INTERACTIVE CUSTOM FUNCTION DEFINITION

FOR THE ENPORT BOND-GRAPH PROCESSOR

By

Charles R. Sherman

A THESIS

Submitted to
Michigan State University
in partial fulfillment of the requirements
for the degree of

MASTER OF SCIENCE

Department of Mechanical Engineering

1987

ABSTRACT

INTERACTIVE CUSTOM FUNCTION DEFINITION
FOR THE ENPORT BOND-GRAPH PROCESSOR

By

Charles R. Sherman

The ability to design custom function definitions for bond-graph elements to supplement the standard function library can make a bond-graph processor much more versatile for the user. This thesis presents an enhancement to ENPORT, an existing bond-graph processor, and allows multiple custom function definitions to be written interactively by the user, without the need to leave the processor environment. The method scans the function definition and builds a control stream of commands which is interpreted during the bond-graph solution.

ACKNOWLEDGEMENTS

I would like to thank my major professor, Dr. Ronald Rosenberg, for his guidance and for his great patience during this work. The effort extended over many more semesters and masters research credits than either of us imagined.

To Guy Allen for his help with the installation of this work into ENPORT, and to Ace Sannier for his help in making source code listings available, I extend thanks.

I also want to recognize the contributions of my former major professor, Dr. M. G. Keeney, for his unflagging efforts to instill in us some of the aspects of compiler design. Both he and my compiler partner, Dr. Zane Mottler, might see similarities in the approach I have taken in this work and the approach we used in the compiler series course. Dr. Keeney is to be especially thanked for making available a grammar analyzer which automated the process of developing a simple precedence grammar for the parser. (I also have never again spelled grammar with an "e.")

Finally, I would like to thank my wife, Debbie, for her love and proofreading. She has learned a lot about program execution, matrix multiplication and developing patience.

TABLE OF CONTENTS

LIST OF FIGURES

Chapter 1

INTRODUCTION

The goal of this thesis was to create a method for an ENPORT user to write his own function definition for a bond-graph model node [1]. It is presumed that the reader is familiar with this modeling technique as well as the ENPORT computer program [2], which can process the bond-graph as input and provide the dynamic system response.

Prior to this work, a user selected functions from a pre-defined library and could only set various constants like amplitude, phase, frequency, etc. Building a customized "external" function was done by writing a new FORTRAN subprogram. This had to be compiled and linked with the rest of ENPORT to make the new external function available for the bond-graph solution. This compile-and-link process could only be done by someone with knowledge of the particular hardware's compiler and linker.

As a result of this work, the ENPORT user is now able to create his own external function definition without the need to leave the ENPORT environment, or to write, compile and link a FORTRAN subprogram. The external function is supported through the use of menus and prompts, allowing easy definition of the function. Each definition is saved and

restored with the problem description and can be archived separately for use in a different bond-graph model.

1.1 Overview

This enhancement affects ENPORT primarily in two areas: (1) during formulation of the system equations and (2) during their solution. (Detailed discussions of these two areas are in chapters two and three, respectively.) In the first area, the program interacts directly with the user to create an external function definition that becomes the system equation for a bond-graph node. Once the function is properly defined, the user moves on to specify the next node equation, which can also be an external function. Up to ten external functions may be defined at one time for a particular problem. This is not a constraint imposed by the method, but rather a space consideration when allocating memory.

After specification of the system equations is complete, ENPORT can solve the equations for the transient response. During the solution, each system equation is repeatedly evaluated. Equation types from the standard library have been compiled and linked with ENPORT and are in object code form. External functions are not in object code form. They are reduced to a series of operations at the level of an assembly language. This transformation of the external function occurs during its definition. The resulting series of assembler-like instructions is interpreted many times during the solution, once at each time-step. The second area of this work is an interpreter, which evaluates the function for the bond-graph solution.

Due to the complexity of reducing user statements to assembler-like instructions and the overhead of user dialog, the first area consumed much more time and effort than the second area. This complexity is reflected in the relative amount of source code written to implement each of these areas. (Source code files are described in Appendix H.)

1.2 Menus

All interaction with the user is controlled through menus and prompts. (See the sample sessions in Appendices A1, A2 and A3.) Two menus are available, each providing a "Help" panel. The first presents the main options for the external function processor. Of the ten options only two ever need to be used: (1) to create a new function definition and (2) to exit the external function processor. However, two other options can be useful with the function definition. One displays a generic form of the node equation which shows the output bond variable and the input bond variable(s). These variables must be used when writing the external function definition. The other allows a current definition to be edited. Selecting the edit option brings up the second menu used in this work: the external function editor options. It contains seven options, which include four kinds of line editing, a save of the edited function and an exit from the editor. The remaining six options on the external function processor menu, excluding help, deal with archived functions. This work has provided a feature allowing an external function definition created for one problem to be archived and then restored for another problem. Since it is unlikely that the input and output variables would be the same for two different problems, the restored function must

be edited. The editor is invoked automatically after a function is restored from archive. This ability to preserve definitions between problems and sessions of ENPORT can save tedious retyping and potential errors when using the same function in several different bond-graph models or when exercising alternate user-defined functions for the same node.

1.3 Features

In addition to the function archive ability described above, the work has included the potential to define parameters, which may then be referenced as constants in any function definition. The only parameter currently defined this way is PI which is set to 3.14159. PI can be referenced in the definition statements and the value 3.14159 will be used during the solution. It is not expected that the user would make these additions, but the ENPORT support programmer may add other such parameters, if desired. The procedure to make additions is discussed thoroughly in Chapter 5, Section 1.

A simple precedence grammar is used to parse FORTRAN-like definition statements input by the user. The full grammar is described in Appendix D1. Statements may begin in any column since no labels or continuation statements are allowed. All of the FORTRAN algebraic operators (+, -, *, /, **, =) may be used, as well as any of the functions in Figure 1. The double-precision versions of these functions are implemented by the interpreter in order to be consistent with the precision maintained elsewhere in ENPORT.

ABS	LOG
ACOS	LOG10
AINT	MAX
ANINT	MIN
ASIN	MOD
ATAN	SIGN
ATAN2	SIN
COS	SINH
COSH	SQRT
DIM	TAN
EXP	TANH

Figure 1. FORTRAN Functions

Also available is the IF-THEN-ELSEIF-ELSE-ENDIF statement. This statement must start with IF, must have a THEN clause and must end with ENDIF. The ELSEIF and ELSE clauses are optional. Logical operators (.AND., .OR. and .NOT.) may be used with the relational operators (.EQ., .NE., .GT., .GE., .LT. and .LE.) to build relational expressions for the IF and ELSEIF clauses. A brief introduction to the syntax for entering definition statements appears in Appendix A and a complete formal language specification appears in Appendix B.

During lexical analysis of user input, an integer value is associated with each input character. This integer identifies the input character position in the character set accepted by this work. Because machine independency was desired, intrinsic FORTRAN functions, which would return the character position in the system collating sequence, were not used. These returned results may not be consistent between machine architectures. Instead, a character string constant called COLATE was defined and used to establish an integer positional value for each character in the character set. Use of the intrinsic function INDEX with the string COLATE returns a positive value for every character accepted

by the work and zero for every character not accepted. This method guarantees machine independence when identifying a positional integer value for each input character.

Another feature required of this work was that a valid external function be in place when the user leaves the external function processor environment. That is, if the user requests an external function type but fails to complete a successful definition, a default function definition will be established. This was accomplished with a check for a valid function after the user selects the exit option from the external function processor menu. If a valid function is found, the exit proceeds, and control returns to ENPORT. If no valid function is found, and the user insists on exiting, a default function sets the output bond variable to zero.

A special feature of this work and one which was of great value during development is a trace option. Debug-aiding output can be generated and controlled through the use of trace flags. This feature is intended for the ENPORT support programmer only. Extensive traces of the lexical analysis and parsing operations can be written to a file for review or comparison with previous traces after a major program modification is made. Chapter 5, Section 3 presents a discussion of this option and Figure 5 lists the available trace flags.

Chapter 2

EXTERNAL FUNCTION DEFINITION

As the user enters each line of input to define the external function, the line is reduced to a stream of assembler-like instructions. There are three phases to this operation: (1) lexical analysis, (2) parsing and (3) code generation.

2.1 Lexical Analysis

Lexical analysis is the process of scanning a line of input one character at a time and identifying input tokens. An input token is a single element of the definition statement. Four types of input tokens exist: (1) operators, (2) variable names, (3) constants and (4) reserve words. A finite-state automaton (FSA) is used to scan through the input and accept or reject strings of characters as input tokens. The FSA (see Figure C1) is implemented in the table-driven subprogram LEX.

LEX is able to detect some user errors based on input character sequence. These are errors like (1) two operators with no intervening operand, (2) a variable name beginning with a digit or (3) any error that can be detected by looking ahead at the next input character. LEX contains many messages for display to the user if an error is encountered.

Any error in the input renders that line invalid, and the user is prompted to correct the error and reenter the line.

If LEX finds that the next input token is a variable name or a constant, the token is entered into a symbol table, which provides storage for the value during function evaluation. LEX distinguishes between real and integer constants, but the integers are converted to double-precision values in the symbol table. All arithmetic is double precision for the function evaluation. Once space is reserved in the symbol table, LEX changes the input tokens for variable names and constants to VAR and CON, respectively. These are "terminal" tokens used by the parsing grammar to represent variable names and constants. Input tokens for operators and reserve words are already terminal tokens in the grammar and are, therefore, unchanged by LEX.

```
SQUARE = BASE * 8
```

(A) User Input Line

Call Number	Input Token Found	Terminal Token Returned	Symbol Table Pointer Returned
1	SQUARE	VAR	1
2	=	=	None
3	BASE	VAR	2
4	*	*	None
5	8	CON	3

(B) Result of Five Calls to LEX

Entry Number	Entry Name	Entry Value
0001	SQUARE	0.0
0002	BASE	0.0
0003		8.0

(C) Symbol Table Entries Generated

Figure 2. Lexical Analysis of a User Input Line

Figure 2 shows the result of lexical analysis on a line of user input. Five successive calls to LEX are needed to process the user input line in Figure 2, Part A. At each call, LEX identifies the next input token and returns a terminal token of the parsing grammar and, if appropriate, a symbol table pointer. Figure 2, Part B shows these results and Figure 2, Part C shows the three entries generated in the symbol table.

