A SOFTWARE DESIGN FOR THE PROGRAMMING LANGUAGE PLANS

By

CAROL ANNE SAMUEL

Bachelor of Arts

University of Rochester

Rochester, New York

1967

Submitted to the Faculty of the
Graduate College of the
Oklahoma State University
in partial fulfillment of
the requirements for
the Degree of
MASTER OF SCIENCE
July, 1982

A SOFTWARE DESIGN FOR THE PROGRAMMING

LANGUAGE PLANS

Thesis Approved:

James R. Van Doren
Thesis Adviser

Donald D. Fisher

D. E. Hedrick

Norman N. Durham
Dean of Graduate College

PREFACE

This thesis serves two purposes. The first is to provide an explanation of the system design of an implementation of the Programming Language for Allocation and Network Scheduling (PLANS). The second is to document the extension of the language to support character string variables and operations. It is assumed that the reader has some knowledge of the PLANS language and will refer to the User's Guide for an explanation of the allowable statement forms [6].

The software referenced is written in FORTRAN 77. This set of programs is in the possession of Dr. James R. Van Doren, Computing and Information Sciences Department, Oklahoma State University.

I would like to thank Dr. Donald D. Fisher and Dr. George E. Hedrick for serving on my committee and for their help and guidance throughout my graduate study. Special thanks are due Dr. Fisher who provided the encouragement I needed to get me going again.

I am particularly grateful to Dr. James R. Van Doren, my major adviser, who was always available to patiently answer my innumerable questions. He provided me with a framework in which I was encouraged to think, to discuss and to learn but in which frustrations were kept to a minimum.

I wish to thank my entire family for their many sacrifices but particularly my husband, Mark, who never faltered in his encouragement and confidence in my abilities.

TABLE OF CONTENTS

Chapter Page

I. INTRODUCTION 1

Overview 2
History 3

II. THE TRANSLATOR 5

Introduction 5
File, Storage and Data Structures 9
The Library File 9
The Reserved Label/Value File . . . 9
Binary Search Trees 10
Dynamic Storage Allocation 11
The Constant Tables 12
The Procedure/Block Table 13
Local Symbol Tables 14
The Reserved Label/Value Table . . . 16
The Parsing Tables 17
The Pseudo-Code Table 18
The Translator Storage Block 19
The Run File 20
Subsystem Organization 23
Overview 23
Main Control 24
Parsing/Code Generation 25
Error Handling 27

III. THE INTERPRETER 29

Introduction 29
Run-Time Storage Organization 30
Pseudo-Machine Storage Space 30
The Pseudo-Machine Space Control Block 35
The Operation Code Control Table . . 36
The Run-Time Stack 37
Subsystem Organization 40
Overview 40
The Fetch/Execute Cycle 41

IV. THE CHARACTER STRING EXTENSION 43

Introduction 43

Chapter | Page

Existing Character String Features . . . 44
Changes Affecting the User 45
Explicit Data Types 45
String Functions 46
Error Handling 47
Translator Changes 47
Explicit Data Types 47
Lexical Analyzer Changes 48
New Operation Codes 48
Parser Changes 50
Interpreter Changes 51
String Handling 51
New Instructions 53

V. CONCLUSIONS 55

Implementation Completion 55
Possible Further Enhancements 57

A SELECTED BIBLIOGRAPHY 59

APPENDIX A - INDEX TO THE TRANSLATOR SUBSYSTEM 61

APPENDIX B - TRANSLATOR COMMON BLOCKS 69

APPENDIX C - TRANSLATOR PARSER/CODE GENERATOR ROUTINE WITH GRAMMAR RULES INCLUDED 72

APPENDIX D - TRANSLATOR SOURCE ROUTINES 78

APPENDIX E - CHANGES TO THE TRANSLATOR AS A RESULT OF THE CHARACTER STRING EXTENSION 100

APPENDIX F - INDEX TO THE INTERPRETER SUBSYSTEM . . . 105

APPENDIX G - INTERPRETER COMMON BLOCKS 110

APPENDIX H - INTERPRETER SOURCE ROUTINES 113

APPENDIX I - PSEUDO-MACHINE INSTRUCTION SET 132

APPENDIX J - INTERPRETER PDL'S FOR THE CHARACTER STRING EXTENSION 151

LIST OF FIGURES

Figure		Page
1.	The Three Translator Modes	6
2.	A PLANS Tree	8
3.	The Translator Storage Block	20
4.	The Run File	21
5.	Run-Time Storage Organization	29
6.	Pseudo-Machine Storage	31
7.	PLANS Tree Node Representation	31
8.	PDL for Fetch/Execute Control Loop	42
9.	A Sample of Valid PLANS Statements	46
10.	New Grammar Rules	50

CHAPTER I

INTRODUCTION

The Programming Language for Allocation and Network Scheduling (PLANS) is a high level language which supports dynamic manipulation of tree data structures. It was originally defined in 1973-74 as a suitable language to support heuristic scheduling algorithms [6].

The prototype version, which functioned in a PL/I environment, was intended primarily to illustrate the language design feasibility. The design of the system reported in this thesis was heavily influenced by several other objectives. Portability was an important concern and FORTRAN 77 was chosen because of its general availability. There was also a critical need for efficient space usage and fast execution relative to previous implementations of the language.

Originally, the language was intended for batch processing and the present software was designed to implement this. However, the data structure manipulation functions in PLANS are sufficiently rich that its extension to support interactive data structure manipulation is considered desirable. Character string variables and conventional string operations were not included in the original design and are needed to accomplish the extension.

The implementation of the PLANS system referenced in this thesis is under continuing development on the IBM 370/168 computer at the Oklahoma State University Computing Center. It is written in FORTRAN 77 and includes a translator and an interpreter. A library of PLANS subroutines (PLUSLIB) relevant for scheduling applications is also available.

Overview

The major objectives of this thesis are to provide an explanation of the system design of an implementation of the PLANS language and to define and document the extension of the language to support character string variables and operations. The software consists of a translator and an interpreter which are described in Chapters II and III respectively. The character string extension is explained in Chapter IV. Chapter V serves as a summary chapter.

Chapter II describes the translator as it was defined prior to extending it. Its primary function is to accept PLANS programs as input and translate them into pseudo-code for a hypothetical "PLANS machine". The techniques used for lexical analysis, parsing, code generation and error handling are described along with the tables and files generated in the process and their respective storage methods. The result of successful translation is a run file which contains the pseudo-code generated, along with tables and other critical information required by the interpreter.

Chapter III descibes the interpreter which accepts the run file produced by the translator and uses the pseudo-code therein to execute a PLANS program. The techniques used for fetching, decoding and executing the pseudo-code are described along with the supporting file and data structures.

Chapter IV discusses the character string extension in terms of how it affects the user, the translator and the interpreter. A complete explanation of the PLANS statements added is presented, along with a detailed discussion of the modifications and additions required.

Chapter V relates the author's experience in managing the modifications mentioned. (The software development environment clearly has a significant impact on one's productivity for a large and complex software project.) Also some possibly desirable language enhancements are presented.

History

In 1973, the NASA Lyndon B. Johnson Space Center contracted the Martin Marietta Corporation, Denver Division (Contract NAS 9-13616) to design a high level language to achieve a single goal:

> to allow the designer of experimental or constantly changing scheduling and resource allocation algorithms to translate his algorithm designs to working code directly from their basic functional descriptions, without intermediate

> detailed program design steps, without highly specialized programming expertise, and at a minimum span time and manpower costs [6, p. 1.1].

The result was the Programming Language for Allocation and Network Scheduling (PLANS) which incorporated dynamic manipulation of tree data structures at execution time allowing easy, direct expression of the kinds of functions frequently found in scheduling and resource allocation programs. The prototype software implementation of PLANS functioned in a PL/I environment.

It is important to note that PLANS is a data structure manipulation language. For this reason, its applicability transcends its original functional design goal.

The major portion of the present version of the PLANS software was designed and implemented by the author's adviser, Dr. James R. Van Doren while employed by Science Applications, Inc. (Englewood, Colorado) during a leave of absence from Oklahoma State University. It was installed on a PRIME minicomputer and a UNIVAC 1110 system under his direction. The funding for this was again provided by the NASA Lyndon B. Johnson Space Center. This version applied the experience and lessons learned from the PL/I based prototype to substantially reduce execution storage requirements and improve the execution speed of PLANS tree manipulation operations.

CHAPTER II

THE TRANSLATOR

Introduction

This chapter is divided into two main sections. The first describes the structure and purpose of the tables and files generated by the translator and the second describes the subsystem organization.

The principal purpose of the translator is to generate a file containing pseudo-code, tables and other necessary information for an idealized hypothetical "PLANS machine". This file is called the "run file" and represents the interface between the translator and the interpreter [7]. Secondary purposes are to create a reserved label/value file and an indexed library file. Special translator options are specified for the creation of the latter. Figure 1 depicts the input and output associated with each of these modes.

It is essential to minimize space usage because of the possibility of large tree structures. Since these trees often contain a significant degree of redundancy in their labels and values, global tables for numeric and string constants are used so that only one copy, rather than multiple copies, of a constant need exist. In particular, due to the dynamic nature of PLANS trees, a reserved

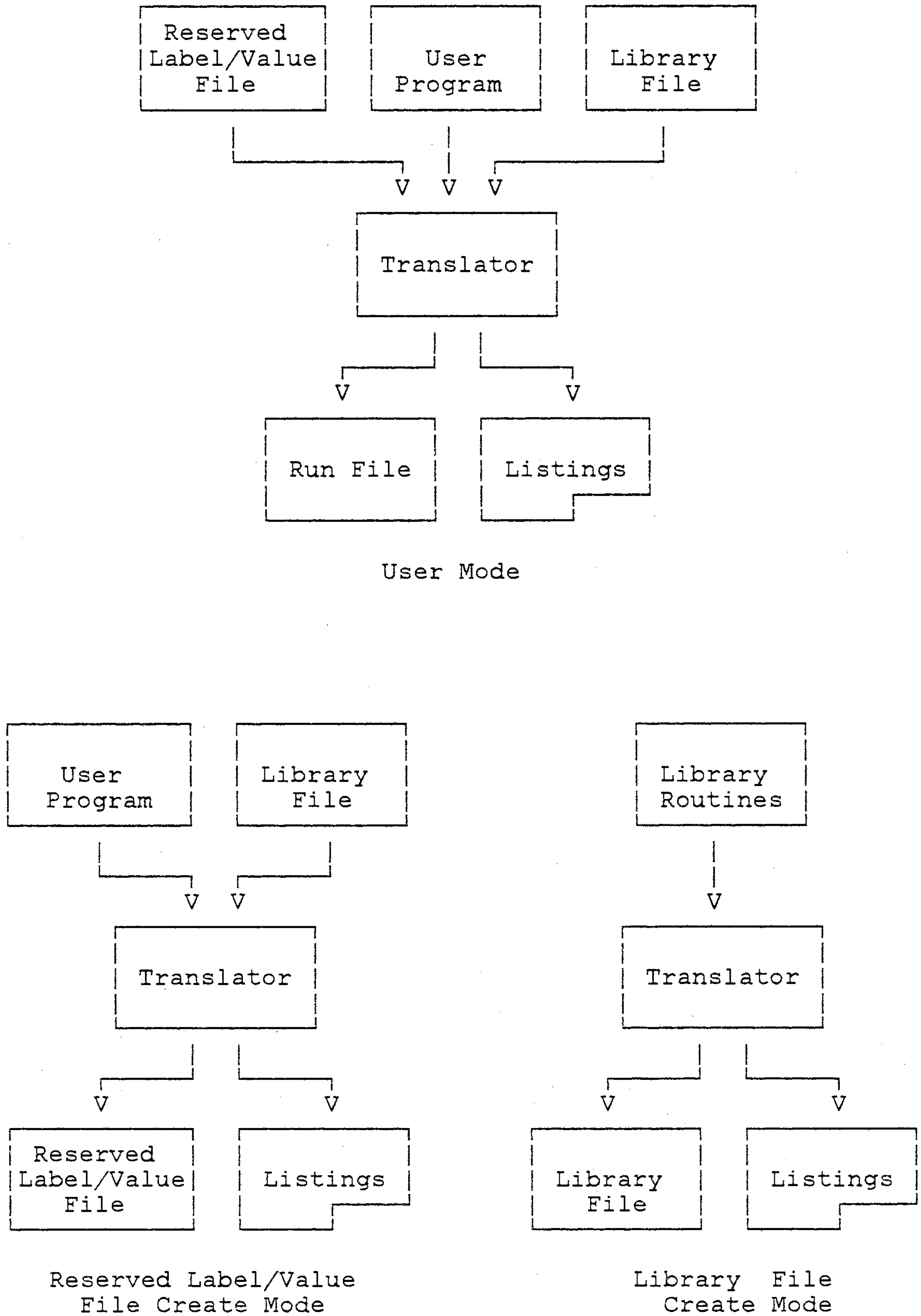
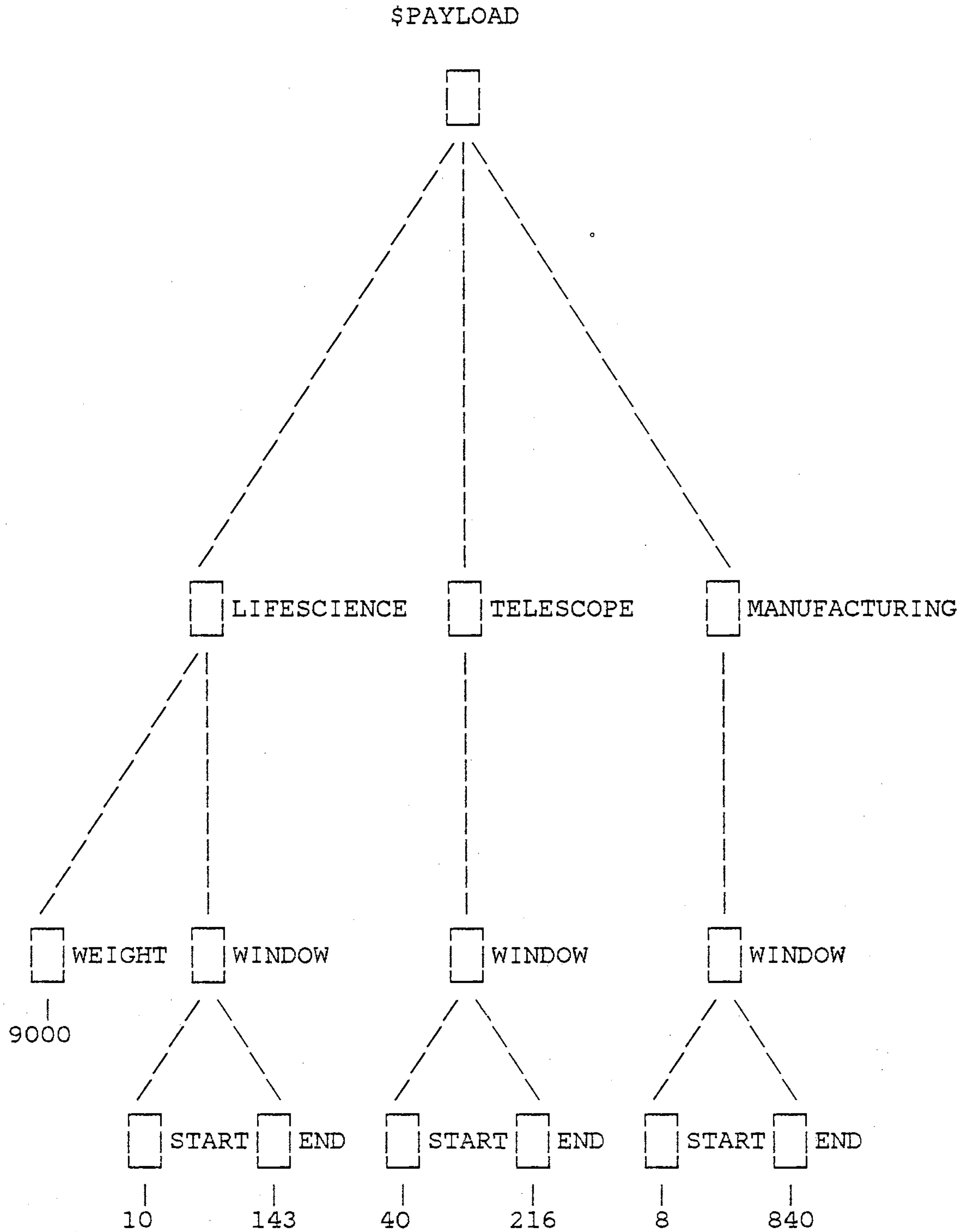


Figure 1. The Three Translator Modes

label/value global table is incorporated. (The reader should reference Figure 2 for an example of the redundancy in PLANS trees and the PLANS User's Guide for the use of strings for tree labels and values [6].) Its use is critical at interpretation time to avoid redundant copying of character strings if at all possible.

As a consequence of the use of global tables, no linkage editor exists. Rather than machine code, very compact pseudo-code designed specifically to support PLANS language features is generated. The translator was designed to be efficient since it is necessary to re-translate all modules, including relevant library routines, each time. Only one pass is made over the source code and the parser is "hard coded" to minimize translation time.

Since the reserved label/value and library files must be created prior to being input to the translator in user mode (see Figure 1), they are described first. This is followed by a description of the tables used in the translator and the support structures necessary to implement them [2,5,12]. There are six classes of tables consisting of the constant tables, the procedure/block table, local symbol tables (identifiers local to a procedure or begin block), the reserved label/value table, parsing tables and the pseudo-code table [8]. The first three and sixth are constructed as parsing proceeds. The fourth one (optional) is constructed from a reserved label/value file at translator initialization time. The fifth class of tables



Source: PLANS User's Guide [6, p. 2.5]

Figure 2. A PLANS Tree

is a static set of tables associated with parsing and lexical analysis. The organization of the tables is then discussed followed by a description of the generated run file. For a detailed description of the translator subsystems, including the common blocks, the grammar rules and the individual subroutines see Appendices B, C and D, respectively. (Appendix A serves as an index, in collating sequence order, into these three appendices.)

File, Storage and Data Structures

The Library File

The library file contains a number of PLANS routines which can be called from a user's PLANS program. If an addition, change or deletion of a library file routine is required, the entire file must be reprocessed using the library creation mode of the translator (*CREATELIB control record). An indexed direct access file is created by subsystem LIB (library) so that later access to it may be effected by procedure name. No code is generated since the library routines are kept in source form but syntax checking is done on the files during creation. (Object files are never created for reasons mentioned above.)

The Reserved Label/Value File

The purpose of the reserved label/value file is to save both time and space, particularly with respect to PLANS tree label and string values. In effect the creation and use of reserved labels and values amounts to a two pass use of the translator. If the reserved label/value feature is not used, an enormous amount of dynamic storage may be required at run time with accompanying execution speed penalties.

If a user's program is being translated for execution, the reserved label/value file, if it exists, is entered into the reserved label/value table in the translator storage block (see below). Both the file and table are static at this time. Any strings not found in this table are entered in the global string constant table instead.

If the translator is in reserved label/value file create mode (*CREATELBVL control record), then all tree label and value strings in the user program detected by the translator are entered in the global string constant table. At the end of the run, that table is traversed in collating sequence order and the string constants are written to the reserved label/value file for later use in actual translation.

Binary Search Trees

The subsystem SRCH (search) contains all the subroutines associated with binary tree management used for the constant and local symbol tables. The common block TABLE contains the binary tree node structure. The trees created and maintained are of the unconstrained variety. (Balancing techniques are not used.)

Each node is of the form

LLEFT	LRIGHT	LNAME	LVALUE

where LLEFT is the left link and LRIGHT is the right link. LNAME and LVALUE are dependent upon the type of token that the node represents.

A common pool of nodes is maintained for all the tables which are in binary tree form. The available list is maintained by using the node in position one as a dummy node. Its left link is used as a root node pointer and its right link is used as a list header for available nodes. Right links of available nodes are used to linearly link this list while the remaining node fields are set to null.

Dynamic Storage Allocation

The subsystem DYNAM (dynamic storage) contains all the subroutines associated with dynamic storage management. In the translator, dynamic storage is used for all strings not in the reserved label/value table. When an exit from translating a PLANS procedure occurs, all the space in dynamic storage used by the strings associated with its local symbol table is released for reuse.

Dynamic storage management is based on the generalized Fibonacci sequence based buddy system [1,3]. Storage is allocated in block sizes according to a specified generalized Fibonacci sequence of the form

$$SIZE(n) = SIZE(n-1) + SIZE(n-k)$$

where the entire set of block sizes are determined by picking k and the first k values of SIZE. Allocation of a block of storage is consistent with the request size. Only sizes in the specified sequence are actually allocated and a large block may be split to satisfy the request. Upon deallocation, the block is merged with its buddy if it is free. This merging continues as far as possible, within the buddy system.

The named common block DSTORP (dynamic storage pool) contains the dynamic storage space in the array ISBLK.

The Constant Tables

The three types of constant tables are the string constant, the integer constant and the floating point constant tables. They are constructed as binary search trees with the lexical analyzer (subsystem LEX) having the responsibility for building them. All of these tables are global to the entire program being translated (including library modules). Thus, if the integer constant 3 occurs in eight external procedures, there will be only one copy of it. None of the three tables is ever cleared or has any entries deleted.

Subsystem LEX (lexical analysis) converts numeric constants to internal form with the LNAME node field used to hold this internal representation. The LVALUE node field is used to represent the integer or floating point address space location.

For string constants, the LNAME node field contains a pointer to the ISBLK dynamic storage block where such a constant is actually stored and the LVALUE field contains a string constant address space location. Subsystem DYNAM (dynamic storage) is used to allocate space from the dynamic storage pool. String constants arise from delimited strings or tree label qualifiers that are not in the reserved label/value table.

The Procedure/Block Table

The procedure/block table contains one entry for each procedure or BEGIN block processed (or procedure CALLed but not translated yet) with the following fields:

BNAME	pointer to procedure name in dynamic storage (null for begin block)
BTYPE	block type code
BLEVEL	block level
BPARNT	index of parent block entry
BPCNT	parameter count (Valid only for procedure blocks)
BBTSPT	block binary tree symbol table pointer (root node of binary tree)
BADDRS	pseudo-code address of the first word of generated code for the block (relative line number for library modules during library creation)
BUNDRF	number of undefined program labels in the block
BASTOR	display storage requirement

The table serves several other purposes as well. It contains procedure control information placed in the run file and used by the interpreter at run time. It contains unresolved procedure name information for consulting a supplied PLANS library. If in library create mode, it contains index information to be placed in the library file as a result of the library creation run.

In addition, there is a "root" block table whose position is always one. The root block entry is the ultimate ancestor of all other entries and the direct ancestor of external procedures. Its BBTSPT field identifies a symbol table whose only correct entries are file names. In effect, these names are external symbols. If a severe block nesting level error occurs, other symbols may get entered, but code generation and run file building are suspended.

The procedure/block table is essentially a tree structured table in which the tree structure corresponds to the procedure/block nesting structure of a PLANS program.

The common block TABLE contains the components of the table and the subsystems BLKTB (block table) and LEX (lexical analysis) contain the routines which access and/or modify it.

Local Symbol Tables

There is a local symbol table for a procedure block or BEGIN block identified by the corresponding procedure/block table entry and each such table is maintained in the form of a binary search tree.

