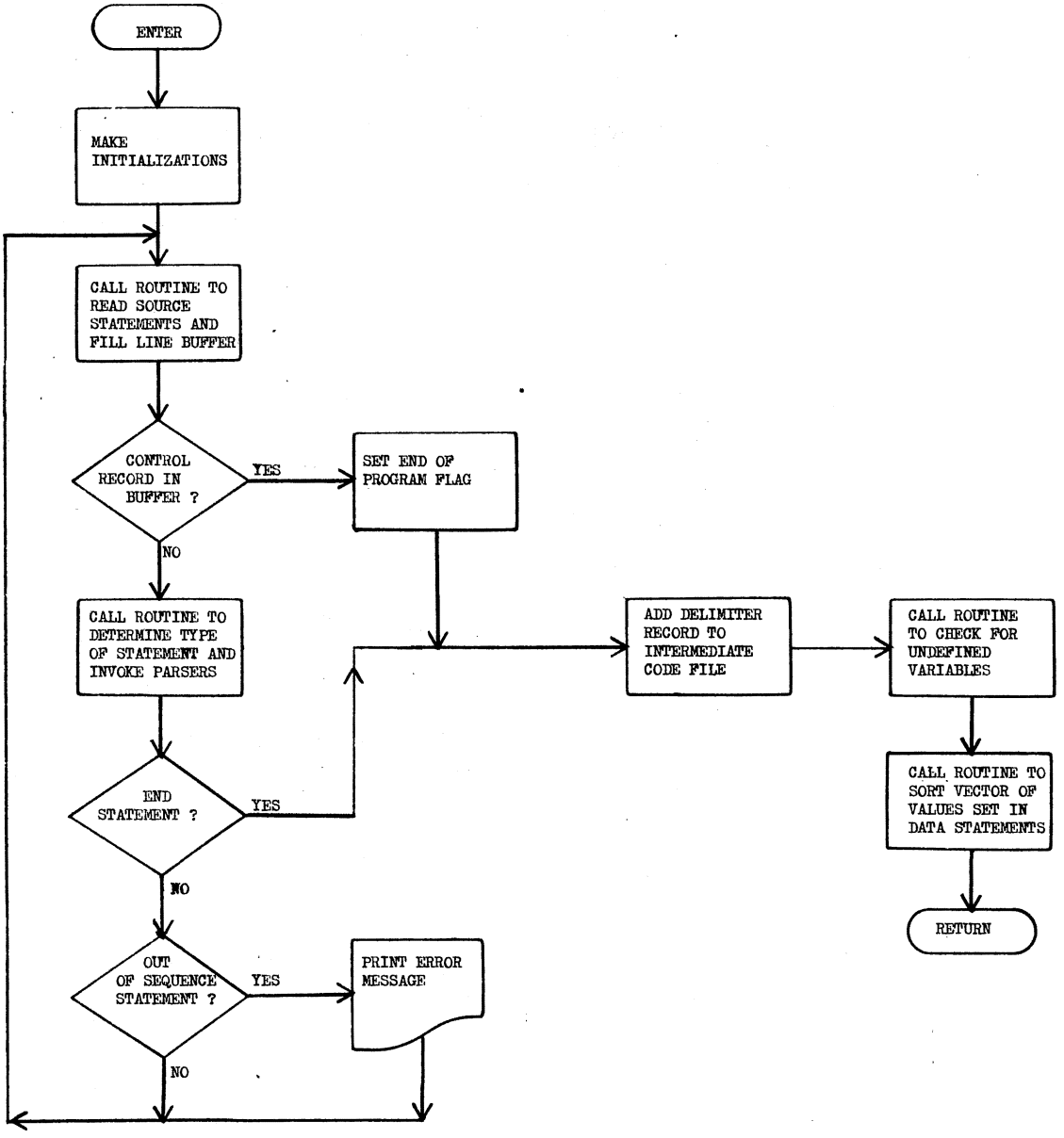

Figure 3. Flowchart of PASS1

lists (1). These statement types are READ, WRITE, GO TO, DO, SUBROUTINE and CALL. Each line in the transition table corresponds to a state of a finite state automaton and contains information regarding the next state to be entered for every type of token encountered. A diagram of the finite state automaton used to parse DO statements is in Figure 4. Information governing intermediate code generation is also contained within each line of the table.

Embedded logic parsers are used to generate intermediate code for trivial statements and statements containing arithmetic expressions. An embedded logic parser is a parser whose operation is not governed by a table, but rather has its logic embedded in the source code comprising the parser. Only the statement number and statement type need to be recorded in the intermediate code for the STOP, RETURN and CONTINUE statements. Assignment statements are recorded in intermediate code in reverse Polish notation and must be parsed using a pushdown list (1) (10). Arithmetic IF statements are divided into an arithmetic expression and an IF statement using a simple variable as an argument. The arithmetic expression is parsed in the same manner as any assignment statement. The table driven parser parses the IF statement after any arithmetic expressions have been removed. FORMAT, DIMENSION, COMMON, and DATA statements do not cause any intermediate code to be generated since they are not executable. Embedded logic parsers exist to store the information contained within these statements into

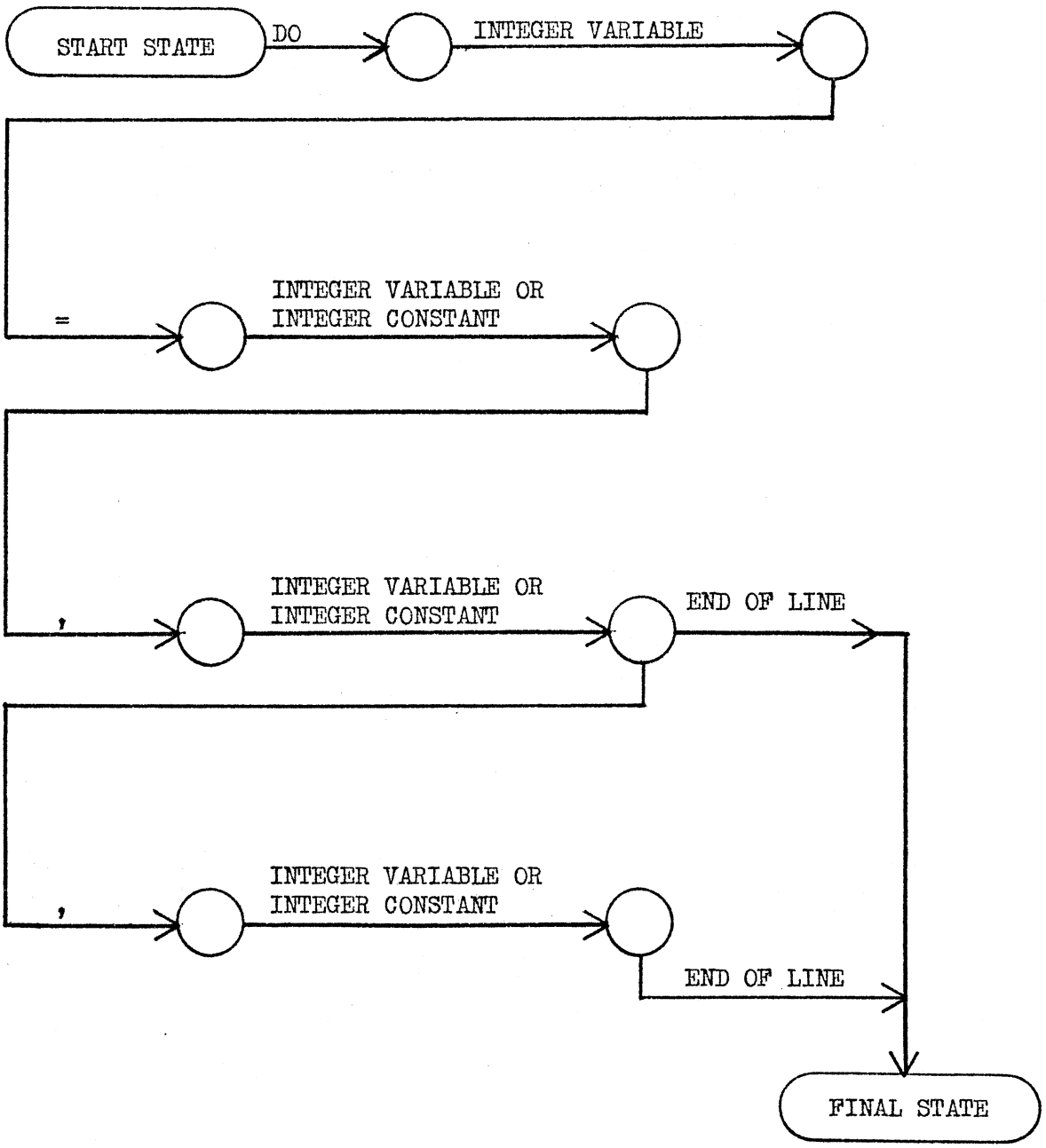

Figure 4. Diagram of Finite State Automaton
Used to Parse DO Statements

tables for use in the second pass of compilation.

There are three lexical analyzers: LEX1 is a table driven lexical analyzer and is capable of recognizing eleven types of tokens found in the FORTRAN source code: LEX2 utilizes LEX1 to recognize eleven types of tokens found within formats: LEX3 utilizes LEX1 to recognize ten types of tokens found within assignment statements. Detailed information regarding the types of tokens recognized by each of these routines may be found in Appendix A.

The second pass of the compiler (flowchart in Figure 5) performs two major functions. The first of these is to generate machine code from the information stored in the tables and the intermediate code. The second is to build a table of statement numbers with corresponding locations, and to build a list of the statement numbers in the order of reference for use by the final pass of the compiler.

Machine code is first generated for all variables and arrays. All values are zero unless defined otherwise in a DATA statement. Code is next generated for character strings found in formats. A separate routine exists for generating object code from intermediate code for every type of executable statement. These routines contain templates of object code associated with the particular type of statement. The code generated by these routines contains no absolute addresses and is therefore not executable. Instead of four hexadecimal digits of an absolute address, four X's are generated and the statement number associated with the

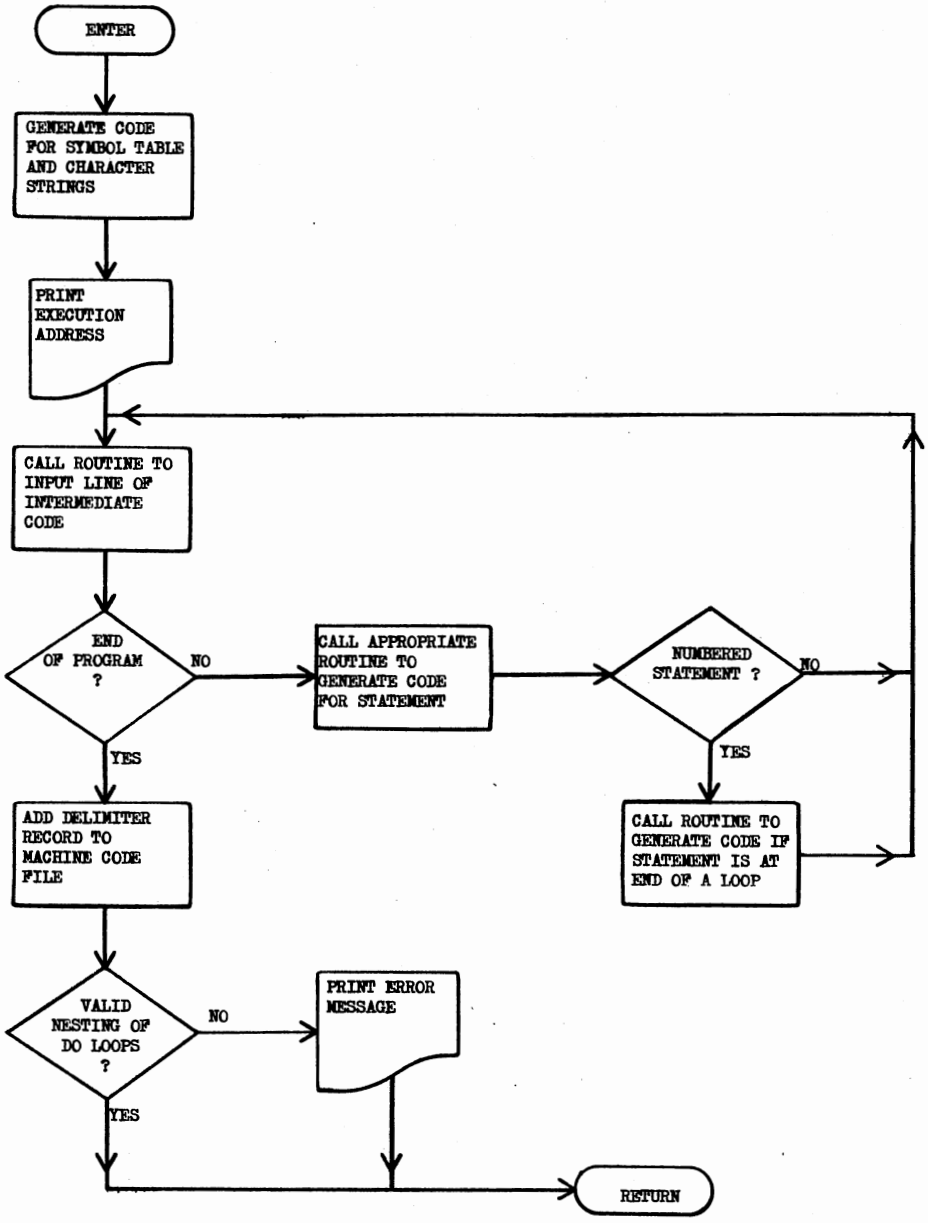

Figure 5. Flowchart of PASS2

address is stored for use by the third pass. After all intermediate code is processed in pass two, a check is made for valid nesting of loops and control is returned to the main program.

The third pass of compilation (flowchart in Figure 6) fills in the absolute addresses in the object code. This is done by the use of the table of statement numbers and associated addresses, and by the use of the list of statement numbers in the order they were referenced. The object code is scanned for a sequence of four X's which were used in the earlier pass to mark the places where addresses are needed. At every such instance, the X's are replaced by four hexadecimal digits representing an address associated with a statement number and a pointer is advanced to the next statement number in the list. At the end of this pass, the object code for the FORTRAN routine parsed by the first pass is complete and control is returned to the main program of the compiler.

