AD-A068 394 SCIENCE APPLICATIONS INC ENGLEWOOD CO F/G 9/2
FLIGHT TEST ORIENTED PRECOMPILER SYSTEM (FLTOPS DESIGN SPECIFIC--ETC(U)
AUG 78 D A OTEY, H R RAMSEY, J K WILLOUGHBY F04611-77-C-0040
UNCLASSIFIED SAI-78-061-DEN AFFTC-TR-78-22 NL

1 OF 4
AD A068394

AD A068394

DDC FILE COPY

AFFTC-TR-78-22

LEVEL II

2

AD INEXPLORATA

FLIGHT TEST ORIENTED PRECOMPILER SYSTEM (FLTOPS) DESIGN SPECIFICATIONS

D. A. Otey
H. R. Ramsey
J. K. Willoughby

Science Applications, Inc.
40 Denver Technological Center West
7935 East Prentice Avenue
Englewood, Colorado 80110

DDC
RECEIVED
MAY 9 1979
C

August 1978

Final Report

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

Prepared for:

6510 TESTW/TEEDM
Software Management Branch
Air Force Flight Test Center
Edwards AFB, CA 93523

AFTC

AIR FORCE FLIGHT TEST CENTER
EDWARDS AIR FORCE BASE, CALIFORNIA
AIR FORCE SYSTEMS COMMAND
UNITED STATES AIR FORCE

79 05 08 022

This report was submitted by Science Applications Inc., 40 Denver Technological Center West, 7935 East Prentice Avenue, Englewood, Colorado 80111, under Contract No. F04611-77-C-0040, Job Order Number SC6453, with the Air Force Flight Test Center, Edwards AFB, California 93523. Richard G. Hector and Larry L. Gorden were the AFFTC Project Engineers in charge of procurement.

Publication of this technical report does not constitute Air Force approval of its findings or conclusions. It is published only as a record of technical effort and for the exchange and stimulation of ideas concerning the Flight Test Oriented Precompiler System (FLTOPS) concept.

Prepared by:

DEAN A. OTEY
Science Applications, Inc.

H. R. RAMSEY
Science Applications, Inc.

JOHN K. WILLOUGHBY
Science Applications, Inc.

This report has been reviewed and is approved for publication:
27 November 1978

WILLIAM T. TWINTING, Colonel, USAF
Commander 6510 Test Wing

DISCLAIMER NOTICE

THIS DOCUMENT IS BEST QUALITY PRACTICABLE. THE COPY FURNISHED TO DDC CONTAINED A SIGNIFICANT NUMBER OF PAGES WHICH DO NOT REPRODUCE LEGIBLY.

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER: AFFTC-TR-78-22	2. GOVT ACCESSION NO.
3. RECIPIENT'S CATALOG NUMBER	
4. TITLE *(and Subtitle)*: FLIGHT TEST ORIENTED PRECOMPILER SYSTEM (FLTOPS) DESIGN SPECIFICATIONS.	5. TYPE OF REPORT & PERIOD COVERED: Final rept. May 1977 ~~through~~ August 1978
	6. PERFORMING ORG. REPORT NUMBER: SAI-78-061-DEN
7. AUTHOR(s): D. A. Otey, H. R. Ramsey, J. K. Willoughby	8. CONTRACT OR GRANT NUMBER(s): F04611-77-C-0040
9. PERFORMING ORGANIZATION NAME AND ADDRESS: Science Applications, Inc. 40 Denver Technological Center West 7935 Prentice Avenue Englewood Colorado 80110	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS: PEC 65805F
11. CONTROLLING OFFICE NAME AND ADDRESS: N/A	12. REPORT DATE: 28 August 1978 13. NUMBER OF PAGES: 318
14. MONITORING AGENCY NAME & ADDRESS *(if different from Controlling Office)*: N/A (12) 319p.	15. SECURITY CLASS. *(of this report)*: UNCLASSIFIED 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT *(of this Report)*

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

17. DISTRIBUTION STATEMENT *(of the abstract entered in Block 20, if different from Report)*

18. SUPPLEMENTARY NOTES

Contract sponsored by: 6510 TESTW/TEEDM
Software Management Branch
Air Force Flight Test Center
Edwards AFB CA 93523

19. KEY WORDS *(Continue on reverse side if necessary and identify by block number)*

Augmented Grammar, Lexical Analyzer, Precompiler, Program Design Language

20. ABSTRACT *(Continue on reverse side if necessary and identify by block number)*

FLTOPS is a tool to assist in setting up a generalized software system to meet the needs of a project. A simple and meaningful interface between the engineer and FLTOPS is provided. Based on engineer supplied input, FLTOPS generates code which will in essence be a generalized software system, however it will be tailored to project requirements.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

392 878

FOREWORD

This document is the final technical report for contract F04611-77-C-0040. It is submitted in response to requirement 3 (Data Item DI-S-3591/A/M) of the contract Data Requirements list.

This report describes work done to produce a detailed design of a software system called the Flight Test Oriented Precompiler System (FLTOPS). Since the study objective was to produce a detailed design for FLTOPS, this document takes the form of a rigorous specification. Appendix A contains a report on the validation of this specification. Appendix B contains additional information regarding changes/modifications incurred during the extended test phase of the contract. Appendix C contains listings of a test case and results of the test case.

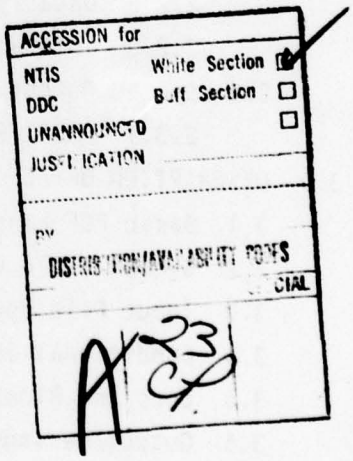


TABLE OF CONTENTS

		Page
1.0	INTRODUCTION	1- 1
1.1	Current Modus Operandi for Preparing Flight Test Analysis Software	1- 1
1.2	Objectives of the Flight Test Oriented Precompiler (FLTOPS)	1- 2
1.2.1	Reduce the Effort and Span Time for Configuring Project-Specific Software	1- 2
1.2.2	Provide a Tool for Customizing Software According to Project-Specific Requirements	1- 3
1.2.3	Increase the Visibility and Comprehension of Analysis Techniques Being Employed	1- 4
1.2.4	Provide Archival Documentation and Easy Reconstitution of Past Methods/Models	1- 5
1.2.5	Provide a Tool Which can be Maintained and Upgraded Easily	1- 5
2.0	OVERVIEW OF THE FLTOPS SYSTEM	2- 1
2.1	The FLTOPS Configuration File	2- 4
2.1.1	Block Structure	2- 4
2.1.2	Precompiler Control Statements	2- 4
2.1.3	Special FCF Rules	2- 5
2.2	The User Input File	2- 8
2.2.1	Hierarchic Structure	2- 8
2.2.2	Data Types	2- 9
2.2.3	Interactive Creation	2-11
2.3	System Output	2-12
2.3.1	PASS1/PASS2 Substitution	2-12
3.0	DESCRIPTION OF FCF CONTROL LANGUAGE	3- 1
3.1	Basic FCF Language Elements	3- 3
3.2	Symbol Table Operations	3- 8
3.3	Input File Operations	3-24
3.4	Conditional IF Statement	3-29
3.5	Loop and Block Structures	3-41
3.6	Output/Messages	3-45
3.7	Computing Precompile Time Values	3-51
3.8	Built In Functions	3-55
3.9	Debugging Aids	3-60
3.10	Switches and Counters	3-62

4.0 ILLUSTRATIVE EXAMPLE 4- 1
4.1 Description of Example 4- 1
4.2 The Input Specification 4- 5
4.3 The FCF 4- 7
4.4 The Precompiler Output 4-11
4.4.1 FORTRAN Output 4-12
4.4.2 Trace and Error Messages 4-13
4.4.3 Summary Statistics 4-21
5.0 SPECIFICATION MECHANISMS USED 5- 1
5.1 Augmented Grammar 5- 2
5.1.1 Syntactic Specification Technique 5- 2
5.1.2 Syntactic and Semantic Specification Technique . . . 5-11
5.2 Program Design Language (PDL) 5-15
5.3 State Transition Diagrams for Lexical Analyzers 5-20
5.4 Text and Glossaries 5-25
6.0 FORMAL SPECIFICATION OF FLTOPS DESIGN 6- 1
6.1 Discussion of Design Concepts 6- 1
6.1.1 Input Preprocessor 6- 1
6.1.2 Lexical Analyzers 6- 1
6.1.3 FCF Statement Processor 6- 3
6.1.4 Functional Primitives 6- 5
6.1.5 Symbol Table Storage 6- 7
6.1.6 Output Processor 6-10
6.2 Specification of Functional Primitives 6-12
6.3 Specification of Precompiler Functions 6-34
6.3.1 Syntactic Specification of FCF Statement Types . . . 6-55
6.4 Specification of Lexical Analyzers 6-59
6.4.1 FCF Lexical Analyzer 6-60
6.4.2 Input Lexical Analyzer 6-62
6.4.3 FORTRAN Lexical Analyzer 6-64
6.5 Glossary 6-67

7.0 IMPLEMENTATION ISSUES 7- 1
7.1 Symbol Table Implementation Options 7- 1
7.2 Alternatives for Implementing "Execute" Statement 7- 3
7.3 File Preprocessing 7- 6
7.4 Language and System Considerations 7- 7

8.0 REFERENCES 8- 1

APPENDIX A Validation of the Specification . . . . . A- 1
APPENDIX B Extended Validation Effort . . . . . B- 1
APPENDIX C LINK4 FCF and Test Results . . . . . C- 1

1.0 INTRODUCTION

1.1 CURRENT MODUS OPERANDI FOR PREPARING FLIGHT TEST ANALYSIS SOFTWARE

Each flight test project is both similar to and different from every other project that has been done. The similarities have resulted in the development of several general and integrated collections of computer codes to assist in analyzing the flight test data. Examples are AFTDS/DAPS, SANDS, UFTAS and GSAP. The development and maintenance of such generalized programs requires a community of professional programmers, software analysts, and computer scientists. The applications users for these codes are another community of flight test engineers and analysts whose expertise lies in the areas of aircraft performance and/or aircraft system or subsystem development.

The analyses associated with each new project would be exceedingly expensive if it were not for the existence and evolution of general software that is possible only because of the similarities among projects.

The differences that exist among analysis projects are, however, still great enough to require the allocation of several man-months to several man-years of effort to accommodate. The general codes must be assembled in unique sequences, the aircraft, atmospheric, geopotential, and kinematic parameters must be specified, engine and drag models must be coded for each new aircraft, etc. The accomplishment of this customization process requires both aircraft and software specialists working in a cooperative and sometimes an intricately dependent fashion.

Some of the problems that arise in this environment are the motivation for the FLTOPS concept. For example, the specification of some analysis options or particular models must now be done several different times. This redundancy is a waste of effort and can lead to inconsistencies and/or

errors that are difficult to detect. There exist so many details to specify that supervisory personnel cannot easily review the analysis techniques that have been implemented by the engineer/program analyst team. Of course, various levels of experience exist in both the software and engineering specialties. The current methods of documenting previous software configurations are inadequate; previous solutions and important insights are apparently too often stored only in individuals' memories.

The redundancy of effort, the tedious and meticulous activities required to find and make simple software changes, the inadequate documentation of the procedures used, and communications difficulties between the engineer and the software specialists are current characteristics of the AFFTC work environment that might be improved by proper exploitation of some additional software technology. This realization by AFFTC analysts led to the development of the Flight Test Oriented Precompiler System concept.

1.2 OBJECTIVES OF THE FLIGHT TEST ORIENTED PRECOMPILER SYSTEM (FLTOPS)

1.2.1 Reduce the Effort and Span Time for Configuring Project-Specific Software

The fundamental objective of the FLTOPS design and development is to effect a substantial reduction in the labor effort and the time span now required to assemble and customize existing data reduction programs. Each flight test project requires data reduction programs that are uniquely linked together and/or modified for the objectives and activities of that project and the characteristics of the aircraft. A high degree of standardization has been achieved by building programs that are quite generic and by providing special programs to construct project-specific data files and program sequence specifications. Complete standardization of flight test data reduction software is not possible; there will always be a need

for new code that has not existed previously and a need to use only a subset or a simplified version of the existing standardized code. These adaptations of the standard programs currently take man-months to man-years of labor to produce. Both flight test engineers and mathematician/program analysts are involved in this activity.

A review of the types of human activity involved in creating project-specific software from generic library routines suggests that major elements of the work are suitable for automation. It is possible, for example, to anticipate places within existing code where modifications are likely to occur. It is also possible to determine several completely definable options that might be selected and to specify a single option using a few key words or phrases. Thus, repeated reimplementations of similar logic or remodifications that are structurally equivalent need not be the burden of human analysts. Similarly, unnecessary repetitions of model or technique options can be eliminated as a human task. Many other examples can be cited, all of which suggest that the tasks associated with configuring project-specific flight test data reduction software can be made much less labor-intensive. A fundamental objective of FLTOPS is to provide computer capabilities that reduce unnecessary human labor and, thus, reduce the time and costs required to support flight testing.

1.2.2 Provide a Tool for Customizing Software According to Project-Specific Requirements

Although the major goal of FLTOPS is to reduce the labor necessary to configure software that will be adequate to support the analysis of the flight test data, it is expected that the software generated by the FLTOPS will use computer resources efficiently. General codes that contain logic that is unneeded in particular analyses need to be trimmed. Storage specifications that are super-sufficient should be reduced to be adequate, but not wasteful. Parameters that are normally computed but become constants in particular cases should be replaced by their numeric values, etc.

The effort required by software maintainers to accomplish these types of customizing functions is quite routine, requires great attention to detail, is difficult to accomplish with complete thoroughness, and may not be worth the time required. If, however, the FLTOPS can accomplish these functions, the benefits could be substantial. The nature of the customizing task is generally quite compatible with precompile-time automated capabilities. Therefore, an objective of the FLTOPS is to provide a means for modifying software so that analyses can be done with efficient use of computer resources. The accomplishment of this objective will, of course, imply the ability to do more analyses with the computer resources that are available.

1.2.3 Increase the Visibility and Comprehension of Analysis Techniques Being Employed

Before standardized computer programs were available, flight test engineers presumably needed to derive the mathematical formulae associated with each flight test project. These formulae served as the basis for the development of specific computer programs for the project. As libraries of programs developed and standardization increased, the need for the engineers to understand the details of the logic within already-developed programs declined. At the current time, it is possible to configure combinations of routines that may execute properly, but may not do precisely what the engineer thinks they do. Experienced analysts can detect misuses of existing software if they have good visibility of the program architecture. One of the needs that FLTOPS can fill is the need to make visible the choices a flight test engineer has made. The insights and experience of colleagues and supervisory personnel can then be applied to assure the integrity and appropriateness of the analysis that has been specified. Therefore, an objective of the FLTOPS system is to provide feedback to the analysis designers about the nature of the analysis procedures (in the form of software) that they have chosen. This objective can be thought of as a quality assurance objective.

1.2.4 Provide Archival Documentation and Easy Reconstitution of Past Methods/Models

A not-uncommon experience among flight test engineers and program analysts is the need to remember how a particular special requirement was handled in a past project. Or similarly, in reactivating software used in the past, the analyst needs to recall the reason that a certain software peculiarity appears. Perhaps the actual software may need to be retrieved and modified further for a new test project. Time and effort may be wasted unnecessarily because archival documentation is inadequate. This documentation requirement may be satisfied if the previous objective is satisfied (Section 1.2.3). Yet there are differences in purpose which may dictate different design properties. Thus, it should be stated explicitly that support of proper archival documentation of flight test analysis software is an objective of the FLTOPS development.

1.2.5 Provide a Tool Which Can be Maintained and Upgraded Easily

Another objective of the FLTOPS is the development of software that conforms to design principles that are consistent with existing applications software at AFFTC. FLTOPS software should, therefore, be as general as practical and have characteristics that desensitize it to a changing environment (e.g., new computing hardware, new functional requirements, etc.). The objectives described below are really summarized by saying that FLTOPS must satisfy certain design and implementation constraints.

It is necessary that the FLTOPS software be usable at Edwards AFB, but also at other computing establishments that use different operating systems, hardware, and language translators. To the maximum degree possible, the capabilities of the FLTOPS software should be provided with features that are common to all or most large computing environments.

Because FLTOPS software will provide some rather sophisticated capabilities, the portability objective will have to be emphasized in both the design and implementation efforts.

The capabilities foreseen for FLTOPS are not necessarily unique to flight test data reduction applications. It is desirable to avoid imbedding into the design or implementation of FLTOPS any characteristics of flight test data reduction that would limit the utility of FLTOPS in other application settings. The degree to which this objective can be met is not completely clear at the outset. Yet, it is clear that application-independence is a proper objective unless its achievement represents a large incremental development cost.

It is anticipated that the utility of the FLTOPS will increase as normal maintenance of existing software and the development of new standard software occurs. An objective of the FLTOPS development is to provide for evolutionary improvements and to avoid the need for costly redevelopments in which the bulk of the software is discarded and rewritten. This objective has implications about where capabilities are provided for in the original design. Its satisfaction has important cost implications to the Air Force.

2.0 OVERVIEW OF THE FLTOPS SYSTEM

The FLTOPS system was designed to fulfill a need. This need, although variously expressed by the individuals involved, is quite simply to get the job done more efficiently. The job, although encompassing many aspects as alluded to in Section 1, has as its focal point the capability to customize existing FORTRAN code to meet the specific requirements associated with current projects. This customization process should be fast, flexible, easy to use, and most importantly, capable to responding to the changing needs of a demanding environment.

The FLTOPS system design fulfills this need. At the center of the FLTOPS system is a new, high-level programming language similar in form and design philosophy to many other modern languages such as PL/1 and ALGOL. Programs written in this new language can read inputs, manipulate data values and produce output as do other high-level languages. The unique quality of the language, however, is that its primary purpose is to generate highly specific FORTRAN code based upon user requests.

The FLTOPS system accomplishes this task by providing users with a system which accepts FORTRAN code along with special instructions indicating how to select, modify, and/or create new code. The FORTRAN and special instructions reside intermixed in what is known as an FCF for FLTOPS Configuration File. The FCF can be thought of as a special program whose function it is to generate FORTRAN source code suitable for input to a compiler. This program can have its functions and, therefore, its output controlled by reading data from an input file. Although the primary output is FORTRAN source code, messages and data can also be output by the system.

This system is under control of a master program called the FLTOPS precompiler. The precompiler actually processes the FCF, or program, and performs the necessary functions of reading in data and outputing source code as instructed by the program requirements. A simple and analogous way of conceptualizing the precompiler is to think of it as a computing machine. Just as a computer executes the functions of a computer program,

the precompiler executes the functions indicated in the FCF. This psuedo-machine concept is schematically represented in Figure 1.

The FLTOPS system, thus, contains four distinct components. A computer program written in a mixture of FORTRAN and a special high-level language is commonly referred to as the FLTOPS Configuration File, or simply, FCF. This program, in turn, can read data from an input file. This data can be used by the FCF to directly or indirectly control output generation. The output is, in general, customized FORTRAN source code that is specifically designed by the user to meet his requirements. This total task is effectively accomplished by executing the FCF program on a pseudo-machine known as the precompiler.

In any computer system the program simply indicates in which order and in what quantities the basic machine operations should be carried out. The FLTOPS system works the same way. The precompiler performs all the functions of reading and writing and processing as directed by the FCF program. From a user's point of view, however, it is quite natural to refer to the program as "reading" inputs or "writing" outputs. In Figure 1, the FCF is shown as receiving and generating outputs even though technically the work is being accomplished by the precompiler.

A discussion of the FCF structure, the inputs it reads, and the outputs it generates follows. Because of its technical nature, a discussion of the precompiler design is postponed until Section 6.

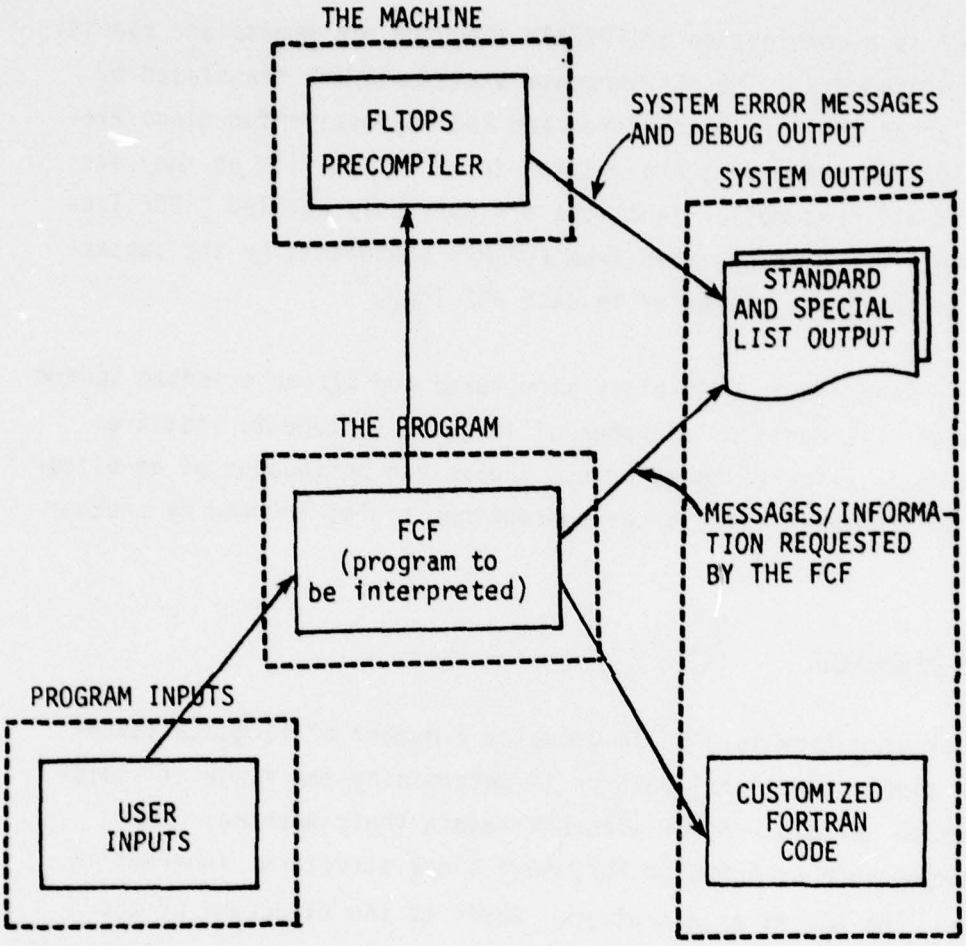


Figure 1. Basic Relationships Among FLTOPS System Components

2.1 THE FLTOPS CONFIGURATION FILE

The FCF is a combination of FORTRAN language statements and special FCF language statements. The FCF language statements are translated by the precompiler as they are encountered and the respective functions executed. The FORTRAN statements are written to an output file as they are encountered if all precompiler functions are currently enabled. FCF language statements are distinguished from FORTRAN statements by the appearance of a P as the first character in each FCF line.

The FCF language is of a block structured and stream oriented modern language design. It contains a number of language statements that are grouped into units referred to as rules. Rules can be thought of as either a main program (the main rule) or as subroutines (rules invoked by another rule).

2.1.1 Block Structure

A block structure is used in grouping a number of language statements into a single functional unit or in determining the range of statements over which certain program elements retain their meaning. Most modern languages such as ALGOL or PL/1 have block structures inherent in their design. The FCF is no exception. Basic to the structure of the language is the capability to group statements by function and conditionally bypass or execute the entire group. Such blocking capability combined with good programming practice can result in an FCF that is more easily understood by others and which is easier to modify.

2.1.2 Precompiler Control Statements

The most basic capability required of the precompiler is the conditional or unconditional inclusion of a block of FORTRAN code. To provide this, a number of FCF control statements with programming-language-like

syntax have been designed. As a result, the FCF control language can be thought of as a specialized interpretive programming language. Table 1 provides a summary of the statements and their basic function. As can be seen, the language elements provide input/output and processing capability to the FCF writer so that sophisticated tests and computations can be made to determine the form of the final FORTRAN output. Not shown, but invaluable in providing capabilities inherent in other languages, is the capability to use general arithmetic as well as boolean expressions as elements of the language.

Precompiler control statements can be intermixed with FORTRAN statements within the FCF. Clearly, the control information is different in type from the FORTRAN and is physically distinguishable by the statement names, as well as by the presence of a "P" as the first character in each line of the FCF that is used for control statements. Although control statements need not be on separate lines from other control statements, the readability of the FCF is enhanced by good programming style.

The precompiler is designed to operate on a single mixed file. However, the control statement "INVOKE" allows for large blocks of code, possibly consisting of just FORTRAN statements, to be physically separated from the main rule or procedure.

2.1.3 Special FCF Rules

The precompiler is designed to process a main rule or program. When the end of this main rule is encountered, the precompiler stops processing the FCF. The writer of the FCF, however, has the same capability afforded by most other programming languages of transferring control of the precompiler to another rule. When this new rule is complete, control is transferred back to the rule from which it was called. This process is accomplished through use of the control statement INVOKE. So, just as in

TABLE 1. SUMMARY OF FCF LANGUAGE STATEMENTS AND BUILT-IN FUNCTIONS

FUNCTION	LEGAL FCF STATEMENTS
SYMBOL TABLE OPERATIONS	DEFINE ENTER CATENATE REPLACE DELETE CLEAR_TABLE SYMBOL_EXISTS TABLE_EMPTY SET INCREMENT DECREMENT DO FOR SYMBOL TABLE ... END
INPUT FILE OPERATIONS	INPUT FIND DO FOR EACH SUBNODE OF ... END NODE_EXISTS SAVE_POINTER RESTORE_POINTER
CONDITIONAL	IF ... THEN ... IF ... THEN ... ELSE
BLOCK/LOOP	DO ... END DO WHILE ... END INVOKE
OUTPUT	OUTPUT MESSAGE
COMPUTE PRECOMPILE TIME VALUES	PARAMETERS EXECUTE
BUILT-IN FUNCTIONS	INDEX LENGTH SUBSTRING VALUE
DEBUG	ASSERT STATUS

FORTRAN, PL/1 OR ALGOL, the programmer can set up arbitrarily complex program structures. In the FCF, this allows multiple use of certain groups of precompiler control and/or FORTRAN statements. As in other languages, there, undoubtedly, will be control functions that can be used by other FCF rules. Such a subroutine or procedure can be used by the FCF writer to make programming tasks much easier. Most likely, a library of special FCF rules could be used as in other languages to facilitate the ease with which an FCF could be built.

A word of caution is in order with respect to using special FCF rules. The precompiler design assumes all values are known to and accessible by every rule. That is, all FLTOPS variable values are essentially global in scope. If restricted access is required, the FCF writer is responsible for establishing, on his own, mechanisms for informing subordinate rules of where information they might need is located. Adequate FCF control statements have been provided to allow this.

FCF rules are also allowed to appear in the INPUT file. This allows a temporary change to be made to an FCF rule by simply copying it into the INPUT file and there making the necessary changes. The precompiler would then process an invocation of a special rule by checking all places the rule could be found in the order of a predetermined priority. The most likely priority would be to check the INPUT file, the current FCF, and lastly, an attached library file, using the copy of the rule found first.

2.2 THE USER INPUT FILE

As in standard programming languages, the FCF writer can use an input file to ultimately customize eventual program output. The precompiler design assumes two basic characteristics of the input file; one, that it is hierarchic in nature; and two, that there is a well defined set of legal data types. For a particular application, the detailed nature of the input file is determined by the FCF which processes it. Thus, the FCF is designed so as to produce the desired input file properties. Thereafter, variation of only the input file allows customization of the FORTRAN output by the precompiler.

2.2.1 Hierarchic Structure

User input to the FCF is a hierarchic arrangement of legal FCF input data types. Each lower level is indicated by indenting exactly three spaces. Lower levels are logically subordinate to the higher level that immediately precedes them. Thus, for example,

```
CONSTANTS
   A=1.0
   B=7.2E+7
   C=0
```

denotes that A, B, and C are all constants. The only restriction on the number of levels possible for this type of hierarchy is the allowable line length. Indentations of more than three spaces are regarded as continuations of the previous line. As an example of how continuation statements might be utilized, consider this example of reading tabular data.

```
DATA
   X=3
   Y=19.E+3
   TABLE_FOR_CMALFA
        CMALFA    MACH
        -.257     0.6
        -.770     0.65
        -.3       0.7
   CMO=-.1526
```

This type of hierarchic arrangement provides an outline-type appearance which is easy to read as problem documentation.

2.2.2 Data Types

The content of the input file is restricted to legal data types. These consist of numbers (with or without a sign), identifiers, strings, special symbols, comments and special FCF rules. Identifiers begin with an alphabetic character and may contain any alphanumeric character and underscore (_). An identifier cannot end with an underscore. Character string literals are any characters, including blanks, enclosed within a pair of apostrophes ('). Comments are any characters starting with /* and ending with */. Special symbols that can be read directly, i.e., not enclosed in apostrophes, are * + - / ** ( ) = >< ; @ $: . Any other special character must be input as a character string literal.

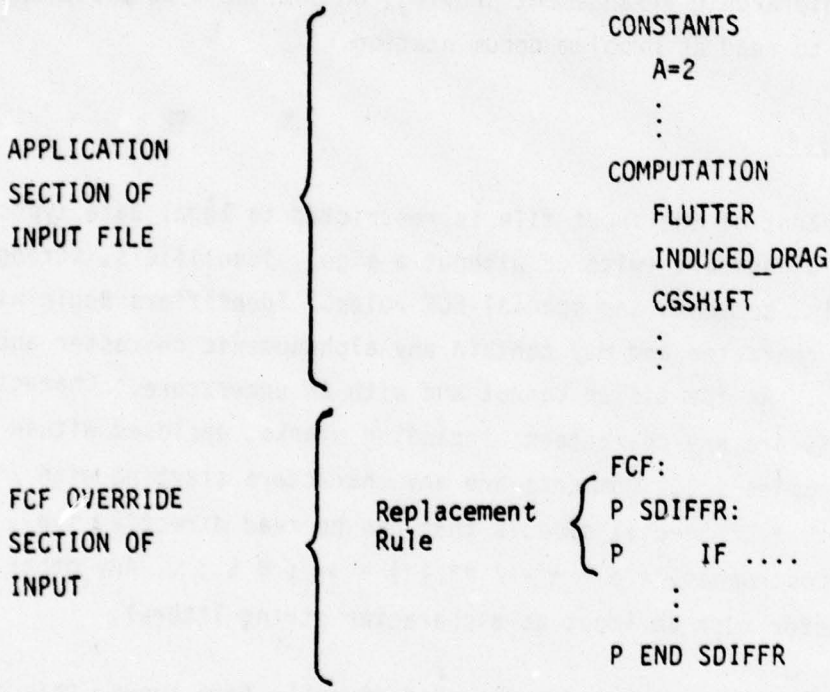
Since algebraic symbols can be read directly from input, this allows the FCF writer to transfer arithmetic assignment statements directly from input to FORTRAN output. For example:

```
DATA
   DELTA = 1.9
   GRAV=-XMU*XM1*XM2/R**2
```

would be suitable since the DELTA and GRAV formulae contain only valid input data types.

Note, however, that the precompiler does not check FORTRAN statements for legal syntax. Hence, assignment statements appearing in the input file should use legal FORTRAN syntax.

If it is necessary to substitute FCF rules for those in the permanent FCF, or to add new rules to be used temporarily, this may be done by placing the new rules in the input file. A special section of the input exists for this purpose. This special section is set off from the application section of the input as illustrated below.



In the FCF override section of the input, the format is identical to that used in constructing the FCF.

2.2.3 Interactive Creation

Creation of the input file to be used by any particular FCF program will require special knowledge of the program requirements. As in any large program, where inputs may be many and variable, it is important to document these input requirements accurately and in detail. To avoid errors in input and to reduce the tedium associated with input structuring, many codes use interactive build techniques to assist users in providing the necessary inputs to the system. Inputs to the FCF are ideal candidates to be created interactively. Considerable thought has been given to including explicit interactive capabilities in the current design. Because of the interpretive nature of the FLTOPS precompiler, to explicitly include capabilities to build input interactively provides questionable utility at considerable cost.

However, the capability to interactively build input files under control of non-FLTOPS software provides a desirable alternative. This capability currently exists in a rudimentary sense with any text editor program available with an interactive system. The degree to which such an editing system should be enhanced is a design issue that is controlled both by the implemented input file structure and the form of the inputs actually used. Until more knowledge is gained in both these areas, an interactive input creation program cannot take an explicit form. It is, however, a very desirable option and could be constructed as a separate program when sufficient information is available.

2.3 SYSTEM OUTPUT

The FLTOPS system generates three forms of output: a printed list output, customized FORTRAN source code, and card images of arbitrary content. The printed output consists of user and/or system generated error messages, optional source listings and various types of debugging/trace information at the user's request. Although important in testing, validation and documentation of the FCF program, list output is not the primary output of the FLTOPS system. The goal is customized FORTRAN code and the FLTOPS system provides a number of mechanisms for the user to produce this code.

The user has the capability to output whole FORTRAN statements directly or synthesized FORTRAN statements built up by the FCF program. Each FORTRAN statement is then checked by the precompiler before being actually written to the output file to perform any substitutions desired by the user. These output substitutions can actually occur twice. Once during actual execution of the FCF, referred to as Pass 1 substitution and again in a post processing operation, that strictly reviews FORTRAN code, known as Pass 2 substitution. More detail on these FORTRAN output mechanisms is given in the following section.

The FLTOPS control language is also capable of producing output in a strict card image format. Known as the DATA output mode, elements written out in this mode are not altered in any way by the precompiler. No pass 1 or pass 2 substitutions are made and no column adjustments are made to conform to FORTRAN syntax requirements. More detail on use of the DATA mode is given in the discussion of the FLTOPS OUTPUT software switch in Section 3.10.

2.3.1 PASS1/PASS2 Substitution

As FORTRAN statements are encountered in the FCF, or as they are synthesized by the user, they are stored in an output buffer. Note that all FORTRAN output is inhibited if the precompiler is, in fact, not fully operational (see Section 3.10, for more details on the precompiler FUNCTION switch). During the processing of the FCF, each individual element in

the FORTRAN statement currently in the output buffer is compared against entries in the PASS1 symbol table. If the FORTRAN element matches a symbol in the symbol table, the value in the PASS1 table is substituted for the FORTRAN element before it is written to the output file. In this way, any FORTRAN statement can be altered by the user to meet his requirements.

The PASS1 symbol table is a standard precompiler table that the user can access just like any other symbol table. Symbol entries in the PASS1 table are substituted for their FORTRAN counterparts for as long as they are in the table. They can be deleted as a normal part of the FCF processing at any time. The PASS1 table is very helpful for inserting values, variable names, special I/O formats, etc., that may be read directly from the input file or computed internally by the FCF program.

Pass 1 substitution occurs as the FCF is processed by the precompiler. This means that only those elements can be replaced whose values are known beforehand. This is not always adequate, however. For example, if the user wants to customize a FORTRAN common statement, all variables and their dimensions may not be known at the time the first common statement is to appear. As a result, Pass 1 substitution may not provide a convenient or adequate mechanism. To provide greater flexibility, the precompiler provides another standard table known as the PASS2 symbol table.

After the FCF has been processed, all the FORTRAN code that was written out is looked at again if the PASS2 table is non-empty. This time, however, token substitution is made from the PASS2 table. Although PASS2 table entries can be deleted or modified during pass 1, they have no effect on FORTRAN written out until the FCF program has been completed. At this point, all entries in the PASS2 table are substituted for the corresponding FORTRAN token wherever they occur in the file produced in pass 1. Since no changes can be made to the PASS2 table during substitution, all changes are global in scope.

See the example in section 4 for uses of pass 1 and pass 2 substitution.

3.0 DESCRIPTION OF FCF CONTROL LANGUAGE

The FCF control language provides FLTOPS precompiler users with a flexible and powerful means of customizing FORTRAN code. Combining standard programming language constructs with the manipulation of symbol table entries and a set of built in functions and user accessible switches, the FCF language provides users with a powerful, yet simple, set of instructions to accomplish customization tasks. Included in the FCF language are the capabilities to conditionally execute blocks of FCF code based upon arbitrarily complex boolean expressions, to compute complex arithmetic expressions, as well as execute already compiled FORTRAN routines, and to read data from a user defined input file, as well as output predefined and synthesized FORTRAN code. These, as well as many other language features, are discussed in more detail in the following sections.

Table 2 gives a summary and simplified categorization of the FCF control language statements. Note that in addition to the statements listed, precompiler function is also affected by the values contained in several software switches. These switches are discussed in detail in Section 3.10. Of the statements, several belong to more than one functional category. This is so because of the complementary nature of the language design. Of special mention are those statements indicated as performing a test function. Even though testing is not their principal function, each can be used to control FCF processing. Each test statement can be used either by itself or in a boolean combination. When used by itself, the statement condition must be met or an optional user supplied FCF statement can be processed. The result for the user is a flexible set of instructions that he can use to insure the integrity of FCF and input file processing. Each statement is discussed in the section representing its primary function except those indicated as built in functions which are discussed in Section 3.8.

TABLE 3. SUMMARY OF FUNCTIONS FOR FCF CONTROL STATEMENTS

X = Primary Purpose
O = Secondary Purposes
* Built In Functions

REFERENCE	STATEMENT	FUNCTION CATEGORIES								
		SYMBOL TABLE OPERATION	INPUT FILE OPERATION	CONDITIONAL	BLOCK/LOOP	OUTPUT	COMPILE TIME VALUES	DEBUG	TEST	STRING MANIPULATION
3-60	ASSERT							X	O	
3-9	CATENATE	X								O
3-11	CLEAR_TABLE	X								
3-11	DECREMENT	X					O			
3-12	DEFINE	X								
3-15	DELETE	X								
3-42	DO...END				X					
3-23	DO FOR EACH SUBNODE OF...END		X		O					
3-16	DO FOR SYMBOL TABLE...END	X			O					
3-43	DO WHILE...END			O	X					
3-15	ENTER	X								
3-54	EXECUTE						X			
3-29	FIND		X						O	
3-39	IF...THEN			X						
3-39	IF...THEN...ELSE			X						
3-19	INCREMENT	X					O			
3-55	INDEX*									X
3-30	INPUT		X						O	
3-44	INVOKE				X					
3-56	LENGTH*									X
3-49	MESSAGE					X				
3-36	NODE_EXISTS		X						O	
3-45	OUTPUT					X				
3-52	PARAMETERS						X			
3-20	REPLACE	X								O
3-38	RESTORE_POINTER		X							
3-38	SAVE_POINTER		X							
3-21	SET	X								
3-57	SUBSTRING*	O								X
3-61	STATUS					O		X		
3-22	SYMBOL_EXISTS	X								
3-23	TABLE_EMPTY	X								
3-58	VALUE*	X								

3.1 BASIC FCF LANGUAGE ELEMENTS

The FCF language consists of a number of control statements that can be combined in a variety of ways with basic language elements to form syntactically correct constructs. The sections that follow discuss not only the uses of the various control statements, but also the correct syntax for each. Basic to understanding the syntax of these commands is a comprehension of the notation conventions of the definitions. The following notation conventions are adopted for consistency and ease of comprehension.

< >	Items appearing within these symbols refer to an element such as another FCF statement or a basic language element defined below.
[ ]	Items appearing within these sumbols are optional and appear only at the discretion of the user.
<...>	Special notation indicating that the syntax pattern already established can be repeated as often as necessary.

Any character appearing in a syntax description not bracketed by < > must appear exactly as written.

Many concepts that help in understanding the use of the language are not language statements themselves, but elements of the language. Although a rigorous syntactic description of all language elements is given in Section 6, an informal discussion of these basic concepts is provided below to make clearer the proper use of the FCF language statements.

*	In the context of symbol table operations, the asterisk is a special symbol that refers to the last data element successfully read from the input file. In arithmetic expressions, it refers to multiplication. Example: See ENTER, SUBSTRING, etc.

@ — The "at" sign is a special symbol that refers to the current pointer value. In the context of symbol table operations, it refers to the symbol table pointer. In the context of input file operations, it refers to the input file pointer.
Example: See ENTER and FIND

<word> — Any alphabetic character followed by zero or more alphanumeric characters or underbars (_). A <word> cannot end with an underbar, nor can two underbars be joined together. A <word>may begin with a pound sign (#) as long as there is at least one alphanumeric character following it. The # by itself is not a word . A word beginning with # is most useful for establishing symbol names that will be used in pass 1 or pass 2 substitutions.
Examples: ABC A_B X7B #A_B

<number> — Any integer, floating point, or exponential representation of a numeric quantity.
Examples: 7 8.93 1.02E-5

<string> — Any characters that appear between standard string delimiters (normally ').
Examples: 'THIS IS A STRING'
'ANOTHER STRING*$+.'

<statement> — Any legal FCF language statement, that appears in Table 3 except for built in functions. Built in functions can be used as a part of many statements but cannot be used by themselves.
Examples: SYMBOL_EXISTS
TABLE_EMPTY
IF...THEN

⟨table name⟩ An FCF element that identifies the name of a table where values can be/are stored. It is most often a ⟨word⟩, but can be any ⟨string expression⟩. For example, an asterisk or VALUE function can be used to reference a table name read from input or symbol table respectively.

Examples: TABLE_17
TABLE_NAME

⟨symbol name⟩ An FCF element that identifies the name of a symbol with which values are associated. It is most often a ⟨word⟩ but can be any ⟨string expression⟩. An asterisk (*) or VALUE function may be used to reference a symbol name read from input or a symbol table respectively.

Examples: SYMBOL_17
SYMBOL_NAME

⟨arithmetic expression⟩ Any algebraic combination of a VALUE, LENGTH, or INDEX function, ⟨number⟩ or ⟨word⟩ or ⟨word⟩with ⟨subscripts⟩. Parentheses can be used to determine the order of operations. A ⟨word⟩ with ⟨subscripts⟩ is assumed to be a ⟨table name⟩ ; a ⟨word⟩ *without* ⟨*subscripts*⟩ is assumed to be a ⟨symbol name⟩ which is an entry in the current default symbol table The order of evaluation of these expressions follows standard FORTRAN conventions.

Examples: 7 * VALUE (A,B)
A**5*(B+8) * X(5)

⟨string expression⟩ A string expression is a ⟨number⟩ , a⟨word⟩, a⟨word⟩ with⟨subscript⟩ , a ⟨string⟩, an * (asterisk) referring to the last element read from input, or an ⟨arithmetic expression⟩. It can be simply a VALUE function or a SUBSTRING function.

Examples: 'STRING EXPRESSION'
7.95321E10
A * VALUE (TABLE1, SYMBOL2) + 5
WORD
VALUE (X,Y)
SUBSTRING ('ABC', 1,1)

⟨value⟩ The quantity that is actually entered into a symbol table. It can be any ⟨string expression⟩. If the ⟨value⟩ is simply a ⟨word⟩ then it is interpreted as referring to a symbol entry in the default symbol table. The table entry corresponding to this symbol is retrieved and used as the ⟨value⟩.

Examples:
```
'THIS IS A SYMBOL ENTRY'
97.5 + A*B
A  (i.e., refers to the value of symbol A)
'A'  (i.e., refers to letter A as a value)
B(5,3)
```

⟨subscripts⟩ Up to three occurrences of an ⟨arithmetic expression⟩, separated by commas and enclosed with parentheses.

Examples:
```
(7, 10, 2)
(7 * N)
(2, VALUE(Y,X) + 1)
```

⟨format⟩ The format expression allows the user to transform the preceding language element to the desired form. It consists of either the word FORMAT with a parenthesized FORTRAN compatible FORMAT element or the special shorthand notation => (an arrow) followed by a FORTRAN compatible FORMAT element.

Examples:
```
FORMAT(A8)
FORMAT(I5)
=> F10.4
=> E12.5
```

⟨input tree⟩ A ⟨word⟩ or @ followed by zero or more occurrences of . (period) and a ⟨word⟩.

Examples:
```
INPUT.TREE.NODE
@.NEXT_LEVEL.DOWN
```

〈boolean expression〉

The boolean expression is the most complicated language element. It consists of relational expressions, SYMBOL_EXISTS and TABLE_EMPTY statements joined by AND, OR and NOT. It may also be a single INPUT statement. Lastly, it may be NODE_EXISTS or FIND followed by an 〈input tree〉 specification enclosed in parentheses. Except for the INPUT statement, these elements may be joined in arbitrarily complex expressions using parenthesis as logical separators. The order of completion of these expressions is left to right with parenthetical groupings taking precedence.

Examples:
```
NODE_EXISTS (INPUT.TREE.NODE)
INPUT ('A')
VALUE (A,B)>= 0
7 * VALUE (N) + 1<= VALUE (X)
SYMBOL_EXISTS (A,B) AND VALUE (A,N)>0
NOT (VALUE(A,B) = 7 AND SYMBOL_EXISTS (B,C))
FIND (A.B) OR VALUE (X,Y) = 0
```

3.2 SYMBOL TABLE OPERATIONS

Central to the capabilities of the FCF language is the ability to store and manipulate entries within a symbol table. Symbol tables are collections of entries that are referenced by either a table name / symbol name combination, or a table name / subscript combination, or simply by symbol name. These table entries are called values and are string literals of arbitrary length.

Several symbol table operations allow the use of a shorthand notation in which the table name need not be explicitly supplied in the statement syntax. In these instances, the symbol is assumed to be an entry in what is known as the default symbol table. Most symbol table operations can explicitly reference the default symbol task by using the key word DEFAULT in the table name field of the statement or implicitly by the syntax of the statement. Predefined as the FLTOPS symbol table, the default symbol table can be reset by the user with the DO FOR SYMBOL TABLE command.

All symbol table operations which alter the contents of a symbol table have an optional format control capability. This format control allows the user to specify the format that the data is to take before it is entered into the symbol table. Thus, if an arithmetic calculation is to be made, a format statement that immediately follows the computation will insure that the data is entered in the form needed by the user. The format specification itself can be any legal FORTRAN format specification such as A10 or I5. The exact nature and breadth of formating capabilities is left as an implementation option. At the very least, the A,E,F, and I FORTRAN format specification types should be implemented.

The following are descriptions of the table manipulation statements available to users.

CATENATE

SYNTAX: CATENATE(<table name>, <symbol name> <value>[<format>] [, <...>])
or CATENATE (DEFAULT, <symbol name>, <value>|<format>||, <...>|)
or CATENATE(<table name> <subscripts>, <value> [<format>] [, <...>])
or CATENATE(@, <value> [<format>] [, <...>])

The CATENATE statement allows the user to append additional value strings onto existing table entries. If a table location has no value associated with it, the CATENATE behaves as an ENTER command. As many values can be catenated to a single location with one command as desired. Any value catenated can be followed by a format specification.

Examples:

```
ENTER ( A, B, 'THE NEW ')
CATENATE ( A, B, 'AIRPLANE ')
```

The string AIRPLANE is appended to the current value entry for symbol B in Table A, making the string THE NEW AIRPLANE.

```
ENTER ( A, C, 'FLY ')
  :
CATENATE ( A, B, 'CAN ', VALUE(A,C))
```

The words CAN FLY are catenated to the contents of symbol B in table A.

```
SET ( N = 8.4 )
SET ( M = 4 )
ENTER(A,B)
  :
CATENATE ( @, 'FASTER THAN MACH', N/M FORMAT(F3.1))
```

The string FASTER THAN MACH 2.1 is appended to the current contents of A,B. Note that the arithmetic expression is evaluated, then converted via the format F3.1 to the desired form. If the format statement had

not appeared, the actual string catenated could have been something similar to 2.09999998 due to round off error in the calculation.

CATENATE (DEFAULT, AB, VALUE (A,B))

Catenated at the end of symbol AB in the currently defined default symbol table are the contents of symbol B in table A.

CLEAR_TABLE

SYNTAX: CLEAR_TABLE_TABLE (<table name>)

All entries for the specified table name are deleted. This action frees up table space used for unneeded or old tables. All symbols and/or associated values are removed from the table. The switches and flags that are predefined entries in the FLTOPS symbol table cannot be deleted or cleared, although user supplied entries in the FLTOPS table can.

Example:

CLEAR_TABLE(A)

Table A no longer has any symbol and/or value entries.

DECREMENT

SYNTAX: DECREMENT(<symbol name>)

A value of one (1) is subtracted from the current value associated with the indicated symbol in the currently defined default symbol table.

Example: DECREMENT(N)

If the current value of N was 1, then the new value of N would be 0.

DEFINE

SYNTAX: DEFINE <table name> <subscripts> [,<...>]
DEFINE <table name> AS <table name> [,<...>]
DEFINE <table name> AS NULL [,<...>]

The DEFINE statement allows FCF users to establish, or remove, special characteristics identified with the indicated symbol tables. By use of the DEFINE command the FCF user can either establish dimensions for symbol tables or, an empty or not yet established table can be equivalenced to a new or existing table so that table entries can be referenced by either table name.

By establishing dimensions for a symbol table, the user can reference the associated table entries by subscript rather than by symbol name. As such, these defined tables are similar in appearance and use to array variables in other programming languages. Standard subscripting conventions are in effect. Any tables whose entries are accessed via subscript notation cannot have entries accessed by symbol name and vice versa. Such DEFINE statements can appear anywhere. The same table name having dimensions assigned more than once uses those established in the last assignment. All current table entries are deleted when a table is defined to be a subscript accessible table. Thus, if a table were being used in which its entries were referenced by various symbol names and appeared in a DEFINE statement as a subscripted table, all its current entries would be automatically deleted.

Another use of the DEFINE statement is to establish table name equivalences. More accurately referred to as the DEFINE...AS command, this statement enables users to refer to the same table entries by more than one table name. The first table name in the list must be a new (not previously used table) or a table that is empty or a table which has been previously equivalenced to some other table. The second table name, immediately following the AS, is the table to which the first is equivalenced. There are no restrictions on the prior defined charac-

teristics of this table. Each reference to the first table name actually performs operations on entries in the table associated with the second table name. Many table names can be equivalenced to a single table either directly or through an "equivalence chain". (See example below). Note any cycles in an equivalence chain are indicated as severe errors.

Both the dimension and equivalence characteristics can be removed at any time by use of the DEFINE...AS NULL command. The table name identified by this command has its dimensions and any direct equivalences removed. If the table so named is a "real" table, i.e., one that has not been equivalenced or one that is the last table in an equivalence chain, all of its entries are also deleted. Note that in this instance a general warning message is issued indicating the table has been emptied.

Example:

```
SET ( N = 20 )
  :
  :
DEFINE X (N, 3 )
          Table X is defined as a 20 x 3 matrix.  (Note that the
          value of N (i.e., 20) is retrieved from the default
          symbol table.)
DEFINE INDEPENDENT AS X
DEFINE DEPENDENT AS Y
DEFINE DEPENDENT ( N + 2 )
          A table which is identified as the independent variable
          is associated with table X.  All characteristics of X
          are assumed by INDEPENDENT, and similarly for DEPENDENT
          and Y.  Table DEPENDENT and consequently table Y is
          defined as a 22 element array.  The dimensions are
          actually associated with Y
SET ( I = 0 )
DO WHILE I<= N
INCREMENT(I)
ENTER (DEPENDENT(I), INDEPENDENT (I,1) * INDEPENDENT (I,2) +
  INDEPENDENT (I,3))
```

END

All entries directed into table DEPENDENT are actually being stored in an area associated with the table Y. DEPENDENT is a pseudo-table in that it simply points to Y.

DEFINE DEPENDENT AS NULL

A reference to DEPENDENT no longer will retrieve/store entries from table Y.

DEFINE TABLE AS INDEPENDENT

An equivalence chain is established. TABLE points to INDEPENDENT which in turn points to X. A reference to TABLE actually references entries in X.

DELETE

SYNTAX: DELETE(<table name>, <symbol name>)
or DELETE(<table name> <subscripts>)
or DELETE(<symbol name>)

The DELETE statement provides the capability to delete individual table entries. Both the symbol (if applicable) and the associated value entry are removed from the table.

Example:

```
DELETE(A,B)
DELETE(X(17))
DELETE(N)
```

Both the 17th value in Table X and symbol B with its associated value in Table A are removed. Symbol N is deleted from the default symbol table.

DO FOR SYMBOL TABLE

SYNTAX: DO FOR SYMBOL TABLE table name

```
        :
        :

END
```

The DO FOR SYMBOL TABLE block construct allows the user to change the definition of the default symbol table name to the indicated table. The use of the default symbol table within the SET, INCREMENT, DECREMENT, DELETE VALUE, REPLACE and CATENATE statements can then be expanded to any table not using subscripts to reference its contents. Since tables accessed via subscripts have no symbols, it is meaningless to use a default symbol table with such tables. The default symbol table name is initially set to FLTOPS. Nesting of DO FOR SYMBOL TABLE blocks is allowed to an arbitrary depth. On encountering the END statement for the block, the default symbol table reverts back to the previously defined default symbol table.

Example:

```
DO FOR SYMBOL TABLE A
      :
      :
SET(SWITCH='ON')
      :
      :
END
```

The value of symbol SWITCH in Table A is set to ON. At the end of the block, the default symbol table is reset to FLTOPS.

```
DO FOR SYMBOL TABLE A
SET ( N = Ø )  /* N in table A set to Ø */
DO FOR SYMBOL TABLE B
      :
      :
SET ( N = Ø )  /* N in table B set to Ø*/
DO WHILE N<=1Ø
```

```
INCREMENT(N)  /*1 added to N in table B */
    :
    :
END
        /* The inner default symbol table definition ends and a 1
        is added to N in table A */
INCREMENT(N)
    :
    :
END
        /*A no longer is the default symbol table */
```

This example shows the use of nested default symbol tables. Note that N in symbol table A is not affected by the increments performed in the inner DO FOR SYMBOL TABLE block.

ENTER

SYNTAX: ENTER(<table name>, <symbol name>, <value> [<format>])
or ENTER(<table name>, <symbol name >)
or ENTER(<table name> <subscripts>, <value> [<format>])
or ENTER(<table name> <subscripts>)
or ENTER(@, <value> [<format>])
or ENTER (DEFAULT, <symbol name> , <value> [< format>])
or ENTER (DEFAULT, < symbol name>)

The ENTER statement allows the user to enter values into symbol tables. Symbol tables are referenced in either of two ways: a table name followed by a symbol name or a table name followed by subscripts. In the latter case, the table name specified must have appeared in an earlier DEFINE statement or a fatal error occurs. When no <value> occurs in the argument list, the table name and symbol name (or subscript location) are located but no value entry is made. Use of the ENTER statement with @ as the first argument would enter a value at the last symbol table position that was referenced. Before values are actually entered, they can be reformatted according to the specification shown in the optional format statement. The user can use the key word DEFAULT to enter data into the default symbol table. A short hand form that accomplishes the same thing is provided through the SET command.

Examples:

```
ENTER(A,B,'C')
```

In this example, string C is entered as the value for symbol B within Table A.

```
ENTER(X(17), VALUE(A,Y)*2=> I5)
```

In this example, the value of Y in symbol table A is multiplied by two and the product is entered as the 17th value in Table X. The value is formatted in an I5 format before being entered into X(17).

```
ENTER (A,F)
  :
  :
ENTER(@, 'DELAYED ENTRY' FORMAT(A15))
```

Symbol F in Table A is located; string DELAYED ENTRY is entered as the value for F if no other symbol table operations were performed between them. Note that the original string has 13 characters and an A15 format is specified. In this instance, two blanks would be appended to the string before it was stored.

INCREMENT

SYNTAX: INCREMENT(<symbol name>)

The INCREMENT statement adds one (1) to the value of the indicated symbol in the currently defined default symbol table.

Example: INCREMENT(N)

If the value associated with N was 1, then the new value after this statement would be 2.

REPLACE

SYNTAX: REPLACE (<table name> , <symbol name> , <value> , <value> [<format>] [,<arithmetic expression>])

or REPLACE (DEFAULT, <symbol name> , <value> , <value> [<format>] [,<arithmetic expression>])

or REPLACE (<table name> <subscripts> , <value> , <value> [<format>] [, <arithmetic expression>])

or REPLACE(@, <value> , <value> [<format>] [,<arithmetic expression>])

The REPLACE statement provides the capability to do substring replacement within already existing table entries. The first <value> string is the string to be replaced. If this string is a substring of the identified table entry, it is replaced with the optional second <value> string. This second string can optionally be formatted according to the format supplied by the user before being inserted. If the first <value> is a null string (i.e., ''), the second <value> is catenated at the beginning of the value in the symbol table. The REPLACE statement does a global replace of all occurrence of the first <value> at the indicated location. If the optional <arithmetic expression> is present, the <arithmetic expression> is evaluated, and truncated to integer form. The resultant integer reflects the number of occurrences of the first <value> that are to be replaced by the second <value>.

Examples:

```
ENTER(A,B,'EXAMPLE-1')
REPLACE(A,B,1,-3.1*2.3 =>I2)
```

The entry for symbol B in Table A is changed to EXAMPLE--7. Note that the calculations are performed; then the resultant value reformatted into a two digit integer format.

```
ENTER(A,B)
REPLACE(@,'-','',1)
```

This results in symbol B in Table A having the value EXAMPLE-7, i.e., the first - was deleted.

```
REPLACE(A,B,'','NEW ')
```

This results in NEW being catenated at the beginning of the symbol table entry. So that the entry for B in Table A becomes NEW EXAMPLE-7 .

SET

SYNTAX: SET(<symbol name> = <value> [<format>])
or SET(<symbol name>)

The SET command is a shorthand form of the ENTER command. The symbol is assumed to be in the currently defined default symbol table. The second form of the SET command simply enters the symbol into the default symbol table and positions the SYMBOL_TABLE_POINTER accordingly.

Examples:

```
SET(B = 8*X(7) + 15 * VALUE(A,B) FORMAT(F10.4))
SET(FUNCTION = 'SYNTAX')
```

The first statement sets the value for symbol B in the default symbol table to be the result of the arithmetic expression. The format statement insures the user the result is in a form he can use. The second statement sets symbol FUNCTION in the default symbol table to be the string SYNTAX. An alternate way of doing the second example might be:

```
SET(FUNCTION)
  :
ENTER(@,'SYNTAX')
```

The SET command here simply positions the SYMBOL_TABLE_POINTER for subsequent entry of the indicated value.

SYMBOL_EXISTS

SYNTAX: SYMBOL_EXISTS(<table name>, <symbol name>) [ERROR:<statement>]
SYMBOL_EXISTS(<symbol name>) [ERROR: <statement >]

The SYMBOL_EXISTS provides the capability for the user to test to insure that the specified symbol name is in fact an entry in the desired table. In the first form of the statement the table name is explicitly provided by the user. In the second form, the symbol is tested to see if it exists in the currently defined default symbol table. If the symbol is not in the table, the optional ERROR: condition is executed. If the ERROR: condition is not present, a default error message is given. If the symbol is present, the ERROR: condition is bypassed. A SYMBOL_EXISTS statement may appear in a boolean expression, but without the optional ERROR: condition.

Example:

```
SYMBOL_EXISTS(A,B) ERROR: DO
                           :
                          END
SYMBOL_EXISTS(N)  ERROR:  MESSAGE ('GW: N NOT AVAILABLE')
```

If symbol B has not been entered into Table A, the ERROR: condition is taken and the DO block executed. Otherwise, all statements within the DO block are bypassed except for syntax checking. In the second SYMBOL_EXISTS construct, the symbol N is tested to see if it is available in the default symbol table; if not, the message is issued.

TABLE_EMPTY

SYNTAX: TABLE_EMPTY(<table name>) [ERROR: <statement>]

The TABLE_EMPTY command allows the user to be sure that a specified table has no entries. The optional ERROR: condition is taken only if the indicated table does, in fact, have an entry. If no ERROR: condition is present, a default error message is given. The TABLE_EMPTY statement may be used in a boolean expression, but without the ERROR: option.

Example:

```
TABLE_EMPTY(A)  ERROR: MESSAGE(G,'W:TABLE A NOT EMPTY')
```

If Table A is not empty, the ERROR: condition is executed and the general class warning message is written.

3.3 INPUT FILE OPERATIONS

The input data file read by the FCF language is assumed to be hierarchic in form. A simple way to think of the input file is to view it as a tree structured file. Figure 2 shows a simplified example of how an input file might be structured and its corresponding tree structure. Tree structures are represented by nodes, with the hierarchy determined by the different levels of the nodes. With respect to input, the top of the tree can be thought of as the beginning of the file. The first line of data appearing in the example is assumed to be the first node, and the second line the second node, both on the same level. However, there are input lines subordinate to this line, and they are represented as lower level nodes in the tree structure. The remaining input is treated similarly, with the number of levels and nodes dependent only upon space limitations.

The input file operations are designed to provide the FCF writer with flexible access to all input data. The INPUT statement reads data in a token-by-token, stream-oriented fashion. Each node in the input tree consists of all elements on a given line including any continuation lines. As shown in figure 2, node (2,2,2) in the hierarchic input structure actually is made up of more than one line in the input file. Since these succeeding lines are continuations of the previous line, they are all assigned to the same node in the tree. The INPUT command is used to read elements on any specific node at the tree. The FIND command is used to find any specified node in the tree, and the NODE_EXISTS command is used to test, without altering the current input node position, for the existence of a particular node in the tree. These, combined with save and restore operations of the current location in the input file, provide FCF users with a flexible and potentially very fast input file processing capability. Below is a syntax description and examples of their use for all input file operations.

SAMPLE INPUT FILE:	NODE IN TREE
(BEGINNING OF FILE)	0
FIRST LINE	1
SECOND LINE	2
SUBORDINATE LINE	2,1
ANOTHER LINE	2,2
DATA VALUE	2,2,1
MORE DATA	2,2,2
CONTINUATION	2,2,2
LAST LINE THIS LEVEL	2,3
NEXT LINE FIRST LEVEL	3
⋮	

CORRESPONDING TREE STRUCTURE:

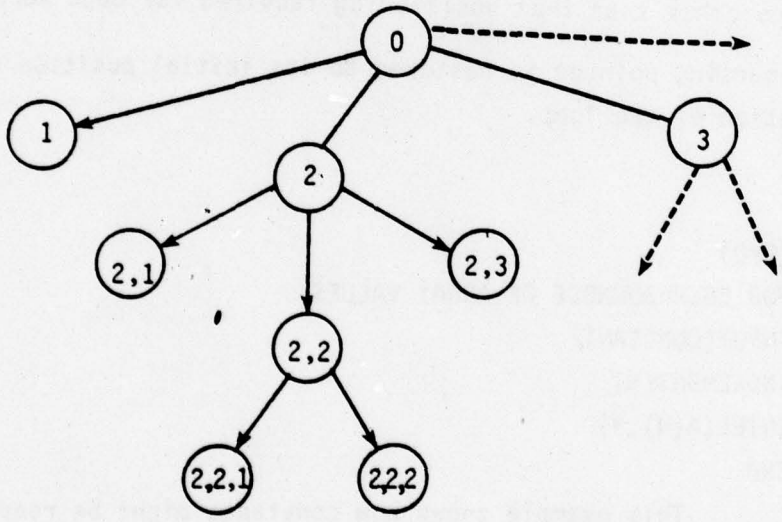


Figure 2. Sample of Hierarchic Input

DO FOR EACH SUBNODE OF

SYNTAX: DO FOR EACH SUBNODE OF <input tree>

```
    :
END
```

or

DO FOR EACH SUBNODE OF @

```
    :
END
```

This DO block allows the user to perform the same operations for each of a group of subnodes on the input tree. If there are no subnodes at this point in the input file, a warning message is written and the DO block is checked only for syntax; no operations are performed. In the second form of this DO construct, the @ implies that no further searching of the input will be done other than that positioning required for each subnode.

The input parsing pointer is restored to its initial position at the completion of the loop.

Example:

```
SET(N=0)
DO FOR EACH SUBNODE OF ARRAY.VALUES
   INPUT(CONSTANT)
   INCREMENT(N)
   ENTER(A(N),*)
   END
```

This example shows how constants might be read into a Table A. The input file is searched for ARRAY.VALUES and all subnodes are entered into symbol Table A. Another way of doing this conditionally might be:

```
IF FIND (ARRAY.VALUES)
   THEN DO
   SET(MAX = 10)
   DEFINE TAB(MAX)
   SET(TABLE_NAME = 'TAB')
   INVOKE TABLE_READ
   END

:
:
TABLE_READ:=
   DEFINE TABLE AS VALUE(TABLE_NAME)
   SET(N=0)
   DO FOR EACH SUBNODE OF @
      INPUT(CONSTANT)    ERROR: MESSAGE(I,'S:INCORRECT DATA IN INPUT FILE')
      INCREMENT(N)
      IF N <= MAX THEN ENTER(TABLE(N),*)
      END
      DEFINE TABLE AS NULL
   END TABLE_READ
```

In this example, a test is made to see if there is a second level node named VALUES subordinate to a first level node named ARRAY in the input file. If so, a table is dimensioned to contain at most 10 elements. The rule to be invoked expects the equivalenced table name to be passed in symbol entry TABLE_NAME in the default symbol table. It also expects to have the maximum allowable subscript passed as symbol MAX in the default table.

The rule can be elsewhere in the FCF, in the input or in a library. The rule assumes that the pointer has already been positioned. Data values are read into TABLE which have been equivalenced to a user supplied table name.

The rule TABLE_READ is invoked to actually input the values. Note that the DO FOR EACH SUBNODE OF loop implicitly saves the input pointer position which is pointed at the next character after VALUES, a subnode of ARRAY. In the first example the pointer position saved was the last position referenced before ARRAY. VALUES was found.

FIND

SYNTAX: FIND(<input tree>) [ERROR: <statement>]

The FIND statement allows the FCF writer to directly access any node in the input hierarchy. If a <word> is the first element in the input tree specification, the search begins at the top of the input file and proceeds to each successive level until the required input node is found. If the tree specification begins with an at sign (@), the search for the required node begins at the current input pointer position. If the search fails at any of the levels, the optional ERROR: condition is processed. The FIND command can be used in boolean expressions. The input pointer position is altered only if the indicated tree was actually found. Otherwise, the input pointer remains at the current position.

Examples:

```
FIND(EARTH.NON_ROTATING)  ERROR: DO...END
FIND(EARTH)
 :
FIND(@.NON_ROTATING)
```

The above examples show both forms of the FIND statement. In the first statement, the input file is searched from the top for a first level node EARTH with a subnode labeled NON_ROTATING. If not found, the ERROR: condition is processed and the DO block executed. In the second statement, the search begins at the current input pointer location, immediately after the node labeled EARTH. The next level down is searched for a node labeled NON_ROTATING.

INPUT

SYNTAX: INPUT(<input element ><...>
[OR <input element> <...>] <...>)
[ERROR: <statement>]

Where<input element>is any one of the following: STRING,WORD, CONSTANT,NUMBER,SIGNED_NUMBER or <string>.

The INPUT statement allows the user to read data from the input file. Reading is sequential within any node. The next term to be input depends upon which node in the tree has been pointed to and how many terms within that node have been previously input. Note that a FIND operation always inputs the first element of the node that it successfully finds. This element is available to the user, the same as all elements read in by the INPUT command, by use of the * notation. (See Section 3.0.) Sequential groups of input elements may be requested in one INPUT statement simply by listing them in the statement separated only by blanks. Any number of these input element requests, whether single or grouped, may be combined in a single INPUT command with use of the optional OR construction. If the input file did not contain any one of the elements requested in the preceding input group, and an OR condition is present in the input statement, the input pointer is repositioned to the place it was when the INPUT statement was initiated and the second group is checked. When the desired input group is found to be present in the input file, the rest of the INPUT statement contents are skipped. If none of the requested input groups are present in the input file at its current location, the input pointer is left at a point following the last term requested that actually was present in the input file. Note also that even though whole groups of input elements can be requested in one input command, only that element that was last read in is available for internal program use with the * notation.

Normal input pointer positioning from node to node is possible and easiest with the FIND statement. However, it is possible to force the input pointer to the next logical node in the input hierarchy with special use of the INPUT command. When the input pointer is positioned at the end of a node, any further normal input requests will fail. The user may force the pointer to the next node, regardless of its level in the hierarchy, by issuing the statement INPUT ('END OF NODE'). The special string END OF NODE will force the pointer to the next node only if the pointer is actually positioned after the last element on a node (including all continuation cards). At any other time this request will fail. Such positioning of the input pointer is very useful in instances where a sequential access is faster. Note that positioning of the pointer in this manner will not automatically input the first element of the next node as happens with the FIND command.

Input requests are either by class of input, e.g., WORD, NUMBER or by specific item, specified in the form of a string literal, e.g., '=' or 'NAME_OF_NODE'. Although most clases of inputs are self-explanatory, the distinction between NUMBER, SIGNED_NUMBER and CONSTANT should be understood. A request for NUMBER will only be successful if the next element is an unsigned numeric quantity, e.g., 7.3, 9, 4.976E+10. Similarly, an input request for a SIGNED_NUMBER will only succeed if the next input element is a signed numeric quantity, e.g., +9.32, -100.76, -7. The use of CONSTANT permits either signed or unsigned numeric quantities to be input. Under most circumstances, the use of CONSTANT is preferred. But in those cases in which signed or unsigned numeric quantities are required for validity, the alternate input classes provide an easy verification check.