The three general classes of symbols which may be placed in a local symbol table are tree name, program label (statement label) and array or variable name. For each type of entry, the binary tree node field LNAME contains a pointer to the character string for the symbol in dynamic storage. (DSALC is used to allocate space for such a name.) The field LVALUE contains a pointer to a two word dynamic storage block which is used as follows:

Word 1 - pseudo-machine display storage address

Word 2 - (4 bytes from left to right)

Byte 1 - major type code
- T - tree name
- L - program label
- I - integer variable
- F - floating point variable
- S - string variable (character string extension)

Byte 2 - subtype code
- Tree names
  - N - not referenced but declared
  - R - standard tree name reference
  - D - defined reference (DEFINE statement or USING pointer clause reference)
  - P - procedure formal parameter
- Program labels
  - U - referenced but undefined
  - D - defined
- Variable names
  - N - normal local reference
  - P - formal procedure parameter

Byte 3 - level code
- Tree name or variable name
  - Block level
- Program label (statement label)
  - Do group nesting level

Byte 4 - number of dimensions (valid only for declared variable array names)

For tree names and variable names, scope considerations apply. If a locally used name is not declared local or is not in the formal parameter list, its table entry defaults to the outer most block in which a local or parameter declaration occurs. If no such explicit declaration exists, it defaults to the containing external procedure's local symbol table. Program labels are local only. A direct jump out of a BEGIN block or internal procedure is strictly prohibited. Such an attempt will result in an undefined program label error in the block in which such a jump (GO TO) is attempted.

When translation of a program block is complete, and no request for a symbol table listing option has been made, the local symbol table is cleared. (If a request for the listing option is made, the table is retained until the containing external procedure has been translated.) Dynamic storage blocks for symbols and values are released and binary tree nodes are made available for additional use.

The Reserved Label/Value Table

The use of this table is the most critical consideration in the generation of efficient PLANS pseudo-code in terms of both time and storage. It is loaded at translator initialization time from a reserved label/value file if it was created during a previous run in which the reserved label/value processing option was selected.

A pointer vector initialized in ISBLK (see DSTORP common block and LBVLP vector equivalence with ISBLK) supports binary searching on variable length character strings. The character strings representing the reserved labels/values are loaded just beyond the pointer vector.

Actual dynamic storage initialization cannot take place until this table is loaded. (The DSTART variable in DSTORP controls the starting point.)

Each delimited character string or the label qualifier in processed PLANS statements is checked against this table. If it is not present, then and only then will the string be entered in the string constant table.

All pseudo-code references to tree label qualifiers or delimited strings which are in the reserved label/value table are by relative order in the table. This table becomes part of the generated run file and eliminates redundant storage of such character strings at interpretation time when trees are constructed and manipulated. In addition, the collating sequence order facilitates simple integer address tests in lieu of character string comparisons.

The subsystem LABVAL (label/value) contains the management routines for this table.

<u>The Parsing Tables</u>

The parsing tables consist of the key symbol and the key word tables. The subsystem LEX (lexical analysis) is used to detect these tokens with a binary search from subsystem SRCH (search) used to traverse the key word table.

The strings which make up the key (reserved) words are initialized in collating sequence order using DATA statements in TRINI (translator initialization). This list is accessed via a table of pointers, KWP, which contains the addresses of the reserved word strings. (Actually this list is stored in the same array used for dynamic storage (ISBLK) in order to facilitate uniform addressing of strings.)

A major type, NEWSTT, and a subtype, NEWTOK, are set in the subsystem LEX (lexical analysis) to classify the tokens extracted. The address values of KWP are equivalenced in the common block TOKDAT (token data) to symbolic constants which represent the key words. Symbolic constants are set up in TRINI to represent the major type classifications and the key symbols. If a key word match is found, NEWSTT is set to the symbolic constant KWORD and NEWTOK is set to the value of the pointer (address) in KWP where the match was found. If a key symbol match is found, NEWSTT is set to KSYM and NEWNUM is set to the constant representing that symbol.

The Pseudo-Code Table

The pseudo-code table (generated pseudo-code) is constructed with the help of subsystem CGSUP (code generation support) and the vector IFORM. The placement of pseudo-instructions in a FORTRAN vector represents a compromise between storage space requirements and execution (interpretation) time.

IFORM is used to determine the instruction format code from the operation code number. For debug purposes, the pseudo-code table can be output using a vector IMNEMO containing four character symbolic operation codes. These can be found in common block GENCOD (generate code).

There are three different formats for instructions:

1. operation code with no address field
2. operation code with display register and offset address (only for display storage)
3. operation code with address relative to the base of specialized addressing areas (such as constant space or pseudo-code space but not dynamic storage)

Format 1 instructions are packed four to a word, if possible. Format 2 and 3 instructions each occupy a full word and are not allowed to cross a FORTRAN word boundary.

All addresses in the pseudo-code are intended to be word addresses. Thus a no-op or null instruction code is required as a filler (all zero bits) for words containing less than four Format 1 instructions. Two examples illustrate the conditions where fillers may be used:

1. the next instruction to be executed is a Format 2 or 3 instruction and the current instruction is a Format 1 instruction at byte offset 0 to 2 of the word containing it
2. the next instruction is the target of a branch and the current instruction is a Format 1 instruction at byte offset 0 to 2 of the word containing it.

<u>The Translator Storage Block</u>

The common block DSTORP contains the translator storage block pictured in Figure 3. The lack of pointer variables in FORTRAN forced the use of a single array with subscripts as pointers in order to easily access any part of the block.

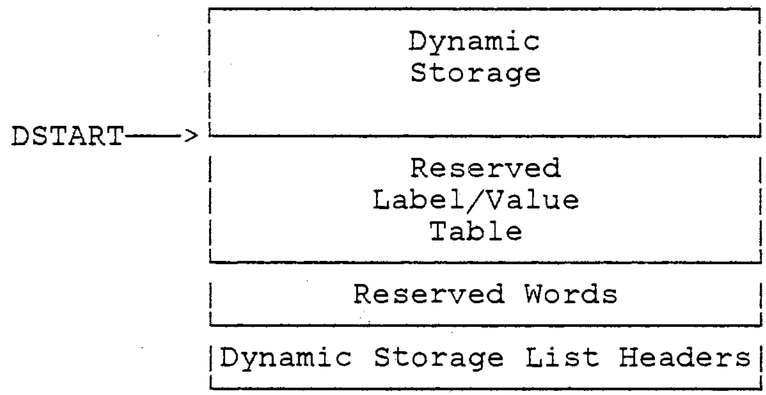


Figure 3. The Translator Storage Block

The first part of the block contains the list headers necessary for the Fibonacci sequence buddy system. The second part is statically initialized with the reserved words. The third part, containing the reserved label/value table, is initialized at translation time. The remainder of the block is available for dynamic storage with DSTART set to point to its beginning. The size of dynamic storage is dependent upon the size of the reserved label/value table.

The subsystem DYNAM (dynamic storage) is responsible for all the dynamic storage management functions.

<u>The Run File</u>

The run file is created by the translator and is made up of some number of fixed length direct access (by relative number) records as depicted in Figure 4.

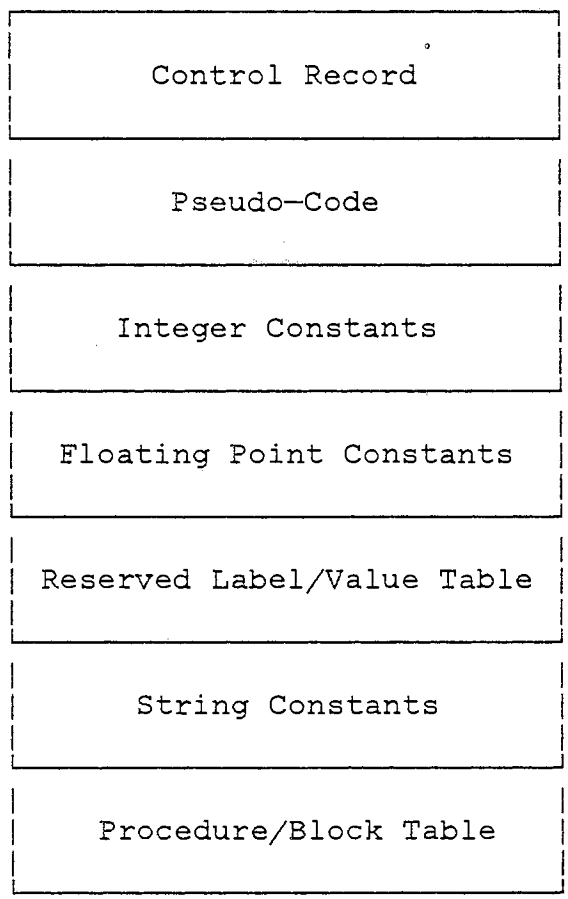


Figure 4. The Run File

The first record in the file is a control record which contains critical information about the remainder of the run file and also certain switches and values which must be used to initialize the interpreter. The remainder of the file contains the pseudo-code generated, the procedure block table and the constant tables (integer constants, floating point constants, reserved label/value strings including the ordered pointer list and string constants not in the reserved label/value list).

The translator does not complete the control record until the rest of the run file has been built. Initially, the translator fills up this record with minus ones and does not place valid information in this record unless compilation completes properly. The existence of minus ones in the control record marks the run file as not executable.

The detailed format of the control record, by word, is:

1	record length of run file records
2	number of binary tree nodes to reserve in the pseudo-machine space
3	number of dynamic storage words to reserve in the pseudo-machine space
4	trace switch
5	statistics switch
6	number of words of generated pseudo-code
7	starting record number for the pseudo-code in the run file
8	number of words for constant tables
9	starting record number in the run file for the constant tables
10	number of integer constants
11	number of floating point constants
12	number of string words
13	number of reserved label/value strings

14	number of reserved label/value string words
15	number of words in the procedure/block table
16	starting record number in the run file for the procedure/block table
17	number of entries in the procedure/block table
18	procedure/block table index of the main program
19	number of standard file units following (presently 2)
20	FORTRAN unit number for SYSIN file
21	FORTRAN unit number for SYSPRINT file

The routines handling the management functions concerning the run file are contained in subsystem RUNFIL.

Subsystem Organization

Overview

Subsystem MNCTL (main control) contains the main program. It handles initialization and wrapup functions, passing control to subsystem PRSCG (parser/code generator) when an external procedure is encountered. Subsystem PRSCG controls the translation of the external procedure. Both rely on the routines described above for table and file management, and subsystems PAGE, ERROR and CHRST (character string) to perform output page management, error handling and character string manipulations, respectively. Subsystem CHRST was necessary because FORTRAN does not adequately support character string functions.

Main Control

Subsystem MNCTL (main control) handles initializations of machine dependent parameters, error processing, the binary tree list, dynamic storage and, if necessary, the run file using subroutines in the appropriate subsystems.

There are four types of control records which may be used to specify translator run options. They are

*CREATELIB
  creates indexed library file for automatic use for unresolved procedure references
*CREATELBVL
  creates reserved label/value file from a PLANS program for later use in a code generation run
*PROCESS(option list)
  normal control record used immediately ahead of every external procedure to be compiled
*FILES(file identifier - FORTRAN unit number equivalence list)
  specifies the FORTRAN unit numbers to be associated with PLANS file names and used at execution time

and each is processed in MNCTL (*PROCESS uses subsystem OPTION to process the option list).

For each external procedure encountered, control is passed to subsystem PRSCG (parser/code generator) until translation of the procedure is completed. Upon completion, error messages and source listings are output to the printer and the run file is output to disk, if any are required. The next *PROCESS record and external procedure is processed until no more remain.

Wrapup consists of outputting the library file if the *CREATELIB record is present or the reserved label/value file from the global string constant file if the *CREATELBVL record is present. Otherwise, the library is processed for unresolved external references and the run file is completed if translation was successful.

Parsing/Code Generation

The main parsing routines form the driver for the translator.

The grammar is type LL(1). The parse is top-down and deterministic. (Iteration is used in lieu of left recursion.) Only one token look ahead is used with no back tracking. It is "hard coded" with pseudo-code (see Chapter III and Appendix I) and some error recovery directly associated with the appropriate parsing routine. The primary reason for this is efficiency and speed. (The reader should keep in mind that library routines, if used, are always retranslated. No object modules are ever created.)

All parsing rules requiring recursion are set up in subsystem PRSCG (parser/code generator). Those which do not are called from PRSCG, when needed. They consist of routines in subsystem PRSSP (parser support) which parse the declaration statements and the formal parameter list.

Because of the lack of recursion in FORTRAN, it is simulated in PRSCG using push down stacks GSTACK (general stack) for parameter passing and ASTACK (address stack) to handle the proper return address. Symbolic labels, which represent the left hand side of each rule, have been set up with a branch to the label serving as the invocation of that rule.

GSTACK also serves several other functions. The grammar does not reflect the data type requirements for expressions, so operands are placed on GSTACK for type checking at the appropriate time. If the data type is not what is required, a conversion instruction is generated, if possible, otherwise, a severe error is generated. GSTACK is also used to save control information from a DO statement until its corresponding END is found where the appropriate instructions are generated.

Since PLANS is a one pass compiler, it is necessary to deal with forward references.

Procedure calls are handled through a transfer vector (BADDRS) in the procedure/block table. If a call is parsed, and no entry for that procedure exists in the table, one is created. A linked list is formed with BADDRS pointing to the pseudo-code instruction generated for the call. If more calls are encountered before the reference can be resolved, they are joined to the linked list through their pseudo-code instruction. When the procedure name is reached, the list is traversed to resolve all the references.

An explicit forward reference (a GO TO for example) is handled in a similar manner except that its list pointer contains a temporary negative address of the pseudo-code instruction through the symbol table.

Structured entities which are nested in nature and cannot cross one another's boundaries (such as DO groups, IF-THEN-ELSE statements, DO FOR ALL SUBNODES) make use of a stack (PSASP(2,*)) and subsystem CGSUP (code generation support). At least two entries are needed. The first contains the value of the program counter when the entity is first encountered and the second is a forward reference pointer which is initialized to null. If multiple transfers to the same reference point are required (as in DO I = 1 TO 10 WHILE J = K;), a linked list is formed. When the transfer point is reached, all its references are resolved by traversing the list and the entries are popped from the stack. For RETURN statements, a different stack is used (PSASP(1,*)) since they can cross boundaries with the other types of implicit transfers. When their references are resolved at the end of the containing procedure, their transfer point is to pseudo-code which prunes all local trees.

Error Handling

The four categories of errors generated are note, warning, severe error and fatal error. The first two categories do not affect whether interpretation will take place but rather serve to inform the user that some remedial action has occurred. A severe error indicates that interpretation is no longer possible but an attempt is made to recover and continue compilation. A fatal error immediately terminates compilation with no attempt at interpretation.

Where possible, an effort is made to handle an error where it occurs and in the least disruptive manner. In the case of a missing key symbol or key noise word, a note or warning is issued when the parser discovers it missing. The grammar does not reflect data type requirements but rather the parser handles the checking and generates a severe error only if conversion is not possible. It is sometimes necessary to recover by scanning until the next DO, BEGIN, PROC, END or EOF (end of file). This is usually the case when an unexpected token appears. To continue processing, TFFLG is set to FALSE to return to the appropriate grammar rule. If table capacity is exceeded or if stack overflow or underflow occur fatal errors ensue.

CHAPTER III

THE INTERPRETER

Introduction

The purpose of the interpreter is to execute PLANS programs (pseudo-code for an idealized "PLANS machine") using the run file produced by the translator.

This chapter begins with a description of the data structures pictured in Figure 5 and their support mechanisms followed by a description of the subsystem organization.

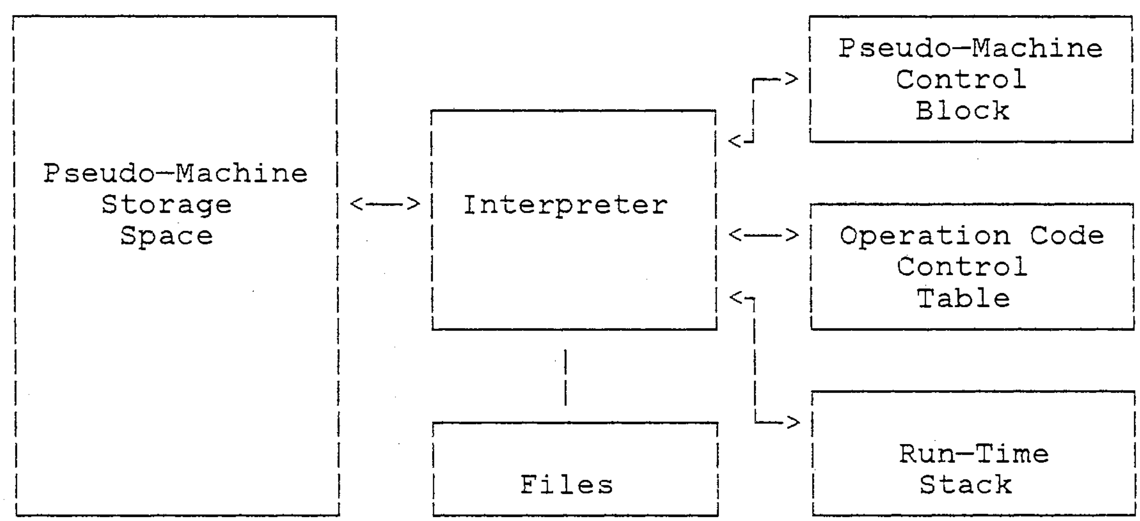


Figure 5. Run-Time Storage Organization

An important aspect of the PLANS language is the manipulation of trees at execution time. Since the tree structures can be highly dynamic, a substantial number of PLANS statements are devoted to accomplish tree functions easily. They include tree naming, structure modification, traversal, accessing, input and output statements. (For a detailed description of all the individual instructions see Appendix I [9, 13].) These functions require substantial support routines and therefore most of the execution time is spent in these routines rather than in instruction decoding. The pseudo-code generated is very compact, with primitive operations designed specifically to support PLANS source language features. This greatly obviates the time degradation normally associated with interpretive execution and contributes to space conservation due to the compactness of the specially designed pseudo-code.

Run-time Storage Organization

Pseudo-Machine Storage Space

Pseudo-machine space (in common block INGLOB), initialized from the run file file, contains the pseudo-code and data space. The organization is depicted in Figure 6.

Tree node space is determined either by default or by a user set parameter in the *PROCESS record at translation time. Available tree nodes are maintained as a linked list with the trees themselves in binary tree form [4]. Tree nodes are represented by two words as shown in Figure 7.

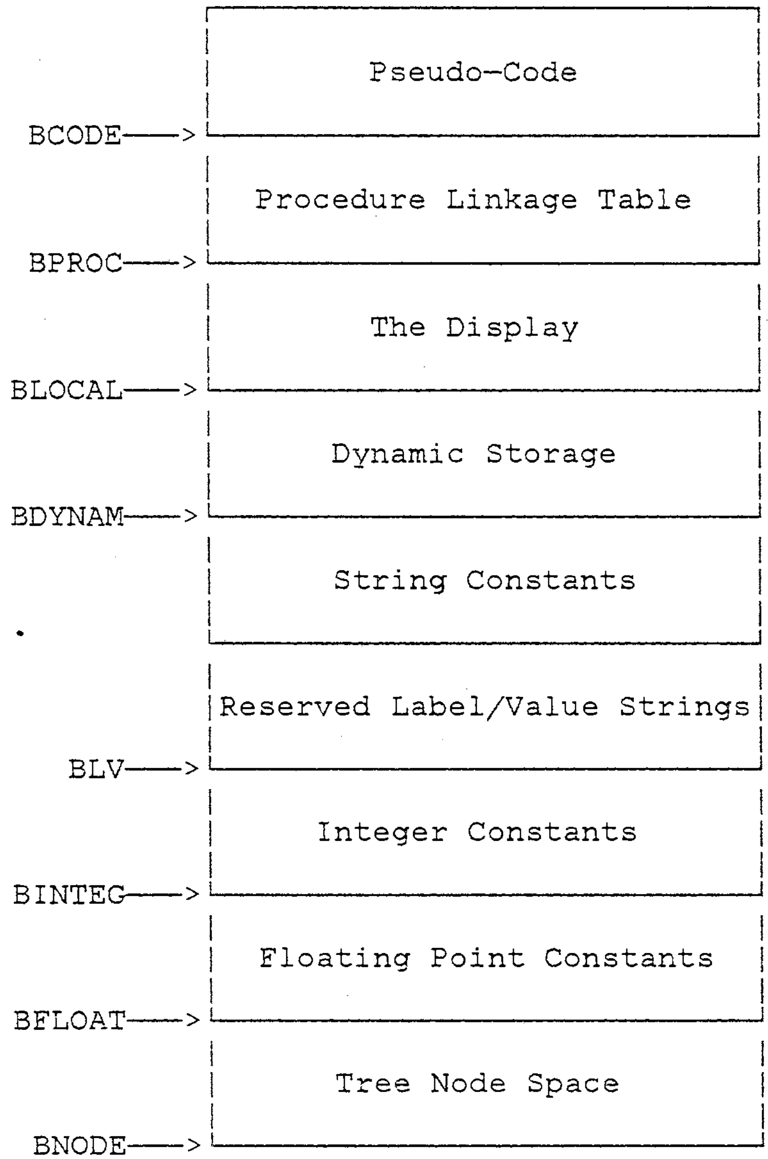


Figure 6. Pseudo–Machine Storage

Figure 7. PLANS Tree Node Representation

Field 1 is two bits and contains the word 2 type code. Field 2 is fifteen bits and contains a label pointer. Field 3 is fifteen bits and contains the sibling pointer, if a sibling exists. Binary tree node manipulation is handled in subsystem NODE.

Arithmetic constants (integer or floating point), the reserved label/value strings and the string constants come from the run file output by the translator and entered in pseudo-machine space at interpreter initialization time. These spaces are never altered during interpretation.

Dynamic storage size is determined either by default or by a user set parameter in the *PROCESS record at translation time and is used for strings not already in string constant or reserved label/value space. Subsystem INDYNAM (interpreter dynamic storage) is responsible for dynamic storage management. The method used is the generalized Fibonacci-based buddy system discussed in Chapter II [1].

When a tree is read in during execution, for each string, a search is made first of the reserved label/value table and then, if necessary, the string constant table until a match is found. (Subsystem BINSR (binary search) handles the search.) If no match is found, the string is then entered in dynamic storage.

Constants in string constant or reserved label/value space require only that their pointers be copied on assignment. For strings in dynamic storage, however, a copy of the string is made and the pointer to the new string is used on assignment. This is necessary because on exit from a procedure, all local trees and strings are pruned. Space from all these pruned strings is released for reuse in the process. Subsystem PREOR (preorder traversal) handles the traversal of the tree for pruning, comparing, copying and output.

Display size consists of the remaining unused space in the run file [10]. Upon entry to a procedure or begin block, the display is loaded with the following information:

Fixed Part:

1. the block table pointer
2. block level words for display entries
3. a pointer to the previously active display
4. a pointer to the display stack top upon entry
5. the return address (only for procedure blocks)
6. parameter addresses (only for procedure blocks)
7. local storage for scalar variables, array descriptors and local tree root node pointers

Variable Part:

1. space for parameter values passed, if any
2. array space

The fixed part is determined at translation time and the required size is entered in the procedure/block table then. All tree and string addresses are zeroed out upon entry so no accidental pruning will take place upon exit of a block.

The variable part is determined at run-time. Parameters are passed by reference but in the case of expressions or constants, the values are determined at procedure entry and placed in the variable part of the display. (Note that procedure names cannot be passed as parameters.) The remainder of the display is available during interpretation for temporary values and pointers. (For a more detailed description of displays refer to [2].)

Arrays are stored in the display using an array descriptor in the fixed part for the block which consists of

```
WORD 1 - virtual origin of the array
WORD 2 - number of dimensions
WORD 3 - first dimension bound
WORD 4 - mapping multiplier for first dimension
   .
   .
   .
   .    - last dimension bound
   .    - mapping multiplier for last dimension
```

Dynamic array allocation requires that the dimension bounds and mapping multipliers be filled in on block entry. It is then possible to set aside the required space for the entire array in the variable part of the display with WORD 1 of the array descriptor pointing to the first array element.

For integer and floating point variables, their values are entered directly in the display upon assignment. For string variables and constants, the display entry contains a pointer to the string in string constant space, reserved label/value space or dynamic storage (see Chapter IV).

The procedure linkage table and the pseudo-code come from the run file generated by the translator and are also never altered during interpretation.