In summation, the FORTRAN source code for each routine is compiled in three passes. The first pass detects syntax errors and generates intermediate code. The second pass generates object code from the intermediate code. The third pass completes the generation of the object code by filling in absolute addresses.

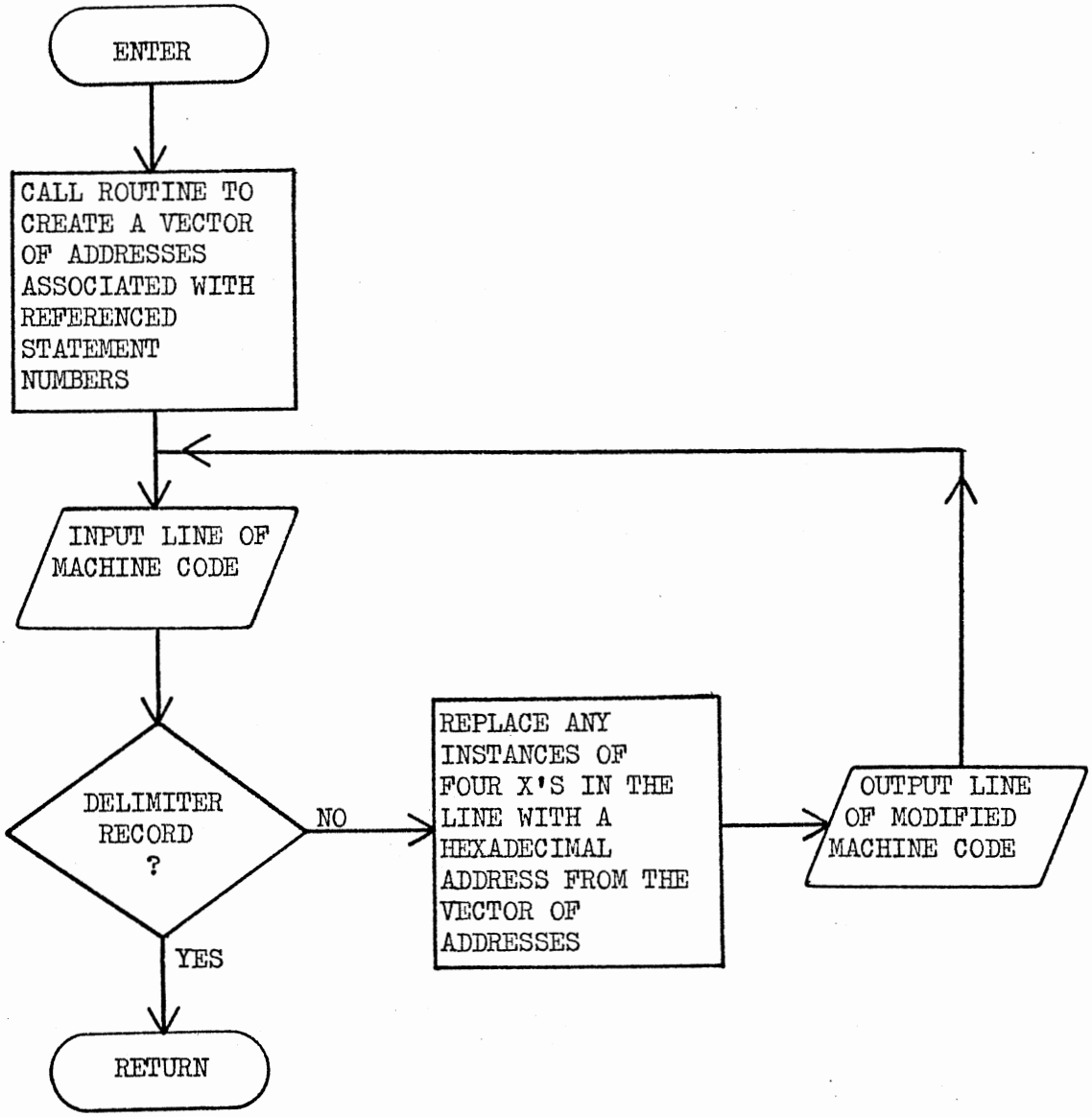

Figure 6. Flowchart of PASS3

TMS 9900 Assembler Support Routines

The object code generated by the compiler is dependent on routines written in assembly language for input, output and certain arithmetic functions. These routines may be loaded into the TMS 9900 with the object code generated from FORTRAN or they can reside in read only memory (ROM) in the object computer. A complete listing of these routines may be found in Appendix B.

Arithmetic function routines are used to perform operations not fully supported by the TMS 9900 hardware. These include subscripting and exponentiation. Multiplication and division are supported by the hardware, but only for positive integer operands. Assembler routines exist to multiply and divide signed numbers.

Input and output are accomplished through the use of a single buffer capable of holding a single line of data. The entire buffer may be input from or output to any of the first sixteen input/output ports (19). The port selected corresponds to the unit number specified in the READ or WRITE statement. Routines exist for the transfer of data between the buffer and memory for all the supported formats. There is a special routine for moving character strings specified within formats from memory to the buffer.

TABLE I

FUNCTIONS OF ASSEMBLY LANGUAGE
SUPPORT ROUTINES

Routine Name	Function
LNIN	Input line of data.
LNOUT	Output line of data.
BLANK	Set I/O buffer to blank.
SKIP	Skip spaces for X format.
INA1	Input character for A1 format.
INA2	Input characters for A2 format.
INI	Input number for I format.
INU	Input number for U format.
OUTA1	Output character for A1 format.
OUTA2	Output characters for A2 format.
OUTI	Output number for I format.
STRIN	Output a character string.
EXPN	Perform exponentiation.
DIVD	Perform signed division.
MULT	Perform signed multiplication.
SBSC	Perform subscripting.

CHAPTER V

USER'S GUIDE

As implemented on the IBM 370/158 at Oklahoma State University, the compiler may be run in either a batch mode or a TSO (time sharing) mode. In both modes, the user's source code is processed by the compiler to produce a source listing and object code suitable for loading onto a TMS 9900 microcomputer system. Information required by the user for running the compiler is presented in this chapter. It is assumed that the user has some familiarity with the protocol involved in running programs on the IBM 370/158. The methods employed for loading the object code onto the user's microcomputer system and executing the code once it is loaded are beyond the scope of this thesis.

Compiler Control Language

The control language allows the user to select the starting location of the generated object code and to compile FORTRAN programs that reference subroutines which are not included in the FORTRAN source code to be compiled. The compiler can be provided with subroutine names and associated absolute addresses to be used for subroutine entry. Hexadecimal machine code can be input for the purpose of

being included in the generated object code. These features of the control language permit the user to interface a FORTRAN program with previously compiled or assembled routines.

Control records should be included in the same file as the FORTRAN source code to be compiled. All control records have an asterisk in the first column. Since no FORTRAN statement can contain an asterisk in column one, the compiler will attempt to process any record starting with an asterisk as a control statement. If a control statement cannot be recognized, an error message is printed and execution of the compiler is terminated. Syntax and semantics of the control statements are as follows:

*SUB nnnnnn hhhh

The compiler adds the subroutine name nnnnnn and the hexadecimal location hhhh to the list of subroutine names and matching execution addresses. This statement may be used to allow the invocation of subroutines in read only memory or subroutines loaded as hex decks.

*HEX

A hex deck is to be read in and added to the dataset of generated object code. The hex deck must follow the *HEX control statement and must conform to the format used for object code generated by the compiler.

*LOC hhhh

The compiler will use the hexadecimal location hhhh as a starting location for generated object code instead of the default location.

*FOR

FORTRAN source code is to follow.

*END

End of input file.

Example of the use of the control language:

```
*SUB ASMSUB 0200 *HEX  0200:  02 2B 00 00 C8 08  01 0E C0 20
020A: 01 84 06 A0 F2 E8 01 86 C8 00
            .
            .
            .  02D0: 01 0E 04 5B *LOC 0300 *FOR
      SUBROUTINE FTSUB
            .
            .
            .
      END
      DIMENSION N(32)
      DATA IX/'X'/
            .
            .
            .
      CALL ASMSUB(IP1,IP2,N)
      CALL FTSUB
      STOP
      END *END
```

In the above example the subroutine ASMSUB has been assembled independently and a hex deck has been produced. The entry point for ASMSUB is 200 hexadecimal.

Output

For every routine compiled, the FORTRAN source code is listed and the execution address is printed. If syntax errors are present in a statement, an error message is included in the listing following the statement in which the error was detected. If errors such as invalid nesting of loops or references to non-existent statement numbers are present in a routine, an error message is included in the listing following the routine in which the error was detected.

Object code is generated for every FORTRAN routine in which no syntax errors are found. Each line of object code

begins with a 4-digit hexadecimal address followed by a colon. The remainder of the line contains 2-digit hexadecimal bytes of data separated by spaces.

TSO Mode

The sample run in Appendix C was produced through the use of the CLIST (command procedure) in Figure 7. The name of the dataset containing the source code and the name of the dataset to receive the object code should be entered by the user. If these dataset names are not entered when the CLIST is executed, the system will prompt the user for these names. In the sample run RAND.FORT is the source code dataset and LOADDECK.DATA is the object code dataset. COMP.DUMMY1.DATA and COMP.DUMMY2.DATA are the names associated with the datasets used by the compiler for temporary storage. These two datasets are created before the compiler is invoked and are destroyed afterward. U16529A.COMP.LOAD(TMS9900) is the name of the load module containing the compiler. A listing of the FORTRAN source and the execution address associated with each routine is printed at the user's terminal.

Batch Mode

An example of JCL (job control language) that may be used to run the compiler is in Figure 8. When submitted, it produces a source listing on paper and punches an object deck on cards.

```
PROC 2 SOURCEDS,OBJDS
ATTRIB INTATTR BLKSIZE(256) DSORG(PS) LRECL(256) RECFM(V B S)
ATTRIB OBJATTR BLKSIZE(400) DSORG(PS) LRECL(80) RECFM(F B)
FREE F(FT06F001 FT05F001)
ALLOC F(FT05F001) DS(&SOURCEDS) SHR
ALLOC DS(*) F(FT06F001) SHR
ALLOC DS(COMP.DUMMY1.DATA) F(FT08F001) NEW SPACE(5,2) TRACKS -
      BLOCK(256) USING(INTATTR) DELETE
ALLOC DS(COMP.DUMMY2.DATA) F(FT09F001) NEW SPACE(5,2) TRACKS -
      BLOCK(400) USING(OBJATTR) DELETE
ALLOC DS(&OBJDS) F(FT10F001) SHR
CALL 'U16529A.COMP.LOAD(TMS9900)'
FREE DS(COMP.DUMMY1.DATA)
FREE DS(COMP.DUMMY2.DATA)
FREE F(FT10F001)
FREE ATTRLIST(INTATTR)
FREE ATTRLIST(OBJATTR)
```

Figure 7. CLIST Used to Run the Compiler

```
//U16529A  JOB (16529,237-88-5308),'HEARD',CLASS=F,TIME=(0,15),
// MSGCLASS=A,NOTIFY=U16529A
/*PASSWORD XXXX
/*ROUTE PRINT RMT1
// EXEC PGM=TMS9900,REGION=256K
//STEPLIB   DD DSN=U16529A.COMP.LOAD,DISP=SHR
//FT05F001  DD DDNAME=SYSIN
//FT06F001  DD SYSOUT=A
//FT08F001  DD UNIT=SYSDA,SPACE=(TRK,(5,2)),
//             DCB=(RECFM=VBS,BLKSIZE=256,LRECL=256)
//FT09F001  DD UNIT=SYSDA,SPACE=(TRK,(5,2)),
//             DCB=(RECFM=FB,BLKSIZE=400,LRECL=80)
//FT10F001  DD SYSOUT=B
//GO.SYSIN  DD *
*FOR
      DIMENSION I1(10)
      DATA I1/0,1,2,3,4,5,6,7,8,9/ J1,J2/'I=','J'/
      DO 100 I=1,10
        WRITE(1,1) J1,I,I1(I)
    1   FORMAT(1X,A2,I2,' I1(I)=',I3)
  100   CONTINUE
      WRITE(1,2) J2
    2 FORMAT(3HJ2=,A1)
      STOP
      END
*END
//
```

Figure 8. JCL Used to Run the Compiler

CHAPTER VI

PROGRAMMER'S GUIDE

This chapter is intended for use by a person wishing to make modifications to the cross compiler or by a person who would like to make use of some of the compiler's subroutines for another application. It is recommended that the reader of this chapter have a source code listing* of the compiler available for reference since the purpose of the Programmer's Guide is to supplement the internal documentation of the compiler and not to provide a complete description of the workings of the compiler. Information presented in this chapter includes descriptions of data structures, generation of intermediate and final code, and the logic of table driven subroutines. Information concerning the functions and parameters of each routine is in Appendix A. A listing of the assembly language support routines required by the compiler is in Appendix II.

Data Structures

All of the major data structures are in common storage.

---

*Such a listing is on a magnetic tape file in the Department of Computing and Information Sciences at Oklahoma State University.

Data structures include line buffers, a symbol table, a table for storing strings encountered in formats and variable values established in DATA statements, a table of subroutine names and addresses and a table for storing information encountered in FORMAT statements. All data structures have set capacities believed to be adequate for compiling most FORTRAN programs intended for microcomputer purposes.

LBUF is the buffer for a single line of FORTRAN source code input. It is capable of holding 72 characters. Since any information in columns 73 through 80 of the FORTRAN source code is intended to be ignored by the compiler, only the first 72 characters of each line are read.

LINE is the buffer for a single FORTRAN statement and is also used as an output buffer for hexadecimal object code. It is capable of holding up to 402 characters. This allows a FORTRAN statement with up to five continuations to be stored. LNLEN is the integer variable indicating the number of characters stored in LINE.

INTL is the buffer for a line of intermediate code. It is capable of holding up to fifty integer values. It is used in the first pass of the compiler as an output buffer and in the second pass as an input buffer.