Example:

```
      (INPUT FILE)
        :
        :
   REQUIRED_AIRCRAFT_PARAMETERS
      WING_SPAN  126.3 FEET
      CRUISING_ALTITUDE  23. THOUSAND FEET
      CLIMB_RATE  275 FEET/SEC
      MAX_ACCELERATION  3.7 GS
        :
        :
   (FCF)
   P     DEFINE PARAM(M)
   P     SET(N=0)
   P     DO FOR EACH SUBNODE OF REQUIRED_AIRCRAFT_PARAMETERS
   P        INPUT(WORD NUMBER)
   P             ERROR: MESSAGE('IS:ILLEGAL AIRCRAFT PARAMETER')
   P        INCREMENT(N)
   P        ENTER(PARAM(N),*)
   P        END
```

In this example, several aircraft parameters are read in. The input file is assumed to be structured as a form filling exercise, i.e., the order is predetermined so that wing span is always the first item under REQUIRED_AIRCRAFT_ PARAMETERS. The engineer fills in the appropriate number next to the corresponding identifier. The FCF in turn reads each node by reading the WORD and NUMBER in one operation. The NUMBER is what is actually entered into table PARAM since it was the last item input. The descriptors at the right of each node are ignored and serve only to give guidance to the user what units are expected for the values.

```
      (INPUT FILE)
        :
        :
      AREA_FORMULA
         PI*RADIUS**2
        :
        :
      (FCF)
P  IF FIND(AREA_FORMULA) THEN DO
P       INPUT('END OF NODE') ERROR : DO END
P       SET(AREA)
P       INVOKE A_EXPR
P       END
          :
          :
P  A_EXPR:
P       IF INPUT('+' OR '-') THEN DO
P          CATENATE(@,*)
P          INVOKE A_TERM
P          END
P       ELSE INVOKE A_TERM
P       DO WHILE INPUT('+' OR '-' OR '*' OR '/' OR '**')
P          CATENATE(@,*)
P          INVOKE A_TERM
P          END
P       END A_EXPR
P A_TERM:
P       IF INPUT('(') THEN DO
P          CATENATE(@,*)
P          INVOKE A_EXPR
P          INPUT(')') ERROR: MESSAGE(I,'S:MISSING RIGHT PARENTHESES',
P            'IN ARITHMETIC EXPRESSION')
P          CATENATE(@,*)
P          END
```

```
P     ELSE IF INPUT(NUMBER) THEN CATENATE(@,*)
P     ELSE DO
P        INPUT(WORD)  ERROR: MESSAGE(I,'S:ILLEGAL TERM IN',
P           ' ARITHMETIC EXPRESSION')
P        CATENATE(@,*)
P        IF INPUT('( ') THEN DO
P           CATENATE(@,*)
P           INVOKE A_EXPR
P           DO WHILE INPUT(',')
P              CATENATE(@,*)
P              INVOKE A_EXPR
P              END
P           INPUT(')')  ERROR: MESSAGE(I,'S:MISSING RIGHT',
P            ' PARENTHESIS ON ARRAY DECLARATION')
P           CATENATE(@,*)
P           END
P        END
P     END A_TERM
```

This example demonstrates several features of the use of the INPUT statement. The input is searched for existence of an AREA_FORMULA. If one is found, then an attempt to force the input pointer to the beginning of the next node is made. If this fails, i.e., there are additional elements at this node, no action is taken. The FCF assumes that the formula is at the same node. Then the symbol AREA in the default symbol table is located. This is where the formula is to be stored. See section 3.5 for a discussion of FCF rules. The rule A_EXPR is invoked. This rule actually reads the formula an element at a time and stores it at the current symbol table location, i.e., AREA. Note that the INPUT commands used show how the

optional OR construct can be of invaluable aide in reading one of a list of items such as the arithmetic operators. Note also that the INPUT command can be used as a boolean expression as in the IF and DO WHILE statements. Lastly, note that the rules A_EXPR and A_TERM are recursive, i.e., they invoke one another. Such use of rules enables a sophisticated input operation such as reading an arbitrarily complex arithmetic expression a relatively compact operation. This is a sophisticated example and should be studied carefully.

NODE_EXISTS

SYNTAX: NODE_EXISTS(<input tree>) [ERROR: <statement>]

The NODE_EXISTS statement provides the user with a capability to test the input file for occurrences of specific inputs. The difference between this statement and the FIND statement is that the input pointer location is not altered with the use of this statement. Thus, for example, it allows the user to read a specific type of data from the current input file based upon conditions more conveniently placed elsewhere. Or it can be used to insure that the input file is structured properly before the FCF is processed. This statement can be used in boolean expressions to make complex conditional judgments.

Example:

```
P IF NODE_EXISTS(EARTH.GEOPHYSICAL) THEN DO
   CALL HETEST
P CATENATE(PASS2,HECOMMON,',HEIGHT,TEMP,GEOP')
P END
```

In this example, the input file is searched to see if the specified node exists. If it does, a call to the appropriate subroutine is placed in the current FORTRAN output stream. In addition, three variable names are catenated to the end of a value for symbol HECOMMON in the PASS2 table to be used on PASS2 substitution. Later, a subroutine named HETEST might actually be customized using the input data associated with the specified node. This does not disturb the current input processing that could be being used to directly customize the current FORTRAN statements.

```
NODE_EXISTS(REQUIRED.INPUT)  ERROR:  MESSAGE(I,'S:THIS INPUT REQUIRED')
```

In this example, the NODE_EXISTS statement is used to insure that the required data is in the input file. If not, a severe error message is issued. Such statements could be grouped at the beginning of the FCF to insure the input contained all items required for processing before unnecessary processing of the FCF takes place.

RESTORE_POINTER

SYNTAX: RESTORE_POINTER

The RESTORE_POINTER statement resets the input pointer to its most recently saved value. (See SAVE_POINTER.)

Example:

```
DO FOR EACH SUBRODE OF AERODYNAMIC_CALCULATIONS
  IF INPUT (WORD) THEN DO
  SET (RULE_NAME = *)
  SAVE_POINTER
  INVOKE  RULE_NAME
  RESTORE_POINTER
  END
ELSE MESSAGE ('IW: ILLEGAL RULE NAME', RULE_NAME)
END
```

In this example, the particular aerodynamic calculations to be performed are indicated by identifiers that appear as subnodes. Each of the words identifies a rule name. Each rule is in turn invoked in the order it appears. Since the rules that are invoked may contain input parsing operations, the pointer location must be saved and restored each time through the loop to insure proper positioning of the input file pointer.

SAVE_POINTER

SYNTAX: SAVE_POINTER

The SAVE_POINTER command allows the user to save the current value of the input pointer. The save is done as a stack operation.

Example: See RESTORE_POINTER

3.4 CONDITIONAL IF STATEMENT

The use of conditional constructs provides a powerful aid in all programming languages. In addition to the special statements identified as "TEST" in table 3, the FCF language provides this kind of capability with the use of IF...THEN and IF...THEN...ELSE statements. The truth or falsity of a boolean expression is tested and, based upon the result, the following FCF code is processed or simply scanned. Since there are no branch instructions that physically bypass FCF code, all code is checked for proper syntax. Scanning the code simply implies that all precompiler functions not related to syntax checking are temporarily inhibited. The importance of this to the user is that all FCF code whether or not it is executed must be syntactically correct or else a severe error might occur and cause an unnecessary (from the user's viewpoint) stoppage in FORTRAN output.

IF STATEMENT

SYNTAX: IF <boolean expression> THEN <statement> [ELSE <statement>]

The IF statement provides the foundation for conditional inclusion of FORTRAN statements. The boolean expression provides a test which can be evaluated as either true or false. If the boolean expression evaluates to true, the statement (or DO block) following the THEN is processed. If the expression is evaluated as false, then the optional ELSE clause is processed. The next statement in sequence following the IF is processed after the conditional statements have been executed or scanned as appropriate.

Example:

```
IF N>0 OR VALUE(A,SWITCH) = 'ON'
   THEN DO
      :
      :
      END
   ELSE IF...THEN
```

In this example, the value for symbol N in the currently defined default symbol table is tested to see if it is greater than zero. The value of SWITCH in Table A is checked to see if it is equal to ON. If either is true, then, the DO block following the THEN is processed. Otherwise, the IF statement following the ELSE is processed.

3.5 LOOP AND BLOCK STRUCTURES

An essential capability in most languages is the ability to classify groups of statements into blocks. These blocks may simply indicate a group of statements separated by function, or a group of statements on which repetitive actions are required, or a group of statements physically set apart from the main program. In FORTRAN, repetitive blocks are known as DO groups, and physically disjoint blocks as subroutines or functions.

The FCF language has several block constructs. Two of the DO block constructs have already been discussed: the DO FOR SYMBOL TABLE block was discussed with the symbol table operations; the DO FOR EACH SUBNODE OF block was discussed with input file operations. The DO FOR SYMBOL TABLE construct separates a group of statements for a special operation, namely, to facilitate symbol table manipulations. Similarly, the DO FOR EACH SUBNODE OF block construct provides a loop structure based upon what appears in the input file. Two additional block constructs and a loop construct are discussed here.

In addition to the loop and block statements discussed below, groups of FCF statements can be blocked together to form what is known as a rule. A rule is a block construct that can be accessed via the INVOKE statement discussed below. A rule is roughly similar to a subroutine in FORTRAN with the INVOKE similar to a FORTRAN call statement. There are, however, no arguments associated with a new rule or an INVOKE statement. Since all FLTOPS variables are global in scope, all rules have access to all variables.

SIMPLE DO

```
SYNTAX:  DO
           :
         END
```

The simple DO statement provides a convenient way to block functional parts of the FCF. It also provides a way of performing a number of statements when a test has failed as in ERROR: conditions or as options in an IF construct.

Example:

```
IF N>0
  THEN DO
         :
  END
```

In this example, the entire DO block is processed if and only if the value for N in the currently defined default symbol table is greater than zero.

DO WHILE

SYNTAX: DO WHILE <boolean expression>

:

END

The DO WHILE block allows for repetitive processing of a block of statements while a boolean expression evaluates to true. If the expression were false, then the block would not be processed.

Example:

```
SET(I=1)
DO WHILE I<=VALUE(CONSTANT,N)
    INPUT(CONSTANT)
    ENTER(A(I),*)
    INCREMENT(I)
    END
```

In this example, N values are read from input and stored in Table A. The test of I and VALUE(CONSTANT,N) is made before each pass through the loop. Once I becomes larger than the value of N, the loop is terminated. As in all loop constructs, care must be taken to avoid conditions which would preclude loop termination.

INVOKE

SYNTAX: INVOKE <string expression>

The INVOKE statement performs the function of logically inserting an entire FCF rule of the specified name at the point of the INVOKE statement. The named rule is processed and then control transferred back to the calling FCF rule. A rule is a block of FCF code that begins with a rule name, followed by a : . Each rule must be terminated with an end statement bearing the name of the rule. Rule names are normally words.

Example:

```
      IF NEWRULE = 'GSUBR'
        THEN INVOKE GSUBR
             :
      GSUBR:
             :    Legal FCF Language and FORTRAN statements
        END GSUBR
```

In this example, if the value of symbol NEWRULE in the default symbol table is GSUBR, then a rule by that same name is processed. The rule is terminated properly with an END statement bearing the rule name.

3.6 OUTPUT/MESSAGES

The FLTOPS precompiler automatically writes out any statements whose first character is not a P that it encounters while operating with its FUNCTION switch set to FULL (see Section 3.10). However, it may often be desirable for a FCF writer to synthesize FORTRAN statements based upon user inputs or precompile time computations. Such specialized output capability is provided via the OUTPUT statement. Additional output capability is provided to the user through the MESSAGE statement which provides the FCF writer with a generalized capability to write out information to be printed.

OUTPUT

SYNTAX: OUTPUT(⟨output element⟩[,⟨...⟩])

where ⟨output element⟩ can be any one of the following:

⟨value⟩ [⟨format⟩]

or $⟨number⟩ (identifies a label)

or # (statement terminator symbol)

or C: (indicates start of a FORTRAN
COMMENT: comment card)

or COL(⟨arithmetic expression⟩) (indicates which column
COLUMN(⟨arithmetic expression⟩) next output element to start with)

or EOF (indicates an end of file mark to
END_OF_FILE be output)

or EOR (indicates an end of record mark
END_OF_RECORD to be output)

The OUTPUT statement provides FCF writers the capability to synthesize FORTRAN statements based upon specific user inputs. Since no syntax checking is done on FORTRAN statements prior to FORTRAN

compilation, the user should take care that statements so synthesized are syntactically correct. Labels can be generated by use of the special label feature (see above). These labels are uniquely generated numbers produced automatically by the precompiler from a user assigned base value (see LBLNUM in Section 3.9). All output is gathered in a buffer until the user explicitly outputs the statement terminator character (#). At that time, the synthesized FORTRAN is output to the FORTRAN file. Continuation cards and statement sequence numbers are provided as appropriate for all FORTRAN statements.

FCF users are also able to synthesize their own FORTRAN comment cards. By explicitly entering C: or COMMENT: as an output element, the FCF indicates to the precompiler that a comment card is being generated. Either a C or COMMENT is placed in the output buffer beginning in character position 1. Any output elements can follow the comment designator as in normal output operations. Upon receipt of the statement terminator the contents of the output buffer are output as a FORTRAN comment card.

The FCF user also can output end of file and end of record marks as befits his situation. These are output as they occur. If the output buffer is not emply, i.e., previous output element was not a statement terminator, this EOF or EOR mark also acts as the statement terminator.

The FCF user also has at his disposal some degree of format control of the output. As in symbol table operations, any value output can be formated to the user specifications before entry into the output buffer. This is especially handy in the DATA output mode in which numbers of particular accuracy are required. Any FORTRAN formats are legal as per implementation option. In addition the OUTPUT statements also allows the user to specify the column, i.e., character position, in which the next output element is to be placed. If the column specified is already

occupied, the precompiler outputs a warning message, then a statement terminator command is implicitly issued emptying the current output buffer contents. At this point the output buffer is padded with blanks to the desired column. Note, the next element output starts in the column identified by the COL command.

Example:

```
      OUTPUT($1, 'DO ', $2, ' I=1,')
        .                                  No FORTRAN statements should
        .                                  appear here.
        .
      OUTPUT(VALUE(N),#)                   Any FORTRAN appearing here will
        .                                  be included in the DO block
        .
        .
      OUTPUT($2,'CONTINUE',#)
```

In this example, the following synthesized FORTRAN statements are written out. The labels generated do not necessarily correspond to what would be generated in an actual implementation.

```
      99001 DO 99002 I=1,100,2
              .
              .
              .
      99002 CONTINUE
```

Here, it is assumed the value of N is 100,2.

```
OUTPUT(C:, COL(11), RULE-NAME,
       'WAS THE FCF RULE THAT GENERATED THIS CODE', #)
        .
        .
        .
SET(OUTPUT='DATA')
OUTPUT(COL(7), 'DATA', NAME => A6, '/', #)
```

```
SET (CARD_COUNT = 1)
DO WHILE CARD_COUNT<= MAX_CARDS
   SET (N=1)
   OUTPUT (COL(6), CARD COUNT => I1)
   DO WHILE N<= 6
      OUTPUT (DATA_VALUES (N, CARD_COUNT) => F8.3)
      IF N = 6 AND CARD_COUNT = MAX_CARDS THEN
         OUTPUT ('/')
         ELSE OUTPUT (', ')
      END
   OUTPUT (#)
   END
SET (OUTPUT = 'FORTRAN')
```

This example shows several features of the output command. Initially a FORTRAN comment card is output which indicates which rule is outputing the following data. Next the OUTPUT switch is set to DATA and a DATA card is written. In the DATA mode, output is not reformatted to conform to any special card-image format. Special format conventions that may be needed by output generated in this mode are controlled entirely by the user. The NAME of the array for the data is output next. Note that it is constrained to six characters by the format A6. Next the correct number of cards are output containing the continuation indicator in column 6 and six data values in F8.3 format per card. Except for the last element or the last card, a comma follows the value. After the data has been output, the OUTPUT switch is set back to FORTRAN.

MESSAGE

SYNTAX: MESSAGE (<value> [<format>][,<...>])

The message statement allows the FCF writer to output error and informational messages to the run time output file for use by those executing the FCF. The message can be made up of any number of symbol table entries or calculated values, as long as the first three characters of the message conform to the following syntax:

TYPE SEVERITY-LEVEL: Text of Message

The first character of the message must be either an I for input file errors or G for miscellaneous message types. The severity level is a single character that identifies the level of error printed. It can be any one of the following:

Here severity level can be

F - Fatal error -- all FCF processing stops. This level of error is not normally set by FCF builders as it constitutes an immediate stop to all processing. It is, however, issued by the precompiler for various internally inconsistent problems.

S - Severe error -- output of FORTRAN statements ceases, but continues to parse the FCF and check for further errors. For input errors, the input file will continue to be read although FORTRAN output ceases.

W - Warning error -- does not affect processing of the FCF, but some likelihood of it being an unintentional error.

N - A note -- a message indicationg some point of documentary interest to the user and/or FCF builder.

The third character position in the message is ignored. It should be either, a colon (:) or blank for readability, however. Type general messages are output immediately, whereas type input messages are saved and printed out at the appropriate point in the input file.

Any symbol table entry can be output with the MESSAGE command as well as internal computations. Successive elements are simply catenated to the end of the message until the closing parenthesis of the statement is encountered. At that point the appropriate message action is taken. Note that as in the OUTPUT statement, the elements output are under user specified format control.

Example:

```
MESSAGE ('GS:  THIS IS ONE WAY MESSAGES ARE WRITTEN')
```

A severe general type message is written to the output listing. The severe error is printed immediately and inhibits further processing of the FCF that is not purely syntactic.

```
MESSAGE ('IW:  A WARNING MESSAGE FOR AN INPUT ERROR')
```

An input warning type message is written. This message will appear in the input file listing at the position it occured.

```
MESSAGE ('GN:  THE VALUE OF A, B, C = ',A = >F8.3,
    B = >F9.4, C=>I5)
```

The message statement outputs the values of various symbol table entries in the specified formats. Since this is a note level message, it is not written unless the NOTES switch is set to ON.

3.7 COMPUTING PRECOMPILE TIME VALUES

To allow the FCF builder to incorporate values that are determinable at precompile time, two statements are provided. Any parameters that depend upon aircraft test data are obviously not determinable at precompile time. However, this capability is useful for determining parameters derivable from aircraft constants, standard atmosphere, or geopotential model parameters, etc. Two statements are used. The PARAMETERS statement identifies the type and quantity of data to be supplied to the execute module. The EXECUTE statement actually transfers the control to the desired routine and passes/returns the correct data.

Data Considerations

To compute precompile time values requires some data considerations. Data that are essentially strings must be converted to equivalent internal values to be used by the executing routine, and they must be stored in a common area that can be accessed both by the precompiler and the routine doing the computations. To provide the proper data communications, two separate requirements are identified: an execute-environment-table and an execute-data-area.

The execute-environment-table contains information necessary for the proper conversion and passing of data to the executing routine. This table contains the following information for each parameter value(s) that is to be passed to the executing program.

1) What type of data -- i.e., REAL, INTEGER, ALPHA, LOGICAL, DOUBLE.
2) How much data -- i.e., how many data values of the identified type for this parameter are required in the executing program.
3) Where is the data found -- i.e., a precompiler unique internal data address where the requested data can be found or stored.

A different environment table is maintained for each routine that can be used to compute precompile time values. A simple, but not necessary, way of thinking of this table is as a matrix in which the i^{th} row contains the above three items of information for the i^{th} parameter data address.

The execute-data-area contains the data that will be processed and/or returned by the executing program. It is in internally-usable format and is saved in the execute-data-area sequentially in the same order that the parameters appeared when the routine was called. Figure 6 shows the data communication between the precompiler and an executing routine.

User interface with these tables is provided through two statements whose form and syntax is provided below.

PARAMETERS

SYNTAX: PARAMETERS <execute routine> (<parameter> [,<parameter> <...>])

The PARAMETERS statement identifies the type and amount of data required by the routine to be executed. The <execute routine> is any legal <word>. Here a valid <parameter> is anyone of the following:

REAL[<subscripts>]
INTEGER[<subscripts>]
LOGICAL[<subscripts>]
DOUBLE [<subscripts>]
ALPHA [<subscripts>]

Only the above five descriptors followed by optional subscripts are valid in the parameter list. As many parameters must be defined here as there are arguments in the calling sequence of the routine to be executed.

Example: See EXECUTE

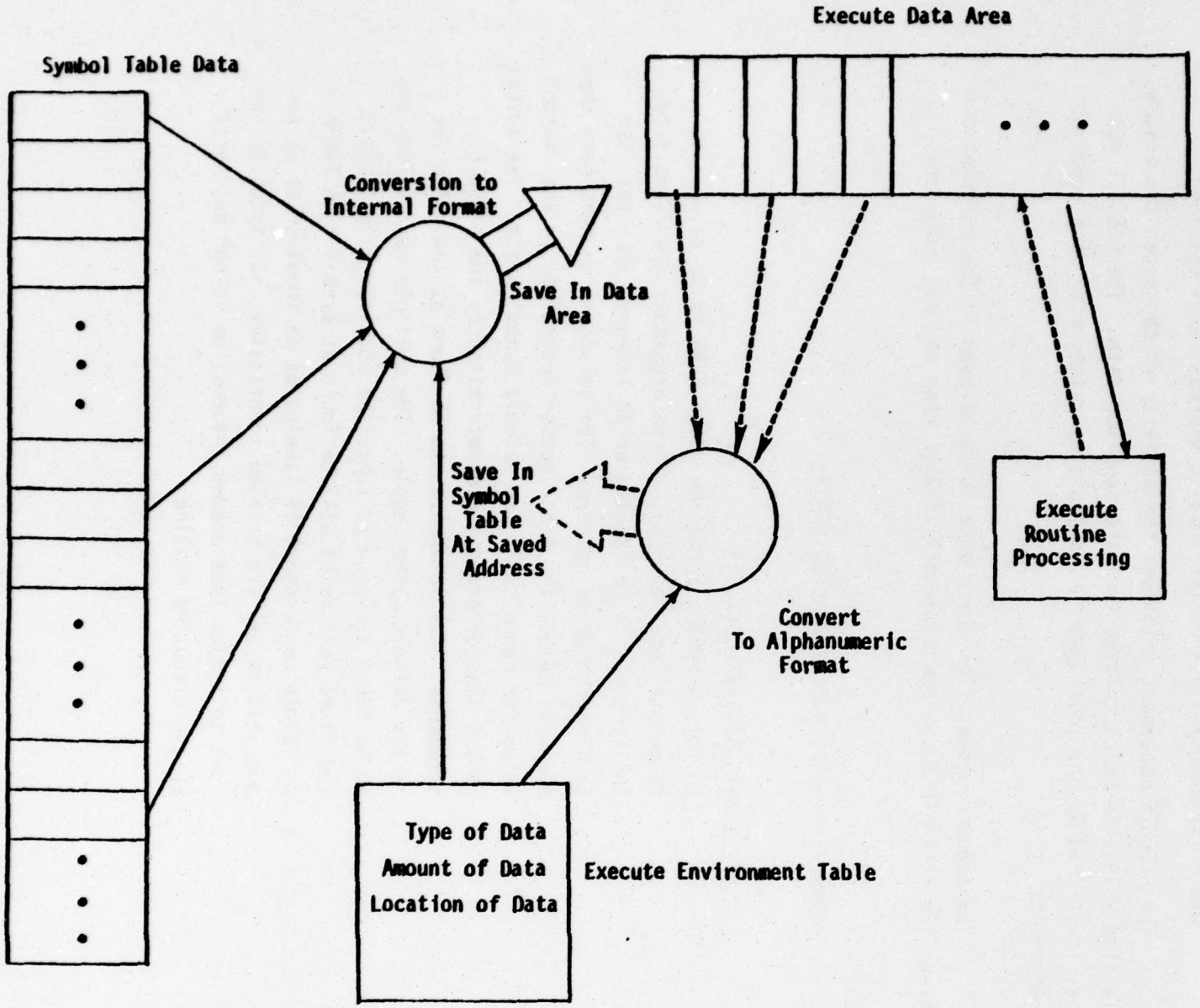


Figure 6. Data Communications Between the Precompiler and the Execute Routine

EXECUTE

SYNTAX: EXECUTE <execute routine> (<argument> [, <argument> <...>])
where <argument> is either <arithmetic expression> or <string>

The EXECUTE statement initiates the activity which causes the precompile time values to be calculated from the supplied data. The EXECUTE routine, identified by any legal <word>, calculates values with data supplied as arguments.

The argument identifies which data is to be used in the calculations whereas the PARAMETERS statement identifies what kind and how much data.

Example:

```
PARAMETERS SUBR(INTEGER,REAL(N),REAL)
   :
EXECUTE SUBR(N,A(1),VALUE(X,Y))
```

In this example, a routine named SUBR is to be used to calculate some value(s). Three arguments are identified. The first is to be treated as an integer, the other two as floating point numbers. The second is to have more than one real value, the exact number determined by the current value for symbol N in the default symbol table. The actual values passed/returned are determined by the EXECUTE arguments. The integer is to be found as the value for N in the default symbol table. The multiple real values are to be found starting at subscript position 1 in Table A. The final real value will be found at symbol Y in Table X. Since no argument is labeled as an input or as an output, all values are treated identically. It is up to the user to insure that needed information is not destroyed by the executing routine.

3.8 BUILT-IN FUNCTIONS

The FLTOPS precompiler contains several built-in functions usable by the FCF writer to provide enhanced flexibility in symbol table access and string manipulation. These built-in functions are not usable by themselves but only in combination with any legal FCF statement. The INDEX, LENGTH, and SUBSTRING functions provide the FCF users with the same specialized string manipulation features as are available in other programming languages. The VALUE function is unique to FLTOPS and enables the user to directly reference any symbol table entry.

INDEX FUNCTION

SYNTAX: INDEX (<string expression>, <string expression>)

The INDEX function returns the starting location of the substring identified by the second <string expression> within the string identified by the first <string expression>. If the second string is not a substring of the first string a value of 0 (zero) is returned.

Example:

```
IF INDEX ('**+-/',*)>0 THEN
   DO
     .
     .
     .
   END
```

In this example the last element read from input is checked to see if it is one of the arithmetic operators. If it is, then some additional FCF statements will be processed.

```
SET (N = INDEX (ALPHABET, SUBSTRING (*, 1, 1) ) )
```

In this example the symbol N in the default symbol table is set to the number of the letter of the alphabet (stored in symbol location ALPHABET) with which the first character of the last element input begins. If it does not start with an alphabetic character N will be set to zero.

LENGTH FUNCTION

SYNTAX: LENGTH (<string expression>)

The LENGTH function returns the integer number of characters that comprise the indicated <string expression>. If the string is null (i.e. empty) the value returned is zero.

Example:

```
SET (N = LENGTH ('ABCDE') )
```
The value of N is set to 5.

```
SET (N = LENGTH (*) )
```
The value of N is set to the number of characters contained in the element last read from the input file.

```
IF LENGTH (ALPHABET) <26 THEN MESSAGE ('GW:  NOT ALL LETTERS PROVIDED
    IN THE ALPHABET TABLE ENTRY')
```

In this example, the symbol ALPHABET in the default symbol table is checked to see if it contains enough characters for the whole alphabet.

SUBSTRING FUNCTION

SYNTAX: SUBSTRING (⟨string expression⟩, ⟨start position⟩[,⟨length⟩])

The SUBSTRING function enables users to obtain a substring of a given ⟨string expression⟩. The substring is created by starting at the character position identified by the parameter ⟨start position⟩, which can be any ⟨arithmetic expression⟩. A maximum of ⟨length⟩ characters starting with ⟨start position⟩ comprise the substring. If less than ⟨length⟩ characters are actually available, the substring is padded with blanks to achieve the desired length. The parameter ⟨length⟩ can be any ⟨arithmetic expression⟩. If the ⟨length⟩ is not given, the substring contains all characters including and to the right of position ⟨start position⟩ in ⟨string expression⟩. If ⟨length⟩ is zero, the substring is null. The SUBSTRING function is not legal by itself as an FCF control statement. It is only legal within the context of other FCF language statements.

Example:

```
SET (NAME = SUBSTRING (*, 1, 6) )
ENTER (A, B, SUBSTRING (VALUE (A, TEMP), 7,10))
SET (OPERATOR = SUBSTRING ('+-/**', N, VALUE (A, LENGTH) ) )
SET (OPERATOR = SUBSTRING ('+-/**', 4) )
```

In these examples the SUBSTRING function is being used to enter portions of a predefined string into a symbol table. In the first example, the last data read from input has its first six characters being stored as the value for symbol NAME in the default symbol table. In the second, a 10 character substring starting at character position 7 is being extracted from symbol entry TEMP in table A. This substring will be stored as symbol

T in Table A. In the third example, a symbol named OPERATOR will have its value be determined by a substring created from the given string literal. The substring will start in the position indicated by the value of symbol N and be of length determined by the value of symbol LENGTH. In the last example, OPERATOR is set to the substring **, which includes all characters to the right of and including position 4 in the original string.

VALUE FUNCTION

SYNTAX: VALUE (<table name>, <symbol name>)
or VALUE (<table name> <subscripts>)
or VALUE (<symbol name>)
or VALUE (@)

The VALUE FUNCTION retrieves symbol table entries and makes them available to the user. The VALUE function is not legal by itself as an FCF language instruction. It is only legal within the context of other FCF language statements. The four forms retrieve values from the specified locations. The first two retrieve the value associated with the indicated table and symbol name or subscript respectively. The third retrieves the value associated with the symbol named in the currently defined default symbol table. The last retrieves the value associated with the current location of the symbol table pointer.

Example:

```
IF VALUE(A,I)>0 THEN . . .
SET(Z=VALUE(X(VALUE(A,I))))
ENTER(A,B,VALUE(N))
```

This example shows how the first three forms of the value function

might appear. Note that in the second statement, VALUE functions can be nested. This can be a valuable aid in actual practice.

```
ENTER (A,B,'N')
  .
  .
  .
SET (VALUE (@) = 7)
```

In this use of the VALUE function, the value is retrieved from the most recent placement of the symbol table pointer. Assuming no intermediate placements of the symbol table pointer, the symbol N is retrieved from table A, symbol B. N is assumed to be a symbol name in the current default symbol task and is assigned the value of 7.

3.9 DEBUGGING AIDS

Two debugging aids are provided in the form of statements to be used by the FCF writer. The ASSERT and STATUS commands are executed only if the DEBUG switch (see Section 3.10) is set to ON. Otherwise, they are ignored. Initially, the DEBUG switch is set to OFF. The user also has access to a TRACE feature described in detail in Section 3.10.

ASSERT

SYNTAX: ASSERT <boolean expression> [ERROR: <statement>]

The ASSERT statement allows the user to test various conditions at different stages of FCF processing. In this way, he can monitor progress and perhaps determine where or why certain run time errors or undesired output are happening. If the boolean expression is false, the optional ERROR: condition is processed. If not present, a default error message is written by the precompiler. No action is taken if the assertion proves to be true.

Example:

```
ASSERT VALUE(A,X)='FALSE' AND SWITCH='ON'
       ERROR:  DO
                 :
               END
```

In this example, the various values are checked to see if the relation holds. If either one is not true, the optional DO...END block is processed.

STATUS

SYNTAX: STATUS(<string>)
or STATUS(<string>, <table name>)
or STATUS(<string>, <table name>, <symbol name>)
or STATUS(<string>, <table name> <subscripts>)

The STATUS statement writes out a comment string and the contents of various tables/symbols depending on the form of the statement. If just a comment string appears in the argument list, the comment string and a complete status of the precompiler is written which includes the contents of all symbol tables. If the second form above is used, then the entire contents of the table name specified are written out along with the comment string. In the last two forms above, the individual symbol table value is written out in addition to the comment string. In all cases, table names, symbol names or subscripts are appropriately labeled on output for easy comprehension.

Example:

```
STATUS('DUMP OF ALL TABLES')
STATUS('TABLE A CONTENTS', A)
STATUS('SYMBOL X IN TABLE A', A,X)
STATUS('VALUE IN TABLE TAB', TAB(3))
```

The above examples demonstrate each form of the STATUS statement. The TRACE feature operates separately from the above STATUS statements.

3.10 SWITCHES AND COUNTERS

An important functional part of the FCF control language is the existence of a number of switches and counters that are made available to the user. By setting these switches and counters, the FCF writer can control basic functions of the precompiler without special instructions or language forms. Each switch/counter is identified as a symbol entry in the FLTOPS symbol table. Since they are normal symbol table entries, they can be accessed using any of the symbol table operations discussed in Section 3.2 Note, however, use of the DELETE or CLEAR_TABLE statements will not affect any of the switches and counters defined here. Table 3 presents a simplified summary of the switches/counters discussed below.

ALPHA/INTEGER/LOGICAL/REAL

Each of the above switches controls data conversion for those values used in an EXECUTE route. If the switch is left at the default value of null, the precompiler will use predefined rules established at implementation time to convert the values for each type. The exact nature of these rules is left to the implementor. If the switches are set to other than null, then the corresponding data types will be converted accordingly to the exact specification given. The setting of these switches to null will actually result in the respective default value being inserted in the symbol table.

Example:

```
PARAMETERS SUBR(REAL(5),INTEGER,REAL)
   :
   :
SET(REAL='F10.4')
EXECUTE SUBR(A(1),5,VALUE(Y,X))
```

In this example, upon completion of the execute, the values identified as REAL will be stored in the symbol table locations as strings using an F10.4 conversion format. The integer will be converted using a predefined precompiler default procedure.

TABLE 4. SWITCHES/COUNTERS SUMMARY

FLTOPS SYMBOL	VALUES	SIGNIFICANCE
ALPHA	null, or legal FORTRAN A format	Controls conversion of string data used with the EXECUTE statement Default - implementation option
DEBUG	ON, OFF	Controls execution of STATUS and ASSERT statements Default - off
FUNCTION	FULL, TEST, SYNTAX	Controls precompiler operation including generation of FORTRAN output Default - FULL
INTEGER	null, or legal FORTRAN I format	Controls conversion of integer values used with the EXECUTE statement Default - implementation option
LBLNUM	any number 0 - 99999	Number of first label to be used for automatically generated labels Default - 99000
LENGTH	OFF or any number	Counter contains estimate of storage requirements of FORTRAN output Default - OFF
LOGICAL	null, or any legal FORTRAN L format	Controls conversion of logical values used with the execute statement Default - implementation option
NOTES	ON, OFF	Controls printing of messages at "NOTES" level of severity Default - OFF
OUTPUT	FORTRAN or DATA	Controls the kind of format used when processing PASS 1 and PASS 2 output files. Default - FORTRAN
REAL	null or any legal FORTRAN F, G, E, or D format	Controls conversion of floating point numbers used with the EXECUTE statement Default - implementation option
STATS	ON, OFF	Controls collection and printing of statistical information on run. Default - OFF
TRACE	OFF, LOW, HIGH	Controls generation of TRACE debugging information on run Default - OFF
XREF	ON, OFF	Controls generation of cross reference map of symbol table usage. Default - OFF

DEBUG

The DEBUG flag allows users to selectively process ASSERT and STATUS statements. It may be turned on or off at any time. As a result, users may leave these debugging aids in the FCF at all times and selectively or collectively execute them.

Example:

```
ENTER(FLTOPS,DEBUG,'ON')
   :
ASSERT  VALUE (A,B) = 'OFF'
  ERROR:  DO
     MESSAGE ('GW:  SWITCH B, TABLE A FORCED TO OFF POSITION')
     ENTER (A,B, 'OFF')
     END

  IF VALUE (A,B) = 'OFF' THEN ENTER (FLTOPS, DEBUG, 'OFF')


  STATUS ( 'FLTOPS SYMBOL TABLE', FLTOPS)
```

In this example the ASSERT command will be executed since the DEBUG switch is on. The STATUS command however will not be executed since the DEBUG switch was turned off.

FUNCTION

Access to the FUNCTION switch provides the FCF user with control over basic precompiler operation. By setting the switch to one of three user assignable values, the user can control the processing of the FCF. Each setting is described in more detail below.

FULL: The default setting of this switch indicates that the precompiler's functions are all operational. The FCF is processed, the input file can be read, and FORTRAN output generated.

TEST: By setting the switch to this value, the user inhibits all FORTRAN output. Otherwise, normal FCF processing occurs and the input file can be read.

SYNTAX: By setting the switch to this value, the user inhibits all FCF processing except basic syntax checking. As a result, once set to this value, the FUNCTION switch cannot be reset to another value.

Example:

```
SET(FUNCTION='TEST')
```

In this example, no FORTRAN output will be generated from this statement on. The FCF will be executed and input read, however.

LBLNUM

The precompiler will generate labels for synthesized FORTRAN statements using the number appearing in LBLNUM as its base. LBLNUM can be reset anywhere without effect on already generated labels.

Example:

```
SET(LBLNUM=1000)
OUTPUT('GO TO ',$1)
   :
   :
SET(LBLNUM=2000)
   :
   :
OUTPUT($1,'CONTINUE')
```

In this example, 1000 is already associated by the precompiler with $1, so that resetting LBLNUM has no effect on already assigned labels.

LENGTH

LENGTH is a counter that can be used to estimate the amount of core that will be used by the FORTRAN code being output. If set to a number, the FORTRAN lexical analyzer will increment the counter by a fixed amount for each different token it processes. When reset to its default value of OFF, the lexical analyzer no longer increments the counter. The user can increment the counter himself to account for commons and dimensioned values not counted by the lexical analyzer. The actual amount that each token will increment the counter is dependent upon the particular machine on which the code is to be run and can be at best a good guess.

Example:

```
P       SET(LENGTH=LENGTH+VALUE(COMMON_LENGTH))
           :
           :
P       IF LENGTH>45000 THEN DO
        CALL OVLY(4HSUBR,1,1)
P       SET(LENGTH=0)
P       ENTER(OVERLAY,SUBR11,'TRUE')
P       END
```

In this example, the LENGTH counter is assumed to have been turned on in an earlier step. The first SET instruction increments the counter by the value of COMMON_LENGTH, which contains the total length of associated commons. LENGTH is later tested to determine if it exceeds a core length requirement. If it does, then the overlay card is inserted, the length reset to zero, and a flag set in the table OVERLAY to indicate which overlay had been called for possible later use.

NOTES, STATS, TRACE, XREF

Each of the above on/off switches controls potential printed output. NOTES, if turned on, allows all note level messages to be issued. NOTES are usually of only passing or documentary interest to the user and so are not always needed.

STATS controls the collection and printing of statistical information. The total extent of what constitutes statistical information is left to the implementor, but should at least be a table of rule invocation frequencies.

TRACE allows the user to optionally follow the execution of the FCF at two levels. If set from its default value of OFF to either LOW or HIGH, the FCF user can trace the flow of FCF processing. At LOW, the trace feature will output FCF rule entries and exits, input parsing progress, and error messages. When set to HIGH, all of the above is indicated along with an indication of which statements in the FCF were executed and symbol table manipulation messages.

XREF specifies that a cross reference listing is to be generated indicating all symbol table entries and where they are referenced. Since the entire FCF must be processed (but not necessarily executed) to obtain the listing, setting the FUNCTION switch to SYNTAX in conjunction with setting XREF on insures that a complete cross reference listing will be made. Otherwise, there exists the possibility that a fatal error will inhibit precompiler functions before the cross reference is complete.

OUTPUT

The OUTPUT switch controls which type of format the customized pre-compiler output will follow. If set to FORTRAN, the initialized value, the output will be processed so as to conform to FORTRAN card image format. Continuation cards will be generated automatically as needed and sequence numbers supplied. If set to DATA, no special card image format is produced, output is transferred directly without checking column alignments or adding sequence numbers.

AD-A068 394 SCIENCE APPLICATIONS INC-ENGLEWOOD CO F/G 9/2
FLIGHT TEST ORIENTED PRECOMPILER SYSTEM (FLTOPS DESIGN SPECIFIC--ETC(U)
AUG 78 D A OTEY, H R RAMSEY, J K WILLOUGHBY F04611-77-C-0040
UNCLASSIFIED SAI-78-061-DEN AFFTC-TR-78-22 NL

2 OF 4
AD A068394

4.0 ILLUSTRATIVE EXAMPLE

To provide a clearer concept of how the precompiler will operate, an example is provided here. Although not all features nor control statements are utilized in this example, it contains sufficient detail to demonstrate most of the salient features of the precompiler design. The example itself consists primarily of FCF control statements, with only enough imbedded FORTRAN actually appearing in the listings to provide an adequate frame of reference. In actual practice, the ratio of FORTRAN statements to FCF control statements will most likely be quite high.

4.1 DESCRIPTION OF EXAMPLE

This example shows how a typical FCF and its associated input file might appear. The portion of the FCF shown is customizing a section of a main program that may or may not include a call to a specific subroutine. The subroutine in question is called HETEST and has as part of its function the calculation of energy height parameters. The subroutine has no arguments, so that the data it calculates is shared with the main program through a common block called HECOM. If the subroutine is called, the common block should be included as part of the main program. To complicate matters, the HETEST subroutine can calculate energy height in several ways. Only the code necessary, based upon the engineer's input requests, is to be generated for input to the compiler.

Additionally, under certain circumstances, some specific data values are needed in the HETEST subroutine to make modifications to the energy height parameters. These data can be calculated at precompile time by anyone of a number of different procedures. Which procedure depends upon the type of results being produced. The data are to appear as a data statement within HETEST after it has been calculated. The engineer on this project has developed a new procedure to calculate these data and wants to incorporate his procedure into the FCF to show that the resulting code provides an improvement over conventional methods. His procedure involves reading in a table of values derived from some existing source as input to an interpolation routine. Further project specific data is read in as input

to the same interpolation routine to evaluate the necessary data velues. The engineer's interpolation routine already exists as a FORTRAN code called POLY.

The following FCF code shows one of many possible ways to accomplish the above. In this procedure there are four FCF rules being used. The MAIN rule which controls all processing invokes a rule named HETEST if the energy height parameters are needed by the main program. The user indicates this by the appearance of ENERGY_HEIGHT as a subnode of REQUIRED_COMPUTATIONS. The HETEST rule is located after the main rule in the FCF. HETEST in turn invokes a specified rule if there is to be a modification of the energy height values. The user indicates this by the appearance of a first level node in input beginning with the expression INTERPOLATION_ROUTIN. The specific rule name is read from input and in this example is named POLY. Since POLY is a special rule designed to accomplish the engineer's specific objectives, it appears in the special rule section of the input file. If there is a subnode of INTERPOLATION_ROUTINE labeled TABLE_INFORMATION, POLY reads the necessary table from input by invoking a rule named TABLE_READ. Because of the general purpose nature of this last rule, it is assumed to have been previously written and tested and to be found in an FCF user's library. For convenience, its listing is provided at the end of the FCF file.

In addition to invoking several rules, this example shows several other useful techniques. The use of the CONSTANTS table in the main rule illustrates how both default values for constants can be incorporated into the FCF and how values input can be checked to insure only legal constants are specified. Specifically, the use of the SYMBOL_EXISTS statement is used to verify that the constant name input matches an entry in the table CONSTANTS. In this instance, only RE and HAV are shown since they are used elsewhere in the example.

Notice that NODE_EXISTS statements appear in many locations throughout the FCF. These tests prove invaluable in structuring FORTRAN code which, based upon the same specified input, might require FORTRAN state-

ments to appear in several different locations. Recall that the NODE_EXISTS statement does not alter the current location of the input parsing pointer as do FIND and INPUT, but simply tests to see if the specified input appears in the input file.

In addition to standard symbol table constructs, this example also shows the use of the special subscript notation for table references by use of the DEFINE statement. To read in groups of values, such as in a table, this proves much more convenient. Additionally, when passing array data to an executable FORTRAN code, this form of specifying tables is much more readable. See the TABLE_READ rule listing for examples of its use.

Also, the use of the DEFINE statement to establish a table name equivalence is shown. The TABLE_READ rule stores information in a table called TABLE; the rule named POLY uses the TABLE_READ rule but refers to data as if it were stored in a table named POLY_TAB. By use of the DEFINE...AS statement, these two names can be made to refer to the same table, so that the data read by TABLE_READ is actually stored in POLY_TAB. After TABLE_READ is finished, the equivalence is removed by use of the DEFINE...AS NULL statement. All table data input is still accessable by the table name POLY_TAB. This use of the DEFINE...AS statement provides a powerful capability for using rules in a subroutine-like context.

The computation of precompile time values can also be very important. This example shows two possible ways of accomplishing this. With use of the PARAMETERS and EXECUTE statements, the user can compute very complicated function values, or whatever, by use of already developed FORTRAN packages. In rule POLY, a FORTRAN routine named POLY is executed to compute the data values for inclusion in a data statement. Another procedure, useful for simpler tasks, is to use an arithmetic expression directly when entering values into a symbol table. An example of this is provided in rule HETEST when the value of GR is entered into the PASS1 table.

Since customized FORTRAN output is a principal aim of the precompiler, numerous instances of both synthesized and direct output of FORTRAN are given. Notice also the use of the PASS1 and PASS2 tables in the modifica-

tion of the FORTRAN code. Both the MAIN rule and HETEST contain examples of the use of these special tables.

The following sections provide listings for the input file and the various rules used from the FCF. Section 4.4 describes the possible outputs generated by the precompiler. Notice, in particular, the precompiler run using the trace feature set to high. By following the FCF code that is highlighted with bold print and the annotations at the far right of the listings, it is possible to see how the FCF is processed.

4.2 THE INPUT SPECIFICATION

```
/*              SAMPLE INPUT FILE                */


CONSTANTS
   HAV = 23000.
GEOPHYSICAL_MODEL
   EARTH
      SPHERICAL
      NON_ROTATING
   GRAVITY
      VARIABLE
REQUIRED_COMPUTATIONS
   ENERGY_HEIGHT
INTERPOLATION_ROUTINE = POLY
   TABLE_INFORMATION
      /* DEFAULT VALUE FOR COLUMNS IS 2 */
      ROWS = 5
      DATA = 0.   26.3
             5.   69.8
            11.   76.3
            17.5  89.1
            20.   99.7
   INTERPOLATION_VALUES = 1.63,2.1,6.2,11.6,19.3
     /* VALUES USED AS INPUT TO INTERPOLATION ROUTINE */
      .
      .
      .
P     /* SPECIAL RULE SECTION */
P     POLY:
P  DO FOR SYMBOL TABLE POLY
P        /* SPECIAL RULE THAT CREATES DATA STATEMENTS   */
P        /* BASED ON VALUES DETERMINED BY INTERPOLATION */
P     IF NODE_EXISTS(@.TABLE_INFORMATION)
P        THEN DO
P        DEFINE TABLE AS POLY_TAB
P        INVOKE TABLE_READ
P        DEFINE TABLE AS NULL
P        END
P          /* TABLE_READ IS AN EXAMPLE OF A RULE THAT MIGHT BE FOUND */
P          /* IN AN FCF USERS LIBRARY--SEE BOTTOM OF RUN             */
P          /* READS DATA INTO  TABLE NAMED TABLE2 AND SIZE INFO INTO */
P          /* SYMBOLS COLUMNS AND ROWS IN CURRENTLY DEFINED DEFAULT  */
P          /* SYMBOL TABLE */
P        ELSE .
                .
                .
P     DEFINE X(10),Y(10)  /* DEFINE TWO TABLES OF FIXED (10) LENGTHS */
P     FIND(INTERPOLATION_ROUTINE.INTERPOLATION_VALUES)
P          ERROR: DO .
P                    .
P                    .
P                 END
```

```
                        /* SAMPLE INPUT FILE CONTINUED */

P       SET(N=0)
P       DO WHILE INPUT(',' OR '=')
P          INCREMENT(N)
P          IF N > 10 THEN MESSAGE(I,'S:TOO MANY VALUES READ IN')
P          INPUT(NUMBER)   ERROR: MESSAGE(I,'S:NUMBERS ARE THE ONLY VALID INPUTS')
P          ENTER(X(N),*)
P          END
P       IF N > 0
P          THEN DO
P             PARAMETERS POLY(REAL(COLUMNS,ROWS),INTEGER,INTEGER,REAL(N),REAL(N))
P             EXECUTE POLY(POLY_TAB(1),COLUMNS,ROWS,X(1),Y(1))
P            /* COMPUTE Y VALUES FROM GIVEN X'S USING DATA FOUND IN POLY_TAB */
P             SET(DATA_ST = 'DATA VALUES/')   /* START BUILDING DATA STATEMENT */
P             SET(I=0)
P             DO WHILE I<N
P                INCREMENT(I)
P                CATENATE(DEFAULT,DATA_ST,VALUE(Y(I))  FORMAT(F4.1))
P                IF I<>N THEN CATENATE(DEFAULT,DATA_ST,',')
P                   ELSE CATENATE(DEFAULT,DATA_ST,'/')
P                END
P             OUTPUT('DIMENSION VALUES(',N => I ,')',#)
P                /* INCLUDE PROPERLY DIMENSIONED VARIABLE NAME */
P             OUTPUT(DATA_ST,#)   /* OUTPUT FORTRAN DATA STATEMENT */
P             END
P          ELSE DO
                  .
                  .
                  .
P                END
P          /* NOW ELIMINATE UNNECESSARY TABLE STORAGE */
P       END   /* END OF DEFAULT SYMBOL TABLE BLOCK */
P       CLEAR_TABLE(X)
P       CLEAR_TABLE(Y)
P       CLEAR_TABLE(POLY_TAB)
P       CLEAR_TABLE(POLY)
P       END POLY

P          /*   END OF INPUT FILE   */
```

```
P   /****          SAMPLE FCF FILE        ****/

P  MAIN:
P     SET(TRACE = 'HIGH')  /* USE TRACE FEATURE TO MONITOR FCF PROCESSING */
P     DO FOR SYMBOL TABLE  CONSTANTS     /* SET UP TABLE FOR LEGAL VARIABLE NAMES */
P        SET(HAV = 20000.)
P        SET(RE = 209264258.)
           .
           .
           .
P        END
P     DO FOR EACH SUBNODE OF CONSTANTS   /* READ IN CONSTANTS FROM INPUT */
P        INPUT(WORD)      ERROR:MESSAGE('IS:ILLEGAL CONSTANT NAME')
P        SYMBOL_EXISTS(CONSTANTS,*)    /* CHECK TO SEE IF THIS IS A LEGAL CONSTANT */
P             ERROR:MESSAGE(I,'S:CONSTANT NAME NOT RECOGNIZED')
P        ENTER(CONSTANTS,*)    /* ENTER SYMBOL FROM INPUT INTO CONSTANTS TABLE */
P        INPUT('=')       ERROR:MESSAGE('IW:MISSING EQUAL SIGN'))
P        INPUT(CONSTANT)    ERROR:MESSAGE('IS:ILLEGAL CONSTANT VALUE');
P        ENTER(a,*)
P        END
           .
           .
           .
      PROGRAM SAMPLE
P     IF NODE_EXISTS(REQUIRED_COMPUTATIONS.ENERGY_HEIGHT)
P        THEN OUTPUT(#HECOM,#)
P        /* HECOM IS A PLACE HOLDER FOR A COMMON BLOCK.  IT WILL     */
P        /* BE REPLACED BY THE PROPER FORTRAN ON PASS2 SUBSTITUTION */
           .
           .
           .
P     IF NODE_EXISTS(REQUIRED_COMPUTATIONS.ENERGY_HEIGHT) THEN
P        /* INCLUDE A CALL TO THE PROPER FORTRAN SUBROUTINE */
      CALL HETEST
           .
           .
           .
      CALL EXIT
      END
P        /* END OF THE MAIN PROGRAM BUT NOT NECESSARILY OF THE MAIN FCF RULE */
           .
           .
           .
P     IF NODE_EXISTS(REQUIRED_COMPUTATIONS.ENERGY_HEIGHT)
P        THEN INVOKE HETEST   /* TRANSFER OF CONTROL TO SPECIAL FCF RULE */
           .
           .
           .
P     END MAIN   /* END OF MAIN FCF RULE */
           .
           .
           .
```

```
P   /****          SAMPLE FCF FILE CONTINUED          ****/

P         /* METEST IS ANOTHER RULE WITHIN THE CURRENT FCF */
P   METEST:
       SUBROUTINE METEST
      #MECOM     /* PLACE HOLDER FOR PASS 2 INSERTION OF COMMON BLOCK */
            .
            .
            .

P           /* PERFORM COMPUTATIONS TO INSERT DATA STATEMENT */
P       IF FIND(INTERPOLATION_ROUTINE)
P          THEN DO
P             INPUT('=')  ERROR:MESSAGE(I,'W:EQUALS SIGN MISSING')
P             INPUT(WORD)    /* READ RULE NAME FROM INPUT */
P             INVOKE *    /* TRANSFER CONTROL TO SPECIAL RULE -- FOUND IN INPUT */
P             END
             .
             .
             .
P       ENTER(PASS1,RE,VALUE(CONSTANTS,RE))
P       ENTER(PASS1,GR,GRAV_CONST*MASS1*MASS2/VALUE(PASS1,RE)**2)
P         /* ENTER COMPUTED VALUE FOR GR INTO PASS1 TABLE */
P         /* GRAV_CONST,MASS1,MASS2 ARE ASSUMED TO BE VALID SYMBOL TABLE ENTRIES */
P       ENTER(PASS2,#MECOM,'COMMON/MECOM/ HT(20), MET(20)')  /* UPDATE MECOM */
            .
            .
            .
        DO 900 J=1,20
P       IF NODE_EXISTS(GEOPHYSICAL_MODEL.EARTH.SPHERICAL) AND
P         NODE_EXISTS(GEOPHYSICAL_MODEL.EARTH.NON-ROTATING) AND
P         NODE_EXISTS(GEOPHYSICAL_MODEL.GRAVITY.CONSTANT)
P           THEN DO
        READ(5,1009) HT(J)
        MET(J) = HT(J)
P               END
P           ELSE DO
P              CATENATE(PASS2,#MECOM,', VTT(20)')  /* ADD MORE TO COMMON BLOCK */
        READ(5,1019) HT(J),VTT(J)
        MET(J) = RE/(RE+HAV)*GSL/GR*HT(J)+VTT(J)**2/(2.*GR)
P            /* RE AND GR HAVE VALUES FROM PASS1 TABLE SUBSTITUTED ON OUTPUT */
P       DELETE(PASS1,RE)
P       DELETE(PASS1,GR)
P     /* RE AND GR ELIMINATED FROM PASS1 TABLE--CANNOT BE SUBSTITUTED ANY LONGER */
P               END
  900 CONTINUE
```

```
P   /****          SAMPLE FCF FILE CONTINUED          ****/

          .
          .
          .
      RETURN
      END
P     END METEST
          .
          .
          .
P     END FCF


    /* FCF RULES USED BY MANY SOURCES CAN BE SAVED IN AN FCF USERS LIBRARY */
    /* THIS IS ONE OF MANY RULES THAT MAY BE LOCATED IN A COMMON LOCATION  */
          .
          .
          .
P   TABLE_READ:
P       /* THIS RULE READS TWO DIMENSIONAL TABLES.  THE FOLLOWING INFORMATION */
P       /* IS READ FROM INPUT:  NUMBER OF COLUMNS (OPTIONAL, DEFAULT = 2)     */
P       /* NUMBER OF ROWS (OPTIONAL, DEFAULT = 10), TABLE DATA ENTRIES BY ROW */
P       /* DATA SHOULD BE ORGANIZED IN INPUT IN THE FOLLOWING FORMAT:         */
P       /*    XXXXXXX   (NODE IDENTIFYING WHICH TABLE--POSITIONED BY USER)    */
P       /*       TABLE_INFORMATION  (IDENTIFIES BEGINNING OF TABLE INFO)      */
P       /*          COLUMNS = (NUMBER -- IF NOT PRESENT, SET TO 2)            */
P       /*          ROWS = (NUMBER -- IF NOT PRESENT, SET TO 10)              */
P       /*          DATA = TABLE VALUES BY ROW SEPARATED BY BLANKS OR COMMAS  */
P       /* TABLE DATA IS STORED IN A TABLE NAMED TABLE .  NO. OF COLUMNS AND  */
P       /* ROWS IS SAVED IN CURRENT DEFAULT SYMBOL TABLE AS COLUMNS AND ROWS  */
P     FIND(@.TABLE_INFORMATION)
P     SAVE_POINTER
P     IF FIND(@.COLUMNS)
P        THEN DO
P           INPUT('=')
P           INPUT(NUMBER)
P           SET(COLUMNS = *)
P           END
P        ELSE SET(COLUMNS = 2)   /* DEFAULT NO. OF COLUMNS */
P     RESTORE_POINTER
P     SAVE_POINTER
P     IF FIND(@.ROWS)
P        THEN DO
P           INPUT('=')
P           INPUT(NUMBER)
P           SET(ROWS = *)
P           END
P        ELSE SET(ROWS = 10)   /* DEFAULT NO. OF ROWS */
P     DEFINE TABLE(COLUMNS,ROWS)
P     RESTORE_POINTER
P     FIND(@.DATA)   ERROR: MESSAGE('IS:TABLE DATA MISSING')
P     INPUT('=')
P     SET(N=1)
P     DO WHILE N <= ROWS
P        SET(M = 1)
P        DO WHILE M <= COLUMNS
```

```
P  /****          SAMPLE FCF FILE CONTINUED          ****/

P            INPUT(CONSTANT)
P            ENTER(TABLE(M,N),*)
P            INCREMENT(M)
P            IF INPUT(',') THEN DO END
P                /* IF THERE IS A COMMA, READ IT IN BUT DO NOTHING */
P            END
P          INCREMENT(N)
P          END
P      END TABLE_READ
```

4.4 THE PRECOMPILER OUTPUT

Understanding the operation of the precompiler can probably be best accomplished by examining its output. Although only a portion of what the actual output might look like, the following subsections provide listings of a simulated precompiler run of the example described above. The first subsection lists all FORTRAN directly output or synthesized by the precompiler run. Although seemingly very little, large blocks of FORTRAN code could have been output in the missing sections.

The following subsection lists output produced by both error and trace conditions. The input file error conditions were written by the MESSAGE command as directed by the FCF. The FCF error messages were written by the precompiler because of the indicated syntactic errors. Both the severe errors in the input and the FCF files would inhibit the successful completion of a precompiler run. The last output in this section provides the details of a completed precompiler run with the TRACE switch set to high. This output provides an excellent means of tracing the actual operation of the precompiler. The annotations to the right of the actual FCF control statements are output during the trace mode and indicate the operations actually performed by the precompiler. Note that SYMBOL indicates a symbol table operation, INPUT an input file operation, and OUTPUT an actual FORTRAN statement written out. Note that on output, an * indicates the line was written out as it appears. Other annotations that appear are self-explanatory. The FCF code actually executed appears in bold type.

In the last subsection, a sample statistical and cross reference summary are produced. Note that both are incomplete and could possibly contain additional information when actually implemented.

4.4.1 FORTRAN Output

SAMPLE FORTRAN OUTPUT FOR ILLUSTRATIVE EXAMPLE

```
      PROGRAM SAMPLE
      COMMON/HECOM/ HT(20), HET(20), VTT(20)
      .
      .
      .
      CALL HETEST
      .
      .
      .
      CALL EXIT
      END
      .
      .
      .
      SUBROUTINE HETEST
      COMMON/HECOM/ HT(20), HET(20), VTT(20)
      DIMENSION VALUES(5)
      DATA VALUES/33.9,41.6,71.5,77.3,97.1/
      .
      .
      .
      DO 900 J=1,20
      READ(5,1019) HT(J), VTT(J)
      HET(J) = 2.09264258E+07/(2.09264258E+07+23000)*GSL/32.2456*HT(J)+
     1VTT(J)**2/(2.*32.2456)
  900 CONTINUE
      .
      .
      .
      RETURN
      END
      .
      .
      .
```

4.4.2 Trace and Error Messages

AN EXAMPLE OF INPUT FILE ERROR MESSAGES

```
INPUT FILE:

      CONSTANTS
         HAU = 23000
               *
*** S ***   CONSTANT NAME NOT RECOGNIZED
      GEOPHYSICAL_MODEL
         EARTH
            SPHERICAL
            NON_ROTATING
         GRAVITY
            VARIABLE
      REQUIRED_COMPUTATIONS
         ENERGY_WEIGHT
        *
*** W ***   UNUSED INPUT
      INTERPOLATION_ROUTINE = POLY
         TABLE_INFORMATION
            /* DEFAULT VALUE FOR COLUMNS IS 2 */
            ROWS = 5
            DATA = 0.   26.3
                   5.   69.8
                  11.   76.3
                  17.5  89.1
                  20.   99.7
         INTERPOLATION_VALUES = 1.63,2.1,6.2,11.6,19.3


*** INPUT ERROR SUMMARY ***
***    1 SEVERE ERRORS   ***
***    1 WARNINGS        ***
***    0 NOTES           ***
```

AN EXAMPLE OF FCF ERROR MESSAGES

```
P   MAIN:
P      SET(TRACE = 'HIGH')  /* USE TRACE FEATURE TO MONITOR FCF PROCESSING */
P      DO FOR SYMBOL TABLE  CONSTANTS   /* SET UP TABLE FOR LEGAL VARIABLE NAMES */
P         SET(HAV = 20000)
P         SET(RE = 209264258)
            .
            .
            .
P         END
P      DO FOR EACH SUBNODE OF CONSTANTS   /* READ IN CONSTANTS FROM INPUT */
P         INPUT(WORD)      ERROR:MESSAGE(I,'S:ILLEGAL CONSTANT NAME')
P         SYMBOL_EXISTS(LEGAL,*)   /* CHECK TO SEE THAT THIS IS A LEGAL CONSTANT */
P            ERROR:MESSAGE(I,'S:CONSTANT NAME NOT RECOGNIZED')
P         ENTER(CONSTANTS,*)   /* ENTER SYMBOL FROM INPUT INTO CONSTANTS TABLE */
P         INPUT('=')       ERROR:MESSAGE(I,'W:MISSING EQUAL SIGN'))
P         INPUT(CONSTANT)   ERROR:MESSAGE(I,'S:ILLEGAL CONSTANT VALUE')
P         ENTER(2,*)
P         END
P      ENTER(PASS2.HECOM)   /* SET UP PLACE HOLDER FOR COMMON STATEMENTS */
                 *
   *** W ***  MISSING COMMA -- ASSUMED PRESENT
                 *
   *** S ***  MISSING OR ILLEGAL SYMBOL NAME
            .
            .
            .
       PROGRAM SAMPLE
P      IF NODE_EXISTS(REQUIRED_COMPUTATIONS.ENERGY_HEIGHT)
P         THN OUTPUT(#HECOM,#)
          *
   *** S ***  EXPECTED 'THEN' CLAUSE IN AN 'IF' STATEMENT WAS NOT FOUND
P         /* HECOM IS A PLACE HOLDER FOR A COMMON BLOCK.  IT WILL    */
P         /* BE REPLACED BY THE PROPER FORTRAN ON PASS2 SUBSTITUTION */
            .
            .
            .
P      IF NODE_EXISTS(REQUIRED_COMPUTATIONS.ENERGY_HEIGHT) THEN
P         /* INCLUDE A CALL TO THE PROPER FORTRAN SUBROUTINE */
       CALL HETEST
            .
            .
            .
       CALL EXIT
       END
P         /* END OF THE MAIN PROGRAM BUT NOT NECESSARILY OF THE MAIN FCF RULE */
            .
            .
            .
P      IF NODE_EXISTS(REQUIRED_COMPUTATIONS.ENERGY_HEIGHT)
P         THEN INVOKE HETEST   /* TRANSFER OF CONTROL TO SPECIAL FCF RULE */
            .
            .
            .
P      END MAIN   /* END OF MAIN FCF RULE */
            .
            .
            .
   *** FCF ERROR SUMMARY ***
   ***  2 SEVERE ERRORS  ***
   ***  1 WARNINGS       ***
   ***  3 NOTES          ***
```

RUN SUMMARY

```
TRACE LOG -- FLTOPS PRECOMPILER:  JULY 15,1978 -- PAGE 1
   (OPERATIONS ACTUALLY PERFORMED ARE INDICATED IN BOLD PRINT)
P      DO FOR SYMBOL TABLE  CONSTANTS     /* SET UP TABLE FOR LEGAL VARIABLE NAMES */
P         SET(HAV = 20000.)                                                  *SYMBOL: (CONSTANTS,HAV) = 20000.
P         SET(RE = 209264258.)                                               *SYMBOL: (CONSTANTS,RE) = 20926425R.
            .
            .
            .
P         END
P      DO FOR EACH SUBNODE OF CONSTANTS    /* READ IN CONSTANTS FROM INPUT */      *INPUT:CONSTANTS
P         INPUT(WORD)      ERROR:MESSAGE(1,'S:ILLEGAL CONSTANT NAME')            *INPUT:HAV
P         SYMBOL_EXISTS(CONSTANTS,*)    /* CHECK TO SEE IF THIS IS A LEGAL CONSTANT */
P              ERROR:MESSAGE(1,'S:CONSTANT NAME NOT RECOGNIZED')
P         ENTER(CONSTANTS,*)    /* POSITION POINTER TO SYMBOL IN CONSTANTS TABLE */
P         INPUT('=')       ERROR:MESSAGE(1,'W:MISSING EQUAL SIGN')             *INPUT:=
P         INPUT(CONSTANT)    ERROR:MESSAGE(1,'S:ILLEGAL CONSTANT VALUE')       *INPUT:23000.
P         ENTER(2,*)                                                           *SYMBOL: (CONSTANTS,HAV) = 23000.
P         END
            .
            .
            .
       PROGRAM SAMPLE                                                         *OUTPUT:*
P      IF NODE_EXISTS(REQUIRED_COMPUTATIONS.ENERGY_HEIGHT)                    *INPUT:REQUIRED_COMPUTATIONS
P                                                                             *INPUT:ENERGY_HEIGHT
P         THEN OUTPUT('#HECOM',#)                                            *OUTPUT:#HECOM
P         /* HECOM IS A PLACE HOLDER FOR A COMMON BLOCK.  IT WILL     */
P         /* BE REPLACED BY THE PROPER FORTRAN ON PASS2 SUBSTITUTION */
            .
            .
            .
P      IF NODE_EXISTS(REQUIRED_COMPUTATIONS.ENERGY_HEIGHT) THEN               *INPUT:REQUIRED.COMPUTATIONS
P                                                                             *INPUT:ENERGY_HEIGHT
P         /* INCLUDE A CALL TO THE PROPER FORTRAN SUBROUTINE */
       CALL HETEST                                                            *OUTPUT:*
            .
            .
            .
       CALL EXIT                                                              *OUTPUT:*
       END                                                                    *OUTPUT:*
P         /* END OF THE MAIN PROGRAM BUT NOT NECESSARILY OF THE MAIN ECF RULE */
            .
            .
            .
P      IF NODE_EXISTS(REQUIRED_COMPUTATIONS.ENERGY_HEIGHT)                    *INPUT:REQUIRED_COMPUTATIONS
                                                                              *INPUT:ENERGY_HEIGHT
P         THEN INVOKE HETEST    /* TRANSFER OF CONTROL TO SPECIAL ECF RULE */ *RULE INVOKED:HETEST
P   /* HETEST IS ANOTHER RULE WITHIN THE CURRENT ECF */
P  HETEST:
     SUBROUTINE HETEST                                                        *OUTPUT:*
     #HECOM   /* PLACE HOLDER FOR PASS 2 INSERTION OF COMMON BLOCK */         *OUTPUT:*
            .
            .
            .
```

```
TRACE LOG CONTINUED -- FLTOPS PRECOMPILER:  JULY 15,1978 -- PAGE 2
P          /* PERFORM COMPUTATIONS TO INSERT DATA STATEMENT */
P       IF NODE_EXISTS(INTERPOLATION_ROUTINE)                                  *INPUT:INTERPOLATION_ROUTINE
P          THEN DO
P             FIND INTERPOLATION_ROUTINE   /* POSITION INPUT TO PROPER PLACE */   *INPUT:INTERPOLATION_ROUTINE
P             INPUT('=')  ERROR:MESSAGE(1,'W:EQUALS SIGN MISSING')               *INPUT:=
P             INPUT(WORD)   /* READ RULE NAME FROM INPUT */                      *INPUT:POLY
P             INVOKE *   /* TRANSFER CONTROL TO SPECIAL RULE -- FOUND IN INPUT */ *RULE INVOKED:POLY
P       POLY:
P    DO FOR SYMBOL TABLE POLY
P          /* SPECIAL RULE THAT CREATES DATA STATEMENTS   */
P          /* BASED ON VALUES DETERMINED BY INTERPOLATION */
P       IF NODE_EXISTS(@.TABLE_INFORMATION)                                    *INPUT:TABLE_INFORMATION
P          THEN DO
P          DEFINE TABLE AS POLY_TAB                                            *TABLE EQUIV: TABLE  POLY_TAB
P          INVOKE TABLE_READ                                                   *RULE INVOKED:TABLE_READ
P    TABLE_READ:
P          /* THIS RULE READS TWO DIMENSIONAL TABLES.  THE FOLLOWING INFORMATION */
P          /* IS READ FROM INPUT:  NUMBER OF COLUMNS (OPTIONAL, DEFAULT = 2)     */
P          /* NUMBER OF ROWS (OPTIONAL, DEFAULT = 10), TABLE DATA ENTRIES BY ROW */
P          /* DATA SHOULD BE ORGANIZED IN INPUT IN THE FOLLOWING FORMAT:         */
P          /*     xxxxxxx   (NODE IDENTIFYING WHICH TABLE--POSITIONED BY USER)   */
P          /*        TABLE_INFORMATION  (IDENTIFIES BEGINNING OF TABLE INFO)     */
P          /*           COLUMNS = (NUMBER -- IF NOT PRESENT, SET TO 2)           */
P          /*           ROWS = (NUMBER -- IF NOT PRESENT, SET TO 10)             */
P          /*           DATA = TABLE VALUES BY ROW SEPARATED BY BLANKS OR COMMAS  */
P          /* TABLE DATA IS STORED IN A TABLE NAMED TABLE2.  NO. OF COLUMNS AND  */
P          /* ROWS IS SAVED IN CURRENT DEFAULT SYMBOL TABLE AS COLUMNS AND ROWS  */
P       FIND(@.TABLE_INFORMATION)                                              *INPUT:TABLE_INFORMATION
P       SAVE_POINTER
P       IF FIND(@.COLUMNS)
P          THEN DO
P             INPUT('=')
P             INPUT(NUMBER)
P             SET(COLUMNS = *)
P             END
P          ELSE SET(COLUMNS = 2)    /* DEFAULT NO. OF COLUMNS */              *SYMBOL: (FLTOPS,COLUMNS) = 2
P       RESTORE_POINTER
P       SAVE_POINTER
P       IF FIND(@.ROWS)                                                        *INPUT:ROWS
P          THEN DO
P             INPUT('=')                                                      *INPUT:=
P             INPUT(NUMBER)                                                   *INPUT:5
P             SET(ROWS = *)                                                   *SYMBOL: (POLY,ROWS) = 5
P             END
P          ELSE SET(ROWS = 10)   /* DEFAULT NO. OF ROWS */
P       DEFINE TABLE(COLUMNS,ROWS)                                             *SYMBOL: TABLE DIMENSIONED TO 2,5
P       RESTORE_POINTER
P       FIND(@.DATA)   ERROR: MESSAGE('IS:TABLE DATA MISSING')                 *INPUT: ROWS
P                                                                              *INPUT: DATA
P       INPUT('=')                                                             *INPUT:=
P       SET(N=1)                                                               *SYMBOL: (POLY,N) = 1
P       DO WHILE N <= ROWS
P          SET(M = 1)                                                          *SYMBOL: (POLY,M) = 1
P          DO WHILE M <= COLUMNS
P             INPUT(CONSTANT)                                                  *INPUT:0.
P             ENTER(TABLE(M,N),*)                                              *SYMBOL: TABLE(1,1) = 0.
P             INCREMENT(M)                                                     *SYMBOL: (POLY,M) = 2
P                                                                                    .  SIMILAR OUTPUT FOR
P                                                                                    .  EACH TIME THROUGH
```

```
P                                                                                    .  LOOP
P            IF INPUT(',') THEN DO END
P               /* IF THERE IS A COMMA, READ IT IN BUT DO NOTHING */
P            END
P         INCREMENT(N)                                                  *SYMBOL: (POLY,N) = 2
P                                                                             .  SIMILAR OUTPUT FOR
P                                                                             .  EACH TIME THROUGH
P                                                                             .  LOOP
P         END
P   END TABLE_READ                                                      *RETURN TO RULE:POLY
P     DEFINE TABLE AS NULL                                              *TABLE EQUIV: TABLE NULLED
P     END
P          /* TABLE_READ IS AN EXAMPLE OF A RULE THAT MIGHT BE FOUND */
P          /* IN AN FCF USERS LIBRARY--SEE BOTTOM OF RUN             */
P          /* READS DATA INTO  TABLE NAMED TABLE2 AND SIZE INFO INTO */
P          /* SYMBOLS COLUMNS AND ROWS IN CURRENTLY DEFINED DEFAULT  */
P          /* SYMBOL TABLE */
P        ELSE .
                .
                .
P     DEFINE X(10),Y(10)  /* DEFINE TWO TABLES OF FIXED (10) LENGTHS */  *SYMBOL: X(10)
P                                                                         *SYMBOL: Y(10)
P     FIND(INTERPOLATION_ROUTINE.INTERPOLATION_VALUES)                    *INPUT:INTERPOLATION_ROUTINE
P                                                                         *INPUT:INTERPOLATION_VALUES
P          ERROR: DO .
                   .
                   .
P                  END
P     SET(N=0)                                                          *SYMBOL: (POLY,N) = 0
P     DO WHILE INPUT(',' OR '=')                                        *INPUT:=
P        INCREMENT(N)                                                   *SYMBOL: (POLY,N) = 1
P        IF N > 10 THEN MESSAGE(I,'S:TOO MANY VALUES READ IN')
P        INPUT(NUMBER)   ERROR: MESSAGE(I,'S:NUMBERS ARE THE ONLY VALID INPUTS') *INPUT:1.63
P        ENTER(X(N),*)                                                  *SYMBOL: X(1) = 1.63
P                                                                               . SIMILAR OUTPUT FOR
P                                                                               . EACH TIME THROUGH
P                                                                               . LOOP
P        END
P     IF N > 0
P        THEN DO
P           PARAMETERS POLY(REAL(COLUMNS,ROWS),INTEGER,INTEGER,REAL(N),REAL(N))
P           EXECUTE POLY(POLY_TAB(1),COLUMNS,ROWS,X(1),Y(1))            *PROCEDURE EXECUTED: POLY
P                                                                       *ARGUMENT PASSED:
P                                                                         TABLE2(1,1) = 0.,26.3,5.,69.8,...
P                                                                               . SIMILAR OUTPUT FOR EACH
P                                                                               . ARGUMENT PASSED
P                                                                               .
P                                                                       *ARGUMENT RETURNED:
P                                                                          TABLE2(1,1) = 0.,26.3,...
P                                                                                 . SIMILAR OUTPUT FOR
P                                                                                 . EACH ARGUMENT UPON
P                                                                                 . RETURN
P
P              /* COMPUTE Y VALUES FROM GIVEN X'S USING DATA FOUND IN TABLE2 */
```

```
TRACE LOG CONTINUED -- FLTOPS PRECOMPILER:  JULY 15,1978 -- PAGE 4
P            SET(DATA_ST = 'DATA VALUES/')   /* START BUILDING DATA STATEMENT */  *SYMBOL: (POLY,DATA_ST) =
P                                                                                            DATA VALUES/
P            SET(I=0)                                                             *SYMBOL: (POLY,I) = 0
P            DO WHILE I<N
P               INCREMENT(I)                                                      *SYMBOL: (POLY,I) = 1
P               CATENATE(DEFAULT,DATA_ST,VALUE(Y(I) FORMAT(F4.1))                 *SYMBOL: (POLY,DATA_ST) =
P                                                                                            DATA VALUES/33.9
P     IF I<>N THEN CATENATE(DEFAULT,DATA_ST,',')                                       . CONTINUE TO BUILD
P            ELSE CATENATE(DEFAULT,DATA_ST,'/')                                       . DATA STATEMENT
P                                                                                     . UNTIL LOOP ENDS
P               END
P            OUTPUT('DIMENSION VALUES(',N =>I ,')',#)                          *OUTPUT:DIMENSION VALUES(5)
P               /* INCLUDE PROPERLY DIMENSIONED VARIABLE NAME */
P            OUTPUT(DATA_ST,#)    /* OUTPUT FORTRAN DATA STATEMENT */          *OUTPUT:DATA VALUES/33.9,41.6,71.5,
P                                                                                        77.3,97.1/
P            END
P         ELSE DO
                .
                .
                .
P               END
P    /* NOW ELIMINATE UNNECESSARY TABLE STORAGE */
P      END   /* END OF DEFAULT SYMBOL TABLE BLOCK */
P      CLEAR_TABLE(X)                                                          *TABLE REMOVED:X
P      CLEAR_TABLE(Y)                                                          *TABLE REMOVED:Y
P      CLEAR_TABLE(POLY_TAB)                                                   *TABLE REMOVED:POLY_TAB
P      CLEAR_TABLE(POLY)
P      END POLY                                                                *RETURN TO RULE: METEST
P            END
           .
           .
           .
P      ENTER(PASS1,RE,VALUE(CONSTANTS,RE))                                     *SYMBOL: (PASS1,RE) = 209264258.
P      ENTER(PASS1,GR,GRAV_CONST*MASS1*MASS2/VALUE(PASS1,RE)**2)               *SYMBOL: (PASS1,GR) = 32.2456
P        /* ENTER COMPUTED VALUE FOR GR INTO PASS1 TABLE */
P    /* GRAV_CONST,MASS1,MASS2 ARE ASSUMED TO BE VALID SYMBOL TABLE ENTRIES */
P      ENTER(PASS2,#HECOM,'COMMON/HECOM/ HT(20), HET(20)')  /* UPDATE HECOM */  *SYMBOL: (PASS2,HECOM) = COMMON/HECOM/ HT(20), H
         .
         .
         .
       DO 900 J=1,20                                                           *OUTPUT:*
P      IF NODE_EXISTS(GEOPHYSICAL_MODEL.EARTH.SPHERICAL) AND                   *INPUT:GEOPHYSICAL_MODEL
P                                                                              *INPUT:EARTH
P                                                                              *INPUT:SPHERICAL
P        NODE_EXISTS(GEOPHYSICAL_MODEL.EARTH.NON_ROTATING) AND                 *INPUT:GEOPHYSICAL_MODEL
P                                                                              *INPUT:EARTH
P                                                                              *INPUT:NON_ROTATING
P        NODE_EXISTS(GEOPHYSICAL_MODEL.GRAVITY.CONSTANT)                       *INPUT:GEOPHYSICAL_MODEL
P                                                                              *INPUT:GRAVITY
P                                                                              *INPUT FAILED:CONSTANT
P         THEN DO
       READ(5,1000) HT(J)
```

```
TRACE LOG CONTINUED -- FLTOPS PRECOMPILER:  JULY 15,1978 -- PAGE 5
      HET(J) = HT(J)
P             END
P          ELSE DO
P             CATENATE(PASS2,#HECOM,', VTT(20)')  /* ADD MORE TO COMMON BLOCK */   *SYMBOL: (PASS2,HECOM) =COMMON/HECOM/
P                                                                                          HT(20), HET(20), VTT(20)
      READ(5,1019) HT(J),VTT(J)                                                    *OUTPUT:*
      HET(J) = RE/(RE+MAV)*GSL/GR*HT(J)+VTT(J)**2/(2*GR)                           *OUTPUT:209264258./(209264258.
                                                                                           *23000.)*GSL/32.2456*HT(J)+
                                                                                           VTT(J)**2/(2.*32.2456)

P           /* RE AND GR HAVE VALUES FROM PASS1 TABLE SUBSTITUTED ON OUTPUT */
P      DELETE(PASS1,RE)                                                            *SYMBOL REMOVED: (PASS1,RE)
P      DELETE(PASS1,GR)                                                            *SYMBOL REMOVED: (PASS1,GR)
P    /* RE AND GR ELIMINATED FROM PASS1 TABLE--CANNOT BE SUBSTITUTED ANY LONGER */
P             END
  900 CONTINUE                                                                     *OUTPUT:*
        .
        .
        .
      RETURN                                                                       *OUTPUT:*
      END                                                                          *OUTPUT:*
P     END HETEST                                                                   *RETURN TO RULE: MAIN
        .
        .
        .
P     END MAIN   /* END OF MAIN FCF RULE */                                       *STOP
END OF TRACE LOG
```

```
FLTOPS STATUS AT END OF RUN          FLTOPS PRECOMPILER:  JULY 15,1978

CONTENTS OF SYMBOL TABLES
      CONSTANTS
         HAV = 25000.
      FLTOPS
         ALPHA =
         DEBUG = OFF
         FUNCTION = FULL
         INTEGER  =
         LOCNUM   = 99000
         LENGTH   = OFF
         LOGICAL  =
         NOTES    = OFF
         REAL     =
         STATS    = OFF
         TRACE    = HIGH
         XREF     = OFF
      PASS1
         HAV = 25000
      PASS2
        #HECOM = COMMON/HECOM/ HT(20),HET(20),VIT(20)

FORTRAN STATEMENTS GENERATED OR MODIFIED DURING RUN:
      DIMENSION VALUES(5)
      DATA VALUES/33.9,41.6,71.5,77.3,97.1/
      HET(J)=209264258./(209264258.+23000.)*GSL/32.2456*HT(J)+
     1VIT(J)**2/(2.*32.2456)
      COMMON/HECOM/ HT(20), HET(20), VIT(20)


*** OVERALL ERROR SUMMARY ***
***    0 SEVERE ERRORS     ***
***    0 WARNINGS          ***
***    0 NOTES             ***
```

4.4.3 Summary Statistics

SAMPLE OF SUMMARY STATISTICS

RULE INVOCATION FREQUENCIES

RULE	INVOKED FROM	FREQUENCY
HETEST	MAIN	1
POLY	HETEST	1
TABLE_READ	POLY	1

.
.
.