2.2 Parsing

The series of terminal tokens identified by the lexical analysis is passed on to the parser, which is implemented in the subprogram PARSER. This work uses a "simple precedence" parsing grammar that defines new tokens called "nonterminals." The grammar rules (Appendix D1) are used to recognize a series of one or more "grammar tokens" (terminal and/or nonterminal tokens as listed in Appendix D2) that can be replaced by a single nonterminal. This replacement is called a "reduction," because usually several grammar tokens are replaced by one nonterminal. A precedence-relation table derived from the grammar defines one of the relations (<<, >> or ==) to be associated with each pair of grammar tokens. The table is only about half full, meaning that no relation exists for many pairs. If grammar tokens having no precedence relation ever appear together during the parsing of a user input line, indicative of a user syntax error, PARSER sends an error message to the user, rejects the input line, and prompts the user to correct the error and reenter the line.

The grammar has many rules which seem to do nothing but reduce one token to another. (See rules 44 and 45 in Appendix D1 for example.) These are necessary to remove conflicts from the precedence-relation table and make the grammar pairwise disjoint.

Figure 3 shows five steps in the parse of grammar tokens from the user input line of Figure 2. Figure 3, Part A shows the first four terminal tokens and the precedence relations between them. The symbol << at the far left is the relation between the bottom of the parsing stack and any grammar token. The appearance of >> signals the parser that a reduction can be made by applying the grammar rules to the series of tokens between the right-most pair of << and >> relations. Figure 3, Part B has the parse continuing after VAR is reduced to FACTOR and the last terminal token CON is added. The relation >> appears after CON because it represents the relation between any grammar token and the end of the user input line. Figure 3, Part C shows the reduction on CON to CEXFAC. The result of the next reduction, FACTOR == * == CEXFAC to CXPR, is shown in Figure 3, Part D. One more reduction is possible and the result is shown in Figure 3, Part E. At this point a statement, represented by the nonterminal SSTMT, has been successfully parsed. PARSER now prompts the user for more input and the user may continue or end the external function definition.

<< VAR == = << VAR >> *

(A) The precedence relations and tokens are stacked until a relation >> is encountered.

<< VAR == = << FACTOR == * << CON >>

(B) The new token FACTOR derived from VAR is stacked and followed by the rest of the input tokens.

<< VAR == = << FACTOR == * == CEXFAC >>

(C) The new token CEXFAC derived from CON is stacked with the appropriate precedence relations.

<< VAR == = == CXPR >>

(D) FACTOR == * == CEXFAC is replaced by CXPR with the appropriate precedence relations.

<< SSTMT

(E) VAR == = == CEXPR has been recognized as a valid statement and been replaced by SSTMT.

Figure 3. Parsing of Grammar Tokens from Figure 2

2.3 Internal Code Generation

The result of the external function definition is a series of assembler-like instructions called "internal code," which can be interpreted during the bond-graph solution. The generation of this internal code occurs simultaneously with the parsing described above in Section 2. Each step of the parse is marked by a reduction, that is, the application of one of the grammar rules to the tokens on the top of the parser stack. Of the fifty-two simple precedence grammar rules used in this work, thirty of those, when used in a reduction, also require internal code generation. The internal code is written to an array, which is made available to the interpreter for the bond-graph solution. Each internal code instruction is an integer four-tuple, consisting of one operator field followed by three operand fields. A list of all the internal code instructions and their descriptions appears in Appendix E.

Two of the reductions in the user input example of Figure 3 generate internal code. The reduction in Figure 3, Part D of FACTOR == * == CEXFAC to CXPR represents both the multiplication of two values from the symbol table and the storage of the result in a third value. The internal code generated is 0013 0002 0003 0004. The four-tuple shown is written as four integers separated by blanks for readability. Internally, the four-tuple is written as entry "i" in an array defined as OBJCOD(4,i). The first field contains the operator 0013, which means multiplication. The three operands 0002, 0003 and 0004 are pointers to symbol table entries that were made by LEX. Figure 4 contains the symbol table generated in Figure 2 by the calls to LEX.

Entry Number	Entry Name	Entry Value
0001	SQUARE	0.0
0002	BASE	7.0
0003		8.0
0004		0.0

Figure 4. Symbol Table after Parse

One more entry for storage of an intermediate value has been added by PARSER. Also, the user variable BASE has the value 7. This could be a parameter or a value previously assigned in the external function definition, or it might stand for a bond variable whose value is supplied by ENPORT. User input tokens SQUARE, BASE and 8 were entered into the symbol table by LEX at locations 0001, 0002 and 0003, respectively, and the intermediate value was entered by PARSER at location 0004. This multiply instruction then says to multiply the value stored at 0002 by the value stored at 0003 and place the result into the value for entry 0004.

Reduction VAR == = == CEXPR to SSTMT shown in Figure 3, Part E also generates internal code. The instruction generated is 0016 0004 0000 0001. Operator 0016 is the assignment operation and this instruction assigns or copies the value of symbol table entry 0004 to the value for entry 0001. The third field in this four-tuple (the second operand field) is 0000. This operand is not used by the assignment operation 0016 and, therefore, is set to 0000.

These two internal code instructions and the symbol table are all that are needed by the interpreter to accomplish the operation specified by the user in his input of Figure 2, Part A.

After processing all of the user input, the generated internal code itself is "post-processed" to remove label instructions and to modify branch and branch-on-condition instructions. These three instruction types are generated by the parser to implement the logic of the IF-THEN-ELSEIF-ELSE-ENDIF structure. Blocks of statements must be executed or jumped around depending on the result of evaluating a relational expression. This post-processing of the internal code makes the interpreter execute much more efficiently. If the internal code were not modified, the interpreter would have to decode every operator, even in the blocks to be skipped, in order to find the label instructions. Instead, the first pass of the post-processor removes the label instructions and retains location pointers for the first executable instruction after each label. During the second pass, the post-processor modifies every branch and branch-on-condition, replacing its label operand with the appropriate location pointer. Now, for the interpreter to execute a branch, the location pointer is merely copied from the branch operand into the interpreter's location counter.

Chapter 3

EXTERNAL FUNCTION EVALUATION

After completion of a successful external function definition, the FORTRAN-like statements input by the user have been parsed and reduced to a series of internal code instructions. This instruction set and the accompanying symbol table are used during the ENPORT bond-graph solution by an interpreter. The internal code is accepted as input and each instruction is performed by the interpreter implemented in subprogram INTERP.

3.1 Exchanging Values with ENPORT

ENPORT has established a vector containing storage for the inputs and outputs of all node equations in the bond-graph. The user references the input bond variables when writing the external function to define the equation output. As LEX identifies a valid bond-variable name in the user input line, the variable name is converted by LEX to the form VBL(offset), in which VBL is the ENPORT I/O vector and offset is the pointer into the vector where the value of the bond variable is stored. The parser generates the appropriate read-ENPORT or write-ENPORT instruction (operator 0023 or 0024, respectively) and the interpreter uses the offset to retrieve input values from the ENPORT I/O vector and to write output

values to it. These instructions carry the offset value as one of the operands in the instruction rather than a symbol table pointer. This imitates an "immediate" instruction in assembler.

3.2 Interpreter

The INTERP subprogram is fairly short, only about 330 lines including comments, and is completely self-contained. ENPORT calls INTERP to evaluate the external function and INTERP processes all of the internal code for the function without calling any other subprograms.

Because the bond-graph problem at hand may have more than one external function definition, each function is "loaded" to an array called MEMORY. Pointers to the start of the internal code for each function are maintained, and the interpreter initializes its location counter for MEMORY with this pointer. Symbol tables for each external function are loaded to another array and pointers are maintained for each of them. Logically, this is the same bookkeeping effort done by a system linker for loading object code programs.

After its location counter and symbol table offset have been initialized, INTERP evaluates the first internal code instruction. The twenty-four operators are numbered one through twenty-four in order to simplify use of a computed GO TO statement which implements operator evaluation. The appropriate action for each instruction is performed, referencing values in the symbol table or values in the array used for exchanging variables with ENPORT. Then the location counter is

incremented by one, and the process continues with the next instruction until a return (operator 0020) is found. One evaluation of the user's external function has now been completed for one time-step in the bond-graph solution, and control is returned to ENPORT.

The interpretive process described above involves a lot of computer processing overhead and one might expect that solution times would be extended when using an external function; however, initial comparisons between a bond-graph problem using a library function and the same problem using an external function indicate little difference in solution times. It is felt that the solution process as a whole is quite CPU intensive and, therefore, the added overhead of the interpreter has little effect on the total solution time. This might no longer be true in a problem involving a large number of time-steps or using several external functions. The task of evaluating solution times using external functions lies outside the scope of this work.

Chapter 4

INSTALLATION INTO ENPORT

Initial installation efforts concentrated on transferring execution from ENPORT to this work and modifying the external function routines to use the standard ENPORT I/O handlers. This also made the ENPORT dialog trace facility available for the remaining development. Installation was completed by providing routines to (1) initialize a new bond-graph problem, (2) read and write an external function definition for a saved problem and (3) clear an external function.

