LINEAR LIBRARY
C01 0068 1205

3 A10.
5

University of Cape Town

Department of Computer Science

The Implementation of a Generalized Table Driven Back End Processor

by

Christopher Frank Broadbent

A thesis

prepared under the supervision of

Professor K. J. MacGregor

in fulfilment of the requirements for the

degree of Master of Science in Computer Science

March 1987

The copyright of this thesis vests in the author. No quotation from it or information derived from it is to be published without full acknowledgement of the source. The thesis is to be used for private study or non-commercial research purposes only.

Published by the University of Cape Town (UCT) in terms of the non-exclusive license granted to UCT by the author.

University of Cape Town

ABSTRACT

This thesis discusses the University of Cape Town implementation of a table driven back end processor. The back end processor takes as input an intermediate tree representation of a high level programming language. It produces as output an object text ready for assembly. The specifications of the input tree and the output object are supplied to the back end processor via two tables.

The initial motivation for this project was the need to provide a back end processor capable of taking the DIANA tree output of the University of Cape Town front end processor and producing a corresponding P-code object.

The University of Cape Town back end processor is implemented using Pascal and C in a Unix V environment.

ACKNOWLEDGEMENTS

Many thanks must go to Professor K. J. MacGregor (head of the department of Computer Science at the University of Cape Town) for his patience and assistance in the completion of this thesis.

CONTENTS

1 INTRODUCTION 1

2 BACKGROUND INFORMATION 4

2.1 DIANA 4

2.1.1 Background and History 4

2.1.2 The U.C.T. Front End Processor 5

2.1.3 Abstract Syntax Trees 6

2.1.4 U.C.T. DIANA Implementation 7

2.2 U.C.S.D. P-machine and P-code 12

2.2.1 Background and History 13

2.2.2 P-machine Operation 14

2.3 Comparison of Compilation Techniques and the U.C.T. System 17

2.4 Other Table Driven Back End Processors 21

3 STRUCTURE OF THE ADA COMPILER 28

3.1 Development History 28

3.2 Back End Processor Components 29

3.2.1 Tables 31

3.2.2 Translators 32

3.2.3 File Converters 32

3.2.4 Code Generator 33

4 THE TABLES AND TREE 35

4.1 Tree Specification Table 36

4.2 Code Template Table 42

4.3 Tree 46

5 TRANSLATORS AND CONVERTERS 52

5.1 Tree Specification Table Translation 52

5.1.1 Input File 52

5.1.2 Translator 53

5.2 Code Template Table Translation 57

5.2.1 Input File 57

5.2.2 Translator 57

5.3 Tree Conversion 62

6 THE CODE GENERATOR 64

6.1 Tree and Table Input 65

6.1.1 Tree Specification Table Input 65

6.1.2 Code Template Table Input 66

6.1.3 Tree Input 67

6.2 Important Data Structures 68

6.2.1 Symbol Table 68

6.2.2 Native Type Identifier Store 71

6.2.3 Current Identifier Store 73

6.2.4 User Flags 74

6.2.5 User Stores 75

6.2.6 History Array and Index 76

6.2.7 Comparison Store 77

6.2.8 Offset Stores 78

6.2.9 Procedure Count Store 79

6.2.10 Control Flags 79

6.2.11 Stash 80

6.3 Conditional Structures 81

6.3.1 Special Command Words 82

6.3.2 Type Conditional 85

6.3.3 Lexical Level Conditional 86

6.3.4 User Flag Conditional 87

6.3.5 History Store Conditional 90

6.3.6 Comparison Store Conditional 92

6.3.7 Pointer Conditional 95

6.3.8 Boolean Field Conditional 97

6.3.9 Rep String Conditional 99

6.4 Pointer Indicators 101

6.5 Command Words 102

6.5.1 Node 0 Natives Special Command Word 103

6.5.2 Node 0 Offset Initialization 105

6.5.3 Node 0 Procedure Count Initialization 106

6.5.4 Node 0 Start Node and Null Pointer Setup 106

6.5.5 Node 0 Global Level Initialization 107

6.5.6 Symbol Table Commands 108

6.5.7 Current Identifier Store Commands 110

6.5.8 User Flag Commands 112

6.5.9 User Store Commands 113

6.5.10 History and Index Commands 116

6.5.11 Comparison Store Commands 118

6.5.12 Offset Store commands 120

6.5.13 Procedure Count Commands 122

6.5.14 Stash and Restore Commands 123

6.5.15 Suspend and Resume Commands 125

6.6 The Code Templates 127

6.6.1 Placement of the Templates 127

6.6.2 The Patterns 128

6.6.3 The Substitute Operators 130

6.7 Traversal of a Tree 133

7 CONCLUSION 148

APPENDICES

A REFERENCES 149

B ADDITIONAL READING 151

C IMPLEMENTED P-CODE 154

C.1 The Main Ada-DIANA Definition P-code Structures 155

C.1.1 Array Definition 156

C.1.2 Record Definition 158

C.1.3 Initializaton at Definition Time 161

C.1.4 Procedure Definition 162

C.1.5 Function Definition 164

C.2 The Main Ada-DIANA Statement P-code Structures 166

C.2.1 Expressions 168

C.2.2 Assignment 169

C.2.3 If Statement 171

C.2.4 Case Statement 173

C.2.5 Loop and Exit Statements 177

C.2.6 Goto Statement 184

C.2.7 Procedure and Functon Calls and Return Statement 185

C.2.8 Blocks 187

C.2.9 Array Element Addressing 188

C.2.10 Record Component Addressing 192

C.2.11 Access (Pointer) Addressing 194

C.3 Additional Standard Procedures 195

C.3.1 The INIT Standard Procedure 196

C.3.2 The DXFR Standard Procedure 198

C.3.3 The SWAPmn Standard Procedure 199

D TABLE LISTINGS 200

E EXAMPLES 278

F ERROR MESSAGES 327

F.1 Tree Specification Table Translator Error Messages 328

F.2 Code Template Table Translator Error Messages 329

F.3 Code Generator Error Messages 330

CHAPTER 1

INTRODUCTION

In the past few years there has been a proliferation of different high level language compilers and computer architectures. The effort required to write, test and debug a compiler for each computer is great. As a result, much research is being done on the automatic generation of compilers from source language and object code definitions. The University of Cape Town is involved in this research.

Conventional compilers usually can be divided into two main functional parts, namely the front end processor and the back end processor. These two components can be produced as independent entities. The front end processor outputs an intermediate representation of a given source code program. The back end processor uses this intermediate representation to generate an equivalent object code. The automation of front end processors is fairly well developed. However, much work still needs to be done on the production of generalized back end processors.

This thesis describes the implementation of the University of Cape Town back end processor project. It describes a table driven back end processor which takes as input an abstract syntax tree. From this tree, the back end processor produces an object code in text form ready for assembly. The information about the structure of the input tree and object code is carried in two tables. The back end processor uses these tables for the translation of the input tree to the output object. If the input tree specification or the target computer is changed, then only the tables need to be modified. Thus the effort required to produce a back end processor for a new target computer is greatly simplified. Similarly, the use of a different input specification can be easily accommodated.

At the University of Cape Town, a front end processor has been produced which takes as input an Ada subset source code program and produces as output a DIANA tree intermediate representation of the program. Ada is a high level programming language developed for the United States Department of Defense. DIANA, developed at Carnegie-Mellon University and the University of Karlsruhe, is an intermediate language designed specifically to represent Ada. A set of tables is provided with the back end processor described in this thesis to enable the production of P-code from DIANA trees. This allows the back end processor to generate P-code from Ada subset source code programs when used with the aforementioned front end processor.