SAMPLE OF SYMBOL CROSS REFERENCE MAP

RULE: MAIN

TABLE	SYMBOL	REFERENCES
CONSTANTS	HAV	9,12*,...
	.	
	.	
	.	
LEGAL	HAV	2,7,...
	RE	3,...
.		
.		
.		
PASS2	HECOM	11,...
.	.	.
.	.	.
.	.	.

RULE: TABLE_READ

TABLE	SYMBOL	REFERENCES
DEFAULT	COLUMNS	7*,9*,...
	M	20*,...
	N	18*,...
	ROWS	14*,15*,...
TABLE2	*ARRAY TABLE*	16,24*,...

5.0 SPECIFICATION MECHANISMS USED

To functionally specify the FLTOPS design, several distinct, yet complementary, mechanisms have been used. The principal specification mechanism used has been an augmented grammar technique. This technique combines classical syntactic specifications with embedded semantic information to rigorously define design requirements.

The semantic content of the augmented grammar is embodied in a set known as the functional primitives. These functions are primitive with respect to the precompiler since they define its primitive, or basic, operations. These primitives are specified within the FLTOPS design by a mechanism known as a Program Design Language or PDL. Using programming-language-like constructs combined with English language statements, a complete, yet simple and clearly understandable, specification of each primitive can be made.

In addition to the use of PDL descriptions, the lexical analyzers required by the FLTOPS design use as part of their specification the use of state-transition diagrams. These tree-structured diagrams describe completely the token identification process of the individual lexical analyzers.

Lastly, the FLTOPS design specification requires the use of supporting text and definitions. This additional information acts as supplemental data to the rigorous specifications already afforded by the augmented grammar technique adopted here.

Each of the above mechanisms with relevant examples is discussed in more detail in the remaining portions of this section.

5.1 AUGMENTED GRAMMAR

In the augmented grammar specification technique, the source language is defined by an ordinary grammar which contains additional information pertaining to its meaning. This additional semantic information is used to specify what elements of the object language are to be generated to correspond to particular source language elements. The augmented grammar thus defines the mapping from source to object language. With the exception of a small amount of supporting text, the augmented grammar gives a complete description of the precompiler.

Since the augmented grammar provides both the syntactic and semantic content of the language, both elements are discussed in detail in the following sections.

5.1.1 Syntactic Specification Technique

The method used to specify language syntax is that typically used in modern linguistics to describe context-free languages by means of a phrase structure grammar. Specifically, a notation based on the Backus-Naur form (Naur, 1960) is used.

Let us consider, first, a simple example containing only syntactic information. Figure 3 shows the syntactic definition of this sample language.

```
GOAL_RULE   :=  SENTENCE ;
SENTENCE    :=  NOUN_PHRASE  VERB_PHRASE  "." ;
NOUN_PHRASE   :=  ARTICLE  NOUN ;
VERB_PHRASE   :=  VERB  NOUN_PHRASE ;
ARTICLE   :=  "THE" | "A" ;
NOUN   :=  "BOY" | "GIRL" | "DOG" | "CAT" ;
VERB   :=  "HIT" | "SCRATCHED" | "FOLLOWED" | "LIKED" ;
.END
```

Figure 3. A Simple, Purely Syntactic Grammar

This particular language is sufficiently simple that its definition can also be expressed in English. By comparing the formal definition in Figure 3 with the natural-language description following, the reader can easily acquire a feeling for the metalanguage used in such formal definitions. The definition says:

1) The goal rule of this grammar is SENTENCE.
2) A sentence consists of a noun phrase followed by a verb phrase followed by a period (".").
3) A noun phrase consists of an article followed by a noun.
4) A verb phrase consists of a verb followed by a noun phrase.
5) An article consists of the word "THE" or the word "A". Note that the vertical line (|) means or.
6) A noun consists of the word "BOY", or the word "GIRL", or the word "DOG", or the word "CAT".
7) A verb consists of the word "HIT", or the word "SCRATCHED", or the word "FOLLOWED" or the word "LIKED".

Such a grammar can be viewed in two ways other than as a simple definition. First, it can be viewed as a generative grammar. This particular grammar is capable of generating 256 sentences, such as, "THE BOY HIT THE CAT", "A DOG FOLLOWED THE GIRL", etc. The generation of such sentences is accomplished simply by starting with the goal (in this case SENTENCE) and substituting its definition (NOUN_PHRASE VERB_PHRASE). Definitions are successively substituted for each variable that occurs until no variables remain. The resulting string is an instance of the goal (in this case, a sentence). This process can be viewed as generating a tree such as that shown in Figure 4, which describes the entire derivation of a sentence, or the phrase structure of the sentence.

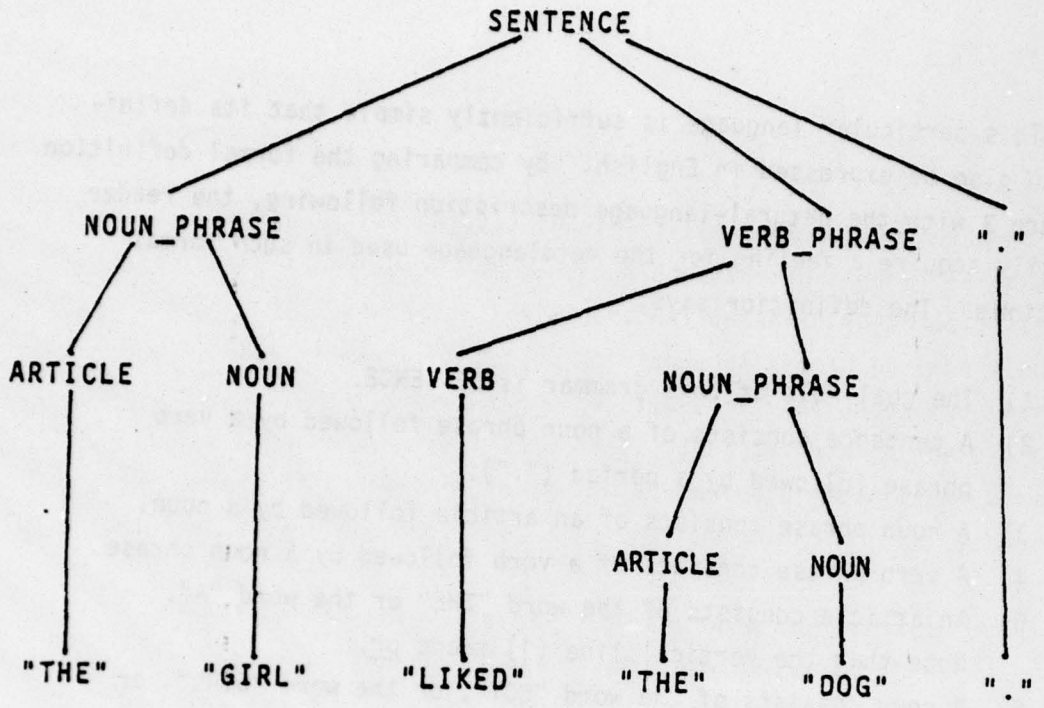


Figure 4. A Phrase-Structure Tree

The other way of viewing such a grammar, and the relevant one here, is as the definition of a parsing process. Parsing is the inverse of the generative process. Parsing starts with, in this case, a sentence like "THE GIRL LIKED THE DOG." and derives the underlying structure. Figure 4 can, therefore, be regarded as a parse tree, in which case the string at the bottom is the initial information and, with the help of the grammar, the structure is derived.

Based on the grammar of Figure 3, the parsing of the sentence "THE GIRL LIKED THE DOG." would proceed as follows. First, the "goal" rule of the grammar is SENTENCE. Assume, therefore, that we are looking

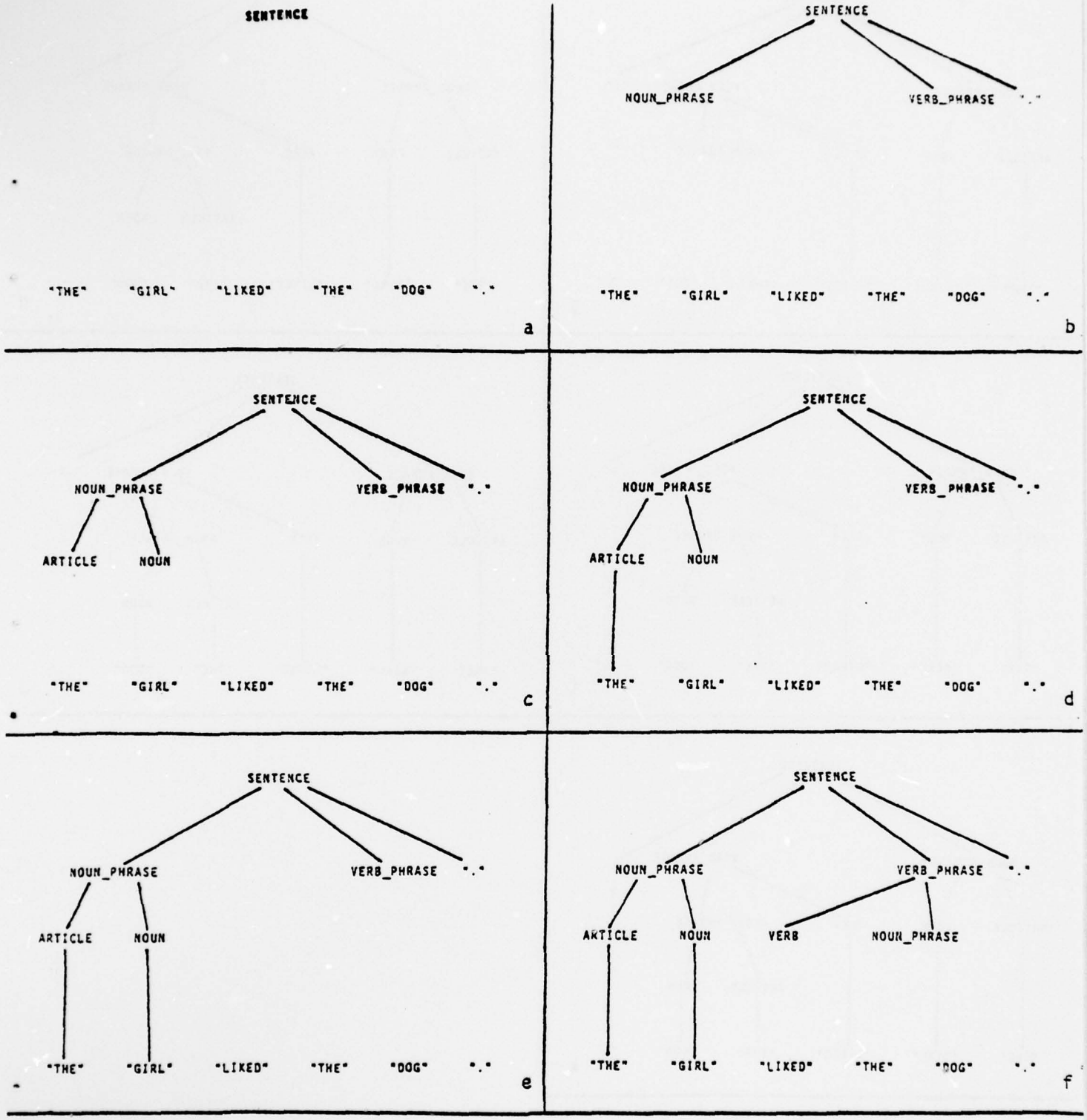


Figure 5. Steps in the Parsing Process

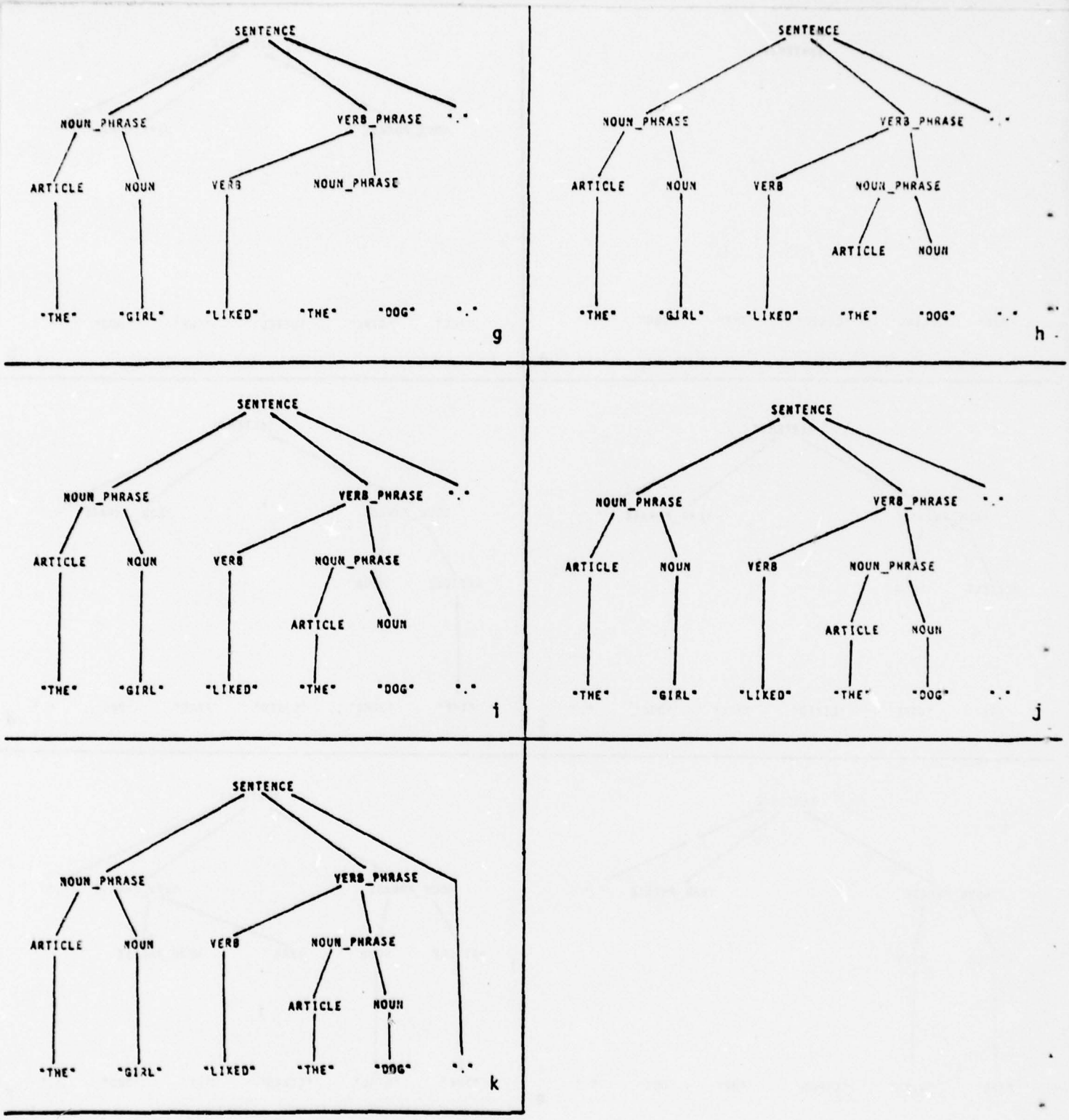


Figure 5 (Continued) Steps in the Parsing Process

for an instance of the entity defined by the rule SENTENCE. At this point, then, the parsing process has proceeded to the state illustrated in Figure 5(a), where the input string is illustrated at the bottom of the figure, and the "top-down" parsing process starts, naturally enough, at the top. SENTENCE, because it is not in quotation marks, is a "nonterminal" symbol. A nonterminal symbol must be defined in the grammar, and its definition can be substituted for the symbol itself. Making this substitution, we have Figure 5(b). Notice that we have added NOUN_PHRASE, VERB_PHRASE, and "." to the parse tree, because these three symbols were contained in the definition of SENTENCE given in Figure 3.

Continuing in the same manner, we substitute the definition of NOUN_PHRASE (ARTICLE NOUN) for the symbol NOUN_PHRASE, yielding Figure 5(c). Notice that, in each step, we are considering the leftmost "unsatisfied" symbol. Whenever that symbol is a nonterminal symbol, as it has been in all cases thus far considered, we substitute its definition. At this point, the leftmost unsatisfied symbol is ARTICLE. ARTICLE, however, is defined in terms of terminal (string literal) symbols. It has, in fact, two alternative definitions, "THE", and "A". Considering the first definition, we observe that "THE" is also the leftmost unused symbol of the input stream. It, therefore, satisfies the definition of ARTICLE, as indicated by Figure 5(d).

In Figure 5(d), the leftmost unsatisfied symbol is now NOUN. Since one of the definitions of NOUN is "GIRL", and this is the leftmost unused symbol in the input stream, these two symbols are connected as shown in Figure 5(e). At this point, the rules ARTICLE, NOUN, and therefore, NOUN_PHRASE, have all been satisfied. Our attention, therefore, turns to VERB_PHRASE, which is further defined and eventually satisfied in Figures 5(f), 5(g), 5(h), 5(i), and 5(j). Finally, the only remaining unsatisfied symbol is "." (top of Figure 5(j)). Since this is a terminal symbol, we have only to verify that it is, in fact, the next unused

symbol in the input stream. Because that is the case in this instance (bottom of Figure 5(j)), we make the final connection of the parse tree, yielding Figure 5(k).

The parsing process just outlined has not only verified that the sentence "THE GIRL LIKED THE DOG." is legal in the language of Figure 3, but has also derived the structure of that sentence (Figure 5(k)). Parsing is the first step of the translation process. Once the input string has been recognized and its structure ascertained, the question of its meaning (the semantic intent) can be addressed. This is not to suggest that parsing must be completed before the semantic intent can be interpreted, but parsing is logically first.

The augmented grammar translation method involves placement of information about meaning (e.g., object code) directly in the grammar. Viewed as a translation process, the result is that the meaning, or semantic intent, is interpreted as parsing occurs. How this is accomplished will be seen in detail in succeeding sections.

A point to be noted is that the structure and meaning of a language defined in this way is dependent for its definition on both the augmented grammar and the parsing properties that the parser is assumed to have. The meaning of the individual symbols of the augmented grammar is not necessarily sufficient to fully define the language. Under certain circumstances, it is also necessary to know the parsing method employed. Consider, for example, the following partial grammar for a programming language.

```
STATEMENT   :=
      CONDITIONAL_STATEMENT
    | UNCONDITIONAL_STATEMENT ;
CONDITIONAL_STATEMENT   :=
      "IF"  CONDITION  "THEN"  STATEMENT
      ( "ELSE" STATEMENT | .EMPTY ) ;
```

To avoid the necessity for complex definitions of CONDITION and UNCONDITIONAL_STATEMENT, the following additional dummy rules will be assumed.

```
CONDITION   :=  "CONDITION" ;
UNCONDITIONAL_STATEMENT   :=  "STATEMENT" ;
```

It can be seen that this grammar provides for nested conditional statements such as

IF CONDITION THEN IF CONDITION THEN STATEMENT ELSE STATEMENT.

The statement is ambiguous, however, because the grammar is ambiguous. Does the "ELSE"-clause go with the first or second "IF"? Considered from a purely grammatical point of view, it may not matter. However, if the statement is to have meaning, the ambiguity must be resolved since the two syntactic structures may not mean the same thing.

There are two approaches available for the resolution of such ambiguities. The first, and undoubtedly the most aesthetically pleasing, is to rewrite the grammar in unambiguous form. The grammar for conditional statements can, in fact, be rewritten so that any "ELSE"-clause will be associated with the innermost "IF" that doesn't yet have an "ELSE"-clause. Unfortunately, the resulting grammar is considerably more cumbersome than that just presented. It is unambiguous, however, and that is an absolute requirement.

The second approach is the one adopted here, both to keep the grammar simple and to allow some more powerful specification "tricks". This approach is to assume a particular parsing method. Specifically, top-down deterministic parsing has been assumed. This assumption allows otherwise ambiguous grammars (such as that just presented for conditional statements) to be unambiguous.

The basic properties of top-down deterministic parsing are: (1) once the first syntactic element in an expression has been recognized in the input string, the parser is committed to that expression and will not consider any alternative; (2) alternatives are considered in order; (3) if the first syntactic element of an alternative is not found, the next alternative is considered; (4) reiterating point (1), no backup ever occurs once the first syntactic element of an expression has been recognized.

Let us consider how top-down deterministic parsing affects the parsing of the statement

IF CONDITION THEN IF CONDITION THEN STATEMENT ELSE STATEMENT

using the previous grammar for conditional statements. The goal is a STATEMENT. Therefore, the input string is examined for concordance with the definition of STATEMENT. The first alternative, CONDITIONAL_STATEMENT, is considered and its first syntactic element, "IF", is recognized in the input string. "CONDITION" is readily identified as an instance of the required metavariable, CONDITION, and "THEN" is recognized. The next required element in the CONDITIONAL_STATEMENT definition is a STATEMENT. This makes STATEMENT the goal again, but the remainder of the input string is now

IF CONDITION THEN STATEMENT ELSE STATEMENT.

This is readily recognized as another conditional statement and the first three symbols are processed by the same mechanism as before. This returns us again to STATEMENT as a goal, but with only

STATEMENT ELSE STATEMENT

remaining in the input string. Since the next symbol in the input string is not "IF", the CONDITIONAL_STATEMENT alternative is rejected. "STATEMENT" is recognized as an UNCONDITIONAL_STATEMENT and removed from the input string.

Now consider the current status of the parser. It has just recognized the STATEMENT element required by the definition of the inside

CONDITIONAL_STATEMENT. It is still in the process of identifying the STATEMENT element of the outside CONDITIONAL_STATEMENT. What does it look for next? The next element it seeks is the symbol "ELSE" as a part of the inside CONDITIONAL_STATEMENT. Since the input string now looks like

ELSE STATEMENT

the requirement is satisfied, and the "ELSE"-clause is always associated with the innermost CONDITIONAL_STATEMENT. Hence, an ambiguous grammar has been rendered unambiguous in application.

5.1.2 Syntactic and Semantic Specification Technique

A purely syntactic grammar, such as that described in the preceding section, specifies the statements which are legal in a language and defines a parsing process. It does not, however, specify the meaning of the language or describe the functions of the language. By adding information to the grammar, however, we can define what functions are executed for any given input. Implicitly, such a definition specifies the "meaning" of the source language in terms of the basic operations or primitives that it performs. A grammar which contains this additional semantic information will be referred to as an "augmented" grammar.

A very simple augmented grammar is shown in Figure 6. This is a modified version of the purely syntactic grammar of Figure 3. Four output statements (.OUT) have been added. Recall that without these additional output elements, the grammar described a simple pseudo-machine which parsed simple English sentences, but which did not do anything with them. The output specifications cause the pseudo-machine to write out those sentences. Thus, the grammar of Figure 6 describes a "translator" which reads sentences and outputs those same sentences.

```
GOAL_RULE   :=  SENTENCE ;
SENTENCE    :=  NOUN_PHRASE  VERB_PHRASE  "."  .OUT(*) ;
NOUN_PHRASE :=  ARTICLE  .OUT(" ",*)  NOUN  .OUT(" ",*) ;
VERB_PHRASE :=  VERB  .OUT(" ",*)  NOUN_PHRASE ;
ARTICLE     :=  "THE" | "A" ;
NOUN        :=  "BOY" | "GIRL" | "DOG" | "CAT" ;
VERB        :=  "HIT" | "SCRATCHED" | "FOLLOWED" | "LIKED" ;
.END
```

Figure 6.

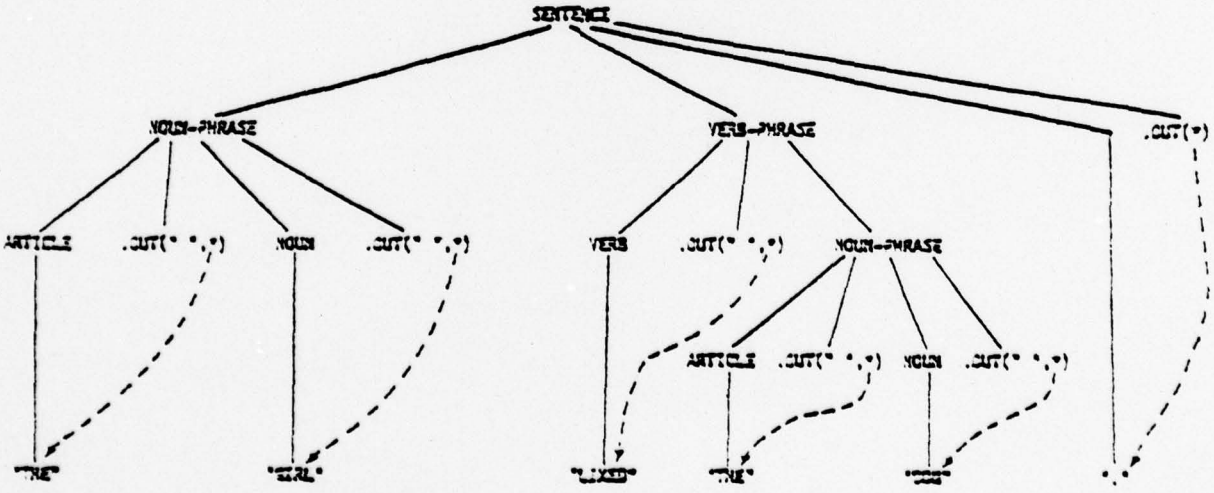
The .OUT statement causes the pseudo-machine to output one or more character strings as specified in an argument list. For example, the statement .OUT(" ") causes a single blank character to be written out. Any string literal can be output similarly. There are also several special symbols which specify output arguments. The asterisk, as in .OUT(*), is one of these. It refers to the last terminal symbol which has been recognized during the parsing process. For example, the SENTENCE rule contains the element "." .OUT(*). These elements specify that the parser is to look for a period (".") and, having found it, is to output the last token found (which is, of course, the period). In this case, the rule might have said "." .OUT("."), with equivalent result. Two or more output arguments can be combined into a list by placing commas between them, as in .OUT(" ",*).

Figure 7 is a parse tree for the grammar of Figure 6., as applied to the sentence "THE GIRL LIKED THE DOG." Except for the added .OUT statements, the parse tree is identical to that shown in Figure 4 and 3. The sequence of events which occur during this parsing process can be determined by reading the tree from left to right. After the article "THE" has been recognized, the statement .OUT(" ",*) causes the output of a blank and the word "THE". In each case, the dashed arrow points to the last recognized token. Since the asterisk refers to the last recognized token, the arrows indicate the character string which will be output by

the .OUT(...*) specifications. The parse tree indicates that eleven outputs occur when this particular sentence is executed. These outputs are:

Figure 7. Parse Tree for Augmented Grammar of Figure 6

```
ø (blank)
THE
ø
GIRL
ø
LIKED
ø
THE
ø
DOG
.
```



The blank at the beginning of the sentence could have been avoided, but the grammar would have been somewhat more complicated.

It should be kept in mind that the grammar describes the function of a pseudo-machine. As such, .OUT in the above example describes an output of the pseudo-machine. The mechanism by which this output is produced at execution time is contained in the description of the .OUT primitive operation. This description might take the form of a detailed object code program, flowchart descriptors of information flow or the use of a program design language. Because of its ease and high degree of readability, the use of a PDL to describe such primitive operations as used by a grammar is recommended. References to any number of primitives can appear embedded within the syntactic description of the language as they are required to describe the translation process.

5.2 PROGRAM DESIGN LANGUAGE (PDL)

The use of a Program Design Language, or PDL, in detailing program design specifications is relatively new. Classical methods, such as flowchart descriptions, have proved to be time-consuming to produce, as well as very difficult to follow because of the technical nature of the presentation. On the other hand, a PDL is a non-technical, English-like form of notation which uses a few programming language structures to convey program logic precisely without sacrificing readability. The PDL syntax and basic structure used here is based on that of the programming language PL/1.

A series of PDL statements which perform a task is called a procedure, or in the context of the augmented grammar specification technique, a primitive. PDL statements are separated by semicolons (;), with each primitive being terminated by an END statement appropriately labeled with the name of that primitive. Each new statement begins on a new line, with those statements requiring multiple lines having each continuation line indented some number of spaces. PDL statements consist of a few control structures that influence the order in which PDL statements are executed. Execute in this context has a special meaning since the reader of the PDL is doing the "executing" and not a computer. In the examples that follow, all control structures will be capitalized so that they stand out. Except for control structures, all PDL statements are written in non-technical English.

The IF control structure is used to make a decision with two possible outcomes. It looks like this:

```
IF condition
THEN statement
ELSE statement
```

Condition is a proposition which may be true or false. If the condition proves to be true, the THEN statement would be executed. If false, the ELSE statement would be executed. The appearance of the ELSE is optional, appearing only if an either/or condition exists. Otherwise a simple IF...THEN... is sufficient.

As an example of an IF structure:

```
IF you have finished your test;
   THEN hand it in;
   ELSE continue working on it;
IF it is after 10 p.m.   THEN go to bed;
```

The IF structures can be nested. This means that the THEN or ELSE (or both) statements may be another IF structure. Therefore, decisions with more than two possible outcomes can be made. As an example, consider:

```
IF you are lost
THEN IF you see a policeman
   THEN ask him for help;
   ELSE ask any grown-up for help;
ELSE IF you have carfare
   THEN take a bus home;
   ELSE walk home.
```

Notice that the nested statements are indented. This is a PDL convention that makes the structure of the program more visible. Notice also that an ELSE statement coming immediately after an IF belongs to that same IF.

The simple DO structure causes a group of statements to be executed as if they were a single statement. A common usage of the DO structure is with the IF to execute a group of statements when a decision is made. The DO structure looks like this:

```
DO;
   statement;
   statement;
      :
      :
   END;
```

As an example of the DO structure combined with an IF:

```
IF this is a weekday
THEN DO;
   Get dressed;
   Eat breakfast;
   Go to school;
   END;
ELSE go back to sleep;
```

The DO WHILE structure is used to repeat a group of statements a controlled number of times. Its form is:

```
DO WHILE condition;
   statement;
   statement;
      :
      :
   END;
```

The condition is a true/false proposition as in an IF structure. The DO WHILE tests the condition and executes the statements if it is true, then tests it again and reexecutes, continuing until the condition proves to be false. At that time, it passes control to the statement following the END. As an example, consider:

```
Take out your work book;
DO WHILE there are problems left
 in today's assignment;
   Solve the next problem;
   Check your solution;
   END;
Put away your work book;
```

The DO FOR structure allows a group of statements to be repeatedly executed based upon a set of values. It has the following form.

```
DO FOR variable=values;
   statement;
   statement;
      :
      :
   END;
```

As an example of this kind of control structure, consider:

```
X = 0;
DO FOR I = 1 to 31;
   IF you have school on day of
    month "I"
   THEN add 1 to X;
   END;
```

In addition to the above control structures, conditions and statements can be joined by using AND and OR. In this way, complex conditions can be made out of simple ones. They have their usual English meanings.

Although there are other logical structures that can be added to a PDL to account for peculiar variations, the above list provides sufficient flexibility to be used in adequately describing most situations.

The primitives used in the augmented grammar are all written using a PDL with these control structures.

5.3 STATE TRANSITION DIAGRAMS FOR LEXICAL ANALYZERS

The lexical analyzer is responsible for recognizing the elementary units of information, known as tokens, contained in the source data. This source can be thought of as an input stream consisting of individual characters. These characters must be grouped into meaningful units (e.g., words) before the parser can operate on them. The lexical analysis procedure is basically the same no matter what input data are assumed. The character groups, however, vary widely depending upon the uses of the input data. To identify what the lexical analysis should produce is a two stage process. First a verbal description should be developed of the various token classes and literal tokens that are acceptable as input. From this verbal description, a state transition diagram can be constructed which allows a simple, yet exact, specification of the token recognition process for the lexical analyzer. Consisting of a tree structured group of arcs and nodes, the state-transition diagram accurately describes how each individual character in the input stream would be processed. Several state transition diagrams have been constructed for FLTOPS.

To understand this concept more exactly, consider the simple example discussed in the sections on the augmented grammar specification technique. In this example, the sentence, "THE GIRL LIKED THE DOG.", is a legal statement. This language includes one token class, the "word" or "identifier" (the latter term is more commonly used in the context of programming languages). This class might have the definition

.ID - Identifier. Unlimited in length, all characters alphabetic.

In addition, this simple language uses one literal token which needs have no class name. This token is the period.

The next step in the process is to convert the definition to a state transition diagram which defines the character-by-character scanning process to be performed by the lexical analyzer. Twenty-eight characters are legal for this particular lexical analyzer: A, B, C, D, E, F, G, H, I,

J, K, L, M, N, O, P, Q, R, S, T, U, V, W, X, Y, Z, period (.) and blank (ß).

Figure 8 shows the state transition diagram for our example. The lexical analyzer always starts in state zero. One state transition is made for each character removed from the input string. Whenever a purely final state is reached, a complete token has been scanned. For example, state 2 is a final state, since no state transitions from state 2 are indicated. Note that state 2 is also identified as a legal final state by the presence of "F" next to the state 2 node. State 1 is a legal final state (it has an "F"), but it is also a possible intermediate state, since one or more state transitions from state 1 are possible. When the current state is an intermediate state, another character must be scanned. If that character causes a legal state transition, the process continues. If not, then either a token has been processed (if the current state is also a possible final state) or an error has occurred.

Notice that state 1 also has a state name ("ID"). This indicates that any token for which the state-transition process terminates at state 1 is an identifier.

By way of illustrating the meaning of this state-transition diagram, let us assume that the first four tokens (in this case, ID's) of the sentence, "THE GIRL LIKED THE DOG.", have already been recognized and removed from the input stream. Five characters remain: blank (ß), D, O, G, and period (.). The lexical analyzer starts in state zero and attempts to determine whether the next character of the input stream allows a transition from its current state. Since the next character (blank), is recognizable from state zero, it is removed from the input stream and the lexical analyzer enters the state indicated by the arc associated with the recognized character (ß). In this case, the new state is again state zero. Since the initial state is reentered, the character scanned is simply ignored. Thus, in this case, any number of blanks between tokens are ignored.

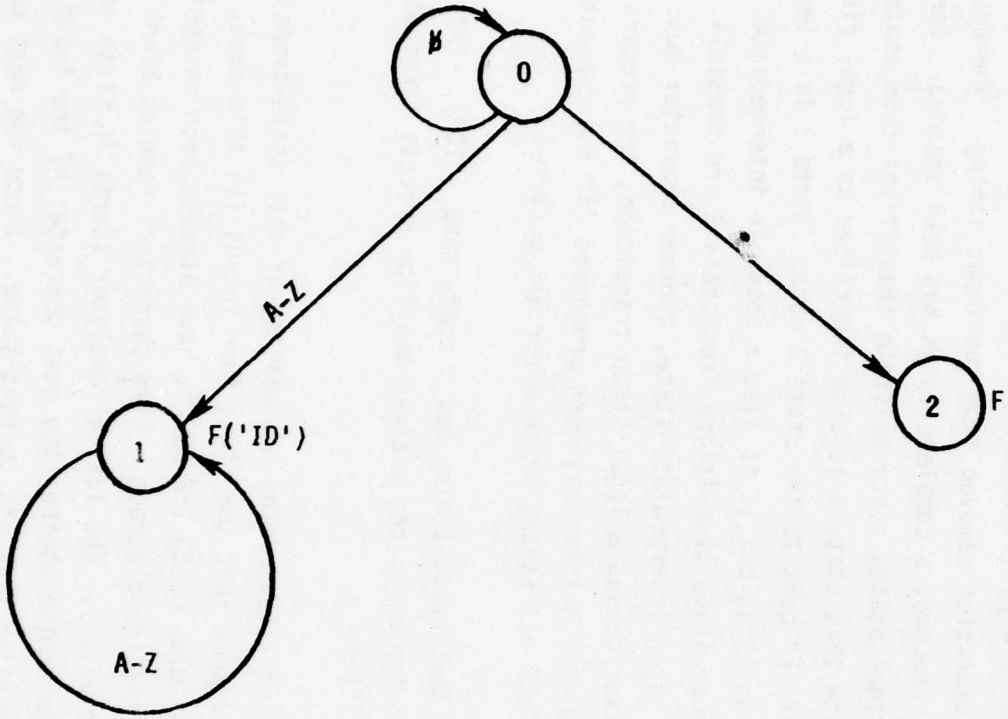


Figure 8. An Example of a Simple Lexical Analyzer State-Transition Diagram

The input stream now contains four characters (DOG.) and the current state is state zero. The next character (D) allows a transition to state 1 via the "alphabetic" arc labelled A-Z. The character scanned (D) is added to the (presently empty) string which represents the token being developed. Thus, the token is now "D", the input stream contains the characters "OG.", and the lexical analyzer is in state 1.

The next input character (O) allows a transition from state 1 to state 1 (the second A-Z arc). The current token is now "DO", and the input stream is "G.". Similarly, the next character (G) allows a transition via the same arc, and the token and input strings are "DOG" and ".", respectively.

Note that a period (.) does not allow a legal transition from state 1. Only alphabetic characters can be processed from state 1. The lexical analyzer is, therefore, finished. It returns the token "DOG", and leaves the input stream (now ".") for processing on a subsequent call.

Figure 8 has one remaining property which deserves comment. The arc which exits and enters state zero causes blanks between tokens to be ignored. In fact, any input stream entity can be ignored, and therefore, not passed to the parser, by returning to state zero of the lexical analyzer. This mechanism is commonly used to handle comments. If, for example, the format for a comment in the source language is "/*", followed by a character string, followed by "*/", the lexical analyzer might provide that any string of this sort causes a return to state zero. Assuming that a standard lexical analyzer procedure is used, this results in a complete restart of the lexical analyzer, with the input pointer advanced past the comment.

State-transition diagrams are actually a little more sophisticated than this, but this example is correct as far as it goes, and should illustrate the concept of lexical analysis.

5.4 TEXT AND GLOSSARIES

In addition to the above specification mechanisms, a small amount of supporting text and glossaries are often needed to completely bring into focus the concepts furnished by the design. Textual supplements serve as philosophical aids more than as technical support for the design. Such an aid is necessary, however, for a full appreciation of the design concepts, and to bring to the forefront basic assumptions inherent in the design.

Glossaries add still another dimension to the design specification. By supplying definitions for key words or elements used in the design, the augmented grammar is made more readable and its various relationships more understandable. Definitions are not supplied for those elements whose use is localized and whose meanings are clearly supplied by their context.

6.0 FORMAL SPECIFICATION OF FLTOPS DESIGN

6.1 DISUCSSION OF DESIGN CONCEPTS

The precompiler design consists of a number of components that perform the functional requirements of the system. Figure 9 schematically shows the functional relationship between components within this design. These components may be separated into several categories: the input preprocessor which transforms the input file into a format easily accessible by precompiler primitives; the lexical analyzers which process the input, FCF and FORTRAN data for token recognition; the statement processor which performs supervisory control over all functions; the functional primitives that perform the actual processing functions of the system; the symbol table storage area containing all values to be used in the customization process; and the output processor which writes out the customized FORTRAN code. Each of these areas is discussed in more detail below.

6.1.1 Input Preprocessor

The input file as described above will contain hierarchic data and, possibly, some special FCF rules under development. Because of the probable frequency with which the input data will be referenced in the FCF, an input preprocessor would provide these types of functions: construct a map of the hierarchic form of the data for speedy access and construct a rule name table of all special FCF rules contained in input. Both of these functions are intended to streamline data access while minimizing the amount of scanning necessary to locate specific input requirements.

6.1.2 Lexical Analyzers

Lexical analysis involves the analysis of the input stream (a stream of individual characters) and recognition of the elementary units of information, referred to as tokens, contained therein. The precompiler design uses three different analyzers; each responsible for a separate data group -- input, FCF and FORTRAN type data. Each lexical analyzer makes the last

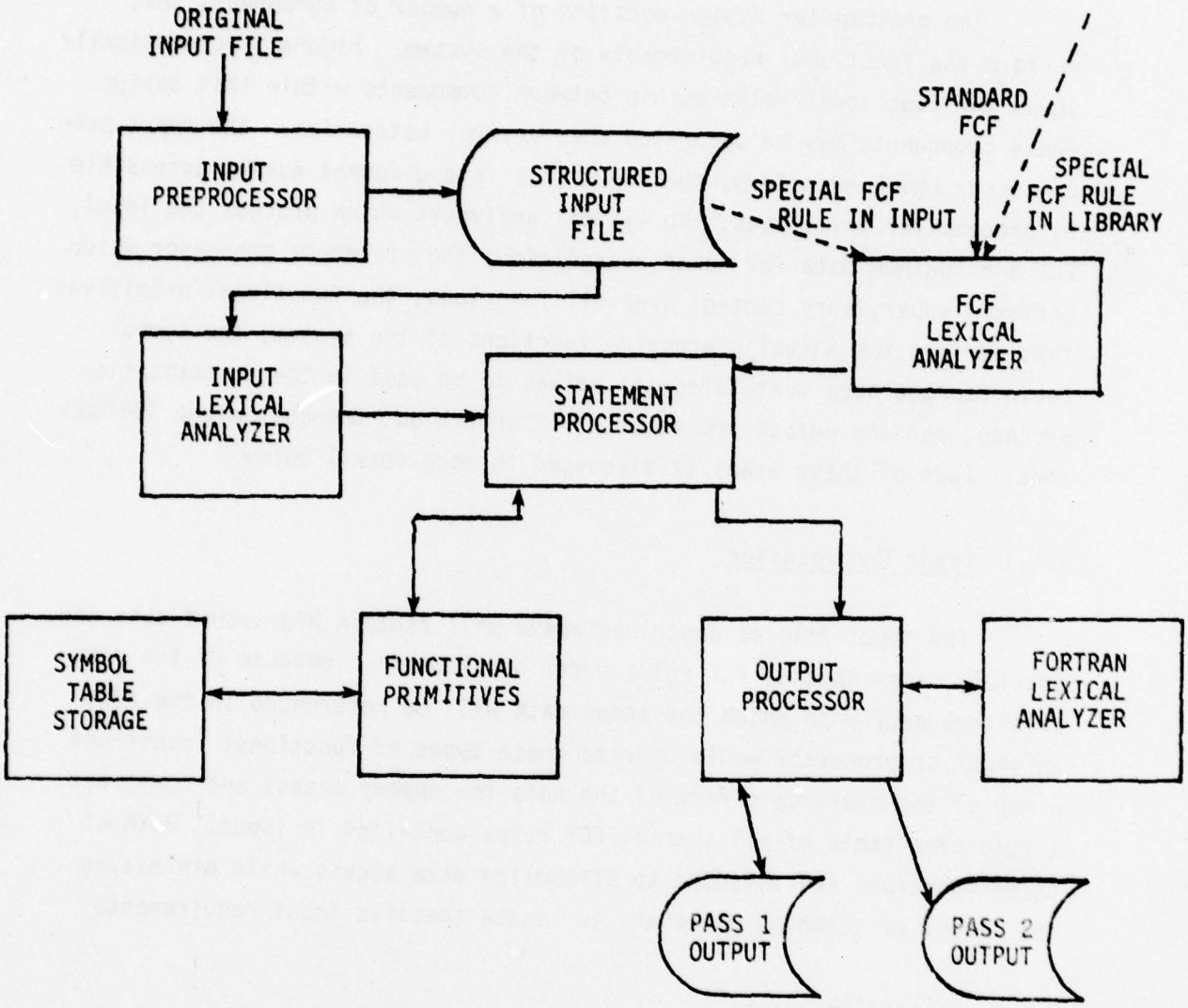


Figure 9. Basic Precompiler Architecture

token recognized available to the precompiler for use in the desired fashion. Although they operate on different data streams and have different types of tokens that they recognize, they all perform similar operations. Table 4 identifies the types of tokens and an example of each that is recognized by each lexical analyzer.

Several things should be noted about the specific lexical analyzers. The FCF lexical analyzer recognizes entire FORTRAN statements by the absence of a P as the first character in a line. The entire FORTRAN line is processed as a single token. Upon output, however, the FORTRAN lexical analyzer performs an analysis of each token to determine if it should be replaced. Note also that the FORTRAN lexical analyzer actually consists of two analyzers: one that performs a special analysis of FORTRAN format statements, and the other for all other statement types. Tokens within a FORTRAN format statement are the individual specifications, e.g., 5A4 and 15HTHIS IS A TOKEN and *ANOTHER TOKEN*.

6.1.3 FCF Statement Processor

The FCF statement processor functions as the supervisor for controlling the overall functions of the precompiler. It serves two principal functions: to determine syntactic correctness of the FCF, issuing error messages if appropriate; and to dispatch precompiler functional primitives in accomplishing the semantic intent of each statement.

In controlling the precompiler functions, the statement processor can be in any one of three functional states. These functions are as follows:

FULL function: all precompiler functional primitives are operational; data can be read from the input file and customized FORTRAN can be output; all statements are checked for proper syntax.

TABLE 4. LEGAL TOKEN CLASSES IDENTIFIED BY LEXICAL ANALYSIS

INPUT LEXICAL ANALYZER

DATA TYPE	EXAMPLES
IDENTIFIER	EARTH NON_ROTATING M1011
NUMBER	7 8.3 9.61E-5
SIGNED_NUMBER	-10 +8.45 -1.E+10
STRING	'ANY CHARACTERS WITHIN DELIMITERS'
SPECIAL SYMBOLS	+ - * / ** ( ) = ,< > @ $: ;
COMMENT	/* THIS IS A COMMENT */
blank	ignored

FCF LEXICAL ANALYZER

DATA TYPE	EXAMPLES
IDENTIFIER	ENTER VARIABLE_X X17 #COMMON
NUMBER	7 26.32E5 18.
STRING	'ANYTHING BETWEEN APOSTROPHES'
LABEL	$7 $15
SPECIAL SYMBOLS	* + - ** / # = ( ) , @
COMMENT	/* COMMENTS LOOK LIKE THIS */
blank	ignored

FORTRAN LEXICAL ANALYZER

DATA TYPE	EXAMPLES
IDENTIFIER	VARBLE I X17
NUMBER	7 18.3 9.E-5
STRING	'ANY CHARACTERS'
SPECIAL SYMBOLS	+ - * / ** = , ( ) $
PLACE HOLDER	#A #COMMON #SYMBOL
COMMENT	any statement with a C or * or $ in column 1
FORMAT STATEMENT	any legal FORTRAN format; individual specifications are treated as tokens, e.g., F10.4 and 10I3 are single tokens.

TEST function: all precompiler functional primitives are operational except those relating to the output processor, i.e., no customized FORTRAN is output, but the input file can be read and the FCF checked for syntax.

SYNTAX function: all precompiler functional primitives are inoperative except for those relating to syntax checking of the FCF. The statement processor enters the syntax function mode automatically whenever a severe error occurs or whenever a portion of the FCF must be bypassed because of a conditional construct.

Figure 10 contains a simplified schematic describing the basic analysis of a token by the statement processor.

Upon receiving the next token from the FCF lexical analyzer, the statement processor determines the syntactic correctness of the token in the context of its location within the FCF. If an error occurs, an appropriate message is issued. If the error was severe, i.e., the precompiler cannot interpret the semantic content of the statement, the precompiler is directed to provide syntax checking only for the remaining FCF to be checked. In the case of a severe error associated with data read from the input file, the precompiler FUNCTION switch is reduced to TEST. As a result, the user can continue checking the input file for additional errors, but will receive no FORTRAN ouptut. After insuring that the FCF should be executed, the relevant functions accomplishing the semantic content of the FCF input stream are executed. FORTRAN output is produced only if the statement processor has all of its functional capability enabled.

6.1.4 Functional Primitives

Associated with each FCF control statement are one or more functions that, when executed with appropriate parameters, perform the actual

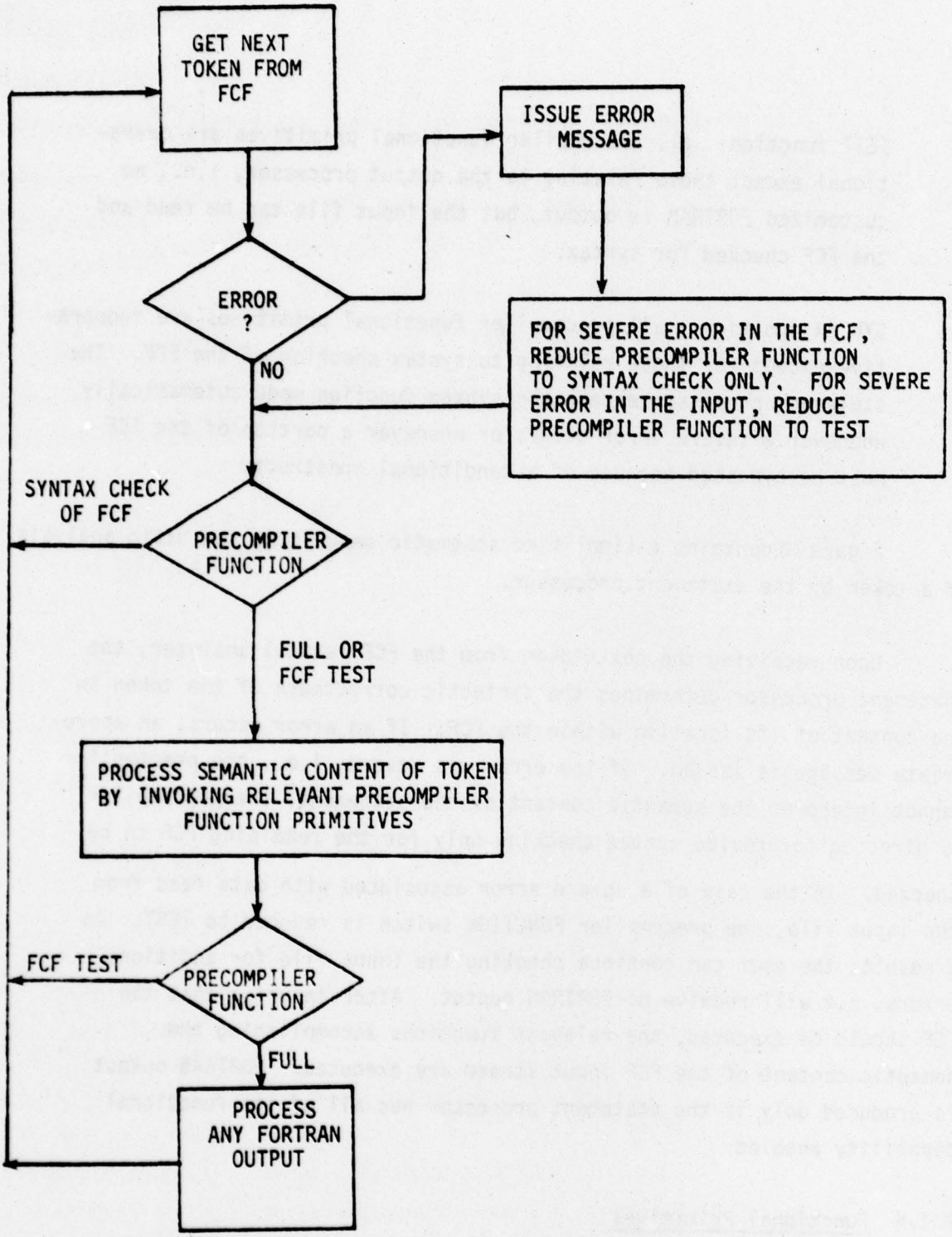


Figure 10. Basic Operation of FCF Statement Processor

operations implied by the FCF control language. Referred to as the functional primitives, this set of processes performs the actual semantic intent of the FCF control statements. They are known as primitives because, with respect to the precompiler, they perform its basic functions. These operations can be categorized briefly into these groups: symbol table access/modification functions, input file control operations, statement processor control functions, arithmetic operations, boolean operations, output control functions, error processing/recovery processes, FCF parsing control operations and processes which allow computation of precompile time values. Each category performs functions related to identifiable portions of the basic precompiler architecture. Although all functions of the precompiler can be thought of as primitives, those relating to input and output have been identified separately for reasons of clarity in understanding.

6.1.5 Symbol Table Storage

Symbol tables play a vital role in the precompiler design. There exists in the FCF control language a fairly general capability for the generation, manipulation and use of the symbol tables by the FCF builder. Symbol tables can be thought of as the main storage area for the precompiler. Once a location has been specified, the data at that location can be retrieved/stored/modified by appropriate precompiler primitive functions.

Implicit in the FLTOPS design are two types of symbol tables. Although their logical structure can be identical, the user references them differently. The first is the classical symbol table structure. In this type of symbol table, a value string is stored/retrieved/modified by reference to an associated symbol name in a specified table.

In the second type of symbol table, value strings are stored/retrieved/modified by reference to a numbered location within a specified table. These array tables are similar in function to dimensioned

variables in other programming languages such as FORTRAN or PL/1. The critical difference, however, is that as in the classical symbol table structure, the value string can be of arbitrary length. Although an implementation scheme might use a different approach, the subscripts of the specified table can be thought of as being a symbol; i.e., they point to the desired table entry. Such a parallelism makes the logical structure of the two table types appear the same.

Although items can be transferred directly from input to output, the majority of FCF functions directly or indirectly use information stored by the user in the symbol tables. Although no implementation procedure is implied, it will be useful to assume that symbol tables be logically structured as shown in Figure 11. This structure allows an arbitrary list of symbol tables, each identifiable by table name. Each table may contain an arbitrary list of symbol entries, each identifiable by sumbol name. Table records and symbol records are assumed to be chained together in some fashion to allow a search for the desired table and symbol. Associated with each symbol entry is a "value." The value is a character string of logically unlimited length and content. The value string may have length zero (i.e., may be a null string).

In addition to providing a general purpose symbol table capability for use by the FCF builder, FLTOPS symbol tables also control certain data-driven functions of the precompiler. Specifically, the FLTOPS pass 1 and pass 2 FORTRAN string replacement functions are controlled by the symbol tables whose names are "PASS1" and "PASS2". The precise mechanism for this is discussed in the next section on the output processor. As a simple example, though, consider again the symbol tables of Figure 11. Notice that the "PASS1" table contains an entry for the symbol "PI" with value "3.14159". The value represents, in this case, a replacement string. While this entry remains in the PASS1 table and the precompiler is active and in pass 1, any occurrence of the variable name "PI" in the FORTRAN generated by the precompiler will result in substitution of the value 3.14159.

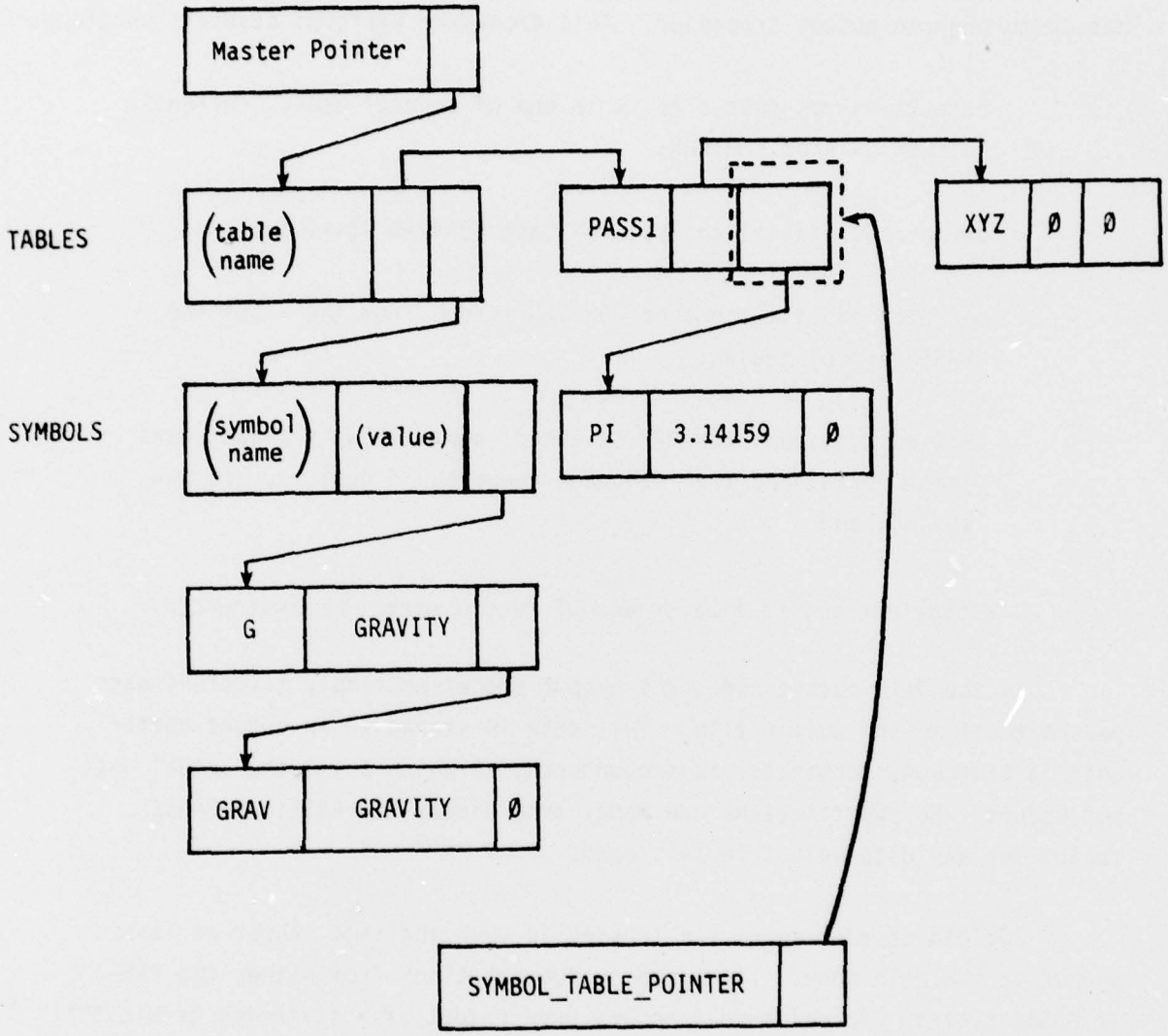


Figure 11. Logical Structure of Symbol Tables

6.1.6 Output Processor

Except for messages and listings all output from the precompiler is passed through an output processor. This processor performs several functions:

outputs values passed to it in one of several modes, currently either DATA or FORTRAN;

performs a lexical analysis of each FORTRAN statement;

performs substitutions of FORTRAN tokens from the PASS1 and PASS2 symbol tables;

insures that each FORTRAN statement complies with proper card image format and that sequence numbers, if desired, are inserted; and

writes out end of file or end of record marks as instructed.

In the DATA output mode, the output processor simply transfers data passed to it to the output file. This data is stored in an output buffer until a statement terminator is encountered, at which point the actual output occurs. No substitutions are made, from either the PASS1 or PASS2 tables for any data output in this mode.

FORTRAN comment cards are treated in much the same manner as items output in the DATA mode. There are no substitutions from either the PASS1 or PASS2 tables. Actual output occurs upon output of a statement terminator.

As each FORTRAN statement is encountered in the FCF, it is passed to the output processor. If the PASS1 table is non-empty, the output processor calls upon the FORTRAN lexical analyzer to perform a lexical analysis of the statement. Each token recognized by the lexical analyzer is matched against those symbols appearing in the PASS1 symbol table. If a match occurs, the symbol table value is substituted for the token before it is

stored in an output buffer. Substitutions of this type can be recursive. The output processor guards against instances of infinite recursion by allowing only a fixed number of replacements for any one token in a FORTRAN statement. Once the last token of the statement has been processed, the output buffer is written to a temporary file referred to as the PASS1 output file.

Upon completion of the FCF, the precompiler checks to see if the PASS2 table is non-empty. If so, the precompiler continues processing, otherwise it is finished. Referred to as PASS2, the precompiler accesses the PASS1 output file which contains output from PASS2 (no FCF control statements appear here). A lexical analysis is performed on each FORTRAN statement, but in this case, substitutions are made from the PASS2 symbol table. Note that no substitutions are made on data which was output in the DATA mode during PASS1.

Note that since no FCF control statements are executed, PASS2 substitution is global in scope. This may be extremely useful for insertions of common blocks and dimension or data statements whose complete structure may not be known until some point after their required insertion point has been processed. Extra processing is required, however, to accomplish this extra substitution and some care should be exercised in its use.

Once the last token of each statement has been processed during PASS2, the precompiler writes the statement out to what is known as the PASS2 output file.

Whether PASS2 substitution actually occurs or not, all FORTRAN statements are checked to insure that proper FORTRAN card image syntax is maintained. Data output in the DATA mode is left unaltered. The input to the compiler is contained on the file referred to in this document as PASS2, regardless of whether or not PASS2 table substitutions occurred.