Once this installation was complete, the value exchanges with the ENPORT I/O vector, VBL, could now be fully tested. The external function processor can recognize bond variables and correctly locate them in VBL. Also, the variables T and TIME are recognized as read-only variables containing the time value set by ENPORT during bond-graph solution.

4.1 Bond-Graph Setup

Whenever ENPORT initializes to restore an old bond-graph problem from a problem file or to create a new problem, certain areas within this work must also be initialized. The routine CLREXT resets (1) MEMORY, (2) the symbol table array TMPVAL, (3) the array TEXT which holds each

line of user input and (4) LOADPT which holds pointers for each external function. This occurs once for each problem.

4.2 Reading or Writing a Saved Problem

Since ENPORT provides the ability to save a bond-graph problem description, this enhancement had to support that feature also. ENPORT optionally writes the node equations to a file, and if this option is selected for a problem having an external function, saves the user input lines. The subprogram EXTPTR returns the pointer to TEXT and the count of user input lines stored there for any node equation having an external function. Installation required a test for a function type of USERDEF, a call to EXTPTR and the writing of TEXT lines to the problem file.

When ENPORT restores a bond-graph problem, all of the text lines must be parsed again to generate internal code and the symbol table for the interpreter. The installation required testing for function type USERDEF, and loading user input lines defining the function into array EARRAY and setting ECOUNT equal to the number of lines read. A call to subprogram READEX causes lexical analysis and parse of all lines in EARRAY. This process exercises the same subprograms LEX and PARSER that are used during an interactive definition, but suppresses user dialog except for error messages. Since these defining statements were successfully accepted before they could be saved in the problem file, no errors should occur.

These file read-and-write procedures are repeated for each external function defined in a bond-graph problem.

4.3 Replacing an External Function

Once a node equation has been defined, the user has the option available to change it. If the original definition used an external function, this definition must be "unloaded" to release space in MEMORY, TEXT, the symbol table and LOADPT. This is done with one call to the subprogram ULDEXT. The only argument passed to ULDEXT is the number ENPORT assigned to the node. Each array is compressed after a definition is unloaded in order to provide the maximum available space for another external function. This work supports up to ten definitions simultaneously for one bond-graph problem, but user function sizes may exceed MEMORY, TEXT or symbol table array limits. These limitations are necessary simply to define finite sizes for program data structures and can be extended by the ENPORT support programmer.

Chapter 5

MAINTENANCE

The intent of this chapter is to provide enough information about several areas of this work to facilitate future maintenance. Included are details regarding parameter table expansion and two hidden main menu options for the external function processor. One of these options activates a trace facility, which has the capability to trace the lexical analysis and parse operations. The trace output is written to a file.

5.1 Parameters

During the initialization performed for every new external function definition, the symbol table is loaded with reserve words and the parameter names and values. This table, in subprogram ANFANG, consists of three data structures. The first, the integer NPARAM, is simply the number of active parameters in the table. The other two data structures are parallel one-dimensional arrays of length NPARAM. The first array, NPARM, is of type CHARACTER*6 and contains the name of the parameters that can be referenced within an external function. The second array, VPARM, is of type DOUBLE PRECISION and contains the value of each parameter named in NPARM. These three structures are local variables initialized with DATA statements and are not referenced outside ANFANG. The table is

expanded by adding the parameter names and values to NPARM and VPARM, respectively, and setting NPARAM to the new table size.

5.2 Hidden Menu Options

Two hidden options are available on the external function processor menu: L and T. Option L is described in this section, while option T is described in Section 3.

During the development and testing of this work, it was helpful to separate the tasks of creating, editing and loading an external function definition. Initially, the external function processor menu offered an option for each of these three operations. After development was complete, it proved more effective to automatically load after a successful create or edit and this greatly smoothed the flow of operations required of the user at the main menu. While the load procedure is automatic, the load option (option L) is still accepted from the main menu prompt. This could be useful during future maintenance of this work. The separation of loading from editing and creating has been maintained by keeping separate subprograms to support these options.

5.3 Trace Option for Debug

This trace option is not presented to the user because it is intended only for the ENPORT support programmer and it can generate large amounts of output. Therefore, the output is routed to a file rather than to the

screen. In order to evaluate the trace information after the ENPORT run, this file can be browsed online or printed.

Entering the hidden option T at the external function processor main menu calls subprogram DEBUG which controls the trace facility for this work. DEBUG provides a toggle to enable or disable all trace printouts and allows trace "flags" which control individual printouts to be turned on. Trace flags can also be turned off and on during the function definition by entering specially-coded comment statements. The trace toggle and all trace flags are initialized off at every entry into the external function processor from ENPORT. The special comment statements have the form CSET# followed by sets of "+nn" and/or "-nn." (+nn means to turn flag nn on and -nn means to turn it off.) Setting flag +00 will turn all trace flags on and -00 will turn them all off. The +nn and -nn can be repeated allowing several flags to be set with one CSET# statement.

Figure 5 lists the trace flag numbers, the subprogram in which they appear and a brief description of the trace output. The trace output file is named DEBUGIT and is opened on unit five. If the file name already exists the first time tracing is toggled on, the option to overwrite or give another file name is presented.

Flag Number	Tracing Subprogram	Trace Description to Debug File
02	NEWFCT	User input line after a good parse
04	LEX	FSA and user input pointers
05	LEX PARSER CODE	Screen error messages
06	PUSH	Parser stacks each time a token added
07	PARSER	User input line before calling LEX
08	PUTINT	Integer constant symbol table entry
09	LEX	Building of integer constant
10	LEX	Building of real constant
11	PUTSYM	Variable name symbol table entry
12	PUTREL	Real constant symbol table entry
13	NEWFCT	Symbol table dump after function definition completed
14	LEX	Building of comparison and relational operators
15	REDUCE	Grammar rule reduction
16	CODEGN	Internal code generation
17	REDUCE FAILRD CODEGN NEXT	Screen error messages
18	PARSER	Pointers to reset after user error
19	CODEGN	Dummy message when rule reduction generates no internal code
24	ULDEXT	Load areas and pointers before and after function unload and at exit from external function processor

Figure 5. Trace Flags

Chapter 6

SUMMARY AND RECOMMENDATIONS

The work presented in this thesis has addressed an ENPORT user's need to write his own node equation definition and allows the definition to be made without leaving ENPORT. Node equations are no longer limited to the standard function library, but can now be fine tuned to more closely model the real (nonlinear) world. Multiple custom function definitions may be defined for a bond-graph problem and a definition can be archived for use in another problem. This enhancement supports ENPORT's ability to save complete problem descriptions to a problem file and to restore the problem during another session.

A review of the evolution of this work reveals some areas which could benefit from further attention. Recommendations for further work appear in the sections which follow and deal generally with user interface and internal code optimization for the interpreter.

6.1 Expanded Error Messages

Writers of compilers and other language processors agree that errors must be detected, but which action should be taken is the subject of much debate. Some batch compilers try to make corrections to recover from

input stream errors [6,7]. In the case of this work the user is at the console, and since only he knows what he meant to do, the approach taken informs the user and lets him make the correction.

Extensive validity checking during lexical analysis and parsing of user input means that all errors in syntax are detected and a message is written to the console. Most error message texts include the element of the user's input that was found to be in error. Almost all errors require entry of a new (and hopefully correct) definition statement.

The error message to the user could be enhanced by echoing the input line and underlining the error. Errors detected in LEX must occur between the two pointers TOPUI and PT2. TOPUI marks the start of the token and PT2 is moved one character at a time to scan the input. Errors found by PARSER result from (1) grammar tokens that cannot appear together or (2) a failure to find a match between the grammar rules and tokens on the parser stack when PARSER tries to do a reduction. Delimiting the error here will be more difficult. Another array parallel to the parsing stack of tokens would have to be defined and it would contain pointers to the start of each token in the user input. Then the message handler could underline the area of the user input that contributed the tokens found to be in error.

Error messages are also produced if some non-recoverable condition occurs within the programs supporting this enhancement. The most likely cause would be the filling of a data structure used to hold external function definitions. In these cases, since some internal limit has been

exceeded already, there is no point asking the user to continue and so a program stop is executed. A stop should only happen in extreme cases involving unusually large complex function definitions, but it is clearly not a desirable action. Now that this work is complete, a comprehensive solution could be devised to abort the current function definition and return control to one of the function processor menus or in the extreme case return to ENPORT and allow selection of a library function.

Most of the program stops are accomplished by calling the subprogram STOPIT and can therefore be tracked down using the subprogram calling tree in Appendix F.

6.2 Improved Validity Checking

All user references to bond-variable names are validated to ensure that only inputs specified for the node are used as input and that only the output variable can have a value assigned to it. Of course, assignment can be made to intermediate variables.

Checks are not made to see if the function definition has referenced all declared input variables. There is also no check to verify that a value has been assigned to the output variable. Both of these checks would have to be made after the user ended his input. One way to make these checks would be to scan the internal code for the operators 0023 and 0024. These are Read-ENPORT and Write-ENPORT, respectively. Each instruction contains the offset in ENPORT I/O vector VBL, which can then be compared to the input and output list for the node. A user warning

message should be produced if the lists do not match. It would be up to the user to determine if the function definition needs to be corrected.

6.3 Internal Code Optimization

Time studies can easily be done to compare the solution time of a bond-graph problem using an external function and the same problem using a compiled function. If these tests show that the interpreter is significantly slower, steps can be taken to improve its performance.

Each internal code instruction contains three operands which are "decoded" before the operator is evaluated. Not every instruction uses all three operands, however. Six use two operands, two use only one and one instruction does not use any. If the operands were decoded after the operator evaluation, only those operands used by each operator would have to be referenced.