ISTBL is the table capable of holding up to 50 subroutine names and addresses. NSTBL is the integer variable indicating the number of subroutines represented in ISTBL. The name of the I'th subroutine is stored in

ISTBL(1,I) through ISTBL(6,I). ISTBL(7,I) contains an integer value for the address of the entry point for the I'th subroutine.

ISYMB is a table that is used for storing information associated with variables and constants during pass one and for storing information associated with DO loops during pass two. This table has a capacity of fifty entries. This limits the level of nesting of loops to fifty and the number of variable names and constants in a single routine to fifty. In the first pass the alphanumeric representation of the I'th variable name or constant is stored in ISYMB(1,I) through ISYMB(6,I). ISYMB(7,I) contains the integer location of the I'th constant or local variable or a coded number to indicate a parameter. This coded number is computed as -100-NPARM where NPARM is the position of the parameter in the parameter list of the SUBROUTINE statement. ISYMB(8,I) is the initial value associated with the I'th variable name or constant. ISYMB(9,I) is a definition indicator for the I'th variable name or constant. The values of the definition indicator are 0 for an undefined variable, 1 for a constant or defined variable and 2 for a parameter. In the second pass ISYMB(1,I) contains the ending statement number of the I'th DO loop. ISYMB(2,I) contains the location of the loop index variable. ISYMB(3,I) contains the location of the variable or constant whose value will terminate the loop. ISYMB(4,I) contains the location of the variable or constant whose value is used as an increment.

ISYMB(5,I) contains the location at the top of the loop to be branched to in order to repeat the loop.

ISTRN is used both for storing character strings encountered in FORMAT statements and values established in DATA statements. The ASCII values for strings found in formats are stored starting in ISTRN(3). NSTRN is the variable that indicates the last position in ISTRN used for storing character strings. The value of NDATA is equal to twice the number of values from DATA statements stored in ISTRN. ISTRN(601-NDATA) contains the last value established in a DATA statement and ISTRN(602-NDATA) contains the location associated with this value.

IFORM is the list for storing all format information other than the contents of character strings. Format information is stored starting at IFORM(1). The A1 and A2 formats are represented by 1 and 2 respectively. The value used to represent an I format is computed as 100 plus the length of the field. For example, an I10 format causes a value of 110 to be stored in IFORM. The U format is represented by a zero. Character strings are represented by a 3 followed by a reference to the position in ISTRN containing the character string. A slash mark is represented by a 3 followed by a 1. The value used to represent an X format is computed as 200 plus the length of the field. The value of -1 in IFORM indicates the end of the representation of a FORMAT statement. IFNDX is the index to formats in IFORM. The statement number associated with the I'th FORMAT state-

ment is stored in IFNDX(1,I). IFNDX(2,I) contains the position in IFORM that the representation of the I'th FORMAT statement starts.

Intermediate Code

Intermediate code is generated by the first pass of the compiler to convey the contents of the FORTRAN source statements to the second pass. The use of intermediate code eliminates the need for syntax error checking during the second pass, since analysis of each FORTRAN statement is performed in the first pass. Intermediate code is generated for executable statements only.

The first number in the intermediate code for any statement is the associated statement number. If the statement is not numbered, a zero value is stored in the first position of the line of intermediate code. The second number in the line of intermediate code indicates the type of statement represented. The values of this number are 1 for READ, 2 for WRITE, 4 for assignment, 8 for GO TO, 9 for IF, 10 for DO, 14 for STOP and 19 for CONTINUE. The other numbers between 1 and 21 are associated with non-executable statements for which no intermediate code is generated. Variables and constants encountered in the FORTRAN source are represented in the intermediate code by their machine code locations. The value used to represent parameters is computed as -100-NPARM where NPARM is the position of the parameter in the parameter list of the SUBROUTINE statement.

TABLE II

USES OF DATA STRUCTURES

Structure Name	Use
LBUF	Buffer for line of input.
LINE	Buffer for FORTRAN statement.
INTL	Buffer for line of intermediate code.
ISTBL	Table of subroutine names and addresses.
ISYMB	Symbol table and table for loop information.
ISTRN	List of character string values and values from DATA statements.
IFORM	List of values representing information from FORMAT statements.
IFNDX	Index to data in IFORM.

TABLE III

USAGE OF THE ISYMB DATA STRUCTURE

Pass of Compilation	ISYMB Subscript	Use
1	1-6	Alphanumeric representation of variable name or constant.
1	7	Location of constant or variable.
1	8	Initial value of constant or variable.
1	9	Definition indicator for constant or variable.
2	1	Ending statement number for loop.
2	2	Location of loop index variable.
2	3	Location of value to terminate loop.
2	4	Location of value to be used to increment loop index.
2	5	Location at top of loop to be branched to in order to repeat the loop.

Subscripts are represented as a -1 followed by the representation of the subscript variable or constant.

In READ and WRITE statements a reference to the variable or constant denoting the logical unit number follows the statement type identifier in the intermediate code. This is followed by the specified format number and zero or more variable references to represent the I/O list.

In assignment statements a reference to the target variable follows the statement type identifier. This is followed by a reverse Polish representation of the assignment expression consisting of variable references and numbers to represent operators. These numbers are -1 for subscripting, -2 for exponentiation, -3 for multiplication, -4 for division, -5 for addition and -6 for subtraction.

In GO TO statements the statement number to be transferred to follows the statement type identifier. The intermediate code for STOP and CONTINUE statements does not extend past the statement type identifier.

If the argument list of an arithmetic IF statement is not a simple variable or constant then the intermediate code for the arithmetic expression portion of an assignment statement is generated to allow the code for the IF statement to contain only a simple reference to represent the argument. The three statement numbers to be used for branching follow the reference to the argument of the IF statement.

For a DO statement the statement number to appear at the

end of the loop follows the statement type identifier. This is followed by references to the index variable, starting value and ending value. If an increment is specified then a reference to the increment appears at the end of the intermediate code for a DO statement.

For a CALL statement the execution address of the called subroutine follows the statement type identifier. In both SUBROUTINE and CALL statements the intermediate code includes references to every parameter in the list.

Table Driven Routines

The logic of one of the parsing routines (PARS2) and the general purpose lexical analyzer (LEX1) are controlled by tables. These tables contain data governing the state transitions of a finite state automaton (1).

ITRAN is a 5 row by 14 column table governing the operation of LEX1 (flowchart in Figure 9). Each column of the table corresponds to a class of character as determined by the character classification routine. Each row corresponds to a non-final state of the finite state automaton. Each entry in the table is the next state to be entered. Final states are numbered 11 through 21 and correspond to token types 1 through 11. The state is initially set to 1 and a state transition occurrs for every character of FORTRAN text processed. Characters are processed until a final state is reached.

IFSA is a 37 row by 12 column table governing the opera-

TABLE IV

GENERATION OF INTERMEDIATE CODE

Statement Type	Construct Encountered in Source Code	Intermediate Code Generated
Any	Local variable or constant	Location of variable or constant
Any	Subscript operator	-1
Any	Parameter	-100-parameter position in SUBROUTINE statement
Any	Statement number	Value of statement number
Assignment	** operator	-2
Assignment	* operator	-3
Assignment	/ operator	-4
Assignment	+ operator	-5
Assignment	- operator	-6
CALL	Subroutine name	Execution address of called subroutine
READ	Start of statement	1
WRITE	Start of statement	2
Assignment	Start of statement	4
GO TO	Start of statement	8
IF	Start of statement	9
DO	Start of statement	10
STOP	Start of statement	14
CONTINUE	Start of statement	19

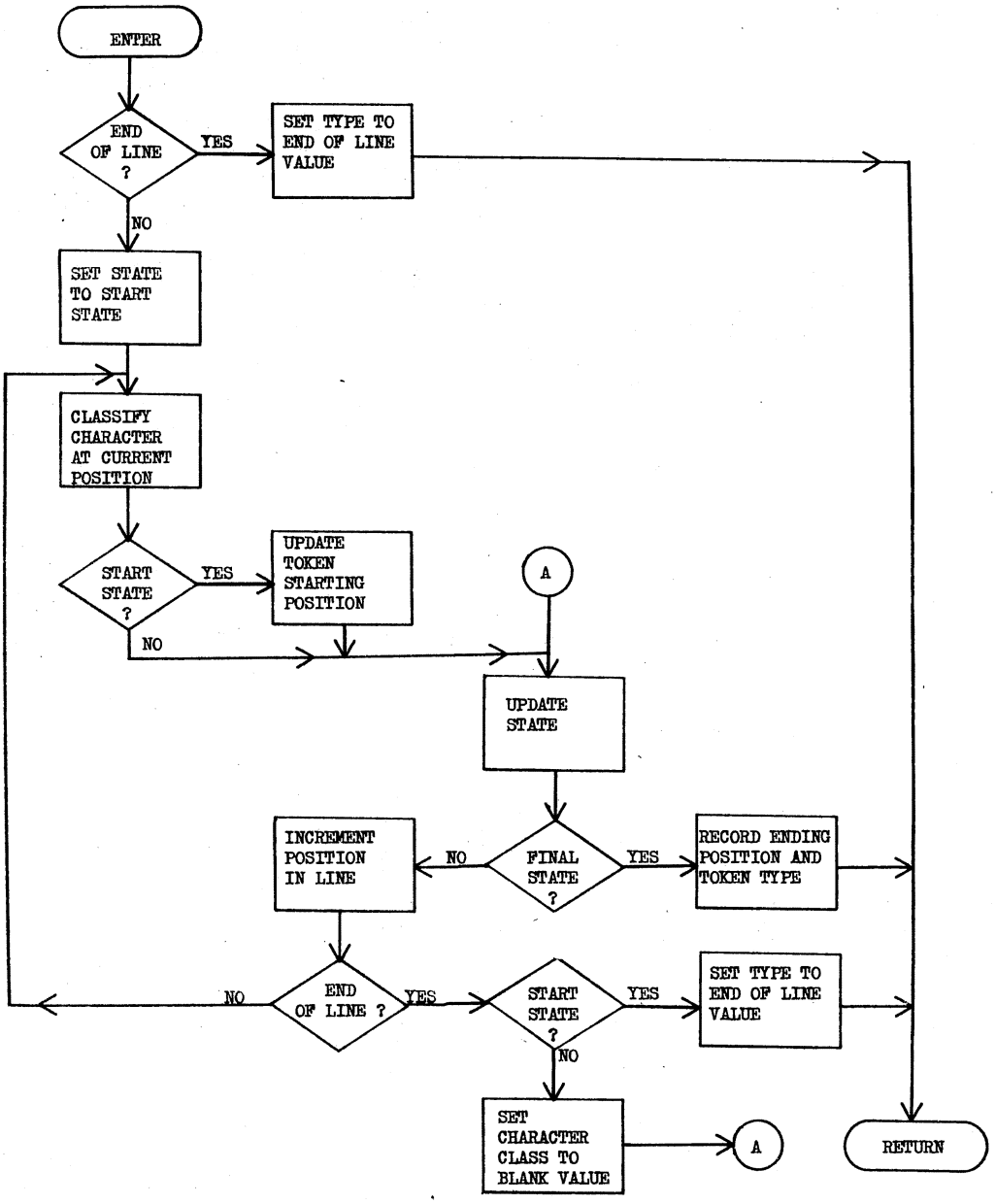

Figure 9. Flowchart of LEX1

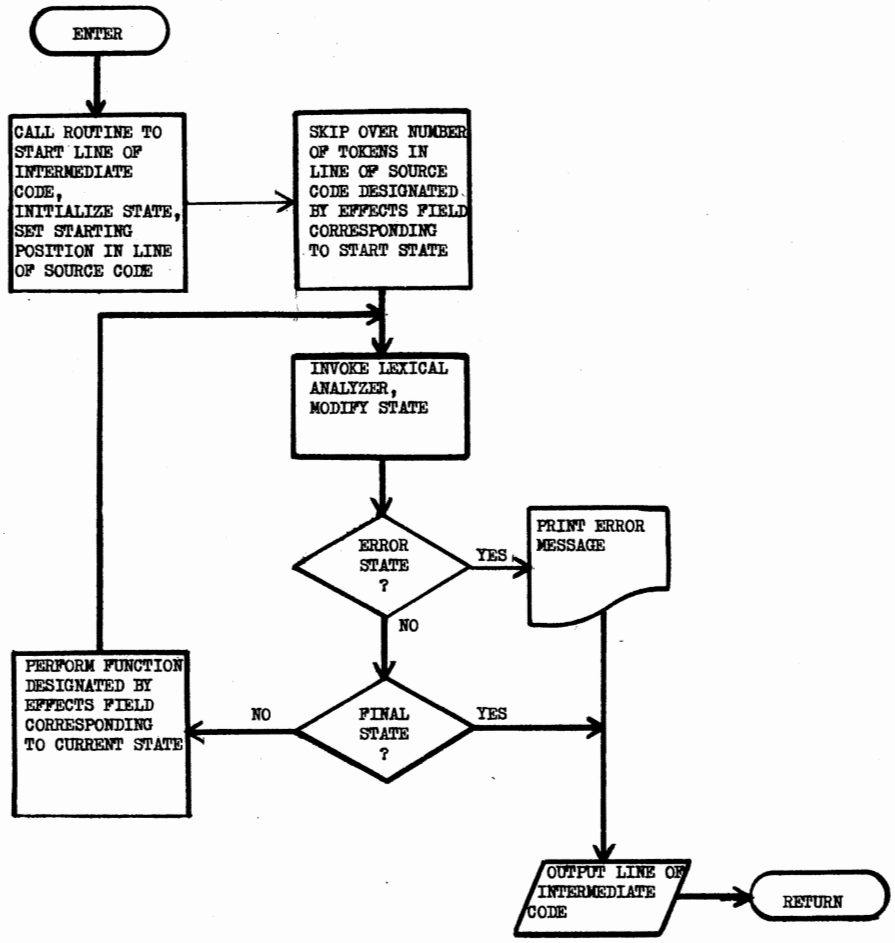

Figure 10. Flowchart of PARS2

tion of PARS2 (flowchart in Figure 10). Each row in the table corresponds to a non-final state in the finite state automaton. The first eleven columns correspond to the eleven types of tokens returned by LEX1. The parameter ISTRT indicates which row of the table to use as a start state. Entries in the first eleven columns of IFSA indicate the displacement from the start state of the next state to be entered. An entry of 99 implies a transfer to the error state and an entry of 100 implies a transfer to a final state with no errors detected. The number in the last column of a row of IFSA that corresponds to a start state indicates how many tokens are to be ignored before processing begins. If the row does not correspond to a start state, entries in the last column of IFSA control the processing of the last token recognized. The functions associated with these values are as follows:

-1 Enter -1 in intermediate code.