The output processor is not operational if the statement processor is not at FULL function capability.

```
/*********************************************************/
/* SPECIAL SYMBOLS USED TO INDICATE CONTROL STRUCTURE */
/*********************************************************/
```

$ PERFORM THE FOLLOWING UNIT ZERO OR MORE TIMES, AS LONG AS IT CONTINUES TO SUCCEED. SUCCESS IS DEFINED AS A TRUE VALUE OF THE FCF_PARSING_TRUE_FALSE_INDICATOR.

(...) CONSIDER THE ELEMENTS WITHIN PARENTHESES TO BE A SINGLE UNIT, WHICH CAN SUCCEED OR FAIL.

! THIS SYMBOL IS A LOGICAL "OR" SYMBOL. IT IMPLIES THE EXISTENCE OF MORE THAN ONE ALTERNATIVE, ANY ONE OF WHICH IS SUFFICIENT TO CAUSE THE LARGE UNIT (RULE OR PARENTHETICAL EXPRESSION) TO BE CONSIDERED "SUCCESSFUL". THUS,

(... ! ... ! ...)

INDICATES THE EXISTENCE OF THREE ALTERNATIVES. THE ELEMENTS OF THE FIRST ALTERNATIVE ARE APPLIED UNTIL A "SYNTACTIC" ELEMENT (ONE WHOSE PRIMITIVE DEFINITION SETS THE FCF_PARSING_TRUE_FALSE_INDICATOR) IS FOUND. IF THAT ELEMENT IS "SUCCESSFUL", THE ALTERNATIVE IS USED AND ANY SUBSEQUENT ELEMENTS OF THAT ALTERNATIVE ARE APPLIED. IF THE ELEMENT FAILS, THE NEXT ALTERNATIVE (AFTER THE OR SYMBOL, "!") IS APPLIED IN THE SAME MANNER. IF ANY ALTERNATIVE IS "SUCCESSFUL", THE OVERALL PARENTHETICAL EXPRESSION IS ALSO "SUCCESSFUL", OTHERWISE NOT.

!! PERFORM STRING CONCATENATION

```
/*********************************************************/
/* SPECIAL SYMBOLS USED AS ARGUMENTS OF PRIMITIVES */
/*********************************************************/
```

* THIS SYMBOL REFERS TO FCF_PREVIOUS_SYMBOL, THE FCF SYMBOL LAST PARSED BY THE PRECOMPILER.

** THIS SYMBOL REFERS TO TEMPORARY_STRING, A UTILITY VARIABLE WHICH MAY CONTAIN A CHARACTER STRING OF ARBITRARY LENGTH AND CONTENT.

```
                    /*******************/
                    /* INITIALIZATION */
                    /*******************/


.PREPROCESS_INPUT_FILE:
      /*****************************************************************/
      /* THIS PROCESS HAS NO LOGICAL FUNCTION, BUT IS INCLUDED HERE  */
      /* IN ORDER TO CALL ATTENTION TO THE PROBABLE EXISTENCE OF AN  */
      /* INPUT FILE PREPROCESSING OPERATION WHICH:                   */
      /*       (A) CONVERTS THE INPUT INFORMATION FROM SEQUENTIAL TO  */
      /*           LIST FORM, OR CONSTRUCTS AN INTERNAL (HIERARCHIC)  */
      /*           MAP OF THE SEQUENTIAL INPUT INFORMATION, AND       */
      /*       (B) SCANS ANY FCF INFORMATION IN THE INPUT FILE IN     */
      /*           ORDER TO CONSTRUCT A RULE NAME TABLE.              */
      /* THESE OPERATIONS ARE THOUGHT TO BE APPROPRIATE FOR REASONS  */
      /* OF EFFICIENCY, BUT DO NOT CAUSE DATA TRANSFORMATIONS WHICH  */
      /* IMPACT THE FLTOPS FUNCTIONAL DEFINITION.                    */
      /*****************************************************************/
      END .PREPROCESS_INPUT_FILE;



               /*************************************/
               /* FCF TOKEN RECOGNITION OPERATIONS */
               /*************************************/


.IDENTIFIER:
      /*****************************************************************/
      /* ATTEMPT TO RECOGNIZE NEXT FCF TOKEN AS IDENTIFIER.          */
      /*****************************************************************/
      IF FCF_SYMBOL_CLASS RETURNED ON PREVIOUS INVOCATION OF FCF LEXICAL
                                    ANALYZER IS 'IDENTIFIER'
         THEN DO;
            CALL FCF_LEXICAL_ANALYZER;
            SET FCF_PARSING_TRUE_FALSE_INDICATOR TO 'TRUE';
            END;
         ELSE SET FCF_PARSING_TRUE_FALSE_INDICATOR TO 'FALSE';
      END .IDENTIFIER;

.NUMBER:
      /*****************************************************************/
      /* ATTEMPT TO RECOGNIZE NEXT FCF TOKEN AS NUMBER.              */
      /*****************************************************************/
      IF FCF_SYMBOL_CLASS RETURNED ON PREVIOUS INVOCATION OF FCF LEXICAL
                                    ANALYZER IS 'NUMBER'
         THEN DO;
            CALL FCF_LEXICAL_ANALYZER;
            SET FCF_PARSING_TRUE_FALSE_INDICATOR TO 'TRUE';
            END;
         ELSE SET FCF_PARSING_TRUE_FALSE_INDICATOR TO 'FALSE';
      END .NUMBER;
```

```
.STRING:
      /*****************************************************************/
      /* ATTEMPT TO RECOGNIZE NEXT FCF TOKEN AS CHARACTER STRING       */
      /* DELIMITED BY SINGLE QUOTATION MARKS.                          */
      /*****************************************************************/
      IF FCF_SYMBOL_CLASS RETURNED ON PREVIOUS INVOCATION OF FCF LEXICAL
                                    ANALYZER IS 'STRING'
         THEN DO;
            CALL FCF_LEXICAL_ANALYZER;
            SET FCF_PARSING_TRUE_FALSE_INDICATOR TO 'TRUE';
            END;
         ELSE SET FCF_PARSING_TRUE_FALSE_INDICATOR TO 'FALSE';
      END .STRING;


"..."          /* ANY STRING LITERAL, BRACKETED BY QUOTATION MARKS */
      /*****************************************************************/
      /* ATTEMPT TO RECOGNIZE NEXT FCF TOKEN AS SPECIFIED LITERAL      */
      /* STRING (E.G. "EXAMPLE").                                      */
      /*****************************************************************/
      IF FCF_SYMBOL RETURNED ON PREVIOUS INVOCATION OF FCF LEXICAL ANALYZER IS
                           IDENTICAL TO LITERAL STRING
         THEN DO;
            CALL FCF_LEXICAL_ANALYZER;
            SET FCF_PARSING_TRUE_FALSE_INDICATOR TO 'TRUE';
            END;
         ELSE SET FCF_PARSING_TRUE_FALSE_INDICATOR TO 'FALSE';
      END "...";

.LABEL:
      /*****************************************************************/
      /* ATTEMPT TO RECOGNIZE NEXT FCF TOKEN AS SYMBOL FOR ABSTRACT    */
      /* FORTRAN STATEMENT LABEL (E.G., $1).                           */
      /*****************************************************************/
      IF FCF_SYMBOL_CLASS RETURNED ON PREVIOUS INVOCATION OF FCF LEXICAL
                              ANALYZER IS 'LABEL'
         THEN DO;
            CALL FCF_LEXICAL_ANALYZER;
            SET FCF_PARSING_TRUE_FALSE_INDICATOR TO 'TRUE';
            END;
         ELSE SET FCF_PARSING_TRUE_FALSE_INDICATOR TO 'FALSE';
      END .LABEL;
```



```
.ADJUST_COLUMN (COLUMN_NUMBER):
      /*****************************************************************/
```

```
.PEEK (ARGUMENT)
      /*****************************************************************/
      /* LOOK AHEAD TO  SEE IF NEXT TOKEN PARSED WILL BE IDENTICAL TO */
      /* THE LITERAL STRING PASSED AS THE ARGUMENT.  NOTE:  THERE MAY */
      /* BE ONE OR MORE ITEMS IN THE ARGUMENT LIST SEPARATED BY OR    */
      /* (!) SYMBOLS                                                  */
      /*****************************************************************/
      COMPARE EACH TOKEN IN ARGUMENT LIST WITH FCF_SYMBOL;
      IF ANY ONE IS EQUIVALENT TO FCF_SYMBOL THEN
           SET FCF_PARSING_TRUE_FALSE_INDICATOR TO TRUE ;
      ELSE   SET FCF_PARSING_TRUE_FALSE_INDICATOR TO FALSE ;
      END  .PEEK


.FORTRAN:          /* ANY COMPLETE FORTRAN STATEMENT */
      /*****************************************************************/
      /* ATTEMPT TO RECOGNIZE NEXT FCF TOKEN AS A FORTRAN STATEMENT. */
      /* FORTRAN STATEMENTS ARE IDENTIFIED BY ABSENCE OF "P" IN       */
      /* COLUMN 1.  IF THERE ARE CONTINUATION CARDS, THE ENTIRE       */
      /* FORTRAN STATEMENT IS CONCATENATED INTO ONE STRING.           */
      /*****************************************************************/
      IF FCF_SYMBOL_CLASS RETURNED ON PREVIOUS INVOCATION OF FCF LEXICAL
                              ANALYZER IS 'FORTRAN'
         THEN DO;
            CALL FCF_LEXICAL_ANALYZER;
            SET FCF_PARSING_TRUE_FALSE_INDICATOR TO 'TRUE';
            END;
         ELSE SET FCF_PARSING_TRUE_FALSE_INDICATOR TO 'FALSE';
      END .FORTRAN;


                  /*****************************/
                  /* SYMBOL TABLE OPERATIONS */
                  /*****************************/

.FIND_TABLE (ARGUMENT):
      /*****************************************************************/
      /* FIND FIRST SYMBOL OF SPECIFIED TABLE.  IF NECESSARY, BUILD   */
      /* NEW TABLE.                                                   */
      /*****************************************************************/
      SEARCH LIST OF SYMBOL TABLE NAMES FOR ONE IDENTICAL TO ARGUMENT;
      IF SEARCH WAS SUCCESSFUL THEN DO
         DO WHILE THIS TABLE POINTS TO AN EQUIVALENCED TABLE;
            UPDATE TABLE POINTER TO THE EQUIVALENCED TABLE VALUE;
            END;
         SET SYMBOL_TABLE_POINTER TO POINT TO FIRST SYMBOL OF THIS TABLE;
         END;
                                    /* NOTE THAT SYMBOL MAY BE NULL */
         ELSE DO;
            ADD NEW TABLE TO LIST, USING NAME PASSED AS ARGUMENT;
            SET SYMBOL_TABLE_POINTER TO POINT TO (NULL) FIRST SYMBOL OF TABLE;
            END;
      SET CURRENT_TABLE_NAME TO ARGUMENT;
      END .FIND_TABLE;
```

```
.FIND_OR_ENTER_SYMBOL (ARGUMENT):
      /*****************************************************************/
      /* SEARCH TABLE FROM CURRENT SYMBOL DOWN, LOOKING FOR            */
      /* SPECIFIED SYMBOL.  IF NOT FOUND, ADD NEW ENTRY.               */
      /*****************************************************************/
      BEGINNING AT CURRENT SYMBOL_TABLE_POINTER LOCATION, SEARCH LIST OF SYMBOL
                                 NAMES FOR NAME IDENTICAL TO ARGUMENT;
      IF SEARCH WAS SUCCESSFUL
         THEN SET SYMBOL_TABLE_POINTER TO POINT TO FOUND SYMBOL;
         ELSE DO;
            ADD NEW SYMBOL TO LIST, USING NAME PASSED AS ARGUMENT;
            SET SYMBOL_TABLE_POINTER TO POINT TO NEWLY ADDED SYMBOL;
            END;
      END .FIND_OR_ENTER_SYMBOL;


.TEST_FOR_SYMBOL (ARGUMENT):
      /*****************************************************************/
      /* SEARCH TABLE FROM CURRENT SYMBOL DOWN, LOOKING FOR            */
      /* SPECIFIED SYMBOL.  IF NOT FOUND, INDICATE FAILURE.            */
      /*****************************************************************/
      BEGINNING AT CURRENT SYMBOL_TABLE_POINTER LOCATION, SEARCH LIST OF SYMBOL
                                 NAMES FOR NAME IDENTICAL TO ARGUMENT;
      IF SEARCH WAS SUCCESSFUL
         THEN DO;
            SET SYMBOL_TABLE_POINTER TO POINT TO FOUND SYMBOL;
            SET OPERATION_TRUE_FALSE_INDICATOR TO 'TRUE';
            END;
         ELSE SET OPERATION_TRUE_FALSE_INDICATOR TO 'FALSE';
      END .TEST_FOR_SYMBOL;

.ENTER_VALUE (ARGUMENT):
      /*****************************************************************/
      /* ENTER SPECIFIED VALUE AT CURRENT SYMBOL_TABLE_POINTER         */
      /* LOCATION.                                                     */
      /*****************************************************************/
      CONSIDER SYMBOL ENTRY TO WHICH SYMBOL_TABLE_POINTER CURRENTLY POINTS;
      REPLACE VALUE FIELD OF THIS SYMBOL ENTRY WITH STRING PASSED AS ARGUMENT;
      END .ENTER_VALUE;
```



```
.MESSAGE (MESSAGE, SPECIAL_INSTRUCTIONS):
      /*****************************************************************/
      /* WRITE ERROR MESSAGE CONTAINED IN "MESSAGE" ARGUMENT, USING    */
```

```
.GET_VALUE (ARGUMENT) :
      /*****************************************************************/
      /* STORE IN THE PLACE INDICATED BY THE ARGUMENT THE VALUE     */
      /* STRING ASSOCIATED WITH SYMBOL TABLE ENTRY IN THE LOCATION */
      /* CURRENTLY POINTED TO BY SYMBOL_TABLE_POINTER.              */
      /*****************************************************************/
      CONSIDER SYMBOL ENTRY TO WHICH SYMBOL_TABLE_POINTER CURRENTLY POINTS;
      IF THERE EXISTS A VALUE ENTRY FOR THIS SYMBOL
         THEN COPY VALUE FIELD INTO LOCATION INDICATED BY ARGUMENT;
         ELSE ENTER NULL STRING INTO LOCATION INDICATED BY ARGUMENT;
      END .GET_VALUE;


.CATENATE_VALUE (ARGUMENT):
      /*****************************************************************/
      /* APPEND SPECIFIED STRING ONTO VALUE FIELD OF SYMBOL TABLE    */
      /* ENTRY POINTED TO BY SYMBOL_TABLE_POINTER.                   */
      /*****************************************************************/
      CONSIDER SYMBOL ENTRY TO WHICH SYMBOL_TABLE_POINTER CURRENTLY POINTS;
      CATENATE STRING PASSED AS ARGUMENT ONTO END OF VALUE FIELD OF THIS ENTRY;
      END .CATENATE_VALUE;


.REPLACE_STRING (STRING1,STRING2,NUMBER)
      /*********************************************************/
      /* REPLACES NUMBER OCCURRENCES OF STRING1 WITHIN       */
      /* ANOTHER STRING WITH STRING2                         */
      /*********************************************************/
      GET VALUE POINTED AT BY SYMBOL_TABLE_POINTER ;
      SET I = 0;
      DO WHILE I < NUMBER;
         INCREMENT I;
         IF STRING1 IS NULL THEN CATENATE STRING2 ONTO BEGINNING OF VALUE;
         ELSE DO;
            SCAN VALUE TO FIND FIRST OCCURRENCE OF STRING1;
            IF MATCH OCCURS
               THEN IF STRING2 IS NULL
                       THEN DELETE STRING1 FROM VALUE;
                       ELSE REPLACE STRING1 WITH STRING2;
            ELSE SET I = NUMBER;
            END;
         END;
      ENTER VALUE INTO LOCATION POINTED AT BY SYMBOL_TABLE_POINTER;
      END .REPLACE_STRING;


.INDEX (STRING,SUBSTRING,START_POS) :
      /*****************************************************************/
      /*  DETERMINE STARTING POSITION OF SUBSTRING WITHIN A STRING   */
      /*****************************************************************/
      SCAN STRING FOR THE FIRST OCCURRENCE OF SUBSTRING;
      IF SUBSTRING WAS FOUND IN STRING THEN SET START_POS TO THE
           CHARACTER POSITION IN STRING WHERE SUBSTRING BEGINS;
      ELSE SET START_POS TO ZERO;
      END .INDEX;
```

```
.LENGTH (STRING,STRING_LENGTH) :
      /*****************************************************************/
      /* DETERMINE THE NUMBERS OF CHARACTERS IN A STRING               */
      /*****************************************************************/
      COUNT THE NUMBER OF CHARACTERS IN STRING AND RETURN THE RESULT IN
           STRING_LENGTH;
      END .LENGTH;


.DELETE_ENTRY:
      /*****************************************************************/
      /* DELETE ENTIRE ENTRY AT CURRENT SYMBOL_TABLE_POINTER           */
      /* LOCATION.                                                     */
      /*****************************************************************/
      DELETE SYMBOL ENTRY TO WHICH SYMBOL_TABLE_POINTER CURRENTLY POINTS;
      UPDATE POINTERS TO MAINTAIN TABLE INTEGRITY;
      END .DELETE_ENTRY;


.SAVE_POINTER:
      /*****************************************************************/
      /* STORE CURRENT SYMBOL_TABLE_POINTER LOCATION FOR LATER USE.    */
      /*****************************************************************/
      PUSH VALUE OF SYMBOL_TABLE_POINTER ONTO TOP OF SAVED_POINTER_STACK;
      END .SAVE_POINTER;


.RESTORE_POINTER:
      /*****************************************************************/
      /* RESTORE PREVIOUS VALUE OF SYMBOL_TABLE_POINTER LOCATION.      */
      /*****************************************************************/
      REMOVE TOP VALUE IN SAVED_POINTER_STACK;
      SET SYMBOL_TABLE_POINTER TO THIS VALUE;
      END .RESTORE_POINTER;
```

```
.SUBSTRING (STRING_IN,STRING_OUT,START_POS,LENGTH):
      /*****************************************************************/
      /* PERFORMS SUBSTRING EXTRACTION FROM STRING_IN TO STRING_OUT.    */
      /* SUBSTRING BEGINS AT CHARACTER POSITION START_POS AND CONTAINS  */
      /* A MAXIMUM OF LENGTH CHARACTERS. IF LENGTH PARAMETER IS NOT     */
      /* PRESENT, SUBSTRING CONTAINS ALL CHARACTERS TO RIGHT OF START_POS */
      /*****************************************************************/
      SET STRING_OUT = NULL;
      IF OPTIONAL LENGTH PARAMETER IS IN CALLING SEQUENCE
         THEN DO;
            IF LENGTH > 0
               THEN DO;
                  STARTING WITH CHARACTER POSITION START_POS;
                  COPY A MAXIMUM OF LENGTH CHARACTERS FROM STRING_IN TO STRING_OUT;
                    END;
               ELSE RETURN;
         ELSE DO;
            STARTING WITH CHARACTER POSITION START_POS;
            COPY ALL CHARACTERS TO THE RIGHT FROM STRING_IN TO STRING_OUT;
            END;
      END .SUBSTRING;

.FIND_BY_SUBSCRIPT (ARGUMENT):
      /*************************************************************/
      /* PERFORMS ABSOLUTE POINTER POSITIONING WITHIN A TABLE */
      /*************************************************************/
      LOCATE THE TABLE VALUE INDICATED BY THE NUMBER IN ARGUMENT;
      SET THE SYMBOL_TABLE_POINTER TO POINT AT THIS VALUE;
      END .FIND_BY_SUBSCRIPT

.COMPUTE_TABLE_POSITION:
      /*****************************************************************/
      /* USES STANDARD FORTRAN ARRAY POSITION COMPUTATIONS TO COMPUTE THE */
      /* POSITION OF A SPECIFIED ARRAY ENTRY.  THE TABLE_HEADER INCLUDES  */
      /* INFORMATION ON THE SIZE OF TABLE SUBSCRIPTS AS PROVIDED BY THE   */
      /* CORRESPONDING DEFINE STATEMENT.                                  */
      /*****************************************************************/
      SET SUBSCRIPT_NUMBER = STACK_SIZE-1;
      SET ARRAY_POSITION = 0
      DO WHILE SUBSCRIPT_NUMBER >= 0 ;
         POP TOP ELEMENT FROM ARRAY_STACK AND SAVE IN TEMPORARY_VALUE;
         IF SUBSCRIPT_NUMBER = 0
            THEN ADD TEMPORARY_VALUE TO ARRAY_POSITION;
         ELSE DO;
             RETRIEVE VALUE FOR SUBSCRIPT_NUMBER FROM TABLE_HEADER
                       OF CURRENT_TABLE_NAME;
             SAVE THIS VALUE IN SUBSCRIPT_VALUE;
             SET ARRAY_POSITION = SUBSCRIPT_VALUE * (ARRAY_POSITION+
                                  TEMPORARY_VALUE-1);
             END;
         DECREMENT SUBSCRIPT_NUMBER BY 1;
         END;
      END .COMPUTE_TABLE_POSITION;
```

```
.ARRAY_SETUP (TABLE_NAME)
      /*****************************************************************/
      /* PERFORMS INITIALIZATION FOR TABLES REFERENCED BY SUBSCRIPT */
      /*****************************************************************/
      .FIND_TABLE(TABLE_NAME)
      SAVE ARRAY_STACK CONTENTS IN TABLE_HEADER OF TABLE TABLE_NAME;
      DELETE ALL ENTRIES ALREADY IN TABLE;
      .END ARRAY_SETUP;


.NULL_OUT_TABLE (NAME);
      /*****************************************************************/
      /* REMOVE ANY TABLE EQUIVALENCE AND DIMENSIONS FOR TABLE NAME  */
      /*****************************************************************/
      IF SYMBOL TABLE NAME IS NON-EMPTW THEN DO;
         WRITE OUT WARNING MESSAGE INDICATING THE TABLE IS BEING CLEARED;
         CLEAR TABLE OF ALL ENTRIES;
         END;
      IF TABLE NAME IS EQUIVALENCED TO ABOTHER TABLE THEN REMOVE THE
           EQUIVALENCE;
      IF TABLE NAME IS DIMENSIONED THEN REMOVE ITS DIMENSIONS;
      END .NULL_OUT_TABLE;


.SETUP_TABLE_EQUIVALENCE:
      /*****************************************************************/
      /* EQUIVALENCE ONE TABLE NAME WITH ANOTHER SO THAT EITHER NAME */
      /* REFERENCES THE SAME GROUP OF SYMBOL TABLE ENTRIES           */
      /*****************************************************************/
      IF TABLE IDENTIFED BY DEFINE_TABLE_NAME ALREADY EXISTS THEN
         IF IT HAS ENTRIES OF ITS OWN (I.E. IT IS A REAL TABLE AND NOT
              PREVIOUSLY EQUIVALENCED TO ANOTHER TABLE ) THEN WRITE AN
              ERROR MESSAGE INDICATING THAT THIS IS AN ILLEGAL EQUIVALENCE;
      IF EQUIVALENCED_TABLE_NAME HAS BEEN PREVIOUSLY EQUIVALENCED TO
           DEFINE_TABLE_NAME THEN WRITE OUT SEVERE ERROR MESSAGE INDICATING
           THE LINK;
      ELSE SET UP LINK THAT INSURES A REFERENCE TO DEFINE_TABLE_NAME POINTS
           TO EQUIVALENCED_TABLE_NAME;
      END .SETUP_TABLE_EQUIVALENCE;
```

```
        /*****************************************************/
        /* INPUT FILE LEXICAL ANALYZER CONTROL OPERATIONS */
        /*****************************************************/


.HOME_INPUT_POINTER:
      /*******************************************************************/
      /* DIRECT INPUT LEXICAL ANALYZER TO FIRST TOP-LEVEL NODE OF     */
      /* INPUT FILE.                                                     */
      /*******************************************************************/
      MOVE INPUT_PARSING_POINTER TO BEGINNING OF FIRST NODE OF INPUT TREE;
      IF SUCH A NODE WAS FOUND
         THEN DO;
            SET OPERATION_TRUE_FALSE_INDICATOR TO TRUE;
            SET INPUT_SYMBOL TO NULL;
            CALL INPUT_LEXICAL_ANALYZER;
            END;
         ELSE DO;
            SET OPERATION_TRUE_FALSE_INDICATOR = "FALSE"
            .MESSAGE(I,30)
            END;
      END .HOME_INPUT_POINTER;


.ADVANCE_INPUT_POINTER:
      /*******************************************************************/
      /* DIRECT INPUT LEXICAL ANALYZER TO NEXT SAME-LEVEL NODE OF     */
      /* INPUT FILE.                                                     */
      /*******************************************************************/
      IF OPERATION_TRUE_FALSE_INDICATOR = "FALSE" THEN RETURN;
      FIND NEXT NODE IN INPUT TREE WHICH IS AT SAME LEVEL AS CURRENT NODE;
      IF SUCH A NODE WAS FOUND
         THEN DO;
            SET OPERATION_TRUE_FALSE_INDICATOR TO "TRUE";
            MOVE INPUT_PARSING_POINTER TO BEGINNING OF THAT NODE;
            SET INPUT_SYMBOL TO NULL;
            CALL INPUT_LEXICAL_ANALYZER;
            END;
         ELSE DO;
            SET OPERATION_TRUE_FALSE_INDICATOR TO "FALSE";
            SET BROTHER_NODE_END = "TRUE";
            END;
      END .ADVANCE_INPUT_POINTER;


.MOVE_INPUT_POINTER_DOWN:
      /*******************************************************************/
      /* DIRECT INPUT LEXICAL ANALYZER TO FIRST NEXT-LEVEL-DOWN NODE */
      /* OF INPUT FILE.                                                  */
      /*******************************************************************/
      IF OPERATION_TRUE_FALSE_INDICATOR = "FALSE" THEN RETURN;
      FIND FIRST NODE BELOW CURRENT NODE IN INPUT TREE;
      IF SUCH A NODE WAS FOUND
         THEN DO;
            SET OPERATION_TRUE_FALSE_INDICATOR TO "TRUE";
            MOVE INPUT_PARSING_POINTER TO BEGINNING OF THAT NODE;
            SET INPUT_SYMBOL TO NULL;
            CALL INPUT_LEXICAL_ANALYZER;
            END;
         ELSE SET OPERATION_TRUE_FALSE_INDICATOR TO "FALSE";
      END .MOVE_INPUT_POINTER_DOWN;
```

```
.PARSE_INPUT (ARGUMENT):
      /*****************************************************************/
      /* ATTEMPT TO RECOGNIZE NEXT INPUT TOKEN AS INDICATED IN         */
      /* ARGUMENT.  FOR SIMPLICITY CLASS NAMES AND LITERAL STRINGS     */
      /* ARE PASSED INDISCRIMINATELY AS ARGUMENTS.  THE IMPLEMENTOR    */
      /* MUST INSURE THAT THIS CONFLICT IS AVOIDED                     */
      /*****************************************************************/
      CONSIDER INPUT_SYMBOL, INPUT_SYMBOL_CLASS RETURNED ON PREVIOUS INVOCATION
                             OF INPUT_LEXICAL_ANALYZER;
      IF ARGUMENT = 'IDENTIFIER' & INPUT_SYMBOL_CLASS = 'IDENTIFIER'
         ! ARGUMENT = 'NUMBER' & INPUT_SYMBOL_CLASS = 'NUMBER'
         ! ARGUMENT = 'STRING' & INPUT_SYMBOL_CLASS = 'STRING'
         ! ARGUMENT = INPUT_SYMBOL
         THEN DO;
            SET OPERATION_TRUE_FALSE_INDICATOR TO 'TRUE';
            CALL INPUT_LEXICAL_ANALYZER;
            END;
         ELSE SET OPERATION_TRUE_FALSE_INDICATOR TO 'FALSE';
      END .PARSE_INPUT;



            /*************************************/
            /* FLTOPS FUNCTION CONTROL OPERATIONS */
            /*************************************/


.SET (ARGUMENT):
      /*****************************************************************/
      /* SET SWITCH OR FLAG, AS INDICATED.                             */
      /*****************************************************************/
      PERFORM THE ARITHMETIC ASSIGNMENT CONTAINED IN THE ARGUMENT;
      END .SET;


.TEST (ARGUMENT):
      /*****************************************************************/
      /* TEST SWITCH OR FLAG FOR INDICATED VALUE OR CONDITION.         */
      /*****************************************************************/
      TEST FOR THE TRUTH OR FALSEHOOD OF THE BOOLEAN EXPRESSION CONTAINED IN
                              THE ARGUMENT;
      IF TRUE
         THEN SET FCF_PARSING_TRUE_FALSE_INDICATOR TO "TRUE";
         ELSE SET FCF_PARSING_TRUE_FALSE_INDICATOR TO "FALSE";
      END .TEST;
```

```
.SCAN_TO_LOCATION (ARGUMENT):
      /*****************************************************************/
      /* DISABLE ALL FLTOPS OPERATIONS EXCEPT BASIC FCF PARSING,       */
      /* UNTIL CORRESPONDING LOCATION (IDENTIFIABLE BY NUMERICAL       */
      /* ARGUMENT) OF SAME INVOCATION OF SAME RULE IS REACHED.         */
      /*****************************************************************/
      IF FLTOPS FUNCTION = SYNTAX OR SCAN THEN RETURN;
         ELSE DO;
            SAVE FLTOPS FUNCTION VALUE AND RESET FUNCTION TO SCAN;
            SAVE INFORMATION INDICATING THE RULE AND LOCATION AT WHICH
               FLTOPS FUNCTIONS WILL RESUME IN LOCATION_INFORMATION;
            END;
      END .SCAN_TO_LOCATION;


.LOCATION (ARGUMENT):
      /*****************************************************************/
      /* SEE .SCAN_TO_LOCATION.                                        */
      /*****************************************************************/
      IF FLTOPS FUNCTION NOT EQUAL TO SCAN THEN RETURN;
      IF CURRENT LOCATION CORRESPONDS TO THAT INDICATED IN LOCATION_INFORMATION
               SAVED BY THE LAST INVOKED .SCAN_TO_LOCATION PRIMITIVE
            THEN RESET FLTOPS FUNCTION TO SAVED VALUE;
      ELSE RETURN;
      END .LOCATION;


.RECOVERY_POINT (ARGUMENT):
      /*****************************************************************/
      /* STORE CURRENT FCF_PARSING_POINTER LOCATION.                   */
      /*****************************************************************/
      PUSH FCF_PARSING_POINTER ONTO RECOVERY_POINT_STACK;
      END .RECOVERY_POINT;


.RETURN_TO_RECOVERY_POINT (ARGUMENT):
      /*****************************************************************/
      /* REINITIATE PARSING AT CORRESPONDING RECOVERY POINT            */
      /* (IDENTIFIABLE BY NUMERICAL ARGUMENT) OF SAME INVOCATION OF    */
      /* SAME RULE.                                                    */
      /*****************************************************************/
      POP TOP ELEMENT FROM RECOVERY_POINT_STACK;
      SET FCF_PARSING_POINTER LOCATION TO THIS VALUE;
      SET FCF_SYMBOL TO NULL;
      CALL FCF_LEXICAL_ANALYZER;
      END .RETURN_TO_RECOVERY_POINT;
```

```
                    /*********************/
                    /* INVOKE, EXECUTE */
                    /*********************/


.INVOKE (ARGUMENT):                    /* ARGUMENT IS FCF RULE NAME */
      /*****************************************************************/
      /* INVOKE INDICATED FCF RULE.                                    */
      /*****************************************************************/
      PUSH RULE NAME (ARGUMENT) ONTO PROCEDURE_NAME STACK;
      IF TRACE NOT EQUAL "OFF"
         THEN PRINT RULE ENTRY TRACE STATEMENT;
      IF TRACE NOT EQUAL "OFF" AND NO SEVERE ERRORS HAVE BEEN ENCOUNTERED
         THEN OUTPUT RULE INVOCATION FORTRAN COMMENT;
      IF RULE NAMED IN ARGUMENT EXISTS IN FCF PORTION OF INPUT FILE
         THEN CONSIDER THAT RULE;
      ELSE IF RULE NAMED IN ARGUMENT EXISTS IN FCF
         THEN CONSIDER THAT RULE;
      IF RULE WAS FOUND
         THEN DO;
            SET FCF_PARSING_POINTER TO FIRST ELEMENT AFTER RULE LABEL;
            SET FCF_SYMBOL TO NULL;
            CALL FCF_LEXICAL_ANALYZER;
            SET OPERATION_TRUE_FALSE_INDICATOR TO "TRUE";
            END;
         ELSE SET OPERATION_TRUE_FALSE_INDICATOR TO "FALSE";
      END .INVOKE;


.END_PROCEDURE (ARGUMENT):
      /*****************************************************************/
      /* RETURN FROM INVOKED FCF RULE.                                 */
      /*****************************************************************/
      VERIFY THAT ARGUMENT MATCHES TOP ELEMENT OF PROCEDURE_NAME STACK;
      IF TRACE NOT EQUAL "OFF"
         THEN PRINT RULE EXIT TRACE STATEMENT;
      POP TOP ELEMENT FROM PROCEDURE_NAME_STACK;
      END .END_PROCEDURE;


.EXECUTE (ARGUMENT):
      /*****************************************************************/
      /* EXECUTE INDICATED FORTRAN SUBROUTINE AT PRECOMPILER TIME.     */
      /*****************************************************************/
      TRANSFER CONTROL TO THE ROUTINE NAMED AS THE ARGUMENT;
      DATA COMMUNICATIONS ARE MAINTAINED THROUGH THE EXECUTE_DATA_AREA;
      END .EXECUTE;
```

```
.EXECUTE_DATA_AREA_UPDATE:
      /*********************************************/
      /* DATA IS MOVED FROM SYMBOL TABLE LOCATIONS  */
      /* TO THE EXECUTE_DATA_AREA OR VICE VERSA.    */
      /* FORMAT OF DATA IS MAINTAINED IN A FORM     */
      /* ACCEPTABLE FOR THE EXECUTING PROGRAM IN THE*/
      /* EXECUTE_DATA_AREA AND IN STRING FORMAT IN  */
      /* THE SYMBOL TABLES                          */
      /*********************************************/
      IF CONVERT_FLAG SET TO "IN"
         THEN DO;
            SET ORIGIN TO PARAMETER_LOCATION;
            SET DESTINATION TO EXECUTE_DATA_AREA_POINTER;
            END;
         ELSE DO;
            SET ORIGIN TO EXECUTE_DATA_AREA_POINTER;
            SET DESTINATION TO PARAMETER_LOCATION;
            END;
      DO FOR EACH OF THE PARAMETER_SIZE VALUES;
         MOVE VALUE FROM ORIGIN TO DESTINATION;
         CONVERT VALUE ACCORDING THE THE SPECIFICATION FOUND IN
              PARAMETER_CLASS AND CONVERT_FLAG;
    /* CONVERT_FLAG = IN    CHANGE VALUE TO FORM USEABLE BY EXECUTING PROGRAM */
    /* CONVERT_FLAG = OUT   CHANGE VALUE TO FORM OF LITERAL STRING FOR        */
    /*                        INSERTION INTO SYMBOL TABLES                    */
         INCREMENT ORIGIN AND DESTINATION;
         END;
      END .EXECUTE_DATA_AREA_UPDATE;


.EXECUTE_PARAMETER_UPDATE:
      /*****************************************/
      /* UPDATES SYMBOL TABLE LOCATIONS AFTER */
      /* COMPLETION OF AN EXECUTE ROUTINE     */
      /*****************************************/
      SET EXECUTE_DATA_AREA_POINTER TO FIRST VALUE TO BE TRANSFERRED;
      DO FOR EACH PARAMETER IN TABLE EXECUTE_ROUTINE_NAME;
         GET VALUE FOR PARAMETER_CLASS, PARAMETER_SIZE AND PARAMETER_LOCATION;
         .EXECUTE_DATA_AREA_UPDATE
         UPDATE EXECUTE_DATA_AREA_POINTER;
         END;
      END .EXECUTE_PARAMETER_UPDATE;
```

```
                    FLTOPS PRIMITIVES

                    /************************/
                    /* BOOLEAN OPERATIONS */
                    /************************/


.AND:
      /*****************************************************************/
      /* TAKE LOGICAL AND OF TWO BOOLEAN ELEMENTS                      */
      /*****************************************************************/
      REMOVE TOP TWO ELEMENTS OF THE BOOLEAN_TRUTH_STACK;
      TAKE LOGICAL AND OF THESE TWO ELEMENTS ;
      PUSH RESULTING VALUE ONTO THE TOP OF THE BOOLEAN_TRUTH_STACK ;
      END .AND;


.OR:
      /*****************************************************************/
      /* COMPUTE LOGICAL OR OF TWO BOOLEAN ELEMENTS                    */
      /*****************************************************************/
      REMOVE TOP TWO ELEMENTS OF THE BOOLEAN_TRUTH_STACK;
      TAKE THE LOGICAL OR THE THESE TWO ELEMENTS ;
      PUSH THE RESULTING VALUE ONTO THE TOP OF THE BOOLEAN_TRUTH_STACK ;
      END .OR;


.NOT:
      /*****************************************************************/
      /* COMPUTE THE COMPLEMENT OF A BOOLEAN ELEMENT                   */
      /*****************************************************************/
      REMOVE TOP ELEMENT OF THE BOOLEAN_TRUTH_STACK ;
      TAKE LOGICAL COMPLEMENT OF THIS ELEMENT ;
      PUSH RESULTING ELEMENT ONTO THE TOP OF THE BOOLEAN_TRUTH_STACK ;
      END .NOT;


.SAVE_TRUTH:
      /*****************************************************************/
      /* SAVE CURRENT VALUE OF OPERATION_TRUE_FALSE_INDICATOR          */
      /* ON THE BOOLEAN_TRUTH_STACK.                                   */
      /*****************************************************************/
      PUSH VALUE OF OPERATION_TRUE_FALSE_INDICATOR ONTO THE TOP OF
               THE BOOLEAN_TRUTH_STACK ;
      END .SAVE_TRUTH;


.BOOLEAN_COMPARE:
      /*****************************************************************/
      /*  EVALUATES RELATIONAL EXPRESSIONS TO DETERMINE TRUE/FALSE     */
      /*****************************************************************/
      CONSIDER THE TYPE OF RELATION SAVED IN THE RELATION_TYPE_FLAG ;
      COMPARE VALUE IN SAVED_VALUE WITH THE VALUE IN COMPUTED_VALUE;
      IF THE RELATION IS TRUE THEN SET OPERATION_TRUE_FALSE_INDICATOR TO TRUE;
         ELSE SET OPERATION_TRUE_FALSE_INDICATOR TO FALSE ;
      END  .BOOLEAN_COMPARE;
```

```
                    /*******************/
                    /* FORTRAN OUTPUT */
                    /*******************/

.OUTPUT (ARGUMENT):
      /*****************************************************************/
      /* OUTPUT FORTRAN STATEMENT WHICH HAS BEEN READ FROM FCF (OR    */
      /* FROM FCF PORTION OF INPUT FILE), SUBSTITUTING REPLACEMENT    */
      /* STRINGS AS INDICATED BY APPROPRIATE TABLE.                   */
      /*****************************************************************/
      IF FLTOPS FUNCTION NOT EQUAL TO FULL THEN RETURN;
      PLACE FORTRAN_PARSING_POINTER ON FIRST CHARACTER OF ARGUMENT;
      CALL FORTRAN_LEXICAL_ANALYZER;
      .SAVE_POINTER;
      IF PASS1_PASS2_SWITCH = 1
         THEN OUTPUT_FILE = PASS1_OUTPUT_FILE;
         ELSE OUTPUT_FILE = PASS2_OUTPUT_FILE;
      DO WHILE (FORTRAN_SYMBOL IS NOT "END OF FILE");
    /* END OF FILE IN THIS CONTEXT REFERS TO LOGICAL END OF ARGUMENT STRING */
         IF FORTRAN_SYMBOL = STATEMENT_TERMINATOR
            THEN DO;
               WRITE OUTPUT_BUFFER TO OUTPUT_FILE ;
                IF FLTOPS OUTPUT = FORTRAN
                   THEN DO;
                      MAKE SURE CORRECT FORTRAN CARD IMAGE SYNTAX IS PROVIDED;
                      PROVIDE CONTINUATION STATEMENTS AND SEQUENCE NUMBERS
                          IF NECESSARY;
                        END;
            CLEAR OUTPUT_BUFFER;
            END;
         IF PASS1_PASS2_SWITCH = 1
            THEN .FIND_TABLE(PASS1);
            ELSE .FIND_TABLE(PASS2);
         .TEST_FOR_SYMBOL(FORTRAN_SYMBOL);
         IF OPERATION_TRUE_FALSE_INDICATOR = "TRUE"
            THEN WRITE VALUE ASSOCIATED WITH CURRENT SYMBOL_TABLE_POINTER
                                        TO OUTPUT_BUFFER;
            ELSE WRITE FORTRAN_SYMBOL TO OUTPUT_BUFFER;
         CALL FORTRAN_LEXICAL_ANALYZER;
      END;
      .RESTORE_POINTER;
      IF TRACE = "HIGH" OR "HIGHER"
         THEN WRITE FORTRAN OUTPUT TRACE MESSAGE;
      IF STRING SUBSTITUTION WAS PERFORMED ABOVE
         THEN ADD ENTIRE OUTPUT FORTRAN STATEMENT TO LIST
                 OF ALTERED OR GENERATED STATEMENTS;
      END .OUTPUT;
```

FLTOPS PRIMITIVES

```
.ADJUST_COLUMN (COLUMN_NUMBER):
      /*****************************************************************/
      /* INSURE THAT NEXT ELEMENT OUTPUT BEGINS IN COL COLUMN_NUMBER */
      /*****************************************************************/
      IF LENGTH OF OUTPUT_BUFFER > COLUMN_NUMBER THEN DO;
         OUPUT CURRENT CONTENTS OF OUTPUT_BUFFER;
         SET OUTPUT_BUFFER TO NULL;
         WRITE WARNING MESSAGE INDICATING THE ACTION JUST TAKEN;
         ENDL";
      PAD OUTPUT_BUFFER WITH SUFFICIENT BLANKS SO THAT ITS LENGTJH""H EQUALS
           COLUMN_NUMBER - 1;
      END .ADJUST_COLUMN;


.FORMAT_STRING (STRING_IN,FORMAT):
      /*****************************************************************/
      /* REFORMAT CONTENTS OF STRING_ ACCORDING TO SPECIFICATION IN  */
      /* FORMAT.  RETURN REFORMATED STRING IN TEMPORARY_STRING       */
      /*****************************************************************/
      IF FIRST CHARACTER OF FORMAT IS NOT LEGAL FORMAT TYPE (I.E. AT LEAST
           A,E,F OR I) THEN DO;
         WRITE OUT SEVERE ERROR MESSAGE INDICATING ILLEGAL FORMAT TYPE;
         RETURN;
         END;
      IF THE CONTENTS OF STRING_IN ARE NOT COMPATABLE WITH THE INDICATED
           FORMAT TYPE  THEN DO;
         WRITE OUT SEVERE ERROR INDICATING THE FORMAT/DATA DISCREPENCY;
         RETURN;
         END;
      IF THE FORMAT CONSISTS OF SIMPLY THE FORMAT TYPE DESIGNATOR ( I.E.
           SIMPLY AN I OR F ETC. ) THEN REFORMAT STRING_IN USING THE DEFAULT
           PROCEDURE (IMPLEMENTATION OPTION) CORRESPONDING TO THE FORMAT TYPE;
           ( E.G. FOR AN I FORMAT, THEN VALUE COULD BE ROUNDED, SCALED
             THEN OUTPUT )
      ELSE IF FORMAT IS A LEGAL FORMAT (E.G. F10.4 NOT FZ.T) THEN REFORMAT
           STRING_IN ACCORDINGLY AND PLACE RESULT IN TEMPORARY_STRING;
      ELSE WRITE OUT SEVERE ERROR MESSAGE INDICATING THE BAD FORMAT;
      END .FORMAT_STRING;


          /*********************************/
          /* ERROR AND STATUS OPERATIONS */
          /*********************************/

.ERROR (MESSAGE, SPECIAL_INSTRUCTIONS):
      /*****************************************************************/
      /* WRITE ERROR MESSAGE IF AND ONLY IF IMMEDIATELY PRECEDING    */
      /* SYNTACTIC ELEMENT WAS NOT SATISFIED.                        */
      /*****************************************************************/
      IF FCF_PARSING_TRUE_FALSE_INDICATOR = "FALSE"
         THEN .MESSAGE(TYPE,MESSAGE,SPECIAL_INSTRUCTIONS)
      END .ERROR;
```

FLTOPS PRIMITIVES

```
/*****************************************************************************/
/***************   E R R O R   M E S S A G E   T A B L E   *******************/
/**            ***************************************************************/
/** NUMBER **            ***** ERROR MESSAGE *****                           */
/**            ***************************************************************/
/**   1     FS:UNRECOGNIZED STATEMENT                                         */
/**   2     IS:DESIRED INPUT NOT FOUND                                        */
/**   3     FS:MISSING END FOR MAIN PROCEDURE                                 */
/**   4     FW:DESIRED SYMBOL DOES NOT EXIST IN TABLE NAME SPECIFIED          */
/**   5     FW:MISSING LEFT PARENTHESIS -- ASSUMED PRESENT                    */
/**   6     FW:MISSING RIGHT PARENTHESIS -- ASSUMED PRESENT                   */
/**   7     FW:MISSING COMMA -- ASSUMED PRESENT                               */
/**   8     FW:NON-EMPTY SYMBOL TABLE                                         */
/**   9     GS:ASSERTION FAILED                                               */
/**  10     FS:MISSING OR ILLEGAL PARAMETER TYPE IDENTIFIER                   */
/**  11     FS:MISSING OR ILLEGAL RELATIONAL OPERATOR IN BOOLEAN EXPRESSION   */
/**  12     FW:MISSING OR ILLEGAL RULE NAME ON FCF PROCEDURE                  */
/**  13     FS:MISSING OR ILLEGAL WORD IN A "DO FOR" BLOCK CONSTRUCT          */
/**  14     FS:MISSING OR ILLEGAL ELEMENT IN VALUE STATEMENT                  */
/**  15     FS:UNRECOGNIZED OR INVALID INPUT REQUEST                          */
/**  16     FS:MISSING OR ILLEGAL ELEMENT IN AN ARITHMETIC EXPRESSION         */
/**  17     FW:MISSING ":" ON COMMENT OR ERROR SPECIFICATION, ASSUMED PRESENT*/
/**  18     FS:EXTRA STATEMENTS BEYOND THE EXPECTED END OF FCF PROCEDURE      */
/**  19     FS:MISSING OR ILLEGAL FORMAT ELEMENT IN FORMAT STATEMENT          */
/**  20     FS:MISSING END STATEMENT ON  "DO WHILE"  BLOCK                    */
/**  21     FS:MISSING END STATEMENT ON  "DO"  BLOCK                          */
/**  22     FS:MISSING END STATEMENT ON  "DO FOR SYMBOL TABLE"  BLOCK         */
/**  23     FW:MISSING WORD "SUBNODE" --  ASSUMED PRESENT                     */
/**  24     FS:MISSING END STATEMENT ON "DO FOR EACH SUBNODE OF" BLOCK        */
/**  25     FS:MISSING COMMA -- MUST BE PRESENT                               */
/**  26     FS:ONLY STRING LITERALS ALLOWED IN MESSAGE STATEMENTS             */
/**  27     FS:MISSING ARRAY DEFINITION OR "AS" CLAUSE IN DEFINE STATEMENT    */
/**  28     FS:MISSING RIGHT PARENTHESIS -- MUST BE PRESENT                   */
/**  29     FS:UNRECOGNIZED ELEMENT IN BOOLEAN EXPRESSION --                  */
/**  30     IW:INPUT FILE EMPTY                                               */
/**  31     IW:DESIRED INPUT TREE NOT FOUND                                   */
/**  32     FS:MISSING OR ILLEGAL BOOLEAN EXPRESSION                          */
/**  33     FS:EXPECTED "THEN" CLAUSE IN "IF" STATEMENT NOT FOUND             */
/**  34     FW:MISSING WORD "TABLE" -- ASSUMED PRESENT                        */
/**  35     FS:MISSING OR ILLEGAL SYMBOL TABLE NAME                           */
/**  36     FW:MISSING WORD "OF" -- ASSUMED PRESENT                           */
/**  37     IW:"DO FOR EACH SUBNODE" BLOCK BYPASSED--NO SUBNODES FOUND        */
/**  38     FS:MISSING OR ILLEGAL RULE NAME IN "INVOKE" STATEMENT             */
/**  39     FS:MISSING END STATEMENT ON FCF RULE                              */
/**  40     FS:MISSING OR ILLEGAL SYMBOL NAME                                 */
/**  41     FS:MISSING OR ILLEGAL VALUE FIELD IN MESSAGE STATEMENT            */
/**  42     FS:VALUE FIELD REQUIRED IN "REPLACE" STATEMENT                    */
/**  43     FW:MISSING OR ILLEGAL MESSAGE TYPE   -- TYPE GENERAL ASSUMED      */
/**  44     FS:MISSING OR ILLEGAL ROUTINE NAME FOR "PARAMETERS" OR "EXECUTE"  */
/**  45     FS:MISSING OR ILLEGAL FORM FOR PARAMETER IN "EXECUTE" STATEMENT   */
/**  46     FS:MISSING OR ILLEGAL ELEMENT WITHIN AN "OUTPUT" STATEMENT        */
/**  47     FS:MISSING OR ILLEGAL ARITHMETIC EXPRESSION                       */
/**  48     FS:MISSING OR ILLEGAL ELEMENT IN A SUBSCRIPT SPECIFICATION        */
/**  49     FS:EXTRANNEOUS "ELSE" CLAUSE                                      */
/**  50     FS:EXTRANNEOUS "THEN" CLAUSE                                      */
/**  51     FS:"PARAMETERS" STATEMENT MUST PRECEDE CORPESPONDING "EXECUTE"    */
/**  52     FW:THE RULE REQUESTED WAS NOT FOUND                               */
/**  53     IW:UNUSED INPUT                                                   */
/**  54     FS:MISSING OR ILLEGAL FORM FOR INPUT TREE SEARCH STATEMENT        */
/**  55     FS:HEADING OUTPUT IN "STATUS" STATEMENT MUST BE A STRING LITERAL  */
/**  56      N:INDICATED SYMBOL NOT PRESENT WHEN DELETE ATTEMPTED --          */
/**  57     IS:SPECIFIED INPUT TREE DOES NOT EXIST                            */
/**  58     FS:ILLEGAL FORM FOR STRING IDENTIFIER IN "SUBSTRING" STATEMENT    */
/**            ***************************************************************/
/*****************************************************************************/
```

```
.MESSAGE (MESSAGE, SPECIAL_INSTRUCTIONS):
     /*****************************************************************/
     /* WRITE ERROR MESSAGE CONTAINED IN "MESSAGE" ARGUMENT, USING    */
     /* SEVERITY LEVEL INDICATED BY FIRST CHARACTER OF MESSAGE, IN    */
     /* FORMAT INDICATED BY "TYPE" ARGUMENT.  IF OPTIONAL             */
     /* "SPECIAL_INSTRUCTIONS" ARGUMENT IS PRESENT, PERFORM THE       */
     /* OPERATIONS INDICATED (IN PDL) THERE.                          */
     /*****************************************************************/
     NOTE THAT IF MESSAGE IS A NUMBER THEN IT WOULD BE RETRIEVED FROM
        THE ERROR_MESSAGE_TABLE; ALL CATENATIONS HAVE ALREADY BEEN DONE;
     NOTE THAT FIRST CHARACTER OF MESSAGE IS TYPE, SECOND IS SEVERITY CODE,
              THIRD IS COLON, REMAINDER IS MESSAGE TEXT;
     IF TYPE = I                      /* I INDICATES INPUT PARSING ERROR */
        THEN STORE INPUT_PARSING_POINTER LOCATION, MESSAGE FOR LATER OUTPUT;
     ELSE IF TYPE = F                 /* F INDICATES FCF PARSING ERROR */
        THEN STORE FCF RULE NAME (TOP ELEMENT OF PROCEDURE_NAME_STACK),
                 POSITION IN RULE, MESSAGE FOR LATER OUTPUT;
     ELSE IF TYPE = G                 /* G INDICATES GENERAL ERROR CLASS */
        THEN OUTPUT MESSAGE IMMEDIATELY UNLESS SEVERITY = "N" AND NOTES = "NO";
     WHENEVER OUTPUT, MESSAGE WILL HAVE FORMAT BELOW, WHERE S IS SEVERITY:

               *** S ***     TEXT OF ERROR MESSAGE
     IF SEVERITY = "F"                                /* FATAL ERROR */
        THEN DO;      /* NOTE THAT MESSAGE HAS ALREADY BEEN OUTPUT OR STORED */
           IF THIS IS SECOND FATAL ERROR
              THEN STOP;
           .OUTPUT_STATUS("FLTOPS STATUS AT TIME OF FATAL ERROR:");
           .OUTPUT_ERRORS;
           TAKE ACTION TO STOP ALL SUBSEQUENT JOB STEPS;
           END;
     ELSE IF SEVERITY = "S" AND TYPE = "F"     /* SEVERE ERROR IN FCF */
        THEN DO;
           IF SEVERE_ERROR_COUNT = 0  THEN
              .OUTPUT_STATUS("FLTOPS STATUS AT TIME OF FIRST SEVERE ERROR:");
           INCREMENT SEVERE_ERROR_COUNT;
           SET FLTOPS FUNCTION TO SYNTAX;  /* CHECK ONLY FCF SYNTAX  */
           DO WHILE FCF_SYMBOL IS NOT A KEYWORD;
     /* KEY WORD TABLE CONTAINS KEY WORDS */
              CALL FCF_LEXICAL_ANALYZER;
              END;
           SET FCF_PARSING_TRUE_FALSE_INDICATOR TO "TRUE";
           TRANSFER PARSING CONTROL TO THE POINT FROM WHICH
              THE "STATEMENT" RULE WAS MOST RECENTLY INVOKED;
           END;
        ELSE IF SEVERITY = "S" AND TYPE = "I"    /* SEVERE ERROR IN INPUT */
           THEN DO;
              IF SEVERE_ERROR_COUNT = 0 THEN
                    .OUTPUT_STATUS('FLTOPS STATUS ON FIRST INPUT SEVERE ERROR')
              INCREMENT SEVERE_ERROR_COUNT;
              IF FLTOPS FUNCTION = FULL THEN SET FUNCTION TO TEST;
                  /* INHIBIT FURTHER FORTRAN OUTPUT */
              END;
     ELSE IF SEVERITY = "W"                           /* WARNING */
        THEN INCREMENT WARNING_COUNT;
     ELSE IF SEVERITY = "N"                           /* NOTE */
        THEN INCREMENT NOTE_COUNT;
     NOTE:  IF THIS ERROR MESSAGE HAS BEEN OUTPUT FOR THE SPECIFIC LOCATION
              ONCE BEFORE THEN DON'T OUTPUT AN ERROR MESSAGE THIS TIME.
     END .MESSAGE;
```

```
        /*********************************/
        /**  KEY WORD TABLE ENTRIES     **/
        /*********************************/
        /**      IF                     **/
        /**      DO                     **/
        /**      INVOKE                 **/
        /**      INPUT                  **/
        /**      SYMBOL_EXISTS          **/
        /**      TABLE_EMPTY            **/
        /**      NODE_EXISTS            **/
        /**      ASSERT                 **/
        /**      STATUS                 **/
        /**      FIND                   **/
        /**      EXECUTE                **/
        /**      PARAMETERS             **/
        /**      MESSAGE                **/
        /**      SET                    **/
        /**      INCREMENT              **/
        /**      DECREMENT              **/
        /**      OUTPUT                 **/
        /**      ENTER                  **/
        /**      CATENATE               **/
        /**      REPLACE                **/
        /**      DELETE                 **/
        /**      CLEAR_TABLE            **/
        /**      DEFINE                 **/
        /**      SAVE_POINTER           **/
        /**      RESTORE_POINTER        **/
        /**      ELSE                   **/
        /**      THEN                   **/
        /**      END                    **/
        /**      END OF FILE            **/
        /*********************************/

.OUTPUT_STATUS (HEADING):
      /*****************************************************************/
      /* OUTPUT "HEADING" ARGUMENT, FOLLOWED BY DETAILED INFORMATION */
      /* CONCERNING CURRENT CONTENTS OF SYMBOL TABLES AND ANY OTHER  */
      /* RELEVANT STATUS INFORMATION.                                */
      /*****************************************************************/
      WRITE HEADING;
      WRITE NAME, CONTENTS OF EACH SYMBOL TABLE;
      IF STATS = "YES"
         THEN WRITE TABLE OF FCF RULE INVOCATION FREQUENCIES;
      IF THERE WAS A FATAL ERROR OR SEVERE_ERROR_COUNT = 1
         THEN DO;
            WRITE ABBREVIATED REFERENCE TO CURRENT INPUT_PARSING_POINTER
                  LOCATION;
            WRITE CURRENTLY INVOKED FCF RULE WITH POINTER TO CURRENT
                  FCF_PARSING_POINTER LOCATION;
            IF TRACE = "HIGHER"
               THEN OUTPUT FLTOPS INTERNAL SWITCHES, VALUES;
            TAKE NECESSARY STEPS TO INHIBIT SUBSEQUENT JOB STEPS THAT
                  RELY ON SUCCESSFUL COMPLETION OF PRECOMPILER;
            END;
         ELSE LIST ALL SYNTHESIZED FORTRAN STATEMENTS;
      END .OUTPUT_STATUS;
```

```
.OUTPUT_ERRORS:
      /*****************************************************************/
      /* WRITE ERROR MESSAGES FOR INPUT, FCF FILES.  THESE MESSAGES    */
      /* WERE PREVIOUSLY SAVED FOR OUTPUT NOW.  IN WRITING INPUT       */
      /* ERROR MESSAGES, GENERATE MESSAGES FOR UNUSED INPUT            */
      /* INFORMATION.  WRITE ERROR MESSAGE FREQUENCY TABLES.           */
      /*****************************************************************/
      DO FOR EACH NODE IN INPUT PORTION OF INPUT FILE;
         WRITE TEXT OF NODE (IN BOLD PRINT IF THERE IS ERROR MESSAGE);
         IF INPUT ERROR MESSAGE WAS STORED FOR THIS NODE
            THEN WRITE POSITION POINTER, MESSAGE IN APPROPRIATE PLACE;
         IF ENTIRE NODE WAS NOT PARSED DURING FLTOPS PROCESSING
            THEN DO;
                 WRITE POSITION POINTER TO SHOW PARSING PROGRESS;
                 .MESSAGE(I,53)
                 INCREMENT WARNING COUNT;
         END;
      WRITE ERROR FREQUENCY TABLE FOR INPUT, INCLUDING SEVERE_ERROR_COUNT,
            WARNING_COUNT, NOTE_COUNT;
      DO FOR EACH FCF RULE FOR WHICH FCF ERROR MESSAGE(S) WAS (WERE) STORED;
         WRITE RULE;
         WRITE POSITION POINTER, ERROR MESSAGE IN APPROPRIATE PLACE;
         END;
      WRITE ERROR FREQUENCY TABLE FOR FCF;
      WRITE OVERALL ERROR FREQUENCY TABLE ;
      END .OUTPUT_ERRORS;

                  /*************************/
                  /* ARITHMETIC OPERATIONS */
                  /*************************/


.COMPUTE_VALUE :
      /*****************************************************/
      /* PERFORMS ARITHMETIC COMPUTATIONS (+,-,*,/,**)     */
      /* IMPLEMENTATION SHOULD PROVIDE MEANS TO CONVERT    */
      /* STRINGS TO NUMERIC FORM AND VICE VERSA            */
      /* COMPUTED_VALUE CONTAINS NUMERIC FORM              */
      /* TEMPORARY_STRING CONTAINS STRING FORM             */
      /*****************************************************/
      POP TOP ELEMENT FROM OPERATOR_STACK;
      POP TOP ELEMENT FROM ARITHMETIC_STACK AND SAVE AS Y;
      POP NEXT ELEMENT FROM ARITHMETIC_STACK AND SAVE AS X;
      PERFORM THE DESIRED OPERATION;
           /*** THE OPERATIONS SHOULD BE PERFORMED AS FOLLOWS ***/
           /***          X+Y    X-Y    X*Y    X/Y    X**Y          ***/
      SAVE THE RESULT IN COMPUTED_VALUE AND ALSO PUSH RESULT ONTO
                ARITHMETIC_STACK;
      END .COMPUTE_VALUE;
```

```
                /*******************************/
                /* MISCELLANEOUS OPERATIONS */
                /*******************************/


.DO (ARGUMENT):
      /*****************************************************************/
      /* CATCH-ALL ELEMENT WHICH ALLOWS DIRECT PDL DESCRIPTION IN    */
      /* GRAMMAR.                                                      */
      /*****************************************************************/
      PERFORM THE FUNCTION(S) DESCRIBED IN PDL IN THE ARGUMENT;
      END .DO;


.FALSE:
      /*****************************************************************/
      /* FORCE FAILURE OF RULE, ALTERNATIVE, ETC.  NOTE THAT THIS    */
      /* ELEMENT IS CONSIDERED SEMANTIC, EVEN THOUGH IT ALTERS THE   */
      /* VALUE OF FCF_PARSING_TRUE_FALSE_INDICATOR.                    */
      /*****************************************************************/
      SET FCF_PARSING_TRUE_FALSE_INDICATOR TO 'FALSE';
      END .FALSE;


.TRUE:
      /*****************************************************************/
      /* FORCE COMMITMENT TO RULE OR ALTERNATIVE.  INDICATE SUCCESS. */
      /*****************************************************************/
      SET FCF_PARSING_TRUE_FALSE_INDICATOR TO 'TRUE';
      END .TRUE;
```

FLTOPS FUNCTIONAL DEFINITION GRAMMAR

```
                    /*********************/
                    /* BASIC STRUCTURE */
                    /*********************/


GOAL_RULE  :=
      FLTOPS  ;


FLTOPS  :=
      .PREPROCESS_INPUT_FILE
      .SET(PASS1_PASS2_SWITCH = 1)
      .SET(DEFAULT_SYMBOL_TABLE = "FLTOPS")
      .SET(ARITH_ELEMENT_FOUND = "FALSE")
      .SET(ARITH_VALUE_STORED = "FALSE")
      .SET(ARITH_DEPTH_COUNTER = 0)
      .SET(ARITH_PAREN = 0)
      .SET(RELATIONAL_EXPRESSION = "FALSE")
      .FIND_TABLE(FLTOPS)
      .FIND_OR_ENTER_SYMBOL(DEBUG)     .ENTER_VALUE(OFF)
      .FIND_OR_ENTER_SYMBOL(FUNCTION)  .ENTER_VALUE(FULL)
      .FIND_OR_ENTER_SYMBOL(STATS)     .ENTER_VALUE(OFF)
      .FIND_OR_ENTER_SYMBOL(TRACE)     .ENTER_VALUE(OFF)
      .FIND_OR_ENTER_SYMBOL(NOTES)     .ENTER_VALUE(OFF)
      .FIND_OR_ENTER_SYMBOL(LBLNUM)    .ENTER_VALUE(99000)
      .FIND_OR_ENTER_SYMBOL(LENGTH)    .ENTER_VALUE(OFF)
      .FIND_OR_ENTER_SYMBOL(REAL)      .ENTER_VALUE('')
      .FIND_OR_ENTER_SYMBOL(INTEGER)   .ENTER_VALUE('')
      .FIND_OR_ENTER_SYMBOL(LOGICAL)   .ENTER_VALUE('')
      .FIND_OR_ENTER_SYMBOL(ALPHA)     .ENTER_VALUE('')
      .FIND_OR_ENTER_SYMBOL(XREF)      .ENTER_VALUE(OFF)
      .FIND_OR_ENTER_SYMBOL(OUTPUT)    .ENTER_VALUE(FORTRAN)
      .INVOKE(MAIN)
      $ STATEMENT
      ( "END"  "MAIN"    .ERROR(12)
      ! .TRUE    .MESSAGE(3) )
      ( .PEEK("END_OF_FILE")
      ! .TRUE   .MESSAGE(18)
        $( STATEMENT ! "END" ) )
      .SET(PASS1_PASS2_SWITCH = 2)
      ( .TEST(SEVERE_ERROR_COUNT = 0)
        .DO(PLACE FCF_PARSING_POINTER ON FIRST CHARACTER OF PASS1 OUTPUT FILE;
            CALL FCF_LEXICAL_ANALYZER;)
        $( .FORTRAN  .OUTPUT(*)  .OUTPUT(STATEMENT_TERMINATOR)  )
        "END OF FILE"
      ! .TRUE )
      .OUTPUT_STATUS("FLTOPS STATUS AT END OF RUN:")
      .OUTPUT_ERRORS  ;
```

FLTOPS FUNCTIONAL DEFINITION GRAMMAR

```
STATEMENT  :=
        IF_STATEMENT
      ! DO_STATEMENT
      ! INVOKE_STATEMENT
      ! INPUT_FILE_PARSING_OPERATION
        .SET(ERROR_NUMBER = 2)
        ERROR_OPTION
      ! FIND_STATEMENT
        .SET(ERROR_NUMBER = 31)
        ERROR_OPTION
      ! SYMBOL_EXISTS_STATEMENT
        .SET(ERROR_NUMBER = 4)
        ERROR_OPTION
      ! TABLE_EMPTY_STATEMENT
        .SET(ERROR_NUMBER = 8)
        ERROR_OPTION
      ! ASSERT_STATEMENT
        .SET(ERROR_NUMBER = 9)
        ERROR_OPTION
      ! NODE_EXISTS_STATEMENT
        .SET(ERROR_NUMBER = 57)
        ERROR_OPTION
      ! SAVE_POINTER_STATEMENT
      ! RESTORE_POINTER_STATEMENT
      ! STATUS_STATEMENT
      ! ENTER_STATEMENT
      ! CATENATE_STATEMENT
      ! REPLACE_STATEMENT
      ! DELETE_STATEMENT
      ! CLEAR_STATEMENT
      ! DEFINE_STATEMENT
      ! SET_STATEMENT
      ! INCREMENT_STATEMENT
      ! DECREMENT_STATEMENT
      ! MESSAGE_STATEMENT
      ! PARAMETERS_STATEMENT
      ! EXECUTE_STATEMENT
      ! OUTPUT_STATEMENT
      ! .FORTRAN
      ! "ELSE"   .MESSAGE(49)
      ! "THEN"   .MESSAGE(50)
      ! .PEEK("END")     .FALSE
      ! .PEEK("END OF FILE")   .FALSE
      ! .TRUE     .MESSAGE(1)   ;
```

```
ERROR_OPTION  :=
      .TEST(OPERATION_TRUE_FALSE_INDICATOR = "FALSE")
      ( "ERROR" ":"    .ERROR(17)  STATEMENT
      ! .TRUE
        .MESSAGE(ERROR_NUMBER) )
    ! ( "ERROR" ":"    .ERROR(17)    .SCAN_TO_LOCATION(1)
        STATEMENT
        .LOCATION(1)
      ! .TRUE ) ;

                    /******************/
                    /* IF STATEMENT */
                    /******************/


IF_STATEMENT  :=
      "IF"
      BOOLEAN_EXPRESSION    .ERROR(32)
      "THEN"              .ERROR(33)
      .DO(PUSH OPERATION_TRUE_FALSE_INDICATOR ONTO BOOLEAN_TRUTH_STACK)
      ( .TEST(OPERATION_TRUE_FALSE_INDICATOR = "FALSE")
        .SCAN_TO_LOCATION(1)
      ! .TRUE )
      STATEMENT
      .LOCATION(1)
      .DO(POP TOP ELEMENT FROM BOOLEAN_TRUTH_STACK AND SAVE AS
               OPERATION_TRUE_FALSE_INDICATOR;)
      ( "ELSE"      /* OPTIONAL ELSE CLAUSE */
        ( .TEST(OPERATION_TRUE_FALSE_INDICATOR = "TRUE")
          .SCAN_TO_LOCATION(2)
        ! .TRUE )
        STATEMENT
      ! .TRUE )
      .LOCATION(2) ;



                    /******************/
                    /* DO STATEMENT */
                    /******************/


DO_STATEMENT  :=
      "DO"
      ( "FOR"  ( DO_FOR_SYMBOL_TABLE
               ! DO_FOR_EACH )     .ERROR(13)
      ! DO_WHILE
      ! SIMPLE_DO )  ;
```