<u>The Psuedo-Machine Space Control Block</u>

The pseudo-machine space control block (in common block BOUNDS) contains the following [8]:

1. absolute address of currently active display — ACTIVE

2. instruction pointer (two values)
word — IPWORD
byte — IPBYTE

3. reserved label/value and string constant base address — BLV

4. floating point constant base address — BFLOAT

5. integer constant base address — BINTEG

6. tree node storage base address — BNODE

7. dynamic storage base address — BDYNAM

8. display storage base address — BLOCAL

9. procedure linkage and environmental control table base address — PROC

10. pseudo-code base address — BCODE

11. null address — NULLAD

All but NULLAD in the pseudo-machine space control block are initialized dynamically on the basis of control information placed in the run file by the translator. The base addresses BVL, BFLOAT, BINTEG, BCODE and BDYNAM correspond with the base address values needed for the various Format 3 instructions described below. These are equivalenced in the array BASE to easily access their values.

Due to the packing of Format 1 instructions, the instruction pointer must consist of both a word pointer and a byte offset. The valid values for the byte offset are 0 to 3 with the zero byte on the left and the third byte on the right. Whenever a filler is encountered prior to the end of a word (byte offset 3) it may be assumed that any remaining bytes of the pseudo-instruction word represent fillers as well.

The Operation Code Control Table

The operation code control table (in common block INCNTL) is referenced by using the integer value of the operation code as a subscript to the table [8]. Each op code in the table has two entries, one in the instruction format array IFORM, and one in the execution address array EXADDR. These arrays are statically initialized in the block data subsystem INDAT (interpreter data). The IFORM array is initialized with eight possible values which prescribe operand address computation:

0 – undefined (system error condition)

1 – Format 1 instruction
no address computation

2 – Format 2 instruction
the value of the referenced display register in the currently active display plus the base address for display storage plus the offset address from the instruction

3 – Format 3 instruction
the base address for reserved label/value strings and string constant space plus the instruction address field

4 – Format 3 instruction
the base address for floating point constants plus the instruction address field

5 – Format 3 instruction
the base address for integer constants plus the instruction address field

6 – Format 3 instruction
the base address for pseudo–code instructions plus the instruction address field (for jump instructions)

7 – Format 3 instruction
a dummy base address of zero plus the instruction address field

(For an explanation of Format 1, 2 and 3 instructions see the Chapter II.)

The Run–Time Stack

The run–time stack (STACK in common block INSTAC) is used to hold integer or floating point values or addresses (FORTRAN integers) needed during the interpretation of a PLANS statement [11]. It should be noted that one level of indirection is always used in tree addressing (that is, the address is always of the node which points to the node desired). This is done to enable proper pointer modification for insertions and deletions in a tree.

Each address has a corresponding entry in the vectors TYPE and SUBTYP coded as follows:

TYPE
SUBTYP

1 – tree address
1 – absolute pseudo–machine address of word containing an address to a tree node
2 – absolute address of tree node

3 – $ELEMENT reference
4 – $NULL reference
5 – $COMBINATION reference
6 – $PERMUTATION reference

2 – string address
1 – reserved label/value string
2 – string constant
3 – dynamic storage string
4 – temporary string

3 – numeric variable absolute address (display only)
1 – integer variable
2 – floating point variable
3 – integer array descriptor
4 – floating point array descriptor
5 – string array descriptor

4 – logical value (0 or 1)

5 – numeric value
1 – integer
2 – floating point

6 – keyword subscript operation (no value on STACK)
1 – FIRST
2 – LAST
3 – NEXT

For tree addresses (TYPE 1), only subtypes 1 or 4 are valid outside a qualification context. For subtype 1, the right-most 15 bits only are extracted or replaced. Subtype 2 does not occur except as an intermediate value in hard or soft qualification or as the result of a SNIP operation in preparation for grafting or graft insertion. Types 3 and 4 may not occur in stack entries used for hard or soft qualification.

The stack barrier stack (BSTAC) is used to facilitate multiple level tree qualification, multiple subtree treatment (ALL:) and array subscripting on multiple dimensions. Entries in this stack point to STACK. This scheme is required because of nested constructions such as nested tree qualification due to indirect reference (#, #LABEL) and/or arithmetic expressions for subscripts.

When a qualification instruction HQAL (hard qualification) or SQAL (soft qualification) is interpreted the top of BSTAC points to the place in STACK where level qualification commences. Qualification continues to the top of the stack. For example, a four level qualification $A.TELESCOPE(FIRST)(2) will be preceeded by loading

1. address of $A pointer on the stack
2. address of TELESCOPE string on the stack
3. key word FIRST operation code
4. numeric value 2

The top of the barrier stack will point to the entry for 1.

Key word subscript operation codes (TYPE 6) are placed on the stack because multiple level qualification operations are done with one qualification instruction rather than one for each level (see the above example). This is rather important in terms of pseudo–code space requirements and pseudo–code interpretation time.

Conditional qualification (ALL: and FIRST:) is effected by first qualifying to the level at which the conditional is to be applied and then applying the conditional using $ELEMENT reference appropriately. ALL: and FIRST: do not have delayed key word subscript operation codes. These two operations do not need to be stacked.

Subsystem Organization

Overview

The interpreter's main program is in subsystem INTCNTL (interpreter control) which, like the translator's main program, performs the initialization and wrapup functions, passing control to subsystem INTFE (interpreter fetch/execute) to drive the interpreter. Both rely on the routines described above for data and storage structure management and subsystems CHRST (character string), INERROR (interpreter error), CONVRT (convert) and ENCODE to perform character string manipulation, error handling and conversion to the required data type. (Only subsystem CHRST is also used by the translator.)

Subsystem INTFE handles instruction fetching, decoding and executing. The more detailed operation codes are handled in separate subroutines called from INTFE. These are found in subsystems ARRAY, CMBPRM (combination or permutation), CVICVF (convert to integer or convert to floating point), GETPUT, ORDER, READ, TREQUAL (hard or soft qualification), TRESUP (tree support) and WRITE. They are generally invoked directly and are organized by localized call relationships. The subroutines in TRESUP, however, may be called from other subroutines as well to support other operations.

The Fetch/Execute Cycle

The logic for the control of the interpreter fetch/execute cycle is heavily dependent on the information described above for the operation code control table and pseudo-machine space control table [8]. The run file must be processed ("loaded") before the fetch/execute control loop is ever entered. The PDL found in Figure 8 describes the logic for the control loop. SPACE refers to pseudo-code space, OPCODE to the operation code and OPADDR to the absolute operand address. ACTIVE, IPWORD, IPBYTE and BCODE are contained in the pseudo-machine space control block while IFORM, EXADDR, and BASE are contained in the operation code control table and all are described in the two corresponding sections above.

```
Zero out IPBYTE.
Initialize IPWORD from MAIN program address field in
  procedure table plus BCODE.
DO until exit from main program
     FETCH:
          Extract OPCODE from address specified by
            IPWORD and IPBYTE.
          IF system trace is on THEN
               Display current instruction word,
                 instruction address and symbolic
                 operation code.
          ENDIF
          Extract instruction form - FORM - from
            IFORM(OPCODE).
          IF FORM is 1 THEN
               Update instruction pointer to next
                 non-null operation code.
               Transfer to EXECUTE_INSTR.
          ELSE IF FORM is 2 THEN
               Assign 4-bit display register field from
                 current full word instruction to
                 DISREG.
               Assign display offset field from current
                 full word instruction to OFFSET.
               Assign OFFSET + SPACE(ACTIVE + DISREG)
                 to OPADDR.
               Update instruction pointer to next word.
               Transfer to EXECUTE_INSTR.
          ELSE /* Format 3 instruction */
               Assign operand address field from
                 current full word instruction to
                 "address".
               Assign "address" + BASE(FORM) to OPADDR.
               Update instruction pointer to next word.
          ENDIF
     EXECUTE_INSTR:
          Execute instruction interpretation code
            specified by EXADDR(OPCODE).
END.
Invoke execution summary.
```

Figure 8. PDL for Fetch/Execute Control Loop

CHAPTER IV

THE CHARACTER STRING EXTENSION

Introduction

A major objective of this thesis is the extension of the PLANS language to support conventional character string manipulation. The motivation for this is the adaptation of PLANS to an interactive data structure manipulation environment.

In incorporating character string variables, two considerations were of highest priority. One was to attempt to fit variable length strings into the system as consistently as possible with all presently allowable variables using as much as possible the mechanisms already set up. The other was to make relatively few structural and logical changes to the existing translator. The modifications required were made to the translator while the interpreter changes have been designed but not implemented.

This chapter is divided into four main sections. The first describes the existing character string features. The second describes the changes affecting the user including a discussion of the use of the new statement forms added and the error handling resulting from improper construction. The remaining sections describe the internal modifications.

The translator section covers explicit data type handling, the new string operation codes generated, the changes necessary to the lexical analyzer and the grammar changes and their effect on the parser. The final section covers the interpreter changes. Several different approaches were considered concerning the type of strings to add and where to locate them during execution. These are discussed along with the final decision and the modifications it necessitated. This is followed by an explanation of the instructions added.

Existing Character String Features

Prior to the changes outlined below, character string usage was rather restricted. PLANS tree nodes may have character string labels and character string values but the notion of character string variables has not been present and operations on character strings other than assignment to tree nodes, input/output and comparisons, have not been supported.

The representation of character strings has been in the form of a variable number of four byte words, the first of which contains the byte count for the string length.

Character strings may be stored in any one of three pseudo-machine storage areas: the reserved label/value table, the string constant table or dynamic storage. References to such strings are by pointer (FORTRAN subscript). Character strings in the first two storage areas are never copied or deleted. Assignment amounts to copying a pointer. Character strings in dynamic storage are copied on assignment and may be deleted by virtue of "pruning" a (sub)tree, replacing such a value in a tree node or label or by leaving a procedure or begin block in which case all local trees are deleted and any tree nodes and dynamic storage units are reclaimed.

Changes Affecting the User

Explicit Data Types

PLANS was originally defined to allow only standard FORTRAN default to determine data type. Variable names beginning with any letter from I to N inclusive were automatically of integer type while names beginning with any other letter defaulted to floating point. (This does not apply to tree nodes as is explained in Chapter II.) The declaration statement served only for arrays and tree names.

Explicit type declaration statements were added which not only allow for string variables but also they serve to override the standard default. They are of the form

```
'declare' ['integer' | 'float' | 'string']1
          variable_list ['local']1 ';'
```

Examples can be found in Figure 9. All rules true for the original declaration statement apply for the new forms except that tree names can appear in the original form only since explicit type declarations do not apply.

```
DECLARE INTEGER X, Y, Z(10) LOCAL;

DECLARE FLOAT I, J, K;

DECLARE STRING S1, S2, S3, S(5);

S1 = S2 || S3;                       /* concatenation */

S1 = S2(I2 : I3);                    /* substring     */

I1 = LENGTH(S2);                     /* length        */

I1 = INDEX(S2 , S3);                 /* index         */

I1 = VERIFY(S2 , S3);                /* verify        */

S(LENGTH(I)) = X || S3(INDEX(S2 , S3) :
     S(VERIFY('A' , Y)));
```

Figure 9. A Sample of Valid PLANS Statements

String Functions

Previously, string constants could only be assigned to tree node labels or values so several string operations were added. They consist of assignment to a string variable, concatenation, substring, length, index and verify. An example of each can be found in Figure 9. The concatenation, index, length and verify functions are patterned after the corresponding PL/I functions while substring is patterned after the corresponding FORTRAN 77 function. A string variable is expected where S1 appears in Figure 9, a string variable or constant where S2 and S3 appear, an integer variable where I1 appears and an integer variable or constant where I2 and I3 appear. If any are of a different data type, automatic type conversion takes place.

The above functions can also be used within expressions and have been set up using PL/I priority rules with concatenation taking priority over the relational operators and with the remaining four functions at the primary level (see Appendix C). In part, this choice is based on the fact that much of the syntax of PLANS is patterned after PL/I.

Error Handling

Severe error messages are generated if one of the new keywords INTEGER, FLOAT or STRING is used as a variable (a potential problem for existing PLANS programs) and if any function form is structurally incorrect. If a tree name appears in an explicit type declaration, it is ignored and a warning is generated. Where applicable, if a comma, colon or closing parenthesis is missing, one is generated and a warning message is issued.

Translator Changes

Explicit Data Types

The explicit data type addition required that two types of changes be made. One was that DFTID (define tree or identifier symbol) in subsystem BLKTB (block table) had to be modified to enable it to properly parse the new types of declaration statements. The other was that several symbol table routines had to be modified in order to allow a variable type of 'S' for string in the table. To accomplish the latter, a parameter was added to DFTID because it enters the type in the symbol table. If the call comes from RFTID (reference to an identifier) or FPARM (formal parameter) then the parameter contains a 'U' for undefined. If the call comes from DCLST (declare statement) then the parameter contains an 'I', 'F' or 'S' for a variable which appears in an explicit type declaration, a 'T' for a tree declaration or a 'U' to force DFTID to find the default.

Lexical Analyzer Changes

The colon previously was not treated as a single character token but rather an identifier followed immediately by a colon was considered a single label unit. Since the colon was needed for the substring function, it was changed to a single character token. To avoid significant modification to existing logic, when a label unit is found, if it is not a valid label context, only then is it parsed as two separate units.

The or symbol ('|') was changed from a single character token to one which could possibly be a double character token as well ('||') for concatenation.

New Operation Codes

A number of new operation codes were needed for the extension and their addition required changes not only in the routines that generate them but also in the parameter statements (FORTRAN 77), equivalence statements and data statements which occur in GENCOD (generate code), TOKDAT (token data) and TRINI (translator initialization), respectively. GENCOD was modified to set the numerical equivalents for the added operation codes, TOKDAT was modified so that new key symbols and words could be referred to by symbolic name instead of number and TRINI was modified to add new key symbol and key word initializations to the respective tables.

String constants and tree node values are stored in the reserved label/value table, string constant table or dynamic storage at interpretation time. The first byte contains the string length which is followed by the characters which make up the string. This easily extends to the storage of variable length strings in dynamic storage with a pointer or, in the case of an array, a set of pointers in the display. New operation codes were added to load a string array descriptor and string variable address both directly and indirectly. These correspond with those for integer and floating point variables except that there is no code to load a string value directly since its value is never stored in the display. They are generated in CRFID (compile a reference to an identifier) and DCLST (declaration list).

To avoid altering the existing routines for relational expressions which automatically convert strings to floating point, separate operation codes were added which are used only if both operands are of string type. These directly correspond to the regular relational expressions.

The remaining set of operation codes were included to handle the assignment to a string variable, concatenation, index, length, substring and verify functions added.

Parser Changes

It was necessary to add several new grammar rules to parse properly the functions added to subsystem PRSCG (parser/code generator) but only one was altered (see Figure 10 and Appendix C).

New Rules:

```
string_expr := arith_expr [cat_op arith_expr]

primary := string_function

string_function := expression '(' expression ':'
     expression ')'
     | 'length' '(' expression ')'
     | ('index' | 'verify') '(' expression ','
     expression ')'
```

Altered Rule:

```
relational_expr := string_expr [relational_op
     string_expr]1
```

Figure 10. New Grammar Rules

STEXPR (string expression) is invoked in REXPR (relational expression) so that the concatenation operator will have priority over the relational operators but not over the arithmetic operators. STFN (string function) is invoked in PRIMRY (primary token) if one of the key words LENGTH, INDEX or VERIFY is encountered or if the context suggests that the substring function is possible. (See Chapter III for an explanation of invocation.) Two separate sections of code were added to handle these new functions. (See Appendix E for a detailed explanation of the changes and the pseudo-code operations generated and Appendix J for the PDL's describing the routines added for the grammar rules "string_expr" and "string_function".)

Interpreter Changes

String Handling

Decisions had to be made concerning the type of variable strings to use and where they were to be stored. The two possible types considered were fixed length strings (like PL/I) or variable length strings. Fixed length strings could be stored in the display since their maximum size was known. It would be necessary to have a two word field containing the maximum and actual lengths. Storage management would consist of automatical release on exit from a block but there would be the distinct possibility of wasted space. True variable length strings could solve the problem of unusable space and a word for maximum length would not be needed. But their storage in the display would be complicated by the fact that their size would not be known in advance. If maintained in dynamic storage, local string variables, including arrays, would have to be "pruned" upon exit from a block.

The true variable length string approach was chosen for several reasons. It would be much less restrictive for the user since the maximum size would not have to be known in advance. It easily fit into the existing program structure. There already was a dynamic storage management package which could be used to allocate and free space. Also, it was more consistent with the approach already taken in terms of space conservation, string constants and tree node management. String constants were already being kept in dynamic storage and tree nodes required pruning upon exit from a block.

String variables will be set up in dynamic storage using the first word for their length followed immediately by the string. Access will be through a pointer in the display. Arrays will contain an array descriptor which points to a set of pointers in the display. It is anticipated that dynamic storage allocation will be handled using subsystem DYNAM (dynamic storage). Temporary strings will be deallocated when they are no longer needed in the routines dealing with the instructions CVI (convert to integer), CVF (convert to floating point), SCAT (string concatenation), SSUB (string substring), SLEN (string length), SIND (string index) and SVER (string verify). The local strings will be pruned by traversing the display upon exit from a block. To include string variables, the run-time stack will be expanded. A subtype of 4 will be added to type 2 to denote a temporary string and a subtype of 5 will be added to type 3 to denote a string array descriptor. (See Chapter III for an explanation of types and subtypes.)

New Instructions

All the new instructions added are Format 1 instructions except for the load instructions which are Format 2. Format 1 instructions contain the operation code with no address field while Format 2 instructions contain the operation code with display register and address offset. (See Chapter III for an explanation of Format types.)

The string relational operators pop the top two items from the stack, replacing them with the Boolean result. AS (assign string) pops the top two entries from the stack assigning the pointer from the first entry to the display address of the second entry, modifying the subtype if 4 for temporary. CVS (convert to string), SCAT (string concatenation) and SSUB (substring) each pop one, two or three items from the stack, respectively, replacing them with a pointer to dynamic storage where the temporary string was created. SLEN (string length) pops one entry and SIND (string index) and SVER (string verify) each pop two entries from the stack. The integer result is then placed on the stack. When a load operation code is encountered, no entries are removed from the stack but each causes an address entry to be added to the stack.

CHAPTER V

CONCLUSIONS

This study has dealt with a software system design for the Programming Language for Allocation and Network Scheduling (PLANS) and character string extensions to that language. It described, in detail, the translator and interpreter as originally defined, including their data, file and storage structures and subsystem organization. Major emphasis was placed on the data structures because of their complexity and the the logical processing was heavily dependent upon them. The modifications required for the addition of character string variables and functions were also described. Those changes were implemented for the translator and designed for the interpreter.

Implementation Completion

Modifications to the interpreter were left incomplete because of the productivity limitations of the development environment available to the author. It is necessary to complete these changes as outlined in Appendix J so that an interactive environment can be implemented.

The PLANS system software is divided into three major sets of modules. The COMMON module set contains all the common blocks in more than 25 units. The PRSCG module set contains the recursive parsing rule routines in more than 50 units. The SOURCE module set, which contains all the remaining subsystems, contains over 35 subsystems which, together, contain over 130 subroutines. An environment which can assist, rather than hinder, the management of a large number of subsystems, routines and common blocks in an orderly fashion is invaluable in terms of productivity. This ideally requires a multiple file structure in the form of a tree which naturally follows the hierarchical nature of not only the source code modules but also the object code produced. The only effective multiple file structure available was the partitioned file feature. The subroutines belonging to a subsystem all reside in one file and therefore it is necessary to re-compile all in the subsystem if a change is made to one.

This problem is further complicated by the fact that there is insufficient automatic space management within partitioned files. A fixed size must be set up in advance and freed space within the file is not reallocated. An environment which supports automatic file space management would free the development programmer from these tasks.

Furthermore, it was necessary to devise an independent common block inclusion processor because the vendor supplied inclusion processor for FORTRAN 77 functions incorrectly.

Operating systems which do support both multi-level tree structured file management and sufficient automatic file space management include VMS, UNIX, MULTICS and PRIMOS. (The development work at Science Applications, Inc. directed by the author's adviser was, in fact, performed in such an environment.)

Possible Further Enhancements

Several further enhancements to the language may be advantageous.

Internally, PLANS trees use a one way linking mechanism for connecting siblings. A search must be performed via a linear linked list to locate the desired sibling for some types of qualification (for example, LAST). It is possible to limit the performance penalties which can accompany such a search over a large number of siblings by careful use of tree pointers but the programming required tends to be rather obscure because of the indirection required. It would be desirable to represent trees internally using a double linking mechanism. This would involve modifying only the interpreter since the interpreter is independent of the translator in the sense of the PLANS tree structure representation.

One of the author's committee members, Dr. D. D. Fisher, suggested an alternative for managing the release of tree space and the dynamic storage associated with the tree node labels and values. These trees could be maintained as a binary tree of available nodes which could then be traversed with dynamic storage space and tree nodes released when either runs out of available space. Time would be saved since it may not be necessary to reuse the space upon block exit. No additional space would be used since both are fixed at interpretation time.

In the context of data structure driven interactive processes, it is highly desirable to have a notion of a vector or array of pointers. At present, scalar tree pointers only are implemented.

There is an extensive variety of tree operations in the PLANS language but the entire tree must reside in memory. Only sequential input of trees is presently implemented. To extend the PLANS language to allow for the manipulation of large data bases in the form of PLANS trees, some form of indexed input/output is necessary. It would be desirable to have some indexing mechanism in which the index key corresponds with tree qualification (for example, $T.R.E) to get just a subtree into memory.

The PLANS language was not intended for extensive numerical computation and so these capabilities are limited. It might be desirable to provide an interfacing mechanism to call FORTRAN subroutines in order to take advantage of the computational capabilities which FORTRAN provides.

A SELECTED BIBLIOGRAPHY

(1) Dasananda, Surapol. "Fibonacci-Based Buddy Systems." (Unpub. M.S. Report, Oklahoma State University, 1974).

(2) Gries, David. _Compiler Construction for Digital Computers_, John Wiley & Sons, Inc., New York, New York, 1971.

(3) Hinds, James A. "An Algorithm for Location Adjacent Storage Blocks in the Buddy System." _Comm. Acm._, 18, 4 (April,1975), 221-222.

(4) Knuth, D. E. _The Art of Computer Programming_, _Vol 1_: _Fundamental Algorithms_, Addison Wesley Publ. Co., Reading, Mass., 1973.

(5) Knuth, D. E. _The Art of Computer Programming_, _Vol 3_: _Sorting and Searching_, Addison Wesley Publ. Co., Reading, Mass., 1973.

(6) Ramsey, Rudy H., Willoughby, John K. and Kullas, Daniel A. _A User's Guide to the Programming Language for Allocation and Network Scheduling_ (_PLANS_), Technical Report SAI-77-068-DEN, Science Applications, Inc., Englewood, Colorado, 1977.

(7) Van Doren, James R. "Format of the Runfile." (Unpub. PLANS system design notes, Science Applications, Inc., Englewood, Colorado, 1980-81).

(8) Van Doren, James R. "Interpreter Design Information." (Unpub. PLANS system design notes, Science Applications, Inc., Englewood, Colorado, 1980-81).

(9) Van Doren, James R. "PLANS Pseudo Machine Instruction Set." (Unpub. PLANS system design notes, Science Applications, Inc., Englewood, Colorado, 1980-81).

(10) Van Doren, James R. "Run Templates." (Unpub. PLANS system design notes, Science Applications, Inc., Englewood, Colorado, 1980-81).

(11) Van Doren, James R. "PLANS Run-Time Stack." (Unpub.

PLANS system design notes, Science Applications, Inc., Englewood, Colorado, 1980-81).

(12) Van Doren, James R. "Table Structures." (Unpub. PLANS system design notes, Science Applications, Inc., Englewood, Colorado, 1980-81).