Chapter 2 of this thesis provides background information relevant to the back end processor project. The initial purpose of the project is to convert the DIANA intermediate code to P-code. Discussions of subjects relating to both codes are thus given here. The DIANA discussion includes DIANA's background and history, the U.C.T. front end processor, abstract syntax trees and the U.C.T. implementation of DIANA. The P-code discussion includes the background and history of P-code and the P-machine, as well as the operation of a P-machine. An illustrative example is included in each discussion. An overview of conventional compiler techniques follows. In this overview, a comparison is made between conventional compilation techniques and the separate front end - back end processor approach. Finally, other important work in the field of table driven back end processing is discussed.

Chapter 3 provides an overview of the back end processor system and its operation with the U.C.T. front end processor. A brief development history of the system is also given. The chapter continues with an overview of the tables and modules of the back end processor system and the interaction of these components.

Chapter 4 covers the formats of the tables and trees expected as input by the back end processor system. The syntax of each input table is explained in detail. This is followed by a description of the node and tree structures. Examples are used in order to assist with the descriptions.

Chapter 5 describes the modules of the system which convert the tables into machine readable formats suitable for input to the code generating module. Use of some of the examples given in chapter 4 is continued here.

Chapter 6 is the largest chapter in this thesis. It describes the code generating module, which is the central component of the system. The chapter covers the function and operation of the module. This includes the relevant data structures, how the tables are interpreted and what effects they have on the tree to object conversion process. A simple but comprehensive example of a table driven code generating scan of a tree concludes the chapter.

Chapter 7 is a conclusion. It summarizes the results and expectations of this project.

CHAPTER 2

BACKGROUND INFORMATION

This chapter provides general background information about the back end processor project. The tables supplied to demonstrate the back end processor implement a DIANA to P-code conversion. Subjects relating to DIANA and P-code are thus discussed. Note, though, that the back end processor is not tied specifically to DIANA and P-code. A short discussion is included which compares separate front end and back end compilation with conventional single pass and two pass compilation. Finally, other work in the field of table driven back end processing is discussed.

2.1 DIANA

Included in this sub-chapter is a section on the background and history of DIANA. The U.C.T. front end processor generates DIANA. So it also has brief description included. The DIANA produced by the U.C.T. front end processor is in the form of abstract syntax trees. Therefore abstract syntax trees are introduced here. The sub-chapter closes with a discussion of the U.C.T. implementation of DIANA.

2.1.1 Background and History

DIANA (Descriptive Intermediate Attribute Notation for Ada) is an intermediate language specification designed primarily for producing an intermediate form for Ada. It is based upon two other intermediate codes, TCOL and AIDA [Goos 82]. DIANA is being used as the intermediate language for many Ada implementations.

The designers of DIANA endeavored not to tie the DIANA specification to a particular representation. This resulted in the definition of DIANA as an 'abstract data type' [Goos 82] which can be implemented using a variety of representations. In addition, DIANA was made to correspond so closely to Ada, that an almost one-to-one mapping occurs between Ada productions and their associated DIANA (sub)trees [Goos 82].

The DIANA intermediate language was produced in an attempt to standardize the intermediate representation of Ada. The idea behind this effort is to ease communication between front end processors which produce the DIANA representations of Ada programs and tools such as formatters, language oriented editors, cross-reference generators as well as the back end processors which produce objects [Goos 82]. With such an arrangement it is possible to have, for example, a variety of back end processors which produce objects for different computers all using the trees produced by one front end processor. The front end - back end communication is of particular relevance to this thesis.

2.1.2 The U.C.T. Front End Processor

The University of Cape Town (U.C.T.) front end processor was developed and co-written by J. Epstein at U.C.T. in 1982-83. It takes as input a source language subset of Ada and produces as output an attributed abstract syntax tree implementation of DIANA which represents the input Ada program [Epstein 83]. The subset chosen was that which resembles the Pascal language. Although the U.C.T. front end processor generates DIANA trees only for that subset, the syntax checker component of the front end processor is constructed to be able to scan almost the whole Ada language and thus prepares the front end processor for possible expansion.

The version of Ada from which the subset used by the front end processor is taken, is that which was released in 1979 at the start of the front end processor project. It is known as Ada-80. The output DIANA release version matches that of the Ada subset input (DIANA-81), but has been modified to be compatible with the 1982 specification for Ada.

The output tree produced by the front end processor consists of a sequential text file containing the nodes which comprise the tree. The tree structure is formed by the various pointer fields within the nodes connecting the nodes together.

The the trees and nodes produced by the U.C.T. front end processor are discussed more fully later in this sub-chapter and in Chapter 4 (The Tables and Tree).

2.1.3 Abstract Syntax Trees [Waite 84]

As mentioned earlier in this chapter, the DIANA produced by Epstein's front end processor is output in the form of attributed abstract syntax trees (attributed ASTs). Therefore a brief description of ASTs and attributed ASTs follows.

When a source code program is translated into an intermediate form, a tree representation is often used. An AST is one such intermediate representation of a source code program. The structure of the AST is an abstract syntax representation of the source language. Each node corresponds to a rule in the language definition. As such, the AST follows an abstract rather than concrete syntax of the source language.

During the scan of a source code program, not only syntactic information but also semantic information is derived. However, an AST structure (on its own) can represent

only the syntax of a language. In order to represent the semantics, additional information (known as attributes) is attached to the nodes of the tree. This results in an attributed AST.

The attributed AST can be an explicit data structure resulting from the scan of a source code program by a front end processor. The attributed AST then can be stored and later processed by a back end processor so as to produce an object code. Processing an attributed AST consists of traversing the tree according to a set of rules or instructions.

This project centralizes around supplying a set of rules or instructions in a table to allow for the scanning of a variety of tree types and the production of a variety of object types.

2.1.4 U.C.T. DIANA Implementation

Each Ada production has a corresponding DIANA production. Thus, the structure of a DIANA subtree represents the syntax of the equivalent Ada statement. The example Ada statement 'A := B * C;' maps onto the DIANA tree depicted in figure 2.1. The example is provided in order to illustrate the DIANA tree structure (as produced by the U.C.T. front end processor).

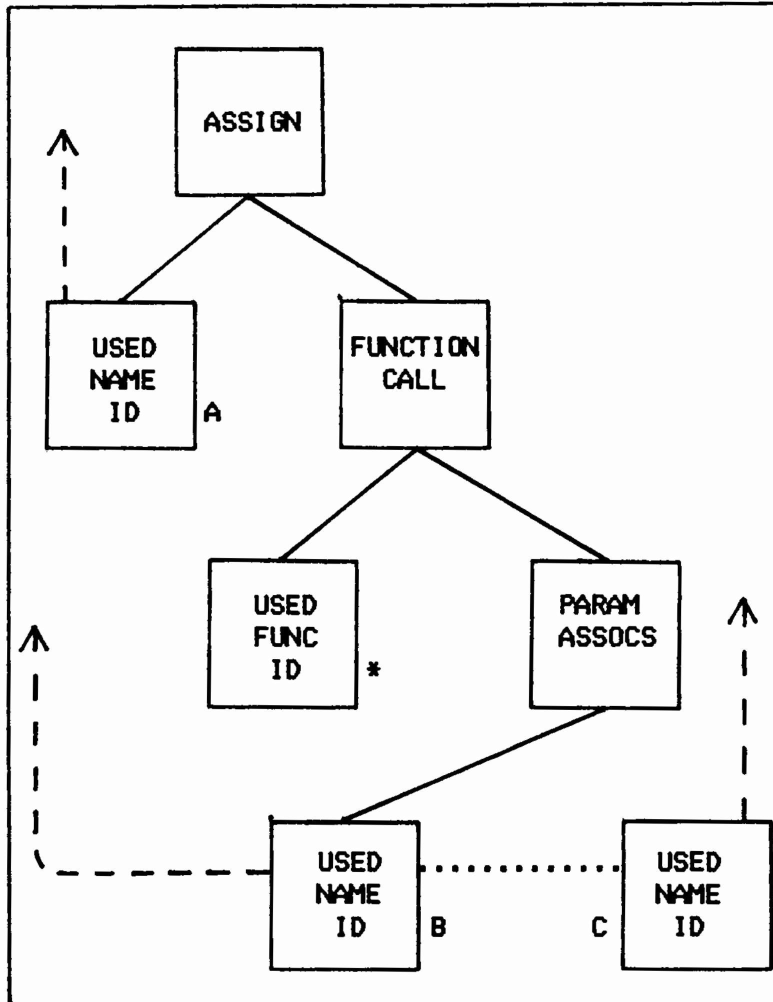


Fig. 2.1: A DIANA Subtree Representing 'A := B * C;'.