This decoding really just assigns the contents of MEMORY to another variable name (OP1, OP2 or OP3) in order to simplify the instruction evaluation procedures. The interpreter could also be improved by referencing MEMORY directly, eliminating operand decoding for any instruction.

Another optimizing step would be a check for any "imbedded" arithmetic, that is, arithmetic that could be performed in the internal code prior to interpreter processing. For instance, if the user had written AREA=2*PI*RADIUS, and since PI is a constant parameter, the

multiplication 2*PI could be performed ahead of time. The result is placed in the symbol table and the internal code modified to reference it directly, and the multiply instruction is removed.

An extension of this same idea is to look for unnecessary temporary variables in assignment statements. For example, if the user sets a variable named TEMP1 equal to a constant, then references TEMP1 later in his function, the assignment instruction can be removed and the internal code modified to reference the constant directly from the symbol table.

Efficient programming by the user would eliminate imbedded arithmetic and unnecessary temporary variables, but the cost to optimize for these cases is small. Even so, it is only worth the effort if the time studies show a need to increase interpreter efficiency.

LIST OF REFERENCES

LIST OF REFERENCES

1. Rosenberg, R. C. and Karnopp, D. C., SYSTEM DYNAMICS: A UNIFIED APPROACH, John Wiley & Sons, New York, 1975.

2. Rosenberg, R. C., 'ENPORT-6 User's Manual,' Rosencode Associates Inc., Lansing, Michigan, 1986.

3. Barret, W. A. and Couch, J. D., COMPILER CONSTRUCTION: THEORY AND PRACTICE, Science Research Associates, Inc., 1979.

4. Aho, A. V., and Johnson, S. C., LR Parsing, Computing Survey, Volume 6, No. 2, June 1974, pp. 99-124.

5. Aho, A. V., and Ullman, J. D., THE THEORY OF PARSING, TRANSLATION AND COMPILING, Volume 1, Prentice-Hall, Englewood Cliffs, New Jersey, 1972, pp. 58-75.

6. Ripley, G. David, A Simple Recovery-Only Procedure For Simple Precedence Parsers, Communications of the ACM, Volume 21, No. 11, November 1978, pp. 928-930.

7. Graham, Susan L. and Rhodes, Steven P., Practical Syntactic Error Recovery, Communications of the ACM, Volume 18, No. 11, November 1975, pp. 639-649.

APPENDICES

APPENDIX A

USER DOCUMENTATION

This appendix provides an introduction to the syntax and statement format used to enter custom external function definitions. Also following are three sample sessions that trace the steps used to create, edit and archive a function definition.

If you know FORTRAN, you should be able to write an external function equation without much trouble, since the syntax for the algebraic expressions is based on FORTRAN. Normally, you will want to code some expression using node inputs, FORTRAN library functions and constants, and then set this expression equal to the output variable for the node. The following is an example (not from any real bond-graph problem) using F.5 as input and E.5 as output:

```
   +- ENPORT prompt
   |
   |      +- enter in this area after the prompt
   |      |
<-----><-------------------------------------------...------>
>> 1>>: E.5=(1/100.0)*SIN((F.5**2)+2) - 1
>> 2>>: IF E.5 .LT. 0 THEN
>> 3>>:   E.5=0.0
>> 4>>: ENDIF
```

Notice that your statements can begin in the first column. Another difference from FORTRAN is that no GO TO's or labels are allowed. In

general, blanks are ignored in the input, so for more readability in that first line, blanks could be added:

```
>> 1>>: E.5 = (1/100.0) * SIN( (F.5**2) + 2 ) - 1
```

You may use all the normal algebraic operators (+, -, *, /, **) and any of the following FORTRAN intrinsic functions:

ABS	LOG
ACOS	LOG10
AINT	MAX
ANINT	MIN
ASIN	MOD
ATAN	SIGN
ATAN2	SIN
COS	SINH
COSH	SQRT
DIM	TAN
EXP	TANH

Also available is the IF-THEN-ELSEIF-ELSE-ENDIF statement. This statement must start with IF, must have a THEN clause and must end with ENDIF. The ELSEIF and ELSE clauses are optional.

Each line must contain a complete statement. Algebraic expressions cannot continue over to the next line. If you must code long expressions, use intermediate variables. The parts of the IF-THEN-ELSEIF-ELSE-ENDIF construct that should stand alone on a single line are:

```
IF  relational-expression  THEN
ELSEIF  relational-expression  THEN
ELSE
ENDIF
```

Some variations on this are possible for the external function processor to understand, but experiment at your own risk.

APPENDIX A1

SAMPLE CREATE SESSION

(From the ENPORT "Element equation options" menu, node equations can be set or changed. Selecting either of these options generates prompts for the element name, the function type and the number and name(s) of the inputs.)

```
Element equation options
--------------------------------
L: List the current equations
D: Details of current equations
C: Change selected equations
S: Specify all equations
U: set to Unit linear
A: Available function types
V: Vector definitions
H: Help
X: eXit from this menu (default)
--------------------------------
Enter option (X): C


Ready to change selected equations ...

Enter the element name (QUIT): R

Enter the function type (GAIN    ): USERDEF

Enter number of inputs ( 1):

Enter input  1 (F.5         ):
```

(One of the function types available is USERDEF. If this type is selected, specify the number and name(s) of the inputs, and an "External function processor options" menu appears.)

```
External function processor options
---------------------------------------------------
Q: show eQuation for this node
C: Create a new external function
E: Edit the current function
D: show Directory of archived functions
S: show Source statements of an archived function
R: Restore an archived function
A: Archive the current function
P: Purge an archived function
H: Help
X: eXit from external function processor (default)
---------------------------------------------------
Enter option (X): Q

The node equation is:  E.5          = USERDEF (F.5          )


 ( Generally, when entering the external function processor, you
   will want to create a new functional relationship of the input
   variables to the output variable for this node.  Option Q
   displays a generic version of the node equation, showing the
   output equal to a function of the inputs.  Initially the name of
   the function shown is USERDEF.  This means an external function
   type has been specified, but no external function has been created
   and loaded.  The variable names shown are the ones that are used
   when writing the function definition statements. )


eQn,Create,Load,Edit,Dir,Source,Restore,Arch,Purge,Help,eXit? (full): C

 ( Option C begins the creation of a new function definition.
   Bypass use of a file as input.  An input file would have to be
   created before entering ENPORT. )


Do you want to define the function using a file as input? (N):
```

```
Processing begun for the current function

>>nn>>: <---- Enter up to 50 source statements of maximum length 72 --->
>> 1>>: C  COMMENTS BEGIN WITH A "C" IN COLUMN ONE
>> 2>>: *  COMMENTS CAN ALSO BEGIN WITH AN "*" IN COLUMN ONE
>> 3>>: C
>> 4>>: C  DEFINITION STATEMENTS CAN START IN COLUMN ONE
>> 5>>: C
>> 6>>: PARM=100.0
>> 7>>: C
>> 8>>: C  OR THEY CAN HAVE LEADING BLANKS
>> 9>>: C
>>10>>:    E.5=F.5/PARM
>>11>>: C
>>12>>: C  TERMINATE INPUT BY TYPING "END"
>>13>>: C  OR JUST HIT <ENTER> AFTER A LINE PROMPT
>>14>>: END


 ( Processing the definition statements for the external function
   has begun.  Source statements are entered after the numbered
   prompts.  All normal algebraic expressions can be used to set a
   value into the output variable.  Comments are optional.  Statements
   may begin in column one or have leading blanks. )


Creation of a new current function successful.

The function will now be loaded to make it available for the problem
  solution.

Hit <return> to continue...

The current function is successfully loaded.

The node equation is:  E.5          = CUR.FCN (F.5          )


 ( Extensive syntax checking is done for each source statement.  A
   particularly complicated expression may take several seconds to
   process.  After terminating the input, the message "Creation of
   a new current function successful" should appear.  The function
   definition is then loaded and the new form of the generic node
   equation is displayed.  The name USERDEF is replaced by
   "CUR.FCN" to indicate that an external function is in place. )


eQn,Create,Load,Edit,Dir,Source,Restore,Arch,Purge,Help,eXit? (full): X

( Option X allows an exit from the external function processor
  and offers a chance to modify the equation definition for the
  node. )

Modify the equation definition? (N):
```

APPENDIX A2

SAMPLE EDIT SESSION

(An external function type can always be replaced by one of the standard library function types like GAIN. However, if an external function type has already been defined and loaded for the node, you may wish to edit it. To do this, select the USERDEF type again to enter the external function processor.)

```
Element equation options
--------------------------------
L: List the current equations
D: Details of current equations
C: Change selected equations
S: Specify all equations
U: set to Unit linear
A: Available function types
V: Vector definitions
H: Help
X: eXit from this menu (default)
--------------------------------
Enter option (X): C


Ready to change selected equations ...

Enter the element name (QUIT): R

Enter the function type (USERDEF ):

Enter number of inputs ( 1):

Enter input  1 (F.5         ):
```

```
External function processor options
---------------------------------------------------
Q: show eQuation for this node
C: Create a new external function
E: Edit the current function
D: show Directory of archived functions
S: show Source statements of an archived function
R: Restore an archived function
A: Archive the current function
P: Purge an archived function
H: Help
X: eXit from external function processor (default)
---------------------------------------------------
Enter option (X): Q
```

(The name "CUR.FCN" shown by option Q indicates that an external function is in place for this node.)

```
The node equation is:  E.5          = CUR.FCN (F.5          )
```