(13) Van Doren, James R. "Tree Addressing." (Unpub. PLANS system design notes, Science Applications, Inc., Englewood, Colorado, 1980-81).

APPENDIX A

INDEX TO THE TRANSLATOR SUBSYSTEM

NAME	ACCESS MECHANISM	APPENDIX
AASTMT	PRSCG.FORT(*)	C-74, E-101
ADDCH	SOURCE.FORT(CHRST)	D-85
ADVST	PRSCG.FORT(*)	C-74
AEXPR	PRSCG.FORT(*)	C-74
BGBLK	PRSCG.FORT(*)	C-74
BINSR	SOURCE.FORT(SRCH)	D-97
BLKBDY	PRSCG.FORT(*)	C-74
BLKEXT	SOURCE.FORT(BLKTB)	D-79
BLKTB	SOURCE.FORT(*)	D-79, E-102
BSTSR	SOURCE.FORT(SRCH)	D-97
BTERM	PRSCG.FORT(*)	C-74
BTINT	SOURCE.FORT(SRCH)	D-98
CALLST	PRSCG.FORT(*)	C-74
CGSUP	SOURCE.FORT(*)	D-82, E-103
CHKAC	SOURCE.FORT(CGSUP)	D-82
CHRST	SOURCE.FORT(*)	D-85
CLRTAB	SOURCE.FORT(BLKTB)	D-79
CMBPRM	PRSCG.FORT(*)	C-74
CODLST	SOURCE.FORT(*)	D-87
COMBCL	PRSCG.FORT(*)	C-74
COMMON	*.FORT	B-70, E-101
CPYST	SOURCE.FORT(CHRST)	D-85
CRFID	SOURCE.FORT(CGSUP)	D-83, E-103
CSTMNT	PRSCG.FORT(*)	C-74
DCLST	SOURCE.FORT(PRSSP)	D-93, E-104
DEFST	PRSCG.FORT(*)	C-74

NAME	ACCESS MECHANISM	APPENDIX
DFBGN	SOURCE.FORT(BLKTB)	D-79
DFLAB	SOURCE.FORT(BLKTB)	D-79
DFPRC	SOURCE.FORT(BLKTB)	D-80
DFPSAD	SOURCE.FORT(CGSUP)	D-83
DFTID	SOURCE.FORT(BLKTB)	D-80, E-102
DFTREE	SOURCE.FORT(BLKTB)	D-80
DOBDEN	PRSCG.FORT(*)	C-74
DOGRP	PRSCG.FORT(*)	C-74
DSALC	SOURCE.FORT(DYNAM)	D-87
DSINFO	COMMON.FORT(*)	B-70
DSINT	SOURCE.FORT(DYNAM)	D-87
DSRLS	SOURCE.FORT(DYNAM)	D-88
DSTORP	COMMON.FORT(*)	B-70
DYNAM	SOURCE.FORT(*)	D-87
EDIT	COMMON.FORT(*)	B-70
EEXPR	PRSCG.FORT(*)	C-74
ERINFO	COMMON.FORT(*)	B-70
ERNUM	SOURCE.FORT(ERROR)	D-88
ERRCV	SOURCE.FORT(PRSSP)	D-94
ERRINT	SOURCE.FORT(ERROR)	D-89, E-103
ERROR	SOURCE.FORT(*)	D-88, E-103
ERRWRP	SOURCE.FORT(ERROR)	D-89, E-103
EXPR	PRSCG.FORT(*)	C-74
EXTPRC	PRSCG.FORT(*)	C-74
FILES	COMMON.FORT(*)	B-70
FILOPT	PRSCG.FORT(*)	C-74

NAME	ACCESS MECHANISM	APPENDIX
FILOPT	SOURCE.FORT(OPTION)	D-92
FINLIB	SOURCE.FORT(LIB)	D-91
FPARM	SOURCE.FORT(PRSSP)	D-94, E-104
FRDUMP	SOURCE.FORT(CODLST)	D-87
GENCOD	COMMON.FORT(*)	B-70, E-101
GENPOP	SOURCE.FORT(PRSSP)	D-94
GENSTK	SOURCE.FORT(PRSSP)	D-94
GENSWP	SOURCE.FORT(PRSSP)	D-95
GETST	PRSCG.FORT(*)	C-74
GNPRUN	SOURCE.FORT(BLKTB)	D-81
GRFTST	PRSCG.FORT(*)	C-74
GSTACK	COMMON.FORT(*)	B-70
HARD	PRSCG.FORT(*)	C-75
HRDSUB	PRSCG.FORT(*)	C-75
IEQST	SOURCE.FORT(CHRST)	D-85
IEXCL	SOURCE.FORT(CHRST)	D-86
IEXCR	SOURCE.FORT(CHRST)	D-86
IGNORE	SOURCE.FORT(PRSSP)	D-95
INCDO	PRSCG.FORT(*)	C-75
INIT	PRSCG.FORT(*)	C-73, E-101
INSRT	PRSCG.FORT(*)	C-75
IODEVS	COMMON.FORT(*)	B-70
LABSTR	PRSCG.FORT(*)	C-75
LABVAL	SOURCE.FORT(*)	D-89
LASTMT	PRSCG.FORT(*)	C-75
LBINFO	COMMON.FORT(*)	B-70

NAME	ACCESS MECHANISM	APPENDIX
LBLQ	PRSCG.FORT(*)	C-75
LBVLI	SOURCE.FORT(LABVAL)	D-89
LEX	SOURCE.FORT(*)	D-90, E-103
LIB	SOURCE.FORT(*)	D-90
LIBINT	SOURCE.FORT(LIB)	D-91
LIBOUT	SOURCE.FORT(LIB)	D-91
LINCNT	SOURCE.FORT(PAGE)	D-93
LVOUT	SOURCE.FORT(LABVAL)	D-89
LXCAL	SOURCE.FORT(LEX)	D-90, E-103
MEXPR	PRSCG.FORT(*)	C-75
MNCTL	SOURCE.FORT(*)	D-92
NENDU	PRSCG.FORT(*)	C-75
NUPAG	SOURCE.FORT(PAGE)	D-93
OPTION	SOURCE.FORT(*)	D-92
OPTION	SOURCE.FORT(OPTION)	D-92
ORDRST	PRSCG.FORT(*)	C-75
OUTAB	SOURCE.FORT(RUNFIL)	D-96
OUTCOD	SOURCE.FORT(RUNFIL)	D-96
OUTIWA	SOURCE.FORT(CGSUP)	D-83
OUTOP	SOURCE.FORT(CGSUP)	D-84
OUTWRD	SOURCE.FORT(CGSUP)	D-84
PAGE	SOURCE.FORT(*)	D-93
PARSE	COMMON.FORT(*)	B-70
PGINFO	COMMON.FORT(*)	B-70
PRCLIB	SOURCE.FORT(LIB)	D-91
PRIMRY	PRSCG.FORT(*)	C-75, E-101

NAME	ACCESS MECHANISM	APPENDIX
PROCBL	PRSCG.FORT(*)	C-75
PROGU	PRSCG.FORT(*)	C-75
PRSCG	*.FORT	C-73, E-101
PRSCGF	PRSCG.FORT(*)	C-73
PRSSP	SOURCE.FORT(*)	D-93, E-104
PRUNST	PRSCG.FORT(*)	C-75
PSAPOP	SOURCE.FORT(CGSUP)	D-84
PSTLST	SOURCE.FORT(CODLST)	D-87
PUTST	PRSCG.FORT(*)	C-75
READST	PRSCG.FORT(*)	C-75
REXPR	PRSCG.FORT(*)	C-76, E-101
RFLAB	SOURCE.FORT(BLKTB)	D-81
RFPRC	SOURCE.FORT(BLKTB)	D-81
RFTID	SOURCE.FORT(BLKTB)	D-82, E-103
RFTREE	SOURCE.FORT(BLKTB)	D-82
RNINT	SOURCE.FORT(RUNFIL)	D-96
RPCHL	SOURCE.FORT(CHRST)	D-86
RPCHR	SOURCE.FORT(CHRST)	D-86
RPSADS	SOURCE.FORT(CGSUP)	D-85
RRETRN	PRSCG.FORT(*)	C-73
RUNBUF	COMMON.FORT(*)	B-70
RUNFIL	SOURCE.FORT(*)	D-95
SBLKT	SOURCE.FORT(BLKTB)	D-82
SBNODE	PRSCG.FORT(*)	C-76
SEMI	SOURCE.FORT(PRSSP)	D-95
SFTSUB	PRSCG.FORT(*)	C-76

NAME	ACCESS MECHANISM	APPENDIX
SOFT	PRSCG.FORT(*)	C-76
SOURCE	*.FORT	D-79, E-102
SRCH	SOURCE.FORT(*)	D-97
STCMP	SOURCE.FORT(SRCH)	D-98
STEXPR	PRSCG.FORT(*)	C-76, E-102
STFN	PRSCG.FORT(*)	C-76, E-102
STGTF	SOURCE.FORT(SRCH)	D-98
STGTN	SOURCE.FORT(SRCH)	D-98
STKUP	SOURCE.FORT(PRSSP)	D-95
STMNT	PRSCG.FORT(*)	C-76
SUBSCR	PRSCG.FORT(*)	C-76
SWITCH	COMMON.FORT(*)	B-71
SYSINT	SOURCE.FORT(MNCTL)	D-92
SYSTEM	COMMON.FORT(*)	B-71
TABLE	COMMON.FORT(*)	B-71
TADDCH	SOURCE.FORT(LEX)	D-90
TASTMT	PRSCG.FORT(*)	C-76
TOKDAT	COMMON.FORT(*)	B-71, E-101
TOKEN	SOURCE.FORT(LEX)	D-90, E-104
TOUT	SOURCE.FORT(PRSSP)	D-95
TRCTRL	SOURCE.FORT(MNCTL)	D-92
TRINI	SOURCE.FORT(*)	D-99, E-104
TRNSLT	PRSCG.FORT(INIT)	C-73
TRVNUM	SOURCE.FORT(RUNFIL)	D-96
USTMNT	PRSCG.FORT(*)	C-76
WHILE	PRSCG.FORT(*)	C-77

NAME	ACCESS MECHANISM	APPENDIX
WRPUP	SOURCE.FORT(RUNFIL)	D-96
WRTST	PRSCG.FORT(*)	C-77

APPENDIX B

TRANSLATOR COMMON BLOCKS

COMMON   COMMON blocks

Purpose:
This subsystem contains all the parameter and common statements for the PLANS routines.

DSINFO   Dynamic Storage INFOrmation

DSTORP   Dynamic STORage Pool

EDIT   EDIT switch

ERINFO   ERror INFOrmation

FILES

Purpose:
This common block contains system unit numbers and names.

GENCOD   GENerate CODe

Purpose:
This unit contains parameter statements which set the numeric equivalent for each valid operation code and which set symbolic constants representing maximum array size. It also contains common statements for variables and arrays which support code generation.

GSTACK   General STACK

Purpose:
This common block contains arrays for supporting general stack needs for compilation.

IODEVS   Input/Output DEViceS

Purpose:
This unit contains a parameter statement of symbolic constants for file unit numbers.

LBINFO   LiBrary INFOrmation

PARSE

Purpose:
This common block contains the parse stack, flags, switches and pointers.

PGINFO   PaGe printing INFOrmation

RUNBUF   RUNfile input/output BUFfer

SWITCH

Purpose:

This common block contains user controlled (partially) option switches.

SYSTEM

Purpose:

This common block contains variables which are used to record system dependent word characteristics.

TABLE

Purpose:

This common block contains the components of the procedure/block, constant and local symbol tables.

TOKDAT TOKen DATa

Purpose:

This unit contains common statements for the major token types, new token and last token information, source line information and subtype information. It also contains key symbol and keyword equivalence statements.

APPENDIX C

TRANSLATOR PARSER/CODE GENERATOR ROUTINE WITH GRAMMAR RULES INCLUDED

PRSCG PaRSer/Code Generator

Purpose:

This subsystem contains all the recursive parsing rules which when joined together in PRSCGF form the parsing subroutine TRNSLT. Each unit within PRSCG contains a different parsing rule except for the units explained on this page. Within each parsing rule unit the labels are not in valid FORTRAN form to make them easier to understand and must be translated into acceptable form using PRSCG.CLIST.

INIT INITialize

Purpose:

This unit contains the beginning of the subroutine TRNSLT and therefore must be the first contained in PRSCGF. It contains all the INCLUDE statements for the required common blocks, the statements which assign meaningful names to the labels which begin each rule and the initial rule call to begin parsing.

PRSCGF PaRSer/Code Generator in valid FORTRAN

Purpose:

This unit contains the complete subroutine TRNSLT in valid FORTRAN.

RRETRN RETuRN from a Rule

Purpose:

This unit contains the stack popping mechanism which simulates the recursion necessary to perform the parsing. It must be the final unit which forms the subroutine TRNSLT.

TRNSLT TRaNSLaTe

Purpose:

This subroutine forms the driver routine for translation of PLANS statements.

Common Blocks:

DSTORP	dynamic storage pool
EDIT	edit switch
FILES	system unit numbers & names
GSTACK	general stack
PARSE	parsing stack, switches & counts
SWITCH	user controlled option switches
SYSTEM	system dependent word characteristics
TABLE	proc/block, constant & local symbol
TOKDAT	token data

```
AASTMT    arith_assign_stmnt := .id [subscript]1 '='
               expression

ADVST     advance_stmnt := 'advance' tree_pointer

AEXPR     arith_expr := mult_expr [add_op mult_expr]

BGBLK     begin_block := 'begin' block_body

BLKBDY    block_body := ';' [declare_stmnt] [non_end_unit]
               'end'
          declare_stmnt := 'declare' ['string' | 'integer'
               | 'float' ]1 var_list ['local']1 ';'

BTERM     boolean_term := relational_expr [and_op
               relational_expr]

CALLST    call_stmnt := 'call' procedure_name [parm_list]1

CMBPRM    comb_or_perm_tree := ('$combination' |
               '$permutation') '(' expression ')'

COMBCL    combinatorial_clause := ('combinations' |
               'permutations') 'of' soft_tree_node 'taken'
               expression 'at' 'a' 'time' do_body_thru_end

CSTMNT    conditional_stmnt := expression 'then' statement
               ['else' statement]1

DEFST     define_stmnt := 'define' tree_name 'as' hard_node

DOBDEN    do_body_thru_end := non_end_unit
               [non_end_unit] 'end'

DOGRP     do_group := 'do' [while_clause | subnodes_clause
               | combinatorial_clause | incremental_do]1
               | do_body_thru_end

EEXPR     exponential_expr := primary ['**' expression]1

EXPR      expression := boolean_term [or_op boolean_term]

EXTPRC    external_proc := .label procedure_block ';'

FILOPT    file_option := 'file' '(' .id ')'

GETST     get_stmnt := 'get' [file_option]1 'edit' '('
               input_list ')' format

GRFTST    graft_stmnt := 'graft' expression 'at'
               hard_tree_node
               | 'graft' 'insert' expression 'before'
               hard_tree_node
```

```
HARD      hard_tree_node := tree_reference hard_subscript

HRDSUB    hard_subscript := 'first:' expression
               | 'next'
               | 'first'
               | 'last'
               | expression

INCDO     incremental_do := .id '=' expression 'to'
               expression ['by' expression]1 [while_clause]1

INSRT     insert_stmnt := 'insert' expression 'before'
               hard_tree_node

LABSTR    label_string := '(' soft_tree_node ')'

LASTMT    label_assign_stmnt := 'label' '(' hard_tree_node
               ')' '=' expression

LBLQ      label_qual := .dot_string
               | indirect_reference

MEXPR     mult_expr := exponential_expr [mult_op
               exponential_expr]

NENDU     non_end_unit := [program_unit]

ORDRST    order_stmnt := 'order' soft_tree_node 'by'
               [order_argument]
          order_argument := ['-']1 ('$element' .id)
               [label_qual '(' soft_subscript ')']

PRIMRY    primary := soft_tree_node
               | ['-' | '+' | '¬' | 'not']1 primary
               | ['number' | 'label']1 label_string
               | string_function
               | .constant
               | .id [subscript]1
               | '(' expression ')'

PROCBL    procedure_block := 'procedure'
               ['(' formal_parm_list ')']1 ['options' '('
               option_list ')']1 ['recursive']1 ';'
               block_body

PROGU     program_unit := .label procedure_block
               | [.label]1 statement

PRUNST    prune_stmnt := 'prune' soft_node_list

PUTST     put_stmnt := 'put' [file_option]1 'edit' '('
               output_list ')' format

READST    read_stmnt := 'read' file_option
```

```
          read_element_list

REXPR     relational_expr := string_expr [relational_op
               string_expr]l

SBNODE    subnodes_clause := 'subnodes' 'of' soft_tree_node
               'using' tree_pointer

SFTSUB    soft_subscript := 'all:' expression
               | 'first:' expression
               | 'first'
               | 'last'
               | expression

SOFT      soft_tree_node := tree_reference [(.dot_string |
     soft_subscript)]

STEXPR    string_expr := arith_expr [cat_op arith_expr]

STFN      string_function := expression '('
               expression ':' expression ')'
               | 'length' '(' expression ')'
               | 'index' '(' expression ',' expression ')'
               | 'verify' '(' expression ',' expression ')'

STMNT     statement := 'if' conditional_stmnt
               | unconditional_stmnt ';'

SUBSCR    subscript := '(' expression [',' expression] ')'

TASTMT    tree_assign_stmnt := hard_tree_node '=' expression

USTMNT    unconditional_stmnt := tree_assign_stmnt
               | arith_assign_stmnt
               | begin_block
               | do_group
               | advance_stmnt
               | call_stmnt
               | define_stmnt
               | get_stmnt
               | graft_stmnt
               | insert_stmnt
               | label_assign_stmnt
               | order_stmnt
               | prune_stmnt
               | put_stmnt
               | read_stmnt
               | write_stmnt
               | 'assert' expression
               | 'go' 'to' .label
               | 'stop'
               | 'return'
               | 'trace' ('off' | 'high' | 'low')
```

```
WHILE      while_clause := 'while' '(' expression ')'

WRTST      write_stmnt := 'write' [file_option]1
                write_element_list
```

APPENDIX D

TRANSLATOR SOURCE ROUTINES

<u>SOURCE</u> SOURCE statements

Purpose:
This subsystem contains all the non-recursive translator and interpreter subsystems in source form.

<u>BLKTB</u> BLocK TaBle subsystem

Purpose:
The subroutines in this subsystem all deal with the block table and symbol table searching, referencing and, if necessary, inserting.

BLKEXT BLocK EXiT

Purpose:
This subroutine backs up a level in the block table to the parent of the current block.
Common Blocks:
PARSE parsing stack, switches & counts
SWITCH user controlled option switches
TABLE proc/block, constant and local symbol

CLRTAB CLeaR TABle

Purpose:
This subroutine clears (and optionally lists) the symbol table associated with the program block identified by BLKPTR.
Common Blocks:
DSINFO dynamic storage control information
DSTORP dynamic storage pool
IODEVS input/output device unit numbers
PGINFO page printing information
SWITCH user controlled option switches
TABLE proc/block, constant & local symbol

DFBGN DeFine BeGiN block

Purpose:
This subroutine puts a begin block entry in the block table.
Common Blocks:
PARSE parsing stack, switches & counts
TABLE proc/block, constant and local symbol

DFLAB DeFine LABel

Purpose:
This subroutine finds or enters the label string pointed to by NMPTR in the symbol table associated

with the current program block. The resulting location (binary tree node) is returned in PLOC. This routine should only be called after a potential label definition has been rejected as a procedure name definition.

Parameters:

NMPTR	pointer to label string
PLOC	pointer to label symbol table location

Common Blocks:

DSTORP	dynamic storage pool
GENCOD	opcode numbers & code generation support
PARSE	parsing stack, switches & counts
SWITCH	user controlled option switches
SYSTEM	system dependent word chararacteristics
TABLE	proc/block, constant and local symbol

DFPRC DeFine PRoCedure name

Purpose:

This subroutine establishes a procedure name definition in the block table.

Parameters:

NMPTR	pointer to procedure name in dynamic storage

Common Blocks:

DSTORP	dynamic storage pool
GENCOD	opcode numbers & code generation support
PARSE	parsing stack, switches & counts
SWITCH	user controlled option switches
TABLE	proc/block, constant and local symbol

DFTID DeFine Tree or IDentifier symbol

Purpose:

This subroutine enters NEWTOK, the current symbol, in the symbol table associated with the block identified by BLKPTR. It establishes default values for the symbol as the NEWNUM subtype.

Parameters:

BLKPTR	pointer to current block in block table
DTYPE	identifier type

Common Blocks:

DSTORP	dynamic storage pool
PARSE	parsing stack, switches & counts
TABLE	proc/block, constant and local symbol
TOKDAT	token data

DFTREE DeFine TREE

Purpose:

This subroutine defines a tree name as a based or indirect reference (SUBNODES USING clause or DEFINE statement) and checks for conflict in usage. NEWTOK (new token) is assumed to contain the relevant tree name.

Common Blocks:

DSTORP	dynamic storage pool
TABLE	proc/block, constant and local symbol
TOKDAT	token data

GNPRUN GeNerate PRUNe

Purpose:

This subroutine traverses the local symbol table and generates tree address load and prune instructions for the locally defined trees. This subroutine applies only to locally defined trees, not tree parameters, defined trees or unreferenced trees.

Parameters:

BLKPTR	pointer to current block in table

Common Blocks:

DSTORP	dynamic storage pool
GENCOD	opcode numbers & code generation support
PARSE	parsing stack, switches & counts
SWITCH	user controlled option switches
TABLE	proc/block, constant and local symbol

RFLAB ReFerence LABel

Purpose:

This subroutine establishes a reference pointer in PLOC (binary tree node index) for the label string identified by NMPTR. This routine should only be called after a 'GO TO' and following recognition of the identifier.

Parameters:

NMPTR	pointer to label string
PLOC	reference pointer

Common Blocks:

DSTORP	dynamic storage pool
GENCOD	opcode numbers & code generation support
PARSE	parsing stack, switches & counts
TABLE	proc/block, constant and local symbol

RFPRC ReFerence to a PRoCedure

Purpose:

This subroutine finds the entry in the block table for the referenced procedure. If none exists, it creates one.

Parameters:

NMPTR	pointer to procedure name in dynamic storage
PLOC	table location upon completion

Common Blocks:

DSTORP	dynamic storage pool
TABLE	proc/block, constant and local symbol

RFTID ReFerence to Tree or IDentifier

Purpose:

This subroutine searches the program block symbol tables within scope, in an 'inside out' fashion to locate the referenced symbol. If it is not found then it defines the symbol in the external block in which it must be. It delivers the result, a binary tree node pointer, as the NEWNUM subtype of the current NEWTOK (new token) symbol.

Common Blocks:

DSTORP	dynamic storage pool
PARSE	parsing stack, switches & counts
TABLE	proc/block, constant and local symbol
TOKDAT	token data

RFTREE ReFerence TREE

Purpose:

This subroutine references a tree name and establishes the usage subtype if it is not already established.

Common Blocks:

DSTORP	dynamic storage pool
GENCOD	opcode numbers & code generation support
TABLE	proc/block, constant and local symbol
TOKDAT	token data

SBLKT Search BLocK Table

Purpose:

This subroutine searches the block table for a name match starting at the index SRCHST and putting the result in RESULT.

Parameters:

NMPTR	Pointer to current name being processed
SRCHST	starting index in block table
RESULT	pointer to current name in block table

Common Blocks:

DSTORP	dynamic storage pool
TABLE	proc/block, constant and local symbol

<u>CGSUP</u> Code Generation SUPport subsystem

Purpose:

This subsystem contains subroutines which deal with pseudo-code generation.

CHKAC CHecK Arithmetic Conversion requirement

Purpose:

This subroutine generates a conversion of a string to floating point instruction.