0 Do nothing.

1 Enter value of constant in intermediate code.

2 Enter token in symbol table as a defined symbol and enter the associated location in intermediate code.

3 Enter token in symbol table as an undefined symbol and enter the associated location in intermediate code.

4 Enter token in subroutine table as a subroutine name.

5 Search subroutine table for token and enter associated location in intermediate code.

Non-executable Object Code

Object code is generated for all character strings encountered in formats, constants and variables. GSYMB is the routine that produces object code for constants and variables. The value generated for a variable is zero unless it has been initialized otherwise in a DATA satatement. GSTRN is the routine that produces object code for character strings encountered in FORMAT statements. The code produced by GSTRN consists of the ASCII representation of the character strings packed two characters per word of object code.

Executable Object Code

Object code is produced for every executable statement. Assembler listings of object code templates are included in the internal documentation of every routine that generates executable code.

The code generated for assignment statements uses register 0 as an accumulator and register 2 as an index register. Calls are generated to the arithmetic function routines (Appendix B) for exponentiation, multiplication, division and subscripting. Add and subtract instructions are generated for addition and subtraction.

The code generated for a CALL statement consists of a branch and link (BL) instruction to the execution address of the called subroutine followed by the addresses associated with the parameters in the list. The code for a SUBROUTINE statement modifies the return address to skip over the

parameter list and stores this address. For a RETURN statement, code is generated that will retrieve the return address and branch to it.

In the case of a DO loop, two different blocks of object code are generated. Before the top of the loop, the index variable is initialized. At the bottom of the loop, the index variable is incremented and tested, and a conditional branch to the top of the loop is generated.

A branch to an absolute address is generated for a GO TO statement. The code for an IF statement consists of a test of the argument and conditional branches to three absolute addresses.

The code for READ and WRITE statements starts by initializing the logical unit number and establishing a workspace for the I/O routines (Appendix B). For a WRITE statement, the I/O buffer is set to blank. For a READ statement, a line of data is read into the I/O buffer. In both READ and WRITE statements, calls are generated to the I/O routines to handle the various operations.

The code for a STOP statement consists of a branch and link workspace pointer (BLWP) instruction to the absolute address of FFFC hexadecimal. This has the same effect on the computer as an externally generated reset (19).

The GETPM routine generates the code to fetch a parameter. This routine may generate code to fetch the value of a parameter to any register or location. The code generated by GETPM also causes the address of the parameter to be

placed in register 10.

Summary

The compiler uses a number of line buffers and tables for storing information. All data structures have set capacities believed to be adequate for compiling most FORTRAN programs intended for use on a microcomputer. These capacities may be altered by the programmer as needed.

Intermediate code is generated for every executable FORTRAN statement processed. The use of intermediate code allows the parsing of source code and the generation of object code to be accomplished in separate passes of compilation.

Table driven routines exist for lexical analysis and generation of intermediate code from FORTRAN source code. Since the logic of these routines is governed by tables, they are comprised of a relatively small number of FORTRAN statements.

Object code is produced for character strings, constants and variables. Executable object code is generated for every executable FORTRAN statement processed.

CHAPTER VII

CONCLUSIONS AND RECOMMENDATIONS

The FORTRAN cross compiler described in this thesis is a useful program product for implementing a restricted class of programs on a TMS 9900 microcomputer system. The versatility of the compiler could be improved by modifying the compiler to support a more powerful FORTRAN subset. Possible improvements and additions include support for multi-dimensional arrays, floating point arithmetic, and function routines. Altering the compiler to be compatible with other object computers would also add to its usefulness.

The code generated by the compiler is compatible with any TMS 9900 or TI 990 computer system. Certain modifications to the assembly language input/output support routines may be required to allow for hardware differences between systems. The compiler may be implemented on many ASCII or EBCDIC host computers that support the FORTRAN language.

BIBLIOGRAPHY

1. Aho, A. V., and Ullman, J.D. The Theory of Parsing, Translation, and Compiling. Prentice-Hall, Englewood Cliffs, N. J., 1972.

2. American National Standards Committee X3J3. Basic FORTRAN. Standard no. X3.10-1966. New York, N. Y., 1966.

3. Barron, D. W. An Introduction to the Study of Programming Languages. Cambridge University Press, Cambridge, England, 1977.

4. Cocke, John, and Schwartz, J. T. Programming Languages and their Compilers. Courant Institute, New York, N. Y., 1970.

5. Ghandour, Z. J. Formal Systems and Syntactical Analys Yale University, New Haven, Conn., 1968.

6. Gries, David. Compiler Construction for Digital Computers. Wiley, New York, N. Y., 1971.

7. Halstead, M. H. A Laboratory Manual for Compiler and Operating System Implementation. American Elsevier, New York, N. Y., 1974.

8. Higman, Bryan. A Comparative Study of Programming Languages. American Elsevier, New York, N. Y., 1970.

9. Hopgood, F. R. A. Compiling Techniques. American Elsevier, New York, N. Y., 1970.

10. Knuth, D. E. The Art of Computer Programming. Addison-Wesley, Reading, Mass., 1973.

11. Lee, J. A. N. The Anatomy of a Compiler. Van Nostrand Reinhold, New York, N. Y., 1974.

12. Lewis, P. M., Rosenkrantz, D. J., and Stearns, R. E. Compiler Design Theory. Addison-Wesley, Reading, Mass., 1976.

13. Milne, Robert. A Theory of Programming Language Semantics. Wiley, New York, N. Y., 1976.

14. Naur, P. "Report on the Algorithmic Language ALGOL 60." Communications of the ACM, Vol. 3, No. 5 (May, 1960), pp. 299-314.

15. Pratt, T. W. Programming Languages: Design and Implementation. Prentice-Hall, Englewood Cliffs, N. J., 1975.

16. Rustin, Randall. Design and Optimization of Compilers Prentice-Hall, Englewood Cliffs, N. J., 1971.

17. Rustin, Randall. Formal Semantics of Programming Languages. Prentice-Hall, Englewood Cliffs, N. J., 1970.

18. Sammet, J. E. Programming Languages: History and Fundamentals. Prentice-Hall, Englewood Cliffs, N. J., 1969.

19. Texas Instruments Digital Systems Division. 990 Computer Family Systems Handbook. Manual no. 945250-9701. Texas Instruments, Austin, Texas, 1976.

20. Weingarten, F. W. Translation of Computer Languages. Holden-Day, San Francisco, Calif., 1973.

21. Wulf, William, Johnsson, R. K., Weinstock, C. B., Hobbs, S. O., and Geschke, C. M. The Design of an Optimizing Compiler. American Elsevier, New York, N. Y., 1975.

APPENDIX A

DESCRIPTIONS OF ROUTINES IN THE COMPILER

MAIN PROGRAM
FUNCTION:
INITIALIZES I/O UNIT NUMBERS, INVOKES CONTROL LANGUAGE PROCESSOR, INVOKES 3 PASSES OF COMPILER, MONITORS ERROR COUNT.
ROUTINES CALLED: CLPRO, PASS1, PASS2, PASS3

SUBROUTINE ADSTR(JPOS1,JPOS2)
FUNCTION:
ADD A STRING TO THE STRING TABLE.
PARAMETERS:
JPOS1 STARTING POSITION OF STRING IN LINE.
JPOS2 ENDING POSITION OF STRING IN LINE.
CALLED BY: PFORM
ROUTINES CALLED: ASCII

SUBROUTINE ASCII(ICHAR,JCODE)
FUNCTION:
CONVERT EBCDIC CHARACTER TO ASCII.
PARAMETERS:
ICHAR CHARACTER TO BE CONVERTED TO ASCII.
JCODE ASCII CODE REPRESENTATION OF ICHAR.
CALLED BY: ADSTR, PDATA
ROUTINES CALLED: SHIFT

SUBROUTINE BUFIO(IFLAG)
FUNCTION:
PRINT FORTRAN SOURCE CODE, KEEP THE INPUT BUFFER FULL AND DETECT THE END OF INPUT DATA.
PARAMETERS:
IFLAG FLAG TO INDICATE END OF SOURCE CODE.
CALLED BY: PASS1
ROUTINES CALLED: NONE

```
      SUBROUTINE CLASS(ICHAR,ICLAS)
FUNCTION:
     DETERMINE THE CLASS OF A GIVEN CHARACTER.
PARAMETERS:
       ICHAR    CHARACTER TO BE CLASSIFIED.
       ICLAS    NUMERICAL CLASSIFICATION OF ICHAR.

     CHARACTER CLASSES ARE DETERMINED AS FOLLOWS
     1     ALPHABETIC (I-N)
     2     ALPHABETIC (A-H,O-Z)
     3     NUMERIC
     4     PERIOD OR DECIMAL POINT
     5     PLUS SIGN
     6     MINUS SIGN
     7     ASTERISK
     8     SLASH MARK
     9     EQUALS SIGN
     10    PARENTHESIS OPEN
     11    PARENTHESIS CLOSED
     12    COMMA
     14    BLANK
     13    OTHER
CALLED BY: LEX1, SINST
ROUTINES CALLED: NONE


      SUBROUTINE CLPRO
FUNCTION:
     PROCESS CONTROL LANGUAGE STATEMENTS.
PARAMETERS: NONE
CALLED BY: MAIN PROGRAM
ROUTINES CALLED: HEX2, HEXIN, SBADD


      SUBROUTINE CSORT(NCOM,ISPOS,NXCOM)
FUNCTION:
     SORT COMMON BLOCK TO FRONT OF SYMBOL TABLE.
PARAMETERS:
       NCOM     NUMBER OF SYMBOLS IN COMMON BLOCK.
       ISPOS    POSITION IN SYMBOL TABLE OF SYMBOL TO BE
                MOVED TO FRONT OF TABLE.
       NXCOM    MACHINE CODE LOCATION FOR SYMBOL.
CALLED BY: PCOM
ROUTINES CALLED: NONE


      SUBROUTINE DSORT
FUNCTION:
     SORT DATA TABLE INTO ORDER OF INCREASING LOCATION.
PARAMETERS: NONE
CALLED BY: PASS1
ROUTINES CALLED: NONE
```

SUBROUTINE DTABL
FUNCTION:
DUMP THE STRING TABLE, SYMBOL TABLE AND FORMAT TABLE.
PARAMETERS: NONE
CALLED BY: PASS1
ROUTINES CALLED: NONE

SUBROUTINE GASSG
FUNCTION:
GENERATE MACHINE CODE FOR ASSIGNMENT STATEMENTS.
PARAMETERS: NONE
CALLED BY: PASS2
ROUTINES CALLED: HEX, WDOUT, GETPM

SUBROUTINE GCALL
FUNCTION:
GENERATE MACHINE CODE FOR A CALL STATEMENT.
PARAMETERS: NONE
CALLED BY: PASS2
ROUTINES CALLED: HEX, GETPM, WDOUT

SUBROUTINE GDO1
FUNCTION:
GENERATE MACHINE CODE AT TOP OF DO LOOP AND RECORD
DO LOOP INFORMATION.
PARAMETERS: NONE
CALLED BY: PASS2
ROUTINES CALLED: HEX, GETPM, WDOUT