Each DIANA node has associated with it attributes which define its relationship to the other nodes in the tree. The important attributes are classified into four groups:

(1) Syntactic: referred to as the 'as-' attributes,

(2) Semantic: referred to as the 'sm-' attributes,

(3) Lexical: referred to as the 'lx-' attributes,

(4) Machine: referred to as the 'cd-' attributes.

The '-' following each of the above attribute names separates a descriptor from the attribute. The descriptor gives the type of the associated attribute and can describe what that attribute is used for.

The syntactic attribute connections (from here referred to as syntactic pointers) join the nodes together such that the resulting (abstract syntax) tree represents the equivalent Ada syntax. In figure 2.1, the syntactic pointers which join the nodes together are represented by solid lines.

The semantic attributes decorate the abstract syntax tree with additional semantic information such as value and type data. Linking a value to a node is an example of adding value information. Type information can be added by using semantic attribute connections (from here referred to as semantic pointers) to join a node to other nodes or subtrees. In figure 2.1, the semantic pointers are represented by dashed lines. The semantic pointers in the example connect the nodes representing the identifiers 'A', 'B' and 'C' to other DIANA subtrees where the definitions of the identifiers are represented (not shown in the figure). The identifier types and any additional information (such as range boundaries) can be obtained from these subtrees.

The lexical attributes provide additional information about the original source text (from which the subtree is derived). This information can be used by syntax directed editors and source text reconstructors. The attributes also include character representations of lexical tokens. These are useful to the U.C.T. back end processor.

In figure 2.1, the lexical tokens 'A', 'B','C' and '*' are the lexical attributes attached to the USED NAME ID and USED FUNC ID nodes. In the case of the USED FUNC ID node, the '*' token is important as it allows the back end processor to determine what function is being represented by the node.

The machine attributes can provide machine specific information. This attribute is not used by the U.C.T. front end processor. All machine specific information is provided to the back end processor in the form of a table. Therefore the back end processor does not need to use these attributes.

There is another important attribute in the U.C.T. implementation of DIANA. This attribute is used to tie together lists of DIANA nodes and subtrees. The attribute is called the 'link' pointer. In the example, a link pointer is depicted by a dotted line.

In addition to the attributes, each DIANA node carries two numbers, namely the node kind number and the node sequence number (not shown in Figure 2.1). The node kind number identifies which DIANA node definition is being represented by the node. This number is used by the U.C.T. back end processor to find the appropriate node definitions in the tables. The node sequence number is used by the attribute pointers to connect the nodes, thus forming the DIANA tree. Every node in a DIANA tree is given a unique node sequence number by the front end processor, thus ensuring that the attribute connections are unambiguous.

The example depicted in figure 2.1 is a DIANA subtree representing an Ada assignment statement. A front end processor must scan the Ada statement and produce an equivalent DIANA subtree. A brief description of the rules used to derive the DIANA

ASSIGN node from the Ada assignment statement is given in order to illustrate how the Ada statement maps onto the DIANA subtree.

The Ada syntax rule for an assignment statement is represented as follows (in a variant of the Backus-Naur form):

```
assignment_statement  ::=
   variable_name := expression
```

A DIANA assignment subtree has the DIANA ASSIGN node as its root. The DIANA syntax definition for the ASSIGN node corresponds to the above Ada syntax rule. The DIANA syntax definition for the ASSIGN node is as follows:

```
ASSIGN  =>  as-name  :  NAME
            as-exp   :  EXP
```

The Ada syntax rule for **expression** is defined by **as-exp** in the above DIANA syntax definition. Similarly, **variable_name** is defined by **as-name**.

The attribute **as-exp** carries the type EXP according to the above DIANA syntax definition. The type EXP may be defined as follows:

```
EXP  ::=  FUNCTION CALL/
          NUMERIC LITERAL/
          PARENTHESIZED/
          USED NAME ID/
          USED OBJECT ID
```

According to the above type definition, the subtree of the as-exp attribute can have one of FUNCTION CALL, NUMERIC LITERAL, etc. as its root node. In the example Ada statement 'A := B * C;', the expression on the right hand side of the assignment

consists of a single multiplication function. Thus the subtree of **as-exp** has a FUNCTION CALL node selected as its root. The order of terms and factors within the expression is not a complication here as this is a simple example. However, in more complex examples, the order is important.

The attribute **as-name** is of type NAME according to the DIANA syntax definition. The subtree of **as-name** has its root node selected from the definition of type NAME. A USED NAME ID node is chosen.

The right hand side expression 'B * C' can be reduced in a manner as described above until the whole DIANA subtree is produced.

Only a simple example is given here in order to briefly illustrate the DIANA tree structure. Further information on DIANA trees (as generated by the U.C.T. front end processor) can be found in Chapter 4 (The Tables and Tree) and Appendix C (Implemented P-code).

2.2 U.C.S.D. P-machine and P-code

This sub-chapter discusses the background and history of the P-machine and P-code. Also included here is a section which briefly describes the structure and operation of a typical P-machine.

2.2.1 Background and History

Although the concept of a "Pseudo-Machine", or P-machine originated at ETH Zuerich, many of the P-machines available today come from work done at the University of San Diego (U.C.S.D.) [SofTech 81]. The U.C.S.D. P-machine and associated language were developed along with a Pascal compiler, editor, file handling system and operating system to form a combination commonly known as the P-system. The P-system is quite popular today and is implemented on many different computers.

The U.C.S.D. P-machine is a stack based sixteen bit computer. In most systems, it is a device implemented in software as an interpreter [Apple 80]. Although not common, the P-machine can be implemented in the hardware or microcode of a computer. One such example is the 'Pascal Micro-Engine'. The instruction set of this computer is the same as that for the P-machine. The advantage of this approach is a fast P-machine.

The language of the P-machine is called "Pseudo-Code" or P-code. This is an intermediate stack based language, the instructions of which are interpreted by the P-machine. The P-code instruction set is at a level higher than most computer native instruction sets and is therefore more convenient to use.

The reasons for the P-machine and its associated P-code language being developed are two-fold:

(1) any application written using a compiler which produces P-code as its object is transportable, with little or no modification, to any computer in which a P-machine interpreter resides. It is much easier to write a new P-machine for

a computer than it is to re-write a compiler. So P-code lends itself to portability,

(2) P-code is often more suited to being an object for many languages than is the native code of most computers. This is because P-code instructions are at a higher level, they are concise and clear in meaning and a stack is used. Additionally, the use of P-code results in more compact object files than does a normal native object code.

P-code normally has a notable disadvantage, namely speed. Simulating the P-machine in software has a large overhead. So, except on hardware implemented P-machines, a P-code program will almost always run slower than the equivalent native code program. One method of overcoming this disadvantage is to assemble the P-code program into the target computer's native machine code. A faster object program results from this operation.

2.2.2 P-machine Operation

A small example is included here so as to illustrate the operation of a U.C.S.D. P-machine. However, before this is shown, a brief (and simplified) description of the U.C.S.D. P-machine is needed.

In the memory of a computer running a P-machine, two dynamic data structures are held, namely the stack and the heap. The stack is used for the evaluation of expressions, holding static variables and other 'house keeping' information. The heap is used to hold, among other things, dynamic variables. The stack and the heap grow towards one

another, with the heap growing from low memory up and the stack growing from high memory down.