Parameters:

OPR	operand type

Common Blocks:

GENCOD	opcode numbers & code generation support

CRFID Compile ReFerence to an IDentifier

Purpose:

This subroutine causes a PLANS pseudo-instruction to be generated for an identifier reference. The operation code generated is dependent upon whether it has an array versus variable, a floating point versus an integer variable, a direct versus an indirect (formal parameter) reference or an address versus a value reference. A secondary purpose is to pass back the dimension count of the identifier to the caller for use in conditioning further the compilation of subscripts and indexing. The identifier type is added to the general stack.

Parameters:

TLOC	pointer to identifier in symbol table
CVAL	switch for address vs value reference
DIMCNT	dimension count for an array identifier

Common Blocks:

DSTORP	dynamic storage pool
GENCOD	opcode numbers & code generation support
SWITCH	user controlled option switches
TABLE	proc/block, constant and local symbol

DFPSAD DeFine PSeudo-code ADdress (on the stack)

Purpose:

This subroutine defines a pseudo-code address at the current pseudo-code instruction counter and resolves forward reference linking which may have occurred.

Parameters:

NSTK	which stack
NADDR	address number in reservation list

Common Blocks:

GENCOD	opcode numbers & code generation support
IODEVS	input/output device unit numbers
SWITCH	user controlled option switches
SYSTEM	system dependent word characteristics

OUTIWA OUTput Instruction Word with pseudo-code Address

Purpose:

This subroutine retrieves a pseudo-code address from a pseudo-code address stack and catenates it with an operation code to form a full word instruction. Forward reference linking may occur.

Parameters:

OPCDE	operation code for instruction
NSTK	which pseudo-code address stack
NADDR	entry number in address stack list

Common Blocks:

GENCOD	opcode numbers & code generation support
SWITCH	user controlled option switches

OUTOP OUTput OPeration code

Purpose:

This subroutine outputs the parameter, given in right justified form, to the pseudo-code array. If the operation code is not a format 1 code then word boundary alignment is forced.

Parameters:

OPCODE	operation code

Common Blocks:

GENCOD	opcode numbers & code generation support
IODEVS	input/output device unit numbers
SWITCH	user controlled option switches
SYSTEM	system dependent word characteristics

OUTWRD OUTput a WoRD to the pseudo-code array

Purpose:

This subroutine constructs a full word pseudo-instruction (must not be format 1 instruction) from the pieces passed as parameters. For instructions with no display register and a full three bytes for the operand field, the DSPREG value should be zero and ASOFF should contain a right justified three byte operand field. If OPCODE is zero, then it is assumed that the word has already been constructed and properly formed in ASOFF.

Parameters:

OPCODE	operation code (right justified)
DSPREG	display register code (right justified)
ASOFF	automatic storage offset

Common Blocks:

GENCOD	opcode numbers & code generation support
IODEVS	input/output device unit numbers
SWITCH	user controlled option switches
SYSTEM	system dependent word characteristics

PSAPOP PSeudo-code Address stack POP

Parameters:

NSTK	which stack

Common Blocks:

GENCOD	opcode numbers & code generation support
IODEVS	input/output device unit numbers
SWITCH	user controlled option switches

RPSADS Reserve PSeudo-code ADdress Stack entries

Parameters:
- NSTK which stack
- NRESRV number of entries to reserve

Common Blocks:
- GENCOD opcode numbers & code generation support
- IODEVS input/output device unit numbers
- SWITCH user controlled option switches

<u>CHRST</u> CHaRacter STring subsystem

Purpose:
This subsystem contains the subroutines which deal with variable length character string manipulation. It is required because FORTRAN 77 does not support variable length character strings.

ADDCH ADD CHaracter

Purpose:
This subroutine adds a character to a variable length string.

Parameters:
- ISTRNG variable length string
- IADCHR character to be added

CPYST CoPY STring

Purpose:
This subroutine copies a variable length string from one array (IORIG) to another (ICOPY).

Parameters:
- IORIG array containing string to be copied
- ICOPY array into which string is copied

IEQST compare for EQuality of two STrings

Purpose:
This function compares two character strings for equality, returning 1 (true) if equal, 0 (false) otherwise. Equality requires complete identity, including length and content (e.g. no dissimilar blank fill).

Parameters:
- ISTR1 variable length string
- ISTR2 variable length string

IEXCL   EXtract a Character and Left justify

Purpose:
This function extracts a specified character from a specified string and delivers it left-justified with blank fill. The string is an array with its length in the first word.

Parameters:

ISTRNG	variable length character string array
IPOS	1-origin index of desired character

IEXCR   EXtract a Character and Right justify

Purpose:
This function extracts a specified character from a specified string and returns it in right-justified form. The string is an array with its length in the first word. The result is returned with zero left fill and hence this routine may be used to extract a 'small' integer which has been packed in character sized units. This version is intended only for the retrieval of integers in the range of 0 - 127 inclusive from an eight bit byte.

Parameters:

ISTRIN	variable length character string array
IPOS	1-origin index of desired character

RPCHL   RePlace CHaracter Left

Purpose:
This subroutine replaces a specified character in a given string with a specified replacement character.

Parameters:

ISTRNG	variable length string (input & output)
IPOS	1-origin index of character to replace
IRPCHR	replacement character (left justified)

RPCHR   RePlace a CHaracter Right justified

Purpose:
This subroutine replaces a specified character with a specified right-justified character. It is only intended for packing a 'small' integer in a one character unit in a variable length string because the value to be inserted (IRPCHR) is right-justified. Left fill is not relevant to the proper functioning of this routine.

Parameters:

ISTRNG	variable length string to be modified
IPOS	1-origin index of character to replace
IRPCHR	replacement character (right-justified)

CODLST   CODe LiSTing subsystem

Purpose:
This subsystem handles the output to the printer of the generated pseudo-code.

FRDUMP   FRequency DUMP of opcode generation

Common Blocks:

GENCOD	opcode numbers & code generation support
IODEVS	input/output device unit numbers

PSTLST   PoST LiST

Purpose:
This subroutine dumps the generated pseudo-code to the printer in semi-symbolic form.

Common Blocks:

GENCOD	opcode numbers & code generation support
IODEVS	input/output device unit numbers
SYSTEM	system dependent word characteristics

DYNAM   DYNAMic storage subsystem

Purpose:
This subsystem contains the subroutines which manage dynamic storage.

DSALC   Dynamic Storage ALloCation

Purpose:
This subroutine allocates a block of storage consistent with the request size. Only sizes in the specified generalized Fibonacci sequence are actually allocated. A large block may be split to satisfy the request.

Parameters:

ASIZE	requested size in words (exclusive of control word)
BLKIDX	index in ISBLK to beginning of allocated block (index is to word beyond control word)

Common Blocks:

DSINFO	dynamic storage control information
DSTORP	dynamic storage pool

DSINT   Dynamic Storage INiTialization

Purpose:
This subroutine divides up the dynamic storage pool (ISBLK) into maximum size blocks according to a specified size sequence (generalized Fibonacci

sequence). Storage blocks determined are marked, fictitiously, as having right buddies. Upon release from use (directly or by a merge) the right buddy does not have a zero left buddy count (split counter), so a merge to a larger than initial size is prevented.

Common Blocks:

DSINFO	dynamic storage control information
DSTORP	dynamic storage pool
IODEVS	input/output device unit numbers

DSRLS Dynamic Storage ReLeaSe

Purpose:

This subroutine deallocates a block of dynamic storage and merges it with its buddy if the buddy is free and unsplit. The merge continues as far as possible within the buddy system requirements. The index to the block to be released from use (BLKIDX) is expected to be one beyond the control word and is nulled out prior to return.

Parameters:

BLKIDX	index in ISBLK of block to be released

Common Blocks:

DSINFO	dynamic storage control information
DSTORP	dynamic storage pool

<u>ERROR</u> ERROR message subsystem

Purpose:

This subsystem contains all the routines which deal with compiler error message handling.

ERNUM ERror NUMber

Purpose:

This subroutine maps the two part error number to the print number and issues the error message. The fact of the error occurrence is recorded for later printing of the literal message.

Parameters:

IECLS	error class
IENUM	error number within class

Common Blocks:

ERINFO	error information
IODEVS	input/output device unit numbers
PARSE	parsing stack, switches & counts
SWITCH	user controlled option switches
TABLE	proc/block, constant and local symbol

ERRINT ERRor INiTialization

Purpose:
This subroutine initializes the error class start and type arrays from the error message file.

Common Blocks:
IODEVS input/output device unit numbers
ERINFO error information

ERRWRP ERRor WRaPup

Purpose:
This subroutine issues the literal messages for detected errors and restores things for a potential next round of external procedure processing.

Common Blocks:
ERINFO error information
IODEVS input/output device unit numbers

LABVAL LABel/VALue subsystem

Purpose:
This subsystem deals with the reserved label/value file.

LBVLI LaBel VaLue Input

Purpose:
This subroutine inputs a file of 'standard' label/value strings for use as key values for both translation and execution. Proper use of this facility can pay handsome dividends in both space and time for execution of PLANS programs. A separate program is provided which will build such a file from PLANS trees.

Common Blocks:
DSINFO dynamic storage control information
DSTORP dynamic storage pool
IODEVS input/output device unit numbers
TABLE proc/block, constant and local symbol

LVOUT LAbel/Values OUT

Purpose:
This subroutine constructs a collating sequence ordered reserved label/value file from the global string constant table.

Common Blocks:
DSTORP dynamic storage pool
IODEVS input/output device unit numbers
TABLE proc/block, constant & local symbol

<u>LEX</u> LEXical analysis subsystem

Purpose:
This subsystem contains the lexical analysis routines.

LXCAL LeXiCAL analyzer

Purpose:
This subroutine serves as an interface between the first pass parser and the token extractor. Key word identification, some contextual recognition and symbol table work is also performed.

Common Blocks:

DSTORP	dynamic storage pool
PARSE	parsing stack, switches & counts
SWITCH	user controlled option switches
TABLE	proc/block, constant and local symbol
TOKDAT	token data

TADDCH Token ADD CHaracter

Purpose:
This subroutine adds a character to a variable length string.

Parameters:

STRPTR	index to variable length string
NCHARS	length of string (input & output)
ADCHR	character to be added

Common Blocks:

DSTORP	dynamic storage pool

TOKEN TOKEN extraction

Purpose:
This subroutine is the basic token (symbol) extraction routine for the PLANS translator.

Common Blocks:

DSTORP	dynamic storage pool
EDIT	edit switch
IODEVS	input/output device unit numbers
LBINFO	library information
PARSE	parsing stack, switches & counts
SWITCH	user controlled option switches
SYSTEM	system dependent word characteristics
TOKDAT	token data

<u>LIB</u> LIBrary subsystem

Purpose:
This subsystem contains the library support routines.

FINLIB FINish LIBrary

Purpose:
This subroutine scans the procedure block table for external procedures to write to the library index and creates the control record.

Common Blocks:

DSTORP	dynamic storage pool
IODEVS	input/output device unit numbers
LBINFO	library information
PGINFO	page printing information
TABLE	proc/block, constant and local symbol

LIBINT LIBrary INiTialization

Purpose:
This subroutine either creates a dummy control record (library creation mode) or retrieves a control record (library use mode).

Common Blocks:

IODEVS	input/output device unit numbers
LBINFO	library information
SWITCH	user controlled option switches

LIBOUT LIBrary OUTput (create mode only)

Common Blocks:

IODEVS	input/output device unit numbers
LBINFO	library information
TOKDAT	token data

PRCLIB PRoCess LIBrary

Purpose:
This subroutine determines, from the block table, whether unresolved external procedure references exist. If so, the matching library modules are processed. New unresolved references from processing a library module are also examined for resolution from the library.

Common Blocks:

DSTORP	dynamic storage pool
ERINFO	error information
GENCOD	opcode numbers & code generation support
GSTACK	general stack
IODEVS	input/output device unit numbers
LBINFO	library information
PARSE	parsing stack, switches & counts
PGINFO	page printing information
SWITCH	user controlled option switches
TABLE	proc/block, constant and local symbol
TOKDAT	token data

<u>MNCTL</u> MaiN ConTroL subsystem

Purpose:
This subsystem contains the main program for the PLANS translator.

SYSINT SYStem INiTialization

Purpose:
This subroutine initializes the system dependent parameters for the PLANS translator.
Common Blocks:

LBINFO	library information
RUNBUF	runfile input/output buffer
SYSTEM	system dependent word characteristics

TRCTRL TRanslator ConTRoL

Purpose:
This is the main control program for the PLANS translator.
Common Blocks:

GENCOD	opcode numbers & code generation support
GSTACK	general stack
IODEVS	input/output device unit numbers
PARSE	parsing stack, switches & counts
SWITCH	user controlled option switches
SYSTEM	system dependent word characteristics
TOKDAT	token data

<u>OPTION</u> OPTIONs subsystem

Purpose:
This subsystem processes the options for PLANS external procedures.

FILOPT FILe OPTion list

Purpose:
This subroutine processes the '*FILES' control records.

OPTION OPTION list

Purpose:
This subroutine processes the optional '*PROCESS' control records and OPTION specifications on a PROCEDURE declaration.
Common Blocks:

DSTORP	dynamic storage pool
PARSE	parsing stack, switches & counts
SWITCH	user controlled option switches

TABLE	proc/block, constant and local symbol
TOKDAT	token data

<u>PAGE</u> PAGE subsystem

Purpose:
This subsystem contains the routines that control the printed output page and line management.

LINCNT LINe CoNTrol

Purpose:
This subroutine increases the line count and determines if page eject and headings are needed.

Parameters:

COUNT	line count

Common Blocks:

PGINFO	page printing information

NUPAG New PAGe

Purpose:
This subroutine does a page eject and prints the page headings for the main print file.

Common Blocks:

IODEVS	input/output device unit numbers
PGINFO	page printing information

<u>PRSSP</u> PaRSer SuPport subsystem

Purpose:
This subsystem contains all the non-recursive parsing rules and the general stack management routines.

DCLST DeCLare STatement

Purpose:
This subroutine parses declaration statements and performs appropriate symbol table work in support of the main parser.

Common Blocks:

DSTORP	dynamic storage pool
GENCOD	opcode numbers & code generation support
IODEVS	input/output device unit numbers
PARSE	parsing stack, switches & counts
SWITCH	user controlled option switches
TABLE	proc/block, constant and local symbol
TOKDAT	token data

ERRCV ERror ReCoVery

Purpose:
This subroutine handles error recovery. It does not allow a scan past EOF, END, DO, BEGIN or PROC.

Parameters:

TYPE	dictates the scan type:	
	0	scan to semicolon
	1	scan to comma
	2	scan to THEN
	3	scan by semicolon
	4	scan to right parenthesis

Common Blocks:

TOKDAT	token data

FPARM Formal PARaMeter

Purpose:
This subroutine parses the formal parameter list and performs the requisite symbol table operations in support of the main parser.

Common Blocks:

DSTORP	dynamic storage pool
IODEVS	input/output device unit numbers
PARSE	parsing stack, switches & counts
TABLE	proc/block, constant and local symbol
TOKDAT	token data

GENPOP GENeral stack POP

Purpose:
This subroutine recovers an integer value from the general stack and pops the stack. If the stack type does not match the type parameter then it issues an error message.

Parameters:

TYPE	identification code
VALUE	integer value recovered

Common Blocks:

GSTACK	general stack
IODEVS	input/output device unit numbers
SWITCH	user controlled option switches

GENSTK GENeral STaCK

Purpose:
This subroutine puts an integer value and identification code on the general stack for later recovery.

Parameters:

TYPE	identification code
VALUE	integer value

Common Blocks:

GSTACK	general stack

```
        IODEVS    input/output device unit numbers
        SWITCH    user controlled option switches

GENSWP  GENeral stack SWaP

    Purpose:
        This subroutine  swaps the  top two  items on  the
        general stack.
    Common Blocks:
        GSTACK    general stack

IGNORE  IGNORE any commas

    Common Blocks:
        TOKDAT    token data

SEMI    SEMIcolon

    Purpose:
        This subroutine checks for a semicolon,  issues an
        error message if  it is not there  and bypasses it
        if it is there.
    Common Blocks:
        PARSE     parsing stack, switches & counts
        TOKDAT    token data

STKUP   STacK UP

    Purpose:
        This subroutine checks the  address (parse)  stack
        top limit and increments it  if it has not reached
        its limit.
    Common Blocks:
        PARSE     parsing stack, switches & counts

TOUT    Trace OUT

    Purpose:
        This  subroutines   outputs  parsing   trace  flow
        information.
    Parameters:
        CODE      numeric rule identification
    Common Blocks:
        IODEVS    input/output device unit numbers
        PARSE     parsing stack, switches & counts
        PGINFO    page printing information
```

<u>RUNFIL</u> RUNFILe subsystem

```
    Purpose:
        This subsystem contains all the runfile management
        routines.
```

OUTAB OUTput TABles

Purpose:
This subroutine outputs tables to the runfile.

Common Blocks:

DSTORP	dynamic storage pool
GENCOD	opcode numbers & code generation support
IODEVS	input/output device unit numbers
RUNBUF	runfile input/output buffer
SWITCH	user controlled option switches
TABLE	proc/block, constant and local symbol

OUTCOD OUTput CODe

Purpose:
This subroutine dumps any generated code to the runfile.

Common Blocks:

GENCOD	opcode numbers & code generation support
IODEVS	input/output device unit numbers
RUNBUF	runfile input/output buffer
SWITCH	user controlled option switches

RNINT RuN INiTialization

Purpose:
This subroutine initializes the runfile, if present.

Common Blocks:

IODEVS	input/output device unit numbers
RUNBUF	runfile input/output buffer
SWITCH	user controlled option switches

TRVNUM TRaVerse NUMeric

Purpose:
This subroutine traverses the numeric table to place numeric constants in address order in the specified output vector.

Parameters:

ROOT	root index of table (binary tree)
OUTPUT	vector for recording output

Common Blocks:

DSTORP	dynamic storage pool
GSTACK	general stack
TABLE	proc/block, constant and local symbol

WRPUP WRaP UP

Purpose:
This subroutine checks for loading map requirements and dumps the block table to the runfile.

Common Blocks:

DSTORP	dynamic storage pool
FILES	system unit numbers & names
GENCOD	opcode numbers & code generation support
IODEVS	input/output device unit numbers
PARSE	parsing stack, switches & counts
RUNBUF	runfile input/output buffer
SWITCH	user controlled option switches
TABLE	proc/block, constant and local symbol

<u>SRCH</u> SeaRCH subsystem

Purpose:
This subsystem contains all the subroutines which deal with binary tree management.

BINSR BINary SeaRch

Purpose:
This subroutine performs a binary search on a list of character strings ordered and linked by a vector of links.

Parameters:

LINK	index links to list of variable length character strings maintained in the dynamic storage pool
LAST	index to last link in list
IARG	index to dynamic storage for character string search argument
IRSLT	index to link where match occurs (0 for failure)

BSTSR Binary Search Tree SeaRch

Purpose:
This subroutine serves three roles as directed by the input parameter ITYPE. One is to search for a given symbol in a binary search tree. The second is to find or enter a given symbol in a binary search tree. The third is to delete a given symbol from a binary search tree. In any case, a preliminary search takes place.

Parameters:

NSFLG	0 - numeric search argument & target 1 - variable length character string search argument & target
ITYPE	role selection
LROOT	link to root node
IARG	search argument
LMATCH	search match or insertion location link
NMCNT	I/O parameter incremented by 1 if a new symbol is in fact entered

Common Blocks:

DSTORP	dynamic storage pool
TABLE	proc/block, constant and local symbol

BTINT Binary Tree INiTialization

Purpose:
This subroutine initializes a pool of binary tree nodes. Right links are used to create a list of available nodes. Left links are null. The node in position 1 is used as a dummy node with its left link used for a root node pointer and its right link used as a list header for available nodes. Name and value links are also set to null.

Common Blocks:

TABLE	proc/block, constant and local symbol

STCMP STring CoMParison

Purpose:
This subroutine performs a collating sequence comparison of two character strings. Note that strings of unequal length cannot be equal.

Parameters:

NM1	left character string operand pointer
NM2	right character string operand pointer
IRSLT	result of the comparison

Common Blocks:

DSTORP	dynamic storage pool

STGTF Symbol Table GeT First

Purpose:
This subroutine determines the first node in a binary tree symbol table by finding the leftmost node (first node in collating sequence order).

Parameters:

LROOT	index to root node of tree
LRSLT	index to leftmost node

Common Blocks:

TABLE	proc/block, constant and local symbol

STGTN Symbol Table GeT Next

Purpose:
This subroutine finds the next node in collating sequence order in a symbol table binary tree.

Parameters:

LROOT	index to root node of tree
LPRVNM	index to dynamic storage for previous name string
LRSLT	index to tree node found

Common Blocks:

DSTORP	dynamic storage pool

TABLE proc/block, constant and local symbol

<u>TRINI</u> TRanslator static INItialization

Purpose:
This blockdata routine contains all the data statements needed to do all the translator initializations for the common blocks.

Common Blocks:

DSINFO	dynamic storage control information
DSTORP	dynamic storage pool
EDIT	edit switch
ERINFO	error information
FILES	system unit numbers & names
GENCOD	opcode numbers & code generation support
PARSE	parsing stack, switches & counts
PGINFO	page printing information
RUNBUF	runfile input/output buffer
SWITCH	user controlled option switches
TABLE	proc/block, constant and local symbol
TOKDAT	token data

APPENDIX E

CHANGES TO THE TRANSLATOR AS A RESULT OF THE CHARACTER STRING EXTENSION

COMMON COMMON blocks

GENCOD GENerate CODe

Changes:
Operation codes for string functions were added (83-88, 108-118).

TOKDAT TOKen DATa

Changes:
Tables which deal with key symbol subtype information and key word information were altered to include the concatenation symbol and the FLOAT, INDEX, INTEGER, LENGTH, STRING and VERIFY key words.

PRSCG PaRSe with Code Generation

AASTMT Arithmetic Assignment STateMenT

Changes:
The possiblity of a string variable on the left side of the equal sign was handled.

String Instructions Generated:

CVS	ConVert to String
AS	Assign String

INIT INITialization

Changes:
String related rule names were added to the initializations.

PRIMRY PRIMaRY

Changes:
A call to STFN (string function) was added if keywords LENGTH, INDEX or VERIFY were found. A call to STFN was also generated if the context suggested that the substring function was possible.

String Instructions Generated:

CVS	ConVert to String
LSA	Load String Address

REXPR Relational EXPRession

Changes:
A call to STEXPR (string expression) was generated so that the concatenation operator could have priority over the relational operators but not

over the arithmetic operators. With the existing relational instructions, if either operand was string type, it would be converted to floating point. String relational operators were added so that if both operands were of string type then string comparisons would be made.

String Instructions Generated:

SEQ	String EQual
SGE	String Greater than or Equal
SGT	String Greater Than
SLE	String Less than or Equal
SLT	String Less Than
SNE	String Not Equal

STEXPR STring EXPRession

Changes:

This unit was added to handle the rule dealing with concatenation. It is called by REXPR (relational expression) and calls AEXPR (arithmetic expression).

String Instructions Generated:

CVS	ConVert to String
SCAT	String conCATenation

STFN STring FuNction

Changes:

This unit was added to handle the index, length, substring and verify functions. It is called by PRIMRY (primary).

String Instructions Generated:

CVS	ConVert to String
SIND	String INDex
SLEN	String LENgth
SSUB	String SUBstring
SVER	String VERify

<u>SOURCE</u> SOURCE statements

<u>BLKTB</u> BLocK TaBle

DFTID DeFine Tree or IDentifier symbol

Changes:

An identifier type parameter was added to allow for explicit type declarations.

RFTID ReFerence to a Tree or IDentifier

Changes:

An identifier type parameter of U (undefined) was added to the call to DFTID (define tree or identifier) to force default type if the identifier is not found.

<u>CGSUP</u> Code Generation SUPport routines

CRFID Compile Reference to an IDentifier

Changes:

This unit was altered to generate string loading instructions for string array , string variable and string parameter references.