FLTOPS FUNCTIONAL DEFINITION GRAMMAR

```
DO_WHILE  :=
      "WHILE"
      .RECOVERY_POINT
      BOOLEAN_EXPRESSION    .ERROR(32)
      ( .TEST(OPERATION_TRUE_FALSE_INDICATOR = "FALSE")
        .SCAN_TO_LOCATION(1)
      ! .TRUE )
      $ STATEMENT
      "END"            .ERROR(20)
      .RETURN_TO_RECOVERY_POINT
      .LOCATION(1)  ;


DO_FOR_SYMBOL_TABLE  :=
      "SYMBOL"  "TABLE"            .ERROR(34)
      .DO(PUSH CURRENT DEFAULT_SYMBOL_TABLE ONTO DEFAULT_TABLE_STACK;)
      .SET(RULE_TABLE_OR_SYMBOL_NAME = "TRUE")
      STRING_EXPRESSION  .ERROR(35)
      .SET(DEFAULT_SYMBOL_TABLE = **)
      $ STATEMENT
      "END"            .ERROR(22)
      .DO(POP TOP ELEMENT FROM DEFAULT_TABLE_STACK AND SAVE AS
             DEFAULT_SYMBOL_TABLE;) ;


DO_FOR_EACH  :=
      "EACH"  "SUBNODE"           .ERROR(23)
      "OF"            .ERROR(36)
      .DO(PUSH VALUE OF INPUT_PARSING_POINTER ONTO INPUT_POINTER_STACK;)
      FIND_INPUT_NODE    .ERROR(54)
      .MOVE_INPUT_POINTER_DOWN
      .SET(BROTHER_NODE_END = "FALSE")
      ( .TEST(OPERATION_TRUE_FALSE_INDICATOR = "FALSE")
        .MESSAGE(37)
        .SCAN_TO_LOCATION(1)
      ! .TRUE )
    $ ( .TEST(OPERATION_TRUE_FALSE_INDICATOR = "TRUE")
        .RECOVERY_POINT
        $ STATEMENT
        "END"          .ERROR(24)
      .ADVANCE_INPUT_POINTER
      ( .TEST(BROTHER_NODE_END = "FALSE")
        .RETURN_TO_RECOVERY_POINT
      ! .TRUE  .DO(POP TOP ELEMENT FROM RECOVERY_POINT_STACK;) ) )
      .LOCATION(1)
      .DO(POP TOP ELEMENT FROM INPUT_POINTER_STACK AND SAVE AS
             INPUT_PARSING_POINTER;)
      .SER(BROTHER_NODE_END = "FALSE") ;
```

```
SIMPLE_DO  :=
      .TRUE
      $ STATEMENT
      "END"          .ERROR(21)  ;


                /**********************/
                /*  INVOKE STATEMENT */
                /**********************/


INVOKE_STATEMENT  :=
      "INVOKE"   /* FCF STATEMENTS ARE NOW PROCESSED IN NAMED FCF RULE */
      .SET(RULE_TABLE_OR_SYMBOL_NAME = "TRUE")
      STRING_EXPRESSION    .ERROR(38)
      .RECOVERY_POINT
      .INVOKE(**)
      ( .TEST(OPERATION_TRUE_FALSE_INDICATOR = "TRUE")
        $ STATEMENT
        "END"    .ERROR(39)
        .IDENTIFIER    .ERROR(12)
        .END_PROCEDURE(*)
        ( .PEEK("END OF FILE")
        ! .TRUE  .MESSAGE(18)
          $( STATEMENT ! "END" ) )
      ! .TRUE    .MESSAGE(52) )
      .RETURN_TO_RECOVERY_POINT  ;
                      /* END OF FILE IN THIS CONTEXT IMPLIES THE LOGICAL */
                      /* END OF THE RULE HAS BEEN FOUND                  */


                /*****************************/
                /*  INPUT FILE STATEMENTS  */
                /*****************************/


INPUT_FILE_PARSING_OPERATION  :=
      "INPUT"          /* GET NEXT TOKEN FROM INPUT SPECIFICATION */
      LEFT_PAREN
      .DO(PUSH VALUE OF INPUT_PARSING_POINTER ONTO INPUT_POINTER_STACK;)
      INPUT_GROUP    .ERROR(15)
      $( "OR"    /* MULTIPLE INPUT INQUIRIES ARE POSSIBLE */
         ( .TEST(OPERATION_TRUE_FALSE_INDICATOR = "TRUE")
           .SCAN_TO_LOCATION(1)
         ! .TRUE )
         INPUT_GROUP  .ERROR(15) )
      .LOCATION(1)
      RIGHT_PAREN ;
```

FLTOPS FUNCTIONAL DEFINITION GRAMMAR

```
INPUT_GROUP  :=
      INPUT_ELEMENT
      .SET(FINISH_FLAG = "FALSE")
    $ ( .TEST(FINISH_FLAG = "FALSE")
        ( .TEST(OPERATION_TRUE_FALSE_INDICATOR = "FALSE")
          .SCAN_TO_LOCATION(1)
        ! .TRUE )
        ( INPUT_ELEMENT  !  .TRUE   .SET(FINISH_FLAG = "TRUE") ) )
      .LOCATION(1) ;



INPUT_ELEMENT  :=
      "CONSTANT"
      .PARSE_INPUT(NUMBER)
      ( .TEST(OPERATION_TRUE_FALSE_INDICATOR = "FALSE")
        SIGNED_NUMBER
      ! .TRUE  )
    ! "NUMBER"  .PARSE_INPUT(NUMBER)
    ! "SIGNED_NUMBER"     SIGNED_NUMBER
    ! "WORD"  .PARSE_INPUT(IDENTIFIER)
    ! "STRING"  .PARSE_INPUT(STRING)
    ! .STRING  .DO(PROCESS FCF_PREVIOUS_SYMBOL TO ELIMINATE DELIMITERS;)
      .PARSE_INPUT(*) ;


SIGNED_NUMBER  :=
      .DO(PUSH INPUT_PARSING_POINTER ONTO INPUT_POINTER_STACK;)
      .PARSE_INPUT("-")
      ( .TEST(OPERATION_TRUE_FALSE_INDICATOR = "FALSE")
        .PARSE_INPUT("+")
      ! .TRUE )
      ( .TEST(OPERATION_TRUE_FALSE_INDICATOR = "TRUE")
        .SET(** = INPUT_PREVIOUS_SYMBOL)
        .PARSE_INPUT(NUMBER)
        ( .TEST(OPERATION_TRUE_FALSE_INDICATOR = "TRUE")
          .DO(INPUT_PREVIOUS_SYMBOL = **!!INPUT_PREVIOUS_SYMBOL;)
          .DO(POP TOP ELEMENT FROM INPUT_POINTER_STACK;)
        ! .DO(POP TOP ELEMENT FROM INPUT_POINTER_STACK AND SAVE AS
                INPUT_PARSING_POINTER;)
          .TRUE )
      ! .TRUE  .DO(POP TOP ELEMENT FROM INPUT_POINTER_STACK AND SAVE AS
                    INPUT_PARSING_POINTER;) ) ;


FIND_STATEMENT  :=
      "FIND"   /* POSITIONS INPUT_PARSING_POINTER  TO SPECIFIED INPUT NODE */
      LEFT_PAREN
      .DO(PUSH VALUE OF INPUT_PARSING_POINTER ONTO INPUT_POINTER_STACK;)
      FIND_INPUT_NODE  .ERROR(54)
      ( .TEST(OPERATION_TRUE_FALSE_INDICATOR = "TRUE")
        .DO(POP TOP ELEMENT FORM INPUT_POINTER_STACK;)
      ! .TRUE   .DO(POP TOP ELEMENT FORM INPUT_POINTER_STACK AND SAVE
                    AS INPUT_PARSING_POINTER;) )
      RIGHT_PAREN ;
```

```
SAVE_POINTER_STATEMENT  :=
      "SAVE_POINTER"
      .DO(PUSH VALUE OF INPUT_PARSING_POINTER ONTO INPUT_POINTER_STACK;)


RESTORE_POINTER_STATEMENT  :=
      "RESTORE_POINTER"
      .DO(POP TOP ELEMENT FROM INPUT_POINTER_STACK AND SAVE
            AS INPUT_PARSING_POINTER;)


            /**********************/
            /*  TEST STATEMENTS  */
            /**********************/


SYMBOL_EXISTS_STATEMENT  :=
      "SYMBOL_EXISTS"        /* TEST FOR PRESENCE OF SYMBOL IN TABLE */
      LEFT_PAREN
      .SET(RULE_TABLE_OR_SYMBOL_NAME = "TRUE")
      STRING_EXPRESSION  .ERROR(35)
      ( .PEEK(")")
        .FIND_TABLE(DEFAULT_SYMBOL_TABLE)
        .TEST_FOR_SYMBOL(**)
      ! COMMA
        .FIND_TABLE(**)
      .SET(RULE_TABLE_OR_SYMBOL_NAME = "TRUE")
        STRING_EXPRESSION  .ERROR(40)   .TEST_FOR_SYMBOL(**) )
      RIGHT_PAREN ;

TABLE_EMPTY_STATEMENT  :=
      "TABLE_EMPTY"          /* TEST FOR EMPTY TABLE */
      LEFT_PAREN
      .SET(RULE_TABLE_OR_SYMBOL_NAME = "TRUE")
      STRING_EXPRESSION  .ERROR(35)   .FIND_TABLE(**)
      ( .TEST(SYMBOL_TABLE_POINTER = NULL)
        .SET(OPERATION_TRUE_TRUE_INDICATOR = "FALSE")
      ! .TRUE  .SET(OPERATION_TRUE_FALSE_INDICATOR = "FALSE") )
      RIGHT_PAREN  ;

NODE_EXISTS_STATEMENT  :=
      "NODE_EXISTS"   /* TEST FOR OCCURRENCE OF SPECIFIED INPUT TREE */
      LEFT_PAREN
      .DO(PUSH INPUT_PARSING_POINTER ONTO INPUT_POINTER_STACK;)
      FIND_INPUT_NODE   .ERROR(54)
      .DO(POP TOP ELEMENT FROM INPUT_POINTER_STACK AND SAVE
            AS INPUT_PARSING_POINTER;)
      RIGHT_PAREN;
```

```
                /************************/
                /*  ASSERT STATEMENT    */
                /************************/


ASSERT_STATEMENT  :=
      "ASSERT"
      ( .TEST(DEBUG = "OFF" )
        .SCAN_TO_LOCATION(1)
        .SET(OPERATION_TRUE_FALSE_INDICATOR = "TRUE")
      ! .TRUE)
      BOOLEAN_EXPRESSION    .ERROR(32)
      .LOCATION(1);

                /**********************/
                /* STATUS STATEMENT */
                /**********************/


STATUS_STATEMENT  :=
      "STATUS"
      ( .TEST(DEBUG = "OFF")
        .SCAN_TO_LOCATION(1)
      ! .TRUE )
      LEFT_PAREN
      .STRING    .ERROR(55)
      .DO(PROCESS * TO REMOVE DELIMITERS, THEN SAVE AS HEADING;)
      ( .PEEK(")")    .OUTPUT_STATUS(HEADING)
      ! COMMA    STRING_EXPRESSION  .ERROR(35)    .SET(TABLE_NAME = *)
        ( .PEEK(")")    .DO(WRITE OUT HEADING AND CONTENTS OF TABLE TABLE_NAME;)
        ! ARRAY_ELEMENTS    .GET_VALUE(**)
          .DO(WRITE OUT HEADING,TABLE_NAME AND CONTENTS OF **;)
        ! COMMA    STRING_EXPRESSION    .ERROR(40)    .GET_VALUE(**)
          .DO(WRITE OUT HEADING,TABLE_NAME, CONTENTS OF * AND CONTENTS OF **;)
      RIGHT_PAREN
      .LOCATION(1) ;
```

```
/*************************************************************************/
/***************  SYMBOL TABLE MANIPULATION STATEMENTS  ***************/
/*************************************************************************/

                /**********************/
                /*  ENTER STATEMENT   */
                /**********************/


ENTER_STATEMENT  :=
      "ENTER"    /*  ENTER VALUES INTO SYMBOL TABLE */
      LEFT_PAREN
      LOCATION_POINTER    .ERROR(35)
      ( ","      /* VALUE FIELD IS PRESENT  */
        STRING_EXPRESSION     .ERROR(14)
        (FORMAT_STATEMENT ! .TRUE)
        .ENTER_VALUE(**)
      ! .TRUE )
      RIGHT_PAREN ;


                /**************************/
                /*  CATENATE STATEMENT   */
                /**************************/


CATENATE_STATEMENT  :=
      "CATENATE"  /* PERFORMS STRING CATENATION ON VALUE FIELDS */
      LEFT_PAREN
      LOCATION_POINTER    .ERROR(35)
    $ ( ","  STRING_EXPRESSION    .ERROR(14)
        (FORMAT_STATEMENT ! .TRUE)
        .CATENATE_VALUE(**) )
      RIGHT_PAREN ;

                /*************************/
                /*  REPLACE STATEMENT   */
                /*************************/


REPLACE_STATEMENT  :=
      "REPLACE"  /* PERFORMS SUBSTRING REPLACEMENT */
      LEFT_PAREN
      LOCATION_POINTER    .ERROR(35)
      ( .PEEK(")")  .MESSAGE(42)
      ! COMMA    STRING_EXPRESSION    .ERROR(14)
        .SET(STRING1 = **)
        COMMA  STRING_EXPRESSION  .ERROR(14)
        (FORMAT_STATEMENT ! .TRUE)
        .SET(STRING2 = **)
        .SET(NUMBER_REPLACES = INFINITY)
        ( ","  ARITHMETIC_EXPRESSION  .ERROR(14)
          .SET(NUMBER_REPLACES = **)
        ! .TRUE )
        .REPLACE_STRING(STRING1,STRING2,NUMBER_REPLACES) )
      RIGHT_PAREN;
```

```
                /************************/
                /*  DELETE STATEMENT  */
                /************************/

DELETE_STATEMENT  :=
      "DELETE"                /* DELETE ENTRY FROM SYMBOL TABLE */
      LEFT_PAREN
      .SET(RULE_TABLE_OR_SYMBOL_NAME = "TRUE")
      STRING_EXPRESSION  .ERROR(35)  .FIND_TABLE(**)
      ( .PEEK(')')
        .FIND_TABLE(DEFAULT_SYMBOL_TABLE)
        .TEST_FOR_SYMBOL(**)
        ( .TEST(OPERATION_TRUE_FALSE_INDICATOR = "TRUE")
          .DELETE_ENTRY
        ! .TRUE )
      ! .TRUE
        .FIND_TABLE(**)
        ( .PEEK("(")
          ARRAY_ELEMENTS
          .DO(POP TOP ELEMENT FROM SUBSCRIPT_COUNTER_STACK AND
                SAVE AS STACK_SIZE;)
          .COMPUTE_TABLE_POSITION
          .FIND_BY_SUBSCRIPT(ARRAY_POSITION)
        ! COMMA  .SET(RULE_TABLE_OR_SYMBOL_NAME = "TRUE")
          STRING_EXPRESSION  .ERROR(40)
          .TEST_FOR_SYMBOL(**)
          ( .TEST(OPERATION_TRUE_FALSE_INDICATOR = "TRUE")
            .DELETE_ENTRY
          ! .TRUE ) ) )
      RIGHT_PAREN ;

                /************************/
                /*  CLEAR STATEMENT  */
                /************************/

CLEAR_STATEMENT  :=
      "CLEAR_TABLE"           /* DELETE ALL ENTRIES IN TABLE */
      LEFT_PAREN
      .SET(RULE_TABLE_OR_SYMBOL_NAME = "TRUE")
      STRING_EXPRESSION  .ERROR(35)  .FIND_TABLE(**)
      RIGHT_PAREN
      $( .TEST(SYMBOL_TABLE_POINTER NOT NULL)  .DELETE_ENTRY );

                /************************/
                /*  DEFINE STATEMENT  */
                /************************/

DEFINE_STATEMENT  :=
      "DEFINE"  /* SETS UP TABLES THAT CAN BE ACCESSED VIA SUBSCRIPTS */
      .SET(RULE_TABLE_OR_SYMBOL_NAME = "TRUE")
      STRING_EXPRESSION          .ERROR(35)
      .SET(DEFINE_TABLE_NAME = ** )
      DEFINED_ELEMENT    .ERROR(59)
      $ ( ","  .SET(RULE_TABLE_OR_SYMBOL_NAME = "TRUE")
          STRING_EXPRESSION    .ERROR(35)
          .SET(DEFINE_TABLE_NAME = ** )
          DEFINED_ELEMENT    .ERROR(59) );
```

```
DEFINED_ELEMENT  :=
      ARRAY_ELEMENTS
      .ARRAY_SETUP(DEFINE_TABLE_NAME)
    ! "AS"    /* DEFINE TABLE EQUIVALENCE */
      ( "NULL"   /* ELIMINATE EQUIVALENCE AND REMOVE DIMENSIONS */
        .NULL_OUT_TABLE(DEFINE_TABLE_NAME)
      ! .SET(RULE_TABLE_OR_SYMBOL_NAME = "TRUE")
        STRING_EXPRESSION
        .SET(EQUIVALENCED_TABLE_NAME = **)
        .SETUP_TABLE_EQUIVALENCE
      ! .TRUE   .MESSAGE(27) ) ;



                /*********************/
                /*  SET STATEMENT  */
                /*********************/


SET_STATEMENT  :=                  /*  SHORTHAND FORM OF THE ENTER STATEMENT  */
      "SET"  LEFT_PAREN              /* SET SYMBOL TO INDICATED VALUE */
      .SET(RULE_TABLE_OR_SYMBOL_NAME = "TRUE")
      STRING_EXPRESSION  .ERROR(40)
      .FIND_TABLE(DEFAULT_SYMBOL_TABLE)
      .FIND_OR_ENTER_SYMBOL(**)
      ( "="    /* OPTIONAL ARITHMETIC ASSIGNMENT STATEMENT */
        .SAVE_POINTER
        STRING_EXPRESSION   .ERROR(54)
        (FORMAT_STATEMENT ! .TRUE )
        .RESTORE_POINTER
        .ENTER_VALUE(**)
      ! .TRUE )
      RIGHT_PAREN;


                /****************************/
                /*  INCREMENT STATEMENT  */
                /****************************/


INCREMENT_STATEMENT  :=
      "INCREMENT"  LEFT_PAREN       /* INCREMENT SWITCH VALUE BY ONE */
      .SET(RULE_TABLE_OR_SYMBOL_NAME = "TRUE")
      STRING_EXPRESSION   .ERROR(40)
      .FIND_TABLE(DEFAULT_SYMBOL_TABLE)
      .FIND_OR_ENTER_SYMBOL(**)
      .GET_VALUE(**)
      .DO(ADD 1 TO TEMPORARY_STRING; NOTE THAT STRING MUST BE TREATED AS
          ARITHMETIC VALUE; IF STRING IS NULL, TREAT AS ZERO;)
      .ENTER_VALUE(**)  RIGHT_PAREN;
```

```
                /****************************/
                /*  DECREMENT STATEMENT  */
                /****************************/


DECREMENT_STATEMENT  :=
      "DECREMENT"  LEFT_PAREN       /* DECREMENT SWITCH VALUE BY ONE */
      .SET(RULE_TABLE_OR_SYMBOL_NAME = "TRUE")
      STRING_EXPRESSION    .ERROR(40)
      .FIND_OR_ENTER_SYMBOL(**)
      .GET_VALUE(**)
      .DO(SUBTRACT 1 FROM TEMPORARY_STRING;)
      .ENTER_VALUE(**)  RIGHT_PAREN ;


/*************************************************************************/
/*******************  END OF TABLE MANIPULATION STATEMENTS  ***************/
/*************************************************************************/


                /***********************************/
                /* BUILT IN FUNCTIONS              */
                /***********************************/



                 /************************/
                 /* INDEX FUNCTION      */
                 /************************/


INDEX_FUNCTION  :=    /* DETERMINE STARTING POSITION OF SUBSTRING */
      "INDEX"
      LEFT_PAREN
      .INCREMENT(ARITH_DEPTH_COUNTER)
      STRING_EXPRESSION    .ERROR(14)
      .SET(STRING1 = **)
      COMMA  STRING_EXPRESSION    .ERROR(14)
      .SET(STRING2 = **)
      .INDEX(STRING1,STRING2,START_POS)
      .DECREMENT(ARITH_DEPTH_COUNTER)
      RIGHT_PAREN ;


                /*************************/
                /* LENGTH FUNCTION      */
                /*************************/


LENGTH_FUNCTION  :=        /* DETERMINE THE LENGTH OF A STRING */
      "LENGTH"
      LEFT_PAREN
      .INCREMENT(ARITH_DEPTH_COUNTER)
      STRING_EXPRESSION    .ERROR(14)
      .LENGTH(**,STRING_LENGTH)
      .DECREMENT(ARITH_DEPTH_COUNTER)
      RIGHT_PAREN ;
```

```
                /*************************/
                /*  SUBSTRING FUNCTION   */
                /*************************/


SUBSTRING_FUNCTION  :=
      "SUBSTRING"     /* GET SUBSTRING FROM INDICATED STRING */
      LEFT_PAREN
      STRING_EXPRESSION      .ERROR(58)
      .SET(STRING_IN = **)
      COMMA   ARITHMETIC_EXPRESSION      .ERROR(47)
      .SET(START_POS = COMPUTED_VALUE)
      ( "," ARITHMETIC_EXPRESSION      .ERROR(47)
        .SUBSTRING(STRING_IN,**,START_POS,COMPUTED_VALUE)
      ! .TRUE
        .SUBSTRING(STRING_IN,**,START_POS,0) )
      RIGHT_PAREN ;



                /***********************/
                /*  VALUE  FUNCTION   */
                /***********************/


VALUE_FUNCTION  :=
      "VALUE"  /* RETRIEVE VALUE FROM DESIGNATED PLACE */
      LEFT_PAREN
      ( "a"   /* SYMBOL TABLE POINTER ALREADY SET */
      ! .SET(RULE_TABLE_OR_SYMBOL_NAME = "TRUE")
        STRING_EXPRESSION
        .SAVE_POINTER
        ( .PEEK(")")  .FIND_TABLE(DEFAULT_SYMBOL_TABLE)
          .FIND_OR_ENTER_SYMBOL(**)
        ! "," .FIND_TABLE(**)
        .SET(RULE_TABLE_OR_SYMBOL_NAME = "TRUE")
          STRING_EXPRESSION    .ERROR(40)
          .FIND_OR_ENTER_SYMBOL(**)
        ! .PEEK("(")
          .FIND_TABLE(**)
          ARRAY_ELEMENTS
          .DO(POP TOP ELEMENT FROM SUBSCRIPT_COUNTER_STACK AND
                SAVE AS STACK_SIZE;)
          .COMPUTE_TABLE_POSITION
          .FIND_BY_SUBSCRIPT(ARRAY_POSITION)
      ! .TRUE   .MESSAGE(35) )
      .SET(VALUE_LOCATION = SYMBOL_TABLE_POINTER)
      .GET_VALUE(**)
      .RESTORE_POINTER
      RIGHT_PAREN ;
```

```
                /*************************/
                /*  MESSAGE STATEMENT    */
                /*************************/


MESSAGE_STATEMENT  :=
      "MESSAGE"  LEFT_PAREN
      STRING_EXPRESSION    .ERROR(26)
      (FORMAT_STATEMENT ! .TRUE)
      .SET(MESSAGE = **)
      .DO(CHECK FIRST TWO CHARACTERS OF MESSAGE FOR LEGAL TYPE
            AND SEVERITY LEVEL; IF EITHER NOT VALID THEN OUTPUT
            WARNING TO THAT EFFECT; CATENATE GW: TO START OF MESSAGE;)
      $ ( ","  STRING_EXPRESSION    .ERROR(26)
          (FORMAT_STATEMENT ! .TRUE )
          .DO(MESSAGE = MESSAGE!!TEMPORARY_STRING;) )
      .MESSAGE(MESSAGE)   RIGHT_PAREN  ;


                /*****************************************/
                /*  EXECUTE AND PARAMETERS STATEMENTS    */
                /*****************************************/


PARAMETERS_STATEMENT  :=
      "PARAMETERS"
      .IDENTIFIER      .ERROR(44)
      .SET(EXECUTE_ROUTINE_NAME = "$"!!*)
      .SET(PARAMETER_COUNT = 1)
      LEFT_PAREN  PARAMETER_TYPE    .ERROR(10)
      .FIND_TABLE(EXECUTE_ROUTINE_NAME)
      .DO(ENTER VALUES FOR PARAMETER_CLASS AND PARAMETER_SIZE INTO TABLE
            IN POSITION INDICATED BY PARAMETER_COUNT;)
      $ ( "," .INCREMENT(PARAMETER_COUNT)
          PARAMETER_TYPE    .ERROR(10)
          .FIND_TABLE(EXECUTE_ROUTINE_NAME)
          .DO(ENTER VALUES FOR PARAMETER_CLASS AND PARAMETER_SIZE INTO TABLE
                IN POSITION INDICATED BY PARAMETER_COUNT;) )
      RIGHT_PAREN  ;
```

```
PARAMETER_TYPE  :=
      ( "REAL" ! "INTEGER" ! "ALPHA" ! "LOGICAL" ! "DOUBLE" )
      .SET(PARAMETER_CLASS = *)
      .SET(PARAMETER_SIZE = 1)
      ( .PEEK("(")
        .SET(POSITION_VALUE = 1)
        ARRAY_ELEMENTS
      .DO(COMPUTE PRODUCT OF ELEMENTS IN ARRAY_STACK AND
            SAVE AS ARRAY_POSITION;)
        .SET(PARAMETER_SIZE = ARRAY_POSITION )
      ! .TRUE ) ;

EXECUTE_STATEMENT  :=
      "EXECUTE"  /*  EXECUTE A ROUTINE TO DETERMINE PRECOMPILE TIME VALUES */
      .IDENTIFIER      .ERROR(44)
      .SET(EXECUTE_ROUTINE_NAME = "$"!!*)
      LEFT_PAREN  .SET(PARAMETER_COUNT = 1)
      .DO(SET EXECUTE_DATA_AREA_POINTER TO LAST LOCATION IN EXECUTE_DATA_AREA
            THAT HAS RECEIVED DATA;)
      EXECUTE_PARAMETER    .ERROR(45)
      $ ( ","  .INCREMENT(PARAMETER_COUNT)
          EXECUTE_PARAMETER    .ERROR(45)  )
      RIGHT_PAREN     .EXECUTE(EXECUTE_ROUTINE_NAME)
      .SET(CONVERT_FLAG = "OUT")
      .EXECUTE_PARAMETER_UPDATE  ;


EXECUTE_PARAMETER :=
      .SET(PARAMETER_LOCATION = NULL)
      .SET(ARITH_DEPTH_COUNTER = 0)
      .SET(ARITH_OPERATOR_FOUND = "FALSE")
      ( .STRING    .DO(PROCESS FCF_PREVIOUS_SYMBOL TO REMOVE DELIMITERS
                        THEN COPY INTO TEMPORARY_STRING;)
      ! ARITHMETIC_EXPRESSION
        .DO(POP TOP ELEMENT FROM ARITHMETIC_STACK;)
        ( .TEST(ARITH_OPERATOR_FOUND = "FALSE")
          .SET(PARAMETER_LOCATION = VALUE_LOCATION)
        ! .TRUE ) )
      .FIND_TABLE(EXECUTE_ROUTINE_NAME)
      .TEST(HAS PARAMETER INFORMATION BEEN ADDED TO TABLE;)    .ERROR(51)
      .DO(FOR ARGUMENT IDENTIFIED BY PARAMETER_COUNT ENTER PARAMETER_LOCATION
            AND RETRIEVE VALUES FOR PARAMETER_CLASS AND PARAMETER_SIZE;)
      .SET(CONVERT_FLAG = "IN")
      .EXECUTE_DATA_AREA_UPDATE;
```

```
                /************************/
                /*  OUTPUT STATEMENT    */
                /************************/


OUTPUT_STATEMENT  :=
      "OUTPUT"  LEFT_PAREN
      OUTPUT_ELEMENT    .ERROR(46)
      $( ","  OUTPUT_ELEMENT    .ERROR(46)  )
      RIGHT_PAREN  ;


OUTPUT_ELEMENT  :=
      .LABEL  .DO(PLACE CORRESPONDING GENERATED STATEMENT NUMBER IN
                  TEMPORARY_STRING;)
      .OUTPUT(**)
    ! "#"  .OUTPUT(STATEMENT_TERMINATOR)
    ! ("EOF" ! "END_OF_FILE")
      .DO(IF OUTPUT_BUFFER NON EMPTY THEN OUTPUT ITS CONTENTS;
          THEN OUTPUT END OF  FILE MARK;)
    ! ("EOR" ! "END_OF_RECORD")
      .DO(IF OUTPUT_BUFFER IS NON-EMPTY THEN OUTPUT ITS CONTENTS;
            THEN OUTPUT END OF RECORD MARK;)
    ! COLUMN_FORMAT
    ! ("COMMENT" ! "C")    .SET(** = *)
      ":"    .ERROR(17)
      .SET(OUTPUT_COMMENT_CARD = "TRUE")
      .OUTPUT(**)
    ! STRING_EXPRESSION
      (FORMAT_STATEMENT ! .TRUE)
      .OUTPUT(**) ;


COLUMN_FORMAT  :=      /* ADJUST COLUMNS */
      ("COL" ! "COLUMN")
      LEFT_PAREN
      ARITHMETIC_EXPRESSION    .ERROR(47)
      .DO(POP TOP ELEMENT FROM ARITHMETIC_STACK;)
      .ADJUST_COLUMN(**)
      RIGHT_PAREN ;
```

AD-A068 394 SCIENCE APPLICATIONS INC ENGLEWOOD CO F/G 9/2
FLIGHT TEST ORIENTED PRECOMPILER SYSTEM (FLTOPS DESIGN SPECIFIC--ETC(U)
AUG 78 D A OTEY, H R RAMSEY, J K WILLOUGHBY F04611-77-C-0040
UNCLASSIFIED SAI-78-061-DEN AFFTC-TR-78-22 NL

3 OF 4
AD A068394

```
FORMAT_STATEMENT  :=      /* FORMATTED OUTPUT */
      ( "FORMAT"    .SET(STRING_IN = **)
        LEFT_PAREN
        FORMAT_ELEMENT    .ERROR(60)
        RIGHT_PAREN
      ! "=>"    .SET(STRING_IN = **)
        FORMAT_ELEMENT    .ERROR(60) )
      .FORMAT_STRING(STRING_IN,**) ;


FORMAT_ELEMENT  :=
      VALUE_FUNCTION
    ! .IDENTIFIER    .SET(** = *)
      ( .NUMBER  .DO(TEMPORARY_STRING = TEMPORARY_STRING!!FCF_PREVIOUS_SYMBOL;)
      ! .TRUE );



                  /*****************************/
                  /*  ARITHMETIC EXPRESSIONS   */
                  /*****************************/


ARITHMETIC_EXPRESSION  :=
      ( .TEST(ARITH_ELEMENT_FOUND = "TRUE")
        ARITHMETIC_TERM
      ! ( "+" ARITHMETIC_TERM
        ! "-" ARITHMETIC_TERM .DO(CHANGE SIGN OF TOP ELEMENT OF ARITHMETIC_STACK;)
          ( .TEST(ARITH_DEPTH_COUNTER = 0)
            .SET(ARITH_OPERATOR_FOUND = "TRUE")
          ! .TRUE )
        ! ARITHMETIC_TERM ) )
      $ ( ("+" ! "-" ) .DO(PUSH FCF_PREVIOUS_SYMBOL ONTO OPERATOR_STACK;)
          ARITHMETIC_TERM    .ERROR(16)
          .COMPUTE_VALUE  )
      ( .TEST(PARENTHESIS_COUNT = 0)
      ! .TEST(ARITH_PAREN = 0)
        ( ")"  .DECREMENT(PARENTHESIS_COUNT)
          ( .PEEK("*" ! "+" ! "**" ! "-" ! "/" ! ")")
            .SET(ARITH_ELEMENT_FOUND = "TRUE")
            .SET(ARITH_VALUE_STORED = "TRUE")
            ARITHMETIC_EXPRESSION
          ! .TRUE )
        ! .TRUE )
      ! .TRUE )  ;


ARITHMETIC_TERM  :=
      ARITHMETIC_FACTOR
      $ ( ( "*" ! "/" ) .DO(PUSH FCF_PREVIOUS_SYMBOL ONTO OPERATOR_STACK;)
          ( .TEST(ARITH_DEPTH_COUNTER = 0)
            .SET(ARITH_OPERATOR_FOUND = "TRUE")
          ! .TRUE )
          ARITHMETIC_FACTOR    .ERROR(16)
          .COMPUTE_VALUE  )  ;
```

FLTOPS FUNCTIONAL DEFINITION GRAMMAR

```
ARITHMETIC_FACTOR  :=
      ARITHMETIC_PRIMARY
      ( "**"  .DO(PUSH FCF_PREVIOUS_SYMBOL ONTO OPERATOR_STACK;)
        ( .TEST(ARITH_DEPTH_COUNTER = 0)
          .SET(ARITH_OPERATOR_FOUND = "TRUE")
        ! .TRUE )
        ARITHMETIC_FACTOR    .ERROR(16)
        .COMPUTE_VALUE
      ! .TRUE ) ;


ARITHMETIC_PRIMARY  :=
      "("    .INCREMENT(ARITH_PAREN)
      ARITHMETIC_EXPRESSION  .ERROR(47)  ")"    .ERROR(28)
      .DECREMENT(ARITH_PAREN)
    ! ( .TEST(ARITH_ELEMENT_FOUND = "TRUE")
        .SET(ARITH_ELEMENT_FOUND = "FALSE")
      ! VALUE_FUNCTION
      ! INDEX_FUNCTION
        ( .TEST(ARITH_DEPTH_COUNTER = 0)
          .SET(ARITH_OPERATOR_FOUND = "TRUE")
        ! .TRUE )
      ! LENGTH_FUNCTION
        ( .TEST(ARITH_DEPTH_COUNTER = 0)
          .SET(ARITH_OPERATOR_FOUND = "TRUE")
        ! .TRUE )
      ! .NUMBER
        ( .TEST(ARITH_DEPTH_COUNTER = 0)
          .SET(ARITH_OPERATOR_FOUND = "TRUE")
        ! .TRUE )
        .DO(COPY FCF_PREVIOUS_SYMBOL INTO TEMPORARY_STRING;)
      ! .IDENTIFIER
        INDIRECT_LOCATION_POINTER )
      ( .TEST(ARITH_VALUE_STORED = "FALSE")
        .DO(PUSH ** ONTO ARITHMETIC_STACK;)
      ! .TRUE    .SET(ARITH_VALUE_STORED = "FALSE") ) ;

ARRAY_ELEMENTS  :=
      "(" .DO(PUSH 1 ONTO SUBSCRIPT_COUNTER_STACK;)
      .SAVE_POINTER
      .INCREMENT(ARITH_DEPTH_COUNTER)
      ARITHMETIC_EXPRESSION    .ERROR(48)
      .DECREMENT(ARITH_DEPTH_COUNTER)
      .DO(SAVE COMPUTED_VALUE ON ARRAY_STACK;)
      .DO(POP TOP ELEMENT FROM ARITHMETIC_STACK;)
      $ ( ","  .DO(ADD 1 TO TOP ELEMENT OF SUBSCRIPT_COUNTER_STACK;)
          .INCREMENT(ARITH_DEPTH_COUNTER)
          ARITHMETIC_EXPRESSION    .ERROR(48)
          .DECREMENT(ARITH_DEPTH_COUNTER)
          .DO(SAVE COMPUTED_VALUE ON ARRAY_STACK;)
          .DO(POP TOP ELEMENT FROM ARITHMETIC_STACK;)
      ")"                .ERROR(28)
      .RESTORE_POINTER;
```

```
                /*****************************/
                /*  BOOLEAN EXPRESSIONS  */
                /*****************************/


BOOLEAN_EXPRESSION  :=
      .SET(PARENTHESIS_COUNT = 0)
      INPUT_FILE_PARSING_OPERATION
    ! BOOLEAN_STATEMENT
      .DO(POP TOP ELEMENT OF BOOLEAN_TRUTH_STACK AND SAVE
            AS OPERATIONS_TRUE_FALSE_INDICATOR;) ;


BOOLEAN_STATEMENT  :=
      BOOLEAN_PRIMARY
      $ (   "AND"  BOOLEAN_PRIMARY  .AND
        !   "OR"   BOOLEAN_PRIMARY  .OR   )



BOOLEAN_PRIMARY  :=
      "("  .INCREMENT(PARENTHESIS_COUNT)
      BOOLEAN_STATEMENT    .ERROR(32)
      ( .TEST(PARENTHESIS_COUNT = 0)
      ! .TRUE  ")"    .ERROR(28)
        .DECREMENT(PARENTHESIS_COUNT) )
    ! "NOT"  BOOLEAN_STATEMENT  .ERROR(32)  .NOT
    ! BOOLEAN_ELEMENT
      ( .TEST(RELATIONAL_EXPRESSION = "TRUE")
        ( "=" ! "<" ! ">" ! ">=" ! "<=" ! "<>" )
              .ERROR(11)
        .SET(RELATION_TYPE_FLAG = *)
        .SET(SAVED_VALUE = TEMPORARY_STRING)
        BOOLEAN_ELEMENT
        .SET(RELATIONAL_EXPRESSION = "FALSE")
        .BOOLEAN_COMPARE
      ! .TRUE )  .SAVE_TRUTH  ;

BOOLEAN_ELEMENT :=
      SYMBOL_EXISTS_STATEMENT
    ! TABLE_EMPTY_STATEMENT
    ! NODE_EXISTS_STATEMENT
    ! FIND_STATEMENT
    ! STRING_EXPRESSION
      .SET(RELATIONAL_EXPRESSION = "TRUE") ;
```

```
                /*************************/
                /*  STRING EXPRESSION    */
                /*************************/


STRING_EXPRESSION  :=
      VALUE_FUNCTION
      ( .PEEK("*" ! "+" ! "/" ! "-" ! "**")

        .SET(ARITH_ELEMENT_FOUND = "TRUE")
        ARITHMETIC_EXPRESSION
        .DO(POP TOP ELEMENT FROM ARITHMETIC_STACK;)
      ! .TRUE )
    ! SUBSTRING_FUNCTION
    ! (LENGTH_FUNCTION ! INDEX_FUNCTION)
      .SET(ARITH_ELEMENT_FOUND = "TRUE")
      ARITHMETIC_EXPRESSION
      .DO(POP TOP ELEMENT FROM ARITHMETIC_STACK;)
    ! .IDENTIFIER    .DO(COPY FCF_PREVIOUS_SYMBOL INTO TEMPORARY_STRING;)
      ( .TEST(RULE_TABLE_OR_SYMBOL_NAME = "TRUE")
        .SET(RULE_TABLE_OR_SYMBOL_NAME = "FALSE")
      ! INDIRECT_LOCATION_POINTER
        .SET(ARITH_ELEMENT_FOUND = "TRUE")
        ARITHMETIC_EXPRESSION
        .DO(POP TOP ELEMENT FROM ARITHMETIC_STACK;)
    ! .SET(RULE_TABLE_OR_SYMBOL_NAME = "FALSE")
      .STRING    .DO(PROCESS STRING IN FCF_PREVIOUS_SYMBOL TO REMOVE DELIMITERS
                     THEN COPY INTO TEMPORARY_STRING;)
    ! "*"  .DO(COPY INPUT_PREVIOUS_SYMBOL INTO TEMPORARY_STRING;)
    ! ARITHMETIC_EXPRESSION
      .DO(POP TOP ELEMENT FORM ARITHMETIC_STACK;) ;


                /*****************************/
                /*  INPUT FILE EXPRESSIONS   */
                /*****************************/


FIND_INPUT_NODE  :=
      .SET(BROTHER_NODE_END = "FALSE")
      .SET(FINISH_FLAG = "FALSE")
      ( "@"      /* USE CURRENT VALUE OF INPUT POINTER */
        .SET(OPERATION_TRUE_FALSE_INDICATOR = "TRUE")
        ( "."  .MOVE_INPUT_POINTER_DOWN
          .IDENTIFIER    .ERROR(54)
        ! .TRUE  .SET(FINISH_FLAG = "TRUE") )
      ! .IDENTIFIER
        .HOME_INPUT )
      $ ( .TEST(FINISH_FLAG="FALSE")
          FIND_NODE_THIS_LEVEL
          ( "."  .MOVE_INPUT_POINTER_DOWN
            .IDENTIFIER     .ERROR(54)
          ! .TRUE
            .SET(FINISH_FLAG="TRUE") ) ) ;
```

FLTOPS FUNCTIONAL DEFINITION GRAMMAR

```
INDIRECT_LOCATION_POINTER  :=
      .SAVE_POINTER
      ( .PEEK("(")
        .FIND_TABLE(**)
        ARRAY_ELEMENTS
        .DO(POP TOP ELEMENT FROM SUBSCRIPT_COUNTER_STACK AND
              SAVE AS STACK_SIZE;)
        .COMPUTE_TABLE_POSITION
        .FIND_BY_SUBSCRIPT(ARRAY_POSITION)
      ! .TRUE
        .FIND_TABLE(DEFAULT_SYMBOL_TABLE)
        .FIND_OR_ENTER_SYMBOL(**)  )
      .GET_VALUE(**)
      .SET(VALUE_LOCATION = SYMBOL_TABLE_POINTER)
      .RESTORE_POINTER ;
```

```
FIND_NODE_THIS_LEVEL  :=
      ( .TEST(FLTOPS FUNCTION = TEST OR FULL)
        .SET(FOUND_FLAG = "FALSE")
      ! .TRUE  .SET(FOUND_FLAG = "TRUE"))
      $ ( .TEST(FOUND_FLAG="FALSE")
          .PARSE_INPUT(*)
          ( .TEST(OPERATION_TRUE_FALSE_INDICATOR="FALSE")
            .ADVANCE_INPUT_POINTER
            ( .TEST(BROTHER_NODE_END = "TRUE")
              .SET(FOUND_FLAG="TRUE")
            ! .TRUE)
          ! .SET(FOUND_FLAG = "TRUE")
            .TRUE ) ) ;



                        /********************/
                        /* MISCELLANEOUS  */
                        /********************/


LEFT_PAREN  :=
      "("
    ! .TRUE   .MESSAGE(5) ;


RIGHT_PAREN :=
      ")"
    ! .TRUE   .MESSAGE(6) ;


COMMA  :=
      ","
    ! .TRUE    .MESSAGE(7) ;


LOCATION_POINTER  :=
      "@"
    ! "DEFAULT"
      .SET(RULE_TABLE_OR_SYMBOL_NAME = "TRUE")
      STRING_EXPRESSION  .ERROR(40)
      .FIND_TABLE(DEFAULT_SYMBOL_TABLE)
      .FIND_OR_ENTER_SYMBOL(**)
    ! .SET(RULE_TABLE_OR_SYMBOL_NAME = "TRUE")
      STRING_EXPRESSION  .FIND_TABLE(**)
      ( "," .SET(RULE_TABLE_OR_SYMBOL_NAME = "TRUE")
        STRING_EXPRESSION  .ERROR(40)  .FIND_OR_ENTER_SYMBOL(**)
      ! ARRAY_ELEMENTS
        .DO(POP TOP ELEMENT FROM SUBSCRIPT_COUNTER_STACK AND
              SAVE AS STACK_SIZE;)
        .COMPUTE_TABLE_POSITION
        .FIND_BY_SUBSCRIPT(ARRAY_POSITION)
      ! .TRUE   .MESSAGE(40) ) ;
```

6.3.1 Syntactic Specification of FCP Statement Types

DEFINITION OF SYMBOLS

$	= Repeat zero or more times
"$"	= The dollar sign is an actual character read from the FCP
!	= Logical 'OR' symbol
{...}	= Select one item within braces
[...]	= Items within brackets are optional
<...>	= Refers to another rule. Words or characters not appearing within <> and other than the special characters defined here, are read from the FCP.

SYNTACTIC DESCRIPTION

```
<stmt>          := ASSERT <bool_ex> [ERROR: <stmt>]

                := CATENATE(<location>,<string_ex> $[,<string_ex> [<format_st>]])

                := CLEAR_TABLE(<table>)

                := DECREMENT(<symbol>)

                := DEFINE <table><subscripts> $[,<table><subscripts>]

                := DEFINE <table> AS {<table> ! NULL}

                := DELETE(<table> [,<symbol> ! <subscripts>])

                := DELETE(<symbol>)

                := DO $<stmt> END

                := DO FOR EACH SUBNODE OF <input_tree> $<stmt> END

                := DO FOR SYMBOL TABLE <table> $<stmt> END

                := DO WHILE <bool_ex> $<stmt> END

                := ENTER(<location> [,<string_ex> [<format_st>]])

                := EXECUTE <id> [(<string> ! <arith_ex>} $[,[<string> !
                                  <arith_ex>]])

                := FIND(<input_tree>) [ERROR: <stmt>]

                := <fortran_stmt>

                := IF <bool_ex> THEN <stmt> [ELSE <stmt>]

                := INCREMENT(<symbol>)

                := INPUT(<input_grp> $[OR <input_grp>]) [ERROR: <stmt>]
```

```
                    := INVOKE <string_ex>
                    := MESSAGE([{I!G!INPUT!GENERAL},]<string> $[,
                       <string_ex> [<format_st>]])
                    := NODE_EXISTS(<input_tree>)  [ERROR: <stmt>]
                    := OUTPUT(<output_el> $[ , <output_el>])
                    := PARAMETERS <id> [(<type> $[,<type>])
                    := REPLACE(<location>,<string_ex>,<string_ex> [<format_st>]
                       [,<arith_ex>])
                    := RESTORE_POINTER
                    := SAVE_POINTER
                    := SET(<symbol> [= <string_ex> [<format_st>]])
                    := STATUS(<string> [(,<table> ! ,<table><subscripts> ! ,<table>
                            ,<symbol>)])
                    := SYMBOL_EXISTS([<table>,]<symbol>)  [ERROR: <stmt>]
                    := TABLE_EMPTY(<table>)  [ERROR: <stmt>]
<index_fn>          := INDEX(<string_ex>,<string_ex>)
<length_fn>         := LENGTH(<string_ex>)
<substring_fn>      := SUBSTRING(<string_ex>,<arith_ex> [,<arith_ex>])
<value_fn>          := VALUE((@ ! <symbol> ! <table><subscripts> ! <table>,<symbol>))
<table>             := <string_ex>
<symbol>            := <string_ex>
<location>          := @ ! <table><subscripts> ! <table>,<symbol> ! DEFAULT,<symbol>
<subscripts>        := (<arith_ex> $[,<arith_ex>])
<type>              := REAL ! INTEGER ! LOGICAL ! DOUBLE ! ALPHA
<input_tree>        := [@ . <id> ! <id>] $[.<id>]
<string_ex>         := * ! <value_fn> ! <id> ! <number> ! <string> ! <arith_ex>
                         ! <substring_fn>
<input_grp>         := <input_el> $[<input_el>]
<input_el>          := CONSTANT ! NUMBER ! SIGNED_NUMBER ! WORD ! STRING ! <string>
```

```
<arith_ex>      := <arith_el> $[(+|-|*|/|**) <arith_el>]
<arith_el>      := (<arith_ex>)
                := <value_fn> | <index_fn> | <length_fn> |
                   <number> | <id> | <id><subscripts>
<bool_ex>       := INPUT(<input_grp> $[OR <input_grp>] | <bool_stmt>
<bool_stmt>     := <bool_primary> $[ (AND | OR) <bool_primary>]
<bool_primary>  := (<bool_stmt>)
                := SYMBOL_EXISTS(<table>,<symbol>)
                := TABLE_EMPTY(<table>)
                := NODE_EXISTS(<input_tree>)
                := FIND(<input_tree>)
                := <relational_ex>
<relational_ex> := <string_ex> <relation> <string_ex>
<relation>      := > | = | < | <= | >= | <>
<output_el>     := <label> | (EOF | END_OF_FILE) | (EOR | END_OF_RECORD)
                   | (COMMENT: | C: ) | (COL | COLUMN) (<arith_ex>) |
                   <string_ex> [<format_st>]
<format_st>     := FORMAT(<format_type>) | => <format_type>
<label>         := "$"<number>
<id>            := <letter> $[(<letter> | <digit> | <attach_el>)]
                := #<id> | #<digit> $[(<digit> | <id>)]
<attach_el>     := _(<letter> | <digit>)
<string>        := '$<character>'
<number>        := [+|-] <digit> $[<digit>] [.$<digit> [<exponent>]]
                := [+|-] .<digit> $[<digit>] [<exponent>]
<exponent>      := (E | D) [+|-] <digit> $<digit>
<letter>        := A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U|V|W|X|Y|Z
<digit>         := 0|1|2|3|4|5|6|7|8|9
<character>     := <letter> | <digit> | <special_symbol>
<special_symbol>:= any character other than <letter> or <digit> including blank
<fortran_stmt>  := any line read from the FCF without a 9 in column 1
<format_type>   := any legal FORTRAN format type as per implementation
                   option.  Should include at least A,E,F,I format types.
```

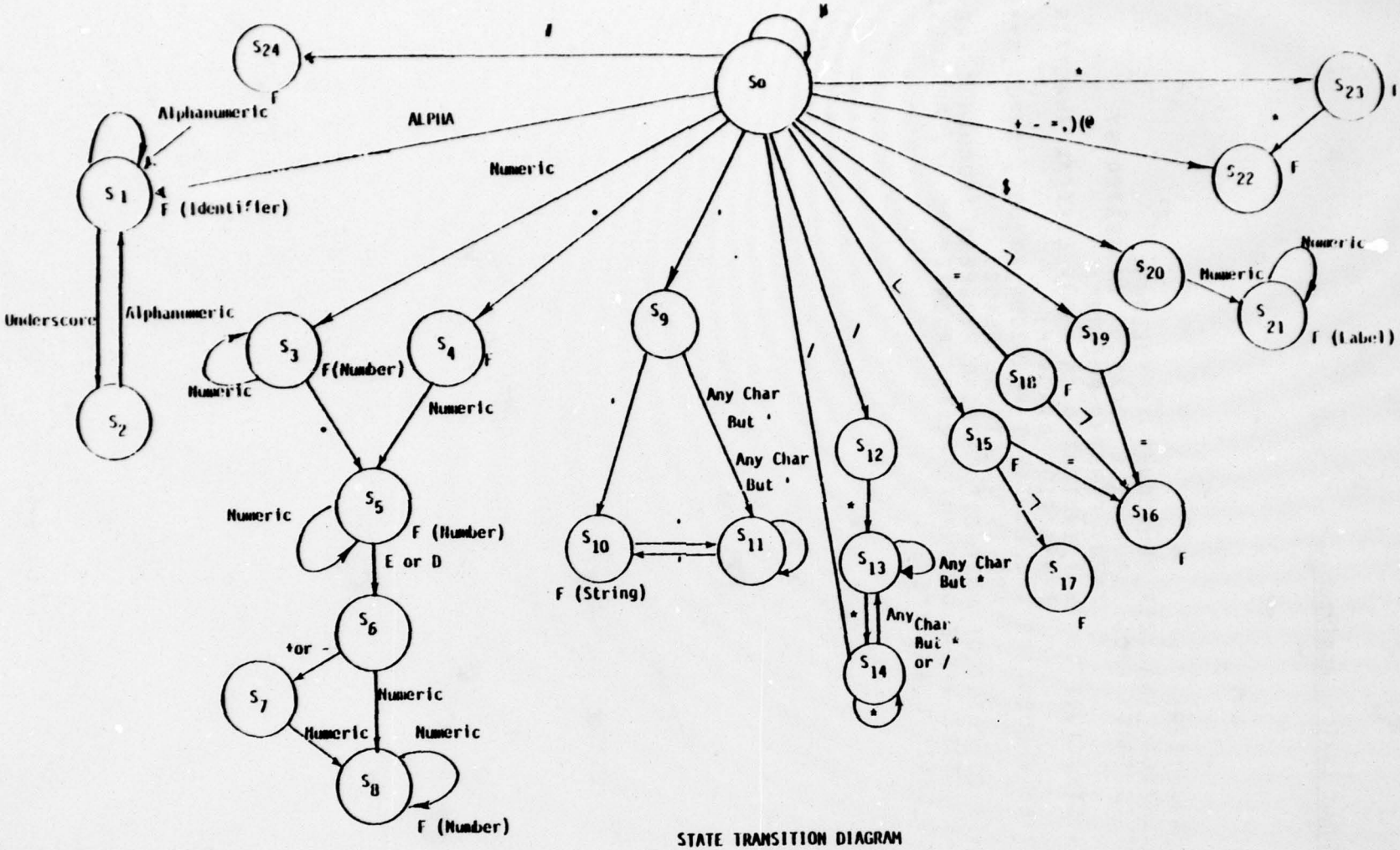
```
<letter>          := A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U|V|W|X|Y|Z

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

<character>       := <letter> | <digit> | <special_symbol>

<special_symbol>:= any character other than <letter> or <digit> including blank

<fortran_stmt>    := any line read from the FCF without a P in column 1

<format_type>     := any legal FORTRAN format type as per implementation
                     option.  Should include at least A,E,F,I format types.
```

6.4 SPECIFICATION OF LEXICAL ANALYZERS

```
LEXICAL_ANALYZER_PROCEDURE:
      INITIALIZE CURRENT_STATE TO ZERO;
      PREVIOUS_SYMBOL = SYMBOL;
      INITIALIZE SYMBOL TO NULL STRING;
      DO FOREVER;
         IF NEXT CHARACTER RESULTS IN LEGAL TRANSITION FROM CURRENT_STATE
            THEN DO;
               CATENATE CHARACTER ONTO SYMBOL;
               SET CURRENT_STATE TO NEW STATE;
               IF CURRENT_STATE = ZERO THEN RESET SYMBOL TO NULL;
               END;
         ELSE IF CURRENT_STATE IS LEGAL FINAL STATE
            THEN RETURN PREVIOUS_SYMBOL, SYMBOL, SYMBOL CLASS NAME;
         ELSE IF NEXT CHARACTER IS END OF FILE
            THEN DO;
               SYMBOL = "END_OF_FILE";
               SYMBOL_CLASS = NULL;
               END;
         ELSE DO;
            WRITE ERROR MESSAGE;
            IF SYMBOL IS STILL NULL THEN IGNORE NEXT CHARACTER
            RESTART LEXICAL_ANALYZER_PROCEDURE FROM TOP;
            END;
         END;
      END LEXICAL_ANALYZER_PROCEDURE;
```

6.4.1. FCF Lexical Analyzer

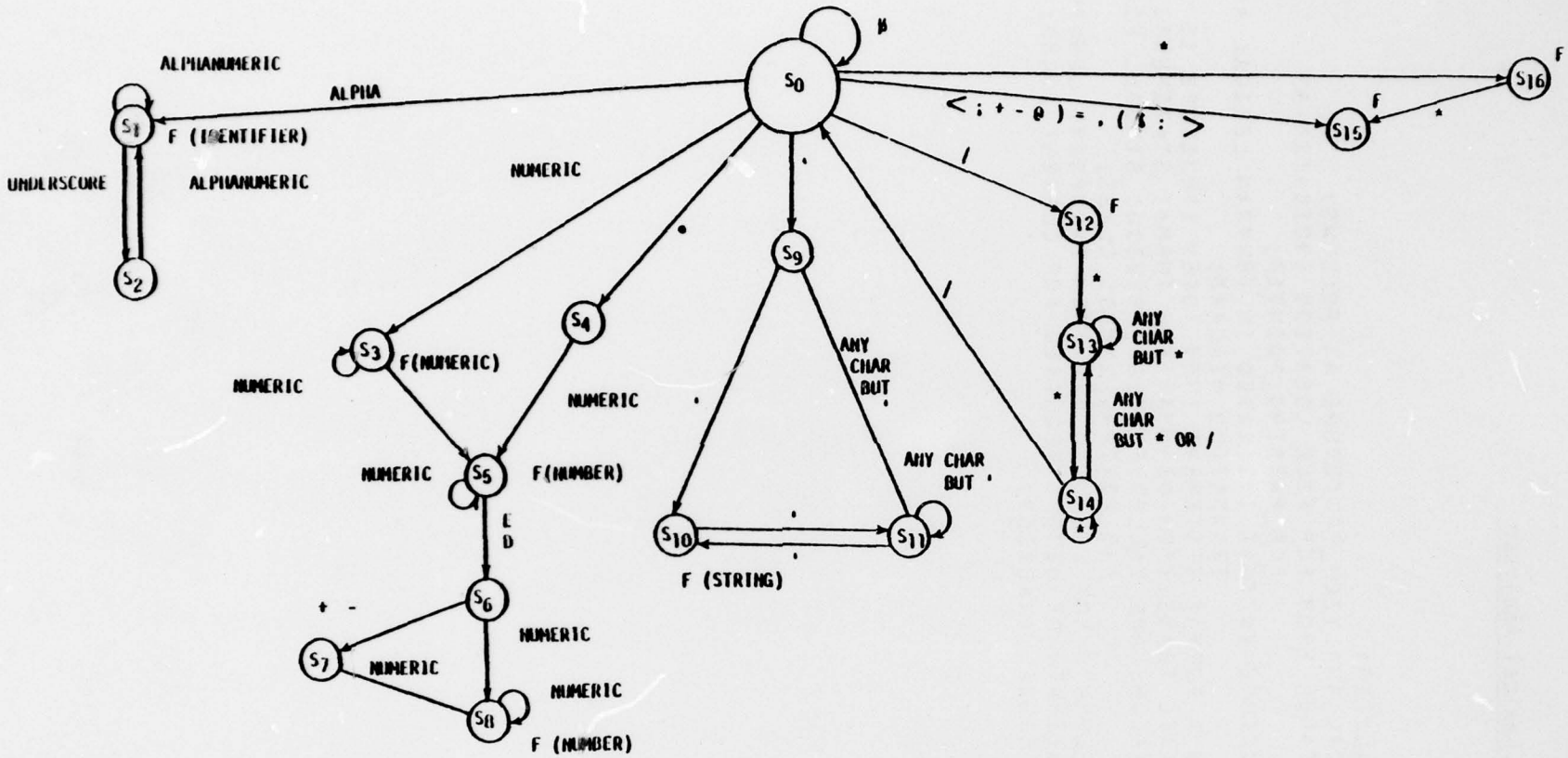
```
FCF_LEXICAL_ANALYZER:
      EXECUTE LEXICAL_ANALYZER_PROCEDURE AS FOLLOWS:
         INPUT IS TAKEN FROM FCF FILE LOCATION INDICATED BY
                              FCF_PARSING_POINTER;
         TOKEN STRUCTURE IS THAT INDICATED IN FCF LEXICAL ANALYZER STATE
                              TRANSITION DIAGRAM;
         RETURNED VALUES ARE CALLED FCF_PREVIOUS_SYMBOL, FCF_SYMBOL,
                              AND FCF_SYMBOL_CLASS;
         IF FCF_SYMBOL_CLASS IS FORTRAN THEN ENTIRE FORTRAN STATEMENT
              IS READ IN AS SINGLE TOKEN;
         IF TRACE = "HIGH" OR "HIGHER"
            THEN IF PRECOMPILER OPERATION IS INHIBITED
                    THEN ADD FCF_PREVIOUS_SYMBOL TO TRACE PRINT;
                    ELSE ADD INTENSIFIED FCF_PREVIOUS_SYMBOL TO TRACE PRINT;
      END FCF_LEXICAL_ANALYZER;
```



STATE TRANSITION DIAGRAM
FCF LEXICAL ANALYZER
(CONTROL STATEMENTS ONLY)

6.4.2. Input Lexical Analyzer

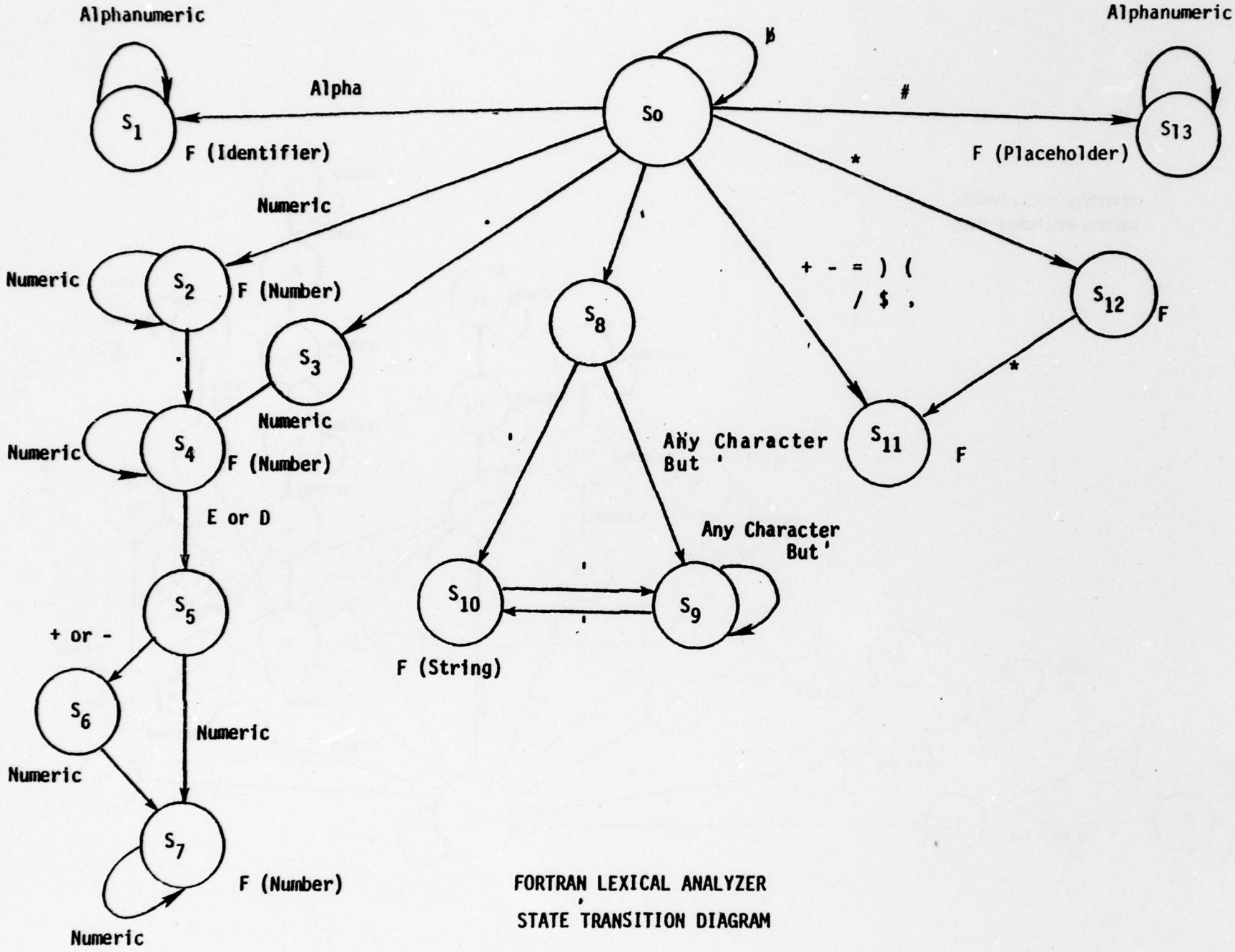
```
INPUT_LEXICAL_ANALYZER:
      EXECUTE_LEXICAL_ANALYZER_PROCEDURE AS FOLLOWS:
         INPUT IS TAKEN FROM INPUT FILE LOCATION INDICATED BY
                              INPUT_PARSING_POINTER;
         TOKEN STRUCTURE IS THAT INDICATED IN INPUT LEXICAL ANALYZER STATE
                              TRANSITION DIAGRAM;
         RETURNED VALUES ARE CALLED INPUT_PREVIOUS_SYMBOL, INPUT_SYMBOL,
                              AND INPUT_SYMBOL_CLASS;
         IF THIS IS POINT OF FARTHEST PARSING PROGRESS THROUGH CURRENT NODE
            THEN STORE VALUE OF INPUT_PARSING_POINTER AS NEW PROGRESS MARKER;
      END INPUT_LEXICAL_ANALYZER;
```



STATE TRANSITION DIAGRAM
INPUT LEXICAL ANALYZER

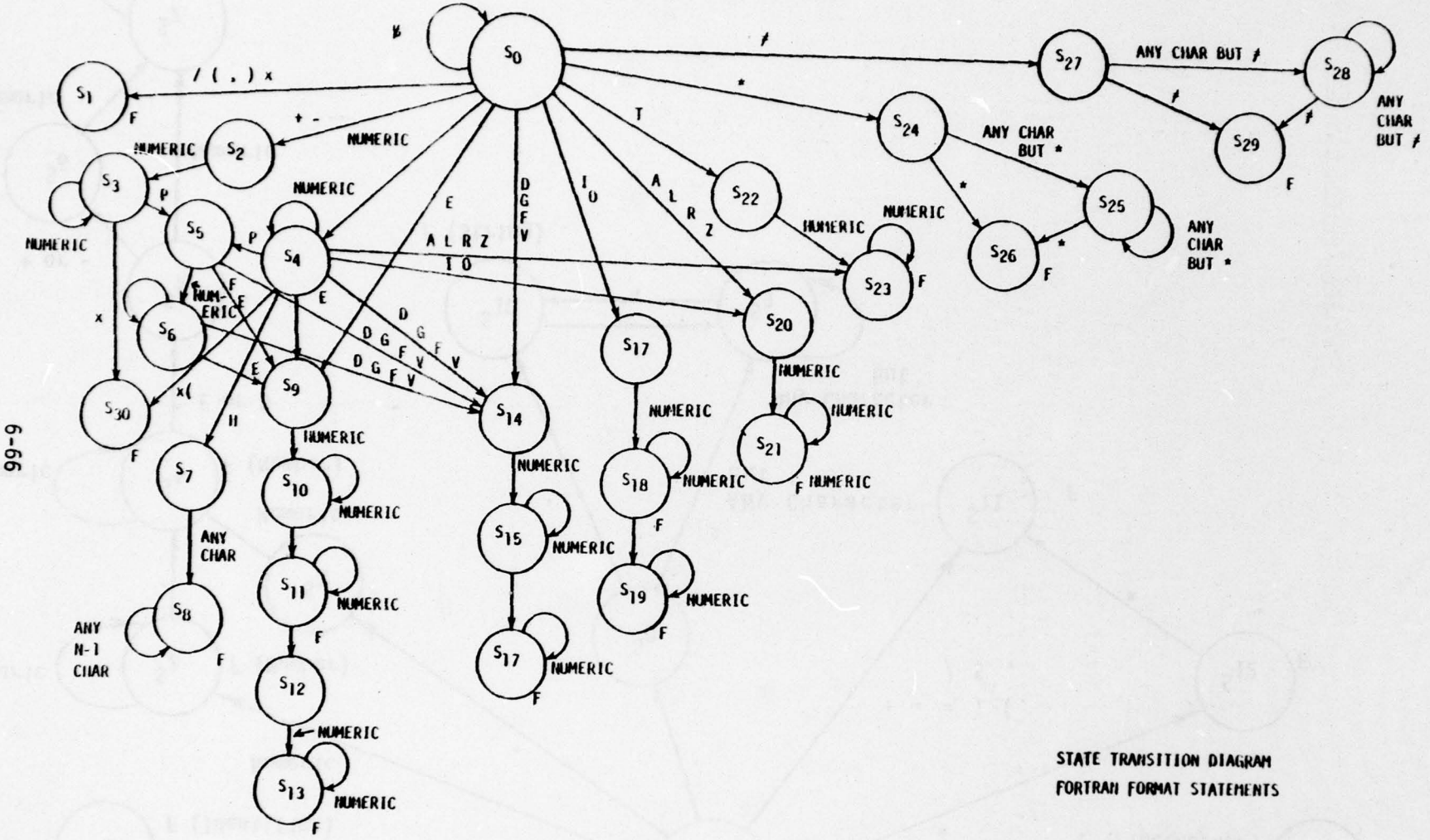
6.4.3. FORTRAN Lexical Analyzer

```
FORTRAN_LEXICAL_ANALYZER:
      EXECUTE LEXICAL_ANALYZER_PROCEDURE AS FOLLOWS:
         INPUT IS TAKEN FROM FCF FILE LOCATION INDICATED BY
                              FCF_PARSING_POINTER;
         TOKEN STRUCTURE IS THAT INDICATED IN FORTRAN LEXICAL ANALYZER STATE
                              TRANSITION DIAGRAM;
         IF THIS IS A FORMAT STATEMENT THEN TOKEN STRUCTURE IS THAT INDICATED
              IN STATE TRANSITION DIAGRAM FOR FORMAT STATEMENTS;
         RETURNED VALUES ARE CALLED FORTRAN_PREVIOUS_SYMBOL, FORTRAN_SYMBOL,
                              AND FORTRAN_SYMBOL_CLASS;
         IF FLTOPS LENGTH NOT EQUAL TO OFF THEN INCREMENT LENGTH
              BY ESTIMATE OF MEMORY REQUIRED FOR CURRENT TOKEN;
      END FORTRAN_LEXICAL_ANALYZER;
```



FORTRAN LEXICAL ANALYZER

STATE TRANSITION DIAGRAM



STATE TRANSITION DIAGRAM
FORTRAN FORMAT STATEMENTS

6.5 GLOSSARY

ALPHA: Default entry in the FLTOPS symbol table. If non-empty, contains format for alphanumeric conversion of data returned from execute modules. If empty, precompiler uses a default conversion procedure, the exact procedure being an implementation option. Initial value is null.

ARITH_DEPTH_COUNTER: Numeric quantity indicating nest level at which an arithmetic expression is being evaluated. Used to control switching of a flag to establish PARAMETER_LOCATION values when initiated by an EXECUTE statement.

ARITH_ELEMENT_FOUND: True/false flag indicating if an arithmetic expression consists of more than one token joined by a valid arithmetic operator.

ARITH_PAREN: Counter indicating the number of parentheses encountered embedded in an arithmetic expression. Used with PARENTHESIS_COUNT in assuring correct parsing of boolean/arithmetic expression combinations.

ARITH_VALUE_STORED: True/false indicator that identifies whether or not the current computed value in an arithmetic expression has already been pushed onto the ARITHMETIC_STACK. Used to insure the proper evaluation of boolean/arithmetic expression combinations.

ARITHMETIC_STACK: A precompiler stack containing those values to be used in evaluating an arithmetic expression.

ARRAY_POSITION: Number representing the relative location in a subscripted table that has been referenced. The number is computed using the subscripts in the current reference and the dimensions for the table that appeared in the last applicable DEFINE statement.

ARRAY_STACK: A precompiler stack containing the values of each subscript in an array reference. This stack is used with the dimension information from the DEFINE statement to computer ARRAY_POSITION.

BOOLEAN_TRUTH_STACK: A precompiler stack containing true/false elements that are used in computing boolean expressions. Also used in temporarily saving the results of IF conditional tests.

BROTHER_NODE_END: A true/false indicator which indicates if all subnodes in the input file at a given level have been found.

COMPUTE_RELATIVE_POSITION: True/false flag indicating whether or not array subscripts should be used in computing the relative value location with respect to predefined dimension values.

COMPUTED_VALUE: The numeric result of an arithmetic computation. Used in the computation of arithmetic expressions. TEMPORARY_STRING contains the equivalent string representation of this numeric quantity.

CONVERT_FLAG: In/out flag indicating whether a given quantity is to be converted to internal (IN) format or to external (OUT) string format.

CURRENT_TABLE_NAME: Name of the last symbol table reference by the FCF.

DEBUG: Default entry in the FLTOPS symbol table. On/off switch controlling the execution/bypassing of STATUS and ASSERT statements. Initial value is off.

DEFAULT_SYMBOL_TABLE: The table name with which non-subscripted variable names are associated in arithmetic expressions and within the SET statement. Initially set to FLTOPS.

DEFAULT_TABLE_STACK: Precompiler stack containing the names of all tables currently being used as DEFAULT_SYMBOL TABLES. The top element of the stack is the most recently defined default symbol table.

DEFINE_TABLE_NAME: Table name appearing in a DEFINE statement which is to be either dimensioned or equivalenced.

ERROR_MESSAGE_TABLE: A predefined table containing precompiler error messages. These error messages can be referenced by their order of occurrence within the table.

ERROR_NUMBER: A number which references the corresponding entry in the ERROR_MESSAGE_TABLE.

EQUIVALENCED_TABLE_NAME: The table name with which another table name is equivalenced in the DEFINE...AS... statement. This table may be non-empty at the time of equivalence or it may be previously equivalenced.

EXECUTE_DATA_AREA: The area shared by both an execute module and the precompiler in which data needed and/or computed by the execute module is stored.