(Option E enters the editor, and its own menu, "External function editor options" appears. The menu allows several ways to edit the definition statements for the current function.)

```
eQn,Create,Load,Edit,Dir,Source,Restore,Arch,Purge,Help,eXit? (full): E

External function editor options
---------------------------------------------------
L: List definition statements
R: Replace a line in the current function
D: Delete a line in the current function
I: Insert a line in the current function
S: Save edits to the current function
H: Help
X: eXit from the external function editor (default)
---------------------------------------------------
Enter option (X): L
```

(Option L shows a listing of the source statements for the external function.)

```
Editor listing for the current function
>> 0>>: <---------------------- TOP - OF - DATA ---------------------->
>> 1>>: C  COMMENTS BEGIN WITH A "C" IN COLUMN ONE
>> 2>>: *  COMMENTS CAN ALSO BEGIN WITH AN "*" IN COLUMN ONE
>> 3>>: C
>> 4>>: C  DEFINITION STATEMENTS CAN START IN COLUMN ONE
>> 5>>: C
>> 6>>: PARM=100.0
>> 7>>: C
>> 8>>: C  OR THEY CAN HAVE LEADING BLANKS
>> 9>>: C
>>10>>:    E.5=F.5/PARM
>>11>>: C
>>12>>: C  TERMINATE INPUT BY TYPING "END"
>>13>>: C  OR JUST HIT <ENTER> AFTER A LINE PROMPT


 ( Use the line numbers shown in the source statement listing
   to make edits.  Option R allows a line to be replaced. )

List,Replace,Delete,Insert,Save,Help,eXit (full): R

What is the line number of the line you want to replace?: 6

>> 6>>: PARM=100.0

Okay to replace this line? (N): Y

Enter the replacement line.
>> 6>>: PARM=50.0

Line  6 is replaced.

 ( Another listing shows the line replaced. )

List,Replace,Delete,Insert,Save,Help,eXit (full): L

Editor listing for the current function
(This function must be SAVEd to retain your edits)
>> 0>>: <---------------------- TOP - OF - DATA ---------------------->
>> 1>>: C  COMMENTS BEGIN WITH A "C" IN COLUMN ONE
>> 2>>: *  COMMENTS CAN ALSO BEGIN WITH AN "*" IN COLUMN ONE
>> 3>>: C
>> 4>>: C  DEFINITION STATEMENTS CAN START IN COLUMN ONE
>> 5>>: C
>> 6>>: PARM=50.0
>> 7>>: C
>> 8>>: C  OR THEY CAN HAVE LEADING BLANKS
>> 9>>: C
>>10>>:    E.5=F.5/PARM
>>11>>: C
>>12>>: C  TERMINATE INPUT BY TYPING "END"
>>13>>: C  OR JUST HIT <ENTER> AFTER A LINE PROMPT
```

```
 ( Option D allows a line to be deleted. )

List,Replace,Delete,Insert,Save,Help,eXit (full): D

What is the number of the line you wish to delete?: 13

>>13>>: C  OR JUST HIT <ENTER> AFTER A LINE PROMPT

Okay to delete this line? (N): Y

Line 13 is deleted.


 ( Again, a listing shows the line deletion. )


List,Replace,Delete,Insert,Save,Help,eXit (full): L

Editor listing for the current function
(This function must be SAVEd to retain your edits)
>> 0>>: <---------------------- TOP - OF - DATA ---------------------->
>> 1>>: C  COMMENTS BEGIN WITH A "C" IN COLUMN ONE
>> 2>>: *  COMMENTS CAN ALSO BEGIN WITH AN "*" IN COLUMN ONE
>> 3>>: C
>> 4>>: C  DEFINITION STATEMENTS CAN START IN COLUMN ONE
>> 5>>: C
>> 6>>: PARM=50.0
>> 7>>: C
>> 8>>: C  OR THEY CAN HAVE LEADING BLANKS
>> 9>>: C
>>10>>:    E.5=F.5/PARM
>>11>>: C
>>12>>: C  TERMINATE INPUT BY TYPING "END"


 ( Option I allows lines to be inserted into the source code.
   Specify the line number after which the line(s) are to be
   inserted.  Unlike replace or delete, insert allows more than one
   line to be inserted at a time.  Hit <enter> after a prompt to
   terminate the insertions. )


List,Replace,Delete,Insert,Save,Help,eXit (full): I

After what line number would you like to insert?: 0

Enter insertion(s) after the following line.
>> 0>>: <---------------------- TOP - OF - DATA ---------------------->
>> 1>>: *  THIS FUNCTION HAS BEEN EDITED.
>> 2>>: *  (MORE THAN ONE LINE CAN BE INSERTED AT ONE TIME)
>> 3>>: *
>> 4>>:

Insertion(s) completed.
```

```
 ( Option L shows the lines which have been inserted. )


List,Replace,Delete,Insert,Save,Help,eXit (full): L

Editor listing for the current function
(This function must be SAVEd to retain your edits)
>> 0>>: <---------------------- TOP - OF - DATA ---------------------->
>> 1>>: *  THIS FUNCTION HAS BEEN EDITED.
>> 2>>: *  (MORE THAN ONE LINE CAN BE INSERTED AT ONE TIME)
>> 3>>: *
>> 4>>: C  COMMENTS BEGIN WITH A "C" IN COLUMN ONE
>> 5>>: *  COMMENTS CAN ALSO BEGIN WITH AN "*" IN COLUMN ONE
>> 6>>: C
>> 7>>: C  DEFINITION STATEMENTS CAN START IN COLUMN ONE
>> 8>>: C
>> 9>>: PARM=50.0
>>10>>: C
>>11>>: C  OR THEY CAN HAVE LEADING BLANKS
>>12>>: C
>>13>>:    E.5=F.5/PARM
>>14>>: C
>>15>>: C  TERMINATE INPUT BY TYPING "END"


 ( A "Save" must be done to preserve your edits.  Continuing to
   exit will lose all edits done during this edit session. )


List,Replace,Delete,Insert,Save,Help,eXit (full): X

The current function has been edited, but not SAVED.
Okay to lose your edits? (N):

Use the SAVE option before exiting the editor.


 ( Option S to save the new function definition invokes the
   syntax checker.  Errors in the edited statements may be detected
   and must be corrected by further edits.  Only a correct function
   definition will be accepted. )


List,Replace,Delete,Insert,Save,Help,eXit (full): S

The current function has been edited and
it will be checked for correct syntax.

Hit <return> to continue...


 ( The syntax checker echos the definition statements of the
   function as they are checked. )
```

```
Processing begun for the current function
>> 1>>: *  THIS FUNCTION HAS BEEN EDITED.
>> 2>>: *  (MORE THAN ONE LINE CAN BE INSERTED AT ONE TIME)
>> 3>>: *
>> 4>>: C  COMMENTS BEGIN WITH A "C" IN COLUMN ONE
>> 5>>: *  COMMENTS CAN ALSO BEGIN WITH AN "*" IN COLUMN ONE
>> 6>>: C
>> 7>>: C  DEFINITION STATEMENTS CAN START IN COLUMN ONE
>> 8>>: C
>> 9>>: PARM=50.0
>>10>>: C
>>11>>: C  OR THEY CAN HAVE LEADING BLANKS
>>12>>: C
>>13>>:    E.5=F.5/PARM
>>14>>: C
>>15>>: C  TERMINATE INPUT BY TYPING "END"

Parse of the current function successful.

 ( The edit session is complete.  Use option X to exit the editor
   and go back to the external function processor main menu.  The
   edited function definition will be loaded to make it available
   for the problem solution. )

List,Replace,Delete,Insert,Save,Help,eXit (full): X

The function will be loaded to make it available for problem solution.

Hit <return> to continue...

The current function is successfully loaded.

The node equation is:  E.5          = CUR.FCN (F.5          )

Hit <return> to continue...

External function processor options
---------------------------------------------------
Q: show eQuation for this node
C: Create a new external function
E: Edit the current function
D: show Directory of archived functions
S: show Source statements of an archived function
R: Restore an archived function
A: Archive the current function
P: Purge an archived function
H: Help
X: eXit from external function processor (default)
---------------------------------------------------
Enter option (X): X

 ( Option X exits the external function processor. )

Modify the equation definition? (N):
```

APPENDIX A3

SAMPLE ARCHIVE SESSION

```
 ( The other options available on the External function
   processor menu are D, S, R, A and P. )


External function processor options
-----------------------------------------------
Q: show eQuation for this node
C: Create a new external function
E: Edit the current function
D: show Directory of archived functions
S: show Source statements of an archived function
R: Restore an archived function
A: Archive the current function
P: Purge an archived function
H: Help
X: eXit from external function processor (default)
-------,---------------------------------------
Enter option (X): D


 ( Option D lists the archived functions in the external
   function directory.  Included are the names of the function,
   the file in which it is stored and the file creation date
   and time. )


Function  External     File Creation
  Name      File       Date       Time
-----------------------------------------
 ARCH2     ARCH2    11/02/86   20:03:55
 ARCHA     ARCHA    11/03/86   19:54:41
 SAMPLE    SAMPLE   11/06/86   12:47:40

External function directory has  3 entries and a limit of 50.
```

```
 ( Option S lists the function "Source" statements for an
   archived function. )


eQn,Create,Load,Edit,Dir,Source,Restore,Arch,Purge,Help,eXit? (full): S

Enter the name of the function: SAMPLE


Source listing for archived external function SAMPLE
--------------------------------------------------------------------
>> 1>>:C  COMMENTS BEGIN WITH A "C" IN COLUMN ONE
>> 2>>:*  COMMENTS CAN ALSO BEGIN WITH AN "*" IN COLUMN ONE
>> 3>>:C
>> 4>>:C  DEFINITION STATEMENTS CAN START IN COLUMN ONE
>> 5>>:C
>> 6>>:PARM=100.0
>> 7>>:C
>> 8>>:C  OR THEY CAN HAVE LEADING BLANKS
>> 9>>:C
>>10>>:   E.5=F.5/PARM
>>11>>:C
>>12>>:C  TERMINATE INPUT BY TYPING "END"
>>13>>:C  OR JUST HIT <ENTER> AFTER A LINE PROMPT


 ( The name USERDEF in the response to option Q indicates
   that no external function is in place yet for this node.  An
   archived function may be restored using option R.  SAMPLE
   is one of the archived function names listed in the directory. )


eQn,Create,Load,Edit,Dir,Source,Restore,Arch,Purge,Help,eXit? (full): Q

The node equation is:  E.5          = USERDEF (F.5          )

eQn,Create,Load,Edit,Dir,Source,Restore,Arch,Purge,Help,eXit? (full): R

Enter the name of the function: SAMPLE

Restore successful for function SAMPLE

The restored function must be edited to insure
the correct use of bond variable names.

Hit <return> to continue...


 ( Because functions restored from archive probably will not
   reference bond variable names appropriate for the current node,
   the function must be edited. )
```

```
External function editor options
--------------------------------------------------
L: List definition statements
R: Replace a line in the current function
D: Delete a line in the current function
I: Insert a line in the current function
S: Save edits to the current function
H: Help
X: eXit from the external function editor (default)
--------------------------------------------------
Enter option (X): L
```