String Instructions Generated:

LSD	Load String array Descriptor address
LSDI	Load String array Descriptor address Indirectly
LSVA	Load String Variable Address
LSVI	Load String Variable address Indirect

<u>ERROR</u> ERROR message routines

ERRINT ERRor INiTialization

Changes:

This unit was changed to expand the error number to three columns.

ERRWRP ERRor WRaPup

Changes:

This unit was changed to expand the error number to three columns.

<u>LEX</u> LEXical analysis

LXCAL LeXiCAL analyzer

Changes:

An identifier followed directly by a colon was treated as one unit. A check was added to see if it was a valid label context and, if not, code was added to treat it as two separate tokens. This was necessary because the colon could be part of the substring function.

TOKEN TOKEN extraction

Changes:

Two new states were added (31 and 32) to change | (the OR symbol) from just a one character token to either a one character or a two character (|| concatenation) symbol.

<u>PRSSP</u> PaRSer SuPport

DCLST DeCLare STatement

Changes:

Parsing of the three new declaration statements was added. This was to allow for explicit type declarations.

String Instructions Generated:

LSD Load String array Descriptor address

FPARM Formal PARaMeter

Changes:

A parameter of U (undefined) was added to the call to DFTID. If an explicit type declaration is encountered in the subroutine, the default type is overridden.

<u>TRINI</u> TRanslator INItialization

Changes:

Changes and additions were made to the data statements to allow for the new key symbols and keywords.

APPENDIX F

INDEX TO THE INTERPRETER SUBSYSTEM

NAME	ACCESS MECHANISM	APPENDIX
ADDCH	SOURCE.FORT(CHRST)	D-85
ALAB	SOURCE.FORT(TRESUP)	H-127
ALLA	SOURCE.FORT(ARRAY)	H-114
ARRAY	SOURCE.FORT(*)	H-114
ATRE	SOURCE.FORT(TRESUP)	H-127
BINIO	COMMON.FORT(*)	G-111
BINSR	SOURCE.FORT(*)	H-115
BINSR	SOURCE.FORT(BINSR)	H-115
BOUNDS	COMMON.FORT(*)	G-111
CCOPY	SOURCE.FORT(*)	H-115
CCOPY	SOURCE.FORT(CCOPY)	H-115
CHRST	SOURCE.FORT(*)	D-85
CMBPRM	SOURCE.FORT(*)	H-115
CNUM	SOURCE.FORT(CONVRT)	H-117
COMMON	*.FORT	G-111
CONVRT	SOURCE.FORT(*)	H-117
CONVRT	SOURCE.FORT(CONVRT)	H-117
CPYST	SOURCE.FORT(CHRST)	D-85
CPYTRE	SOURCE.FORT(TRESUP)	H-128
CVF	SOURCE.FORT(CVICVF)	H-117
CVI	SOURCE.FORT(CVICVF)	H-117
CVICVF	SOURCE.FORT(*)	H-117
DSALC	SOURCE.FORT(INDYNAM)	H-119
DSINFO	COMMON.FORT(*)	B-70
DSINT	SOURCE.FORT(INDYNAM)	H-120
DSRLS	SOURCE.FORT(INDYNAM)	H-120

NAME	ACCESS MECHANISM	APPENDIX
ELMT	SOURCE.FORT(TRESUP)	H-128
ENCODE	SOURCE.FORT(*)	H-118
ENCODE	SOURCE.FORT(ENCODE)	H-118
EQ	SOURCE.FORT(TRESUP)	H-128
ERINFO	COMMON.FORT(*)	B-70
ERNUM	SOURCE.FORT(INERROR)	H-120
ERRINT	SOURCE.FORT(INERROR)	H-121
ERWRP	SOURCE.FORT(INERROR)	H-121
FCMB	SOURCE.FORT(CMBPRM)	H-115
FPRM	SOURCE.FORT(CMBPRM)	H-116
GET	SOURCE.FORT(GETPUT)	H-118
GETED	SOURCE.FORT(GETPUT)	H-118
GETPUT	SOURCE.FORT(*)	H-118
GPLAB	SOURCE.FORT(NODE)	H-122
GPSIB	SOURCE.FORT(NODE)	H-122
GRFT	SOURCE.FORT(TRESUP)	H-128
GRIS	SOURCE.FORT(TRESUP)	H-128
GTVAL	SOURCE.FORT(NODE)	H-123
HARD	SOURCE.FORT(TREQUAL)	H-126
IDNT	SOURCE.FORT(TRESUP)	H-129
IDXA	SOURCE.FORT(ARRAY)	H-114
IDXV	SOURCE.FORT(ARRAY)	H-114
IEQST	SOURCE.FORT(CHRST)	D-85
IEXCL	SOURCE.FORT(CHRST)	D-85
IEXCR	SOURCE.FORT(CHRST)	D-86
INCNTL	COMMON.FORT(*)	G-111

NAME	ACCESS MECHANISM	APPENDIX
INDAT	SOURCE.FORT(*)	H-119
INDAT	SOURCE.FORT(INDAT)	H-119
INDEVS	COMMON.FORT(*)	G-111
INDYNAM	SOURCE.FORT(*)	H-119
INEDIT	COMMON.FORT(*)	G-111
INERROR	SOURCE.FORT(*)	H-120
INGLOB	COMMON.FORT(*)	G-111
ININT	SOURCE.FORT(INTCNTL)	H-121
INNODE	COMMON.FORT(*)	G-112
INSTAC	COMMON.FORT(*)	G-112
INTCNTL	SOURCE.FORT(*)	H-121
INTCTL	SOURCE.FORT(INTCNTL)	H-121
INTFE	SOURCE.FORT(*)	H-122
INTFE	SOURCE.FORT(INTFE)	H-122
ISRT	SOURCE.FORT(TRESUP)	H-129
MVLBVL	SOURCE.FORT(WRITE)	H-130
NCMB	SOURCE.FORT(CMBPRM)	H-116
NEWNOD	SOURCE.FORT(NODE)	H-123
NODE	SOURCE.FORT(*)	H-122
NPRM	SOURCE.FORT(CMBPRM)	H-116
ORDER	SOURCE.FORT(*)	H-124
ORDER	SOURCE.FORT(ORDER)	H-124
PPLAB	SOURCE.FORT(NODE)	H-123
PPSIB	SOURCE.FORT(NODE)	H-123
PREOR	SOURCE.FORT(*)	H-125
PREOR	SOURCE.FORT(PREOR)	H-125

NAME	ACCESS MECHANISM	APPENDIX
PREOR2	SOURCE.FORT(PREOR)	H-125
PRUNE	SOURCE.FORT(TRESUP)	H-129
PTVAL	SOURCE.FORT(NODE)	H-124
PUT	SOURCE.FORT(GETPUT)	H-118
PUTED	SOURCE.FORT(GETPUT)	H-119
READ	SOURCE.FORT(*)	H-125
READ	SOURCE.FORT(READ)	H-125
RESERV	SOURCE.FORT(READ)	H-126
RPCHL	SOURCE.FORT(CHRST)	D-86
RPCHR	SOURCE.FORT(CHRST)	D-86
RTREE	SOURCE.FORT(READ)	H-126
SBST	SOURCE.FORT(TRESUP)	H-129
SHELLM	SOURCE.FORT(ORDER)	H-124
SNIP	SOURCE.FORT(TRESUP)	H-130
SOFT	SOURCE.FORT(TREQUAL)	H-127
SOURCE	*.FORT	H-114
STAT	SOURCE.FORT(INTCNTL)	H-122
SYSTEM	COMMON.FORT(*)	B-71
TRAVER	COMMON.FORT(*)	G-112
TREQUAL	SOURCE.FORT(*)	H-126
TRESUP	SOURCE.FORT(*)	H-127
WORK	COMMON.FORT(*)	G-112
WRIT	SOURCE.FORT(WRITE)	H-130
WRITE	SOURCE.FORT(*)	H-130
WTREE	SOURCE.FORT(WRITE)	H-130

APPENDIX G

INTERPRETER COMMON BLOCKS

COMMON   COMMON blocks

Purpose:
This subsystem contains all the parameter and common statements for the PLANS routines.

BINIO   BiNary Input/Output

Purpose:
This common block contains control variables used to access and manage the buffer used by PLANS for input/output of trees in binary form.

BOUNDS

Purpose:
This common block serves as the PLANS pseudo-machine control block. It defines physical boundaries within the PLANS run file that separate it into its nine logical address spaces.

INCNTL   INterpreter CoNTroL

Purpose:
This common block contains data needed for control of the PLANS interpreter fetch/execute cycle.

INDEVS   INterpreter DEViceS

Purpose:
This parameter block contains the default file unit numbers used by the PLANS interpreter.

INEDIT   INterpreter EDIT input/output

Purpose:
This common block contains information used by the PLANS interpreter support routines that implement the GET EDIT and PUT EDIT statements.

INGLOB   INterpreter GLOBal

Purpose:
This common block serves as the global address space for the PLANS interpreter pseudo-machine. It contains all the pseudo-code and run-time data. (The boundary markers that separate it into the nine major sections are defined in the BOUNDS common block.)

INNODE INterpreter NODE storage

Purpose:
This common block contains data used to manage the tree node storage space of the PLANS interpreter. Node storage is dynamically managed by controlled allocation from a "free" list.

INSTAC INterpreter STAck

Purpose:
This common block contains the PLANS pseudo-machine run-time stack, the stack barrier stack and their associated data.

TRAVER TRAVERsal stacks

Purpose:
This common block contains two stacks used for preorder tree traversal (TSTAC and TSTAC2).

WORK

Purpose:
This common block serves as a place to record pointers to common work areas maintained in the common block INGLOB.

APPENDIX H

INTERPRETER SOURCE ROUTINES

<u>SOURCE</u> SOURCE statements

Purpose:
This subsystem contains all the non-recursive translator and interpreter subsystems in source form.

<u>ARRAY</u> ARRAY subsystem

Purpose:
This subsystem handles array management insructions.

ALLA Array ALLocation instruction

Purpose:
This subroutine allocates array space in display storage for a declared array, computes a mapping descriptor for the array and places it in already allocated display storage space.

Common Blocks:

BOUNDS	pseudo-machine control block
INCNTL	interpreter fetch/execute control
INGLOB	interpreter pseudo-machine storage
INSTAC	run-time & stack barrier stack

IDXA InDeX to an Address instruction

Purpose:
Given an array descriptor address and subscripts on the run-time stack, this subroutine computes the address of the referenced element and places this address on the stack.

Common Blocks:

BOUNDS	pseudo-machine control block
INCNTL	interpreter fetch/execute control
INGLOB	interpreter pseudo-machine storage
INSTAC	run-time & stack barrier stack

IDXV InDeX to a Value instruction

Purpose:
Given an array descriptor address and subscripts on the run-time stack, this subroutine locates the referenced array element and puts its value on the run-time stack.

Common Blocks:

INGLOB	interpreter pseudo-machine storage
INSTAC	run-time & stack barrier stack

BINSR   BINary SEarch subsystem

BINSR   BINary SEarch

Purpose:
This subroutine performs a binary search on a list of character strings ordered and linked by a vector of links.

Parameters:

LINK	index links to list of variable length character strings maintained in dynamic storage pool
LAST	index to last link in list
IARG	index to dynamic storage for character string search argument
IRSLT	index to link where match occurs (0 for failure)

CCOPY   Conditional COPY subsystem

CCOPY   Conditional COPY

Purpose:
This subroutine reserves dynamic storage space for the character string pointed to by POLD. If the string already resides in string constant space or reserved label/value space, it is not necessary to reserve new space for an extra copy. A pointer to the existing or newly allocated string is returned.

Parameters:

POLD	pointer to input character string
PNEW	pointer to location of character string (output)

Common Blocks:

BOUNDS	pseudo-machine control block
INGLOB	interpreter pseudo-machine storage

CMBPRM   CoMBination and PeRMutation subsystem

Purpose:
This subsystem contains the routines necessary to handle the instructions dealing with combinations and permutations of tree subnodes.

FCMB   First CoMBination instruction

Purpose:
This subroutine computes the list of pointers for the first combination of a specified (sub)tree.

Parameters:
- CSIZE — to assist FPRM which calls this routine

Common Blocks:
- BOUNDS — pseudo-machine control block
- INCNTL — interpreter fetch/execute control
- INDEVS — interpreter device unit numbers
- INGLOB — interpreter pseudo-machine storage
- INSTAC — run-time & stack barrier stack

FPRM First PeRMutation instruction

Purpose:

This subroutine computes control information for (sub)tree permutation processing. FCMB (first combination) is called first since permutations are determined by permuting combinations.

Common Blocks:
- BOUNDS — pseudo-machine control block
- INCNTL — interpreter fetch/execute control
- INGLOB — interpreter pseudo-machine storage

NCMB Next CoMBination instruction

Purpose:

This subroutine determines the next combination in the current combinatorial loop by updating the pointer list in display storage. If all combinations have been exausted, this is indicated on the run-time stack by a logical 0. A successful next combination returns a logical 1.

Common Blocks:
- INCNTL — interpreter fetch/execute control
- INGLOB — interpreter pseudo-machine storage
- INSTAC — run-time & stack barrier stack

NPRM Next PeRMutation instruction

Purpose:

This subroutine establishes the next permutation for the current combination loop from information in display storage. If failure occurs (indicated by an attempt to get the next combination), additional display storage allocated for permutation processing is released.

Common Blocks:
- INCNTL — interpreter fetch/execute control
- INGLOB — interpreter pseudo-machine storage
- INSTAC — run-time & stack barrier stack

<u>CONVRT</u> CONVeRT subsystem

Purpose:
This subsystem contains subroutines which attempt to convert a string to internal numeric form.

CNUM Convert to NUMeric

Purpose:
This subroutine attempts to convert a string to internal numeric form.

Parameters:

NCHARS	number of characters in string
SPCPTR	pseudo-machine space subscript
TYPE	resulting type
IVAL	internal value of integer string found
RNUM	internal value of real string found

Common Blocks:

INGLOB	interpreter pseudo-machine storage

CONVRT CoNVeRT

Purpose:
This subroutine attempts to locate an integer or floating point number in the string argument and, if found, converts it to internal form.

Parameters:

STRING	vector of characters
NCHAR	number of characters in string
TYPE	resulting type
INUM	internal value of integer string found
RNUM	internal value of real string found
CLOC	location of first non-blank character

<u>CVICVF</u> ConVert to Integer and ConVert to Floating point subsystem

CVF ConVert to Floating point instruction

Purpose:
This subroutine converts the item on top of the run-time stack to floating point.

Common Blocks:

BOUNDS	pseudo-machine control block
INGLOB	interpreter pseudo-machine storage
INSTAC	run-time & stack barrier stack

CVI ConVert to Integer instruction

Purpose:
This subroutine converts the item on top of the run-time stack to integer.

Common Blocks:

BOUNDS	pseudo-machine control block
INGLOB	interpreter pseudo-machine storage
INSTAC	run-time & stack barrier stack

ENCODE ENCODE subsystem

ENCODE

Purpose:
This subroutine is used to get around FORTRAN 77 implementation restrictions.

Common Blocks:

INGLOB	interpreter pseudo-machine storage

GETPUT GET and PUT subsystem

GET GET instruction

Purpose:
This subroutine implements the GET EDIT statement.

Common Blocks:

INEDIT	interpreter EDIT I/O
INGLOB	interpreter pseudo-machine storage
INSTAC	run-time & stack barrier stack

GETED GET EDit

Purpose:
This subroutine performs formatted input of numeric values into PLANS display storage.

Parameters:

LENFMT	character length of user format string
IFMT	user supplied format string

Common Blocks:

INCNTL	interpreter fetch/execute control
INEDIT	interpreter EDIT I/O
INGLOB	interpreter pseudo-machine storage
INSTAC	run-time & stack barrier stack

PUT PUT instruction

Purpose:
This subroutine implements the PLANS PUT EDIT instruction.

Common Blocks:

BOUNDS	pseudo-machine control block
INEDIT	interpreter EDIT I/O
INGLOB	interpreter pseudo-machine storage
INSTAC	run-time & stack barrier stack

PUTED PUT EDit

Purpose:
This subroutine performs formatted output of numeric values and character strings.

Parameters:

LENFMT character string length of user supplied format string

IFMT user supplied format specification string

Common Blocks:

INCNTL interpreter fetch/execute control
INEDIT interpreter EDIT I/O
INGLOB interpreter pseudo-machine storage
INSTAC run-time & stack barrier stack

<u>INDAT</u> INterpreter block DATa subsystem

INDAT INterpreter block DATa

Purpose:
This block data routine initializes static data values in the PLANS interpreter common blocks.

Common Blocks:

DSINFO dynamic storage control information
ERINFO error information
INCNTL interpreter fetch/execute control
INDEVS interpreter device unit numbers

<u>INDYNAM</u> INterpreter DYNAMic storage subsystem

Purpose:
This subsystem handles interpreter dynamic storage management.

DSALC Dynamic Storage ALloCation

Purpose:
This subroutine allocates a block of storage consistent with the request size. Only sizes in the specified generalized Fibonacci sequence are actually allocated. A large block may be split to satisfy the request.

Parameters:

ASIZE request size in words (exclusive of control word)

BLKIDX index in ISBLK to beginning of allocated block

Common Blocks:

DSINFO	dynamic storage control information
INGLOB	interpreter pseudo-machine storage

DSINT Dynamic Storage INiTialization

Purpose:
This subroutine divides up the dynamic storage pool (ISBLK) into maximum size blocks according to the specified generalized Fibonacci sequence.

Common Blocks:

DSINFO	dynamic storage control information
INGLOB	interpreter pseudo-machine storage

DSRLS Dynamic Storage ReLeaSe

Purpose:
This subroutine deallocates a block of dynamic storage. If its buddy is free and not split they are merged. Merging continues as far as possible within the buddy system requirements.

Parameters:

BLKIDX	index to block to be released from use

Common Blocks:

DSINFO	dynamic storage control information
INGLOB	interpreter pseudo-machine storage

<u>INERROR</u> INterpreter ERROR subsystem

Purpose:
This subsystem contains the routines which handle errors.

ERNUM ERror NUMber

Purpose:
This subroutine maps the two part error number to the entry in the table of error messages and issues a message. The occurrence of the error is recorded so that the literal message will be printed later.

Parameters:

IECLS	error class
IENUM	error number within class

Common Blocks:

BOUNDS	pseudo-machine control block
ERINFO	error information
INCNTL	interpreter fetch/execute control
INDEVS	interpreter device unit numbers
INGLOB	interpreter pseudo-machine storage

ERRINT ERRor INiTialization

Purpose:
This subroutine initializes the error type array and class start array.

Common Blocks:
ERINFO error information
INDEVS interpreter device unit numbers

ERWRP ERrorWRaPup

Purpose:
This subroutine issues the literal messages for detected errors and restores things for a potential next round of external procedure processing.

Common Blocks:
ERINFO error information
INDEVS interpreter device unit numbers

<u>INTCNTL</u> INTerpreter CoNTroL subsystem

Purpose:
This subsystem contains the main interpreter program, initialization and wrapup routines.

ININT INterpreter INiTialization

Purpose:
This is the main initialization subroutine for the PLANS interpreter. It sets up the PLANS pseudo-machine address space with values obtained during translation. It also assigns initial values to other common areas used by the interpreter.

Common Blocks:
BOUNDS pseudo-machine control block
DSINFO dynamic storage control information
INCNTL interpreter fetch/execute control
INDEVS interpreter device unit numbers
INGLOB interpreter pseudo-machine storage
INNODE interpreter node storage management
INSTAC run-time & stack barrier stack
TRAVER preorder traversal stacks
SYSTEM system dependent word characteristics
WORK record pointers to common work areas

INTCTL INTerpreter ConTroL

Purpose:
This is the interpreter's main program.

Common Blocks:
INCNTL interpreter fetch/execute control

INDEVS   interpreter device unit numbers
SYSTEM   system dependent word characteristics

STAT   STATistics

Purpose:
This subroutine dumps the performance statistics to the printer.
Common Blocks:
INCNTL   interpreter fetch/execute control
INDEVS   interpreter device unit numbers

<u>INTFE</u>   INTerpreter Fetch/Execute subsystem

INTFE   INTerpreter Fetch/Execute

Purpose:
This subroutine is the main fetch/execute control loop for the PLANS pseudo-machine. It iteratively decodes and executes pseudo-machine instructions.
Common Blocks:
BOUNDS   pseudo-machine control block
INCNTL   interpreter fetch/execute control
INDEVS   interpreter device unit numbers
INSTAC   run-time & stack barrier stack
SYSTEM   system dependent word characteristics

<u>NODE</u>   NODE subsystem

Purpose:
This subsystem handles tree node management.

GPLAB   Get Pointer to a LABel

Purpose:
This function returns a label pointer of a tree node.
Parameters:
INDEX   index in tree node storage space
Common Blocks:
INGLOB   interpreter pseudo-machine storage
SYSTEM   system dependent word characteristics

GPSIB   Get Pointer to a SIBling

Purpose:
This function retrieves the low order half word size -1 bits of any word in pseudo-machine space and is frequently used for other than sibling pointer retrieval.

Parameters:
INDEX index in tree node storage space
Common Blocks:
INGLOB interpreter pseudo-machine storage
SYSTEM system dependent word characteristics

GTVAL Get Type and VALue

Purpose:
This subroutine retrieves the type code for a node and a pointer (descendant or string), integer value or real value, depending on the value of ITYPE.
Parameters:
INDEX index in tree node storage space
ITYPE type code for word 2
IVALUE integer number or pointer
RVALUE floating point number
Common Blocks:
INGLOB interpreter pseudo-machine storage
SYSTEM system dependent word characteristics

NEWNOD NEW NODe

Purpose:
This subroutine removes a node from the "free" list, initializes its components and returns a pointer to it.
Parameters:
PNODE pointer to new tree node
Common Blocks:
INGLOB interpreter pseudo-machine storage
INNODE interpreter node storage management

PPLAB Put Pointer to a LABel

Purpose:
This subroutine sets a label pointer of a tree node.
Parameters:
INDEX index in tree node storage space
VALUE label pointer value
Common Blocks:
INGLOB interpreter pseudo-machine storage
SYSTEM system dependent word characteristics

PPSIB Put Pointer to a SIBling

Purpose:
This subroutine sets the low order half word -1 bits of a word. A sibling pointer may or may not be involved.
Parameters:
INDEX index in tree node storage space

VALUE pointer value

Common Blocks:

INGLOB interpreter pseudo-machine storage
SYSTEM system dependent word characteristics

PTVAL Put Type and VALue

Purpose:

This subroutine sets the type code for a node and a descendant pointer, string pointer, integer value or floating point value, depending upon the value of type.

Parameters:

INDEX index in tree node storage space
TYPE type code for word 2
IVALUE integer number or pointer
RVALUE floating point number

Common Blocks:

INGLOB interpreter pseudo-machine storage
SYSTEM system dependent word characteristics

<u>ORDER</u> ORDER subsystem

ORDER

Purpose:

This subroutine supports the ORDR primitive of the PLANS pseudo-machine. It processes the qualifications to locate the subtree values to sort and constructs a vector of subtree node pointers and an array of values on which to sort. Display storage is used as "working" storage for this information. It sorts the array of values carrying along subtree pointers and processes the ordered subtree pointers to position (logically) the subtrees in the prescribed order.

Common Blocks:

BOUNDS pseudo-machine control block
INCNTL interpreter fetch/execute control
INDEVS interpreter device unit numbers
INGLOB interpreter pseudo-machine storage
INSTAC run-time & stack barrier stack

SHELLM Multiple field extended SHELL sort

Purpose:

This subroutine performs a multiple field sort in support of the PLANS ORDER feature.