EXECUTE_DATA_AREA_POINTER: A pointer indicating the location in the EXECUTE_DATA_AREA where the next data value is to be stored or retrieved.

EXECUTE_ENVIRONMENT_TABLE: An internal table maintained for every routine that appears in a PARAMETERS statement. For every parameter in the argument list a table entry is saved for PARAMETER_CLASS, PARAMETER_SIZE AND PARAMETER_LOCATION information.

EXECUTE_ROUTINE_NAME: A coded identifier consisting of a $ catenated with the routine name appearing in a PARAMETERS statement. It refers both to an EXECUTE_ENVIRONMENT_TABLE and without the $ to the execute module.

FATAL_ERROR: Any error from which any means of error recovery is impossible. Except for output of status information, all precompiler functions cease.

FCF_LEXICAL_ANALYZER: That portion of the precompiler which analyzes the FLTOPS configuration file input stream and identifies the individual tokens.

FCF_PARSING_POINTER: A pointer which identifies the location in the FCF at which token recognition will resume on the next invocation of the FCF_LEXICAL_ANALYZER.

FCF_PARSING_TRUE_FALSE_INDICATOR: Master precompiler control flag. Set to true, if the current syntactic element is satisfied. Thus, it is set to true if the token in FCF_SYMBOL matches the current syntactic element, or it can also be set directly by various functional primitives. Once set to false, the remaining mandatory syntactic and associated semantic constructs in a given rule are not processed.

FCF_PREVIOUS_SYMBOL: The last token from the FCF successfully recognized as a syntactic element in the grammar. Often references by the shorthand notation * (asterisk).

FCF_SYMBOL: The token just recognized by the FCF_LEXICAL_ANALYZER; the next token to be parsed by the precompiler.

FCF_SYMBOL_CLASS: The class of the token in FCF_SYMBOL. It may be a number, an identifier, etc.

FINISH_FLAG: True/false flag whether or not an entire input tree specification has been parsed. Also used to indicate whether or not all elements in the catenated form of the INPUT command have been pursed.

FLTOPS: A predefined symbol table available for all FCF users. It contains several predefined symbols that can be used to control the operation of the precompiler.

FLTOPS FUNCTION: A flag indicating which of four functional modes the precompiler is in:SCAN, SYNTAX, TEST or FULL. In SCAN the precompiler is conditionally bypassing a portion of the FCF and does only syntax checking; in SYNTAX, the precompiler will only do syntax checking, once set to SYNTAX it cannot be set to anything else. In TEST, the precompiler is fully operational except no FORTRAN output is generated. In FULL, all precompiler functions including FORTRAN output are operational. Except for SCAN, these precompiler functions are controllable by the user through the symbol FUNCTION in the FLTOPS symbol table.

FORTRAN_LEXICAL_ANALYZER: That portion of the precompiler which analyzes FORTRAN statements to identify the individual tokens.

FORTRAN_PARSING_POINTER: A pointer which identifies the location within a FORTRAN statement at which token recognition will resume on the next invocation of the FORTRAN_LEXICAL_ANALYZER.

FORTRAN_PREVIOUS_SYMBOL: The last token recognized in the current FORTRAN statement as returned by the lexical analyzer.

FORTRAN-SYMBOL: The token just recognized by the FORTRAN_LEXICAL_ANALYZER.

FORTRAN_SYMBOL_CLASS: The class of the token in FORTRAN_SYMBOL. Could be number, identifier, etc.

FOUND_FLAG: True/false flag indicating whether or not a specific node of an input tree actually was found in the input file.

HEADING: A string literal that is written out whenever the STATUS command ie executed.

INPUT_LEXICAL_ANALYZER: That portion of the precompiler which analyzes the input data stream to identify individual tokens.

INPUT_PARSING_POINTER: A pointer which identifies the location in the input file at which token recognition will resume on the next invocation of the input lexical analyzer.

INPUT_POINTER STACK: A precompiler stack containing values of the INPUT_PARSING_POINTER. This is the only precompiler stack directly accessible to the FCF builder through use of the SAVE_POINTER and RESTORE_POINTER commands.

INPUT_PREVIOUS_SYMBOL: The last token recognized as input by the FCF as returned by the INPUT_LEXICAL_ANALYZER.

INPUT_SYMBOL: The token just recognized by the INPUT_LEXICAL_ANALYZER.

INPUT_SYMBOL_CLASS: The class of the token in INPUT_SYMBOL. Could be identifier, number, etc.

INTEGER: Default entry in the FLTOPS symbol table. If non-empty, contains format for conversion of integer values returned from execute modules. If empty, precompiler uses a default conversion procedure. The exact procedure is an implementation option. Initial value is null.

KEY_WORD_TABLE: An internal table containing key words used in error recovery. Once a severe error occurs, the FCF is scanned for a key word. Once a key word is found, FCF parsing resumes at that location.

LBLNUM: Default entry in the FLTOPS symbol table. Used as the base value in generating statement lables. Initial value is 99000.

LENGTH: Default entry in the FLTOPS symbol table. If set to a number greater than or equal to zero, LENGTH is a counter incremented by the FORTRAN_LEXICAL_ANALYZER by fixed amount for each token required Initial value is off.

LOCATION_INFORMATION: A storage location containing sufficient information to indicate to the precompiler when scanning of the FCF, caused by a conditional test, should cease and regular processing (even if simply syntax checking) should resume.

LOGICAL: Default entry in the FLTOPS symbol table. If non-empty, contains the format for conversion of logical data returned from execute modules. If empty, the precompiler uses a default conversion procedure. The exact procedure is an implementation option. Initial value is null.

MESSAGE: String literal containing the message being output via the MESSAGE statement.

NOTE: A message of possible documentary or informative interest. It has no effect on precompiler functions.

NOTES: Default entry in the FLTOPS symbol table. Used as an on/off switch indicating whether or not note level messages are to be written out or suppressed respectively.

NOTE_COUNT: A number indicating how many times a note level message has been issued.

OPERATION_TRUE_FALSE_INDICATOR: True/false indicator identifying the success or failure of basic precompiler operations.

OPERATOR_STACK: A precompiler stack containing those arithmetic operators to be used in evaluating an arithmetic expression.

OUTPUT: Default entry in the FLTOPS symbol table. Switch controlling the format of the customized code produced. Initially set to FORTRAN but also can be set to DATA in which no special card image format is produced.

OUTPUT_BUFFER: Collection area where a FORTRAN statement is stored until a STATEMENT_TERMINATOR is encountered.

OUTPUT_COMMENT_CARD: True/false indicator set to true whenever a comment card is output via the OUTPUT command. Upon receipt of a statement terminator it is reset to false.

PARAMETER_CLASS: The type of value, i.e., REAL, INTEGER, etc., to be passed to an execute module through the EXECUTE_DATE_AREA.

PARAMETER_COUNT: A number identifying which parameter in an argument list is currently being processed.

PARAMETER_LOCATION: The symbol table location where the current parameter being processed can be found.

PARAMETER_SIZE: The number of values starting at the location identified by PARAMETER_LOCATION to be transferred into or restored from the EXECUTE_DATA_AREA.

PARENTHESIS_COUNT: A number used to keep track of parenthesis used in a boolean/arithmetic expression combination.

PASS1_OUTPUT_FILE: Temporary file where precompiler non-printing output is kept while the precompiler interprets the FCF.

PASS2_OUTPUT_FILE: FORTRAN output suitable for input to a compiler. Generated as the final product of a precompiler run.

PASS1_PASS2_SWITCH: Internal precompiler switch indicating whether or not this is the first or second pass. On Pass 1 the FCF is interprested and the PASS1_OUTPUT_FILE created. On Pass 2, the PASS1_OUTPUT_FILE is read and the PASS2_OUTPUT_FILE is written.

PASS1_TABLE: Symbol table used by the precompiler on Pass 1 substitution. Any FORTRAN token matching a symbol entry during Pass 1 is replaced by the corresponding table value.

PASS2_TABLE: Symbol table used by the precompiler on Pass 2 substitution. Any FORTRAN token matching a symbol entry during Pass 2 is replaced by the corresponding table value.

PROCEDURE_NAME_STACK: Precompiler stack containing all FCF rules (procedures) that are currently invoked.

REAL: Default entry in the FLTOPS symbol table. If non-empty, contains a format for the conversion of real data returned from an execute module. If empty, the precompiler uses a default conversion procedure. The exact procedure is an implementation option. Initial value is null.

RECOVERY_POINT_STACK: Precompiler stack containing FCF_PARSING_POINTER values. Used to identify that location to which control will be transferred upon completion of a loop or invoke construct.

RELATIONAL_EXPRESSION: True/false flag indicating whether or not the current element in a boolean expression will take the form of a relational expression.

RELATION_TYPE_FLAG: A flag indicating which relation type (e.g. ,>, =, etc.) to use in evaluating the truth or falsity of a relational expression.

SAVED_POINTER_STACK: Precompiler stack containing values of the SYMBOL_TABLE_POINTER. Used to save the location where a specified value is to be entered/retrieved.

SAVED_VALUE: Location containing the first of two values to be compared in a relational expression.

SEVERE_ERROR: An error such that the precompiler cannot perform its functions satisfactorily on the data received.

SEVERE_ERROR_COUNT: A number indicating how many times a SEVERE_ERROR has occurred. Once a severe error occurs in the FCF, all precompiler functions except syntax checking are shut down.

STACK_SIZE: A value indicating the number of values to be popped from ARRAY_STACK and used in subscript computations.

START_POS: Numeric quantity indicating the character position at which a substring begins. Used both in the INDEX and SUBSTRING statements.

STATEMENT_TERMINATOR: A unique character or group of characters which indicate when the OUTPUT_BUFFER is to be written.

STATS: Default entry in the FLTOPS symbol table. Used as an on/off switch indicating whether or not statistical information is to be computed and written out. Initial value is off.

STRING1: Temporary string variable used to store substring to be replaced in a REPLACE statement.

STRING2: Temporary string variable used to store the string to be inserted in another string with the REPLACE statement.

STRING_IN: Temporary string variable containing the string upon which the SUBSTRING function will operate.

STRING_LENGTH: Variable indicating both the length that a substring is to be in the SUBSTRING function and the length of an actual string in the LENGTH function.

SUBSCRIPT_COUNTER_STACK: A precompiler stack containing values, each referencing the number of subscripts in an array specification. A stack is necessary since array references can be nested, e.g., A(7,B(3,2)).

SYMBOL_TABLE_POINTER: An internal pointer indicating the symbol table location most recently referenced.

TABLE_HEADER: General purpose location for each table. May contain a variety of information useful to managing table entries. Always contains dimension information for those tables appearing in a DEFINE statement. Subscripts are assumed stored in the order of their appearance.

TEMPORARY_STRING: All purpose string variable -- often referred to by the shorthand notation ** (double asterisk).

TRACE: Default entry in the FLTOPS symbol table. Used as an on/off switch controlling the generation of trace information. Initial value is off.

VALUE_LOCATION: The symbol table location from which the last table value was retrieved.

WARNING: An error which most likely is an unintended user mistake, but which does not confuse the semantic intent of the statement.

WARNING_COUNT: A number indicating how many times a WARNING message has been issued.

XREF: Default entry in the FLTOPS symbol table. Used as an on/off switch indicating whether or not a cross reference map of the FCF should be generated. Initial value is off.

7.0 IMPLEMENTATION ISSUES

Inherent in any design are areas which must be paid special attention when implementation begins. In the FLTOPS precompiler design there are several of these special areas. The method by which symbol table storage is accessed and the means by which the execute command is implemented can both have a tremendous impact on the efficiency of the precompiler. Depending on the actual file structure adopted for input and FCF data, file preprocessing may also be an important consideration for the implementor. Although there can be little actually done about the system and language which the implementor uses, these, too, can have an important effect on the implementation effort. Although incomplete in many ways, the following sections provide some considerations for the implementor in each of the aforementioned areas.

7.1 SYMBOL TABLE IMPLEMENTATION OPTIONS

The symbol table operations as specified in the FLTOPS design are not assumed to be ordered or partitioned in any particular way for purposes of execution efficiency, nor is any particular (non-sequential) search method assumed. However, the implementor should keep in mind that data storage/retrieval times are one of the primary factors in precompiler run time. As a result, the implementor is expected to select a highly efficient approach to symbol table operations, based upon considerations of design, usage and the implementation environment. For example, table and symbol entries might actually be structured in the form of binary trees for efficient search (see Wegner, 1968, p. 168). Table names, symbol names, and values are not restricted in length, as such an efficient means of allocating new and releasing old storage is a must. One of several approaches the implementor might take is the "buddy system" (see Knuth, 1968).

Since there are two distinct methods of referencing symbol table storage as outlined in the design specification, the implementor might want to specify different mechanisms for accessing array tables than for regular symbol tables. This may not be necessary, however, especially if sparse matrix storage/retrieval techniques prove to be applicable.

Obviously, there are many more options available to the implementor than listed here. The key concept, however, is to implement symbol table access in a highly efficient manner. Care taken in this area could result in a highly efficient precompiler.

7.2 ALTERNATIVES FOR IMPLEMENTING THE "EXECUTE" STATEMENT

The FLTOPS precompiler design provides the capability of interfacing with already compiled program modules. How this interface actually occurs can be extremely important in determining precompiler efficiency and the time and effort required for actual implementation. Although other schemes might be found in actual implementation, three procedures have been identified to date. All can utilize the design methodology in which values are shared between programs in a known common area. This common area may be actual memory locations or an external medium such as a disk file.

The first workable procedure is through use of the system job control language. Essentially, the precompiler and execute routine would be separate and distinct jobs. When the precompiler encounters the EXECUTE command, it suspends processing after loading the common area with the designated values, and enough information is saved so that the precompiler can be restarted properly. The precompiler sets a switch which can be accessed via the job control stream. The switch setting indicates which execute routine should be loaded and executed. After completion of this routine, the precompiler is restarted at the appropriate program step and the values computed by the execute routine are entered into the associated symbol table locations.

Notice that there are several requirements for this procedure to work. The first requirement is a high level job control language that has the capability to repeatedly, as well as conditionally execute job control statements. Additionally, the capability to set a JCL accessible switch from inside a program is essential. Although it appears that there are various JCL's in existence that could do this (e.g., some versions of KRONOS or NOS-BE for CDC systems), it is not a universal capability that most computer vendors supply. As such, the procedure might have to be entirely redone, instead of simply being modified, if computer systems were changed. In addition, the job control decks needed for any specified FCF might be very long and complicated to construct. It has not as yet been determined that the operating system currently installed on AFFTC computers can perform all necessary JCL variations.

An alternative procedure is to cause the system to load the execute routine into memory either at an absolute location or as part of the precompiler. Each time a new routine is to be executed, it would replace the one currently in memory. A simple way to think of this would be as an emulation of an overlay procedure. Once loaded, the execute module would process the data and transfer control back to the precompiler.

Notice that the requirements for using this kind of system are simply a process to load the correct module into the designated location in memory and a procedure to transfer control properly. Although few in number, the requirements could be much harder to implement. To actually load a module into core is a complicated process. It could be as simply as calling the system load routine from the precompiler or could be as difficult as writing a specialized load routine. In either case, some expert knowledge of the operating system, compiler table structures, and loading procedures is necessary. Even more than with the JCL procedure mentioned earlier, this procedure is very dependent upon the system used. However, once working, this could be very efficient.

A third alternative combines simplicity of operation with transportability. In this procedure, all execute routines for a particular FCF are linked together with the precompiler in an overlay structure. Each execute routine would be overlayed at the same level. A special subroutine would be written to control the calling of the separate execute routine overlays. The necessary data (e.g., the overlay file name and overlay level numbers) controlling the overlay calling would be a part of this subroutine would be written to control the calling of the separate execute routine overlays. The necessary data (e.g., the overlay file name and overlay level numbers) controlling the overlay calling would be a part of this subroutine. It is expected that a special subroutine would have to be written for each separate FCF. Each time a new execute module were added, a new overlay with the precompiler would have to be built. Although every system has its own overlay structure, this concept is sufficiently general to accommodate them all. A particularly nice feature of CDC overlay structures is the capability to save separate overlay levels on independent files if desired.

In this application, each separate FCF could have its own execute library that could be overlayed with a single copy of the precompiler overlay.

Although the details are not completely worked out, the third procedure seems the most favorable implemention mechanism at this time. It, too, is somewhat system specific, but is easier to implement and use than either of the other two. The implementor in any case should endeavor to provide a procedure that minimizes the effect of possible system changes and yet remains easy to use, and, if possible, to implement.

7.3 FILE PREPROCESSING

Depending on the file structure adopted at implementation time, it might be desirable to preprocess the input and FCF files to make direct access possible to the various segments of each file. Preprocessing of the input file has already been flagged as a part of the design. This is so because of the hierarchic nature of the FCF inputs and the nature of various input commands.

Not actually mentioned in the design, but clearly an issue for the implementor, is preprocessing of the FCF and the FCF users library so that direct access to each rule is possible. Direct access techniques would enable the FCF to transfer parsing control among routines without seriously degrading precompiler execution times. If forced to scan the FCF or users library sequentially in search of the rule to receive parsing control, a severe increase in I/O and processing times could be expected for even moderately large FCF files. As such, the FCF user would be forced to place a premium on file organization. With the increased file management support offered by most major computer vendors, it is entirely possible that no specialized software need be written to accomplish direct access to each FCF rule. Instead, the FCF could be "managed" by system software. Before using any such standard package, however, the implementor should weigh the impact of its use against possible effects caused by future system changes. In any case, the implementor should be aware of the desirability of direct access in both the input and FCF files and provide a scheme that accomplishes this aim without severely restricting precompiler portability.

7.4 LANGUAGE AND SYSTEM CONSIDERATIONS

The language and system that the FLTOPS design is initially implemented on will have a direct affect on some aspects of the implementation. FORTRAN is the designated target language for FLTOPS implementation. Because of some symbol table operations and the ability to use arithmetic expressions, string manipulation and data conversion capabilities are required aspects of the design implementation. ANSI standard FORTRAN does not define as a standard language element the capability to do either of the above. As a result, the way each can be done varies with the FORTRAN provided by the vendor. Most vendors provide intrinsic functions callable from FORTRAN which can be used directly, or in some combination, to provide the above functions. For example, ENCODE and DECODE statements can be used in some FORTRAN compilers for data conversions. Such statements are not available with, among others, the IBM FORTRAN H compiler. Functions which allow masking, shifting and boolean operations are also supplied with many compilers. However, the form of these functions is very rarely compatible. As a result, the implementor should make judicious use of any compiler specific functions. When used, they should be adequately documented and structured so as to make future changes relatively straightforward.

The operating system on which the implementation is done will unavoidably affect some aspects of the implementation. Although the design requires no system peculiarities, the implementor is cautioned that in instances where system peculiarities exist that would enhance implementation efforts, use of these alternatives should occur only after properly weighing the downstream effect of future, possibly adverse, system changes. For example, if the decision were made to use the specialized features of a JCL to implement EXECUTE commands (see Section 7.2), a change to a new computer system without that feature would result in a demand for a new implementation procedure. Thus, the EXECUTE command would need to be reimplemented. Some small changes are to be expected when installing any large program on a different system. However, the implementor can cause the

number of changes to be kept to a minimum by paying close attention to all implementation alternatives available and choosing judiciously.

8.0 REFERENCES

Knuth, D. E. The Art of Computer Programming. Vol. 1. Addison Wesley Publishing Company, Inc., Reading, Massachusetts, 1968.

Naur, P. (Ed.) Report on the Algorithmic Language ALGOL60. Communications of the ACM, 1960. 3, 299-314.

Wegner, P. Programming Languages, Information Structures and Machine Organization. McGraw-Hill, New York, 1968.

APPENDIX A

FLTOPS DESIGN VALIDATION STUDY

A.1 INTRODUCTION

The FLTOPS design as originally conceived provided a formal, complete and rigorous specification of the actual system proposed for implementation. However, the uniqueness of the concept combined with the necessary complexity of the design specification made it extremely difficult to verify by conventional means its logical correctness and completeness. Yet, it was determined that just such a verification and validation was needed to prove the soundness and usefulness of the concept. This need, combined with the existence of some unique software previously written by SAI staff members, provided the incentive and mechanism for this effort. The validation effort took the form of an executing functional prototype which served to:

- validate that the functional design specified was an integrated and complete design;
- provide a means for AFFTC to evaluate the utility and potential benefits of the FLTOPS design in a set of realistic AFFTC scenarios;
- provide a means for demonstrating and communicating the FLTOPS concept that was less sterile than formal specification documents or technical briefings;
- provide a means for transferring technical familiarity from the designers to the potential users of FLTOPS;
- provide a means for discovering potential implementation cost drivers before implementation commitments are made;
- develop the insights necessary to estimate the scope of making a transition from existing source code libraries to quasi-source code libraries.

Under certain circumstances the costs of such a functional prototype would be prohibitively high. However, the requirements of FLTOPS and the choice of design specification mechanisms were compatible with the capabilities of the Translater Writing System, developed for NASA by members of the SAI

staff. This combination provided a unique opportunity for the Air Force to set a hands-on capability and served to provide a continuing medium for teaching and evaluation before actual FLTOPS implementation is begun.

Although the end product of this validation effort is a functional prototype which actually performs the FLTOPS operations, it cannot perform efficiently or realistically enough in this form to be considered an implemented FLTOPS. The "quick and dirty" approach served the purpose of validating the design, but cannot satisfy actual implementation requirements where memory utilization and processing efficiency are key requirements.

The validation effort consisted of three basic tasks:

- develop a set of executable procedures that perform the FLTOPS functions as described by the functional primitives presented in section 6.
- adapt the augmented grammar presented in section 6 into a form compatible with the Translator Writing System. Execute the TWS with this adapted grammar as input to produce an executable program that serves as the FLTOPS statement processor.
- combine the first two activities into an integrated functional FLTOPS that is capable of processing actual FCF. Included in this task was the development of several test FCF's that would test the logical and functional adequacy of both the FLTOPS primitives and augmented grammar specifications.

A discussion of these tasks is provided below. In addition, the concluding section in this appendix details the major changes in the design that resulted from this effort.

A-2 DEVELOPMENT OF EXECUTABLE CODE THAT PERFORMS THE FLTOPS FUNCTIONS

All code development for FLTOPS was done in the PL/1 programming language. This was so for two basic reasons:

1) the PDL used in the specification mechanisms is similar in many respects to PL/1 code. As a result, transformation of the PDL was much simplified.
2) the output of the TWS is PL/1 code. Since the statement processor was generated in PL/1 code, efficiency and compatability demanded the primitives be similarly coded.

Each primitive was coded as a separate PL/1 procedure and was tested for functional adequacy in its performance before being integrated with the other primitives into the functional prototype. Such off line testing eliminated many deficiencies in the program at the outset.

Although all primitives were coded, some simplifying assumptions were made with respect to symbol table access, the execute related primitives, as well as the input/FCF file structures so as to hasten the completion of a working emulator. This "quick and dirty" approach was justified because of the goal of the validation effort, i.e. to test the FLTOPS design for deficiencies, not provide an implemented system. As a result; sequential searches are made through each table for the various entries; array tables are handled exactly as are symbol tables; table entries are limited to 50 characters; the use of comment cards in input and FCF files are mildly restricted; only medium size FCF ($<$ 2000 lines) files are permitted; execute modules can be run, but must be in PL/1 and linked at the same time as the rest of the emulator; there can be no libraries of FCF code nor can FCF rules appear in the input file.

Although these and other deficiences exist that would detract from an implemented system, the emulator does functionally perform as prescribed by the design for those input/FCF files that conform to this simplified format.

Very few changes were made in the functional primitives from that given in the previous draft of this document. Several new primitives were added, however, to perform the functions required by new capabilities added during the validation study.

A-3 DEVELOPMENT OF THE STATEMENT PROCESSOR THROUGH USE OF THE TRANSLATOR WRITING SYSTEM

Central to the FLTOPS design is a statement processor that functions as the central control mechanism for the entire precompiler system. The augmented grammar specification rigorously defines the interrelationships among the various elements of the system, i.e. which and when primitives are to be executed for any of the control statements in the FCF language.

Providing an executable program that performed the indicated tasks could be time consumming if it were to be developed wholly from this initial state. The existence of some specialized software greatly simplified and shortened this development effort. The Translator Writing System (TWS) produces PL/1 programs from inputs that appear in an augmented grammar format similar to that used in the FLTOPS specification. It was a simple, straight-forward procedure to modify the FLTOPS specification grammar into a form compatible with the input format of the TWS.

The use of the TWS in the development of the functional prototype of FLTOPS provided several distinct advantages. It was very fast, efficient and relatively inexpensive to employ. The format of the augmented grammar lends itself to rapid, modular enhancements. These were then easily accommodated for by simply rerunning the TWS with the new modified grammar specification. As a result, there was a one to one correspondence with changes in the grammar specification and the statement processor, produced by the TWS, that controlled the precompiler functions. This fast and easy update mechanism was invaluable in providing reliable code that performed the desired functions. An additional advantage of this procedure was the

ease with which errors could be traced and corrected. Once learned, the reading of the augmented grammar specification provides a logically compact, easily read format for describing system functions. Processing errors could then be more easily found with reference to the grammar specification than the resultant PL/1 code generated by the TWS. This fact combined with the proven reliability of the TWS in its code generation enabled the location of errors in system processing logic to be found and corrected sooner than would have been possible through conventional debugging of the PL/1 code generated by the TWS. The impact of this symbiotic effect was most easily seen when contrasted with the debugging procedures necessary in perfecting the procedures written for each of the primitives. Even though changes to the grammar far exceeded those changes made to the primitives, total debugging effort was about equal. Such performance enhancement has established from a functional standpoint the usefulness of such software technology as the TWS in system validation efforts like this.

Many minor and a few major logical errors to the original grammar specification were uncovered as a result of this effort. In addition, several enhanced capabilities were introduced to provide the system with greater flexibility and usefulness to its eventual benefactors. Each change was made directly to the grammar and reflected in the functional prototype by use of the TWS. The combination of the statement processor generated by the TWS from the augmented grammar specification with the procedures developed for each of the functional primitives provided an integrated program that emulated actual FLTOPS operation.

A-4 DEVELOPMENT AND EXECUTION OF FLTOPS TEST CASES

Once an integrated functional prototype of the FLTOPS emulator was available, several tests were formulated to be executed by the emulator. The goal of these test scenarios was not to test the coding efficiency or accuracy of the emulator, but to test the FLTOPS design itself.

Each test case was evaluated for proper functional behavior of the emulated precomplier and with respect to the useability of the FLTOPS control language with respect to its satisfaction of user requirements. In addition to those tests created by the contractor, several "real world" tests were created by the programming staff at AFFTC. Although these test cases themselves proved valuable in testing the design, of more practical and immediate importance was the hands on experience they provided the eventual users of the system. As the programmers involved tried to use the control language to accomplish their desired goals, they were able to provide a valuable interchange of ideas that resulted in several important enhancements to the design. The coupling of the feedback resulting from actual user hands on experience with the easily modified nature of the augmented grammar, provided an ideal mechanism for insuring that the design was a feasible and useful expression of user requirements. The actual results of the test cases provided by AFFTC are provided to the individuals involved.

To validate the design, several test scenarios were developed. Each utilized to some extent the input file/FCF combination shown in Figure A-1. Each of the tests was designed to exercise a certain subset of the FLTOPS control language. Those elements of the language added as a result of this effort may not be completely tested although all have been included in the validation package. Figures A-2 through A-8 exhibit the test cases as actually executed. The following paragraphs briefly describe some of the salient features of each example. The changes that were made as a result of these tests and those enhancements made due to perceived inadequacies in the original design are given in the following section.

Test cases A-2, A-3 exercise symbol table operations including the use of tables defined as arrays. Note that sophisticated usage of indirect pointers as well as expected table manipulations were tested. Most table operations were exercised in some fashion in these tests. Time did not permit exhaustive testing of all possible combinations permissible in FLTOPS symbol table operations. All table operations tested performed according to specification requirements as appear in the body of this report.

```
CONSTANTS
   HAV = 23000
GEOPHYSICAL_MODEL
   EARTH
      SPHERICAL
      NON_ROTATING
   GRAVITY
      VARIABLE
REQUIRED_COMPUTATIONS
   ENERGY_HEIGHT
INTERPOLATION_ROUTINE = POLY
   TABLE_INFORMATION
      ROWS = 5
      DATA =  0. 26.3   /* COMMENT */
              5. 69.8
             11. 76.3
           17.5 89.1
           20.  99.7
   INTERPOLATION_VALUES = 1.63,2.1,6.2,11.6,19.3
STRING  'STRING OF CHARS +-%&/(*:$,.'
SIGNED_NUMBER = -39.5
SPECIAL_SYMBOLS  * + - , ( )
EQUATION = (7.+5.*SQRT(Y/2.))**(5.3+Z)-(A(7,3)+B(ALT(3),K1)*2.)
DO_INPUT_TEST
   STRING  'THIS IS A STRING'
   WORD    SYMBOL_TABLE_ENTRY
   SPECIAL_SYMBOLS  '+-(*)**/'
      PLUS  +
      MINUS  -
      L_PAR  (
      MULT   *
      R_PAR  )
      EXP   **
      DIVIDE /
   NUMBER  97.967845
   SIGNED_NUMBER  -89.361
FCF:
P MAIN:
P     SET(DEBUG = 'ON')
P     SET(TRACE = 'HIGHER')
P     SET(NOTES = 'ON')
P     STATUS('FLTOPS STATUS AT START OF TEST')
$$$$$ INSERT TEST FCF HERE
P     MESSAGE(G,'N:END OF FCF TEST PROCEDURE')
P     END MAIN
```

BASIC INPUT/FCF FILES FOR VALIDATION TESTS

FIGURE A-1

```
P     MESSAGE(G,'N:START OF SYMBOL TABLE TEST')
P     DO FOR SYMBOL TABLE SYM_TAB
P        SET(N = 'TABLE ENTRY')
P        STATUS('INITIAL VALUE OF N ',SYM_TAB,N)
P        DELETE(N)
P        STATUS('N AFTER DELETE',SYM_TAB,N)
P        SET(N='TABLE VALUE')
P        SET(M='SYMBOL ENTRY')
P        SET(X = 'IN SYM_TAB')
P        STATUS('TABLE AFTER INITIAL SETUP',SYM_TAB)
P        REPLACE(SYM_TAB,N,'ENTRY',SUBSTRING(VALUE(SYM_TAB,X),5,7))
P        STATUS('N AFTER REPLACE WITH PART OF X',SYM_TAB,N)
P        CATENATE(SYM_TAB,M,VALUE(X))
P        STATUS('M AFTER CATENATE OF X',SYM_TAB,M)
P        REPLACE(SYM_TAB,M,VALUE(X),'')
P        STATUS('M AFTER REMOVAL OF X',SYM_TAB,M)
P        SET(M)
P        REPLACE(@,'','A ')
P        STATUS('M AFTER PREFIX A ADDED',SYM_TAB,M)
P        DELETE(X)
P        DELETE(SYM_TAB,N)
P        STATUS('SYM_TAB AFTER X AND N DELETED',SYM_TAB)
P        SET(M = (7.3+2.7)**2)
P        SET(N = 5.*M)
P        SET(A = 'N')
P        STATUS('SYM_TAB AFTER M,N,A SET',SYM_TAB)
P        SET(VALUE(A) = (20+10)*5+10.**2)
P        STATUS('SYM_TAB AFTER INDIRECT ENTER',SYM_TAB)
P        SET(D = 'A')
P        SET(VALUE(VALUE(D)))
P        ENTER(@,0)
P        STATUS('SYM_TAB AFTER DOUBLE INDIRECT SET',SYM_TAB)
P        END
P     SET(I=1)
P     DO WHILE I<=5
P        INCREMENT(I)
P        END
P     STATUS('VALUE OF I AFTER INCREMENT',FLTOPS,I)
P     DO WHILE I>0
P        DECREMENT(I)
P        END
P     STATUS('VALUE OF I AFTER DECREMENT',FLTOPS,I)
P     MESSAGE(G,'N:TEMPORARY END OF SYMBOL TABLE TEST')
```

FCF FOR SYMBOL TABLE OPERATIONS

FIGURE A-2

```
P     MESSAGE(G,'N:START OF ARRAY TABLE TEST')
P     SET(N=5)
P     DEFINE TABLE(4,5),ARRAY(N),TAB2(1,1)
P        /* NOW USE THESE TABLES IN VARIOUS WAYS */
P     ENTER(TAB2(1,1),97.5)
P     ENTER(ARRAY(1),'N')
P     SET(VALUE(ARRAY(1)) = 20)
P     SET(I=1)
P     DO WHILE I <= 4
P        SET(J=1)
P        DO WHILE J<=5
P           ENTER(TABLE(I,J),I*J)
P           INCREMENT(J)
P           END
P        INCREMENT(I)
P        END
P     STATUS('TABLE OUTPUT AFTER TABLE SETUP')
P     IF TABLE(3,5) > 0 THEN ENTER(ARRAY(2),TABLE(3,5)*2)
P     ELSE ENTER(ARRAY(2),2)
P     ENTER(ARRAY(3),2)
P     ENTER(ARRAY(4),3)
P     IF TABLE(ARRAY(3),3) = TABLE(ARRAY(3),ARRAY(4)) THEN
P          ENTER(TABLE(ARRAY(3),ARRAY(4)),TABLE(4,5)*3+N)
P     ELSE MESSAGE(G,'W:THIS MESSAGE SHOULD NOT BE PRINTED')
P     CATENATE(ARRAY(1),'EW ENTRY')
P     SET(N = ARRAY(3)**ARRAY(4)+TABLE(2,5)*TABLE(3,3)/TABLE(1,4))
P     IF TABLE(1,2) > ARRAY(3) THEN ENTER(ARRAY(5),TABLE(2,4))
P     ELSE ENTER(ARRAY(5),2.0*TABLE(2,4))
P     STATUS('SYMBOL TABLES AFTER SOME CHANGES MADE')
P     MESSAGE(G,'N:END OF ARRAY TEST')
```

FCF FOR ARRAY TABLE CONSTRUCTS

FIGURE A-3

Figure A-4 shows the test used to validate those commands related to DO block usage. All DO types were exercised in various combinations, including nested DO blocks and DO blocks that conditionally should be skipped. As in all tests the presence of MESSAGE and STATUS statements monitor the progress of the test. This test was successfully executed with requirements as specified in the body of this report.

Figure A-5 shows the test scenario used in validating the proper evaluation of IF...THEN...ELSE constructs. Since these can be among the most complicated in the language, a fairly complicated test was constructed to ennumerate many of the possible constructs. Both complicated boolean constructs and boolean constructs with imbedded arithmetic expressions were tested. Proper evaluation of all tested constructs was achieved.

The evaluation of input file commands is shown in the test cases presented in figures A-6 and A-7. Various legal combinations for both the FIND and INPUT statements were tested, all proving eventually successful. Of special note in figure A-7 is the test of the INVOKE statement. Two rules were written which together would read in any legal FORTRAN arithmetic expression and save it in the preset location. The rules A_EXPR and A_TERM are recursive, a legal construct in the FLTOPS control language, and as such prove to be a comprehensive test of invocation handling by the precompiler. In addition, the rules themselves demonstrate some uses of the INPUT statement, both conditional and required. These tests, too, were successfully completed by the FLTOPS emulator. Several minor changes were incorporated into the original specification, however, to enable proper execution of the test.

The last test formally executed was a test of the OUTPUT command, PASS1 and PASS2 substitutions, comment cards and format control. Shown in figure A-8 is the FCF used in this test. Although not lengthy, it provided sufficient information to establish the correct operation of the FLTOPS output processor. Many aspects of output generation need further testing not possible given the resource/time constraints of this effort.

```
P     MESSAGE(G,'N:START OF DO STATEMENT TESTS')
P        /* DO BLOCK TEST */
P     SET(ONE = 1)
P     IF 1 = ONE THEN DO
P        MESSAGE(G,'N:START OF FIRST DO BLOCK')
P        SET(I = 1)
P        DO WHILE I <= 5
P           IF I=1 THEN MESSAGE(G,'N:START OF DO WHILE BLOCK')
P           MESSAGE(G,'N:THIS MESSAGE SHOULD PRINT 5 TIMES')
P           INCREMENT(I)
P           SET(J = 1)
P           SET(TEST = 'T')
P           DO WHILE J<=3 AND TEST = 'T'
P              MESSAGE(G,'N:THIS MESSAGE PRINTS 3 FOR 1 ABOVE')
P              IF I = 5 AND J = 3 THEN DO
P                 MESSAGE(G,'N:SWITCH TEST TO F')
P                 SET(TEST = 'F')
P                 END
P              ELSE IF I = 5 THEN MESSAGE(G,'N:I = 5 AND J < 3')
P              INCREMENT(J)
P              END
P           END
P        MESSAGE(G,'N:COMPLETION OF OUTER DO WHILE BLOCK')
P        DO FOR SYMBOL TABLE DO_TAB
P           SET(ONE = 1)
P           SET(TWO = 2*ONE)
P           SET(THREE = 3*TWO)
P           SET(FOUR = 4 * THREE)
P           IF THREE <= FOUR THEN DO
P              SET(THREE)
P              ENTER(@,FOUR)
P              MESSAGE(G,'N:THREE SET EQUAL TO FOUR')
P              END
P           END
P        END
P     ELSE DO
P        MESSAGE(G,'N:ALTERNATE PATH TAKEN 1<>ONE')
P        DO FOR SYMBOL TABLE DO_TAB
P           SET(ONE = 10)
P           SET(TWO = 2.*ONE)
P           SET(THREE = 3*TWO)
P           SET(FOUR = 4.0 * THREE)
P           END
P        END
```

FCF FOR DO BLOCK STRUCTURES

(Page 1 of 2)

FIGURE A-4

```
P     MESSAGE(G,'N:END OF LARGE DO BLOCK')
P     FIND(DO_INPUT_TEST) ERROR: DO END
P     DO FOR EACH SUBNODE OF @
P        INPUT(WORD)  ERROR: DO END
P        SET(*)
P        INPUT(CONSTANT OR STRING OR WORD)
P        ENTER(@,*)
P        END
P     STATUS('SYMBOL TABLES WHEN SUBNODE INPUT COMPLETED')
P     DO FOR SYMBOL TABLE DO_TAB
P        SET(NODES = 0)
P        DO FOR EACH SUBNODE OF DO_INPUT_TEST.SPECIAL_SYMBOLS
P           INPUT(WORD) ERROR: DO END
P           SET(*)
P           INPUT('*' OR ')' OR '(' OR '**' OR '/' OR '+' OR '-')
P              ERROR: MESSAGE(G,'W:ILLEGAL SPECIAL SYMBOL IN INPUT')
P           ENTER(@,*)
P           INCREMENT(NODES)
P           END
P        END
P     STATUS('SYMBOL TABLES AFTER SUBNODES OF SPECIAL SYMBOLS READ')
P     MESSAGE(G,'N:END OF DO BLOCK TEST')
```

FCF FOR DO BLOCK STRUCTURES

(Page 2 of 2)

FIGURE A-4

```
P     MESSAGE(G,'N:START OF IF STATEMENT TEST')
P        /* BOOLEAN EXPRESSIONS WITHIN IF STATEMENTS */
P     FIND(CONSTANTS)
P        /* IF STATEMENT #1 */
P     ENTER(IF_TAB,*)
P     IF INPUT(WORD) THEN MESSAGE(G,'W:FAILURE IN IF #1')
P     ELSE DO
P        INPUT('END OF NODE') ERROR: MESSAGE(G,'W:ERROR IN IF #1')
P        MESSAGE(G,'N:CORRECT PATH TAKEN IN IF #1')
P        END
P     IF NODE_EXISTS(GEOPHYSICAL_MODEL.EARTH.NON_ROTATING) THEN
P        MESSAGE(G,'N:CORRECT IDENT OF NODES IN IF #2')
P         /* IF STATEMENT #3 */
P     IF NOT SYMBOL_EXISTS(IF_TAB,CONSTANTS) THEN MESSAGE
P          (G,'W:INCORRECT BRANCH ON NOT CONDITION')
P     ELSE DO FOR SYMBOL TABLE IF_TAB
P        SET(CONSTANTS = 97.5)
P        MESSAGE(G,'N:CORRECT BRANCH ON IF #3')
P        SET(STRING = 'ARBITRARY #+$#% STRING')
P        END
P          /* IF STATEMENT #4 */
P     STATUS('IF_TAB AFTER IF STATEMENT #3')
P     IF TABLE_EMPTY(IF_TAB) THEN DO
P        MESSAGE(G,'N:IF_TAB EMPTY -- #4')
P        SET(I=0)
P        DO WHILE I <= 5
P           CATENATE(IF_TAB,CONSTANTS,*)
P           INCREMENT(I)
P           END
P        END
P     ELSE CLEAR_TABLE(IF_TAB)
P     STATUS('IF_TAB AFTER IF STATEMENT #4')
P        /* TEST OF RELATIONAL CONSTRUCTS */
P     FIND(CONSTANTS.HAV) ERROR: DO END
P     INPUT('=')
P     INPUT(CONSTANT)
P     ENTER(IF_TAB,HAV,*)
P        /* IF STATEMENT #5 */
P     IF VALUE(IF_TAB,HAV) = 23000 THEN MESSAGE
P          (G,'N:HAV IDENTIFIED AS EQUALING 23000')
P     ELSE DO WHILE VALUE(IF_TAB,HAV) < 23000
P        ENTER(IF_TAB,HAV,VALUE(IF_TAB,HAV)*2.)
P        END
P        /* IF STATEMENTS #6,7,8 */
P     DO FOR SYMBOL TABLE IF_TAB
P        IF 27.9*8./36.5 <= HAV THEN SET(HAV2 = HAV/2.)
P        IF HAV <> 23000 THEN MESSAGE(G,'N:HAV <> 23000')
P        IF HAV < 23000 THEN MESSAGE(G,'N:HAV < 23000  #8')
P        IF VALUE(FLTOPS,FUNCTION) = 'FULL' THEN DO END
P        ELSE MESSAGE(G,'W:FLTOPS FUNCTION NOT SET TO FULL')
P        END
```

FCF FOR CONDITIONAL STATEMENTS

(Page 1 of 2)

FIGURE A-5

```
P     MESSAGE(G,'N:TEST OF NESTED IF STATEMENTS')
P        /* IF STATEMENT #10 */
P     IF VALUE(IF_TAB,HAV) > 10 THEN
P     IF VALUE(IF_TAB,HAV) > 100 THEN
P     IF VALUE(IF_TAB,HAV) > 1000 THEN
P     IF VALUE(IF_TAB,HAV) > 10000 THEN DO
P        MESSAGE(G,'N:VALUE(IF_TAB,HAV) > 10000')
P        END
P     ELSE MESSAGE(G,'N:VALUE(IF_TAB,HAV) <= 10000')
P     ELSE MESSAGE(G,'N:VALUE(IF_TAB,HAV) <= 1000')
P     ELSE MESSAGE(G,'N:VALUE(IF_TAB,HAV) <= 100')
P     MESSAGE(G,'N:END OF NESTED IF TEST')
P        /* TEST OF TRICKY NESTED IF */
P        /* IF STATEMENT # 11 */
P     IF SYMBOL_EXISTS(IF_TAB,NOT_THERE) THEN MESSAGE
P           (G,'W:ILLEGAL STATEMENT BRANCH  #11')
P     ELSE IF NOT TABLE_EMPTY(IF_TAB) THEN
P     IF TABLE_EMPTY(IF_TAB) THEN DO END
P     ELSE IF VALUE(IF_TAB,HAV) <> 0 THEN MESSAGE
P           (G,'N:CORRECT EXECUTION OF #11')
P     ELSE DO END
P     ELSE MESSAGE(G,'W:TABLE IF_TAB TESTS EMPTY  #11')
P     MESSAGE(G,'N:TEST OF COMPLICATED BOOLEAN EXPRESSION')
P        /* TEST OF COMPLICATED BOOLEAN EXPRESSIONS  #12-#15 */
P     IF 1=1 AND 7>4 THEN MESSAGE
P           (G,'N:1 = 1 AND 7>4  #12')
P     IF 1<1 OR 7>4 THEN MESSAGE
P           (G,'N:1<1 OR 7>4 #13')
P     IF NOT(1<1 OR 7>4) THEN MESSAGE(G,'W:INCORRECT PATH -- #14')
P     ELSE MESSAGE(G,'N:CORRECT EVALUATION  #14')
P     IF(1=1 AND 7>4) AND NOT(1<1 OR 7>4) THEN MESSAGE
P           (G,'W:INCORRECT PATH -- #15')
P     ELSE MESSAGE(G,'N:CORRECT PATH TAKEN -- #15')
P     MESSAGE(G,'N:TEST OF ARITHMETIC EXPRESSIONS WITHIN BOOLEAN')
P     IF (15.5+4.5)*(4+1.) > 16.2*5. AND (6.3+3.7)*10. < 85 THEN
P           MESSAGE(G,'W:ERROR IN PROCESSING IF #16')
P     ELSE MESSAGE(G,'N:CORRECT EVALUATION -- #16')
P        /* IF STATEMENT #17 */
P     IF NOT((7.+3.)*(5.-4.) < 9) OR 7>8 THEN MESSAGE
P           (G,'N:CORRECT EVALUATION -- #17')
P     ELSE MESSAGE(G,'W:IF #17 IMPROPERLY EVALUATED')
P     IF NOT (7.+3.)*(5.-4.) < 9 OR 7>8 THEN MESSAGE
P           (G,'N:CORRECT PATH TAKEN -- #18')
P     ELSE MESSAGE(G,'W:INCORRECT EVALUATION  -- #18')
P     IF (((7+3)*10) = 100) THEN MESSAGE
P           (G,'N:CORRECT EVALUATION -- #19')
P     MESSAGE(G,'N:END OF IF STATEMENT TEST ')
```

FCF FOR CONDITIONAL STATEMENTS

(Page 2 of 2)

FIGURE A-5

```
P     MESSAGE(G,'N:BEGINNING OF FIND TEST')
P        /* TEST OF INPUT FILE SEARCH -- FIND */
P     FIND(GEOPHYSICAL_MODEL.EARTH.NON_ROTATING)
P          ERROR: MESSAGE(G,'W:CANNOT FIND GEOPHYSICAL_MODEL, ET AL')
P     FIND(GEOPHYSICAL_MODEL) ERROR: DO END
P     FIND(@.EARTH)  ERROR: DO END
P     FIND(@.NON_ROTATING)  ERROR: DO END
P     FIND(INTERPOLATION_ROUTINE)  ERROR: DO END
P     SAVE_POINTER
P     FIND(@.TABLE_INFORMATION.DATA)  ERROR:
P        MESSAGE(G,'W:COULD NOT FIND TABLE DATA')
P     RESTORE_POINTER
P        /* NOW FIND A BROTHER NODE */
P        /* FAILURE TO FIND SPECIFIC NODE -- TAKE ERROR PATH */
P     FIND(GEOPHYSICAL_MODEL.VENUS.ELLIPSE)
P        ERROR: MESSAGE(G,'N:NODE NOT FOUND -- PLANNED ERROR')
P     FIND(GEOPHYSICAL_MODEL)
P     SAVE_POINTER
P     FIND(@.EARTH.SPHERICAL.NEXT_LEVEL.ANOTHER_LEVEL)
P        ERROR: MESSAGE(G,'N:THIS MESSAGE SHOULD BE PRINTED')
P     RESTORE_POINTER
P     MESSAGE(G,'N:END OF FIND STATEMENT TEST')
```

FCF FOR INPUT SEARCH STATEMENTS

FIGURE A-6

```
P     MESSAGE(G,'N:BEGINNING OF INPUT STATEMENT TEST')
P       /* TEST OF INPUT STATEMENT */
P     FIND(CONSTANTS)      /* POSITION FILE */
P     INPUT('END OF NODE')    /* FORCE TO NEXT NODE */
P     INPUT(WORD)  ERROR: MESSAGE(G,'W:COULD NOT FIND WORD')
P     ENTER(INPUT,*)
P     INPUT('=')    ERROR: DO END
P     INPUT(NUMBER)    ERROR: DO END
P     ENTER(@,*)
P     FIND(INTERPOLATION_ROUTINE)
P     INPUT('=' OR WORD) ERROR: MESSAGE(G,'W:DID NOT FIND =')
P     INPUT(CONSTANT OR WORD OR STRING)  ERROR: MESSAGE
P          (G,'W:DID NOT FIND POLY')
P     FIND(STRING)  ERROR: DO END
P     INPUT(STRING)
P     ENTER(INPUT,STRING,*)
P     INPUT('END OF NODE') ERROR: DO END
P     INPUT(WORD)  ERROR: DO END
P     IF * = 'SIGNED_NUMBER' THEN DO
P        /* ORDER PURPOSELY WRONG */
P        ENTER(INPUT,SIGNED_NUMBER)
P        INPUT(SIGNED_NUMBER)
P         ERROR: IF INPUT('=') THEN INPUT(WORD)
P         ERROR: INPUT(SIGNED_NUMBER)
P         ERROR: MESSAGE(G,'W:THIS MESSAGE SHOULD NOT BE PRINTED')
P         ELSE MESSAGE(G,'W:DID NOT FIND THE = SIGN')
P        ENTER(@,*)
P        END
P        /* NOW INPUT CATENATED INPUT, I.E. AN INPUT GROUP */
P     FIND(INTERPOLATION_ROUTINE.INTERPOLATION_VALUES)
P     INPUT( '=' NUMBER ',' NUMBER)    ERROR: DO END
P        /* NOW INPUT SPECIAL SYMBOLS */
P     FIND(SPECIAL_SYMBOLS) ERROR: DO END
P     ENTER(INPUT,*)
P     DO WHILE INPUT('*' OR '+' OR '-' OR ',' OR '(' OR ')' )
P        CATENATE(@,*)
P        END
P     INPUT('END OF NODE') ERROR:MESSAGE(G,'W:END OF NODE WASNT READ')
P     INPUT(WORD)  ERROR: DO END
P     ENTER(INPUT,*)
P     INPUT('=')  ERROR: DO END
P     MESSAGE(G,'N:START READING AN ARITHMETIC EXPRESSION')
P     INVOKE A_EXPR
P     INPUT(WORD)  ERROR: MESSAGE
P        (G,'N:THIS MESSAGE SHOULD APPEAR -- END OF FILE')
P     MESSAGE(G,'N:END OF INPUT STATEMENT TEST')
```

FCF FOR INPUT READ AND RULE INVOCATION

(Page 1 of 2)

FIGURE A-7

```
P A_EXPR:
P     IF INPUT('+' OR '-') THEN DO
P        CATENATE(@,*)
P        INVOKE A_TERM
P        END
P     ELSE INVOKE A_TERM
P     DO WHILE INPUT('+' OR '-' OR '*' OR '/' OR '**')
P        CATENATE(@,*)
P        INVOKE A_TERM
P        END
P     END A_EXPR
P A_TERM:
P     IF INPUT('(') THEN DO
P        CATENATE(@,*)
P        INVOKE A_EXPR
P        INPUT(')')  ERROR: MESSAGE(I,
P           'S:MISSING RIGHT PARENTHESIS IN ARITHMETIC EXPRESSION')
P        CATENATE(@,*)
P        END
P     ELSE IF INPUT(NUMBER) THEN CATENATE(@,*)
P     ELSE DO
P        INPUT(WORD)  ERROR: MESSAGE(I,
P             'S:ILLEGAL TERM IN ARITHMETIC EXPRESSION')
P        CATENATE(@,*)
P        IF INPUT('(') THEN DO
P           CATENATE(@,*)
P           INVOKE A_EXPR
P           DO WHILE INPUT(',')
P              CATENATE(@,*)
P              INVOKE A_EXPR
P              END
P           INPUT(')')  ERROR: MESSAGE(I,
P              'S:MISSING RIGHT PARENTHESIS ON ARRAY DECLARATION')
P           CATENATE(@,*)
P           END
P        END
P     END A_TERM
```

FCF FOR INPUT READ AND RULE INVOCATION

(PAGE 2 of 2)

FIGURE A-7

```
P          /* TEST OF FORTRAN OUTPUT */
       WRITE(1,1000) INPUT
1000   FORMAT(40A2///)
       IF(N.EQ.3) GO TO 7
C          THIS IS AN IN LINE COMMENT
       DIMENSION A(17,3)
15     X = SQRT(7.3*10.)**(7.6+Y*M)
       A(3,5) = EXP(A(1,2))*5.+Z-3.
P      ENTER(PASS1,'#JBUF','(J)')
       X#JBUF = Y#JBUF * 5.
       EQUIVALENCE(JBUF(3),IO)
       X     =     7.3 * X #JBUF
P      DELETE(PASS1,'#JBUF')
P          /* NOW SYNTHESIZE A STATEMENT */
P      OUTPUT('25    CONTINUE',#)
P      OUTPUT('IF (Y.EQ.0.) CALL SUBR(J,X)',#)
P      OUTPUT('1000')
P      OUTPUT('FORMAT(I5,G14.7,3(3X,F10.4/))',#)
P      ENTER(OUT_TAB,STATEMENT,'IF(J.GE.7) X=7.3*Z')
P      OUTPUT('75',VALUE(OUT_TAB,STATEMENT),#)
P      OUTPUT(COMMENT:,' THIS IS A GENERATED COMMENT',#)
P      SET(OUTPUT = 'DATA')
P      OUTPUT(COL(6),7.3 FORMAT(F5.2),COL(21),(3.7+8.5)*5. FORMAT(I),#)
P      MESSAGE(G,'N:END OF OUTPUT TEST')
```

FCF FOR SIMPLE FORTRAN OUTPUT

FIGURE A-8

The following section details the major changes in the specification affected by the validation effort.

A.5 CHANGES/MODIFICATIONS TO FLTOPS DESIGN

As a direct result of the validation effort, both through the test cases developed at AFFTC and those developed by SAI, many minor and several major modifications to the FLTOPS design have been incorporated into the specification included in the body of this report. Although all changes would be tedious and unproductive to document, major modifications and enhancements are listed here to provide adequate documentation of the usefulness of such an effort. All changes would have, of necessity, been made eventually. The factor of importance here is that changes/corrections have been made early in the development cycle where they can be incorporated at minimum cost. If left to the implementation phase, the discovery of these design changes may well have resulted in increased span time to delivery of an implemented system. Instead, the Air Force has a functional prototype system that can be utilized for training and development purposes for actual end user personnel as well as provide a continuing medium for design validation and bench mark development.

Modifications actually are separated into two different classes: those changes to the original design due to logical inconsistencies or omissions in the original design and those changes made due to the need for additional capability not originally specified. Figure A-9 lists those major corrections made to the original design. Figure A-10 provides a summary of major additions to the original design.

Several rules proved to be written inadequately to perform the desired function. Notably, the DO FOR EACH SUBNODE command was rewritten entirely so as to provide for proper recovery and scanning when each subnode had been processed. The INVOKE statement was also written so as to insure that it properly handled not only invocations that were to be executed, but also those invocations resulting in syntax checking and scanning due to conditional bypassing of FCF code.

RULES REWRITTEN:

DO_FOR_EACH_SUBNODE_STATEMENT

INVOKE_STATEMENT

RULES TO WHICH SIGNIFICANT CORRECTIONS WERE MADE:

STRING_EXPRESSION

ARITHMETIC_EXPRESSION

TABLE_EMPTY_STATEMENT

SET_STATEMENT

INCREMENT_STATEMENT

DECREMENT_STATEMENT

VALUE_FUNCTION

ARRAY_ELEMENTS

RULES TO WHICH MINOR CORRECTIONS WERE MADE:

DO_FOR_SYMBOL_TABLE_STATEMENT

FIND_STATEMENT

DEFINE_STATEMENT

ARITHMETIC_PRIMARY

FIND_INPUT_NODE

FIND_NODE_THIS_LEVEL

(Not included specifically on this list are changes that were needed as a result of the significant correction to STRING_EXPRESSION.)

DESIGN CORRECTIONS AS A
RESULT OF THE FLTOPS VALIDATION EFFORT

FIGURE A-9

Although major rewrites did not occur, important minor corrections were made to the following command operations: TABLE_EMPTY, DEFINE, SET, INCREMENT, DECREMENT, ARITHMETIC EXPRESSION processing, STRING EXPRESSION processing, ARRAY table processing, and FIND INPUT NODE processing. Of special note is the change incoprorated into STRING EXPRESSION processing. Although only several lines in length, all statements that utilized STRING EXPRESSIONS in their syntax format had to be modified slightly. The change was necessary to insure that the following condition was satisfied:

```
INVOKE A
ENTER(A, B, A)
```

A rule named A is invoked in the first statement. A symbol B in table A is referenced in the second. In this instance, the string A is not entered into the table location but the value of symbol A in the default symbol table. This operation is exactly what was intended in the original design specification, but was not accomplished because the grammar was incorrect. Since nearly half the rules in the grammar were in some way modified because of this change, it was, therefore, very important that it was caught early as it may have had an adverse effect on the implementation effort. In addition, minor changes were also made to insure proper stack operations especially with respect to the arithmetic stack and input pointer stack.

As depicted in figure A-10 many more changes were made as a result of enhancements to the original design. A principal reason for these enhancements was that the end user actually formulated and solved problems utilizing the capabilities of the FCF control language. As problems were encountered, those solutions that seemed feasible and useful were incorporated into the design. Such user/designer interaction during the design phase served the dual purpose of training the end users of the system as well as providing insight into what capabilities they felt would be helpful in doing their job better.

DESIGN ADDITION	RESULT
INDEX_FUNCTION	Determination of the starting position of sub-string within a string.
LENGTH_FUNCTION	Determination of number of characters in a string (includes any symbol table entry).
DEFINE...AS...	Provides user with multiple names for the same table.
DEFINE...AS NULL	Provides ability to remove multiple table names and/or dimensions.
Expanded MESSAGE	Provides users capability to output symbol table or computed values along with messages.
Expanded INPUT	Provides users with the capability to read multiple INPUT elements simultaneously.
Expanded OUTPUT	Provides users explicit control over EOF and EOR marks, comment card generation and format control of output elements.
FORMAT Control	Provides users format control over elements inserted into symbol tables or output via the MESSAGE or OUTPUT commands.
Expanded Boolean Expression	Provides users with additional capabilities in constructing Boolean Expressions, including any FIND statement or STRING_EXPRESSION.

Rules Added to the Design Specification:

DEFINED_ELEMENT, INPUT_GROUP, INDEX_FUNCTION, LENGTH_FUNCTION, SIGNED_NUMBER, COLUMN_FORMAT, FORMAT_STATEMENT, FORMAT_ELEMENT

Primitives Added to the Design Specification:

.INDEX, .LENGTH, .SETUP_TABLE_EQUIVALENCE, .NULL_OUT_TABLE, .ADJUST_COLUMN, .FORMAT_STRING

DESIGN ENHANCEMENTS AS A
RESULT OF THE FLTOPS VALIDATION EFFORT

FIGURE A-10

Several new built-in functions were added to the design and the functions of several other control statements were extended as a result of the validation effort. The LENGTH and INDEX functions were added to provide better user control of string quantities. These combined with the SUBSTRING function and REPLACE/CATENATE statements, provide a sophisticated repertory of string manipulation commands.

The function of the INPUT command was expanded so as to allow the user the ability to catenate input requests within one command structure. Now a word and number can be read in one operation, INPUT(WORD NUMBER). Although functionally equivalent to the original design, the inputing of a SIGNED_NUMBER is controlled by the grammar and not as a special token class in the input lexical analyzer. As a result, the user can input -39.7 either as INPUT('–') INPUT(NUMBER) or as INPUT(SIGNED_NUMBER).

The DEFINE statement was expanded to serve two additional purposes. With the DEFINE...AS NULL statement the user can eliminate previous dimensions or equivalences and reuse an array table as any other symbol table. With the DEFINE...AS... statement, the user can establish several table names to refer to the same table. This greatly simplifies the use of rules that may be invoked from several locations with the intent of accessing or storing different table data.

The CATENATE statement was modified to allow for multiple catenations at the same location with one CATENATE command. This reduces the number of statements needed to accomplish some tasks and has proven to be a useful extension.

Like the CATENATE statement, the MESSAGE statement syntax has been modified to provide greater feasibility to the user. A string literal identifying the type and severity of the message is required but now any symbol table value or arithmetic calculation can be output directly by the MESSAGE command.

The largest addition to the design has been with respect to the OUTPUT statement. The OUTPUT has been expanded to permit explicit control over END OF FILE and END OF RECORD marks, the explicit generation of COMMENT cards and, more importantly explicit FORMAT control. The FORMAT control on OUTPUT allows both column control and FORTRAN like formatted data control. The introduction of the format control on OUTPUT prompted the recognition of a similar need for whenever a value is entered into a symbol table. As a result, the FORTRAN-like format control was also made optional for all symbol table operations in which data is being inserted into a symbol table.

As is evident, very nearly all the rules in the original design specification have been modified so as to conform to the original and enhanced requirements. The important difference with the new specification contained in the body of this report is that it has been validated that it meets user requirements. Furthermore, the existence of the functional prototype insures that further testing and modifications can continue at a cost effective rate until implementation begins. In addition, the functional prototype can be used throughout the implementation phase to develop and test actual modules that are needed to bench mark the implemented FLTOPS. As a result, the prototype provides the mechanism whereby the implemented FLTOPS can be effectively used from the time of its installation.

APPENDIX B

EXTENDED FLTOPS VALIDATION EFFORT

B.1 INTRODUCTION

The original validation effort succeeded in verifying and validating the worth and useability of the FLTOPS design. However, inherent in the validation effort were extensive changes to the original design which necessitated both changes to the design specification and the emulator package that performed the FLTOPS functions. Many of these changes to the design were not adequately tested in the initial validation effort both because of time and resource constraints.

In addition, an FCF provided by AFFTC/DOEDM personnel that customized a portion of the UFTAS system known as LINK4, had not yet been successfully run by the end of the initial validation effort. The complexities of designing an FCF by those unfamiliar with the operation of the host language which in and by itself was subject to change throughout the validation effort, combined with an emulator that had not been thoroughly tested, caused many slow downs to a straight forward but subtly complex task.

Both of the above tasks, were deemed to be sufficiently important to warrant a small extended effort of the validation phase of the contract. The objectives, then, of this phase of the contract were two-fold:

- provide continued emulator support of AFFTC/DOEDM personnel. Specifically, this was to result in successful processing of the LINK4 FCF using four input test cases as provided by AFFTC.

- Provide enhanced FCF test cases which would include tests of those elements of the design added during the validation phases of this contract.

The following sections discuss the work done on each of these tasks. Although minimal compared with earlier validation efforts, several changes

were also made to the design specification as a result of this effort. The concluding section of this appendix details these changes.

B.2 EMULATOR SUPPORT ACTIVITY

The emulatory built as a task under the initial FLTOPS validation effort provided some valuable insights as to areas in which the design of the FLTOPS system could be strengthened. Also key to insuring that the design would provide functional support for real world problems was the development of FCF's by the end users of the system. The LINK4 FCF generated by AFFTC/DOEDM personnel provided an excellent example of a real-world problem that demonstrated the effectiveness of the concept. Although the FCF was received as a part of the initial validation effort, its complexity combined with minor errors in the emulator caused successful completion of the LINK4 test runs to be postponed until this phase of the contract. What follows are some reflections and results of this initial real-world operation of the FLTOPS concept. Of particular note are the comparisons listed in table B-1 and discussed in section B.2.3.

B.2.1 PROCESSING RESTRICTIONS OF THE FLTOPS EMULATOR

The emulation of the FLTOPS functions for purposes of test and validation of the design was accomplished through the creation of an emulator. This "quick and dirty" program performed its mission in the desired fashion and was invaluable in analyzing and validating the design. However, because it was an emulation and not an implementation of the design, it had limitations that immediately came to bear upon the successful running of the LINK4 FCF. Among these limitations were:

1) The total limit on size of an FCF, initially set to 400, later increased to 2100, card images. The design does not establish a limit.
2) The total limit on size of a symbol table entry, initially set to 50, later increased to 100 characters. This was

still not sufficient; changes to the FCF were necessary to work around this emulator restriction. The design does not establish a limit for length of symbol table entries.

3) The total number of tables was initially set to 20 and later increased to 40. No limit is identified in the design.
4) The total limit on the number of symbol table entries was initially set to 200 and later increased to 300. No limit is identified in the design.
5) Limits were violated on various internal operations stacks. Each was increased appropriately. No stack size limits have been identified in the design.
6) The emulator does not process comments between rules and at the beginning of the input file. Actual implementation will allow comments to appear anywhere.

Each of the above restrictions provided some slow down in the eventual successful processing of the LINK4 FCF. An actual implementation would remove each restriction in a satisfactory manner. Time and purpose did not allow the emulator to be so configured. The importance of identifying these restrictions is that it identifies an area of concern for the implementor, and it provides the rational for some changes that were necessary in the LINK4 FCF so that it could be processed.

B.2.2 THE LINK4 FCF – THE SOURCES OF ERROR

Just as the emulator was changed in a number of instances to accommodate the FCF, so too was the FCF changed in a number of instances to promote its successful processing. As in any programming language, a certain number of coding errors are expected as a result of the complexity of the problem. The LINK4 FCF was changed in a number of places because of coding or syntax errors, in several other places to eliminate cumbersome or faulty use of the language and in several other places to work around emulator restrictions (see above). In total, well over 100 lines

of original code were corrected or modified in some way to facilitate processing. Although this may seem like a great deal, overall it was less than 10 percent of the total FCF that needed changing. Many changes were of a pure syntactic nature, i.e. missing parenthesis, commas, apostrophes, etc. Other changes were due to faulty coding logic, (e.g., SAVE-POINTER command out of sequence with the corresponding RESTORE-POINTER command, the value of a parameter counter being inadvertently set to the wrong value, etc.). These types of errors are to be expected in any coding exercise. The importance here is that the capability to trace the cause of error should be greatly enhanced by the debugging capabilities of the language at hand. The existence of such errors provided a reason and means to test the utility of the design debugging features, notably the TRACE, and DEBUG switches and the STATUS and ASSERT commands. As discussed below, some changes were made and insights gained in this area as a result of this debugging effort.

Of most importance with regard to the design are those changes made to correct improper uses of the language. At the time that this FCF was written, certain enhancements to the design had not yet been implemented. As a result, an attempt was made to utilize certain improper language constructs to perform necessary functions. Examples of these constructs are as follows:

```
IF VALUE (VALUE (UNITS) (I) ) <>  ' ' then . . .

CATENATE (DEFAULT, VALUE (DATA2) ),
   LENGTH (VALUE (UNITS) (I) ), 'H', VALUE (UNITS) (I) )
```

The syntax of the VALUE function does not permit the kind of indirect table name reference implied by these constructs. However, the need for this kind of indirect reference was part of the motivation for adapting the DEFINE statement to allow for table name equivalencing. The above statements were transformed into the following kinds of constructs:

```
DEFINE UNITS AS VALUE (UNITS)
     /*  A TABLE UNITS IS EQUIVALENCED     */
     /*  TO THE TABLE NAME IDENTIFIED      */
     /*  BY THE ENTRY UNITS IN THE DEFAULT */
     /*  SYMBOL TABLE                      */
    :
IF UNITS (I) <> ' ' THEN . . .
    :
CATENATE (DEFAULT, VALUE (DATA2),
    LENGTH (UNITS (I) ), 'H', UNITS (I)
    :
DEFINE UNITS AS NULL
    / * REMOVE THE TABLE EQUIVALENCE       */
```

The resulting code is much more readable, compact and correct. Another possible, more subtle misconception involved nesting of VALUE statements. The statement

```
IF VALUE (VALUE (UNITS) (I) ) <> ' ' THEN . . .
```

could, using the above procedures, be changed directly into

```
IF VALUE (UNITS  (I) ) <> ' ' THEN . . .
```

but this is an unnecessary use of the VALUE function. The result of this expression is the same as the following:

```
IF UNITS (I) <> ' ' THEN . . .
```

The language constructs are such that "implied value" statements can be used in situations such as this where there is no ambiguity in meaning.

Still other changes were necessary in the LINK4 FCF to accommodate inadequacies in the emulator. Specifically, the length of symbol table entries was restricted to 100 characters. The building of various common block statements as well as data statements required symbol table entries consistently longer than what was available. Because it would have required significant effort to alter the emulator to handle arbitrarily long strings, it was decided to modify the original FCF. The method adopted utilized array tables in place of the single, long table entry previously required. Upon output, a loop was created which output all non-empty table entries before the statement terminator was issued. Although somewhat more lengthy than the original method, this procedure clearly demonstrated the adaptability of the FCF control language.

Because the LINK4 FCF was a real world example whose intent was to customize FORTRAN as opposed to a validation, of the design it proved to be a very good test of the solidarity of the emulator itself. Several implementation concerns immediately surfaced because of this exercise. Especially important was the provision of adequate internal safeguards against overflow and stack size violations. Of particular importance was the need for providing sufficient stack sizes to inhibit overflow conditions. Other concerns brought to light were the need for different default conversion guidelines when arithmetic calculations resulted in a value being converted to string format. The emulator uses the P1/1 guideline which invariably allows for too many leading blanks. Some concerns regarding the form of the FORTRAN output were also brought to light. Although significant enhancements were made to the emulator's ability to output exactly what was input, there is still a need to have statements in which a substitution has been made written without having variable names split between lines, without large blank spaces in the middle of lines, and without other unsightly output forms.

Of particular importance in this exercise was the insight derived from debugging the FCF. Although the debugging tast was decidedly more

complicated because of the need for concurrent debugging of the FCF and the emulator package, the overall impact was to provide necessary experience concerning the content and level of useful information. Both the TRACE and STATUS debugging facilities proved invaluable in system debugging. The STATUS command has sufficient flexibility to make it very valuable in monitoring FCF execution performance. The TRACE feature, too, has proved to be very useful. However, the level of printout received was decidedly unbalanced prior to execution of this FCF. Many things that would be useful to the system debugger were being printed out at TRACE level HIGH where they simply obscurred readability. Similarly, several items that should have been included at a TRACE level of LOW were only turned on when set to HIGH. The total debugging process provided some valuable insights in both what and when certain items became useful. Surely more experience will be necessary in establishing all debugging guidelines, but this experience has provided a big step in identifying levels of useful debug output.

B.2.3 COMPARISON OF TEST RESULTS

The LINK4 FCF was processed using the four test cases as provided by AFFTC. The resulting FORTRAN code was subsequently compiled at the AFFTC computer facility. The number of lines of code and the memory locations required for each of the four cases were then compared with the original LINK4 as used in UFTAS. Table B-1 lists these comparisons as derived from the actual compilation statistics. As easily seen from the table, dramatic reductions in both the number of lines of code and in memory requirements resulted in several test cases. A good portion of the savings resulted because of fairly simple operations. Note that in Test 2-4 the number of values stored in common were cut significantly from the original. This savings was primarily a result of eliminating the need for dimensioned variables in those instances where only the first entry of an array was actually used. This savings will result not only in decreased core requirements, and hence a greater likelihood of qualifying for faster access to computer resources, but also in faster

run times as branching and array position calculations are eliminated.