In the memory space between the stack and the heap resides the code pool. The code pool contains the executable program code. This can exist in contiguous groupings called segments. The segments can be dynamically swapped in and out of the code pool. Also, the code pool can be moved about during program execution. Thus memory usage is optimized as the demands on memory by the stack and the heap vary.

Figure 2.2 illustrates the memory usage by the stack, heap and code pool.

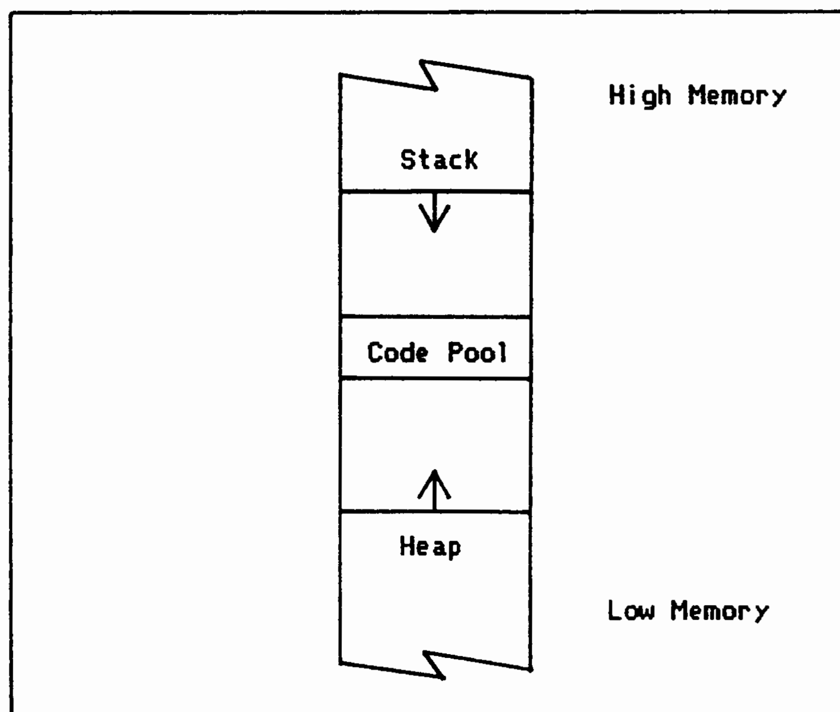


Fig. 2.2: Simplified Illustration of P-machine Main Memory Usage.

Each segment consists of a collection of routines (procedures). When a routine is invoked, space is assigned on the stack for the storage of any parameters, local

variables, temporary variables, function return value and 'house keeping' data. This space is called the activation record for the routine.

Local variables and parameters (or their addresses) which are stored in the activation record are referenced relative to the beginning of the activation record. The first location in the activation record is referred to as location 0. In some P-machine implementations, two words for the function return value are always reserved (even if no return value exists, such as in a procedure). Locations 0 and 1 are reserved for the function return value. Therefore the starting place for parameters and local variables is location 2.

The Ada assignment statement 'A := B * C;' is chosen as the example here (thus continuing its use from the previous sub-chapter). Assume that all three identifiers are locally defined integer variables. Assume also that 'A' is defined first, followed by 'B' and finally 'C'. The above assumptions are made simply to set the scene for the chosen P-code instructions. If different assumptions are made, different P-code instructions are needed.

With the assumptions made, the Ada assignment statement results in the following P-code fragment:

```
LDL  3
LDL  4
MPI
STL  2
```

LDL 3 pushes a copy of the one-word value stored at location 3 in the local activation record onto the top of the stack. LDL 4 does the same for the value stored at location 4. The MPI instruction pops the two one-word values from the top of the stack, multiplies them together and pushes the one-word result. Finally, STL 2 pops the one-word result and stores that value at location 2 in the local activation record. Thus the assignment statement 'A := B * C;' is executed in P-code.

This is a simple example provided only to give a brief introduction to P-code and the P-machine. Further information on the DIANA to P-code conversion implemented for this thesis can be found in Appendix C (Implemented P-code).

2.3 Comparison of Compilation Techniques and the U.C.T. System

Conventional compilers usually can be broken into two groups, namely single-pass and multi-pass compilers. The Two-pass compiler, which falls into the multi-pass category, is of relevance to this thesis. So only single-pass and two-pass compilers are covered here.

Whatever the form of the compiler, the basic compilation process is as follows:

(1) in the first phase the high level source language is scanned by a lexical analyzer, sometimes called a scanner. This process takes the source program, which is suitable for humans to read and converts it into a representation more suitable for automatic analysis. This is done by discarding comments and blanks, converting all multi-character reserved words and symbols into single characters, set element tokens or atoms and generally filtering out the information represented by the program from the 'noise',

(2) in the next phase the output from the lexical analyzer is parsed. In other words the syntax and semantics of the source program are checked, and some form of intermediate representation is produced. This intermediate representation can take a number of forms such as trees, a list of quadruples or a form of Polish notation [Gries 71]. Only tree representations are discussed here because of their special relevance to this thesis. In addition to producing a tree, often a symbol table is produced to carry additional information about identifiers,

(3) the final phase in this process is the conversion of the tree into an object code. This process usually requires all the tree information plus much of the symbol table information. The resulting output code is usually in the form of a machine code for a target computer or an assembler listing. In some cases this output may be another high level language form [Calingaert 79].

Figure 2.3 gives an illustration of the above process.

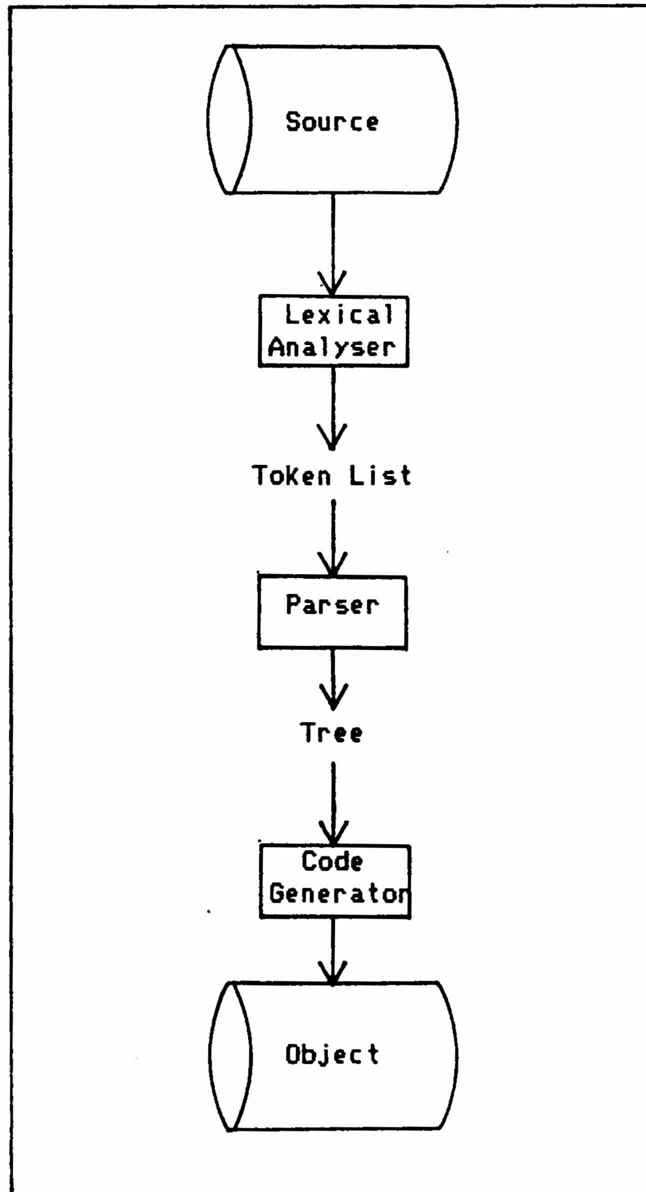


Fig. 2.3: The Structure of the Compilation Process

If the above phases are linked such that each statement in a program goes through all the phases before the the next statement is read, then the system is known as a single-pass compiler.