Parameters:

N number of columns of SORTF
NPROP number of rows of SORTF
TAG vector of tags to be ordered

SORTF   array of sort fields
AD   ascending(1)/descending(0) flag

PREOR   PREORder traversal subsystem

PREOR   PREORder traversal

Purpose:
This subroutine traverses a PLANS tree in preorder.
Parameters:
PNODE   pointer to current node in tree
LEVEL   level of node in tree (input)
           level of next node (output)
PNEXT   pointer to next node in preorder
Common Blocks:
INGLOB   interpreter pseudo-machine storage
TRAVER   preorder traversal stacks

PREOR2   PREORder traversal

Purpose:
This subroutine traverses a PLANS tree in preorder. It is used in comparisons when two trees are being traversed at the same time.
Parameters:
PNODE   pointer to current node in tree
LEVEL   level of node in tree (input)
           level of next node (output)
PNEXT   pointer to next node in preorder
Common Blocks:
INGLOB   interpreter pseudo-machine storage
TRAVER   preorder traversal stacks

READ   READ subsystem

READ   READ instruction

Purpose:
This subroutine implements the PLANS READ statement. All formats are supplied by the interpreter, not the user.
Common Blocks:
INCNTL   interpreter fetch/execute control
INDEVS   interpreter device unit numbers
INGLOB   interpreter pseudo-machine storage
INSTAC   run-time & stack barrier stack

RESERV   RESERVe string space

Purpose:
This subroutine searches the reserved label/value table for a copy of the string and if not found, it has space allocated in dynamic storage.

Parameters:

LENGTH	number of characters in the string
STRPTR	pointer to string to be found or allocated
PSTRIN	pointer to string in reserved label/value table or in dynamic storage

Common Blocks:

BOUNDS	pseudo-machine control block
INCNTL	interpreter fetch/execute control
INGLOB	interpreter pseudo-machine storage

RTREE   Read TREE

Purpose:
This subroutine inputs a PLANS tree structure in standard format and returns a pointer to its root. Standard format uses three column indentation for each node sub-level, '-' separating label from value, '@' for a null label and END in column one following the entire structure.

Parameters:

PROOT	pointer to root node of tree
TLEVEL	tree level

Common Blocks:

BOUNDS	pseudo-machine control block
INCNTL	interpreter fetch/execute control
INDEVS	interpreter device unit numbers
TRAVER	preorder traversal stacks
WORK	record pointers to common work areas

<u>TREQUAL</u>   TREe QUALification subsystem

HARD   HARD qualification instruction

Purpose:
This subroutine uses information on the run-time stack to qualify a "hard" node. A "hard" node means that the node must exist so if it does not, one is created. Note that the barrier stack (BSTAC) points to the first item in the run-time stack for qualification and this must be a tree address. (The barrier stack allows nested qualification to occur.)

Common Blocks:

BOUNDS	pseudo-machine control block
INCNTL	interpreter fetch/execute control

INDEVS interpreter device unit numbers
INGLOB interpreter pseudo-machine storage
INSTAC run-time & stack barrier stack work;

SOFT SOFT qualification instruction

Purpose:
This subroutine uses qualification information on the run-time stack to qualify a "soft" node. A "soft" node means that the node need not exist and if it does not, the result is a null node reference. Note that the barrier stack (BSTAC) points to the first item in the run-time stack for qualification and this must be a tree address.

<u>TRESUP</u> TREe SUPport subsystem

Purpose:
This subsystem contains both support routines for trees and instruction routines which also occasionally serve as support routines.

ALAB Assign LABel

Purpose:
This subroutine assigns a new label to a tree node. Although a string address is required for assignment, the source of the assignment may not be in string form. If so, it is converted.
Common Blocks:
BOUNDS pseudo-machine control block
INGLOB interpreter pseudo-machine storage
INSTAC run-time & stack barrier stack
WORK record pointers to common work areas

ATRE Assign TREe instruction

Purpose:
This subroutine assigns a numeric value, string or (sub)tree to the node of a tree. In the case of a string, copying may be required. In the case of a (sub)tree as the source, copying is required.
Common Blocks:
BOUNDS pseudo-machine control block
INCNTL interpreter fetch/execute control
INGLOB interpreter pseudo-machine storage
INNODE interpreter node storage management
INSTAC run-time & stack barrier stack

CPYTRE CoPY TREe

Purpose:
This subroutine makes a copy of an existing tree.

Parameters:

ORIGIN	address of original tree
NEWADR	address of new tree

Common Blocks:

BOUNDS	pseudo-machine control block
INGLOB	interpreter pseudo-machine storage
TRAVER	preorder traversal stacks

ELMT ELeMenT of instruction

Purpose:
This subroutine determines whether the node (P) on the top of the run-time stack has a subnode which is identical in every respect (including substructure) to the node (Q) next to the top of the stack. If true, a one is put on the top of the stack, otherwise, a zero.

Common Blocks:

BOUNDS	pseudo-machine control block
INGLOB	interpreter pseudo-machine storage
INSTAC	run-time & stack barrier stack

EQ EQual instruction

Purpose:
This subroutine performs the relational operation equal. Each operand on the stack can be a numeric value, a string pointer or a tree node pointer.

Common Blocks:

BOUNDS	pseudo-machine control block
INDEVS	interpreter device unit numbers
INGLOB	interpreter pseudo-machine storage
INSTAC	run-time & stack barrier stack

GRFT GRaFT instruction

Purpose:
This subroutine grafts a "snipped" subtree to a target node.

Common Blocks:

BOUNDS	pseudo-machine control block
INCNTL	interpreter fetch/execute control
INGLOB	interpreter pseudo-machine storage
INNODE	interpreter node storage management
INSTAC	run-time & stack barrier stack

GRIS GRaft InSert instruction

Purpose:
This subroutine does a graft and insert tree

operation.

Common Blocks:

INGLOB	interpreter pseudo-machine storage
INSTAC	run-time & stack barrier stack

IDNT IDeNTical to instruction

Purpose:

This subroutine compares two trees. If they are identical in every way, a one is returned. Otherwise a zero is returned.

Parameters:

PTREE	address of first tree
QTREE	address of second tree
ANSWER	result of comparison

Common Blocks:

BOUNDS	pseudo-machine control block
INGLOB	interpreter pseudo-machine storage
TRAVER	preorder traversal stacks

ISRT InSeRT instruction

Purpose:

This subroutine inserts a (sub)tree or creates a new node with the numeric or string value on the next to top of the run-time stack at the tree address found on the top of the stack.

Common Blocks:

BOUNDS	pseudo-machine control block
INGLOB	interpreter pseudo-machine storage
INSTAC	run-time & stack barrier stack

PRUNE PRUNE instruction

Purpose:

This subroutine prunes the requested node and any substructure and updates the link to the node.

Parameters:

PNODE	index of word containing pointer to node to be pruned

Common Blocks:

BOUNDS	pseudo-machine control block
INCNTL	interpreter fetch/execute control
INDEVS	interpreter device unit numbers
INGLOB	interpreter pseudo-machine storage
INNODE	interpreter node storage management
TRAVER	preorder traversal stacks

SBST SuBSeT of instruction

Purpose:

This subroutine checks if the tree at the address given by the next to the top item on the run-time stack is a subset of the tree at the address given by the top item on the stack. If it is, a one is returned on the stack, otherwise a zero.

Common Blocks:

INGLOB interpreter pseudo-machine storage
INSTAC run-time & stack barrier stack

SNIP SNIP instruction

Purpose:

This subroutine detaches a subtree from its present location in preparation for a following GRFT or GRIS instruction.

Common Blocks:

INGLOB interpreter pseudo-machine storage
INSTAC run-time & stack barrier stack

WRITE WRITE subsystem

MVLBVL MoVe LaBel or VaLue string

Purpose:

This subroutine moves a label or value string from the global common block location identified by STRPTR to the TARGET array.

Parameters:

STRPTR pointer to string
NCHARS number of characters in string
TARGET output array

WRIT WRITe instruction

Purpose:

This subroutine implements the PLANS WRITE statement. All formats are supplied by the interpreter, not the user.

Common Blocks:

INCNTL interpreter fetch/execute control
INGLOB interpreter pseudo-machine storage
INSTAC run-time & stack barrier stack

WTREE Write TREE

Purpose:

This subroutine outputs the PLANS tree in standard indented form. Each node label and value is output on a separate line. Indentation is used to show the level of each node in the tree structure.

Parameters:

PNODE pointer to root node of tree

Common Blocks:

BOUNDS pseudo-machine control block
INCNTL interpreter fetch/execute control

INDEVS	interpreter device unit numbers
INGLOB	interpreter pseudo-machine storage
TRAVER	preorder traversal stacks

APPENDIX I

PSEUDO-MACHINE INSTRUCTION SET

This appendix contains a table of the pseudo-machine instruction set, by category, followed by a detailed description of these instructions also by category.

Symbolic op code	Numeric op code	IFORM entry	
Logical Operations			
OR	1	1	
AND	2	1	
NOT	3	1	
Tree Relation Operations			
ELMT	4	1	ELEMENT OF
IDNT	5	1	IDENTICAL TO
SBST	6	1	SUBSET OF
NULL	7	1	null node or tree reference test
Scalar Relational Operations			
EQ	10	1	equal
LT	11	1	less than
LE	12	1	less than or equal
GT	13	1	greater than
GE	14	1	greater than or equal
NE	15	1	not equal
Scalar Arithmetic Operations			
ADD	16	1	
SUB	17	1	subtract
MULT	18	1	multiply
DIV	19	1	divide
EXP	20	1	exponentiation
NEG	21	1	negation
Assignment Operations			
AI	23	1	assign integer
AF	24	1	assign floating point
ALAB	25	1	assign label

ATRE	26	1	assign tree
AETA	27	1	assign $ELEMENT tree address
ATA	28	1	assign tree address

Other Tree Operations

PRUT	22	2	PRUNE (tree address operand)
SNIP	29	1	detach a subtree from its present location
HQAL	30	1	"hard" qualification
SQAL	31	1	"soft" qualification
SIBL	32	1	sibling reference
GRFT	33	1	GRAFT
PRUN	34	1	PRUNE (run-time stack operand)
ISRT	35	1	INSERT
GRIS	36	1	GRAFT INSERT
LABL	37	1	LABEL function (retrieve label string address)
NUMB	38	1	NUMBER function (count subnodes of current (sub)tree)
ORDR	39	1	sort subtree according to specified properties

Delayed Subscript Qualification Operations

FRST	40	1	first
LAST	41	1	
NEXT	42	1	

Elementary Transfer of Control Operations

JMP	45	6	jump
JMPT	46	6	jump true
JMPF	47	6	jump false

Other Transfer of Control Operations

JMPN	48	6	jump if top of stack is null node reference
JPLE	49	6	jump on iterative loop end

Indexing Operations

IDXA	50	1	index to an address

IDXV	51	1	index to a value

Elementary Stack Loading Operations

SWAP	53	1	swap the top two stack items
DUP	54	1	duplicate the stack top with a copy of the current stack top
LSA	55	3	load string address
LIC	56	5	load integer constant
LFC	57	4	load floating point constant
LIF	58	1	load infinity (floating point)
LIA	59	2	load integer variable address
LIAI	60	2	load integer variable address indirectly
LIV	61	2	load integer variable value
LIVI	62	2	load integer variable value indirectly
LFA	63	2	load floating point variable address
LFAI	64	2	load floating point variable address indirectly
LFV	65	2	load floating point variable value
LFVI	66	2	load floating point variable value indirectly
LID	67	2	load integer array descriptor address
LIDI	68	2	load integer array descriptor address indirectly
LFD	69	2	load floating point array descriptor address
LFDI	70	2	load floating point array descriptor address indirectly
LNTA	71	1	load $NULL tree address
LTA	72	2	load tree address
LTAI	73	2	load tree address indirectly
LETA	74	1	load $ELEMENT tree address
LCMB	75	1	load combinatorial tree address

Conversion Operations

CVI	76	1	convert to integer
CVF	77	1	convert to floating point

Operations Affecting Display Storage

CALL	79	7	procedure entry
BENT	80	7	block entry
EXIT	81	1	block or procedure exit

ALLA	82	7	array space allocation

String Relational Instructions

SEQ	83	1	string equal
SLT	84	1	string less than
SLE	85	1	string less than or equal
SGT	86	1	string greater than
SGE	87	1	string greater than or equal
SNE	88	1	string not equal

I/O Operations

GET	90	7	
PUT	91	7	
READ	92	7	
WRIT	93	7	
WCMP	94	7	write compressed
RDBN	95	7	read binary
WRBN	96	7	write binary

Combinatorial Operations

FCMB	100	1	first combination
NCMB	101	1	next combination
FRPM	102	1	first permutation
NPRM	103	1	next permutation

String Related Operations

AS	108	1	assign string
CVS	109	1	convert to string
LSVA	110	2	load string variable address
LSVI	111	2	load string variable address indirectly
LSD	112	2	load string array descriptor address
LSDI	113	2	load string array descriptor address indirectly
SCAT	114	1	string concatenation
SIND	115	1	string index
SLEN	116	1	string length
SSUB	117	1	string substring
SVER	118	1	string verify

Miscellaneous Operations

STMT	121	7	statement number
ASRT	122	7	assertion debugging test
TLOW	123	1	set trace low if master trace switch is on
THGH	124	1	set trace high if master trace switch is on
TOFF	125	1	turn off tracing
STOP	126	7	stop interpretation

An Explanation of the Operations

Logical Operations

The obvious interpretation of run-time stack items (logical values) is applied. No conversion is required. Improper run-time stack values represent a system error.

Tree Relation Operations

See the PLANS User's Guide [6] for legitimate combinations of tree and scalar references and the meaning for the first three tree relation operations. The top two items on the run time stack represent the two operands. With normal tree operands, subtypes 1, 2 and 4 may occur. Note that when comparing values of leaf nodes, mismatching data types may require data conversion.

Scalar Relational Operations

The obvious binary meaning is applied. An operand which is a tree address requires value extraction. Mismatched data types imply conversion. For all relational operations except EQ and NE, the operands are numeric (either integer or floating point). For EQ and NE string comparisons are possible. (If both operands are string variables and/or string constants however, string relational operations are used.) Tree operands require value extraction and type testing to determine the proper comparison mode. If either of the operands in the source implies numeric mode, the conversion of the other, if necessary, will have been performed by a convert to floating point (CVF) pseudo-instruction.

Scalar Arithmetic Operations

The obvious binary operator meaning is applied. An operand which is represented by a tree address requires value extraction. Mismatching data types may occur but only in a numeric mode. (There may be integer to floating point conversion but that is all. Other conversions of string or tree values will have been performed by a convert to floating point (CVF) operation.)

Assignment Operations

For AI and AF the top of the run-time stack is guaranteed to be an integer or floating point number. A conversion (CVI or CVF) will have been performed immediately preceding one of these instructions, if necessary. The other assignment instructions, in general, require testing to determine conversion requirements of the source of the assignment (top of the run-time stack). The item next to the top in the run-time stack contains the address of the target of the assignment.

ALAB - assign label

This operation requires a string address for assignment but the source of the assignment may not be in string form. Thus, some work may be required to get the address of the string.

The top of the stack may contain a numeric value, an absolute string address, or a tree address. If the value to be used as the source of the assignment is numeric, it is converted to string form and stored in dynamic storage. The pointer to this storage is placed in the label pointer field of the target node (minus string address base). If the source is already a string, it is checked for location. A string in the reserved label/value table or string constant space is not copied. A copy of the pointer to it is stored in the label pointer field of the target node (minus string address base). If the source string is in dynamic storage, then a copy of the string is made in dynamic storage and the pointer to this new copy is used for the label pointer (minus string address base). If the top of the stack is a tree address, the value of the identified node is retrieved.

Next to the top is the tree address of the target of the assignment (or rather the address of the word which contains a pointer to the target node). If the target node contains a label, it is deleted. If the string pointer (relative to the string address base) points to a reserved label/value string or a string constant, the string pointer is zeroed out. If it points to a string in dynamic storage, the space is released.

The run-time stack and stack barrier stack must be empty upon completion.

ATRE - assign tree

The top of the run-time stack represents the source of the assignment and may be a tree address, a numeric value or a string address. If it is a tree address, it may have a null subtype.

If the source is a tree address, it is tested for null subtype (4) or subtype 1 and zero node pointer. In either case nothing remains to be done except stack popping. Otherwise, it is traversed and copied in its entirety (new nodes, values and pointers). Any string pointers for labels or values are examined for location. If in label/value or constant space, only the pointers are copied. If in dynamic storage, then a new copy of the string is made (in dynamic storage) and the pointer of the copy used.

If the source is a numeric value, then the numeric value is copied into the second word of the target node and the type field in the word node is set accordingly.

If the source is a string address, then it is checked for location. If it is in dynamic storage, a copy of the string is made (in dynamic storage) and this string address is used. The string address is placed in the second word of the target node and the type field in the first word of the target node is set accordingly.

The next to the top of the stack is a tree address of the target of the assignment. The node identified requires pruning of any value or substructure. If the global "replace label switch" is on, the label of this node is deleted as well.

The target node is guaranteed to exist - it cannot be null. (This is a result of "hard" qualification.) The top two stack items are popped and the stack must then be empty.

ATA - assign tree address

This operation is only used to update tree pointer variables which must be located in the display. Since it is used only to support assignment of a tree address to a PLANS tree pointer variable, it may be a full word assignment. No concern for the number of bits of the assignment is required.

The top of the stack must contain a tree address and must not be zero. The next to the top of the stack must be a tree address and must be in the display. The address on the top of the stack is stored as is (full word) at the pseudo-machine word identified by the stack item next to the top.

Both the run-time stack and the stack barrier stack must be set to null upon completion.

Other Tree Operations

PRUT - prune a tree whose address is computed by instruction fetching
PRUN - prune a (sub)tree whose address is on the run-time stack

For PRUN the address is immediately removed from the run-time stack and is stored in the FORTRAN variable used for operand addresses.

If the word addressed by the operand address has a zero (right-most 15 bits) then nothing is done. Otherwise, the content represents an absolute binary tree node address. This (sub)tree is pruned in its entirety. All binary tree nodes are placed on the availability list and all strings addressed in dynamic storage are released. The word containing the absolute binary tree node address is set to the sibling pointer of the binary tree node addressed (right-most 15 bits to right-most 15 bits).

SNIP - detach a subtree

A subtree is detached from its present location in preparation for a following GRFT (graft) or GRIS (graft insert) instruction.

The top of the run-time stack contains a tree address (type 1). If a null subtype (4) or a subtype 1 with zero value is at the top of the stack then a new node with no label and a null descendant is allocated. Its absolute binary tree node address is put on the run-time stack in place of the tree address there. A subtype code of 2 is required.

If a non-null subtree exists, the word pointed at by the tree address has its value retrieved and this replaces the tree address on top of the stack. (It is an absolute binary tree node address of subtype 2.) The word containing this binary tree node pointer is then updated with the binary tree node sibling pointer unless the binary tree node pointer is zero.

Example:

Before interpretation
- tree address on stack - 1000 (type 1, subtype 1)
- word 1000 (right 15 bits) - 1234
- word 1234 (right 15 bits) - 2311

After interpretation
- top of stack - 1234 (type 1, subtype 2)
- word 1000 (right 15 bits) - 2311

HQAL - hard qualification

This operation references a context in which a node must exist. If it does not exist, one is created (e.g. assignment).

SQAL - soft qualification

This operation references a context in which a node need not exist (e.g. source of an assignment).

SIBL - sibling reference

This operation is only used for updating a tree pointer variable in the sense of advancing one link down a linked list.

The address at the top of the stack must be a display address with its value pointing to another word, unless the value is already zero. The value at this "other word" is stored back in the display word, unless this new value is zero, in which case nothing is done. The value extracted from the "other word" must come from the right-most 15 bits because, in general, it will be the first word of a tree node which contains a sibling pointer.

The stack must be empty upon completion.

GRFT - GRAFT tree operation

This operation grafts a snipped subtree to a target node. The top of the run-time stack contains the target tree address and the next to top contains the source of the graft. (Note that the target address being on top differs from the convention used for the other operations.)

The source for the graft must be a tree address of subtype 2 and the absolute binary tree node address on the stack must be positive.

If the global "label replace" flag is on, then an existing label from the target node is removed. If the target node has any substructure then it is pruned. If it has a value and it is a string address then it is removed. (If any of the string addresses are in dynamic storage then the storage is released.)

If the global "replace label" flag is on then the source node label pointer is copied to the target node label pointer. The source node type code and word 2 contents is copied to the target node type code and word 2 contents. The source node is put on the binary tree node availability list.

The top two items are popped from the run-time stack.

ISRT - insertion tree operation

The top of the run-time stack must contain a tree address (type 1, subtype 1) and this is the target of the operation. Next to the top of the run-time stack is the source of the operation. This may be a tree address (type 1, subtype 1 or 4), a numeric value or a string address.

If the source is a tree address then it is checked for a null subtype (4) or a zero value of subtype 1. For either of these, a new binary tree node, with a zero label pointer, a zero descendant pointer and corresponding type field, is allocated. Otherwise, the subtree is copied in its entirety. The address of either new node is substituted for the next to top of the run-time stack item and is coded with a subtype of 2. A jump to GRIS (GRAFT INSERT tree operation) interpretation code is then made.

If the source is a numeric value or string address, a new binary tree node is allocated. The label pointer is set to zero and the type code in the node is set according to the source type. The second word of the node is filled in with the numeric value or the string address. (If the latter is in dynamic storage, a copy is made in dynamic storage and the new address is used.) The new binary tree node address, with a type of 1 and subtype of 2, replaces the entry in the next to top of the stack position. A Jump to GRIS instruction interpretation is then effected.

GRIS - GRAFT INSERT tree operation

The top of the run-time stack must contain a tree address (type 1, subtype 1) and this represents the target of the operation. The source of the operation is the next to the top entry and must be an absolute binary tree node address (type 1, subtype 2).

The node is inserted in a linked list before the target node address and requires only binary tree node pointers to be updated (right-most 15 bits). Any subtree detaching required will previously have been performed by a SNIP operation.

The run-time stack is popped upon completion and must then be empty.

Example:

```
     Before interpretation
          top of stack - 1000 (type 1, subtype 1)
          word 1000 (right 15 bits) - 1234
          next to top of stack - 2000 (type 1, subtype 2)
          word 2000 (right 15 bits) - anything
     After interpretation
          word 1000 (right 15 bits) - 2000
          word 2000 (right 15 bits) - 1234
```

LABL - LABEL function

The top of the stack contains a (sub)tree node pointer. The label field is extracted and placed on the run time stack (with the base of string address space added). Note that a null address (0) will result in a pointer to the first word of the reserved label/value table which always has a string of length zero at that location.

NUMB - NUMBER function

This operation is used to count the number of subnodes of a given node by following sibling pointers.

The top of the stack contains a tree pointer. After locating the node identified, its descendant pointer is extracted. If there is no descendant pointer or if the descendant pointer is zero the the resulting count is zero. Otherwise, the count is obtained by following the sibling pointers. The top of the stack is replaced by the integer number value.

Delayed Subscript Qualification Operations

These instructions have a delayed execution. The stack is loaded with a keyword subscript type code (see run-time stack description) for interpretation at the time a hard or soft qualification is done.

Elementary Transfer of Control Operations

These instructions may modify the instruction pointer. The target of a jump is on a full word boundary so the byte indicator of the two word instruction pointer is set to zero.

These are full word instructions of Format type 6. The base address of pseudo-code storage (BCODE) plus the jump instruction address field form the absolute operand address that is the target of the jump. The jump is effected by changing the instruction pointer (IPWORD and IPBYTE).

Whenever a JMP instruction is encountered, the run-time stack should be empty. Whenever a JMPT or JMPF instruction is encountered, only one item, either a 0 or 1 representing logical false or true, should be on the stack. After execution of any of these instructions, the stack must be empty.

Other Transfer of Control Operations

JMPN conditions a jump on the existence of a null tree node reference on top of the run-time stack. JPLE conditions a jump on an iterative loop end condition on the stack. For the latter, the top of the stack contains a numeric increment, the next value down contains the final loop index value and the next value down contains the current loop index value. The direction of the loop end test depends on whether the increment value is positive or negative.