(Option L lists the restored function. Change the bond variables used, if necessary, using the edit features. When editing is complete, SAVE the function.)

```
Editor listing for the current function
>> 0>>:<------------------------ TOP - OF - DATA ---------------------->
>> 1>>:C  COMMENTS BEGIN WITH A "C" IN COLUMN ONE
>> 2>>:*  COMMENTS CAN ALSO BEGIN WITH AN "*" IN COLUMN ONE
>> 3>>:C
>> 4>>:C  DEFINITION STATEMENTS CAN START IN COLUMN ONE
>> 5>>:C
>> 6>>:PARM=100.0
>> 7>>:C
>> 8>>:C  OR THEY CAN HAVE LEADING BLANKS
>> 9>>:C
>>10>>:   E.5=F.5/PARM
>>11>>:C
>>12>>:C  TERMINATE INPUT BY TYPING "END"
>>13>>:C  OR JUST HIT <ENTER> AFTER A LINE PROMPT

List,Replace,Delete,Insert,Save,eXit (full): X

The current function has been restored and must be saved.

Use the SAVE option before exiting the editor.

List,Replace,Delete,Insert,Save,eXit (full): S

The current function has not been edited, but
it will be checked for correct syntax.

Hit <return> to continue...
```

(In this case, no editing is necessary, so the function is saved. Each line of text is checked for correct syntax and echoed to the screen.)

```
Processing begun for the current function
>> 1>>:C  COMMENTS BEGIN WITH A "C" IN COLUMN ONE
>> 2>>:*  COMMENTS CAN ALSO BEGIN WITH AN "*" IN COLUMN ONE
>> 3>>:C
>> 4>>:C  DEFINITION STATEMENTS CAN START IN COLUMN ONE
>> 5>>:C
>> 6>>:PARM=100.0
>> 7>>:C
>> 8>>:C  OR THEY CAN HAVE LEADING BLANKS
>> 9>>:C
>>10>>:   E.5=F.5/PARM
>>11>>:C
>>12>>:C  TERMINATE INPUT BY TYPING "END"
>>13>>:C  OR JUST HIT <ENTER> AFTER A LINE PROMPT

Parse of the current function successful.


 ( After a successful parse, use option X to exit the editor.
   The restored function is loaded for use in the problem solution
   and the node equation is displayed.  "CUR.FCN" indicates that the
   new external function is in place. )


List,Replace,Delete,Insert,Save,eXit (full): X

The function will now be loaded to make it available for the problem
solution.

Hit <return> to continue...

The current function is successfully loaded.

The node equation is:  E.5          = CUR.FCN (F.5         )

Hit <return> to continue...


External function processor options
---------------------------------------------------
Q: show eQuation for this node
C: Create a new external function
E: Edit the current function
D: show Directory of archived functions
S: show Source statements of an archived function
R: Restore an archived function
A: Archive the current function
P: Purge an archived function
H: Help
X: eXit from external function processor (default)
---------------------------------------------------
Enter option (X): A
```

```
( Option A archives the definition statements for the current
  function to an external file and adds the function to the
  directory. )


Enter the name (limit 6 char) for the archived function: SAMPL2

External function directory has  3 entries and a limit of 50.

Enter external file name for saving SAMPL2, (SAMPL2):

Current function definition successfully archived to SAMPL2.


( Option D shows the updated directory with the newly-archived
  function added. )


eQn,Create,Load,Edit,Dir,Source,Restore,Arch,Purge,Help,eXit? (full): D

Function  External     File Creation
  Name      File       Date       Time
----------------------------------------
 ARCH2     ARCH2    11/02/86   20:03:55
 ARCHA     ARCHA    11/03/86   19:54:41
 SAMPLE    SAMPLE   11/06/86   12:47:40
 SAMPL2    SAMPL2   11/06/86   12:50:08

External function directory has  4 entries and a limit of 50.


( Option P allows an archived function to be purged from the
  directory.  This action also deletes the storage file from
  your library. )


eQn,Create,Load,Edit,Dir,Source,Restore,Arch,Purge,Help,eXit? (full): P

Enter the name of the function: SAMPL2

Function selected for purge:  SAMPL2
     Saved on external file:  SAMPL2
             Created on day:  11/06/86
            Created at time:  12:50:08

Okay to purge this function? (N): Y

External file SAMPL2 for function SAMPL2 will be deleted.

Directory entry purged for function SAMPL2
```

```
( Another listing of the archive entries in the external function
  directory shows the function is purged. )


eQn,Create,Load,Edit,Dir,Source,Restore,Arch,Purge,Help,eXit? (full): D

Function  External     File Creation
  Name      File      Date       Time
----------------------------------------
 ARCH2     ARCH2    11/02/86   20:03:55
 ARCHA     ARCHA    11/03/86   19:54:41
 SAMPLE    SAMPLE   11/06/86   12:47:40

External function directory has  3 entries and a limit of 50.

eQn,Create,Load,Edit,Dir,Source,Restore,Arch,Purge,Help,eXit? (full): X

Modify the equation definition? (N):
```

APPENDIX B

LANGUAGE SYNTAX

The descriptions below use the following symbols in their definitions for elements of the language. Blanks in the definitions of variable names, integer constants and real constants are for clarity only and are not part of the definition. Blanks terminate each of these language elements.

[ ]	square brackets enclose optional elements or lists of optional alternate elements
[[ ]]	doubled square brackets enclose optional elements or lists of optional elements that can be repeated
{ }	braces enclose lists of alternate elements, one of which must be selected
|	a vertical line separates alternate items when one item is to be chosen exclusive of the others

Variable names
---

variable: letter [[letter | digit]]

Variable names must begin with a letter and are made up of letters and digits. The maximum length is six characters. Use of names longer than six characters causes a warning message stating that the name will be truncated to six characters and processing will continue. Logical variables are not supported and there is no typing of variables either explicitly or implicitly. All calculations are done using double-precision arithmetic.

Integer constants
---

integer: digit [[digit]]

Integer constants are recognized, but are converted to double precision (real) when they are stored, since all subsequent calculations will be done in double precision. The length of an integer constant is limited to ten characters. An integer that is too long is truncated to the left-most digits and processing continues. Of course, a warning message is issued. This does not yield the program the user desires, but it does allow the rest of the line to be checked for syntax errors. The lexical analyzer does not force the user to reenter a line after a warning message, only after an error.

Real constants

```
            real:  [[ digit ]]  { digit. | .digit }  [[ digit ]]
real-exponential:  { integer | real }  E  [ + | - ] integer
```

Real numbers must use a decimal point or be expressed by using exponential notation. The decimal must precede or follow at least one digit. If an exponent is specified, it must immediately follow the last character in the mantissa (either a digit or the decimal point). The exponent is written with an "E" followed by the value of the exponent. The value may be signed or unsigned (assumed positive) and must be an integer.

External function syntax

Figure B1 presents the complete formal syntax for user input statements needed to define an external function. The sequence of syntax from top to bottom proceeds from the most general to the most specific. The first syntax rule, then, defines an external function as a series of statements followed by an optional "END." Succeeding rules further delineate the syntax of correct statements.

```
External Function:  statements  [ END ]

statements:  statement  [[ statements ]]

statement:  variable = expression

statement:  IF relational THEN statements
            [[ ELSEIF relational THEN statements ]]
            [ ELSE statements ]
            ENDIF

relational:  ( relational )

relational:  .NOT. relational

relational:  relational  { .OR. | .AND. }  relational

relational:  expression  { .EQ. | .NE. | .LT. | .LE. | .GT. | .GE. }
               expression

expression:  ( expression )

expression:  expression  { + | - | / | * | ** }  expression

expression:  [ + | - ]  { variable | constant | FORTRAN-function }

FORTRAN-function:  FORTRAN-function-name ( argument-list )

argument-list:  argument-list , expression

argument-list:  expression
```

Figure B1. External Function Syntax

Note on blanks in the input

In general, blanks are ignored by the processor. However, blanks are not allowed to appear within any single operand, operator or reserve word. Lines can have leading blanks and there can be any number of blanks between operators, operands and reserve words. All statements must be completed on one line, that is, statements are not allowed to be continued to another line. Statements that are a part of the control for the IF-THEN-ELSEIF-ELSE-ENDIF structure can span several lines. The parts that should stand alone on a single line are:

```
IF  relational-expression  THEN
ELSEIF  relational-expression  THEN
ELSE
ENDIF
```

Assignment statements within the blocks must still be on a single line, however.