SUBROUTINE GDO2
FUNCTION:
GENERATE MACHINE CODE AT END OF DO LOOPS.
PARAMETERS: NONE
CALLED BY: PASS2
ROUTINES CALLED: HEX, GETPM, WDOUT

```
      SUBROUTINE GETPM(LOCP,IDEST,ISUB)
FUNCTION:
     GENERATE CODE TO FETCH THE VALUE OF A PARAMETER
     TO A GIVEN LOCATION OR REGISTER AND TO FETCH THE
     LOCATION OF THE PARAMETER TO REGISTER 10.
PARAMETERS:
     LOCP       LOCATION CODE FOR PARAMETER
     IDEST      DESTINATION CODE FOR PARAMETER VALUE.
                IDEST>0 FOR FETCH TO MEMORY.
     ISUB       SUBSCRIPTING INDICATOR.
     ISUB       SUBSCRIPTING INDICATOR.
                ISUB=0 FOR NO SUBSCRIPTING.
                ISUB=1 TO USE VALUE IN R2 AS A SUBSCRIPT.
CALLED BY: GASSG, GCALL, GDO1, GDO2, GIF, GIO
ROUTINES CALLED: HEX, WDOUT


      SUBROUTINE GGOTO
FUNCTION:
     GENERATE MACHINE CODE FOR A GO TO STATEMENT AND RECORD
     REFERENCED STATEMENT NUMBER.
PARAMETERS: NONE
CALLED BY: PASS2
ROUTINES CALLED: WDOUT


      SUBROUTINE GIF
FUNCTION:
     GENERATE MACHINE CODE FOR IF STATEMENTS AND RECORD
     REFERENCED STATEMENT NUMBERS.
PARAMETERS: NONE
CALLED BY: PASS2
ROUTINES CALLED: HEX, GETPM, WDOUT


      SUBROUTINE GIO
FUNCTION:
     GENERATE MACHINE CODE FOR READ AND WRITE STATEMENTS.
PARAMETERS: NONE
CALLED BY: PASS2
ROUTINES CALLED: HEX, GETPM, WDOUT


      SUBROUTINE GOCHK(IOK)
FUNCTION:
     CHECK FOR THE KEYWORD 'TO' FOLLOWING THE KEYWORD 'GO'.
PARAMETERS:
     IOK=1 IF 'TO' FOLLOWS GO.
     IOK=0 IF 'TO' DOES NOT FOLLOW 'GO'.
CALLED BY: PARS1
ROUTINES CALLED: LEX1
```

```
      SUBROUTINE GRETN
FUNCTION:
     GENERATE MECHINE CODE FOR A RETURN STATEMENT.
PARAMETERS: NONE
CALLED BY: PASS2
ROUTINES CALLED: HEX, WDOUT


      SUBROUTINE GSTOP
FUNCTION:
     GENERATE MACHINE CODE FOR A STOP STATEMENT.
PARAMETERS: NONE
CALLED BY: PASS2
ROUTINES CALLED: WDOUT


      SUBROUTINE GSTRN
FUNCTION:
     GENERATE MACHINE CODE FOR STRING TABLE.
PARAMETERS: NONE
CALLED BY: PASS2
ROUTINES CALLED: HEX, MCOUT


      SUBROUTINE GSUBR
FUNCTION:
     GENERATE MACHINE CODE FOR A SUBROUTINE STATEMENT AND
     STORE THE EXECUTION ADDRESS IN THE SUBROUTINE TABLE.
PARAMETERS: NONE
CALLED BY: PASS2
ROUTINES CALLED: HEX, WDOUT


      SUBROUTINE GSYMB
FUNCTION:
     GENERATE MACHINE CODE FOR SYMBOL TABLE.
PARAMETERS: NONE
CALLED BY: PASS2
ROUTINES CALLED: HEX, MCOUT


      SUBROUTINE HEX(IHEX,IVAL)
FUNCTION:
     CONVERT A BINARY VALUE TO FOUR HEX DIGITS.
PARAMETERS:
     IHEX    FOUR HEX DIGITS.
     IVAL    BINARY VALUE FOR CONVERSION.
CALLED BY: GASSG, GCALL, GDO1, GDO2, GETPM, GIF, GIO, GRETN,
     GSTOP, GSUBR, GSYMB, MCOUT, PASS2, PASS3
ROUTINES CALLED: NONE
```

```
      SUBROUTINE HEX2(IHEX,IVAL)
FUNCTION:
     CONVERT FOUR HEX DIGITS TO A BINARY VALUE.
PARAMETERS:
     IHEX    FOUR HEX DIGITS.
     IVAL    BINARY VERSION OF IHEX.
CALLED BY: CLPRO
ROUTINES CALLED: NONE


      SUBROUTINE HEXIN
FUNCTION:
     READ HEXADECIMAL INPUT AND WRITE IT DIRECTLY
     TO THE MACHINE CODE FILE UNTIL A CONTROL STATEMENT IS
     ENCOUNTERED.
PARAMETERS: NONE
CALLED BY: CLPRO
ROUTINES CALLED: NONE


      SUBROUTINE INITZ
FUNCTION:
     INITIALIZE THE STRING, SYMBOL AND FORMAT TABLES.
PARAMETERS: NONE
CALLED BY: PASS1
ROUTINES CALLED: NONE


      SUBROUTINE INTIN(IFLAG)
FUNCTION:
     INPUT LINE OF INTERMEDIATE CODE AND DETECT END OF
     PROGRAM.
PARAMETERS:
     IFLAG     FLAG SET=0 AT END OF INTERMEDIATE CODE.
CALLED BY: PASS2
ROUTINES CALLED: NONE


      SUBROUTINE INTLN(KPOS)
FUNCTION:
     OUTPUT LINE OF INTERMEDIATE CODE.
PARAMETERS:
     KPOS     LENGTH OF LINE OF INTERMEDIATE CODE.
CALLED BY: PASS1, PARS2, PASSG, PTRIV
ROUTINES CALLED: NONE
```

```
      FUNCTION JCONV(JPOS1,JPOS2)
FUNCTION:
     RETURN A NUMERIC VALUE FOR AN INTEGER CONSTANT IN
     THE LINE OF INPUT.
PARAMETERS:
     JPOS1     STARTING POSITION OF INTEGER IN LINE.
     JPOS2     ENDING POSITION OF INTEGER IN LINE.
CALLED BY: LEX2, PARS2, PDIM, PFORM, SINST, STRTI
ROUTINES CALLED: NONE


      SUBROUTINE LEX1(IPOS,JPOS1,JPOS2,JTYPE)
FUNCTION:
     GENERAL PURPOSE LEXICAL ANALYZER.
PARAMETERS:
     IPOS      POSITION IN LINE TO BEGIN SCAN.
     JPOS1     POSITION IN LINE OF FIRST CHARACTER OF TOKEN.
     JPOS2     POSITION IN LINE OF LAST CHARACTER OF TOKEN.
     JTYPE     TYPE OF TOKEN

     TOKEN TYPES
     1.  INTEGER IDENTIFIERS.
     2.  FLOATING POINT IDENTIFIERS.
     3.  INTEGER CONSTANTS.
     4.  FLOATING POINT CONSTANTS.
     5.  OPERATORS.  ( +,-,*,/ )
     6.  EQUALS SIGN.
     7.  LEFT PARENTHESIS.
     8.  RIGHT PARENTHESIS.
     9.  COMMA.
    10.  ERROR.
    11.  END OF FIELD.
CALLED BY: GOCHK, LEX2, LEX3, PASSG, PCOM, PDATA, PDIM,
     PFORM, PIF, PTRIV, STRTI, STYPE
ROUTINES CALLED: CLASS
```

```
      SUBROUTINE LEX2(IPOS,JPOS0,JPOS1,JPOS2,JTYPE,JNUM)
FUNCTION:
     LEXICAL ANALYZER FOR USE WITH FORMATS.
PARAMETERS:
     IPOS       POSITION IN LINE TO BEGIN SCAN.
     JPOS0      POSITION IN LINE OF NUMBER PRECEEDING TOKEN.
     JPOS1      POSITION IN LINE OF FIRST CHARACTER OF TOKEN.
     JPOS2      POSITION IN LINE OF LAST CHARACTER OF TOKEN.
     JTYPE      TYPE OF TOKEN
     JNUM       NUMBER PRECEEDING TOKEN ( FOR 5I10, JNUM=5 ).

     TOKEN TYPES
     1.  I FORMAT.
     2.  A FORMAT.
     3.  H FIELD.
     4.  X FIELD.
     5.  SLASH MARK.
     6.  SPECIAL U FORMAT.
     7.  LEFT PARENTHESIS.
     8.  RIGHT PARENTHESIS.
     9.  COMMA.
    10.  ERROR.
    11.  END OF FIELD.
CALLED BY: PDATA, PFORM
ROUTINES CALLED: LEX1, JCONV
```

```
      SUBROUTINE LEX3(IPOS,JPOS1,JPOS2,JCODE,LCODE)
FUNCTION:
     IDENTIFY TOKENS IN ASSIGNMENT STATEMENTS AND HAVE
     IDENTIFIERS AND CONSTANTS INSERTED IN THE SYMBOL TABLE.
PARAMETERS:
     IPOS      POSITION IN LINE TO BEGIN SCAN.
     JPOS1     POSITION IN LINE OF FIRST CHARACTER OF TOKEN.
     JPOS2     POSITION IN LINE OF LAST CHARACTER OF TOKEN.
     JCODE     INDICATOR FOR TYPE OF TOKEN.
     LCODE     PREVIOUS VALUE OF JCODE.

     VALUES OF JCODE
      0      END OF FIELD
     -1      SUBSCRIPT OPERATOR
     -2      EXPONENTIATION
     -3      MULTIPLICATION
     -4      DIVISION
     -5      ADDITION
     -6      SUBTRACTION
     -7      LEFT PARENTHESIS
     -8      RIGHT PARENTHESIS
     >0      LOCATION CORRESPONDING TO INTEGER IDENTIFIER
             OR CONSTANT
     <-100   LOCATION CODE FOR A PARAMETER
CALLED BY: PASSG
ROUTINES CALLED: LEX1, SINST


      SUBROUTINE MCOUT(IHEX)
FUNCTION:
     ADD FOUR HEX DIGITS TO GENERATED MACHINE CODE.
PARAMETERS:
     IHEX    FOUR HEX DIGITS.
CALLED BY: GSTRN, GSYMB, WDOUT
ROUTINES CALLED: HEX


      SUBROUTINE PARS1(ITYPE)
FUNCTION:
     SELECT THE APPROPRIATE SUBROUTINE TO GENERATE