For either of these the stack is popped the requisite number of elements.

Indexing Operations

The run-time stack must previously have been loaded with the address of the array descriptor and the subscript values for addressing. The subscript values on the stack are "guaranteed" to be numeric. The descriptor address loading primitive is responsible for setting the top of the stack barrier stack to point to the run-time stack location of this address. The subscripts are above this in the stack. Upon completion of either of these two instructions, the stack barrier stack is popped.

The result of either of these two operations is a properly coded entry on the run-time stack. It is a function of how the array descriptor address is coded.

See the discussion of descriptor address loading below for further information.

Elementary Stack Loading Operations

LTA - load tree address

As a function of instruction fetching and address computation, an absolute pseudo-machine address is computed and is an address in the display. This address is pushed onto the run-time stack and marked as a tree address of subtype 1. The stack barrier stack is loaded with a pointer to this stack.

LTAI - load tree address indirectly

This instruction is like LTA except the loading is done indirectly. The address computed by instruction fetching is used as a pointer to the word containing the address to be loaded. Full word extraction is acceptable because the word containing the address to be loaded is in the display and is not part of a binary tree node.

Conversion Operations

CVI - convert to integer
CVF - convert to floating point

The top of the run-time stack may be loaded with a tree address, a string address or a numeric value. The result of the interpretation is to replace the top of the stack with an integer numeric value (CVI) or a floating point numeric value (CVF).

In the case of a tree address, the value of the identified node is retrieved. If the node has substructure and no explicit value then, by convention, the value is interpreted to be zero. If the value is a string (string address in the second word of the node), then it is treated as described below for string conversion. If the value of the node is numeric, it is checked for conversion to a different data type. (Coversion of a tree reference to a numeric data type does not necessarily imply data conversion. It may be that the retrieved value is of the proper data type.)

In the case of a string address, the string value is converted to the proper internal numeric form. By convention, a string value which will not convert to numeric form is interpreted as having a numeric value of zero and a warning message results. If the byte count is zero then, by convention, the string converts to a numeric zero.

If the top of the stack is numeric, it should never be coded as integer if CVI is being interpreted and it should never be coded as floating point if CVF is being interpreted.

<u>Operations Affecting Display Storage</u>

CALL - procedure entry
BENT - block entry

CALL/BENT cause local display storage management in that a new activation record with display vector must be allocated. The operand references the procedure/block table. The fixed display storage requirement is determined from this table. In the case of either a procedure entry or a block entry the very first word of display storage is loaded with the table entry. This is followed by the block level pointers for display entries. Then the address of the previously active display is entered followed by the pointer to the display stack top upon entry.

In the case of a procedure entry, the next word contains the return address in pseudo-code address space followed by parameter addresses, if any, which are retrieved from the run-time stack. (The procedure/block table contains the parameter count which may be zero.) The stack should contain nothing but parameter information. A stack entry which is a value due to an actual parameter which is a constant or an evaluated expression, is stored in the variable part of the display and the corresponding parameter address in the fixed part is filled in accordingly.

EXIT - block or procedure exit

This operation causes an exit from a procedure or begin block. In either case, pruning of local trees is done. A list of local tree addresses is on the run-time stack. The stack barrier mechanism is used to indicate the first item on the list. Interrogation of the procedure/block table according to the currently active block indicates whether a procedure return is required. In either case, local display storage management is required because the previously active storage area must be restored and the display storage stack top must be restored to its value upon block entry.

If the exit is from the main procedure, DONE is set to TRUE and subroutine INTFE is exitted.

ALLA - array space allocation

This operation causes the descriptor for an array, stored in the fixed part of the display, to be computed according to subscript information on the stack. (This information is preceded by the descriptor address which has been loaded with a descriptor loading instruction. The stack barrier identifies the location on the stack of this item.) The space for the array is allocated in the variable part of the display and the local display stack top is adjusted accordingly. (Array space is allocated separately from fixed display storage because of variable bounds capabilities.) The operand field for this instruction specifies the number of dimensions and is recorded as the second word of the descriptor.

String Relational Operations

The obvious binary meaning is applied and both operands are guaranteed to be of string type.

I/O Operations

Input/output list information is on the run-time stack, last list item on top.

Combinatorial Operations

These instructions are also responsible for setting the top of the run-time stack with a logical value to indicate whether generation of permutations or combinations has been completed. Also display storage management for the list of combinatorial subtrees is required.

String Related Operations

AS - assign string

The top of the run-time stack is guaranteed to be a string because a conversion (CVS) will have been performed immediately before, if necessary. The next to the top item on the run-time stack contains the address of the target of the assignment.

CVS - convert to string

The top of the run-time stack may be loaded with a tree address or a numeric value. The result of the interpretation is to replace the top of the stack with a pointer to the newly created string.

In the case of a tree address, the value of the identified node is retrieved. If the node has substructure and no explicit value then the value is interpreted to be the null string (the first string in the reserved label/value table). If the value is a string then nothing is done. Otherwise, it is treated in the same manner as a numeric value.

An integer is converted to a string of length 12 while a floating point number is converted to a string of length 16. The new string is created in dynamic storage with a type of 2 and subtype of 4 (temporary).

SCAT - string concatenation

The top two items on the run time stack contain the addresses of the strings to be joined with the top string coming last. Both are guaranteed to be strings because conversion (CVS) will have been performed just before, if necessary. A new string is created in dynamic storage with a type of 2 and subtype of 4 (temporary). A pointer to this string is placed on the run-time stack.

If either of the two source strings had a subtype of 4 their space in dynamic storage is released.

SIND - string index

The next to the top item on the stack contains a pointer to the string which is to be searched for the first occurrence of the string whose pointer is contained on the top of the stack. These two items are guaranteed to be strings because of conversion (CVI) just prior to this instruction, if necessary. They are popped from the top of the stack and replaced by the integer value which represents the starting location of the top string within the next to top string.

SLEN - string length

The top item on the run-time stack is guaranteed to be a string address because of conversion (CVS) just preceding, if necessary. This item is replaced by the integer value which is the length of the source string.

SSUB - string substring

The top item on the run-time stack contains the integer ending location for the substring operation and the next to top item contains the integer starting location. The next item down contains the address of the string upon which the substring operation is to take place. All are guaranteed to be of the proper type because of conversion (CVI and CVS) which will have taken place just prior, if necessary. A new string is created in dynamic storage with a type of 2 and subtype of 4 (temporary). The top three items are popped from the run-time stack and the address of the new string just created is placed on the top.

If the starting or ending values are negative, the number is replaced by 1. If the starting value is greater than the ending value, a null string is returned. If the ending value is greater than the length of the string, it is replaced by the length of the string value.

SVER - string verify

Each character in the string pointed to by the item in the next to top position in the run-time stack is examined to see if it is contained in the string to which the top item in the run-time stack points. Both are guaranteed to be strings because of conversion (CVI) which would have taken place just prior to the execution of this instruction, if necessary. These two items are popped from the stack and replaced by the integer value 0 if all characters are represented. Otherwise, it is replaced by the integer value which is the index of the first character in the next to top string that is not represented in the top string.

Miscellaneous Operations

ASRT - assertion debugging test

The ASRT statement outputs the PLANS statement number if the logical value at the top of the run-time stack is false and pops the top item from the stack.

STMT - statement

STMT instructions are generated by the translator if the STMT option was selected at compile time. (The option should not be used indiscriminately because about 20% of the code may then be such instructions.) The operand is a literal operand containing the statement number of the corresponding PLANS statement. This value is recorded in a global location so detected run-time errors can be "tagged" with the source statement number. It is also used if tracing is selected.

STOP - stop interpretation

The STOP statement outputs the statement number and procedure name where the stop occurs before setting the DONE switch to TRUE and exitting from subroutine INTFE.

APPENDIX J

INTERPRETER PDL'S FOR THE CHARACTER STRING EXTENSION

Translator PDL's

STEXPR string expression

This routine parses the concatenation rule for strings.

```
string_expr:=arith_expr $(cat_op arith_expr)
```

The general stack contains the type of the first operand upon entry to this rule. This is replaced by the string type if the concatenation operator is found.

```
Invoke arith_expr rule.
IF error THEN
      Return.
ENDIF.
DO WHILE new symbol is concatenation symbol.
     Pop stack to OPR.
     IF OPR not string THEN
          Generate convert to string instruction.
     ENDIF.
     Get new symbol.
     Invoke arith_expr rule.
     IF error THEN
          Generate missing or erroneous expression error
            message.
          Return.
     ENDIF.
     Pop stack to OPR.
     IF OPR not string THEN
          Generate convert to string instruction.
     ENDIF.
     Push string type onto stack.
     Generate concatenation instruction.
ENDDO.
```

STFN string function

This routine parses the string functions.

```
string_function:=expr '(' expr ':' expr ')'
          | LENGTH '(' expr ')'
          | INDEX '(' expr ',' expr ')'
          | VERIFY '(' expr ',' expr ')'
```

Upon entry the general stack contains the operator of the function being parsed at the top. It is replaced by the result type on exit.

```
IF symbol is opening parenthesis THEN
     Get new symbol.
ELSE
     Generate missing opening parenthesis error message.
ENDIF.
Invoke expression rule.
IF error THEN
     Pop 2 items from stack.
     Return.
ENDIF.
Pop stack to TYPE.
Pop stack to TEMPOP.
IF substring function THEN
     IF TYPE not integer THEN
          Generate convert to integer instruction.
     ENDIF.
     IF new symbol is colon THEN
          Get new symbol.
     ELSE
          Generate missing colon error message.
     ENDIF.
ELSE
     IF TYPE not string THEN
          Generate convert to string instruction.
     ENDIF.
     IF not length function THEN
          IF new symbol is comma THEN
               Get new symbol.
          ELSE
               Generate missing comma error message.
          ENDIF.
     ENDIF.
ENDIF.
IF not length function THEN
     Invoke expression rule.
     IF substring function THEN
          IF TYPE not integer THEN
               Generate convert to integer instruction.
          ENDIF.
     ELSE
          IF TYPE not string THEN
               Generate convert to string instruction.
          ENDIF.
     ENDIF.
ENDIF.
Generate whatever instruction is in TEMPOP.
IF new symbol is closing parenthesis THEN
     Get new symbol.
ELSE
     Generate missing closing parenthesis error message.
ENDIF.
```

Interpreter PDL's

INTFE interpreter fetch/execute

This is the main fetch/execute routine from which calls are generated to subroutines which handle the details of the more involved instructions. Since this routine already exists the changes are in FORTRAN 77 rather than in PDL form.

```
C  SEQ - STRING EQUAL
8300  CONTINUE
      TOPSTA=TOPSTA + 1
      IF (TOPSTA .GT. MAXSTA) THEN
C
C--STACK OVERFLOW
         CALL ERNUM(6,2)
         GO TO 50000
      ENDIF
      STACK(TOPSTA)=0
      CALL SEQ
      GO TO 50000
C
C  SLT - STRING LESS THAN
8400  CONTINUE
      TOPSTA=TOPSTA + 1
      IF (TOPSTA .GT. MAXSTA) THEN
C
C--STACK OVERFLOW
         CALL ERNUM(6,2)
         GO TO 50000
      ENDIF
      STACK(TOPSTA)=-1
      CALL SEQ
      IF (OPCODE .EQ. 87) GO TO 300
      GO TO 50000
C
C  SLE - STRING LESS THAN OR EQUAL (NOT GREATER THAN)
8500  CONTINUE
C
C  SGT - STRING GREATER THAN
8600  CONTINUE
      TOPSTA=TOPSTA + 1
      IF (TOPSTA .GT. MAXSTA) THEN
C
C--STACK OVERFLOW
         CALL ERNUM(6,2)
         GO TO 50000
      ENDIF
      STACK(TOPSTA)=1
      CALL SEQ
```

```
      IF (OPCODE .EQ. 85) GO TO 300
      GO TO 50000
C
C  SGE - STRING GREATER THAN OR EQUAL (NOT LESS THAN)
8700  CONTINUE
      GO TO 8400
C
C  NE - NOT EQUAL
8800  CONTINUE
      TOPSTA=TOPSTA + 1
      IF (TOPSTA .GT. MAXSTA) THEN
C
C--STACK OVERFLOW
         CALL ERNUM(6,2)
         GO TO 50000
      ENDIF
      STACK(TOPSTA)=0
      CALL SEQ
      GO TO 300
C
C  AS - ASSIGN STRING INSTRUCTION
10800 CONTINUE
      CALL AS
      GO TO 50000
C
C  CVS - CONVERT TO STRING INSTRUCTION
10900 CONTINUE
      CALL CVS
      GO TO 50000
C
C  LSVA - LOAD STRING VARIABLE ADDRESS INSTRUCTION
11000 CONTINUE
      TOPSTA=TOPSTA + 1
      IF (TOPSTA .GT. MAXSTA) THEN
C
C--STACK OVERFLOW
         CALL ERNUM(6,2)
         GO TO 50000
      ENDIF
      STACK(TOPSTA)=OPADDR
      TYPE(TOPSTA)=2
      SUBTYP(TOPSTA)=1
      IF (OPADDR .GE. BSTRIN) SUBTYP(TOPSTA)=2
      IF (OPADDR .GE. BDYNAM) SUBTYP(TOPSTA)=3
      GO TO 50000
C
C  LSVI - LOAD STRING VARIABLE ADDRESS INDIRECTLY
C  INSTRUCTION
11100 CONTINUE
      TOPSTA=TOPSTA + 1
      IF (TOPSTA .GT. MAXSTA) THEN
C
C--STACK OVERFLOW
         CALL ERNUM(6,2)
```

```
         GO TO 50000
      ENDIF
      STACK(TOPSTA)=SPACE(OPADDR)
      TYPE(TOPSTA)=2
      SUBTYP(TOPSTA)=1
      IF (OPADDR .GE. BSTRIN) SUBTYP(TOPSTA)=2
      IF (OPADDR .GE. BDYNAM) SUBTYP(TOPSTA)=3
      GO TO 50000
C
C  LSD - LOAD STRING DESCRIPTOR INSTRUCTION
11200 CONTINUE
      TOPSTA=TOPSTA + 1
      IF (TOPSTA .GT. MAXSTA) THEN
C
C--STACK OVERFLOW
         CALL ERNUM(6,2)
         GO TO 50000
      ENDIF
      STACK(TOPSTA)=OPADDR
      TYPE(TOPSTA)=3
      SUBTYP(TOPSTA)=5
C
C--SET BARRIER STACK
      BTOP=BTOP + 1
      BSTAC(BTOP)=TOPSTA
      GO TO 50000
C
C  LSDI - LOAD STRING DESCRIPTOR INDIRECTLY INSTRUCTION
11200 CONTINUE
      TOPSTA=TOPSTA + 1
      IF (TOPSTA .GT. MAXSTA) THEN
C
C--STACK OVERFLOW
         CALL ERNUM(6,2)
         GO TO 50000
      ENDIF
      STACK(TOPSTA)=SPACE(OPADDR)
      TYPE(TOPSTA)=3
      SUBTYP(TOPSTA)=5
C
C--SET BARRIER STACK
      BTOP=BTOP + 1
      BSTAC(BTOP)=TOPSTA
      GO TO 50000
C
C  SCAT - STRING CONCATENATION INSTRUCTION
11400 CONTINUE
      CALL SCAT
      GO TO 50000
C
C  SIND - STRING INDEX INSTRUCTION
11500 CONTINUE
      CALL SIND
      GO TO 50000
```

```
C
C  SLEN - STRING LENGTH INSTRUCTION
11600 CONTINUE
      IF (TOPSTA .LT. 1) THEN
C
C--STACK UNDERFLOW
         CALL ERNUM(6,1)
         GO TO 50000
      ENDIF
      STACK(TOPSTA)=SPACE(STACK(TOPSTA))
      TYPE(TOPSTA)=5
      SUBTYP(TOPSTA)=1
      GO TO 50000
C
C  SSUB - STRING SUBSTRING INSTRUCTION
11700 CONTINUE
      CALL SSUB
      GO TO 50000
C
C  SVER - STRING VERIFY INSTRUCTION
11800 CONTINUE
      CALL SVER
      GO TO 50000
```

<u>SEQ</u> string equal

This PDL describes the string comparison routine invoked from INTFE. The top of the stack contains the type of comparison result required - less than, equal or greater than. The next item down contains the pointer to the to the second string and the next item down contains the pointer to the first string. These are popped and replaced by the Boolean result.

```
IF stack underflow can occur below THEN
     Generate stack underflow error message.
     Return.
ENDIF.
Pop stack to CHOICE.
Pop stack to PTR2.
Pop stack to PTR1.
I<--1.
DO WHILE I<=length of first string at PTR1 and
  I<=length of second string at PTR2.
     IF character at position (PTR1 + I) >
       character at position (PTR2 + I) THEN
          IF CHOICE is greater than THEN
               Push TRUE onto stack.
          ELSE
               Push FALSE onto stack.
          ENDIF.
```

```
          Return.
     ELSE IF character at position (PTR1 + I) <
       character at position (PTR2 + I) THEN
          IF CHOICE is less than THEN
               Push TRUE onto stack.
          ELSE
               Push FALSE onto stack.
          ENDIF.
          Return.
     ELSE
          I<--I + 1.
     ENDIF.
ENDDO.
IF CHOICE is equal and length of first string =
  length of second string THEN
     Push TRUE onto stack.
ELSE IF CHOICE is greater than and length of first string >
  length of second string THEN
     Push TRUE onto stack.
ELSE IF CHOICE is less than and length of first string <
  length of second string THEN
     Push TRUE onto stack.
ELSE
     Push FALSE onto stack.
ENDIF.
```

<u>AS</u> assign string

This PDL describes string assignment. The top of the stack contains the pointer to the source string while the next to the top of the stack contains the target address.

```
IF stack underflow can occur below THEN
     Generate stack underflow error message.
     Return.
ENDIF.
Pop stack to PTR.
Pop stack to ADDRES.
IF ADDRES not in display THEN
      Generate stack addressing error message.
      Return.
ENDIF.
IF string at PTR in dynamic storage THEN
     Get space in dynamic storage for copy of string.
     Copy string.
     Set PTR to point to new string.
ENDIF.
Store PTR in display location ADDRES.
```

CVS convert to string

This PDL describes the conversion of an integer, floating point or tree node value to string. The top of the stack contains an integer or floating point number or a tree node pointer. It is replaced by a pointer to the string generated.

```
IF stack underflow can occur below THEN
     Generate stack underflow error message.
     Return.
ENDIF.
Pop stack to VALUE.
IF VALUE type is tree THEN
     IF subtype not 1 or 4 THEN
          Generate wrong data subtype error message.
          Return.
     ENDIF.
     IF VALUE not null tree node reference THEN
          Set VALUE to point to its sibling.
     ENDIF.
     IF VALUE null tree node reference or has descendant
       pointer THEN
          Push pointer to null string onto stack.
          Set its type to string.
          Set its subtype to reserved label/value.
          Return.
     ENDIF.
     Set VALUE to the value in its tree node.
     IF type of VALUE is string THEN
          Push VALUE onto the stack.
          Set its type to string.
          Set its subtype to 1, 2 or 3 depending upon its
               location.
          Return.
     ENDIF.
     Set type to the VALUE type.
ENDIF.
IF VALUE type is integer THEN
     Get space in dynamic storage for string of length 12.
     Convert VALUE to string of length 12 and place in new
       string location.
ELSE
     Get space in dynamic storage for string of length 16.
     Convert VALUE to string of length 16 with 4 decimal
       digits and place in new string location.
ENDIF.
Push new string pointer onto stack.
Set type to string.
Set subtype to dynamic storage.
```

SIND string index function

This PDL describes the index function which is of the form INDEX(S1, S2). The top of the stack contains a pointer to the second string while the next to top of the stack contains a pointer to the first string. These are popped from the stack and replaced by the integer value indicating the position of the leftmost occurrence of the second string in the first string. If either string is null or the second string does not occur in the first, they are replaced by zero.

```
IF stack underflow can occur below THEN
     Generate stack underflow error message.
     Return.
ENDIF.
Pop stack to PTR2.
Pop stack to PTR1.
I<--1.
DO UNTIL result is pushed onto stack.
     IF string starting at position (PTR1 + I) =
       second string THEN
          Push I onto stack.
     ELSE
          IF I > (length of first string - length of second
            string) THEN
               Push 0 onto stack.
     ELSE
          I<--I + 1.
     ENDIF.
ENDDO.
IF subtype of first string is temporary THEN
     Release its space in dynamic storage.
ENDIF.
IF subtype of second string is temporary THEN
     Release its space in dynamic storage.
ENDIF.
```

SSUB string substring function

This PDL describes the substring function which is of the form S1(I1 : I2). The top of the run-time stack contains the integer limit value I2. The next to the top of the run-time stack contains the integer value I1. The next item down contains a pointer to the string S1. These items are popped from the stack and are replaced by the pointer to the new string created.

```
IF stack underflow can occur below THEN
```

```
     Generate stack underflow error message.
     Return.
ENDIF.
Pop stack to HIGH.
Pop stack to LOW.
Pop stack to PTR.
IF HIGH < LOW THEN
     Push pointer to null string onto stack.
     Set the subtype to reserved label/value.
ELSE
     LOW <-- maximum (LOW , 1).
     HIGH <-- minimum (HIGH , string length).
     Get space in dynamic storage for string of length
       (HIGH - LOW + 1).
     Set length of new string to (HIGH - LOW + 1).
     Copy the characters from position (PTR + LOW) to
       (PTR + HIGH) inclusive into the new string.
     Push the pointer to the new string onto the stack.
     Set the subtype to temporary.
ENDIF.
IF subtype of source string is temporary THEN
     Release its space in dynamic storage.
ENDIF.
```

<u>SVER</u>   string verify function

This PDL describes the verify function which is of the form VERIFY(S1 , S2). The top of the stack contains a pointer to the second string while the next to top of the stack contains a pointer to the first string. These are popped and replaced by the integer value zero if each of the characters in the first string occur in the second. Otherwise, they are replaced by the integer indicating the leftmost character of the first string which does not occur in the second.

```
IF stack underflow can occur below THEN
     Generate stack underflow error message.
     Return.
ENDIF.
Pop stack to PTR2.
Pop stack to PTR1.
DO UNTIL result is pushed onto stack.
     I<--1.
     J<--1.
     IF character at position (PTR1 + I) =
       character at position (PTR2 + J) THEN
          IF I = length of first string THEN
               Push 0 onto stack.
          ELSE
               I<--I + 1.
```

```
                J<--1.
            ENDIF.
        ELSE IF J = length of the second string THEN
            Push I onto stack.
        ELSE
            J<--J + 1.
        ENDIF.
    ENDDO.
    IF subtype of first string is temporary THEN
        Release its space in dynamic storage.
    ENDIF.
    IF subtype of first string is temporary THEN
        Release its space in dynamic storage.
    ENDIF.
```

VITA

Carol Anne Samuel

Candidate for the Degree of

Master of Science

Thesis: A SOFTWARE DESIGN FOR THE PROGRAMMING LANGUAGE PLANS

Major Field: Computing and Information Sciences

Biographical:

Personal Data: Born in Montreal, Canada, August 18, 1946, the daughter of Mr. and Mrs. A. Shostak.

Education: Graduated from Outremont High School, Montreal, Canada, in May, 1963; enrolled in the Bachelor of Science program at McGill University, Montreal, Canada, 1963-1965; received Bachelor of Arts degree in Mathematics from the University of Rochester in May, 1967; completed requirements for the Master of Science degree at Oklahoma State University in July, 1982.

Professional Experience: Work/study student, Computing Center, University of Rochester, 1966-1967; programmer/analyst, Eastman Kodak Company, 1967-1969; graduate teaching assistant, Department of Computing and Information Sciences, Oklahoma State University, 1976-1977, 1978-1979; analyst/programmer, City of Stillwater, 1980-1982; analyst/programmer Time Management Software Incorporated, 1982-.