Comment lines

Any line with an asterisk in column one will be treated as a comment. Any line with a "C" in column one will be treated as a comment, unless the "C" (or variable name beginning with "C") is followed by an "=", as in an assignment statement. Blank lines signify the end of user input and cannot, therefore, be used as comments.

APPENDIX C

FSA DIAGRAM

The Finite State Automaton (FSA) shown in Figure C1 controls the lexical analysis portion of this work and has twelve states. State zero (0) is a final state from which LEX returns the user input token to PARSER. The FSA recognizes arithmetic operators, relational operators, variables, integer constants and real constants, which may be in decimal or exponential form. Two of the final states indicate input character sequence errors. Other errors may occur but are not detected until the return to PARSER.

From the initial state one (1), the FSA changes states by scanning one character of the user input and moving along a directed arrow to the next state. Next to each arrow leaving a state are indicators of the input needed to take that path. The legend below the FSA diagram explains the notation next to the arrows. Boxes drawn around states are titled by the input token related to those states.

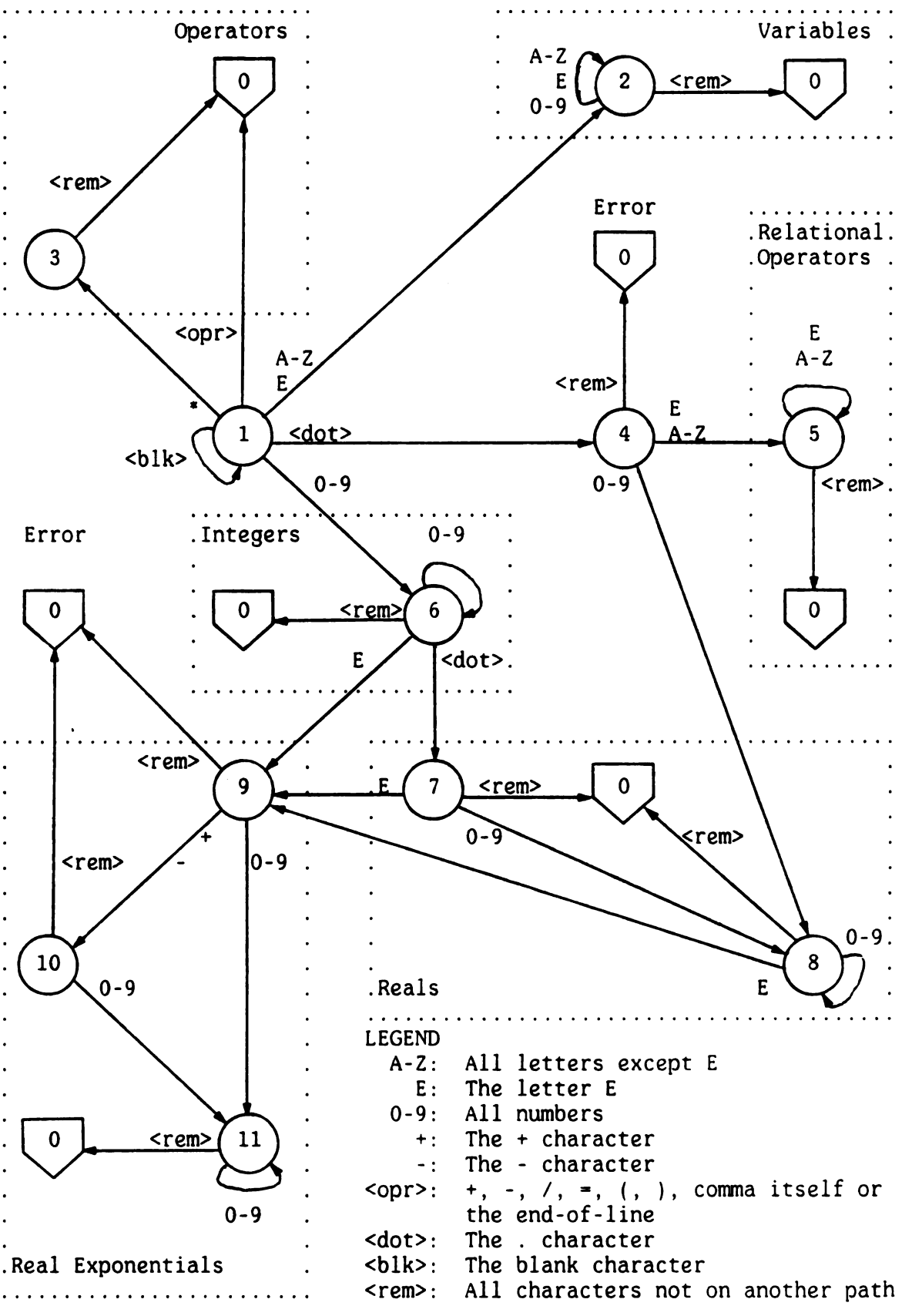

Figure C1. FSA Diagram

APPENDIX D

SIMPLE PREDECENCE GRAMMAR

The simple predecence grammar governs the parse of the user input lines. The rules are comprised of grammar tokens called terminals and nonterminals. Terminals are tokens returned by the lexical analysis to the parser and consist of operators, reserve words, operands and a special token "ENDLNE" which indicates a carriage return used to end the input line. The nonterminals are defined by the grammar rules in terms of terminals and other nonterminals. These grammar tokens are listed in Appendix D2.

The rules are written with a nonterminal on the left of an arrow symbol (-->) and the tokens it replaces in a grammar reduction written on the right. The grammar rules for the simple precedence grammar used in this work are listed in Appendix D1.

APPENDIX D1

GRAMMAR RULES

Rule No.	Grammar Production Rules
1.	FUNCT --> CSTMTS END ENDLNE
2.	CSTMTS --> STMTS
3.	STMTS --> STMTS SSTMT ENDLNE
4.	STMTS --> STMTS SSTMT
5.	STMTS --> SSTMT ENDLNE
6.	STMTS --> SSTMT
7.	SSTMT --> VAR = CXPR
8.	SSTMT --> IFLLSE CSTMTS ENDIF
9.	SSTMT --> IFTHNN CSTMTS ENDIF
10.	CXPR --> BTERM
11.	IFLLSE --> IFELSE ENDLNE
12.	IFLLSE --> IFELSE
13.	IFTHNN --> IFCOND ENDLNE
14.	IFTHNN --> IFCOND
15.	IFELSE --> IFTHNN CSTMTS ELSE
16.	IFCOND --> IFSTMT BRELAT THEN
17.	IFCOND --> IF BRELAT THEN
18.	IFSTMT --> IFTHNN CSTMTS ELSEIF
19.	BRELAT --> CRELAT
20.	CRELAT --> .NOT. CRELAT
21.	CRELAT --> RELATE .OR. CRELAT
22.	CRELAT --> RELATE .AND. CRELAT
23.	CRELAT --> RELATE

```
Rule
No.   Grammar Production Rules
----  ----------------------------

 24.  RELATE --> XPR .EQ. BTERM
 25.  RELATE --> XPR .NE. BTERM
 26.  RELATE --> XPR .LT. BTERM
 27.  RELATE --> XPR .LE. BTERM
 28.  RELATE --> XPR .GT. BTERM
 29.  RELATE --> XPR .GE. BTERM
 30.  RELATE --> ( BRELAT )

 31.  BTERM  --> XPR

 32.  XPR    --> XPR + CTERM
 33.  XPR    --> XPR - CTERM
 34.  XPR    --> + CTERM
 35.  XPR    --> - CTERM
 36.  XPR    --> CTERM

 37.  CTERM  --> FACTOR

 38.  FACTOR --> FACTOR * CEXFAC
 39.  FACTOR --> FACTOR / CEXFAC
 40.  FACTOR --> CEXFAC

 41.  CEXFAC --> EXFAC

 42.  EXFAC  --> EXFAC ** STERM
 43.  EXFAC  --> STERM

 44.  STERM  --> VAR
 45.  STERM  --> CON
 46.  STERM  --> VAR ( CXPR )
 47.  STERM  --> VAR ( CXPR , CXPR )
 48.  STERM  --> VAR ( CXPR , CXPR , CXPR )
 49.  STERM  --> VAR ( CXPR , CXPR , CXPR , CXPR )
 50.  STERM  --> VAR ( CXPR , CXPR , CXPR , CXPR , CXPR )
 51.  STERM  --> VAR ( CXPR , CXPR , CXPR , CXPR , CXPR ,
                      CXPR )

 52.  STERM  --> ( CXPR )
```

APPENDIX D2

GRAMMAR TOKENS

	Terminals			
Nonterminals	Operators	Reserve Words	Operands	Carriage Return
FUNCT	=	IF	VAR	ENDLNE
CSTMTS	+	THEN	CON	
STMTS	-	ELSEIF		
SSTMT	/	ELSE		
CXPR	*	ENDIF		
IFLLSE	**	END		
IFTHNN	(			
IFELSE	)			
IFCOND	,			
IFSTMT	.NOT.			
BRELAT	.OR.			
CRELAT	.AND.			
RELATE	.EQ.			
BTERM	.NE.			
XPR	.LT.			
CTERM	.LE.			
FACTOR	.GT.			
CEXFAC	.GE.			
EXFAC				
STERM				

APPENDIX E

INTERNAL CODE

The internal code operators are listed below in numeric order. Also included are the simple precedence grammar rule numbers which generate the instructions during the parse, the operator type and description of operand usage. There are twenty-four instructions in the set. Each one is an integer four-tuple of the form OPER OP1 OP2 OP3, in which OPER is the operator field and OP1, OP2 and OP3 are the operands. All four fields are integers and the value of OPER is between one and twenty-four. When the operand is preceded in the description with an "@", it means the operand is a pointer to the value to be used; otherwise, the operand is the value itself. Pointers refer to symbol table locations or, in the case of operators 0023 and 0024, to the ENPORT I/O vector, VBL.