The other large segment of code that was eliminated resulted from the elimenation of those subroutines not needed for certain types of analysis. Tests 3 and 4 resulted in eliminating fairly large subroutines (e.g., 734 lines of code in the original version of DALAG) from even being processed and subsequently loaded with the LINK4 package. Such savings could easily result in eliminating the need for one or more overlays for a particular analysis.

Note also that in every subroutine, the number of lines of code was changed indicating some structural changes in the way elements were processed. Although the details of these changes may be quite varied among the test cases, more than likely the readability and processing speed of each routine was greatly enhanced by the removal of superflous code not relevant to the application.

Figures B-2 and B-3 provide listings of the input files used for test cases one and four respectively. Note, that the inputs are readable and understandable by those who most need to know what the code is doing, the flight test engineer. Such self documentation of the code makes immediately clear what is supposed to be accomplished by executing this FORTRAN code. Note also the similarities between the two inputs. A few change in the input file reduces greatly the amount of code produced. Although dependent on FCF construction, this attribute demonstrates the power and flexibility of the input file associated with each FCF. Inputs for tests two and three are very similar in content to test four and are omitted here. Similarly, the LINK4 FCF is quite lengthy and has not been reproduced here.

MEMORY

WORDS IN DECIMAL (OCTAL)

	Original		Test 1		Test 2		Test 3		Test 4	
LINK4	56	(70)	54	(66)	26	(32)	26	(32)	26	(32)
INPUT4	1393	(2,561)	1160	(3,838)	1145	(2,171)	1102	(2,116)	1105	(2,121)
DALAG	734	(1,336)	673	(1,241)	———		———		———	
SFEXT	468	(724)	453	(705)	363	(553)	314	(472)	250	(372)
DAUPR	188	(274)	127	(177)	115	(163)	———		———	
SDABB	64	(100)	66	(102)	66	(102)	———		———	
SNXNZ	280	(430)	105	(151)	82	(122)	82	(122)	82	(122)
COMMON	4570	(10,633)	4239	(10,217)	1090	(2,102)	1102	(2,116)	1105	(2,121)
TOTAL	7753	(17,111)	6877	(15,335)	2887	(5,507)	2626	(5,102)	2568	(5,010)
SAVINGS	0%		11%		63%		66%		67%	

LINES CODE

	Original	Test 1	Test 2	Test 3	Test 4
LINK4	53	42	26	26	26
INPUT4	232	139	108	91	91
DALAG	158	125	—	—	—
SFEXT	193	143	129	118	89
DAUPR	81	56	52	—	—
SDABB	22	23	23	—	—
SNXNZ	101	42	37	37	37
TOTAL	840	570	375	272	243
REDUCTION	0%	32%	55%	68%	71%

Comparison of FORTRAN Code
produced by various LINK4 FCF Test Cases

Table B-1

```
CONSTANTS
   STANDARD
      CENTER_OF_GRAVITY = 20.                    /* % MAC */
      NZ_WIND_AXIS = 1.                          /* NORMALLY DEFAULT IS 1 */
      GROSS_WEIGHT = 20000.2
      INCREMENT_ADDED_TO_MODEL_TEMPERATURE = .5
      USE_ALFA_IN_THRUST_AND_DRAG_CALCULATIONS
      STANDARD_CLIMBS_TO_TEST_MACH
LINK4
   CONSTANTS
      DISTANCE_FROM_VANE_LEADING_EDGE_TO_MAC = 38.5              /* FT */
              /* THE FOLLOWING 2 CONSTANTS ARE NEEDED ONLY IF UPWASH */
              /* OR PITCH RATE CORRECTIONS ARE BEING MADE            */
      ALFA_CONVERGENCE_CRITERION = .0015                         /* RAD */
      ALFA_ITERATION_LIMIT = 10
      CL_CONVERGENCE_CRITERION = .02
      CL_ITERATION_LIMIT = 15
      LOAD_FACTOR_DATA_SOURCE =  CG_ACCELEROMETER
      REFERENCE_ACCELERATION_DUE_TO_GRAVITY = 32.17405    /* FT/SEC? */
                                              /* NORMALLY, 32.17405 */
      THETA_EXPONENT = 0.5        /* NORMALLY, 0.5 IS USED */
      ALFA_NON_CONVERGENCE =   USE_LAST_VALUE
      CL_NON_CONVERGENCE =   RECOMPUTE
      RECOMPUTE_CL                            /* OPTIONAL */
      COMPUTE_CD                              /* OPTIONAL */
   ANGLE_OF_ATTACK
      CORRECTIONS
         DYNAMIC_LAG
            VANE_ACCELEROMETER_SYSTEM
               FREQUENCY
                  COMPUTE
                     CONSTANTS
                        CHARACTERISTIC_AREA_OF_BOOM = .5        /* FT */
                        MOMENT_OF_INERTIA_OF_VANE_ACCELEROMETER = 18
               DAMPENING_RATIO
                  COMPUTE
               CONSTANTS
                  ALFA_CONVERGENCE_CRITERION = .0012         /* RAD */
                  ALFA_ITERATION_LIMIT = 12
                  NUMBER_OF_POINTS_REQUIRED_TO_MEET_CONVERGENCE_CRITERION
                                                    = 85  /* % */
                  CHARACTERISTIC_AREA_OF_VANE = .115        /* FT2 */
                  DCL_OVER_DALFA_FOR_VANE = 13.33             /* PER RAD */
                  GAIN_CONSTANT = 1.25
                  DISTANCE_FROM_VANE_CG_TO_VANE_AERODYNAMIC_CENTER =
                                                   .015 /* FT */
         BOOM_BENDING
            YES
               CONSTANTS
                  DCL_OVER_DALFA_FOR_BOOM = 13.34           /* PER RAD */
                  WEIGHT_OF_BOOM = 19.50               /*  LB  */
               FUNCTION_OF_FORCE_ON_BOOM
                  CURVE_NUMBER  100
         PITCH_RATE
            YES
         UPWASH
            FUNCTION_OF  ALFA
               CURVE_NUMBER  5010

   LOAD_FACTORS
      CONSTANTS
         DATA_SOURCE = 2
         CG_ACCELEROMETER_MISALIGNMENT_ANGLE = 0.0     /* RADIANS */
         BOOM_ACCELEROMETER_MISALIGNMENT_ANGLE = 0.0   /* RADIANS */
   ADDITIONAL_INPUTS
      BETA,   GAMA(' TWO'),
              GAMA(' THREE')
   ADDITIONAL_OUTPUTS
      DELTA, ZETA,  OMEGA
```

Figure B-2

INPUTS FOR TEST 1

```
CONSTANTS
   STANDARD
      CENTER_OF_GRAVITY = 20.                    /* % MAC */
      NZ_WIND_AXIS = 1.                          /* NORMALLY DEFAULT IS 1 */
      GROSS_WEIGHT = 20000.2
      INCREMENT_ADDED_TO_MODEL_TEMPERATURE = .5
                                                          /* DEG K */
      USE_ALFA_IN_THRUST_AND_DRAG_CALCULATIONS
      STANDARD_CLIMBS_TO_TEST_MACH
LINK4
   CONSTANTS
      DISTANCE_FROM_VANE_LEADING_EDGE_TO_MAC = 38.5              /* FT */
            /* THE FOLLOWING 2 CONSTANTS ARE NEEDED ONLY IF UPWASH */
            /* OR PITCH RATE CORRECTIONS ARE BEING MADE             */
      ALFA_CONVERGENCE_CRITERION = .0015                        /* RAD */
      ALFA_ITERATION_LIMIT = 10
      CL_CONVERGENCE_CRITERION = .02
      CL_ITERATION_LIMIT = 15
      LOAD_FACTOR_DATA_SOURCE =  CG_ACCELEROMETER
      REFERENCE_ACCELERATION_DUE_TO_GRAVITY = 32.17405     /* FT/SEC2 */
                                              /* NORMALLY, 32.17405 */
      THETA_EXPONENT = 0.5         /* NORMALLY, 0.5 IS USED */
      ALFA_NON_CONVERGENCE = USE_LAST_VALUE
      CL_NON_CONVERGENCE = TERMINATE
      RECOMPUTE_CL                              /* OPTIONAL */
      COMPUTE_CD                                /* OPTIONAL */
   ANGLE_OF_ATTACK (ALFA)
      CORRECTIONS
         DYNAMIC_LAG
            NO
         BOOM_BENDING
            NO
         PITCH_RATE
            NO
         UPWASH
            NO
   LOAD_FACTORS
      CONSTANTS
         DATA_SOURCE = 2
         CG_ACCELEROMETER_MISALIGNMENT_ANGLE = 0.0    /* RADIANS */
         BOOM_ACCELEROMETER_MISALIGNMENT_ANGLE = 0.0  /* RADIANS */
   ADDITIONAL_INPUTS
      BETA,   GAMA(' TWO'),
              GAMA(' THREE')
   ADDITIONAL_OUTPUTS
      DELTA, ZETA,  OMEGA
```

Figure B-3

INPUTS FOR TEST 4

B.3 DESIGN OF ADDITIONAL TEST CASES

In addition to the performance of actual FLTOPS runs with the emulator, some other work was done to enhance the repertory of test cases. The initial validation effort produced a number of tests which proved to be incomplete, primarily because of new capabilities added throughout the validation effort. In addition, those capabilities recently added were not subject to the same level of testing as those identified in the original specification. With these test cases, it was possible to more completely verify the functional soundness of the design specification, as well as provide an initial framework for verification testing of an implemented system.

Figures B-4 through B-8 exhibit the test cases as instructed by this effort. Note that only one test, Figure B-4, is completely new. The others are merely modified versions of the originals constructed as a part of the initial validation effort. (See Appendix A.) A brief explanation of each of the test cases follows.

The test of the EXECUTE and PARAMETERS statement capability was not previously constructed. Shown in figure B-4, this test proved to be significant in uncovering some logical flaws in the original design. Using two execute modules, SQUARE and AVERAGE, the test exercises both the setup and passing of parameters. Both real and integer values are passed and calculated. This test does not exercise the special FORTRAN format switches provided in the design for control of the data conversion in executable modules. Being of a quick and dirty design, the emulator uses the PL/1 default conversion rules for these conversions.

Figure B-5 lists an enhanced INPUT statements test. An important addition to the design was the capability to read catenated inputs. Only minimally tested in the previous version, this test now provides an expanded test capability in this area. Specifically, the test

```
P     MESSAGE('GN:START OF EXECUTE ROUTINE TEST')
P     FIND(INTERPOLATION_ROUTINE.TABLE_INFORMATION.DATA)  ERROR: DO END
P     DEFINE X(10), Y(5)
P     INPUT('=')   ERROR: DO   END
P     SET(I=0)
P     DO WHILE I < 10
P        INCREMENT(I)
P        INPUT(CONSTANT) ERROR: DO END
P        ENTER(X(I),*)
P        END
P     INPUT('END OF NODE')  ERROR: DO END
P     INPUT('INTERPOLATION_VALUES' '=')  ERROR: DO END
P     SET(I=0)
P     DO WHILE I < 5
P        INPUT(CONSTANT OR ',' CONSTANT)  ERROR: DO END
P        INCREMENT(I)
P        ENTER(Y(I),*)
P        END
P     STATUS('CONTENTS OF SYMBOL TABLES BEFORE BEGIN OF EXECUTE')
P     SET(MAX = 10)
P     PARAMETERS SQUARE(REAL,REAL)
P     PARAMETERS AVERAGE(REAL,INTEGER,REAL(MAX))
P     MESSAGE('GN:PARAMETERS IDENTIFIED--EXECUTE')
P     EXECUTE AVERAGE(AVGX,MAX,X(1))
P     MESSAGE('GN:VALUE OF AVGX = ',AVGX => F10.4)
P     EXECUTE SQUARE(AVGX,AVGX2)
P     MESSAGE('GN:VALUE OF AVGX SQUARED = ',AVGX2 => F10.4)
P     STATUS('SYMBOL TABLES AFTER AVERAGE AND SQUARE EXECUTED')
      PARAMETERS(REAL,INTEGER,REAL(5))
P     EXECUTE AVERAGE(AVGY,5,Y(1))
P     EXECUTE SQUARE(AVGY,AVGY2)
P     STATUS('CONTENTS OF SYMBOL TABLES AFTER Y USED IN EXECUTE')
P     EXECUTE SQUARE(7.3*5.4,SQR)
P     MESSAGE('GN:ANSWER SHOULD BE 1553.94 ',SQR => F10.2)
P     EXECUTE AVERAGE(AVGPX,5,X(6))
P     MESSAGE('GN:AVERAGE OF LAST 5 VALUES IN X ',AVGPX => F10.2)
P        /* END OF EXECUTE TEST */
```

Figure B-4

TEST OF EXECUTE CAPABILITY

```
P     MESSAGE('GN:BEGINNING OF INPUT STATEMENT TEST')
P       /* TEST OF INPUT STATEMENT */
P     FIND(CONSTANTS)      /* POSITION FILE */
P     INPUT('END OF NODE')    /* FORCE TO NEXT NODE */
P     INPUT(WORD)  ERROR: MESSAGE('GN:COULD NOT FIND WORD')
P     ENTER(INPUT,*)
P     INPUT('=')    ERROR: DO END
P     INPUT(NUMBER)    ERROR: DO END
P     ENTER(0,*)
P     FIND(INTERPOLATION_ROUTINE)
P     INPUT('=' OR WORD) ERROR: MESSAGE('GN:DID NOT FIND =')
P     INPUT(CONSTANT OR WORD OR STRING)  ERROR: MESSAGE
P          (G,'N:DID NOT FIND POLY')
P     FIND(STRING)  ERROR: DO END
P     INPUT(STRING)
P     ENTER(INPUT,STRING,*)
P     INPUT('END OF NODE') ERROR: DO END
P     INPUT(WORD)  ERROR: DO END
P     IF * = 'SIGNED_NUMBER' THEN DO
P        /* ORDER PURPOSELY WRONG */
P        ENTER(INPUT,SIGNED_NUMBER)
P        INPUT(SIGNED_NUMBER)
P         ERROR: IF INPUT('=') THEN INPUT(WORD)
P         ERROR: INPUT(SIGNED_NUMBER)
P         ERROR: MESSAGE('GN:THIS MESSAGE SHOULD     BE PRINTED')
P         ELSE MESSAGE('GN:DID NOT FIND THE = SIGN')
P        ENTER(0,*)
P        END
P        /* NOW INPUT CATENATED INPUT, I.E. AN INPUT GROUP */
P     FIND(INTERPOLATION_ROUTINE.INTERPOLATION_VALUES)
P     SAVE_POINTER
P     INPUT( '=' NUMBER ',' NUMBER)    ERROR: DO END
P     RESTORE_POINTER
P     INPUT( '=' NUMBER ',' WORD)
P          ERROR: MESSAGE('GN:THIS MESSAGE SHOULD NOT BE PRINTED')
P        /* NOW INPUT SPECIAL SYMBOLS */
P     FIND(SPECIAL_SYMBOLS) ERROR: DO END
P     ENTER(INPUT,*)
P     DO WHILE INPUT('*' OR '+' OR '-' OR ',' OR '(' OR ')' )
P        CATENATE(G,*)
P        END
P     INPUT('END OF NODE') ERROR:MESSAGE('GN:END OF NODE WASNT READ')
P     INPUT(WORD)  ERROR: DO END
P     ENTER(INPUT,*)
P     INPUT('=')  ERROR: DO END
P     MESSAGE('GN:START READING AN ARITHMETIC EXPRESSION')
P     INVOKE A_EXPR
P     FIND(DO_INPUT_TEST.SPECIAL_SYMBOLS)
P     INPUT(NUMBER)
P          ERROR: INPUT(WORD)
P          ERROR: INPUT(STRING)
P     SET(STRING = *)
P     SET(CHARACTER = SUBSTRING(STRING,5,1))
P     SET(I = 0)
P     DO FOR EACH SUBNODE OF G
P        INCREMENT(I)
P        IF INPUT(WORD '+' OR WORD '*' OR WORD ')') THEN
P           IF CHAR = * THEN MESSAGE('GN:FOUND CHARACTER, SUBCODE ',I=>I)
P           ELSE MESSAGE('GN:ONE OF TEST CHARACTERS FOUND, SUBCODE ',I=>I)
P        ELSE MESSAGE('GN:NO MATCH, CHARACTER ',*,' SUBCODE ',I=>I)
P        END
P     MESSAGE('GN:END OF INPUT STATEMENT TEST')
```

Figure B-5

TEST OF INPUT CAPABILITY

includes not only the reading of the input groups but also this capability combined in an OR construct. Note also that all message statements have been changed to include the type identifier in the first parameter. This consolidation not only simplifies the format of the statement but also allows for error messages to be referenced as table entries. The SAVE_POINTER and RESTORE_POINTER statements are also included in this test.

An expanded version of the OUTPUT statement test is listed in figure B-6. Exercised quite extensively in the actual execution of the LINK4 FCF, the OUTPUT capability was not tested thoroughly in the original test. This test has been upgraded to correct this and provide an independent, fast test of the more explicit output functions. Included are uses of the end of record and end of file and formatting capabilities. In addition, tests of double and triple substitutions during pass 1 and pass 2 are used. Synthesized as well as modified FORTRAN statements are also output. Several FORMAT statements are included which use many variations of legal FORTRAN formats as a test of the format lexical analyzer.

Figure B-7 lists additional enhancements to the symbol table manipulation capability. Specifically, changes were made to test the equivalencing of symbol table names using the DEFINE statement. Changes made to the REPLACE statement were also included in the test. The DEFINE statement tests includes chaining, nullifying and re-equivalencing to test the various functions of the statement. In addition, both array and regular symbol tables are included in the test. The REPLACE statement uses both global and a specified number of replaces. Also included was the use of DEFAULT to reference the default symbol table.

The FIND statement test, Figure B-8, was modified to check out the redefined operation of the FIND instruction, notably that the input pointer is only repositioned if the identified node is actually found. Note that tests identifying both conditions are made.

```
P        /* TEST OF FORTRAN OUTPUT */
      WRITE(1,1000) INPUT
1000  FORMAT(40A2///)
      IF(N.EQ.3) GO TO 7
C        THIS IS AN IN LINE COMMENT
      DIMENSION A(17,3)
15    X = SQRT(7.3*10.)**(7.6+Y**4)
      A(3,5) = EXP(A(1,2))*5.+Z-3.
P     ENTER(PASS1,#JBUF,'(#J)')
P     SET(N = 7.3*14.+5.)
P        /* N AS DIMENSION SHOULD BE 99 */
P     ENTER(PASS1,#J,'#N')
P     ENTER(PASS2,#N,VALUE(N)=>I)
      X#JBUF = Y#JBUF * 5.
      EQUIVALENCE(JBUF(#J),IO)
      XEND =     7.3 * X(#J)
     1         + 2.9 * X(1)
P     DELETE(PASS1,#JBUF)
P     DELETE(PASS1,#J)
P        /* NOW SYNTHESIZE A STATEMENT */
P     OUTPUT('25    CONTINUE',#)
P     OUTPUT('IF (Y.EQ.0.) CALL SUBR(J,X)',#)
P     OUTPUT('1000')
P     OUTPUT('FORMAT(I5,G14.7,3(3X,F10.4/))',#)
P     ENTER(OUT_TAB,STATEMENT,'IF(J.GE.7) X=7.3*Z')
P     OUTPUT('75',VALUE(OUT_TAB,STATEMENT),#)
P     OUTPUT(COMMENT:,' THIS IS A GENERATED COMMENT',#)
P     SET(OUTPUT = 'DATA')
P     OUTPUT(COL(6),7.3 FORMAT(F5.2),COL(21),(3.7+8.5)*5. FORMAT(I),#)
P     OUTPUT(END_OF_RECORD)
P     OUTPUT(COMMENT:,'  GENERATED STATEMENT',#)
P     OUTPUT(C:,COL(10),VALUE(OUT_TAB,STATEMENT)=>A17,'***',#)
P     OUTPUT(C:,'-----PREVIOUS COMMENT SHOULD END WITH ****',#)
P        /* TEST AUTOMATIC OUTPUT OF BUFFER ON EOR */
P     OUTPUT('1000CONTINUE',EOR)
P     OUTPUT(C:,'-----PREVIOUS CARD SHOULD BE INCORRECT FTN, OUTPUT AS DATA',#)
P     OUTPUT(10.3**.5+36.3*(7.4-1.2) => I,COL(60),N FORMAT(F10.5),EOF)
P     SET(OUTPUT = 'FORTRAN')
10                          X =      3      +      2
C-------------------NEXT FORTRAN IS COMPLICATED FORMAT STATEMENT
200   FORMAT(10X,//17(F10.9,3(5X,G14.7,I7,5A13),5HTEST1/7E7.4,2L1))
C          STATEMENT WITH SEVERAL PRETTY CONTINUATION CARDS
      DATA A/ 7.30, 15.62, 17.32, 24.50, 78.93,
     1        87.33,122.60, 14.95,399.21, 99.68,
     2        89.00, 23.44,673.96,  3.54, 92.04,
     3        74.95,895.74,808.08, 44.98, 74.99/
      DIMENSION Z(#N),X(2,#N),AREA(15),W(#N,#N)
P     MESSAGE('GN:END OF OUTPUT TEST')
```

Figure B-6

```
      DO FOR SYMBOL TABLE SYM_TAB
         SET(N = 'TABLE ENTRY')
         STATUS('INITIAL VALUE OF N ',SYM_TAB,N)
         DELETE(N)
         STATUS('N AFTER DELETE',SYM_TAB,N)
         SET(N='TABLE VALUE')
         SET(M='SYMBOL ENTRY')
         SET(X = 'IN SYM_TAB')
         STATUS('TABLE AFTER INITIAL SETUP',SYM_TAB)
         REPLACE(SYM_TAB,N,'ENTRY',SUBSTR(INDEX(VALUE(SYM_TAB,'),5,7))
         STATUS('N AFTER REPLACE WITH PART OF X',SYM_TAB,N)
         CATENATE(SYM_TAB,M,VALUE(X))
         STATUS('M AFTER CATENATE OF X',SYM_TAB,M)
         REPLACE(SYM_TAB,M,VALUE(Y),'')
         STATUS('M AFTER REMOVAL OF Y',SYM_TAB,M)
         SET(M)
         REPLACE(B,'','A ')
         STATUS('M AFTER PREFIX ' ADDED',SYM_TAB,M)
         DELETE(X)
         DELETE(SYM_TAB,Y)
         STATUS('SYM_TAB AFTER X AND N DELETED',SYM_TAB)
         SET(A = (7.3+2.7)**2)
         SET(N = 5.**M)
         SET(A = 'N')
         STATUS('SYM_TAB AFTER M,N,A SET',SYM_TAB)
         SET(VALUE(A) = (20+10)*5+10.**2)
         STATUS('SYM_TAB AFTER INDIRECT ENTER',SYM_TAB)
         SET(D = 'A')
         SET(VALUE(VALUE(D)))
         ENTER(B,C)
         STATUS('SYM_TAB AFTER DOUBLE INDIRECT SET',SYM_TAB)
         END
      SET(I=1)
      DO WHILE I<=5
         INCREMENT(I)
         END
      STATUS('VALUE OF I AFTER INCREMENT',FLTOPS,I)
      DO WHILE I>0
         DECREMENT(I)
         END
      STATUS('VALUE OF I AFTER DECREMENT',FLTOPS,I)
         /* TEST OF DEFINE/EQUIVALENCE STATEMENT */
      DEFINE SYMBOLS(7)
      FIND(DO_INPUT_TEST.SPECIAL_SYMBOLS)
      SET(I=0)
      DO FOR EACH SUBNODE OF B
         INPUT(WORD)
         INPUT('+' OR '-' OR '*' OR '(' OR ')' OR '/' OR '**')
         INCREMENT(I)
         ENTER(SYMBOLS(I),*)
         END
      STATUS('SYMBOLS AFTER INPUT COMPLETE',SYMBOLS)
      DEFINE SPECIAL AS SYMBOLS
      STATUS('SPECIAL DEFINED AS SYMBOLS',SPECIAL)
      ENTER(SPECIAL(2),'>')
      ENTER(SPECIAL(5),'<')
      STATUS('SPECIAL AFTER 2 AND 5 MODIFIED',SPECIAL)
      DEFINE TABLE AS SPECIAL
      STATUS('OUTPUT OF TABLE, SECOND ITEM IN CHAIN',TABLE)
      DEFINE SPECIAL AS NULL
      ASSERT TABLE_EMPTY(TABLE) ERROR: MESSAGE('GW:TABLE NOT EMPTY')
      DEFINE NEW AS SYMBOLS
      STATUS('NEW TABLE NAME ASSIGNED',NEW)
      CATENATE(NEW(2),'=')
      CATENATE(NEW(5),'=')
      STATUS('NEW AFTER 2 AND 5 MODIFIED',NEW)
         /* TEST OF FANCY REPLACES */
      SET(A = 'AAAAAAAA')
      REPLACE(DEFAULT,A,'A','C')
      MESSAGE('GW: A = CCCCCCC ',A)
      REPLACE(DEFAULT,A,'CC','B')
      MESSAGE('GW: A = BBBC ',A)
      REPLACE(DEFAULT,A,'C','')
      MESSAGE('GW: A = BB ',A)
      REPLACE(DEFAULT,A,'B','',2)
      REPLACE(DEFAULT,A,'','7',7)
      MESSAGE('GW: A = 7777770 ',A)
      IF INDEX(A,'M') <> 7 THEN MESSAGE
         ('GW:M NOT POSITIONED AS 7TH CHARACTER OF A',A=>A15)
      IF LENGTH(A) <> INDEX(A,'0') THEN MESSAGE
         ('GW:0 IS NOT LAST CHARACTER OF A ',A)
      MESSAGE('GW:END OF SYMBOL TABLE TEST')
```

Figure B-7

```
P      MESSAGE('GN:BEGINNING OF FIND TEST')
P         /* TEST OF INPUT FILE SEARCH -- FIND */
P      FIND(GEOPHYSICAL_MODEL.EARTH.NON_ROTATING)
P           ERROR: MESSAGE('GW:CANNOT FIND GEOPHYSICAL_MODEL, ET AL')
P      FIND(GEOPHYSICAL_MODEL) ERROR: DO END
P      FIND(@.EARTH)  ERROR: DO END
P      FIND(@.NON_ROTATING)  ERROR: DO END
P      FIND(INTERPOLATION_ROUTINE)  ERROR: DO END
P      SAVE_POINTER
P      FIND(@.TABLE_INFORMATION.DATA)  ERROR:
P         MESSAGE('GW:COULD NOT FIND TABLE DATA')
P      RESTORE_POINTER
P         /* NOW FIND A BROTHER NODE */
P      FIND(@.INTERPOLATION.VALUES)
P         ERROR: MESSAGE('GW:IMPROPER FIND FUNCTION')
P         /* FAILURE TO FIND SPECIFIC NODE -- TAKE ERROR PATH */
P      FIND(GEOPHYSICAL_MODEL.VENUS.ELLIPSE)
P         ERROR: MESSAGE('GN:NODE NOT FOUND -- PLANNED ERROR')
P      FIND(GEOPHYSICAL_MODEL)
P      FIND(@.EARTH.SPHERICAL.NEXT_LEVEL.ANOTHER_LEVEL)
P         ERROR: MESSAGE('GN:THIS MESSAGE SHOULD BE PRINTED')
P      IF NOT FIND(@.EARTH.SPHERICAL) THEN
P           MESSAGE('GW:IMPROPER OPERATION OF FIND')
P         /* TEST ASSERT ERROR BRANCH */
P      ASSERT NODE_EXISTS(REQUIRED_COMPUTATIONS.ENERGY_WEIGHT)
P         ERROR: DO
P            FIND(REQUIRED_COMPUTATIONS.ENERGY_HEIGHT)
P               ERROR: MESSAGE('GW:ENERGY_HEIGHT NODE NOT FOUND')
P            MESSAGE('GN:ASSERTION FALSE -- ENERGY_WEIGHT NOT FOUND')
P            END
P         /* SCAN PAST COMPLICATED ERROR ASSERTION */
P      ASSERT NOT FIND(REQUIRED_COMPUTATIONS.ENERGY_WEIGHT)
P         ERROR: DO FOR EACH SUBNODE OF CONSTANTS
P                   INPUT(WORD CONSTANT)
P                      ERROR: DO WHILE INPUT(CONSTANT)
P                                 ENTER(FLTOPS,*,*)
P                                 IF INPUT(',') THEN DO END
P                                 END
P                   END
P      MESSAGE('GN:END OF FIND STATEMENT TEST')
```

Figure B-8

The appearance of ASSERT, STATUS and MESSAGE commands throughout these tests have been increased as a method of testing the internal debugging capabilities of the precompiler. Extensive uses of these capabilities have proved worthwhile even in the development of the test cases themselves.

B.4 DESIGN CHANGES

There have been several design changes that have resulted from the extended validation effort. Figure B-9 lists these changes. Most notably, execution of the EXECUTE statement test identified several linking errors that existed between primitives and rules as identified in the original design. Specifically, some additional bookkeeping operations were necessary to insure that proper accounting was done on the addresses of parameters passed into an execute routine. In addition, the use of the ARRAY-ELEMENTS rule was somewhat altered to insure an unambiguous use of array structures when defining and using array tables.

Another significant change was the identification of the need for modifying the basic operation of the FIND command. Although adequate as originally conceived, it was discovered that programmer errors could be reduced and programmer acceptance commensurately increased by slightly altering its operation. Instead of always moving the input pointer in search of the place indicated by the command, the precompiler now alters the input pointer position only if the FIND was successfully executed. This insures that the programmer always knows the pointer position and enables him to considerably simplify input file searches.

Still another change made was the addition of a capability to access the default symbol table by using the key word DEFAULT. Useable from several statements such as REPLACE and CATENATE, this affords access to the default symbol table from most symbol table operations. The single

exception is the CLEAR-TABLE command. By using the key word DEFAULT in the table name position, the user can indirectly reference entries. Note that DEFAULT cannot refer to an array table, and, as a result, cannot be a subscripted entity.

It was found that a change in the definition of an FCF identifier was needed to decrease probability of user errors. In the design, the # can be used in the in-line FORTRAN code as a placeholder for eventual substitution. The need for the capability of having symbol entries in the PASS1 and PASS2 symbol tables with the # as the first character became clear. Although the ability to enter these symbol names as strings was a part of the original design, a preponderance of user errors resulted in the test FCF's because of a natural bent to leave off the string delimiters. The obvious solution was to allow identifiers to begin with #. The # by itself is not an identifier, but is still a special symbol.

The DO FOR EACH SUBNODE statement was also found to not operate properly when they were nested. A change was employed which effectively does an implied save and restore input pointer at the time the loop is begun. Thus, after the input pointer is positioned at the proper place, i.e., the node for which each subnode is to be processed, the input pointer is saved. At the completion of the activity on the last subnode, the input pointer is repositioned to the saved location.

DESIGN ADDITION/CHANGE	RESULT
Add ARITH-DEPTH-COUNTER variable to several rules.	Allows for proper parameter location bookkeeping for EXECUTE statements
Modify function of ARRAY-ELEMENTS rule	Allows for proper evaluation of subscripts when used to identify array bounds and array position
Add DEFAULT key word	Allows all symbol table operations except CLEAR-TABLE to reference the default symbol table indirectly
Alter operation of FIND statement	Only successful FIND operations reposition input pointer. User never loses track of input pointer position
Modify definition of an identifier in the FCF	A # followed by at least one alphanumeric character can be an identifier, resulting in fewer FCF syntax errors
alter syntax and function of REPLACE statement	REPLACE does a global replace of identified string. Optionally, the user can specify a 5th parameter giving a number of replaces
Allow nesting of DO FOR EACH SUBNODE OF rules	The user can now nest DO FOR EACH statements to an arbitrary depth. Input pointer repositioned to location at start of loop after last subnode

RULES ADDED: NONE

RULES MODIFIED: FIND_STATEMENT, DO_FOR_EACH, STATUS_STATEMENT, DELETE_STATEMENT, ARRAY ELEMENTS,DEFINE_STATEMENT, INDEX_FUNCTION, LENGTH_FUNCTION, MESSAGE_ STATEMENT, PARAMETERS_STATEMENT, EXECUTE_STATEMENT, LOCATION_POINTER, INDIRECT_LOCATION_POINTER, ALL ARITHMETIC EXPRESSION RULES,

PRIMITIVES ADDED: NONE

PRIMITIVES MODIFIED: .REPLACE

Figure B-9

FLTOPS DESIGN CHANGES

Extended Validation Effort

APPENDIX C

LINK4 FCF AND TEST RESULTS

Appendix B describes the FCF that was coded by AFFTC personnel to customize the UFTAS LINK4 program. This FCF was executed through the FLTOPS emulator with four FLTOPS INPUT files for testing. The following pages contain listings of the FCF, figure C-1, and the results of Test 4, figure C-2. See figure B-3, page B-11, for a listing of the Test 4 INPUT file.

```
P       /*****************************************************/
P       /******          UFTAS LINK4 FCF              ******/
P       /*****************************************************/
P
P         MAIN:    /* UFTAS DRIVER */
P
P         IF NCDE_EXISTS (CONSTANTS.STANDARD.
P                         USE_ALFA_IN_THRUST_AND_DRAG_CALCULATIONS) THEN
P            ENTER(UFTAS,THRUST,'YES')
P         ELSE ENTER(UFTAS,THRUST,'NO')
P         INVOKE LINK4
P         END
```

Figure C-1
FCF FOR LINK4
(page 1 of 41)

```
P       LINK4:    /* LINK4 SETUP */
P
P         DO FOR SYMBOL TABLE LINK4
P            INVOKE LINK4_INITIAL
P            IF DAW <> 'NO' THEN DO
P               INVOKE LINK4B
P               INVOKE INPUT4B
P               INVOKE DALAC
P               END
P            ELSE DO
P               INVOKE LINK4A
P               INVOKE INPUT4
P               END
P            INVOKE SFEXT
P            IF DAU <> 'NO' OR DAQ <> 'NO' THEN INVOKE DALFR
P            IF DABB <> 'NO' THEN INVOKE DABB
P            INVOKE SNXNZ
P            END
P         STATUS('LINK4 SYMBOL TABLE', LINK4)
P         STATUS('PASS1 SYMBOL TABLE',PASS1)
P                                                  /* CLEAR TABLES */
P         CLEAR_TABLE(LINK4)
P         DELETE(PASS1,#LDIN)
P         DELETE(PASS1,#LDBUF)
P         DELETE(PASS1,#LDOUT)
P         DELETE(PASS1,#IBUF)
P         DELETE(PASS1,#IDIM)
P         DELETE(PASS1,#COML4DAT)
P         DELETE(PASS1,#COMCURVE)
P         END  LINK4
```

Figure C-1
FCF FOR LINK4
(page 2 of 41)

```
P       LINK4_INITIAL:    /* LINK4 INITIALIZATION */
P
P             /*  READ INPUT FOR LINK4 AND SETUP INTERNAL FLAGS AND SYMBOLS  */
P             /*  CHECK FOR ERRORS                                           */
P             /*  SETUP COMMON BLOCKS                                        */
P             /*  CREATE DEFAULT INITILIZATION STATEMENTS                    */
P             /*  SETUP B-FILE I/O DATA STATEMENTS                           */
P
P       FIND (LINK4. CONSTANTS)
P       INVOKE L4CONTAB   /*  LINK4 CONSTANTS TAB  */
P       INVOKE CON_READ
P       CLEAR_TABLE(CONST)
P       FIND (LINK4. ANGLE_OF_ATTACK)
P
P       IF FIND 3.LOOKUP THEN  DO              /*  ALPHA BY LOOKUP  */
P          FIND 3.FUNCTION_OF
P          INVOKE FUN_READ     /*  ALOOKUP.KFALK,KLALF  */
P          SET (KFALF = KFIRST)
P          SET (KLALF = KLAST)
P          SET (ALOOKUP = FUN_TYP)
P          SET (DAU = 'NO')
P          SET (DAQ = 'NO')
P          SET (DABB = 'NO')
P          SET (DAW = 'NO')
P          END
P       ELSE IF FIND (3.CORRECTIONS) THEN
P                          INVOKE ACORRECT  /*  ALPHA BY CORRECTIONS  */
P       ELSE MESSAGE (I,'S:ANGLE_OF_ATTACK NEEDS EITHER LOOKUP'
P                     ,' OR CORRECTIONS SUBNODE')
P       FIND (LINK4. LOAD_FACTORS. CONSTANTS)
P                     ERROR: MESSAGE(I,'S:LOAD_FACTORS NOT AVAILABLE')
P       INVOKE L4LFTAB    /*  LINK4 LOAD_FACTOR TAB  */
P       INVOKE CON_READ
P       CLEAR_TABLE(CONST)
P       IF FIND (LINK4.ADDITIONAL_INPUTS) THEN DO
P          SET(NAME = 'IN_PARN')
P          SET(UNITS = 'IN_PARU')
P          INVOKE PAR_READ
P          SET(N_IN_PAR = NPAR)
P          END
P       IF FIND (LINK4.ADDITIONAL_OUTPUTS) THEN DO
P          SET(NAME = 'OUT_PARN')
P          SET(UNITS = 'OUT_PARU')
P          INVOKE PAR_READ
P          SET(N_OUT_PAR = NPAR)
P          END
P       INVOKE  L4PARIO   /*  SETUP B FILE I/O COMMON AND DATA STMTS */
P       INVOKE  L4_COMMON /*  SETUP LINK4 COMMON BLOCKS */
P
P       ENTER(PASS1,#LDIN,N_IN_PAR)
P       ENTER(PASS1,#LDBUF,'60')
P       ENTER(PASS1,#LDOUT,N_OUT_PAR)
P       IF DAW='YES' THEN DO
P          ENTER(PASS1,#IBUF,'(I)')
P          ENTER(PASS1,#IDIM, '(#LDBUF)' )
P          END
P       ELSE DO
P          ENTER(PASS1,#IBUF,'')
P          ENTER(PASS1,#IDIM,'')
P          END
P       END  LINK4_INITIAL
```

Figure C-1
FCF FOR LINK4
(page 3 of 41)

```
P       ACORRECT:  /* ALFA TO BE FOUND BY CORRECTIONS TO MEASURED ALFA */
P
P       SET(ALOOKUP = 'NO')
P       FIND LINK4.ANGLE_OF_ATTACK.CORRECTIONS
P          SAVE_POINTER
P
P          FIND @.DYNAMIC_LAG  ERROR: MESSAGE(I,'S: DYNAMIC_LAG NOT SPECIFIED')
P             INVOKE DAW_SETUP
P             RESTORE_POINTER
P             SAVE_POINTER
P
P          FIND @.BOOM_BENDING ERROR: MESSAGE(I,'S: BOOM_BENDING NOT SPEIFIED')
P             IF FIND @.NO THEN SET (DABB = "NO')
P             ELSE IF FIND @.YES THEN DO
P                DO FOR EACH SUBNODE OF @
P                   IF INPUT("CONSTANTS') THEN DO
P                      INVOKE L488TAB
P                      INVOKE CON_READ
P                      END
P                   ELSE IF INPUT('FUNCTION_OF') THEN DO
P                      INVOKE FUN_READ
P                      SET(KDABB = KFIRST)
P                      END
P                   ELSE MESSAGE(I,'S: INVALID INPUT UNDER BOOM_BENDING")
P                   END
P                SET(DABB = 'YES')
P                END
P             ELSE MESSAGE(I,'S: BOOM_BENDING MUST BE EITHER "YES" OR "NO""')
P             SAVE_POINTER
P             RESTORE_POINTER
P
P          FIND @.PITCH_RATE  ERROR: MESSAGE(I,'S: PITCH_RATE NOT SPECIFIED')
P             INPUT(WORD)
P             IF FIND @.YES OR FIND @.NO THEN  SET(DAC = 0)
P             ELSE MESSAGE(I,'S: UPWASH MUST BE EITHER "YES" OR "NO"")
P             SAVE_POINTER
P             RESTORE_POINTER
P
P          FIND @.UPWASH  ERROR: MESSAGE(I,'S:UPWASH NOT SPECIFIED')
P             IF FIND @.FUNCTION_OF THEN DO
P                INVOKE FUN_READ
P                SET (DAU = FCN)
P                SET (KFDAU = KFIRST)
P                SET (KLDAU = KLAST)
P                END
P             ELSE IF FIND @.NO THEN SET(DAU = "NO')
P             ELSE MESSAGE(I,"S:INVALID INPUT UNDER UPWASH')
P       END  ACORRECT
```

Figure C-1
FCF FOR LINK4
(page 4 of 41)

```
P       DAW_SETUP:   /* DYNAMIC LAG CORRECTIONS */
P
P       INPUT('END OF NODE')
P       IF INPUT ('NO') THEN SET (DAW = 'NO')
P       ELSE DO
P          INPUT(WORD)
P          IF * = 'VANE_ACCELEROMETER_SYSTEM' THEN
P             SET (DAW = 'YES')
P          ELSE IF SUBSTRING(*, 1, 4) = 'VANE' THEN DO
P             MESSAGE(I,'W: ILLEGAL NODE UNDER DYNAMIC_LAG')
P             SET (DAW = 'YES')
P             END
P          ELSE DO
P             MESSAGE(I,'W: NO VALID INPUT UNDER DYNAMIC_LAG.')
P             SET (DAW = 'NO')
P             END
P          END
P       IF DAW = 'YES' THEN DO
P          SAVE_POINTER
P          FIND (3,FREQUENCY) ERROR: MESSAGE(I,'S: FREQUENCY NOT SPECIFIED')
P             INPUT ('END OF NODE')
P             IF INPUT('COMPUTE') THEN DO
P                SET (FREQ = 4)
P                INPUT ('END OF NODE')
P                INPUT('CONSTANTS')
P                INVOKE L4DLTAB
P                INVOKE CON_READ
P                END
P             ELSE DO
P                SET(FREQ = 'CURVE')
P                INPUT ('FUNCTION_OF') ERROR:MESSAGE(I,'S:FREQUENCY ERROR')
P                INVOKE FUN_READ
P                END
P          RESTORE_POINTER
P          SAVE_POINTER
P          FIND (3,DAMPENING_RATIO)
P             INPUT ('END OF NODE')
P             IF INPUT('COMPUTE') THEN SET (DAMP =0)
P             ELSE DO
P                SET(DAMP = 'CURVE')
P                INPUT ('FUNCTION_OF') ERROR: MESSAGE(
P                         'IS NO DAMPENING_RATIO' )
P                INVOKE FUN_READ
P                END
P          RESTORE_POINTER
P          FIND (3,CONSTANTS) ERROR: MESSAGE(
P                                    'IS DYNAMIC LAG CONSTANTS NOT SPECIFIED')
P             INVOKE L4DLTAB
P             INVOKE CON_READ
P          END
P       END  DAW_SETUP
```

Figure C-1
FCF FOR LINK4
(page 5 of 41)

```
P       CON_READ:   /* CONSTANTS READ RULE                                  */
P                   /*  CONST(I,1) = VALID USER INPUT                       */
P                   /*  CONST(I,2) = NULL, WORD, OR CONSTANT                */
P                   /*  CONST(I,3) = INTERNAL SYMBOL NAME                   */
P                   /*  ALL CONST(I,1) ARE CHECKED UNTIL NULL FOUND         */
P                   /*                                                      */
P          DO FOR EACH SUBCODE OF 8
P             INPUT(WORD)
P             SET (I = 1)
P             DO WHILE CONST(I,1) <> '' AND CONST(I,1) <> *
P                INCREMENT(I)
P                END
P             IF CONST(I,1) = '' THEN MESSAGE(I,'W:INVALID INPUT')
P             ELSE IF CONST(I,2) = '' THEN
P                   SET (VALUE(CONST(I,3)) = 'YES')
P             ELSE DO
P                INPUT('=')
P                IF VALUE(CONST(I,2)) = 'WORD'  THEN INPUT(WORD)
P                ELSE INPUT(CONSTANT)
P                SET(VALUE(CONST(I,3)) = *)
P                IF VALUE(CONST(I,2)) = 'WORD' THEN DO
P                   SET (VALUE(CONST(I,3)))
P                   DO WHILE INPUT (',' CR '/')
P                      CATENATE(3,*)
P                      INPUT(WORD)
P                      CATENATE(3,*)
P                      END
P                   END
P                END
P             END
P          SET (I = 1)
P          DO WHILE CONST(I,1) <> ''
P             IF CONST(I,2) = '' AND VALUE(VALUE(CONST(I,3))) <> 'YES'
P                   THEN SET (VALUE(CONST(I,3)) = 'NO')
P             ELSE IF VALUE(VALUE(CONST(I,3))) = '' THEN
P                     MESSAGE(I,'S:CONSTANT NOT AVAILABLE')
P             INCREMENT(I)
P          END
P       END  CON_READ
```

Figure C-1
FCF FOR LINK4
(page 6 of 41)

```
P       L4CONTAB:    /* LINK4,CONSTANTS DEFINITION  */
P
P       DEFINE CONST(14,3)
P       ENTER(CONST(1,1),'DISTANCE_FROM_VANE_LEADING_EDGE_TO_MAC')
P       ENTER(CONST(2,1),'ALFA_CONVERGENCE_CRITERION')
P       ENTER(CONST(3,1),'ALFA_ITERATION_LIMIT')
P       ENTER(CONST(4,1),'CL_CONVERGENCE_CRITERION')
P       ENTER(CONST(5,1),'CL_ITERATION_LIMIT')
P       ENTER(CONST(6,1),'LOAD_FACTOR_DATA_SOURCE')
P       ENTER(CONST(7,1),'REFERENCE_ACCELERATION_DUE_TO_GRAVITY')
P       ENTER(CONST(8,1),'THETA_EXPONENT')
P       ENTER(CONST(9,1),'ALFA_NON_CONVERGENCE')
P       ENTER(CONST(10,1),'CL_NON_CONVERGENCE')
P       ENTER(CONST(11,1),'RECOMPUTE_CL')
P       ENTER(CONST(12,1),'COMPUTE_CD')
P       ENTER(CONST(13,1),'CORRECT_FOR_PITCH_ACCELERATION')
P       ENTER(CONST(14,1),'')
P       ENTER(CONST(1,2),'CONSTANT')
P       ENTER(CONST(2,2),'CONSTANT')
P       ENTER(CONST(3,2),'CONSTANT')
P       ENTER(CONST(4,2),'CONSTANT')
P       ENTER(CONST(5,2),'CONSTANT')
P       ENTER(CONST(6,2),'WORD')
P       ENTER(CONST(7,2),'CONSTANT')
P       ENTER(CONST(8,2),'CONSTANT')
P       ENTER(CONST(9,2),'WORD')
P       ENTER(CONST(10,2),'WORD')
P       ENTER(CONST(11,2),'')
P       ENTER(CONST(12,2),'')
P       ENTER(CONST(13,2),'')
P       ENTER(CONST(1,3),'XLVC')
P       ENTER(CONST(2,3),'CVALF4')
P       ENTER(CONST(3,3),'ITRALF')
P       ENTER(CONST(4,3),'CVCLT')
P       ENTER(CONST(5,3),'ITRCLT')
P       ENTER(CONST(6,3),'ISOURC')
P       ENTER(CONST(7,3),'GR')
P       ENTER(CONST(8,3),'ETHETA')
P       ENTER(CONST(9,3),'LCVALF')
P       ENTER(CONST(10,3),'LCVCLT')
P       ENTER(CONST(11,3),'LCLT')
P       ENTER(CONST(12,3),'LCDT')
P       ENTER(CONST(13,3),'LODCT')
P       END  L4CONTAB
```

Figure C-1
FCF FOR LINK4
(page 7 of 41)

```
P       L4DLFTAB:   /*  LINK4 DNAMIC LAG-FREQUENCY CONSTANTS DEFINITION  */
P
P       DEFINE CONST(3,3)
P       ENTER(CONST(1,1),'CHARACTERISTIC_AREA_OF_BODY')
P       ENTER(CONST(2,1),'MOMENT_OF_INERTIA_OF_VANE_ACCELEROMETER')
P       ENTER(CONST(3,1),'')
P       ENTER(CONST(1,2),'CONSTANT')
P       ENTER(CONST(2,2),'CONSTANT')
P       ENTER(CONST(1,3),'ABODY')
P       ENTER(CONST(2,3),'XIYY')
P       END  L4DLFTAB
```

Figure C-1
FCF FOR LINK4
(page 8 of 41)

```
P       L4DLTAB: /* LINK4 DYNAMIC LAG CONSTANTS DEFINITION */
P
P       DEFINE CONST(8,3)
P       ENTER(CONST(1,1),'ALFA_CONVERGENCE_CRITERION')
P       ENTER(CONST(2,1),'ALFA_ITERATION_LIMIT')
P       ENTER(CONST(3,1),
P                   'NUMBER_OF_POINTS_REQUIRED_TO_MEET_CNVERGENCE_CRITERION')
P       ENTER(CONST(4,1),'CHARACTERISTIC_AREA_OF_VANE')
P       ENTER(CONST(5,1),'DCL/DALFA_FOR_VANE')
P       ENTER(CONST(6,1),'GAIN_CONSTANT')
P       ENTER(CONST(7,1),'DISTANCE_FROM_VANE_CG_TO_VANE_AERODYNAMIC_CENTER')
P       ENTER(CONST(8,1),'')
P       ENTER(CONST(1,2),'CONSTANT')
P       ENTER(CONST(2,2),'CONSTANT')
P       ENTER(CONST(3,2),'CONSTANT')
P       ENTER(CONST(4,2),'CONSTANT')
P       ENTER(CONST(5,2),'CONSTANT')
P       ENTER(CONST(6,2),'CONSTANT')
P       ENTER(CONST(7,2),'CONSTANT')
P       ENTER(CONST(1,3),'ITRDAW')
P       ENTER(CONST(2,3),'CVDAW')
P       ENTER(CONST(3,3),'PCTDAW')
P       ENTER(CONST(4,3),'AVANE')
P       ENTER(CONST(5,3),'CLAV')
P       ENTER(CONST(6,3),'XKT')
P       ENTER(CONST(7,3),'XL')
P       END  L4DLTAB
```

Figure C-1
FCF FOR LINK4
(page 9 of 41)

```
P          L4BBTAB:     /*  LINK4 BOOM BENDING CONSTANTS  */
P
P          DEFINE CONST(4,3)
P          ENTER(CONST(1,1),'CHARACTERISTIC_AREA_OF_BOOM')
P          ENTER(CONST(2,1),'DCL/DALFA_FOR_BOOM')
P          ENTER(CONST(3,1),'WEIGHT_OF_BOOM')
P          ENTER(CONST(4,1),'')
P          ENTER(CONST(2,2),'CONSTANT')
P          ENTER(CONST(3,2),'CONSTANT')
P          ENTER(CONST(1,3),'ABOOM')
P          ENTER(CONST(2,3),'CLAB')
P          ENTER(CONST(3,3),'WBOOM')
P          END  L4BBTAB
```

Figure C-1
FCF FOR LINK4
(page 10 of 41)

```
P          L4LFTAB:    /*  LINK4 LOAD FACTORS CONSTANTS  */
P
P          DEFINE CONST(4,3)
P          ENTER(CONST(1,1),'DATA_SOURCE')
P          ENTER(CONST(2,1),'CG_ACCELEROMETER_MISALIGNMENT_ANGLE')
P          ENTER(CONST(3,1),'BCOM_ACCELEROMETER_MISALIGNMENT_ANGLE')
P          ENTER(CONST(4,1),'')
P          ENTER(CONST(1,2),'NUMBER')
P          ENTER(CONST(2,2),'NUMBER')
P          ENTER(CONST(3,2),'NUMBER')
P          ENTER(CONST(1,3),'JAXIS')
P          ENTER(CONST(2,3),'ACCMA')
P          ENTER(CONST(3,3),'ACCMAP')
P          END  L4LFTAB
```

Figure C-1
FCF FOR LINK4
(page 11 of 41)

```
P       PAR_READ:   /* READ B-FILE PARAMETER NAMES AND UNITS */
P
P             /* NAME = NAME OF TABLE FOR STORING NAMES   */
P             /* UNITS = NAME OF TABLE FOR STORING UNITS  */
P             /* NPAR = NUMBER OF PARAMETERS READ (OUTPUT)*/
P
P       /* FIRST COUNT NUMBER OF PARAMETERS */
P         SAVE_POINTER
P         INPUT(WORD OR STRING)  ERROR: MESSAGE(1,'*** NO PARAMETERS SPECIFIED')
P         SET (NPAR = 1)
P         IF INPUT('(') THEN DO
P            INPUT(WORD OR STRING)
P            INPUT(')')
P            END
P         DO WHILE INPUT(',')
P            INCREMENT(NPAR)
P            INPUT(WORD OR STRING)
P            IF INPUT('(') THEN DO
P               INPUT(WORD OR STRING)
P               INPUT(')')
P               END
P            END
P
P       /* NOW READ IN NAMES AND UNITS */
P         RESTORE POINTER
P         DEFINE VALUE(NAME)(NPAR), VALUE(UNITS)(NPAR)
P         INPUT(WORD OR STRING)
P         ENTER(VALUE(NAME)(1) , *)
P         IF INPUT ('(') THEN DO
P            INPUT(WORD OR STRING)
P            ENTER(VALUE(UNITS)(1) , *)
P            INPUT(')')
P            END
P         ELSE ENTER(VALUE(UNITS)(I) , '')
P         SET (NPAR = 1)
P         DO WHILE INPUT(',')
P            INPUT(WORD OR STRING)
P            INCREMENT(NPAR)
P            ENTER(VALUE(NAME)(NPAR) , *)
P            IF INPUT ('(') THEN DO
P               INPUT(WORD OR STRING)
P               ENTER(VALUE(UNITS)(NPAR) , *)
P               INPUT(')')
P               END
P            ELSE ENTER(VALUE(UNITS)(NPAR) , '')
P            END
P         END PAR_READ
```

Figure C-1
FCF FOR LINK4
(page 12 of 41)

```
P       FUNREAD:   /* READ CURVE NUMBERS OR SETUP A TABLE FOR LOOKUP */
P
P                  /* OUTPUT                                          */
P                  /*    FUN_TYP = 'CURVE' OR 'TABLE'                 */
P                  /*    KFIRST = FIRST CURVE NUMBER                  */
P                  /*    KLAST = LAST CURVE NUMBER                    */
P                  /*    DIMENSION = CURVE OR TABLE DIMENSION         */
P                  /*    FCN = WHAT FOLLOWS 'FUNCTION_OF'             */
P                  /*    TABLE = TABLE (IF USED)                      */
P       INPUT(WORD OR '(')
P       SET(FCN = *)
P       IF * = '(' THEN DO
P          INPUT(WORD)
P          CATENATE(8,*)
P          END
P       SET(DIMENSION = 2)
P       SET(FCN)
P       DO WHILE INPUT(',' OR '+' OR '-' OR '*' OR '/' OR '**' OR ')' OR ')'
P                      OR WORD)
P          IF * = ',' THEN  INCREMENT(DIMENSION)
P          CATENATE(DEFAULT,FCN, *)
P          END
P       INPUT('END OF NODE')
P       IF INPUT('CURVE_NUMBER') THEN DO
P          SET(FUN_TYP = 'CURVE')
P          INPUT(NUMBER OR WORD)
P          SET(KFIRST = *)
P          IF DIMENSION = 4 THEN DO
P             INPUT(',')
P             INPUT(NUMBER OR WORD)
P             SET(KLAST = *)
P             END
P          ELSE SET(KLAST=0)
P          END
P       ELSE DO
P          INPUT('TABLE')
P          SET(FUN_TYP = 'TABLE')
P          MESSAGE(6,'N:TABLE READ HAS NOT YET BEEN DESIGNED')
P          END
P       END  FUN_READ
```

Figure C-1
FCF FOR LINK4
(page 13 of 41)

```
P       L4_COMMON:   /* SETUP LINK4 COMMON BLOCKS */
P
P       IF FREQ = 'CURVE' OR DAMP = 'CURVE' OR DABB = 'YES' THEN
P             ENTER(PASS1,'COMCURVE,'COMMON /CURVES/ CURVE(1000), LOCATE(4)')
P       ELSE ENTER(PASS1,'COMCURVE,'')
P       SET(COML4INTR = 'COMMON /L4INTR/ NREM, REMARK(10), NRSECT, NSIE, JEND')
P       IF DAW = 'YES' THEN DO
P          SET(COML4INTR)
P          CATENATE(8,',IAOUT,IBOUT,IAIN,IBIN,LAPOVR,NUM(8LCIN)')
P          END
P       IF FREQ = 'CURVE' OR DAMP = 'CURVE' OR DABB = 'CURVE' THEN DO
P          SET (COML4INTR)
P          CATENATE(8,',KOIM,LCC')
P          END
P       ENTER(PASS1,'COML4DAT,'COMMON /L4DATA/ CVALF,CVCLT,CVDAW,IOBUF4,')
P       CATENATE(8,'IQ,ISM4,ITRALF,ITRCLT,ITRDAW,KDABB,KFALF,KLALF,KWNW,')
P       CATENATE(8,'DZV,MB,MF,NANC4,PCTDAW,SDIF')
P       END L4_COMMON
```

Figure C-1
FCF FOR LINK4
(page 14 of 41)

```
P       L4PARIO:    /* DEFINE LINK4 B FILE NAMES AND COMMON STORAGE */
P
P       INVOKE L4PARIN    /* DEFINE INPUT PARAMETERS */
P  /* BUILD INPUT COMMON BLOCK */
P       SET(NPAR = N_IN_PAR)
P       SET(COMNAME = 'COMIN')
P       SET(NAME = 'PNIN')
P       SET(UNITS = 'PUIN')
P       SET(COMIN = 'COMMON /L4IN/ TIME &IDIP')
P       INVOKE COMBUILD
P  /* BUILD INPUT DATA STATEMENTS */
P       SET(PNIN1 = 'DATA PN1/')
P       SET(PNIN2 = 'DATA PN2/')
P       SET(DATA1 = 'PNIN1')
P       SET(DATA2 = 'PNIN2')
P       INVOKE DATABUILD
P       INVOKE L4PAROUT    /* DEFINE OUTPUT PARAMETERS */
P  /* BUILD OUTPUT COMMON BLOCK */
P       SET(NPAR = N_OUT_PAR)
P       SET(COMNAME='COMOUT')
P       SET(NAME = 'PNOUT')
P       SET(UNITS = 'PUOUT')
P       SET(COMIN = 'COMMON /L4OUT/')
P       INVOKE COMBUILD
P  /* BUILD OUTPUT DATA STATEMENTS */
P       SET(PNOUT1 = 'DATA PN1/')
P       SET(PNOUT2 = 'DATA PN2/')
P       SET(DATA1 = 'PNOUT1')
P       SET(DATA2 = 'PNOUT2')
P       INVOKE DATABUILD
P       END L4PARIO
```

Figure C-1
FCF FOR LINK4
(page 15 of 41)

AD-A068 394 SCIENCE APPLICATIONS INC ENGLEWOOD CO F/G 9/2
FLIGHT TEST ORIENTED PRECOMPILER SYSTEM (FLTOPS DESIGN SPECIFIC--ETC(U)
AUG 78 D A OTEY, H R RAMSEY, J K WILLOUGHBY F04611-77-C-0040
UNCLASSIFIED SAI-78-061-DEN AFFTC-TR-78-22 NL

4 OF 4
AD A068394

END
DATE FILMED
6-79
DDC

```
P       L4PARIN:    /* DEFINE B FILE INPUTS */
P
P       DEFINE PNIN(27+N_IN_PAR),  PUIN(27+N_IN_PAR)
P       ENTER(PNIN(1), 'ALFAM')
P       ENTER(PNIN(2), 'APCT')
P       ENTER(PNIN(3), 'DAT')
P       ENTER(PNIN(4), 'D2T')
P       ENTER(PNIN(5), 'CGT')
P       ENTER(PNIN(6), 'FET')
P       ENTER(PNIN(7), 'FGT')
P       ENTER(PNIN(8), 'FRT')
P       ENTER(PNIN(9), 'P')
P       ENTER(PNIN(10), 'C')
P       ENTER(PNIN(11), 'QDCT')
P       ENTER(PNIN(12), 'QBART')
P       ENTER(PNIN(13), 'R')
P       ENTER(PNIN(14), 'THAT')
P       ENTER(PNIN(15), 'TH2T')
P       ENTER(PNIN(16), 'VTT')
P       ENTER(PNIN(17), 'WFT')
P       SET(N=17)
P       IF JAXIS=2 OR JAXIS=4 OR JAXIS=8 THEN DO
P          INCREMENT(N)
P          ENTER(PNIN(N),'XNXM')
P          END
P       ELSE IF JAXIS=1 OR JAXIS=6 OR JAXIS=7 THEN DO
P          INCREMENT(N)
P          ENTER(PNIN(N),'XNXMP')
P          END
P       IF JAXIS=2 OR JAXIS=3 OR JAXIS=7 OR ISOURC=2 THEN DO
P          INCREMENT(N)
P          ENTER(PNIN(N),'XNXM')
P          END
P       ELSE IF JAXIS=1 OR JAXIS=5 OR JAXIS=8 OR ISOURC=1 THEN DO
P          INCREMENT(N)
P          ENTER(PNIN(N),'XNZMP')
P          END
P       IF JAXIS=4 OR JAXIS=6 OR JAXIS=9 OR ISOURC>2 THEN DO
P          INCREMENT(N)
P          ENTER(PNIN(N),'XNZWT')
P          END
P       IF JAXIS=3 OR JAXIS=5 OR JAXIS=9 THEN DO
P          INCREMENT(N)
P          ENTER(PNIN(N),'XNXWT')
P          END
P       IF LQCOT='YES' THEN DO
P          INCREMENT(N)
P          ENTER(PNIN(N),'QDCT')
P          INCREMENT(N)
P          ENTER(PNIN(N),'Q')
P          ENTER(PUIN(N),' TWC')
P          END
P       IF DAW='YES' THEN DO
P          INCREMENT(N)
P          ENTER(PNIN(N),'ALFAP')
```

Figure C-1
FCF FOR LINK4
(page 16 of 41)

```
P             ENTER(PUIN(N),' TWO')
P             INCREMENT(N)
P             ENTER(PNIN(N),'ALFA')
P             ENTER(PUIN(N),' THREE')
P             END
P          IF N_IN_PAR > 0 THEN DO
P             SET(NADD = N_IN_PAR)
P             SET(NAME = 'PNIN')
P             SET(UNITS = 'PLIN')
P             SET(ADDN = 'IN_PARN')
P             SET(ADDU = 'IN_PARU')
P             INVOKE ADD_PAR
P             END
P          SET(N_IN_PAR = N+N_IN_PAR)
P          END L4PARIN
```

Figure C-1
FCF FOR LINK4
(page 17 of 41)

```
P       L4PARCUT:    /* DEFINE B FILE CUTPUT PARAMETERS */
P
P       DEFINE PNOUT(20+N_OUT_PAR),  PUOUT(20+N_CUT_PAR)
P       ENTER(PNOUT(1), 'ALFAT' )
P       ENTER(PNOUT(2), 'CLT'   )
P       ENTER(PNOUT(3), 'FEXT'  )
P       ENTER(PNOUT(4), 'FNDA'  )
P       ENTER(PNOUT(5), 'FNDT2' )
P       ENTER(PNOUT(6), 'FNT'   )
P       ENTER(PNOUT(7), 'HETDCT')
P       ENTER(PNOUT(8), 'SFCORA')
P       ENTER(PNOUT(9), 'SFCORT')
P       ENTER(PNOUT(10),'TSFC'  )
P       ENTER(PNOUT(11),'XNZWTP')
P       SET(N=11)
P       IF JAXIS <>4 AND JAXIS<>6 AND JAXIS<>9 THEN DO
P          INCREMENT(N)
P          ENTER(PNOUT(N),'XNZWT')
P          END
P       IF JAXIS <>3 AND JAXIS<>5 AND JAXIS<>9 THEN DO
P          INCREMENT(N)
P          ENTER(PNOUT(N),'XNXWT')
P          END
P       IF LCDT = 'YES' THEN DO
P          INCREMENT(N)
P          ENTER(PNOUT(N),'CDT')
P          END
P       IF DABB = 'YES' THEN DO
P          INCREMENT(N)
P          ENTER(PNOUT(N),'DABB')
P          END
P       IF DAC = 'YES' THEN DO
P          INCREMENT(N)
P          ENTER(PNOUT(N),'DAC')
P          END
P       IF DAU = 'YES' THEN DO
P          INCREMENT(N)
P          ENTER(PNOUT(N),'DAU')
P          END
P       IF DAL = 'YES' THEN DO
P          INCREMENT(N)
P          ENTER(PNOUT(N),'DAL')
P          END
P       IF ALLOKUP <> 'YES' THEN DO
P          INCREMENT(N)
P          ENTER(PNOUT(N),'DALFA')
P          END
P       IF N_CUT_PAR > 0 THEN DO
P          SET(NADD = N_OUT_PAR)
P          SET(NAME = 'PNOUT')
P          SET(UNITS = 'PLOUT')
P          SET(ADDN = 'OUT_PARN')
P          SET(ADDU = 'OUT_PARU')
P          INVOKE ADD_PAR
P          END
P       SET(N_OUT_PAR = N+N_OUT_PAR)
P       END L4PARCUT
```

Figure C-1
FCF FOR LINK4
(page 18 of 41)

```
P          ADD_PAR:     /* ADD PARAMETERS */
P
P               /* NAME = TABLE NAME CONTAINING STANDARD PARAMETER NAMES  */
P               /* UNITS = TABLE NAME CONTAINING STANDARD PARAMETER UNITS */
P               /* ADDN  = TABLE NAME CONTAINING ADDED PARAMETER NAMES    */
P               /* ADDU  = TABLE NAME CONTAINING ADDED PARAMETER UNITS    */
P               /* N = NUMBER OF STANDARD PARAMETERS                      */
P               /* NADD = NUMBER OF PARAMETERS TO BE ADDED                   */
P          SET(I = 1)
P          DO WHILE I<=NADD
P             SET(J = 1)
P             DO WHILE J<=N
P                      AND VALUE(VALUE(NAME)(I)) <>  VALUE(VALUE(ADDN)(J))
P                      AND VALUE(VALUE(UNITS)(I))<>  VALUE(VALUE(ADDU)(J))
P                INCREMENT(J)
P                END
P             IF J<=N THEN DO
P                INCREMENT(N)
P                ENTER(VALUE(NAME)(N), VALUE(VALUE(ADDN)(I)) )
P                ENTER(VALUE(UNITS)(N), VALUE(VALUE(ADDU)(I)) )
P                END
P             INCREMENT(I)
P             END
P          END ADD_PAR
```

Figure C-1
FCF FOR LINK4
(page 19 of 41)

```
P          COMBUILD:   /* BUILD COMMON BLOCK FOR 8 FILE PARAMETERS */
P
P                      /* NPAR = NUMBER OF PARAMETERS */
P                      /* COMNAME = NAME OF CONSTANT INTO WHICH COMMON STATEMENT    */
P                      /*              TO BE STORED                                 */
P                      /* NAME = NAME OF TABLE CONTAINING PARAMETER NAMES */
P                      /* UNITS = NAME OF TABLE CONTAINING PARAMETER UNITS */
P
P          CATENATE(DEFAULT,VALUE(COMNAME), VALUE(VALUE(NAME)(1)) )
P          IF VALUE(VALUE(UNITS)(1)) <> '' THEN
P             IF VALUE(VALUE(UNITS)(1)) = ' TWO' THEN
P                CATENATE(DEFAULT,VALUE(COMNAME),'2')
P             ELSE IF VALUE(UNITS)(1))  = ' THREE' THEN
P                CATENATE(DEFAULT,VALUE(COMNAME),'3')
P             ELSE MESSAGE('IS ILLEGAL UNITS')
P          CATENATE(DEFAULT,VALUE(COMNAME), '$ICIN')
P
P          SET(I = 2)
P          DO WHILE I<=NPAR
P             CATENATE(DEFAULT,VALUE(COMNAME), ',' , VALUE(VALUE(NAME)(I)) )
P             IF VALUE(VALUE(UNITS)(I)) <> '' THEN
P                IF VALUE(VALUE(UNITS)(I)) = ' TWO' THEN
P                   CATENATE(DEFAULT,VALUE(COMNAME), '2')
P                ELSE IF VALUE(VALUE(UNITS)(I)) = ' THREE' THEN
P                   CATENATE(DEFAULT,VALUE(COMNAME), '3')
P                ELSE MESSAGE(I,'$:ILLEGAL UNITS')
P             CATENATE(DEFAULT,VALUE(COMNAME), '$ICIN')
P             INCREMENT(I)
P             END
P          END COMBUILD
```

Figure C-1
F&F FOR LINK4
(page 20 of 41)

```
P       DATABUILD:   /* BUILD PARAMETER NAME DATA STATEMENTS */
P
P                    /* NPAR = NUMBER OF PARAMETERS */
P                    /* DATA1 = NAME OF CONSTANT INTO WHICH NAME DATA STMT STORED*/
P                    /* DATA2 = NAME OF CONSTANT INTO WHICH UNITS DATA STMT      */
P                    /*           STORED                                        */
P                    /* NAME = NAME OF TABLE CONTAINING PARAMETER NAMES */
P                    /* UNITS = NAME OF TABLE CONTAINING PARAMETER UNITS */
P       CATENATE(DEFAULT,VALUE(DATA1),  LENGTH(VALUE(NAME)(1)),'H',
P                                       VALUE(NAME)(1) )
P       IF LENGTH(VALUE(UNITS)(1)) > 0 THEN CATENATE(DEFAULT,VALUE(DATA2),
P            LENGTH(VALUE(UNITS)(1)),'H',VALUE(UNITS)(1))
P       ELSE CATENATE(DEFAULT,VALUE(DATA2), '1H ')
P       SET(NBLANK=0)
P       SET(I=2)
P       DO WHILE I <= NPAR
P          CATENATE(DEFAULT,VALUE(DATA1), ',' ,LENGTH(VALUE(NAME)(I)),'H',
P               VALUE(NAME)(I) )
P          IF VALUE(VALUE(UNITS)(I)) = ' ' THEN INCREMENT(NBLANK)
P          ELSE DO
P             IF NBLANK > 0 THEN DO
P                CATENATE(DEFAULT,VALUE(DATA2), ',',NBLANK,'*1H ')
P                SET(NBLANK=0)
P                END
P             CATENATE(DEFAULT,VALUE(DATA2), ',',LENGTH(VALUE(UNITS)(I)),
P                  'H',VALUE(UNITS)(I) )
P             END
P          INCREMENT(I)
P          END
P       IF NBLANK > 0 THEN
P          CATENATE(DEFAULT,VALUE(DATA2), ',' ,NBLANK,'*1H ')
P       CATENATE(DEFAULT,VALUE(DATA1), '/')
P       CATENATE(DEFAULT,VALUE(DATA2), '/')
P       END DATABUILD
P
P
```