     INTERMEDIATE CODE FOR A FORTRAN STATEMENT.
PARAMETERS:
     ITYPE    TYPE OF STATEMENT AS DETERMINED BY STYPE.
CALLED BY: PASS1
ROUTINES CALLED: STYPE, PARS2, PFORM, PASSG, PDIM, PDATA,
     PCOM, GOCHK, PIF, PTRIV
```

```
      SUBROUTINE PARS2(ISTRT,ITYPE)
FUNCTION:
     GENERATE INTERMEDIATE CODE FOR NONTRIVIAL STATEMENTS
     THAT CAN BE PARSED WITH A DETERMINISTIC FINITE STATE
     AUTOMATON WITHOUT PUSHDOWN LISTS.
PARAMETERS:
     ISTRT     NUMBER OF START STATE TO USE.
     ITYPE     TYPE OF STATEMENT.
CALLED BY: PARS1, PIF
ROUTINES CALLED: STRTI, JCONV,  LEX1, SBSRC, SBADD, SINST,
     SYNER, INTLN


      SUBROUTINE PASS1(IEOF)
FUNCTION:
     DRIVER FOR FIRST PASS OF COMPILER.
PARAMETERS:
     IEOF    END OF FORTRAN SOURCE FILE FLAG.
CALLED BY: MAIN PROGRAM
ROUTINES CALLED: INITZ, BUFIO, PARS1, INTLN, UDCHK, DSORT,
     DTABL


      SUBROUTINE PASS2
FUNCTION:
     DRIVER FOR SECOND PASS OF COMPILER.
PARAMETERS: NONE
CALLED BY: MAIN PROGRAM
ROUTINES CALLED: GSYMB, GSTRN, HEX, INTIN, GASSG, GGOTO,
     GIF, GIO, GDO1, GSTOP, GSUBR, GRETN, GCALL, GDO2


      SUBROUTINE PASS3
FUNCTION:
     FILL IN ADDRESSES IN MACHINE CODE GENERATED DURING THE
     SECOND PASS OF THE COMPILER.
PARAMETERS: NONE
CALLED BY: MAIN PROGRAM
ROUTINES CALLED: STNUM, HEX


      SUBROUTINE PASSG
FUNCTION:
     PARSE AND GENERATE INTERMEDIATE CODE FOR ASSIGNMENT
     STATEMENTS.
PARAMETERS: NONE
CALLED BY: PARS1, PIF
ROUTINES CALLED: STRTI, LEX1, SINST, LEX3, SYNER, INTLN
```

```
      SUBROUTINE PCOM
FUNCTION:
     PROCESS COMMON STATEMENT
PARAMETERS: NONE
CALLED BY: PARS1
ROUTINES CALLED: LEX1, SSRCH, SINST, CSORT, SYNER


      SUBROUTINE PDATA
FUNCTION:
     PARSE DATA STATEMENTS AND STORE VALUES AND LOCATIONS OF
     VARIABLES ESTABLISHED IN DATA STATEMENTS.
PARAMETERS: NONE
CALLED BY: PARS1
ROUTINES CALLED: LEX1, SINST, SSRCH, LEX2, ASCII, SYNER


      SUBROUTINE PDIM
FUNCTION:
     PARSE DIMENSION STATEMENTS AND MODIFY LOCATIONS IN
     SYMBOL TABLE TO ALLOW ROOM FOR DIMENSIONED VARIABLE.
PARAMETERS: NONE
CALLED BY: PARS1
ROUTINES CALLED: LEX1, JCONV,  SSRCH, SINST, SYNER


      SUBROUTINE PFORM
FUNCTION:
     PARSE FORMAT STATEMENTS AND RECORD INFORMATION
     CONTAINED WITHIN FORMATS.
PARAMETERS: NONE
CALLED BY: PARS1
ROUTINES CALLED: STRTI, JCONV,  LEX1, LEX2, ADSTR, SYNER


      SUBROUTINE PIF
FUNCTION:
     GENERATE INTERMEDIATE CODE FOR AN ARITHMETIC IF.
PARAMETERS: NONE
CALLED BY: PARS1
ROUTINES CALLED: LEX1,PASSG, PARS2, SYNER


      SUBROUTINE PTRIV
FUNCTION:
     GENERATE INTERMEDIATE CODE FOR RETURN, CONTINUE AND
     STOP STATEMENTS.
PARAMETERS:
     ITYPE     TYPE OF STATEMENT AS DETERMINED BY STYPE
CALLED BY: PARS1
ROUTINES CALLED: STRTI, LEX1, INTLN
```

```
      SUBROUTINE SBADD(JPOS1,JPOS2,JLOC)
FUNCTION:
     ADD A SUBROUTINE NAME TO THE SUBROUTINE TABLE.
PARAMETERS:
       JPOS1     STARTING POSITION OF SUBROUTINE NAME IN
                 LINE.
       JPOS2     ENDING POSITION OF SUBROUTINE NAME IN LINE.
       JLOC      MACHINE CODE LOCATION ASSOCIATED WITH
                 SUBROUTINE NAME.
CALLED BY: CLPRO, PARS2
ROUTINES CALLED: NONE


      SUBROUTINE SBSRC(JPOS1,JPOS2,JLOC)
FUNCTION:
     SEARCH FOR A SUBROUTINE NAME IN THE SUBROUTINE TABLE
     AND RETURN THE ASSOCIATED MACHINE CODE LOCATION.
PARAMETERS:
       JPOS1     STARTING POSITION OF SUBROUTINE NAME IN
                 LINE.
       JPOS2     ENDING POSITION OF SUBROUTINE NAME IN LINE.
       JLOC      MACHINE CODE LOCATION ASSOCIATED WITH
                 SUBROUTINE NAME.
CALLED BY: PARS2
ROUTINES CALLED: NONE


      SUBROUTINE SHIFT(ICHAR,JCHAR)
FUNCTION:
     SHIFT A CHARACTER FROM THE LEFTMOST BYTE OF A WORD TO
     THE RIGHTMOST BYTE.
PARAMETERS:
     ICHAR     ORIGINAL UNSHIFTED WORD.
     JCHAR     SHIFTED VERSION OF ICHAR.
CALLED BY: ASCII
ROUTINES CALLED: NONE


      SUBROUTINE SINST(JPOS1,JPOS2,LOC,IVAL,DEF)
FUNCTION:
     INSERT AN IDENTIFIER INTO THE SYMBOL TABLE.
PARAMETERS:
     JPOS1     STARTING POSITION OF IDENTIFIER IN LINE.
     JPOS2     ENDING POSITION OF IDENTIFIER IN LINE.
     LOC       ZERO BASED ASSEMBLER LOCATION OF SYMBOL.
     IVAL      VALUE ASSOCIATED WITH SYMBOL.
     IDEF      DEFINITION INDICATOR. IDEF=1 IMPLIES THAT
               SYMBOL IS TO BE INSERTED AS A DEFINED SYMBOL.
CALLED BY: LEX3, PARS2, PASSG, PCOM, PDATA, PDIM
ROUTINES CALLED: SSRCH, JCONV,  CLASS
```

```
      SUBROUTINE SSRCH(JPOS1,JPOS2,ISPOS)
FUNCTION:
     SEARCH FOR A SYMBOL IN THE SYMBOL TABLE AND RETURN ITS
     POSITION IN THE TABLE IF FOUND.
PARAMETERS:
     JPOS1      POSITION IN LINE OF START OF SYMBOL.
     JPOS2      POSITION IN LINE OF END OF SYMBOL.
     ISPOS      POSITION OF SYMBOL IN SYMBOL TABLE.
CALLED BY: PCOM, PDATA, PDIM, SINST
ROUTINES CALLED: NONE


      SUBROUTINE STNUM
FUNCTION:
     RECORD ADDRESSES ASSOCIATED WITH REFERENCED STATEMENT
     NUMBERS.
PARAMETERS: NONE
CALLED BY: PASS3
ROUTINES CALLED: NONE


      SUBROUTINE STRTI
FUNCTION:
     START LINE OF INTERMEDIATE CODE.
PARAMETERS: NONE
CALLED BY: PARS2, PASSG, PTRIV
ROUTINES CALLED: JCONV
```

```
      SUBROUTINE STYPE
FUNCTION:
     DETERMINE THE TYPE OF A STATEMENT.
PARAMETERS:
     ITYPE     NUMERIC VALUE FOR TYPE OF STATEMENT.
               VALUES FOR ITYPE:
                 1  READ
                 2  WRITE
                 3  FORMAT
                 4  ASSIGNMENT
                 5  DIMENSION
                 6  DATA
                 7  COMMON
                 8  GO TO
                 9  IF
                10  DO
                11  EQUIVALENCE
                12  REAL
                13  INTEGER
                14  STOP
                15  END
                16  SUBROUTINE
                17  RETURN
                18  FUNCTION
                19  CONTINUE
                20  CALL
                21  ERROR
CALLED BY: PARS1
ROUTINES CALLED: LEX1


      SUBROUTINE SYNER(JPOS)
FUNCTION:
     PRINT SYNTAX ERROR MESSAGE.
PARAMETERS:
     JPOS      POSITION IN LINE THAT ERROR WAS DETECTED.
CALLED BY: PARS2, PASSG, PCOM, PDATA, PDIM, PFORM, PIF
ROUTINES CALLED: NONE


      SUBROUTINE UDCHK
FUNCTION:
     CHECK FOR UNDEFINED VARIABLES IN SYMBOL TABLE.
PARAMETERS: NONE
CALLED BY: PASS1
ROUTINES CALLED: NONE
```

```
      SUBROUTINE WDOUT(MCODE,MSTRT,MLEN)
FUNCTION:
     OUTPUT A BLOCK OF WORDS OF MACHINE CODE
PARAMETERS:
       MCODE            TEMPLATE OF MACHINE CODE.
       MSTRT            FIRST WORD IN TEMPLATE TO BE OUTPUT.
       MLEN             LENGTH OF BLOCK OF WORDS TO BE OUTPUT
CALLED BY: GASSG, GCALL, GDO1, GDO2, GETPM, GGOTO, GIF,
     GIO, GRETN, GSTOP, GSUBR
ROUTINES CALLED: MCOUT
```