A single-pass compiler runs through the source program text only once and no intermediate representation is produced and stored for later scanning. In other words, for every statement in that source, the appropriate code is generated at the time the statement is read.

Although a single-pass compiler is space efficient, there are many languages which cannot be scanned by such a system. In addition, such a compiler is often difficult to define and construct.

A two-pass compiler splits the compilation into two separate processes. This division usually is placed between the parsing phase and the code generation phase. On the first pass, the source program is scanned by the lexical analyzer and the tokens are passed on to the syntax and semantic checking phase, as before. But here the output of the second phase is in the form of an intermediate tree representation which is stored usually on the host computer's secondary storage system. The code generation phase of the compiler then takes the tree and uses it to produce the necessary code. In cases where the tree does not carry enough information to produce the output code, the symbol table is also stored on disk to complement the tree information.

It can be seen that more storage space is required in the two-pass system. Also, the disk I/O overhead usually adds significantly to the compilation time. But such disadvantages are more than offset by the increased flexibility provided with the two-pass compiler [Bornat 79].

The U.C.T. front end and back end processors together form a variation of the two pass compiler. However, the front end processor and back end processor are two separate systems, each totally independent of the other. This means that a variety of front end

processors can be used with the back end processor and vice versa. Also a variety of tools can be applied to the intermediate tree form. In the case of the U.C.T. front end processor, almost all the additional attributes normally stored in a symbol table decorate the tree. So no separate symbol table is supplied with the tree.

The difference between the U.C.T. back end processor and conventional back end processors is that the former draws from user written tables the information about how to scan the tree and what to do at each node. This greatly increases the flexibility of the back end processor and allows it to handle a variation of tree structures and output codes.

2.4 Other Table Driven Back End Processors [Graham 80]

The automation of compiler front end processors is fairly well developed. However, the back ends of compilers are still produced in a mostly ad-hoc manner. Compilers which are written with such back end processors are specific to only one target machine. So, if the target machine is changed, a whole new back end processor must be written. This results in much duplication of work and complicates verification (because of the ad-hoc nature). For such back end processors, all the actual code information is distributed throughout the program. The idea behind the table driven code generation process is to centralize this information in a table. By doing this, back end processor production could be made simpler, quicker and result in a better product. Thus there are a number of projects researching the production of table driven compiler back end processors (including this one). They are all attempts to develop more formal and well defined techniques for producing back end processors. Each has addressed the problem in a different manner.

This sub-chapter gives a brief description of three other interesting approaches to the table driven back end processor. All three systems take as input an intermediate level representation of a program and produce some form of object code representation of the program. The intermediate representation of the program is usually produced by a front end processor scanning the equivalent source code program. In this sub-chapter, the three different approaches are contrasted with one another and with the U.C.T. back end processor.

One method was produced at the University of California, Berkeley by R. S. Glanville and S.L. Graham. Their system takes as input a low level intermediate tree program, representing the original source code program. This tree is scanned by the back end processor. The tree traversal is fixed into a depth first, left to right walk without backup (a prefix walk). The instructions are generated "on the fly". Namely, they are generated during the traversal of a (sub)tree in the intermediate tree program and not after the (sub)tree has been scanned and considered. This limits the amount of pre-planning that can occur, so does not allow for the generation of optimum code. Graham and Glanville state that a pre-planning strategy for such a compiler would choose the appropriate traversal order and then scan the corresponding sequence of nodes to generate code. They chose the prefix walk without backup (along with a simple rule for generating choices) to facilitate efficiency and ease of automation.

The table for this system is produced by a table generator program. This program takes as input a description of the target machine. The description resembles a set of context free grammar rules. The table generator program ensures that no situation can occur where, when output code is required, none is produced. In other words, the description is checked.

Such a system is claimed never to loop and always to produce correct code given a "well formed input". However, a recurring problem is that of CPU register allocation. One solution suggested in Graham's paper was to decorate the intermediate tree representation with register allocation information. But this would complicate the input language specifications for the front end processor and limit the back end processor to use with front end processors with such a capability. Also, this system assumes that the intermediate language is at the level of labels and jumps, which implies a low level intermediate language. Many intermediate representations are at a level higher than this.

Another approach to the problem was produced by S. C. Johnson. This system is known as the portable C compiler. The goal of this project is easy retargetability (as well as to produce code of reasonable quality). This has been demonstrated by using the system for the production of C compilers for a variety of target machines.

The portable C compiler uses a pre-planning strategy when scanning an input tree so as to avoid running out of target machine registers. This requires additional information to be added to the input (sub)trees in order to determine which temporary values (in expressions, for example) should be stored. By doing this, the code generator portion of the system is always presented with an input tree for which it will not run out of registers to allocate.

The table for the portable C compiler contains a collection of templates. Each template consists of, among other things, patterns for tree searching and an encoded form of assembly language instructions for the generation of code. The additional information needed for the allocation of registers and temporary storage must be included in these templates.

Unlike the Glanville and Graham system, the templates here are written by the implementor rather than being generated automatically. This results in a more general system (being able to handle difficult situations), but at the expense of performance. In addition, the correctness and quality of the table definitions depends upon the implementor.

Much effort has been put into solving the register allocation problem in the portable C compiler. However, the additional information required to do this complicates the table entries. Also, this time consuming process of allocating registers and temporary storage reduces the overall speed of the system. Finally, the system is meant only to produce C compilers. It therefore does not contain a back end processor which is designed to accept a variety of input tree specifications.

The third system under discussion here is the Production Quality Compiler Compiler (PQCC) produced by R. G. G. Cattell, J. M. Newcomer and B. W. Leverett. As with the other two systems, PQCC is a table driven back end processor system. PQCC is designed to accept programs represented by the intermediate language TCOL as input. The goal of the PQCC project is to produce a system capable of generating very high quality object code.

In order to achieve the production of high quality object code, the system employs a pre-planning system for scanning the input tree, as does the portable C compiler. This allows for the selection of the best sequence of object instructions for a given TCOL language structure. Also, so as to handle the register allocation problem, PQCC goes through a pseudo-code generation phase prior to the actual code generation phase. This allows for the optimal allocation of registers and temporary storage beforehand.

However, as with the portable C compiler, this incurs a time penalty which slows the code generation process.

As with the Glanville and Graham system, the table is produced from a set of production like assertions describing the instructions. From these productions, patterns are derived by a code generator-generator. Like the portable C compiler, code generation is done by matching pattern trees to the tree input.

PQCC is the most automated of the systems described here, in that it requires the least effort from the implementor in producing the table. This is because the system is highly specialized towards using TCOL as the program input. Of course, this specialization highly restricts the input representation of the programs and reduces the generality of PQCC. Note, once again, that significant effort was given to the solving of the register allocation problem.

In all three of the above described systems, the common problem is how to allocate the target machine registers efficiently. The solution was either to have inefficient register allocation or to include additional information in the input tree. Such additional information is added either by the front end processor which produces the trees or is added afterwards (the input trees are amended or pre-scanned). All the above solutions result in significant compromises. Either the table entries are (relatively) simple but inefficient code is produced. Or efficient code is produced but the table entries increase in complexity. Otherwise the table entries are simple and the code produced is efficient, but the generality is impaired. If the approach for the U.C.T. back end processor was to adopt one of the above techniques, it would have to select the system whereby the input trees are pre-scanned. This is because the DIANA trees for which the U.C.T. back end processor was originally intended have no

additional register allocation information. As such, the tables used by the U.C.T. back end processor would have to contain significantly more information, thereby making the system more complex to use and less easy to debug. Alternatively, the system would be restricted to DIANA as the input language, thereby reducing the input generality of the system.