Figure C-1
FCF FOR LINK4
(page 21 of 41)

```
P
P         LINK4B:         /* LINK4 WITH BUFFER (WITH DYNAMIC LAG) */
      OVERLAY(SFEXT,4,0)
      PROGRAM LINK4
C-------------------------------------------------------------------------
C-----PERFORMANCE TEST CONDITIONS PRIMARY OVERLAY
C----- THIS PROGRAM CONTROLS CALCULATION OF TEST PERFORMANCE PARAMETERS.
C----- ITS ONLY FUNCTION IS TO CALL INTO EXECUTION THE SECONDARY
C----- OVERLAYS INPUT4, DALAG AND SFEXT.
C*****NOTE - THE THREE "OVERLAY CALLS" IN THIS PROGRAM ARE NON-ANSI.
C-------------------------------------------------------------------------
      COMMON /ALL/LUIN,LUOUT,LUCURV,LTHRUS,S,XIF,CHERD
      #COMLADAT
      COMMON /LIDATA/ ZUD, NDFLAG
C-------------------------------------------------------------------------
P         OUTPUT(COMIN,8)
P         OUTPUT(COMOUT,8)
      #COMCURVE
P         OUTPUT(COML4INTR,8)
      COMMON  /SAVALF/ KURVIA
      COMMON  /SAVDAU/ KURVI,CVDAU(400)
      KURVIA = 0
      KURVI = 0
      NSC = 0
      WRITE (6,6000)
 6000 FORMAT (14HOENTERED LINK4)
   10 CALL OVERLAY(5HSFEXT,4,1)
      IF (NDFLAG .EQ .3)  GO TO 999
      CALL OVERLAY(5HSFEXT,4,2)
      CALL OVERLAY(5HSFEXT,4,3)
      IF (JEND .NE. 1)  GO TO 999
  999 WRITE(6,6999)
 6999 FORMAT(/1HOEXIT LINK4 )
      END
P         END  LINK4B
```

Figure C-1
FCF FOR LINK4
(page 22 of 41)

```
P         INPUT4B$
      OVERLAY (SFEXT,4,1)
      PROGRAM INPUT4
C-------------------------------------------------------------------
C-----LINK4 FILE INPUT SECONDARY OVERLAY
C----- THIS PROGRAM LOADS NEEDED CURVES FROM A CURVE FILE INTO CORE AND
C----- READS PARAMETERS FROM AN INPUT B FILE. THESE CURVES AND B FILE
C----- PARAMETER VALUES ARE THEN PASSED TO THE OTHER LINK4 SECONDARY
C----- OVERLAYS (DALAG,SFEXT) THROUGH COMMON BLOCKS FOR PROCESSING.
C-------------------------------------------------------------------
      COMMON /ALL/ LUIN,LUOUT,LLCURV,LTHRUS,S,XIF,CHCRD
      #COML4DAT
      COMMON /L1DATA/ ZUD, NOFLAG
C-------------------------------------------------------------------
P         OUTPUT(COMIN,#)
P         OUTPUT(COMOUT,#)
      #COMCURVE
P         OUTPUT(COML4INTR,#)
      DIMENSION PN1(#LDIN), PN2(#LDIN), VAL(#LDBLF,#LDIN), TN1(200), TN2(200),
     +          LISTNO(200), V(200)
      EQUIVALENCE (VAL(1),ALFAM(1))
P         OUTPUT(PNIN1,#)
P         OUTPUT(PNIN2,#)
      NVAR = #LDIN
      KDIM = 1000
      NSC = NSC+1
      WRITE (6,6000)
 6000 FORMAT (13HOENTERED INPUT4,//)
      IF (NSC.GT.1) GO TO 200
C-------------------------------------------------------------------
C-----PRINT INPUTS-----
C-------------------------------------------------------------------
    6 WRITE(6,6101) LUIN,LLCLT,LLCURV,LTHRUS,S,XIF,CHCRD
 6101 FORMAT(6X,32HPARAMETERS FROM COMMON BLOCK ALL,//,
     110X,7HLUIN  =,I5    ,//,10X,7HLUOUT =,I5    ,//,10X,7HLLCURV=,I5    ,//,
     210X,7HLTHRUS=,L5    ,//,10X,7HS     =,E14.6,//,10X,7HXIF   =,E14.6,//,
     310X,7HCHCRD =,E14.6,/)
      WRITE (6,6102) CVALF,CVCLT,CVDAW,IDBUG4,IC,TSP4,ITRALF,ITRCLT,
     1KDARB,KFALF,KFDAU,KLALF,KLDAU,KWNV,KZV,MP,MF,NANC4,FCTDAW,SDIF
     2KDARB,KFALF,KFDAU,KLALF,KLDAU,KWNV,KZV,
 6102 FORMAT(6X,38HPARAMETERS FROM COMMON BLOCK L4DAT#,//,
     110X,7HCVALF =,E14.6,//,10X,7HCVCLT =,E14.6,//,10X,7HCVDAW =,E14.6,//,
     210X,7HIDBUG4=,I5    ,//,10X,7HIC    =,I5    ,//,10X,7HISP   =,I5    ,//,
     310X,7HITRALF=,I5    ,//,10X,7HITRCLT=,I5    ,//,10X,7HITRDAW=,I5    ,//,
     4                        10X,//,10X,7HKDARB =,I5    ,//,
     *10X,7HKFALF =,I5    ,//,10X,7HKFDAU =,I5    ,//,10X,7HKLALF =,I5    ,//,
     610X,7HKLDAU =,I5    ,//,10X,7HKWNV  =,I5    ,//,10X,7HKZV   =,I5    ,//,
     710X,7HMP    =,I5    ,//,10X,7HMF    =,I5    ,//,
     *10X,7HNANC4 =,I5    ,//,10X,7HNREM4 =,I5    ,//,10X,7HFCTDAW=,E14.6,//,
     910X,7HSDIF  =,E14.6)
      REWIND LUIN
      REWIND LLCLT
      IOC = 1
      IAPOVR = (MF + MP + 1)/2
P         IF FREQ = 'CURVE' OR DAMP = 'CURVE' OR DABB = 'YES' THEN
P           DO
C-------------------------------------------------------------------
```

Figure C-1
FCF FOR LINK4
(page 23 of 41)

```
C-----LOAD CURVES INTO THE ARRAY CURVE-----
P             IF FREQ = 'CURVE' THEN
C-----KWNV  ..ACCNF = F(MACH,CTBAR)...USED IN DALAG...BEGINS AT LOCATE(1)
P             IF DAMP = 'CURVE' THEN
C-----KZV   ..ACCDR = F(MACH,CTBAR)...USED IN DALAG...BEGINS AT LOCATE(2)
P             IF DABB = 'CURVE' THEN
C-----KDABP.. DABP = F(FB)           ...USED IN SDABB...BEGINS AT LOCATE(3)
C-------------------------------------------------------------------------
      LCC = 1
P             DEFINE TABLE(3), KURV(3)
P             ENTER (TABLE(1),FREQ)
P             ENTER (KURV(1),KFREQ)
P             ENTER (TABLE(2),DAMP)
P             ENTER (KURV(2),KDAMP)
P             ENTER (TABLE(3),DABB)
P             ENTER (KURV(3),KDABB)
P             SET (I = 1)
P             DO WHILE I <= 3
P                IF VALUE(TABLE(I)) = 'CURVE' THEN
P                   DO
P                   ENTER (PASS1,N,I)
P                   ENTER (PASS1,K,VALUE(KURV(I)))
      LOCATE(N) = LCC
      CALL CREAD(K,LLCURV,CURVE(LCC),IF1,IDBLG4)
      IGC = IGC*IF1
      ND = CURVE(LCC+1)
      NX = CURVE(LCC+2)
      NZ = CURVE(LCC+3)
      LCC = LCC + NX + NZ * (NX + ND - 2) + 7
P                   END
P                INCREMENT(I)
P                END
P             DELETE (PASS1,N)
P             DELETE (PASS1,K)
      IF (LCC .LT. KDIM)  GO TO 100
      IGC = 0
      WRITE(6,6090) LCC
 6090 FORMAT('0H ERROR IN INPUT4.  SIZE OF ARRAY CURVE MUST BE INCREASE,
     +       14HD TO AT LEAST ,I5,1H.)
  100 CONTINUE
P             CLEAR_TABLE (TABLE)
P             CLEAR_TABLE (KURV)
P             END
C-------------------------------------------------------------------------
C-----READ INPUT R FILE LABEL RECORD-----
C-------------------------------------------------------------------------
      READ (LUIN) L,NRSECT,NR,(L,I=1,NR), NLAB,(TN1(I),TN2(I),LISTNO(I),
     +            I=1,NLAB)
      CALL LABSOR (PN1,PN2,TN1,TN2,LISTNO,NUM,NVAR+1,NLAB)
C-------------------------------------------------------------------------
C-----DETERMINE IF NEEDED PARAMETERS ARE AVAILABLE-----
C-------------------------------------------------------------------------
      DO 150 I=1,NVAR
      IF (NUM(I).GT.0) GO TO 150
C----NEEDED PARAMETER NOT AVAILABLE-----
  140 WRITE(6,6140) PN1(I), PN2(I)
```

Figure C-1
FCF FOR LINK4
(page 24 of 41)

```
 614C FORMAT(28H ERROR IN LINK4.  PARAMETER ,2A6,18H NOT AVAILABLE ON ,
     +       19HINPUT B FILE.)
       IGO = 0
   15C CONTINUE
C-------------------------------------------------------------------------
C-----IF ANY ERRORS WERE ENCOUNTERED, SET NDFLAG=9 AND RETURN-----
C-------------------------------------------------------------------------
       IF (IGO.EQ.0) NDFLAG=9
       IF (IGO.EQ.0) GO TO 270
       IAIN = 1
       IAOUT = 1
       IBIN = 60
       GO TO 220
C-------------------------------------------------------------------------
C-----OVERLAP INPUT ARRAYS-----
C-------------------------------------------------------------------------
   20C CONTINUE
       IF (LAPOVR.EQ.0) GO TO 220
       IA = 2*LAPOVR
       IAIN = IA+1
       IAOUT = LAPOVR+1
       IP = 60-IA
       DO 210 I=1,IA
       IP = IP+1
       TIME(I) = TIME(IP)
       DO 210 J=1,NVAR
       VAL(I,J) = VAL(IP,J)
   21C CONTINUE
C-------------------------------------------------------------------------
C-----READ VALUES FOR PARAMETERS-----
C-------------------------------------------------------------------------
   220 IBOUT = 60-LAPOVR
       NTOUT = 0
       I = IAIN-1
   225 I = I+1
       READ (LUIN) N,T,(V(J),J=1,N)
       IF (N.EQ.0 .AND. T.EQ.0.0) GO TO 240
       IF (I.GE.61) GO TO 235
       NTOUT = NTOUT+1
       TIME(I) = T
       DO 230 J=1,NVAR
       IF (NUM(J).EQ.0) GO TO 230
       JA = NUM(J)
       VAL(I,J) = V(JA)
   230 CONTINUE
       GO TO 225
   235 BACKSPACE LUIN
       GO TO 270
   240 JEND = 1
       IBOUT = NTOUT+IAIN-1
       IBIN = IBOUT
   270 CONTINUE
       END
P          END  INPUT48
```

Figure C-1
FCF FOR LINK4
(page 25 of 41)

```
P         LINK4A:
      OVERLAY(SFEXT,4,0)
      PROGRAM LINK4
C-----------------------------------------------------------------------
C-----PERFORMANCE TEST CONDITIONS PRIMARY OVERLAY
C----- THIS PROGRAM CONTROLS CALCULATION OF TEST PERFORMANCE PARAMETERS.
C----- ITS ONLY FUNCTION IS TO CALL INTO EXECUTION THE SUBROUTINES
C----- INPUT4 AND SFEXT.
C-----------------------------------------------------------------------
      COMMON /ALL/LUIN,LLOUT,LLCURV,LTHRLS,S,XIF,CHCRD
     #COML4DAT
      COMMON /LIDATA/ ZUD, NDFLAG
C-----------------------------------------------------------------------
P         OUTPUT(COML4INTR,8)
      NSG = 0
      WRITE (6,6000)
 6000 FORMAT (14HOENTERED LINK4)
   10 CALL INPUT4
      IF (JEND .EQ. 1)  GO TO 999
      CALL SFEXT
      GO TO 10
  999 WRITE(6,6999)
 6999 FORMAT(11HOEXIT LINK4 )
      END
P         END LINK4A
```

Figure C-1
FCF FOR LINK4
(page 26 of 41)

```
P         INPUT4:
P         /*  WITHOUT DYNAMIC_LAG  */
P         /*  (IF NO DATA BUFFER)  */
      SUBROUTINE INPUT4
C-----------------------------------------------------------------------
C-----LINK4 FILE INPUT SECONDARY SUBROUTINE
C-----  THIS PROGRAM LOADS NEEDED CURVES FROM A CURVE FILE INTO CORE AND
C-----  READS PARAMETERS FROM AN INPUT B FILE.  THESE CURVES AND B FILE
C-----  PARAMETER VALUES ARE THEN PASSED TO THE OTHER LINK4 SUBROUTINE
C-----  THROUGH COMMON BLOCKS FOR PROCESSING.
C-----------------------------------------------------------------------
      COMMON /ALL/ LUIN,LUOUT,LUCURV,LTHRUS,S,XIF,CHORD
      #COMLADAT
      COMMON /LIDATA/ ZUD, NOFLAG
C-----------------------------------------------------------------------
P         OUTPUT(COMIN,#)
P         OUTPUT(COMOUT,#)
      #COMCURVE
      DIMENSION PN1(#LDIN), PN2(#LDIN), VAL(#LDIN), TN1(200), TN2(200),
     +          LISTNO(200), V(200)
      EQUIVALENCE (VAL(1),ALFAM)
P         OUTPUT(PNIN1,#)
P         OUTPUT(PNIN2,#)
      NVAR = #LDIN
      KDIM = 1000
      NSG = NSG + 1
      WRITE (6,6000)
 6000 FORMAT (1*0ENTERED INPUT4,//)
      IF (NSG .GT. 1)  GO TO 200
C-----------------------------------------------------------------------
C-----PRINT INPUTS-----
C-----------------------------------------------------------------------
    6 WRITE(6,6101) LUIN,LUOUT,LUCURV,LTHRUS,S,XIF,CHORD
 6101 FORMAT(6X,32HPARAMETERS FROM COMMON BLOCK ALL,/,
     110X,7HLUIN  =,I5    ,/,10X,7HLUOUT =,I5    ,/,10X,7HLUCURV=,I5    ,/,
     210X,7HLTHRUS=,I5    ,/,10X,7HS     =,E14.6,/,10X,7HXIF   =,E14.6,/,
     310X,7HCHORD =,E14.6,/)
      WRITE (6,6102) CVALF,CVOLT,CVDAW,IDBUG4,IC,ISM4,ITRALF,ITRCLT,
     1ITRDAW,KDABR,KFALF,KFDAU,KLALF,KLDAU,KWNV,KZV,MB,MF,NANC4,
     2PCTDAW,SDIF
 6102 FORMAT(6X,35HPARAMETERS FROM COMMON BLOCK L4DATA,/,
     110X,7HCVALF =,E14.6,/,10X,7HCVOLT =,E14.6,/,10X,7HCVDAW =,E14.6,/,
     2E14.6,/,10X,7HIDBUG4=,I5    ,/,10X,7HIC    =,I5    ,/,
     310X,7HISM   =,I5    ,/,10X,
     410X,7HITRALF=,I5    ,/,10X,7HITRCLT=,I5    ,/,10X,7HITRDAW=,I5    ,/,
     5          10X,/,10X,7HKDABR =,I5    ,/,
     610X,7HKFALF =,I5    ,/,10X,7HKFDAU =,I5    ,/,10X,7HKLALF =,I5    ,/,
     710X,7HKLDAU =,I5    ,/,10X,7HKWNV  =,I5    ,/,10X,7HKZV   =,I5    ,/,
     8/,10X,7HMB    =,I5    ,/,10X,7HMF    =,I5    ,/,
     910X,7HNANC4 =,I5    ,/,10X,7HNREM4 =,I5    ,/,10X,7HPCTDAW=,E14.6,/,
     *10X,7HSDIF  =,E14.6)
      REWIND LUIN
      REWIND LUOUT
P         IF DABB = 'YES' THEN
P           DO
```

Figure C-1
FCF FOR LINK4
(page 27 of 41)

```
C-----------------------------------------------------------------------
C-----LOAD CURVES INTO THE ARRAY CURVE-----
C-----KDARR.. DARR = F(FB)          ...USED IN SDARR...BEGINS AT LOCATE(3)
C-----------------------------------------------------------------------
      IGC = 1
      LOCATE(3) = 1
P           OUTPUT('CALL CREAD(',KDARR,',LUCURV,CURVE(1),IF1,IDBLG4)',1)
      IGC = IGC * IF1
      ND = CURVE(2)
      NX = CURVE(3)
      NZ = CURVE(4)
      LOC = NX + NZ * (NX + ND - 2) + 8
      IF (LOC .LT. KDIM)  GO TO 100
      IGC = 0
      WRITE(6,6090) LOC
 6090 FORMAT(39H ERROR IN INPUT4.  SIZE OF ARRAY CURVE MUST BE INCREASE,
     +       14HD TO AT LEAST ,I5,1H.)
  100 CONTINUE
P           END
C-----------------------------------------------------------------------
C-----READ INPUT B FILE LABEL RECORD-----
C-----------------------------------------------------------------------
      READ (LUIN) L,NRSECT,NREM,(REMARK(I),I=1,NREM), NLAB,(TN1(I),
     +            TN2(I),LISTNO(I),I=1,NLAB)
      CALL LABSOR (PN1,PN2,TN1,TN2,LISTNO,NUM,NVAR+1,NLAB)
C-----------------------------------------------------------------------
C-----DETERMINE IF NEEDED PARAMETERS ARE AVAILABLE-----
C-----------------------------------------------------------------------
      DO 150 I=1,NVAR
      IF (NUM(I).GT.0) GO TO 150
C----NEEDED PARAMETER NOT AVAILABLE-----
  140 WRITE(6,6140) PN1(I), PN2(I)
 6140 FORMAT(28H ERROR IN LINK4.  PARAMETER ,2A6,18H NOT AVAILABLE ON ,
     +       13HINPUT B FILE.)
      IGC = 0
C-----------------------------------------------------------------------
C-----IF ANY ERRORS WERE ENCOUNTERED, SET NDFLAG=3 AND RETURN-----
C-----------------------------------------------------------------------
  150 IF (IGC .EQ. 0)  NDFLAG = 3
      IF (IGC .EQ. 0)  GO TO 260
C-----------------------------------------------------------------------
C-----READ VALUES FOR PARAMETERS-----
C-----------------------------------------------------------------------
  200 READ (LUIN) N,T,(V(J),J=1,N)
      IF (N .NE. 0)  GO TO 270
      IZ = 0
      Z = 0
      WRITE(LUOUT) IZ,Z,Z
  260 JEND = 1
  270 RETURN
      END
P         END  INPUT4
```

Figure C-1
FCF FOR LINK4
(page 28 of 41)

```
P          SFEXT1
P          IF DAB = 'YES' THEN
P            DO
      OVERLAY (SFEXT,4,3)
      PROGRAM SFEXT
P            END
P          ELSE
      SUBROUTINE SFEXT
C-------------------------------------------------------------------------
C-----TEST EXCESS THRUST SECONDARY OVERLAY
C-----  THIS SECONDARY OVERLAY COMPUTES THE ROOM PENDING, UPWASH AND
C-----  PITCH RATE CORRECTIONS TO THE ANGLE OF ATTACK (DABB,DAL,DAC),
C-----  TRUE ANGLE OF ATTACK (ALFAT), LOAD FACTORS (XNX,XNZ), LIFT
C-----  COEFFICIENTS (CLT,CDT), RATE OF CHANGE OF ENERGY HEIGHT (HETDOT),
C-----  AND EXCESS THRUST (FEXT) FOR TEST CONDITIONS.  IT THEN WRITES THEM
C-----  ON THE OUTPUT B FILE.
C-------------------------------------------------------------------------
      COMMON /ALL/ LLIN,LLOUT,LLCURV,LTHRUS,S,XIF,CHORD
      #COML4DAT
C-------------------------------------------------------------------------
P          OUTPUT(COMIN,#)
P          OUTPUT(COMOUT,#)
      #COMCURVE
P          OUTPUT(COML4INTR,#)
P          IF DAB = 'YES' THEN
P            DO
      COMMON /SAVAIR/ KURV1
      DIMENSION XLV(#LOBUF)
      EQUIVALENCE (XLV(1),HETDOT(1))
      DIMENSION V(#LOBUF,#LOCUT),PN1(#LOCLT),PN2(#LOCLT)
      EQUIVALENCE (V(1),XN2WT(1))
P            END
P          ELSE
      DIMENSION  V(#LLOUT), PN1(#LOOUT), PN2(#LOCLT)
P          OUTPUT(PNOUT1,#)
P          OUTPUT(PNOUT2,#)
      WRITE (6,6000)
 6000 FORMAT(14HOENTERED SFEXT)
      I2SFEX = IDBUG4
      I2SNXN = IDBUG4
      IF (I2SFEX.EQ.1) WRITE(6,6005)
 6005 FORMAT(17H0           TIME    ,/)
      NDIM = KDIM-LOC
P          IF DABB = 'YES' THEN
P            DO
      LOCDAB = LOCATE(3)
      NXDAB = CURVE(LOCDAB+2)
      NZDAB = CURVE(LOCDAB+3)
P            END
P          IF DAB = 'YES' THEN
      DO 200 I=IAOUT,IBOUT
      IF(I2SFEX.EQ.1) WRITE(6,6010)TIME#IBUF
 6010 FORMAT(1H ,7X,F10.3)
C-------------------------------------------------------------------------
C-----COMPUTE INITIAL CLT AND XLV----
```

Figure C-1
FCF FOR LINK4
(page 29 of 41)

```
C-------------------------------------------------------------------
P         IF ISOURC = 'FPA' THEN
      XNZWP#IBUF = XNZPP#IBUF
P         ELSE IF ISOURC = 'CG_ACCELEROMETER' THEN
      XNZWP#IBUF = XNZP#IBUF
P         ELSE IF ISOURC = 'RADAR' OR ISOURC = 'AIRSPEED/ALTITUDE' THEN
      XNZWP#IBUF = XNZWT#IBUF
      CLTT = XNZWP#IBUF *WT#IBUF / (CPART#IBUF *9)
      XLV#IBUF = XLVC+CGT#IBUF *CHORD/100.
      IF (I2SFEX.EC.1) WRITE(6,6090)XLV#IBUF
 6090 FORMAT(1H ,10X,26HOUTPUT BY SFEXT      XLV  =,F10.5)
P         IF ALOOKUP <> 'NO' OR DAU = 'CL,MACH,CG' OR DABB = 'YES' THEN
P            DO
P            IF  ALOOKUP <> 'NO' THEN
      ALFAT#IBUF = ALFAM#IBUF
C-------------------------------------------------------------------
C-----BEGIN ITERATIVE LOOP FOR CLT CONVERGENCE-----
C-------------------------------------------------------------------
      CLTP = -99999999.
      ITR = 0
   99 ITR = ITR+1
P            IF DABB <>'NO' THEN
P               DO
C-------------------------------------------------------------------
C-----CALL SDABB FOR ALFA CORRECTION DUE TO BODY BENDING (DABB)-----
C-------------------------------------------------------------------
      IF (KDABB.NE.0) CALL SDABB
     +      (DABB#IBUF, ALFAM#IBUF,DAW#IBUF,CPART#IBUF,TIME#IBUF,XNZWP,I
     +      BPLF,I,ABCOM, CLAB, WBCOM, CURVE, KDABB, LCCDAB, NXLAB, NZCA
     +      B,IDBUG4)
P               END
P            END
P            IF ALOOKUP = 'NO' THEN DO
P               OUTPUT('ALFAT#IBUF = ALFAM#IBUF')
P               IF DAW = 'YES' THEN OUTPUT(' + DAW#IBUF')
P               IF DABB <> 'NO' THEN OUTPUT(' + DABB#IBUF')
P               OUTPUT (#)
P               IF DAU <> 'NO' OR DAQ = 'YES' THEN
P                  DO
C-------------------------------------------------------------------
C-----CALL DALPB FOR ALFA CORRECTION DUE TO UPWASH-----
C-------------------------------------------------------------------
      CALL DALPB (ALFAT#IBUF, DALFA#IBUF, DAC#IBUF, DAL#IBUF, APCT#IBUF,
     +            ALFAM#IBUF, CGT#IBUF, CLTI, DABB#IBUF, DAW#IBUF,C#IBUF
     +            , TIME#IBUF, VTT#IBUF, XLV#IBUF, I, KFDAU, KLDAU, LDAC
     +            , ITRALF,CVALF,LCVALF, LLCURV,IDBUG4)
P                  END
P               END
C-------------------------------------------------------------------
C-----CALL SNXNZ FOR XNZWT AND XNZWP -----
C-------------------------------------------------------------------
      INDEX = 2
      CALL SNXNZ (ACCMA,ACCMAP,ALFAT,CGT,DALFA,DABB,P,C,R,CCCT,CR,
     +            XLVC, CHORD, XNXP, XNXPP, XNZP, XNZPP, XNXWT, XNZWT,
     +            INDEX, JAXIS, I, XNZWP, I2SNXN )
```

Figure C-1
FCF FOR LINK4
(page 30 of 41)

```
C-----------------------------------------------------------------------
C-----IF CLT IS TO BE CORRECTED FOR, COMPUTE CLT-----
C-----------------------------------------------------------------------
P         IF NODE_EXISTS(LINK4.CONSTANTS.RECOMPUTE_CL) THEN
P            DO
P            OUTPUT('CLTI = (XNZWT#IBUF * WT#IBUF'
P            IF VALUE(UFTAS,THRUST) = 'YES' THEN
P               OUTPUT(' - FGT#IBUF * SIN(ALFAT#IBUF + XIF)'
P            OUTPUT(')/(QBART#IBUF * S)',#)
P            END
P         IF ALCOKUP <> 'NO' THEN
P            DO
C-----------------------------------------------------------------------
C-----FOUR DIMENSIONAL CURVE LOOKUP -- ALFAT = F(AMCT,CLT,CGT)-----
C-----------------------------------------------------------------------
      CALL FOURD (ALFAT#IBUF,CLTI,AMCT#IBUF,CGT#IBUF,LLCURV,CURVE(LOC),
     +            NDIM,KFALF,KLALF,TIME#IBUF,SMSFEXT,I,KURV1)
P            END
P         IF ALCOKUP <> 'NO' OR DAU = 'CL,MACH,CG' OR DABB = 'YES' THEN
P            DO
C-----------------------------------------------------------------------
C-----CHECK FOR CLT CONVERGENCE-----
C-----------------------------------------------------------------------
      IF (ABS(CLTI-CLTP).LT.CVCLT) GO TO 160
      CLTP = CLTI
      IF (ITP.LT.ITRCLT) GO TO 55
C-----------------------------------------------------------------------
C-----CLT ITERATION LIMIT HAS BEEN EXCEEDED WITHCLT CONVERGENCE-----
C-----------------------------------------------------------------------
      CV = ABS(CLTI-CLTP)
      WRITE(6,6100) I,CVCLT,ITRCLT,CV
 6100 FORMAT(22H0ERROR IN SFEXT.  CLT(,I2,22H) DID NOT CONVERGE TO ,
     +       F8.5,3H IN,I5,33H ITERATIONS.  IT DID CONVERGE TO ,
     +       F8.5,1H.)
P            IF CVCLT = 'TERMINATE' THEN
P               DO
      WRITE(6,6105)
 6105 FORMAT(19X,21HEXECUTION TERMINATED.)
      STOP
P               END
P            ELSE DO
      WRITE(6,6110)
 6110 FORMAT(19X,24HCLT WILL BE RECALCULATED)
      ALFAT#IBUF = ALFAM#IBUF
P            END
P            IF ISOURC = 'FPA' THEN
      XNZ = XNZMP#IBUF
P            ELSE IF ISOURC = 'CG_ACCELEROMETER' THEN
      XNZ = XNZM#IBUF
P            ELSE IF ISOURC = 'RADAR' OR ISOURC = 'AIRSPEED/ALTITUDE' THEN
      XNZ = XNZWT#IBUF
      CLT = XNZ *WT#IBUF / (QBART#IBUF * S)
P         END
C-----------------------------------------------------------------------
```

Figure C-1
FCF FOR LINK4
(page 31 of 41)

```
C-----CALL SNXNZ FOR XNXWT AND COMPUTE HETDOT AND FEXT-----------------
C-----------------------------------------------------------------------
  160 INDEX = 1
      CALL SNXNZ (ACCMA,ACCMAP,ALFAT,CGT,DALFA,DABB,F,G,R,CDC1,GR,
     +            XLVC, CHORD, XNXP, XNXMP, XNZP, XNZPP, XNXWT, XNZWT,
     +            INDEX, JAXIS, I, XNZWP, I2SNXN )
      HETDOT#IBUF = XNXWT#IBUF*VTT#IBUF
      FEXT#IBUF = XNXWT#IBUF*WT#IBUF
      CLT#IBUF = CLTI
C-----------------------------------------------------------------------
C----COMPUTE CDT AND FNT -----
C-----------------------------------------------------------------------
P         OUTPUT('FNT#IBUF = FGT#IBUF')
P         IF VALUE(UFTAS,THRUST) = 'YES' THEN
P            OUTPUT (' * COS(ALFAT#IBUF + XIF)')
P         OUTPUT(' - FET#IBUF',#)
P         IF CD = 'COMPUTE' THEN
      CDT#IBUF = (FNT#IBUF - FEXT#IBUF) / (QBART#IBUF * S)
C-----------------------------------------------------------------------
C----COMPUTE FNDA, FNDT2, TSFC, SFCDRA, SFCDRT, AND TSFC---------------
C-----------------------------------------------------------------------
      FNDA#IBUF  = FNT#IBUF /DAT#IBUF
      FNDT2#IBUF = FNT#IBUF /D2T#IBUF
      SFCDRA#IBUF = WFT#IBUF /(THAT#IBUF**ETHETA * FNT#IBUF)
      SFCDRT#IBUF = WFT#IBUF /(TH2T#IBUF**ETHETA * FNT#IBUF)
      TSFC#IBUF = WFT#IBUF /(FGT#IBUF - FRT#IBUF)
      IF (T2SFEX.EQ.1) WRITE(6,6190) CDT#IBUF,ALFAT#IBUF,FEXT#IBUF
 6190 FORMAT(19X,19HOUTPUT BY SFEXT,5X,4HCDT=,F10.5,6X,6HALFAT=,F10.5,
     +       5X,5HFEXT=,F10.2)
  200 CONTINUE
      NPASS = 0
      IF (NSG.EQ.1) NPASS = 1
P         IF DAW = 'NO' THEN DO
P            ENTER(PASS1,IAOUT,'1')
P            ENTER(PASS1,IBOUT,'1')
P            ENTER(PASS1,ND,'1')
P            END
P         ELSE ENTER(PASS1,ND,VALUE(PASS1,LDBUFF))
      CALL WTBCAS(LUOUT,0,V,TIME,ND,IAOUT,IBOUT,#LOOUT,PN1,PN2,NPASS,JEND,
     +            NRSECT,NREM,REMARK,I2SFEX)
      IF (T2SFEX.EQ.1) WRITE(6,6200)
 6200 FORMAT(37HORETURN TO CALLING ROUTINE FROM SFEXT)
P         IF DAW <> 'YES' THEN
      RETURN
P         ELSE DO
P            DELETE(PASS1,IAOUT)
P            DELETE(PASS1,IBOUT)
P            END
P         DELETE(PASS1,ND)
      END
P         END  SFEXT
```

Figure C-1
FCF FOR LINK4
(page 32 of 41)

```
P         DAUPR:
      SUBROUTINE DAUPR (ALFAT, DALFA, DAC, DAL,
     +       AMCT, ALFAM, CGT, CLT, DABB, DAW, C, TIME, VTT, XLV, II,
     +       KFDAL, KLDAU, LDAC, ITRALF, CVALF, LCVALF, LLC, I2CALF)
C-----------------------------------------------------------------------
C-----UPWASH AND PITCH RATE CORRECTIONS SUBROUTINE
C----- THIS SUBROUTINE CALCULATES THE UPWASH AND PITCH RATE CORRECTIONS
C----- TO THE ANGLE OF ATTACK AND A TRUE WING ANGLE OF ATTACK.
C-----------------------------------------------------------------------
      COMMON /SAVDAU/ KURV1, CVDAL(400)
      LOGICAL LDAC,LCVALF
      ITR = 1
      NDIM = 400
P         IF DAU = 'ALFA' OR DAU = 'ALFA,MACH' THEN
P            DO
C-----------------------------------------------------------------------
C-----DAL = F(ALFA) ---- OR --- DAL = F(ALFA, AMCT) -------------------
C-----------------------------------------------------------------------
   10 CALL FOURD(DAU,ALFAT,AMCT,DUMY,LLC,CVDAL,NDIM,KFDAL,KFDAL,TIME,
     +             SHDAUPR, II, KURV1)
P            END
P         ELSE
P         IF DAU = 'CL,MACH,CG' THEN
P            DO
C-----------------------------------------------------------------------
C-----DAU = F(CLT,AMC,CGT)-----
C-----------------------------------------------------------------------
   10 CALL FOURD (DAU,CLT,AMCT,CGT,LLC,CVDAU,NDIM,KFDAL,KLDAL,TIME,
     +             SHDAUPR, II, KURV1)
P            END
P         ELSE IF DAU <> 'NO' THEN MESSAGE(I,'S:UPWASH LOOKUP INVALID')
P         IF DAC = 'YES' THEN
P            DO
C-----------------------------------------------------------------------
C-----COMPUTE DAC-----
C-----------------------------------------------------------------------
      DAC = ATAN( XLV * Q * COS(ALFAT) /(VTT - XLV * SIN(ALFAT) * C))
P            END
C-----------------------------------------------------------------------
C-----COMPUTE DALFA AND ALFAT-----
C-----------------------------------------------------------------------
P         OUTPUT('DALFA = ')
P         IF DAL <> 'NO' THEN OUTPUT('DAL')
P         IF DABB <> 'NO' THEN OUTPUT(' + DABB')
P         IF DAW <> 'NO' THEN OUTPUT(' + DAW')
P         IF DAC <> 'NO' THEN OUTPUT(' + DAC')
P         OUTPUT(0)
      ALFAT = ALFAP + DALFA
C-----------------------------------------------------------------------
C-----CHECK FOR CONVERGENCE OF ALFA-----
C-----------------------------------------------------------------------
      IF (ITR .EQ. 1)  GO TO 6C
      IF (ABS(ALFAT - ALFAP) .LE. CVALF)  GO TO 9C
      IF (ITR .GE. ITRALF)  GO TO 7C
C-----------------------------------------------------------------------
```

Figure C-1
FCF FOR LINK4
(page 33 of 41)

```
C-----ALFA HAS NOT CONVERGED AND ITERATION LIMIT HAS NOT BEEN EXCEEDED--
C-------------------------------------------------------------------
   60 ALFAP = ALFAT
      ITR = ITR + 1
      GO TO 10
C-------------------------------------------------------------------
C-----ALFA HAS NOT CONVERGED AND ITERATION LIMIT HAS BEEN EXCEEDED-----
C-------------------------------------------------------------------
   70 WRITE (6,6070) II, CVALF, ITRALF
 6070 FORMAT(23HOERROR IN DALPR.  ALFA(,I2,22H) DID NOT CONVERGE TO ,
     +       F8.5,3H IN,I5,12H ITERATIONS.)
P         IF LCVALF = 'USE_LAST_VALUE' THEN
P            DO
      DITR = ALFAT - ALFAP
      WRITE (6,6075) II, ALFAT, DITR
 6075 FORMAT(19X,5HALFA(,I2,4H) = ,F8.5,20H WHICH CONVERGED TO ,F8.5,
     +       14H WILL BE USED.)
P            END
P         ELSE IF LCVALF = 'RECOMPUTE' THEN DO
P            IF DAW <> 'NO' AND DABB <> 'NO' THEN
P               DO
P               OUTPUT('DALFA',' = ')
P               IF DABB <> 'NO' THEN OUTPUT('DABB')
P               IF DAW <> 'NO' THEN OUTPUT (' + DAW')
P               OUTPUT(#)
P               END
P            ELSE
      DALFA = 0.
      ALFAT = ALFAP + DALFA
      WRITE(6,6080)
 6080 FORMAT(19X,36HALFAT = ALFAP+DAW+DABB WILL BE USED.)
P            END
P         ELSE MESSAGE(I,'S:DALPR ALFA NO-CONVERGENCE FLAG INVALID')
C-------------------------------------------------------------------
C-----PRINT SECONDARY OUTPUT AND RETURN-----
C-------------------------------------------------------------------
   90 IF (I2DALP.EQ.1) WRITE(6,6090) DAL,DAO,ITR
 6090 FORMAT(1H ,18X,26HOUTPUT BY DALPR      DAL  =,F10.5,3X,5HDAO =,
     +       F10.5,5X,18HALFAT CONVERGED IN,I4,12H ITERATIONS.)
      RETURN
      END
P         END  DALPR
```

Figure C-1
FCF FOR LINK4
(page 34 of 41)

```
P       SDABB:
      SUBROUTINE SDABB (DABB, ALFAM, DAW, QBART, TIME, XNZWP, II,
     +       ABCCM, CLAB, WBCCM, CURVE, KDABB, LCCCAB, NXDAB, NZDAB,
     +       I2SDAB)
C-----------------------------------------------------------------------
C-----BODY BENDING CORRECTION SUBROUTINE
C----- THIS SUBROUTINE COMPUTES THE AERODYNAMIC AND INERTIAL LOADS IN
C----- THE BODY AND A CORRECTION TO THE ANGLE OF ATTACK DUE TO THOSE
C----- LOADS.
C-----------------------------------------------------------------------
      DIMENSION CURVE(8)
      ALFAV = ALFAM + DAW
      FB = QBART * ABCCM * CLAB * ALFAV + XNZWP * WBCCM
C-----------------------------------------------------------------------
C-----TABLE LOOKUP .. DABB = F(FB)-----
C-----------------------------------------------------------------------
      CALL TAPENT (FB,DABB,0.,NXDAB,NZDAB,CURVE(LCCCAB+7),INDIC ,
     +             TIME,5HSDABB,1,II,KDABB)
      IF (I2SDAB .EQ. 1) WRITE(6,6000) DABB,FB
 6000 FORMAT(1H ,1PX,26HOUTPUT BY SDABB     DABB =,F10.5,3X,5HFB  =,
     +       F10.3)
      RETURN
      END
P         END  SDABB
```

Figure C-1
FCF FOR LINK4
(page 35 of 41)

```
P         SNXNZ:
      SUBROUTINE SNXNZ (ACCMA,ACCMAP,ALPHA,CGT,DA,DABB,DBANK,DTHET,DYAW,
     +                  D2THET,GR,XLVO,XMAC,XNXM,XNXMP,XNZM,XNZMP,XNXW,
     +            XNZW,INDEX,JAXIS,I,XNZWF,JFLG3)
C-------------------------------------------------------------------
C-----LOAD FACTOR CALCULATIONS SUBROUTINE
C-----  THIS SUBROUTINE COMPUTES CORRECTED VALUES FOR NORMAL AND FLIGHT
C-----  PATH LOAD FACTORS.  MEASURED VALUES OF NX AND NZ ARE ROTATED
C-----  THROUGH THE REQUIRED ANGLES AND TRANSLATED BETWEEN BCOP AND CG
C-----  AS NECESSARY.
C-------------------------------------------------------------------
      DIMENSION ALPHA(1),CGT(1),DA(1),DABB(1),DBANK(1),
     +          DTHET(1),DYAW(1),D2THET(1), XNXM(1),XNXMF(1),
     +          XNZM(1),XNZMP(1),XNZW(1),XNXW(1),XNZWF(1)
C
C     LOAD FACTOR CALCULATIONS
C
      XLV1   = XLVO + CGT#IBUF * XMAC/100.0
      DNXP   = (XLV1    /GR) * ((DTHET#IBUF ** 2 + DYAW#IBUF ** 2) * COS
     +         (ALPHA#IBUF) - (DBANK#IBUF * DYAW#IBUF - D2THET#IBUF) *
     +         SIN(ALPHA#IBUF))
      DNZP   = (XLV1    /GR) * ((DTHET#IBUF ** 2 + DYAW#IBUF ** 2) *
     +         SIN(ALPHA#IBUF) + (DBANK#IBUF * DYAW#IBUF - D2THET#IBUF)
     +         * COS(ALPHA#IBUF))
C
C     CHECK JFLG3=1 TO SEE IF NECESSARY TO PRINT SECONDARY OUTPUT
C
      IF (JFLG3 .EQ. 1)  WRITE(6,38) DNXP, DNZP
   38 FORMAT(1H ,18X,26HOUTPUT BY SNXNZ      DNXP =,F10.5,3X,5HDNZP=,
     +       F10.5)
C
P        IF JAXIS = '1' THEN
P           DO
C
C     JAXIS=1
C
      IF (INDEX .EQ. 2)  GO TO 50
      XNXWP = XNXMP#IBUF * COS(DA#IBUF - DABB#IBUF + ACCMAP) -
     +        XNZMP#IBUF * SIN(DA#IBUF - DABB#IBUF + ACCMAP)
      XNXW#IBUF=XNXWP + DNXP
      IF (INDEX .EQ. 1)  GO TO 99
   50 XNZWP#IBUF = XNXMP#IBUF * SIN(DA#IBUF - DABB#IBUF + ACCMAP) +
     +            XNZMP#IBUF * COS(DA#IBUF - DABB#IBUF + ACCMAP)
      XNZW#IBUF = XNZWP#IBUF + DNZP
P           END
P        ELSE IF JAXIS = '2' THEN
P           DO
C
C     JAXIS=2
C
      IF (INDEX .EQ. 2)  GO TO 51
      XNXW#IBUF=XNXM#IBUF * COS(ALPHA#IBUF + ACCMA) -
     +          XNZM#IBUF * SIN(ALPHA#IBUF + ACCMA)
      IF (INDEX .EQ. 1)  GO TO 99
   51 XNZW#IBUF=XNXM#IBUF * SIN(ALPHA#IBUF + ACCMA)
```

Figure C-1
FCF FOR LINK4
(page 36 of 41)

```
     +          + XNZM#IBUF * COS(ALPHA#IBLF + ACCMA)
      XNZWP#IBLF = XNZW#IBLF - DNZP
P           END
P         ELSE IF JAXIS = '3' THEN
P           DO
C
C     JAXIS=3
C
      IF (INDEX .EQ. 1)  GO TO 99
      XNZWP#IBUF=(XNZM#IBUF + XNXW#IBUF * SIN(ALPHA#IBLF +
     +           ACCMA)) /COS(ALPHA#IBLF + ACCMA)
      XNZWP#IBLF = XNZW#IBLF - DNZP
P           END
P         ELSE IF JAXIS = '4' THEN
P           DO
C
C     JAXIS=4
C
      IF (INDEX .EQ. 2)  GO TO 52
      XNXW#IBUF=(XNXM#IBUF - XNZW#IBUF * SIN(ALPHA#IBLF +
     +           ACCMA))/COS(ALPHA#IBLF + ACCMA)
      IF (INDEX .EQ. 1)  GO TO 99
   52 XNZWP#IBLF = XNZW#IBLF - DNZP
P           END
P         ELSE IF JAXIS = '5' THEN
P           DO
C
C     JAXIS=5
C
      IF (INDEX .EQ. 1)  GO TO 99
      XNZWP#IBLF = (XNZMP#IBUF + (XNXW#IBLF - DNXP) * SIN
     +             (DA#IBUF - DABB#IBUF + ACCMAP))/
     +            COS(DA#IBLF - DABB#IBLF + ACCMAP)
      XNZW#IBUF = XNZWP#IBLF + DNZP
P         END
P         ELSE IF JAXIS = '6' THEN
P           DO
C
C     JAXIS=6
C
      IF (INDEX .EQ. 2)  GO TO 53
      XNXWP = (XNXMP#IBUF - (XNZW#IBLF - DNZP) * SIN
     +        (DA#IBLF - DABB#IBLF + ACCMAP))/
     +        COS(DA IBUF - DABB IBUF + ACCMAP)
      XNXW#IBLF = XNXWP - DNXP
      IF (INDEX .EQ. 1)  GO TO 99
   53 XNZWP#IBLF = XNZW#IBLF - DNZP
P         END
P         ELSE IF JAXIS = '7' THEN
P           DO
C
C     JAXIS=7
C
P         MESSAGE(I,'W=NO EQUATION YET FOR DATA_SOURCE = 7')
```

Figure C-1
FCF FOR LINK4
(page 37 of 41)

```
      XNXWP =
P        ELSE IF JAXIS = '8' THEN
P           DO
C
C     JAXIS=8
C
P        MESSAGE (I,'W=NO EQUATIONS YET FOR DATA_SOURCE = 8')
      XN2WP =
P           END
P        ELSE IF JAXIS = '9' THEN
P           DO
C
C     JAXIS=9
C
      IF (INDEX .EQ. 1)  GO TO 99
      XN2WP#IBLF = XN2W#IBLF - CN2F
P           END
   99 RETURN
      END
P        END SNXN2
```

Figure C-1
FCR FOR LINK4
(page 38 of 41)

```
P         DALAG1
      OVERLAY (SFEXT,4,2)
      PROGRAM DALAG
C-----------------------------------------------------------------------
C-----DYNAMIC LAG CORRECTION SECONDARY OVERLAY
C-----  THIS PROGRAM COMPLTES THE DYNAMIC LAG CORRECTIONS TO THE ANGLE OF
C-----  ATTACK (DAW).
C-----------------------------------------------------------------------
P         OUTPUT(COMIN,8)
P         OUTPUT(COMOUT,8)
      #CCMCURVE
P         OUTPUT(COML4,8)
      DIMENSION ALFAV1 #IDIM, ACCNF #IDIP, ACCDR #IDIP, ALFAV #IDIP,
     +          ALFAV2 #IDTM, DU1 #IDIM
      WRITE (6,6000)
 6000 FORMAT(14HOENTERED DALAG,//)
P         IF FREQ <> 'COMPUTE' OR
P            DAMP <> 'COMPUTE' THEN
P            DO
C-----------------------------------------------------------------------
C-----PREPARE FOR CURVE LOCKUP-----
C-----------------------------------------------------------------------
P            IF FREQ = 'CURVE' THEN
P               DO
      LOCW = LOCATE(1)
      NXW = CURVE (LOCW+2)
      NZW = CURVE(LOCW+3)
P               END
P            IF DAMP = 'CURVE' THEN
P               DO
      LOCZ = LOCATE(2)
      NXZ = CURVE(LOCZ+2)
      NZZ = CURVE(LOCZ+3)
P               END
P            END
C-----------------------------------------------------------------------
C-----PREPARE FOR DIFFERENTIATION-----
C-----------------------------------------------------------------------
      MFP = MF
      MBP = MB
      INPTS = MFP
      NC = 0
C-----------------------------------------------------------------------
C-----FROM HERE TO 50, INITIALIZE ALFAV AND ALFAV1, AND COMPUTE
C-----      ACCNF AND ACCDR   -----
C-----------------------------------------------------------------------
      DO 50 I=1,IBIN
      ALFAV(I) = -9999.
      ALFAV1(I) = 0.0
C-----------------------------------------------------------------------
C-----COMPUTE ACCNF-----
C-----------------------------------------------------------------------
P         IF FREQ = 'COMPUTE' THEN
      ACCNF(I) = SQRT (CPART(I)*AVANE*XL*CLAV/XIYY)
P         ELSE DO
```

Figure C-1
FCF FOR LINK4
(page 39 of 41)

```
      NC = NC + 1
      CALL TABENT ( AMCT(I), ACCNF(I), CBART(I), NXX, NZX, CURVE(LCC(X+7)
     +              ,INDIC, TIME(I), SHDALAG, NC, I, KXNV )
P           END
C-------------------------------------------------------------------
C-----COMPUTE ACCDR-----
C-------------------------------------------------------------------
P        IF DAPP = 'COMPUTE' THEN
      ACCDR(I) = XL * ACCNF(I)/(2.0 * VTT(I))
P        ELSE DO
      NC = NC + 1
      CALL TABENT ( AMCT(I), ACCDR(I), CBART(I), NXZ,NZZ, CURVE(LCC2+7),
     +              INDIC, TIME(I), SHDALAG, NC, I, KZV  )
P        END
   50 CONTINUE
C-------------------------------------------------------------------
C-----COMPUTE ALFAV-----
C-------------------------------------------------------------------
      ITR = 1
   60 JCNTR = 0
      DO 70 I=1,IPIN
      ALFAVT = ALFAV(I)
      ALFAV(I) = (ALFAM2(I) + 2.0*ACCDR(I)*ACCNF(I)*(ALFAM1(I)-ALFAV1(I))
     +          +ACCNF(I)**2*ALFAM(I)) / (XKT*ACCNF(I)**2)
      IF (T.LT.TACUT .OR. I.GT.IPCLT) GO TO 70
      IF (ABS(ALFAV(I)-ALFAVT).LE.CVDAW) JCNTR = JCNTR+1
   70 CONTINUE
C-------------------------------------------------------------------
C-----CHECK FOR CONVERGENCE OF ALFAV-----
C-------------------------------------------------------------------
      IF (ITR.EQ.1) GO TO 75
      PCTCV = 100.*FLOAT(JCNTR)/FLOAT(IBCUT-IACUT+1)
      IF (PCTCV.GE.PCTDAW) GO TO 90
      IF (ITR.GE.ITRDAW) GO TO 80
C-------------------------------------------------------------------
C-----COMPUTE ALFAV1 AND GO BACK TO 60 FOR ANOTHER ITERATION-----
C-------------------------------------------------------------------
   75 INDCR = 1
      IDIF2=0
      CALL DIFFR(TIME,ALFAV,IC,MF,MB,MFM,MBM,INDCR,INFTS,SDIF,IBIN,CL1,
     +           ALFAV1,ALFAV2,5HALFAV,IDIF2,ISM4,NANC4)
      ITR = ITR+1
      GO TO 60
C-------------------------------------------------------------------
C-----ALFAV DID NOT CONVERGE-----
C-------------------------------------------------------------------
   80 WRITE(6,6080) PCTDAW,CVDAW,ITRDAW,PCTCV
 6080 FORMAT(1HOERROR IN DALAG.  ,F8.3,28H PERCENT OF ALFAV DID NOT CO,
     +       10HNVERGE TO ,F8.5,3H IN,I5,12H ITERATIONS./,17X,F8.3,
     +       52H PERCENT DID CONVERGE.  CURRENT VALUES WILL BE USED.)
C-------------------------------------------------------------------
C-----COMPUTE DAL-----
C-------------------------------------------------------------------
   90 DO 100 I=IACUT,IBCUT
```

Figure C-1
FCF FOR LINK4
(page 40 of 41)

```
      DAL(I) = ALFAV(I)-ALFAM(I)
  100 CONTINUE
C-------------------------------------------------------------------------
C-----SECONDARY OUTPUT-----
C-------------------------------------------------------------------------
      IF (IDBUG4.NE.1) GO TO 200
      WRITE(6,6150) PCTCV,ITR
 6150 FORMAT(1H ,10X,F7.2,31H PERCENT OF ALFAV CONVERGED IN ,I3,
     +       12H ITERATIONS.)
      WRITE(6,6160)
 6160 FORMAT(1H0,12X,4HTIME,10X,5HALFAV,10X,6HALFAV1,10X,5HACCNF,11X,
     +       5HACCDR, 9X,5HALFAM,10X,6HALFAM1, 9X,6HALFAM2)
      WRITE(6,6161)
 6161 FORMAT(1H ,11X,6H(SECS),7X,9H(RAD/SEC),6X,9H(RAD/SEC),6X,6H(RAD/S,
     +       3HEC),6X,9H(RAD/SEC),7X,5H(RAD),9X,9H(RAD/SEC),4X,6H(RAD/S,
     +       6HEC**2),//)
      WRITE(6,6170) (TIME(I),ALFAV(I),ALFAV1(I),ACCNF(I),ACCDR(I),
     +              ALFAM(I),ALFAM1(I),ALFAM2(I),I=IACUT,IBCUT)
 6170 FORMAT ((8X,8(3X,F12.5)))
      WRITE(6,6160)
      WRITE(6,6162)
 6162 FORMAT(1H ,11X,6H(SECS),7X,9H(DEG/SEC),6X,9H(DEG/SEC),6X,6H(DEG/S,
     +       3HEC),6X,9H(DEG/SEC),7X,5H(DEG),9X,9H(DEG/SEC),4X,6H(DEG/S,
     +       6HEC**2),//)
      RAD = 57.2957
      DO 190 I=IACUT,IBCUT
      AV  = ALFAV (I) * RAD
      AV1 = ALFAV1(I) * RAD
      ANF = ACCNF (I) * RAD
      ACR = ACCDR (I) * RAD
      AM  = ALFAM (I) * RAD
      AM1 = ALFAM1(I) * RAD
      AM2 = ALFAM2(I) * RAD
      WRITE(6,6170) TIME(I),AV,AV1,ANF,ACR,AM,AM1,AM2
  190 CONTINUE
      WRITE(6,6190)
 6190 FORMAT(11HOEXIT DALAG)
  200 CONTINUE
      END
P         END  DALAG
```

Figure C-1
FCF FOR LINK4
(page 41 of 41)

```
      OVERLAY(SFEXT,4,0)
      PROGRAM LINK4
C-----------------------------------------------------------------------
C-----PERFORMANCE TEST CONDITIONS PRIMARY OVERLAY
C-----  THIS PROGRAM CONTROLS CALCULATION OF TEST PERFORMANCE PARAMETERS.
C-----  ITS ONLY FUNCTION IS TO CALL INTO EXECUTION THE SUBROUTINES
C-----  INPUT4 AND SFEXT.
C-----------------------------------------------------------------------
      COMMON /ALL/LUIN,LUOUT,LUCURV,LTHRUS,S,XIF,CHORD
      COMMON /L4DATA/ CVALF,CVCLT,CVDAW,IDBUG4,IQ,ISM4,ITRALF,ITRCLT,ITR
     1DAW,KDABR,KFALF,KLALF,KWNV
      COMMON /L4DATA/ DZVMB,MB,MF,NANC4,PCTDAW,SDIF
      COMMON /L1DATA/ ZUD, NDFLAG
C-----------------------------------------------------------------------
      COMMON /L4INTR/ NREM,REMARK(10),NRSECT,NSG,JEND
      NSC = 0
      WRITE (6,6000)
 6000 FORMAT (14HJENTERED LINK4)
   10 CALL INPUT4
      IF (JEND .EQ. 1)  GO TO 999
      CALL SFEXT
      GO TO 10
  999 WRITE(6,6999)
 6999 FORMAT(11HGEXIT LINK4 )
      END
```

Figure C-2
RESULTS OF TEST 4
(page 1 of 6)

```
      SUBROUTINE INPLT4
C-----------------------------------------------------------------------
C-----LINK4 FILE INPUT SECONDARY SUBROUTINE
C-----  THIS PROGRAM LOADS NEEDED CURVES FROM A CURVE FILE INTO CORE AND
C-----  READS PARAMETERS FROM AN INPUT B FILE.  THESE CURVES AND B FILE
C-----  PARAMETER VALUES ARE THEN PASSED TO THE OTHER LINK4 SUBROUTINE
C-----  THROUGH COMMON BLOCKS FOR PROCESSING.
C-----------------------------------------------------------------------
      COMMON /ALL/ LUIN,LUOUT,LUCURV,LTHRUS,S,XIF,CHORD
      COMMON /L4DATA/ CVALF,CVCLT,CVDAW,IDBUG4,IO,ISM4,ITRALF,ITRCLT,ITR
     1DAW,KDABB,KFALF,KLALF,KWNV
      COMMON /L4DATA/ DZVMB,MB,MF,NANC4,PCTDAW,SDIF
      COMMON /L1DATA/ ZOD, NDFLAG
C-----------------------------------------------------------------------
      COMMON /L4IN/ TIME,ALFAM,AMCT,DAT,D2T,CGT,FET,FGT,FRT,P,Q,QDOT,QBA
     1RT,R,THAT,TH2T,VIT,WFT,XNXM,XNZM,BETA,GAMA2,GAMA3
      COMMON /L4OUT/ALFAT,CL1,FEAT,FNDA,FNDT2,FNT,HETDOT,SFCORA,SFCORT,
     1TSFC,XNZWTP,XNZWT,XNXWT,CDT,DALFA,DELTA,ZETA,OMEGA

      COMMON /L4INTR/ NREM,REMARK(10),NRSECT,NSG,JEND
      DIMENSION PN1(22), PN2(22), VAL(22), TN1(200),
     1 TN2(200), LISTNO(200), V(200), NUM(22)
      EQUIVALENCE (VAL(1),ALFAM)
      DATA PN1/5HALFAM,4HAMCT,3HDAT,3HD2T,3HCGT,3HFET,3HFGT,3HFRT,1HP,1H
     1Q,4HQDOT,5HQBART,1HR,4HTHAT,4HTH2T,3HVIT,3HWFT,4HXNXM,4HXNXM,4HBET
     2A,4HGAMA,4HGAMA/
      DATA PN2/1H ,19*1H ,4H TWO,6H THREE/
      NVAR = 22
      KLIM = 1000
      NSG = NSG + 1
      WRITE (6,6000)
 6000 FORMAT (15HOENTERED INPUT4,//)
      IF (NSG .GT. 1)  GO TO 200
C-----------------------------------------------------------------------
C-----PRINT INPUTS-----
C-----------------------------------------------------------------------
    4 WRITE(6,6101) LUIN,LUOUT,LUCURV,LTHRUS,S,XIF,CHORD
 6101 FORMAT(6X,32HPARAMETERS FROM COMMON BLOCK ALL,/,
     110X,7HLUIN  =,I5    ,/,10X,7HLUOUT =,I5    ,/,10X,7HLUCURV=,I5    ,/,
     210X,7HLTHRUS=,I5    ,/,10X,7HS     =,E14.6,/,10X,7HXIF   =,E14.6,/,
     310X,7HCHORD =,E14.6,/)
      WRITE (6,6102) CVALF,CVCLT,CVDAW,IDBUG4,IO,ISM4,ITRALF,ITRCLT,
     1ITRDAW,KDABB,KFALF,KFDAU,KLALF,KLDAU,KWNV,KZV,MB,MF,NANC4,
     2PCTDAW,SDIF
 6102 FORMAT(6X,35HPARAMETERS FROM COMMON BLOCK L4DATA,/,
     110X,7HCVALF =,E14.6,/,10X,7HCVCLT =,E14.6,/,10X,7HCVDAW =,E14.6,/,
     2E14.6,/,10X,7HIDBUG4=,I5    ,/,10X,7HIO    =,I5    ,/,
     310X,7HISM   =,I5    ,/,10X,
     410X,7HITRALF=,I5    ,/,10X,7HITRCLT=,I5    ,/,10X,7HITRDAW=,I5    ,/,
     5                       10X,/,10X,7HKDABB =,I5    ,/,
     610X,7HKFALF =,I5    ,/,10X,7HKFDAU =,I5    ,/,10X,7HKLALF =,I5    ,/,
     710X,7HKLDAU =,I5    ,/,10X,7HKWNV  =,I5    ,/,10X,7HKZV   =,I5    ,/,
     8/,10X,7HMB    =,I5    ,/,10X,7HMF    =,I5    ,/,
     910X,7HNANC4 =,I5    ,/,10X,7HNREM4 =,I5    ,/,10X,7HPCTDAW=,E14.6,/,
     D10X,7HSDIF  =,E14.6 )

      REWIND LUIN
      REWIND LUOUT
C-----------------------------------------------------------------------
C-----READ INPUT B FILE LABEL RECORD-----
C-----------------------------------------------------------------------
      READ (LUIN) L,NRSECT,NREM,(REMARK(I),I=1,NREM), NLAB,(TN1(I),
     1          TN2(I),LISTNO(I),I=1,NLAB)
      CALL LABSOR (PN1,PN2,TN1,TN2,LISTNO,NUM,NVAR+1,NLAB)
C-----------------------------------------------------------------------
C-----DETERMINE IF NEEDED PARAMETERS ARE AVAILABLE-----
C-----------------------------------------------------------------------
      DO 150 I=1,NVAR
      IF (NUM(I).GT.0) GO TO 150
C-----NEEDED PARAMETER NOT AVAILABLE-----
  140 WRITE(6,6140) PN1(I), PN2(I)
 6140 FORMAT(28H ERROR IN LINK4.  PARAMETER ,2A6,16H NOT AVAILABLE ON ,
     1       13HINPUT B FILE.)
      IGO = 0
C-----------------------------------------------------------------------
C-----IF ANY ERRORS WERE ENCOUNTERED, SET NDFLAG=3 AND RETURN-----
C
```

Figure C-2
RESULTS OF TEST 4
(page 2 of 6)

```
C-----------------------------------------------------------------------
  150 IF (IGC .EC. 0)  NDFLAG = 3
      IF (IGC .EC. 0)  GO TO 260
C-----------------------------------------------------------------------
C-----READ VALUES FOR PARAMETERS-----
C-----------------------------------------------------------------------
  200 READ (LUIN) N,I,(V(J),J=1,N)
      IF (N .NE. 0)  GO TO 270
      IZ = 0
      Z = 0
      WRITE(LUOUT) IZ,Z,Z
  260 JEND = 1
  270 RETURN
      END
```

Figure C-2
RESULTS OF TEST 4
(page 3 of 6)

```
      SUBROUTINE SFEXT
C-----------------------------------------------------------------------
C-----TEST EXCESS THRUST SECONDARY OVERLAY
C-----  THIS SECONDARY OVERLAY COMPUTES THE BOOM BENDING, UPWASH AND
C-----  PITCH RATE CORRECTIONS TO THE ANGLE OF ATTACK (DABB,DAU,DAQ),
C-----  TRUE ANGLE OF ATTACK (ALFAT), LOAD FACTORS (XNX,XNZ), LIFT
C-----  COEFFICIENTS (CLI,CDT), RATE OF CHANGE OF ENERGY HEIGHT (HETDOT),
C-----  AND EXCESS THRUST (FEXT) FOR TEST CONDITIONS. IT THEN WRITES THEM
C-----  ON THE OUTPUT B FILE.
C-----------------------------------------------------------------------
      COMMON /ALL/ LUIN,LUOUT,LUCURV,LTHRUS,S,XIF,CHORD
      COMMON /L4DATA/ CVALF,CVCLT,CVDAW,IDBUG4,IG,ISM4,ITRALF,ITRCLT,ITR
     1DAW,KDABB,KFALF,KLALF,KWNV
      COMMON /L4DATA/ DZVMB,MB,MF,NANC4,PCTDAW,SDIF
C-----------------------------------------------------------------------
      COMMON /L4IN/ TIME,ALFAM,AMCT,DAT,D2T,CGT,FET,FGT,FRT,P,Q,QDOT,QBA
     1RT,R,THAT,TH2T,VTT,WFT,XNXM,XNZM,BETA,GAMA2,GAMA3
      COMMON /L4OUT/ALFAT,CLT,FEXT,FNDA,FNDT2,FNT,HETDOT,SFCORA,SFCORT,
     1TSFC,XNZWTP,XNZWT,XNXWT,CDT,DALFA,DELTA,ZETA,OMEGA

      COMMON /L4INTR/ NREM,REMARK(10),NRSECT,NSG,JEND
      DIMENSION  V(18), PN1(18), PN2(18)
      DATA PN1/5HALFAT,3HCLT,4HFEXT,4HFNDA,5HFNDT2,3HFNT,6HHETDOT,6HSFCO
     1RA,6HSFCORT,4HTSFC,6HXNZWTP,5HXNZWT,5HXNXWT,3HCDT,5HDALFA,5HDELTA,
     24HZETA,5HOMEGA/
      DATA PN2/1H ,17*1H /
      WRITE (6,6000)
 6000 FORMAT(14HCENTERED SFEXT)
      I2SFEX = IDBUG4
      I2SNXN = IDBUG4
      IF (I2SFEX.EQ.1) WRITE(6,6005)
 6005 FORMAT(17H0            TIME    ,/)
      NDIM = KDIM-LOC
      IF(I2SFEX.EQ.1) WRITE(6,6010)TIME
 6010 FORMAT(1H ,7X,F10.3)
C-----------------------------------------------------------------------
C-----COMPUTE INITIAL CLT AND XLV----
C-----------------------------------------------------------------------
      XNZWP= XNZM
      CLTI = XNZWP*WT/ (QBART*S)
      XLV= XLVO+CGT*CHORD/100.
      IF (I2SFEX.EQ.1) WRITE(6,6050)XLV
 6050 FORMAT(1H ,18X,26HOUTPUT BY SFEXT      XLV  =,F10.5)
      ALFAT= ALFAM
C-----------------------------------------------------------------------
C-----CALL SNXNZ FOR XNZWT AND XNZWP -----
C-----------------------------------------------------------------------
      INDEX = 2
      CALL SNXNZ (ACCMA,ACCMAP,ALFAT,CGT,DALFA,DABB,P,Q,R,QDOT,GR,
     1            XLVO, CHORD, XNXM, XNXMP, XNZM, XNZMP, XNXWT, XNZWT,
     2            INDEX, JAXIS, I, XNZWP, I2SNXN )
C-----------------------------------------------------------------------
C-----IF CLT IS TO BE CORRECTED FOR, COMPUTE CLT-----
C-----------------------------------------------------------------------
      CLTI = (XNZWT* WT - FGT* SIN(ALFAT+ XIF))/(QBART* S)
C-----------------------------------------------------------------------
C-----CALL SNXNZ FOR XNXWT AND COMPUTE HETDOT AND FEXT----------------
C-----------------------------------------------------------------------
  160 INDEX = 1
      CALL SNXNZ (ACCMA,ACCMAP,ALFAT,CGT,DALFA,DABB,P,Q,R,QDOT,GR,
     1            XLVO, CHORD, XNXM, XNXMP, XNZM, XNZMP, XNXWT, XNZWT,
     2            INDEX, JAXIS, I, XNZWP, I2SNXN )
      HETDOT= XNXWT*VTT
      FEXT= XNXWT*WT
      CLT= CLTI
C-----------------------------------------------------------------------
C-----COMPUTE CDT AND FNT -----
C-----------------------------------------------------------------------
      FNT= FGT * COS(ALFAT+ XIF) - FET
C-----------------------------------------------------------------------
C-----COMPUTE FNDA, FNDT2, TSFC, SFCORA, SFCORT, AND TSFC--------------
C-----------------------------------------------------------------------
      FNDA= FNT/DAT
      FNDT2= FNT/D2T
      SFCORA= WFT/(THAT**ETHETA * FNT)
      SFCORT= WFT/(TH2T**ETHETA * FNT)
      TSFC= WFT/(FGT- FRT)
```

Figure C-2
RESULTS OF TEST 4
(page 4 of 6)

```
      IF (I2SFEX.EQ.1) WRITE(6,6190) CUT,ALFAT,FEXT
 6190 FORMAT(19X,15HOUTPUT BY SFEXT,5X,4HCDT=,F10.5,6X,6HALFAT=,F10.5,
     1       5X,5HFEXT=,F10.2)
  200 CONTINUE
      NPASS = 0
      IF (NSG.EQ.1) NPASS = 1
      CALL WTBDAS(LUOUT,0,V,TIME,1,1,1,16,PN1,PN2,NPASS,
     1END,NRSECT,NREM,KEMARK,I2SFEX)
      IF (I2SFEX.EQ.1) WRITE(6,6200)
 6200 FORMAT(37HORETURN TO CALLING ROUTINE FROM SFEXT)
      RETURN
      END
```

Figure C-2
RESULTS OF TEST 4
(page 5 of 6)

```
      SUBROUTINE SNXNZ (ACCMA,ACCMAP,ALPHA,CGT,DA,DABB,DBANK,DTHET,DYAW,
     1                  D2THET,GR,XLVO,XMAC,XNXM,XNXMP,XNZM,XNZMP,XNXW,
     2            XNZW,INDEX,JAXIS,I,XNZWP,JFLG3)
C-----------------------------------------------------------------------
C-----LOAD FACTOR CALCULATIONS SUBROUTINE
C-----  THIS SUBROUTINE COMPUTES CORRECTED VALUES FOR NORMAL AND FLIGHT
C-----  PATH LOAD FACTORS. MEASURED VALUES OF NX AND NZ ARE ROTATED
C-----  THROUGH THE REQUIRED ANGLES AND TRANSLATED BETWEEN BOOM AND CG
C-----  AS NECESSARY.
C-----------------------------------------------------------------------
C
C     LOAD FACTOR CALCULATIONS
C
      XLV1   = XLVO + CGT* XMAC/100.0
      DNXP   = (XLV1    /GR) * ((DTHET** 2 + DYAW** 2) * COS          (ALP
     1HA) - (DBANK* DYAW- D2THET) *                SIN(ALPHA))
      DNZP   = (XLV1    /GR) * ((DTHET** 2 + DYAW** 2) *              SIN(
     1ALPHA) + (DBANK* DYAW- D2THET)                 * COS(ALPHA))
C
C     CHECK JFLG3=1 TO SEE IF NECESSARY TO PRINT SECONDARY OUTPUT
C
      IF (JFLG3 .EQ. 1)  WRITE(6,30) DNXP, DNZP
   30 FORMAT(1H ,1PX,26HOUTPUT BY SNXNZ     DNXP =,F10.5,3X,5HDNZP=,
     1       F10.5)
C
C
C     JAXIS=2
C
      IF (INDEX .EQ. 2)  GO TO 51
      XNXW=XNXM* COS(ALPHA+ ACCMA) -                                XNZM* S
     1IN(ALPHA+ ACCMA)
      IF (INDEX .EQ. 1)  GO TO 99
   51 XNZW=XNXM* SIN(ALPHA+ ACCMA)                                  + XNZM*
     1 COS(ALPHA+ ACCMA)
      XNZWP= XNZW- DNZP
   99 RETURN
      END
```

Figure C-2
RESULTS OF TEST 4
(page 6 of 6)