APPENDIX B

LISTING OF ASSEMBLER SUPPORT ROUTINES

```
label   op code &
        operands            comments
*
*       TMS 9900 FORTRAN CROSS COMPILER I/O ROUTINES
*
        AORG >80            START GENERATING CODE AT
*                           HEXADECIMAL LOCATION 80
WKSP    BSS  32             I/O WORKSPACE
BUFFR   BSS  76             I/O BUFFER
COUNT   BSS  2              I/O COUNTER
UNIT    BSS  2              LOGICAL UNIT NUMBER
INDEX   BSS  16             UNIT NUMBER TO BAUD RATE INDEX
*                           NUMBER IN INDEX=63300/BAUD RATE-5
SAVE    BSS  4              REGISTER SAVE AREA
        AORG >F000          GENERATE EXECUTABLE CODE AT F000
*
*       ROUTINE TO PRODUCE DELAY OF 1/2 I/O CLOCK CYCLE
*
WAIT    MOV  R12,R13        R13=2*UNIT NUMBER=OFFSET IN INDEX
        SLA  R13,1
        MOV  @INDEX(R13),R14   R14=NUMBER FROM INDEX
WAIT1   DEC  R14            LOOP TO CAUSE DELAY
        JNE  WAIT1
        B    *R11
*
*       ROUTINE TO INPUT A CHARACTER INTO R4
*
INPT    MOV  R11,R9         SAVE RETURN
        MOV  @UNIT,R12      R12=UNIT NUMBER
INPT1   TB   0              WAIT FOR START BIT
        JEQ  INPT1
        CLR  R4
        LI   R0,>101        R0=BIT MAP
        BL   @WAIT          WAIT FOR MID BIT
INPT2   BL   @WAIT          WAIT FOR NEXT BIT
        BL   @WAIT
        TB   0              INPUT BIT
        JNE  INPT3          TEST FOR 1 BIT
        XOR  R0,R4
INPT3   SLA  R0,1           UPDATE MASK
        JNC  INPT2          IF NOT LAST BIT, PROCESS NEXT BIT
```

```
        BL   @WAIT          WAIT FOR STOP BIT
        BL   @WAIT
        ANDI R4,>7F         REMOVE PARITY BIT
        B    *R9            RETURN
*
*       ROUTINE TO OUTPUT RIGHT BYTE OF R4, THEN IF LEFT BYTE
*       OF R4 IS NOT ZERO, OUTPUT IT ALSO.
*
TYPE    MOV  @UNIT,R12      R12=UNIT NUMBER
        MOV  R11,R9         SAVE RETURN
TYPE0   LI   R6,1           R6=BIT MAP
        BL   @WAIT          WAIT ON LAST I/O
        BL   @WAIT
TYPE1   SBZ  0              BIT=0
TYPE2   BL   @WAIT          WAIT FOR PREVIOUS BIT
        BL   @WAIT
        CI   R6,>100        TEST FOR END
        JEQ  TYPE4
        COC  R6,R4          TEST CURRENT BIT
        JEQ  TYPE3
        SLA  R6,1           BIT=0
        JMP  TYPE1
TYPE3   SLA  R6,1           BIT=0
        SBO  0
        JMP  TYPE2
TYPE4   SBO  0              ISSUE STOP BIT
        BL   @WAIT
        BL   @WAIT
        SRL  R4,8           CHECK LEFT BYTE
        JNE  TYPE0          IF NOT ZERO THEN OUTPUT IT
        B    *R9            RETURN
*
*       ROUTINE TO INPUT LINE OF DATA
*
LNIN    MOV  R13,@SAVE      SAVE REGISTERS
        MOV  R14,@SAVE+2
        LI   R5,WKSP        INVOKE ROUTINE TO CLEAR COUNTER AND
        LI   R6,BLANK       SET BUFFER TO BLANKS
        BLWP R5
        LI   R3,BUFFR
LNIN2   BL   @INPT          READ CHARACTER
        CI   R4,>0D         IF CR THEN RETURN
        JEQ  LNIN3
        CI   R3,COUNT       TEST FOR END OF BUFFER
        JEQ  LNIN2
        SLA  R4,8           STORE CHARACTER
        MOVB R4,*R3+
        JMP  LNIN2
LNIN3   LI   R4,>0A0D       OUTPUT CR, LF
        BL   @TYPE
        MOV  @SAVE,R13      RESTORE REGISTERS
        MOV  @SAVE+2,R14
        RTWP                RETURN
*
```

```
*       ROUTINE TO OUTPUT LINE OF DATA
*
LNOUT   MOV   R13,@SAVE     SAVE REGISTERS
        MOV   R14,@SAVE+2
        CLR   @COUNT        RESET I/O COUNTER
        LI    R2,COUNT      FIND LAST NON BLANK CHARACTER IN
LNOUT0  DECT  R2            I/O BUFFER
        CI    R2,BUFFR
        JEQ   LNOUT1
        MOV   *R2,R4
        CI    R4,>2020
        JEQ   LNOUT0
LNOUT1  INCT  R2
        LI    R1,BUFFR
LNOUT2  C     R1,R2         TEST FOR LAST CHARACTER
        JEQ   LNOUT4
        MOVB  *R1+,R4       OUTPUT CHARACTER
        SRL   R4,8
        BL    @TYPE
        LI    R5,200        DELAY BETWEEN CHARACTERS
LNOUT3  DEC   R5
        JNE   LNOUT3
        JMP   LNOUT2
LNOUT4  LI    R4,>0A0D      SEND CR, LF
        BL    @TYPE
        LI    R5,20000      DELAY FOR CR, LF
LNOUT5  DEC   R5
        JNE   LNOUT5
        MOV   @SAVE,R13     RESTORE REGISTERS
        MOV   @SAVE+2,R14
        RTWP                RETURN
*
*       ROUTINE TO TEST VALUE OF I/O COUNTER
*
TESTC   LI    R12,76        R12=76
        C     @COUNT,R12    TEST FOR COUNTER > 76
        JLE   TESTC1
        MOV   R12,@COUNT    RESET COUNTER TO 76
TESTC1  B     *R11          RETURN
*
*       ROUTINE TO SET BUFFER TO BLANKS
*
BLANK   LI    R4,>2020      R4=ASCII BLANK
        LI    R2,BUFFR      R2=ADDRESS OF BUFFER
BLANK1  MOVB  R4,*R2+       SET BYTE OF BUFFER TO BLANK
        CI    R2,COUNT      TEST FOR END OF BUFFER
        JNE   BLANK1
        CLR   @COUNT        SET I/O COUNTER TO 0
        RTWP                RETURN
*
*       ROUTINE TO SKIP SPACES FOR X FORMAT
*       ( NUMBER OF SPACES IN R4 )
*
SKIP    MOV   @>8(R13),R4   GET VALUE OF R4
```

```
        A     R4,@COUNT     ADD VALUE TO COUNTER
        BL    @TESTC        TEST FOR VALUE IN RANGE
        RTWP                RETURN
*
*       INPUT ROUTINE FOR A1 FORMAT
*       ( DESTINATION ADDRESS IN R4 )
*
INA1    MOV   @>8(R13),R4   GET VALUE OF R4
        INC   @COUNT        INCREMENT COUNTER
        BL    @TESTC        TEST VALUE
        CLR   *R4           CLEAR DESTINATION
        MOV   @COUNT,R2     R2=VALUE OF COUNTER
        MOVB  @BUFFR-1(R2),@1(R4) MOVE CHARACTER TO
*                           DESTINATION ADDRESS
        RTWP                RETURN
*
*       INPUT ROUTINE FOR A2 FORMAT
*       ( DESTINATION ADDRESS IN R4 )
*
INA2    MOV   @>8(R13),R4   GET DESTINATION ADDRESS
        INCT  @COUNT        INCREMENT COUNTER BY 2
        BL    @TESTC        TEST VALUE
        MOV   @COUNT,R2     R2=VALUE OF COUNTER
        MOVB  @BUFFR-1(R2),R5  PICK UP LOW ORDER BYTE
        SRL   R5,8
        MOVB  @BUFFR-2(R2),R5  PICK UP HIGH ORDER BYTE
        MOV   R5,*R4        STORE RESULT
        RTWP                RETURN
*
*       INPUT ROUTINE FOR I FORMAT
*       (DESTINATION ADDRESS IN R4, LENGTH IN R10)
*
INI     MOV   @>8(R13),R4   GET DESTINATION ADDRESS
        MOV   @>14(R13),R10 GET FIELD LENGTH
        CLR   R5            R5=0
        CLR   R8            R8=0
INI1    INC   @COUNT        INCREMENT COUNTER
        BL    @TESTC        TEST VALUE
        LI    R7,10         R7=10
        MPY   R7,R5         R6=R5*10
        MOV   R6,R5
        MOV   @COUNT,R2     R2=COUNTER VALUE
        MOVB  @BUFFR-1(R2),R6  R6=CHARACTER FROM BUFFER
        SRL   R6,8
        CI    R6,>2D        TEST FOR MINUS SIGN
        JNE   INI2
        MOV   R6,R8         RECORD MINUS SIGN FOR LATER USE
        CLR   R6            R6=0
INI2    ANDI  R6,>000F      REMOVE ALL BUT LOW ORDER NIBBLE
        A     R6,R5         UPDATE VALUE IN R5
        DEC   R10           DECREMENT LENGTH
        JNE   INI1
        CI    R8,0          TEST FOR NEGATIVE NUMBER
        JEQ   INI3
```

```
        NEG  R5             NEGATE VALUE
INI3    MOV  R5,*R4         STORE VALUE
        RTWP                RETURN
*
*       INTEGER INPUT FOR U FORMAT
*       ( DESTINATION ADDRESS IN R4 )
*
INU     MOV  @>8(R13),R4    GET DESTINATION ADDRESS
        CLR  R5             R5=0
        CLR  R9             R9=0
INU1    MOV  @COUNT,R2      R2=COUNT VALUE
        CI   R2,76          TEST FOR END OF BUFFER
        JEQ  INU3
        INC  @COUNT         INCREMENT COUNTER
        BL   @TESTC         TEST VALUE
        MOVB @BUFFR(R2),R8  R8=CHARACTER FROM BUFFER
        SRL  R8,8
        CI   R8,>20         TEST FOR BLANK
        JEQ  INU1
        CI   R8,>2D         TEST FOR MINUS SIGN
        JNE  INU2
        MOV  R8,R9          RECORD MINUS SIGN FOR LATER USE
        JMP  INU1
INU2    LI   R7,10          R7=10
INU3    MPY  R7,R5          R6=R5*10
        MOV  R6,R5
        ANDI R8,>000F       REMOVE ALL BUT LOW ORDER NIBBLE
        A    R8,R5          UPDATE VALUE IN R5
        MOV  @COUNT,R2      R2=COUNTER VALUE
        CI   R2,76          TEST FOR END OF BUFFER
        JEQ  INU4
        INC  @COUNT         INCREMENT COUNTER
        BL   @TESTC         TEST VALUE
        MOVB @BUFFR(R2),R8  R8=CHARACTER FROM BUFFER
        SRL  R8,8
        MOV  R8,R10         TEST FOR NUMERIC
        ANDI R10,>00F0
        CI   R10,>0030
        JEQ  INU3
INU4    CI   R9,0           TEST FOR NEGATIVE NUMBER
        JEQ  INU5
        NEG  R5             NEGATE VALUE IN R5
INU5    MOV  R5,*R4         STORE NUMBER
        RTWP                RETURN
*
*       OUTPUT ROUTINE FOR A1 FORMAT
*       ( SOURCE ADDRESS IN R4)
*
OUTA1   MOV  @>8(R13),R4    GET SOURCE ADDRESS
        INC  @COUNT         INCREMENT COUNTER
        BL   @TESTC         TEST VALUE
        MOV  @COUNT,R2      R2=COUNTER VALUE
        MOVB @1(R4),@BUFFR-1(R2)  STORE CHARACTER IN BUFFER
        RTWP                RETURN
```

```
*
*       OUTPUT ROUTINE FOR A2 FORMAT
*       ( SOURCE ADDRESS IN R4 )
*
OUTA2   MOV   @>8(R13),R4    GET SOURCE ADDRESS
        INCT  @COUNT         INCREMENT COUNTER BY 2
        BL    @TESTC         TEST VALUE
        MOV   @COUNT,R2      R2=COUNTER VALUE
        MOVB  *R4+,@BUFFR-2(R2)  STORE HIGH ORDER BYTE
        MOVB  *R4,@BUFFR-1(R2)   STORE LOW ORDER BYTE
        RTWP                 RETURN
*
*       OUTPUT ROUTINE FOR I FORMAT
*       ( DESTINATION ADDRESS IN R4, LENGTH IN R10 )
*
OUTI    MOV   @>8(R13),R4    GET DESTINATION ADDRESS
        MOV   @>14(R13),R10  GET FIELD LENGTH
        MOV   *R4,R5         GET VALUE
        ABS   R5             USE ABSOLUTE VALUE
        ..OV  @COUNT,R1      R1=COUNTER VALUE
        A     R10,@COUNT     INCREMENT COUNTER BY FIELD LENGTH
        BL    @TESTC         TEST VALUE
        MOV   @COUNT,R2      R2=COUNTER VALUE
        LI    R7,10          R7=10
OUTI1   MOV   R5,R6          SET UP R5,R6 FOR DIVISION
        ..LR  R5
        DIV   R7,R5          R5=R6/10, R6=REMAINDER
        C     R1,R2          TEST FOR END OF FIELD
        JEQ   OUTI2
        AI    R6,>30         ADD ACSII OFFSET TO R6
        SRC   R6,8
        MOVB  R6,@BUFFR-1(R2)  STORE CHARACTER IN BUFFER
        DEC   R2
        CI    R5,0           TEST FOR LAST DIGIT
        JNE   OUTI1
        MOV   *R4,R5         TEST SIGN
        JGT   OUTI4
        JEQ   OUTI4
        C     R1,R2          TEST FOR END OF FIELD
        JEQ   OUTI2
        LI    R6,>2D2D       INSERT MINUS SIGN
        MOVB  R6,@BUFFR-1(R2)
        JMP   OUTI4
OUTI2   LI    R6,>2A2A       VALUE ERROR, INSERT ASTERISKS
OUTI3   INC   R1
        MOVB  R6,@BUFFR-1(R1)
        C     R1,@COUNT
        JNE   OUTI3
OUTI4   RTWP                 RETURN
*
*       ROUTINE TO MOVE A STRING TO THE I/O BUFFER UNTIL
*       A ZERO BYTE IS ENCOUNTERED IN THE CHARACTER STRING
*       ( STARTING ADDRESS OF STRING IN R4 )
*
```

```
STRIN  MOV  @>8(R13),R4   GET STARTING ADDRESS
STRINO INC  @COUNT        INCREMENT COUNTER
       BL   @TESTC        TEST VALUE
       MOV  @COUNT,R2     R2=COUNTER VALUE
       CLR  R3            R3=0
       CB   *R4,R3        TEST FOR ZERO BYTE
       JEQ  STRN1
       MOVB *R4+,@BUFFR-1(R2)  MOVE CHARACTER TO BUFFER
       JMP  STRINO
STRN1  RTWP               RETURN
*
*      TMS 9900 FORTRAN CROSS COMPILER
*      ARITHMETIC FUNCTION ROUTINES
*
*****************************************************************
*
*      EXPONENTIATOR ROUTINE TO RAISE VALUE IN R0 TO POWER
*      SPECIFIED BY PARAMETER AT RETURN ADDRESS.
*
EXPN   CLR  R10           R10=0
       MOV  R0,R8         R8=BASE
       MOV  *R11,R2       R2=PARAMETER ADDRESS
       MOV  *R2,R12       R12=EXPONENT
       INCT R11
       LI   R0,1          R0=1
       MOV  R0,R15        R15=1
       CI   R8,-1         TEST FOR NEGATIVE BASE
       JGT  EXPN1
       COC  R15,R12       TEST FOR LSB OF R12 = 1
       JNE  EXPN1
       DEC  R10           R10=-1
EXPN1  ABS  R8            R8=ABSOLUTE VALUE OF R8
EXPN2  CI   R12,0         TEST FOR R12 = 0
       JEQ  EXPN4
       COC  R15,R12       TEST FOR LSB OF R12 = 1
       JNE  EXPN3
       MPY  R8,R0         R0=R0*R8
       MOV  R1,R0
EXPN3  MPY  R8,R8         SQUARE R8
       MOV  R9,R8
       SRL  R12,1         R12=R12/2
       JMP  EXPN2
EXPN4  CI   R10,-1        TEST FOR NEGATIVE
       JNE  EXPN5
       NEG  R0
EXPN5  B    *R11          RETURN
*
*      SIGNED DIVISION ROUTINE TO DIVIDE  IN R0 BY VALUE
*      VALUE OF PARAMETER AT RETURN ADDRESS.
*
DIVD   LI   R10,1         R10=1
       MOV  *R11,R2       R2=PARAMETER ADDRESS
       MOV  *R2,R8        R8=DIVISOR
       INCT R11
```

```
        CI    R8,-1         TEST FOR NEGATIVE DIVISOR
        JGT   DIVD1
        NEG   R8            R8=-R8
        NEG   R10           R10=-R10
DIVD1   CI    R0,-1         TEST FOR NEGATIVE DIVIDEND
        JGT   DIVD2
        NEG   R0            R0=-R0
        NEG   R10           R10=-R10
DIVD2   MOV   R0,R1         SET UP 32 BIT DIVIDEND
        CLR   R0
        DIV   R8,R0         R0=R0/R8
        CI    R10,-1        TEST FOR NEGATIVE RESULT
        JNE   DIVD3
        NEG   R0
DIVD3   B     *R11          RETURN
*
*       SIGNED MULTIPLICATION ROUTINE TO MULTIPLY VALUE IN R0
*       BY VALUE OF PARAMETER AT RETURN ADDRESS.
*
MULT    LI    R10,1         R10=1
        MOV   *R11,R2       R2=PARAMETER ADDRESS
         OV   *R2,R8        R8=MULTIPLIER
        INCT  R11
        CI    R8,-1         TEST FOR NEGATIVE
        JGT   MULT1         MULTIPLIER
        NEG   R8            R8=-R8
        NEG   R10           R10=-R10
MULT1   CI    R0,-1         TEST FOR NEGATIVE MULTIPLICAND
        JGT   MULT2
        NEG   R0            R0=-R0
        NEG   R10           R10=-R10
MULT2   MPY   R8,R0         MULTIPLY ABSOLUTE VALUES
        MOV   R1,R0
        CI    R10,-1        TEST FOR NEGATIVE RESULT
        JNE   MULT3
        NEG   R0            R0=-R0
MULT3   B     *R11          RETURN
*
*       SUBSCRIPTOR ROUTINE TO SUBSCRIPT ARRAY ADDRESS
*       LOCATED AT RETURN ADDRESS WITH THE VALUE OF THE
*       PARAMETER LOCATED AT THE RETURN ADDRESS + 2 AND
*       LOAD THE VALUE INTO R0.
*
SBSC    MOV   *R11,R10      R10=ARRAY ADDRESS
        INCT  R11
        MOV   *R11,R2       R2=SUBSCRIPT ADDRESS
        MOV   *R2,R8        R8=SUBSCRIPT VALUE
        INCT  R11
        SLA   R8,1          COMPUTE OFFSET
        DECT  R8
        A     R8,R10        COMPUTE ADDRESS
        MOV   *R10,R0       R0=ARRAY VALUE
        B     *R11
        END
```