Some instructions do not make use of all of the operand fields and this is noted in the description.

Operator	Rule(s)	Instruction Type and Description
0001	20	Relational. Put .NOT. @OP1 in @OP3.
0002	21	Relational. Put @OP1 .OR. @OP2 in @OP3.
0003	22	Relational. Put @OP1 .AND. @OP2 in @OP3.
0004	24	Compare. Set @OP3 true, if @OP1 .EQ. @OP2 is true, otherwise set @OP3 false.
0005	25	Compare. Set @OP3 true, if @OP1 .NE. @OP2 is true, otherwise set @OP3 false.
0006	26	Compare. Set @OP3 true, if @OP1 .LT. @OP2 is true, otherwise set @OP3 false.
0007	27	Compare. Set @OP3 true, if @OP1 .LE. @OP2 is true, otherwise set @OP3 false.
0008	28	Compare. Set @OP3 true, if @OP1 .GT. @OP2 is true, otherwise set @OP3 false.
0009	29	Compare. Set @OP3 true, if @OP1 .GE. @OP2 is true, otherwise set @OP3 false.
0010	35	Numeric. Put the negative of @OP1 in @OP3. OP2 is not used.
0011	32	Numeric. Put @OP1 + @OP2 in @OP3.
0012	33	Numeric. Put @OP1 - @OP2 in @OP3.
0013	38	Numeric. Put @OP1 * @OP2 in @OP3.
0014	39	Numeric. Put @OP1 / @OP2 in @OP3.
0015	42	Numeric. Put @OP1 ** @OP2 in @OP3.
0016	7	Assignment. Put @OP1 in @OP3. OP2 is not used.
0017	46-51	Function. Call function OP1 with OP2 arguments and put the result in @OP3. The arguments are in data instructions following this instruction.
0018	17	Branch-on-Condition. Branch to location OP3 if @OP1 is false. OP2 is not used.
0019	15,18	Branch. Branch to location OP1. OP2 and OP3 are not used.

Operator	Rule(s)	Instruction Type and Description
0020	1	Return. End interpreter and return to ENPORT. No operands are used.
0021	46-51	Data. Non-executable instruction to hold an argument in @OP1 for a function call. OP2 and OP3 are not used.
0022	8,9, 15,18	Label. Marks the destination for a branch instruction. OP1 is the label number. OP2 and OP3 are not used.
0023	46	Read-ENPORT. Put @OP1 in @OP3 of ENPORT I/O vector. OP2 is not used.
0024	7	Write-ENPORT. Get @OP1 from the ENPORT I/O vector and put it in @OP3. OP2 is not used.

APPENDIX F

SUBPROGRAM CALLING TREE

```
CHGFCN (ENPORT)
      DEFINE[3]
            ARCHIV[3]
                  GOON[1]
                  LSTDIR[4]
            DEBUG[3]
                  GETIN[1]
                  GOON[1]
            EDTFCN[3]
                  DLLINE[2]
                        GETIN[1]
                        PROMPT[1]
                        YORN[1]
                  EXTPTR
                  GOON[1]
                  INLINE[2]
                        GETIN[1]
                        GETWD[1]
                        PROMPT[1]
                  MENPAG[1]
                  RHFILE[1]
                  RPLINE[3]
                        GETLIN[1]
                        OUTBUF[1]
                  SAVFCN[2]
                        GOON[1]
                        NEWFCT[4]
            EXITCK[2]
                  SHOWEQ[4]
                  VBLNAM[1]
            FLAG
            GOON[1]
            LDCHK
            LDFCT[2]
                  ULDEXT
                        FLAG
                        LDCHK
            LSTDIR[2]
                  GOON[1]
                  MENPAG[1]
```

```
LSTSRC
        GOON[1]
        DIRCHK[3]
                LSTDIR[4]
        MENPAG[1]
        OPNFCT
                OUTBUF[1]
        OUTBUF[1]
MENPAG[1]
MENSET[1]
NEWFCT[3]
        ANFANG
                INTGER
                PUSH
                        FLAG
                PUTSYM
                        FLAG
        CMTCHK
        FLAG
        GOON[1]
        MENPAG[1]
        NEXT[2]
                FLAG
                STOPIT[2]
        PARSER[2]
                ERRTXT
                        EXPAND
                        WRTSTR[1]
                FLAG
                INTGER
                LEX[2]
                        CVTINT
                        CVTREL
                        PUTINT
                                FLAG
                        PUTREL
                                FLAG
                        PUTSYM
                                FLAG
                        TRANSIT
                        VARCVT
                                VBLIX[1]
                PREC
                        CODE[2]
                                FLAG
                PUSH
                        FLAG
```

```
                REDUCE[2]
                      CODEGN[2]
                            DISPLY
                            FLAG
                            INTGER
                            NEXT[2]
                                  FLAG
                                  STOPIT[2]
                      FLAG
                      STOPIT[2]
                STKCHK
                STOPIT[2]
          OPTCOD
                FLAG
                PROMPT[1]
                PRTCOD
                YORN[1]
          PRTSYM
    OUTBUF[1]
    PRGFCT
          BLNKLN[1]
          DIRCHK[4]
          OPNFCT[4]
          OUTBUF[1]
          PROMPT[1]
          YORN[1]
    RESTOR[2]
          DIRCHK[4]
          GOON[1]
          OPNFCT[4]
          PROCED[1]
    RHFILE[1]
    SHOWEQ
          EXTPTR
          VBLNAM[1]
          WRTSTR[1]
```

[1] ENPORT routine; no sub-tree given
[2] Calls ENPORT routines BLNKLN and WRTSTR
[3] Calls ENPORT routines BLNKLN, GETWD, PROMPT, WRTSTR and YORN
[4] Sub-tree given elsewhere in this Appendix

APPENDIX G

SUBPROGRAM LIST

ANFANG	(DEFINE.F77)
ARCHIV	(DEFINE.F77)
CLREXT	(OLDFCT.F77)
CMTCHK	(DEFINE.F77)
CODE	(PARSER.F77)
CODEGN	(REDUCE.F77)
CVTINT	(LEXIC.F77)
CVTREL	(LEXIC.F77)
DEBUG	(DEBUGT.F77)
DEFINE	(DEFINE.F77)
DIRCHK	(OLDFCT.F77)
DISPLY	(REDUCE.F77)
EDTFCN	(EDTFCN.F77)
ELLINE	(EDTFCN.F77)
ERRTXT	(PARSER.F77)
EXITCK	(DEFINE.F77)
EXPAND	(PARSER.F77)
EXTPTR	(OLDFCT.F77)
FAILRD	(REDUCE.F77)
FLAG	(DEBUGT.F77)
INLINE	(EDTFCN.F77)
INTERP	(INTERP.F77)
INTGER	(REDUCE.F77)
LDCHK	(DEBUGT.F77)
LDFCT	(OLDFCT.F77)
LEX	(LEXIC.F77)
LSTDIR	(OLDFCT.F77)
LSTSRC	(OLDFCT.F77)
NEWFCT	(DEFINE.F77)
NEXT	(REDUCE.F77)
OPNFCT	(OLDFCT.F77)
OPTCOD	(DEFINE.F77)
PARSER	(PARSER.F77)
PREC	(PARSER.F77)
PRGFCT	(OLDFCT.F77)
PRTCOD	(DEBUGT.F77)
PRTSYM	(DEBUGT.F77)
PUSH	(PARSER.F77)
PUTINT	(LEXIC.F77)
PUTREL	(LEXIC.F77)

PUTSYM	(LEXIC.F77)
READEX	(OLDFCT.F77)
REDUCE	(REDUCE.F77)
RESTOR	(OLDFCT.F77)
RPLINE	(EDTFCN.F77)
SAVFCN	(EDTFCN.F77)
SHOWEQ	(DEFINE.F77)
STKCHK	(DEBUGT.F77)
STOPIT	(PARSER.F77)
TRANSIT	(LEXIC.F77)
ULDEXT	(OLDFCT.F77)
VARCVT	(LEXIC.F77)

APPENDIX H

SOURCE CODE

There are nine files containing FORTRAN subprogram source code with extensive comments. In addition, there are seven files containing COMMON blocks which are referenced with INCLUDE statements in the source code files. The file contents are briefly described below. The source code itself has not been included here because it extends over 140 pages.

Subprogram files:

BLOCKS.F77 - BLOCK DATA routines to initialize data structures within the project.

DEBUGT.F77 - Subprograms supporting the project trace feature used for debugging and development.

DEFINE.F77 - External function processor menu and subprograms to create and archive a function definition.

EDTFCN.F77 - Menu and subprograms to support external function edit.

INTERP.F77 - Internal code interpreter called during bond-graph solution.

LEXIC.F77 - Subprograms supporting the lexical analysis of user input lines.

OLDFCT.F77 - Utilities to maintain the external function load area used by the interpreter and the directory of archived function definitions.

PARSER.F77 - Subprograms that implement the simple precedence parsing algorithm.

REDUCE.F77 - Implements the grammar rule reductions and generates internal code for the interpreter.

COMMON Files:

EXEDBK.COM - Data structures for external function editing.

GLOBAL.COM - General data structures used by many subprograms.

INTDBK.COM - Load area structures used by the interpreter.

LEXDEF.COM - Data structures used during lexical analysis.

PARSER.COM - Data structures to support parsing.