However, there is an approach whereby the register allocation problem can be side-stepped. This is achieved by generating target code for a P-machine, namely P-code. The advantages of generating P-code are mentioned earlier in this chapter (P-code is a higher level object code, is somewhat standard and is fairly widespread). But in addition, P-code has been in use for some time. This implies that P-machines are (often) efficient and well debugged. The stack based nature of the P-machine lends itself well to efficient code production and also completely avoids the problems associated with register allocation. So although the U.C.T. back end processor is not tied specifically to P-code generation, it is felt advantageous to remain with P-code (for the above-mentioned reasons).

The U.C.T. back end processor is also very flexible as to the structure of the input trees. So whereas the three systems described above tend to be more restrictive in their choice of associated front end processors, the U.C.T. back end processor can scan trees produced by a variety of sources (provided, of course, the tables are set up to allow the back end processor to correctly scan the given input trees). In addition, the U.C.T. back end processor is not tied to a prefix walk as the Glanville and Graham system is. The order in which the U.C.T. back end processor scans a tree depends upon the table entries and can be context sensitive. In other words, pre-planning can take place and decisions can be made as to how (sub)trees are scanned, based upon the meaning carried in these (sub)trees.

The input trees to the U.C.T. back end processor can, if wished, include register allocation information. The U.C.T. back end processor could then use this additional information to generate more optimal object code (again, given the correct table entries). However this is an option and is not essential for the production of correct object code.

All the above systems have their various unique formats for the definition tables. The U.C.T. back end processor tables are also unique. This system uses two tables in order to assist in correctness. The one table carries the formal definition of the input tree. The format of this table is derived from Epstein and is very similar to the format she used to describe DIANA. The other table carries all the scanning, code generating and decision making entries for producing object code from input trees. The format of this table was governed by an attempt to maintain a similarity to the other table.

The formats of the U.C.T. back end processor tables are covered more fully in Chapter 4 (The tables and Tree).

CHAPTER 3

STRUCTURE OF THE ADA COMPILER

The U.C.T. front end processor produces DIANA intermediate tree representations of Ada (subset) source programs. The U.C.T. back end processor, when set up with the correct tables, takes these trees and produces a corresponding P-code output object text. This P-code text then can be assembled to produce an object code, or it can be converted into a form compatible with a P-code interpreter.

3.1 Development History

The final form of the U.C.T. Ada compiler may differ somewhat from that intended by Epstein. She originally envisaged that the whole compiler would exist on a Sperry Univac 1100 series mainframe, and wrote much of the compiler's front end processor using Univac specific languages, assemblers and utilities.

The development of the back end processor started on the Univac but was later transferred to the NCR Tower series Micro/Mini, which supports the Unix operating system. Unfortunately, because of the aforementioned reasons, Epstein's front end processor cannot be transferred. So an alternative route for generating DIANA trees from Ada programs is, at the time of this writing, being explored at the University of Cape Town. This new route uses the routines Lex (a lexical analyzer) and YACC (acronym for Yet Another Compiler Compiler - a parser).

Figure 3.1 illustrates the evolutionary development path of the system from the Univac environment to a Unix environment.

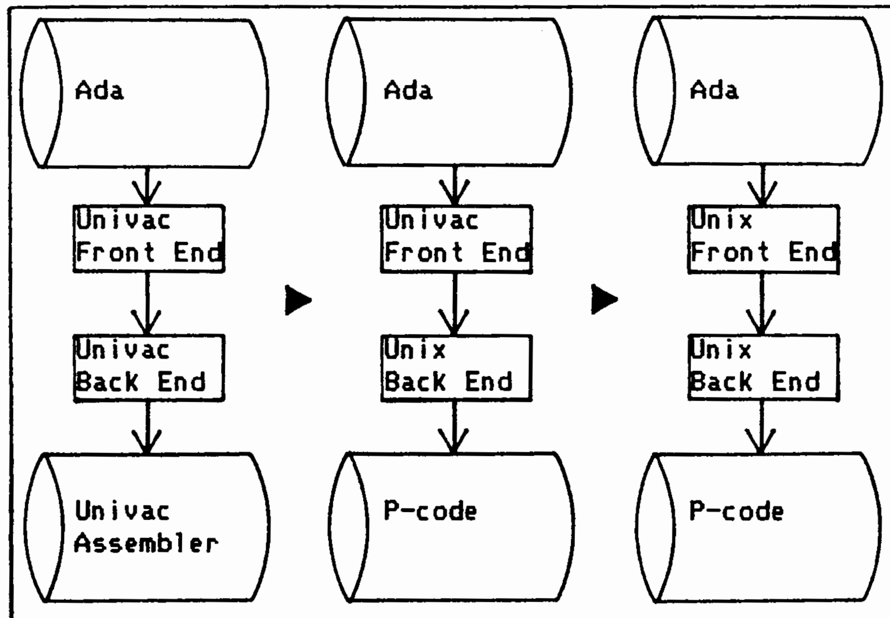


Fig. 3.1: Evolution of the Ada Compiler from Univac to Unix.

3.2 Back End Processor Components

The back end processor system consists of a number of modules and tables. It is composed of three Pascal (main) programs, three C (minor) programs and two tables. The structure of the system is illustrated in figure 3.2. The figure also depicts the level at which the back end processor system interacts with the front end processor system.

An introduction to the back end processor system's illustrated components follows after figure 3.2.