APPENDIX C

SAMPLE RUN

```
EX COMP.CLIST 'RAND.FORT LOADDECK.DATA'
    1         SUBROUTINE RAND(IX1)
        C
        C     RANDOM NUMBER GENERATOR.
        C
        C     VALUES OF IX1 WILL RANGE FROM 1 TO 32767
        C
    2          DIMENSION N(32)
    3          DATA L/12345/, NDIM/32/, M2/16384/, LA/205/,
               *     LC/6925/
    4          DATA NRAN/23456/
    5          DATA N/22291,155,31814,26768,20735,17400,21861,
               *       1138,13116,23810,1089,21391,13235,27179,
               *       7912,31503,29998,6509,30399,1613,17736,
               *       15345,5454,20861,16503,19272,3742,29190,
               *       18043,24419,2604,12011                    /
    6          NDIV=2**10
    7          J=1+NRAN/NDIV
    8          IF(J) 175,175,180
    9     175  J=1-J
   10     180  IF(J-NDIM) 190,190,185
   11     185  J=J/NDIM
   12          GO TO 180
   13     190  NRAN=N(J)
   14          IX1=NRAN
   15          L=L*LA+LC
   16          IF(L/2-M2) 220,220,210
   17     210  L=(L-M2)-M2
   18     220  IF(L) 230,240,240
   19     230  L=(L+M2)+M2
   20     240  N(J)=L
   21          RETURN
   22          END


 EXECUTION ADDRESS=018E
        C
    1         SUBROUTINE PLOT(NAVG,NTOT)
        C
        C     ROUTINE TO PLOT DISTRIBUTIONS OF NTOT NUMBERS
        C     COMPUTED AS AVERAGES OF NAVG RANDOM NUMBERS
        C     EACH ON A 10 X 10 GRAPH.
```

```
      C
    2       DIMENSION IDIST(10),LINE(10)
    3       COMMON IN,LP
      C
      C     SET PLOT CHARACTERS
      C
    4       DATA IAST,IBLNK /'**','  '/
      C
      C     INITIALIZE DISTRIBUTION DATA TO ZERO AND
      C     LINE TO BLANK
      C
    5       NMAX=0
    6       DO 100 I=1,10
    7         LINE(I)=IBLNK
    8   100   IDIST(I)=0
      C
      C     LOOP TO RECORD DISTRIBUTIONS
      C
    9       DO 300 I=1,NTOT
      C
      C     COMPUTE NUMBER AS AVERAGE OF RANDOM NUMBERS
      C
   10       NUM=0
   11       DO 200 J=1,NAVG
   12         CALL RAND(JRAN)
   13   200   NUM=NUM+JRAN/NAVG
      C
      C     RECORD DISTRIBUTION
      C
   14       ISUB=NUM/3277+1
   15       IDIST(ISUB)=IDIST(ISUB)+1
   16       IF(IDIST(ISUB)-NMAX) 300,300,250
   17   250 NMAX=IDIST(ISUB)
   18   300 CONTINUE
      C
      C     MAKE PLOT OF DISTRIBUTION
      C
   19       NDIV=NMAX/10
   20       DO 1000 I=1,10
   21         NMAX=NMAX-NDIV
   22         DO 500 J=1,10
   23           IF(IDIST(J)-NMAX) 500,500,400
   24   400     LINE(J)=IAST
   25   500     CONTINUE
      C
      C     OUTPUT LINE OF GRAPH
      C
   26  1000   WRITE(LP,1) LINE(1),LINE(2),LINE(3),LINE(4),
          * LINE(5),LINE(6),LINE(7),LINE(8),LINE(9),
   27     1 FORMAT(10A2)
          * LINE(10)
   28       RETURN
   29       END
```

```
 EXECUTION ADDRESS=0332
       C
       C     MAIN PROGRAM TO READ A NUMBER (N) FROM A
       C     TERMINAL, CALL A ROUTINE TO PLOT DISTRIBUTIONS
       C     OF 1000 NUMBERS CALCULATED AS AVERAGES OF N
       C     RANDOM NUMBERS EACH AND REPEAT THE PROCESS
       C     UNTIL A ZERO IS ENTERED.
       C
    1         COMMON IN,LP
       C
       C     SET I/O UNIT NUMBERS
       C
    2         IN=0
    3         LP=0
       C
       C     INITIALIZE RANDOM NUMBER GENERATOR
       C
    4         WRITE(LP,1)
    5       1 FORMAT(
             *   ' ENTER 4-DIGIT NUMBER FOR INITIALIZATION')
    6         READ(IN,2) N
    7       2 FORMAT(I2)
    8         DO 50 I=1,N
    9      50   CALL RAND(J)
       C
       C     READ NUMBER OF RANDOM VALUES FOR AVERAGE
       C
   10     100 WRITE(LP,3)
   11       3 FORMAT(' ENTER N')
   12         READ(IN,4) N
   13       4 FORMAT(I2)
       C
       C     TEST FOR LAST N
       C
   14         IF(N) 1000,1000,200
   15     200 CONTINUE
       C
       C     CALL ROUTINE TO PLOT DISTRIBUTION

   16         CALL PLOT(N,1000)
   17         GO TO 100
   18    1000 STOP
   19         END


 EXECUTION ADDRESS=061C
READY
L LOADDECK.DATA NONUM
IKJ52827I  LOADDECK.DATA
 0132: 57 13 00 9B 7C 46 68 90 50 FF 43 F8 55 65 04 72
 0142: 33 3C 5D 02 04 41 53 8F 33 B3 6A 2B 1E E8 7B 0F
 0152: 75 2E 19 6D 76 BF 06 4D 45 48 3B F1 15 4E 51 7D
 0162: 40 77 4B 48 0F 9E 72 06 46 7B 5F 63 0A 2C 2E EB
 0172: 30 39 00 20 40 00 00 CD 1B 0D 5B A0 00 00 00 02
```

```
0182: 00 0A 00 00 00 01 00 00 0D 0A 00 00 02 2B 00 02
0192: C8 0B 01 10 C0 20 01 80 06 A0 F2 E8 01 82 C8 00
01A2: 01 7E C0 20 01 7C 06 A0 F3 26 01 7E C8 00 01 30
01B2: C0 20 01 86 A0 20 01 30 C8 00 01 84 C0 20 01 84
01C2: 15 05 13 02 04 60 01 D2 04 60 01 D2 04 60 01 DE
01D2: C0 20 01 86 60 20 01 84 C8 00 01 84 C0 20 01 84
01E2: 60 20 01 74 C8 00 01 88 C0 20 01 88 15 05 13 02
01F2: 04 60 02 10 04 60 02 10 04 60 01 FE C0 20 01 84
0202: 06 A0 F3 26 01 74 C8 00 01 84 04 60 01 DE 06 A0
0212: F3 80 01 32 01 84 C8 00 01 7C C0 20 01 7C 02 0A
0222: FF FC A2 A0 01 10 C2 AA 00 02 C2 1A C6 80 C0 20
0232: 01 72 06 A0 F3 54 01 78 A0 20 01 7A C8 00 01 72
0242: C0 20 01 72 06 A0 F3 26 01 80 60 20 01 76 C8 00
0252: 01 88 C0 20 01 88 15 05 13 02 04 60 02 78 04 60
0262: 02 78 04 60 02 68 C0 20 01 72 60 20 01 76 60 20
0272: 01 76 C8 00 01 72 C0 20 01 72 15 05 13 02 04 60
0282: 02 8C 04 60 02 9C 04 60 02 9C C0 20 01 72 A0 20
0292: 01 76 A0 20 01 76 C8 00 01 72 C0 20 01 72 C0 A0
02A2: 01 84 0A 12 06 42 C8 80 01 32 C2 E0 01 10 04 5B
02D6: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
02E6: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
02F6: 00 00 00 00 00 00 00 00 00 00 00 00 00 2A 2A 20
0306: 00 00 00 00 00 00 00 01 00 0A 00 00 00 00 00 00
0316: 00 00 0C CD 00 00 00 00 00 02 00 03 00 04 00 05
0326: 00 06 00 07 00 08 00 09 0D 0A 00 00 02 2B 00 04
0336: C8 0B 02 B4 C0 20 03 08 C8 00 03 06 C8 20 03 0C
0346: 03 0A C0 20 03 04 C0 A0 03 0A 0A 12 06 42 C8 80
0356: 02 EE C0 20 03 08 C0 A0 03 0A 0A 12 06 42 C8 80
0366: 02 DA 05 A0 03 0A 88 20 03 0A 03 0E 15 02 04 60
0376: 03 48 C8 20 03 0C 03 0A C0 20 03 08 C8 00 03 10
0386: C8 20 03 0C 03 12 06 A0 01 8E 03 14 02 0A FF FA
0396: A2 A0 02 B4 C2 AA 00 02 C8 1A 02 B2 C0 20 03 14
03A6: 06 A0 F3 26 02 B2 C8 00 02 D4 C0 20 03 10 A0 20
03B6: 02 D4 C8 00 03 10 05 A0 03 12 02 0A FF FA A2 A0
03C6: 02 B4 C2 AA 00 02 C8 1A 02 B6 88 20 03 12 02 B6
03D6: 15 02 04 60 03 8C C0 20 03 10 06 A0 F3 26 03 18
03E6: A0 20 03 0C C8 00 03 16 06 A0 F3 80 02 DA 03 16
03F6: A0 20 03 0C C0 A0 03 16 0A 12 06 42 C8 80 02 DA
0406: 06 A0 F3 80 02 DA 03 16 60 20 03 06 C8 00 03 1A
0416: C0 20 03 1A 15 05 13 02 04 60 04 36 04 60 04 36
0426: 04 60 04 2A 06 A0 F3 80 02 DA 03 16 C8 00 03 06
0436: 05 A0 03 0A 02 0A FF FA A2 A0 02 B4 C2 AA 00 04
0446: C8 1A 02 B6 88 20 03 0A 02 B6 15 02 04 60 03 7E
0456: C0 20 03 06 06 A0 F3 26 03 0E C8 00 03 1C C8 20
0466: 03 0C 03 0A C0 20 03 06 60 20 03 1C C8 00 03 06
0476: C8 20 03 0C 03 12 06 A0 F3 80 02 DA 03 12 60 20
0486: 03 06 C8 00 03 1A C0 20 03 1A 15 05 13 02 04 60
0496: 04 B0 04 60 04 B0 04 60 04 A0 C0 20 03 02 C0 A0
04A6: 03 12 0A 12 06 42 C8 80 02 EE 05 A0 03 12 88 20
04B6: 03 12 03 0E 15 02 04 60 04 7C C8 20 02 D8 00 EE
04C6: 02 05 00 80 02 06 F1 26 04 05 02 04 02 EE C2 20
04D6: 03 0C 0A 18 06 48 A1 08 02 06 F2 4E 04 05 02 04
04E6: 02 EE C2 20 03 1E 0A 18 06 48 A1 08 02 06 F2 4E
04F6: 04 05 02 04 02 EE C2 20 03 20 0A 18 06 48 A1 08
```

```
0506: 02 06 F2 4E 04 05 02 04 02 EE C2 20 03 22 0A 18
0516: 06 48 A1 08 02 06 F2 4E 04 05 02 04 02 EE C2 20
0526: 03 24 0A 18 06 48 A1 08 02 06 F2 4E 04 05 02 04
0536: 02 EE C2 20 03 26 0A 18 06 48 A1 08 02 06 F2 4E
0546: 04 05 02 04 02 EE C2 20 03 28 0A 18 06 48 A1 08
0556: 02 06 F2 4E 04 05 02 04 02 EE C2 20 03 2A 0A 18
0566: 06 48 A1 08 02 06 F2 4E 04 05 02 04 02 EE C2 20
0576: 03 2C 0A 18 06 48 A1 08 02 06 F2 4E 04 05 02 04
0586: 02 EE C2 20 03 0E 0A 18 06 48 A1 08 02 06 F2 4E
0596: 04 05 02 06 F0 C0 04 05 05 A0 03 0A 88 20 03 0A
05A6: 03 0E 15 02 04 60 04 6A C2 E0 02 B4 04 5B
05D8: 00 00 00 00 00 00 00 01 00 00 03 E8 0D 0A 00 00
05E8: 20 45 4E 54 45 52 20 34 2D 44 49 47 49 54 20 4E
05F8: 55 4D 42 45 52 20 46 4F 52 20 49 4E 49 54 49 41
0608: 4C 49 5A 41 54 49 4F 4E 00 00 20 45 4E 54 45 52
0618: 20 4E 00 00 C0 20 05 D8 C8 00 02 D6 C0 20 05 D8
0628: C8 00 02 D8 C8 20 02 D8 00 EE 02 05 00 80 02 06
0638: F1 26 04 05 02 04 05 E8 02 06 F2 CA 04 05 02 06
0648: F0 C0 04 05 C8 20 02 D6 00 EE 02 05 00 80 02 06
0658: F0 82 04 05 02 04 05 DA 02 0A 00 02 02 06 F1 82
0668: 04 05 C8 20 05 DE 05 DC 06 A0 01 8E 05 E0 05 A0
0678: 05 DC 88 20 05 DC 05 DA 15 02 04 60 06 70 C8 20
0688: 02 D8 00 EE 02 05 00 80 02 06 F1 26 04 05 02 04
0698: 06 12 02 06 F2 CA 04 05 02 06 F0 C0 04 05 C8 20
06A8: 02 D6 00 EE 02 05 00 80 02 06 F0 82 04 05 02 04
06B8: 05 DA 02 0A 00 02 02 06 F1 82 04 05 C0 20 05 DA
06C8: 15 05 13 02 04 60 06 E4 04 60 06 E4 04 60 06 D8
06D8: 06 A0 03 32 05 DA 05 E2 04 60 06 86 04 20 FF FC
READY
```

APPENDIX D

BNF SYNTAX OF FORTRAN STATEMENT TEXT

```
stmt ::= read_stmt | write_stmt | format_stmt |
         assignment_stmt | dimension_stmt | data_stmt |
         common_stmt | go_to_stmt | if_stmt | do_stmt |
         stop_stmt | end_stmt | call_stmt |
         subroutine_stmt | return_stmt | continue_stmt

stop_stmt ::= STOP

end_stmt ::= END

return_stmt ::= RETURN

continue_stmt ::= CONTINUE

read_stmt ::= READ( int , int_const ) io_list

write_stmt ::= WRITE( int , int_const ) io_list

io_list ::= var | var , io_list

var ::= int_id | int_id ( int )

format_stmt ::= FORMAT( format_list )

format_list ::= format_item | format_item , format_list

format_item ::= format_type | int_const format_type |
                string | / | int_const X

format_type ::= A1 | A2 | I int_const | U

string ::= ' char_string ' | int_const H char_string

assignment_stmt ::= var = rhs

rhs ::= var | rhs op rhs | ( rhs ) | const

op ::= + | - | * | / | **

dimension_stmt ::= DIMENSION dimension_list

dimension_list ::= dimension_item |
```

```
                    dimension_item , dimension_list

dimension_item ::= int_id ( int_const )

data_stmt ::= DATA data_block

data_block ::= id_list / data_list / |
               id_list / data_list / data_block

id_list ::= int_id | int_id , id_list

data_list ::= data_item | data_item , data_list

data_item ::= const | string

common_stmt ::= COMMON id_list

go_to_stmt ::= GO TO int_const

if_stmt ::= IF( rhs ) int_const , int_const , int_const

do_stmt ::= DO int_id = int , int |
            DO int_id = int , int , int

subroutine_stmt ::= SUBROUTINE id |
                    SUBROUTINE id ( subroutine_list )

subroutine_list ::= int_id | int_id , subroutine_list

call_stmt ::= CALL id | CALL id ( call_list )

call_list ::= const | int_id | const , call_list |
              int_id , call_list

int_const ::= pos_num | pos_num num_list

pos_num ::= 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

num_list ::= num | num num_list

num ::= 0 | pos_num

const ::= 0 | int_const | - int_const

int_id ::= int_alpha | int_alpha id

id ::= alpha | id num | id alpha

int_alpha ::= I | J | K | L | M | N

alpha ::= A | B | C | D | E | F | G | H | int_alpha | O |
          P | Q | R | S | T | U | V | W | X | Y | Z

int ::= int_const | int_id
```

```
char_string ::= char ¦ char char_string

char ::= alpha ¦ num ¦ blank ¦ ! ¦ @ ¦ # ¦ $ ¦ & ¦ * ¦
         ( ¦ ) ¦ - ¦ + ¦ = ¦ ; ¦ ' ¦ , ¦ . ¦ /
```

Supplemental notes:

1. The range of int_const is from 1 to 32767.
2. The range of const is from -32768 to 32767.
3. The maximum length of id and int_id is 6 characters.
4. If string is defined using int_const H char_string, then the value of int_const must be equal to the length of char_string.

APPENDIX E

GLOSSARY

ASCII - American Standard Code for Information Interchange.

Cross compiler - A compiler that executes on one computer and produces object code for a different computer.

EBCDIC - Extended Binary Coded Decimal Information Code.

Finite state automaton - A parser that consists of a finite number of states and recognizes input strings by making transitions between states until a final state is reached.

Grammar - A set of definitions which specify the sequences of characters that form allowable programs in a language.

Hex deck - File of hexadecimal records of object code suitable for loading onto the object computer.

Hexadecimal - Base 16 number system using the characters A through F to represent the digits with values 10 through 15.

Host computer - Machine on which the cross compiler executes.

Intermediate code - Code generated by the compiler for internal representation of FORTRAN statements.

Lexical analyzer - A subroutine whose function is to recog-

nize tokens in a string of input characters.

Object code - Hexadecimal machine code generated by the compiler to be loaded onto the object computer.

Object code templates - Hexadecimal machine code stored in the compiler for use in generation of object code.

Object computer - Machine for which the compiler generates object code.

Parse - Process of determining the syntactic structure associated with a string of input characters.

Parser - A subroutine whose function is to parse a statement in the source code.

Source code - FORTRAN program input to the compiler.

Syntax - Relation associating sentences of a language with the structure that is specified by the grammar for the language.

VITA

Steven Roger Heard

Candidate for the Degree of

Master of Science

Thesis: A FORTRAN CROSS COMPILER FOR A TMS 9900 MICROCOMPUTER SYSTEM

Major Field: Computing and Information Sciences

Biographical:

Personal Data: Born in Winston-Salem, North Carolina, April 18, 1956, the son of Mr. and Mrs. Burton E. Heard.

Education: Graduated as valedictorian from Dale High School, Dale, Oklahoma, in May, 1973; received the Bachelor of Science degree in Computing and Information Sciences from Oklahoma State University, Stillwater, Oklahoma, in December, 1977; completed the requirements for the Master of Science degree in July, 1979, at Oklahoma State University.

Professional Experience: Employed by Oklahoma State University as a Student Assistant from September, 1977 to December, 1977; employed by Oklahoma State University as a Graduate Assistant from January, 1978 to May, 1979.