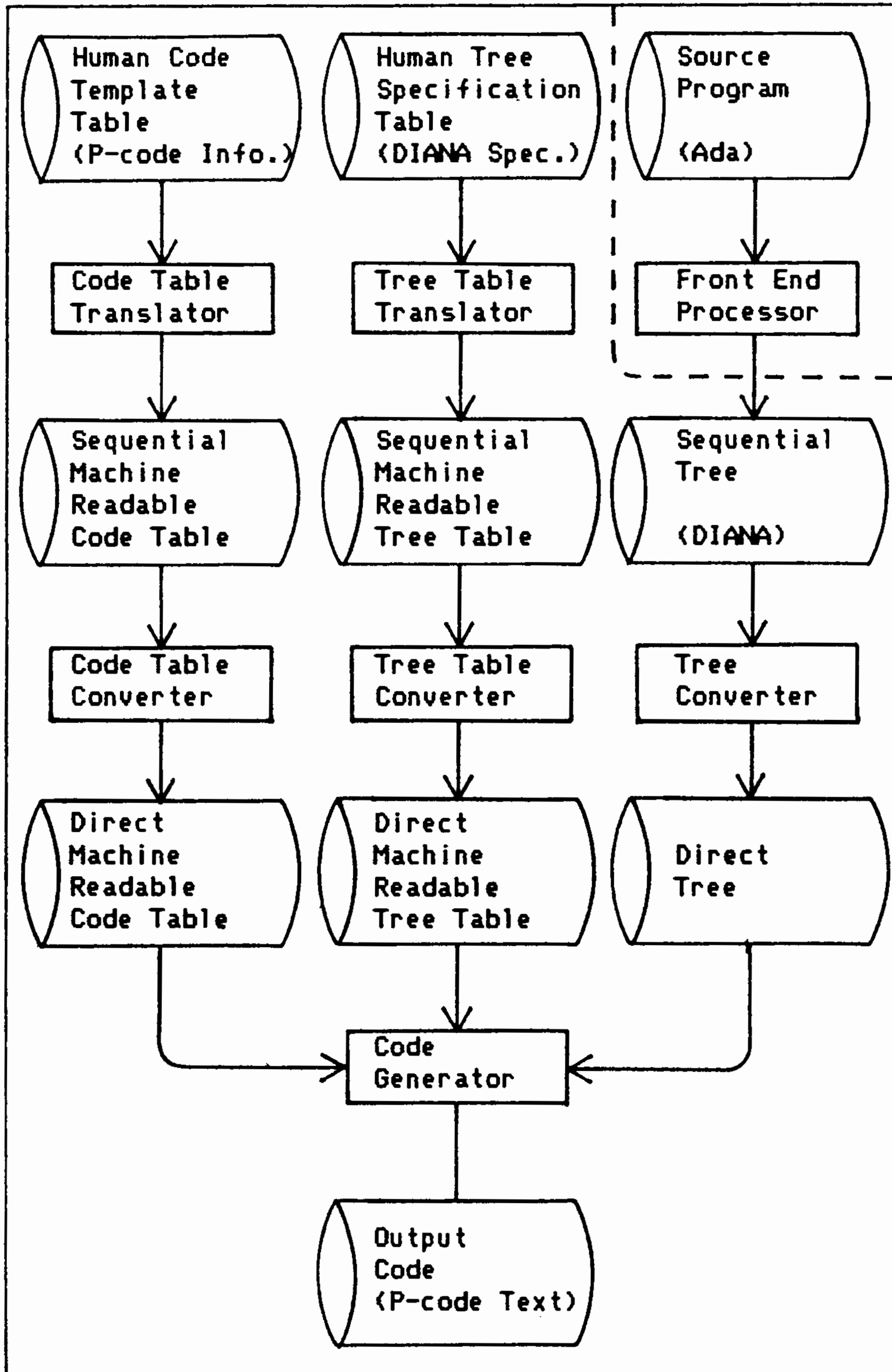


Fig. 3.2: Structure of the Back End and its Placement Relative to the Front End.

3.2.1 Tables

So as not to limit this system to a single input tree or output code format, the code generating module (from here referred to as the code generator) is guided by two tables when translating the input trees to the output code. These tables are derived from two user written tables, namely the tree specification table and the code template table (see figure 3.2).

The tree specification table contains a complete specification of the tree nodes implemented. The code template table contains the code templates and other information necessary for the generation of the appropriate object code for each node. An example of each table is provided with this thesis. The example tree specification table carries a specification of all the DIANA nodes implemented in the U.C.T. front end processor. The matching example code template table carries the information which describes what must be done for each DIANA node kind in order to generate the appropriate P-code instructions.

As the tables are user written, an effort has been made to keep the table formats readable by humans. Therefore the tables are not very suited to being read by programs. It would be inefficient for the code generator to have to decode such information. To remedy this, translators are provided to convert the tables into a form more suitable for the efficient operation of the code generator.

3.2.2 Translators

These programs convert the user written tables into their machine readable equivalents. The tree specification table and the code template table each has its own translator (see figure 3.2). During the translation, a certain amount of syntax and rudimentary semantic checking is done on the two tables by the respective translators. This enables some of the more common (table) errors to be trapped before use is made of the translated tables.

As a result of the limited file handling capabilities of Pascal, the machine readable tables must be generated as sequential files. In addition, the DIANA trees generated by the U.C.T. front end processor are in the form of sequential files. Obviously, if the code generator had to read sequential files for randomly located information, it would be unacceptably slow. So sequential file to direct file converters had to be written.

3.2.3 File Converters

The file converters rewrite the sequential file outputs of the two translators and front end processor into direct files. There are three converter programs, one for each translated table and one for the output of the front end processor (see figure 3.2). After the conversion, the code generator can use the direct files as its input. The code generator must have some C routines linked in to enable it to read the direct files because of the limited file handling capabilities of Pascal.

3.2.4 Code Generator

The code generator is the central hub of the whole back end processor. It takes as input the translated and converted versions of the two tables and tree. It uses the information provided in the two tables to guide it in its walk around the tree and to indicate what actions to take at each tree node. This action results in an output object code representation of the input tree (see figure 3.2). It produces the output code in a sequential file.

Being able to walk rigidly around a tree and having unconditional code templates for the appropriate nodes does not provide enough flexibility for generating correct code. For some nodes, decisions have to be made to determine which pointers emanating from these nodes are to be followed. Also, decisions often have to be made to determine which piece of code, from a number of options, is to be produced. Take, for example, an operation involving a declared identifier in a DIANA tree. Here, the identifier's place of definition in the tree has to be found (by following a semantic pointer) so as to determine that identifier's attributes. Only then is it possible for the correct code to be selected for this identifier and associated operation. To provide the required flexibility, certain instructions and structures are included in the code template table 'vocabulary' to allow:

(1) execution of special built in routines,

(2) decision making capabilities.

The code generator could have been 'hard-wired' to handle the DIANA nodes requiring special treatment. But then the system would be restricted to DIANA, thereby

severely limiting its generality. Instead, in this system, the above DIANA nodes have the necessary special routine calls and conditional structures included in their respective code template table definitions.

As the system's primary requirement is to handle DIANA, the special routines and conditional structures provided are aimed at the DIANA to P-code conversion. But these features should be general enough to handle a wide variety of different tree types and P-code dialects (or P-code like objects).

CHAPTER 4

THE TABLES AND TREE

The input to the back end processor system consists of two user written tables, namely the tree specification table and the code template table, and a tree as produced by the front end processor.

The tree specification table contains the full specification of the implemented nodes in the tree. It describes, for each node type, the fields and pointers which exist. The code template table contains all the information necessary to guide the code generator in its walk around the tree as well as to select what code to generate for each node type in the tree.

Although the information in both tables could have been combined, resulting in a single table, it is considered better to have the two tables because,

-- if a table driven tree interpreter is produced, it could use the tree specification table and have its own execution table instead of the code template table. In other words, separate systems could share common information about the tree,

-- the separation of the tree specification and the code generation information (or, in an interpreter's case, execution information) is a logical division and possibly produces a clearer picture of the functioning of the system.

The tree is an intermediate representation of a source language program. The source program is scanned by the front end processor which then produces the tree as its

output. The code generator scans the (converted) tree and, with the guidance of the two (translated and converted) tables, produces an output code which represents the tree.

What follows is a description of the structure of the tables and the tree.

4.1 Tree Specification Table

The tree specification table provides the full specification of the implemented nodes of the tree. This consists of a list of definitions for every node kind that can occur in a tree using this tree specification. Any use of node fields in the code template table is compared with the information, for the relevant node, in this table.

When deciding upon a format for this table, much thought was given to making it readable. In addition, if this format is already used elsewhere, it then would not be necessary for the users to learn yet another table format. The format of the table provided in Epstein's thesis fits both requirements in that it is easy to read and is already used [Epstein 83]. There is another advantage to using this format. Noting that the primary requirement of this work is to produce a back end processor to complement Epstein's front end processor, it seems appropriate that this system uses the table listed in her thesis to describe the specification of the trees her system produces.

Figure 4.1 depicts the format of a node definition in the tree specification table.

```
<node name>  <node number>  <attribute> <reference>
                            <attribute> <reference>
                                 .           .
                                 .           .
                                 .           .
                            <attribute> <reference>;
```

Fig. 4.1: Tree Specification Node Format.

The <node name> is the alphanumeric name given to the node. This may be any name the user wishes and is ignored by the system. Although ignored, the name must exist.

The <node number> is the node index. This number must exist and be unique for each node. The system uses this number to access the required node definition.

The <attribute> field describes the type of the corresponding reference fields attached to the node. There are six different attribute types:

(1) as-: represents a syntactic attribute,

(2) sm-: represents a semantic attribute,

(3) lx-: represents a lexical attribute,

(4) cd-: represents a machine dependent attribute,

(5) link: chains the concatenated tree statements together,

(6) denoted: used for denoted type pointers, see text later.

The machine dependent attribute (cd-) is unused by the U.C.T. front end processor and is also, at present, unused in this system. But it may still exist in the table and is translated into the corresponding machine readable equivalent. So future modification of the front end processor and code generator will have this attribute available for use. Any machine dependent information is supplied in the code template table in a form more suitable for the generation of P-code.

The '-' following most of the attribute symbols separates from the symbol an attribute descriptor. This gives the associated attribute's type and can describe what the field is used for in the tree. The descriptors are of little use to this system and are not rigid enough to provide much definite information. So they are ignored.

The <reference> field defines the existence of a field in the tree node. There are six different reference field types:

(1) reference(<number>): defines a pointer field of ordinal number <number> which connects the the node to another node in the tree, where <number> is in the range 2 to 12,

(2) value(<number>): defines a value field of ordinal number <number> attached to the node which is used to carry a numeric value, where <number> is in the range 1 to 2,

(3) boolean(<number>): defines a boolean field of ordinal number <number> attached to the node which is used to carry a boolean value, where <number> is in the range 1 to 2,

(4) rep: defines a string field attached to the node and carries any alphanumeric or non alphanumeric strings,

(5) unimplemented: indicates that the node can exist but, in this case, has not been defined,

(6) attribute(<number>): used for denoted type pointers, see text.

The ranges of <number> attached to the above field are for the tree nodes as produced by the U.C.T. front end processor (covered later).

The first line in each node definition must have four items, namely the <name>, <number>, <attribute> and <reference>. Also, for every line in the node definition, there must be an item in the <reference> field. But this is not the case for the <attribute> field (except for the first line). If the <attribute> item is the same for a number of consecutive <reference> items, then the <attribute> item need be used only once at the beginning of the aforementioned list of <reference> items.

With only three exceptions, the tree specification table follows exactly the format of Epstein's table. The exceptions are as follows:

-- the rep field in Epstein's thesis consists of four separate four letter strings grouped together, forming one sixteen letter string. This is indicated as such

in her table. All strings in the code generator are loaded into sixteen letter arrays and are manipulated from there. Therefore in the tree specification table used here, the rep field is treated as one sixteen letter string, so the numeric references to the rep field are dropped,

-- a semicolon is used to indicate the end of a node definition,

-- there is a special node definition which defines denoted type pointers. Such pointers are used in DIANA to identify the native types of numeric literals and variables. The format for the definition differs in this table to that in Epstein's table. Its format here is as follows:

```
<node name>  <node number>   denoted   attribute(<number>);
```

This change is made only to ease the generation of machine readable nodes by the translator. While discussing this node definition, note that the code generator does not make use of such definitions because all native type information is included in the code template table in a manner more suited to the generation of code. Although unused, such node definitions may exist in the table and are translated to their machine readable equivalent. They are thus available for any future modification of the code generator.

An example of a tree specification table node definition is given in figure 4.2. The example is not a DIANA node definition. It is a fabricated table entry shown for illustrative purposes.

```
EXAMPLE 250   as-synpntr   reference(2)
              as-synpntr   reference(3)
              sm-sempntr   reference(4)
              sm-valbool   value(1)
                           boolean(1)
              lx-string    rep
              link         reference(12);
```

Fig. 4.2: Example of a Tree Specification Table Node Definition.

The tree specification table node entries are not order sensitive. In other words, the order in which the attribute - reference pairs occur does not affect the operation of the code generator. This is because the table is used to verify the existence of fields in the tree node and not the order in which they occur.

Initially it was considered to include some walk controlling information, such as order of entry, into the tree specification table, but this was rejected for two reasons:

(1) such modification would seriously affect the use of this table by any other system. For example, a tree interpreter may need to traverse the tree in a totally different order to the code generator,

(2) in order to match the entries in the code template table it would be necessary to include conditional structures into this table. This would be unnecessary duplication and wasteful of effort and table space.

4.2 Code Template Table

The code template table carries all the information necessary to guide the code generator when traversing the tree and generating code in the code output file. The table takes the form of a definition for every node that can occur in a tree with the specification as described in the tree specification table.

The format for this table originated in the same manner as that for the tree specification table. But after removing all the duplicate information and adding the command words, code templates and conditional structures, this table no longer so closely resembles Epstein's table. Nevertheless, it is felt that the readability of the code template table has not suffered.

Figure 4.3 shows the format of a node definition in the code template table.

```
<node name>  <node number>   <details>
                             <details>
                                 .
                                 .
                                 .
                             <details>;
```

Fig. 4.3: Code Template Node Format.

The <node name> is the alphanumeric name given to the node. As with the other table, this name can be anything the user wishes as it is ignored by the system. However, the name must exist. Although not compulsory, this name should be the same as the name for the definition of the same node in the tree specification table.

The <node number> is the node index. This number must exist and be unique for each node definition. In addition, the <node number> must be the same for the corresponding definitions in both tables for the same node. The system uses this number to access the node definition.

The <details> field may be one of the following:

-- SynReference(<number>): indicates that reference field <number> in the tree node carries a syntactic pointer, where <number> is in the range 2 to 12. Note that for the purposes of traversing a tree, there is no distinction between a syntactic pointer and a link pointer. So, in the code template table, link pointers are also listed as syntactic,

-- SemReference(<number>): indicates that reference field <number> in the tree node carries a semantic pointer, where <number> is in the range 2 to 12,

-- 'code': special command word which indicates the existence of code templates and any conditional structures controlling these templates,

-- 'comm': another special command word which allows the conditional structures to control the traversal of syntactic and semantic pointers. In addition, it allows these structures to control the execution of the normal command words,

-- all the other (normal) command words. There are too many of these to list here. The command words are all described in detail in Chapter 6 (The Code Generator),

-- unimplemented: identifies that the node definition exists but is not defined. When encountered, no action is taken,

-- null: indicates that the node exists and is defined, but that no entries exist for it in the code template table. This field also doubles as a filler for the <details> field in the first line of a node definition if it is wished to have a whole line for the first <details> item. When encountered, no action is taken.

All conditional structures may exist only within the bounds of the special command words. In addition, all the code templates must exist within the bounds of the 'code' special command word.

The first line of each node definition must have three items, namely <node name>, <node number> and a <details> item. Thereafter, only the <details> items are expected. Note that the format of this table is somewhat more flexible than that of the tree specification table. This is most apparent within the code special command word, where the code templates are situated.

The order of placement of the <details> items is of great importance as this determines the order of execution of the command words, traversal of the pointers and generation of the code. The code generator starts its scan of a node definition from the beginning of that definition and then works its way sequentially towards the end.

Figure 4.4 gives a fabricated example of a code template table node definition. This example matches the tree specification table example in figure 4.3. A few normal command words and conditional structures are included to complete the example, but their meaning is not too important at this stage.

```
EXAMPLE        250     null
   code an unconditional template:
                       SynReference(2)
                       Lflag1Set
                       SynReference(3)
                       LStore2Reset(9)
                       SynReference(2)
                       Lflag1Clear
                       SemReference(4)
   comm ?T12f          Lflag10Set
                       SynReference(2)
                       Lflag10Clear:
   code ?B1t a conditional template:
                       SynReference(12);
```

Fig 4.4: Example of a Code Template Table Node Specification.

Any <details> item which refers to a field in the tree node must have a corresponding item in the tree specification table confirming the existence of that field. Remember that the tree specification table is used to verify that all node fields are used correctly in the code template table. The code generator will detect any discrepancy in the table definitions for the same node and will generate an error message and take the appropriate action.

Note the multiple occurrences of SynReference(2) in the above example. In the matching tree specification table node definition, only one entry indicating the existence of the syntactic pointer is necessary.

In addition to carrying the code templates and pointer indicators, this table has another important function. In order to initialize the code generator before it starts to scan a tree, a special node definition (node number 0) is used to carry initialization information. The node has the same format as all the other nodes in the code template

table, but it has its own set of command words. The node carries, among other things, the native type information. Natives types are the types which are inherent in the language concerned (Ada, for example). This information is essential to the generation of P-code. In addition to native types, the initialization node carries other information. This is covered in detail in Chapter 6 (The Code Generator).

4.3 Tree

The tree is the intermediate representation of a source program written in a high level language. The nodes in the tree have a number of fields which carry pointer information to join the nodes and other attributes to give additional information about the represented program. The format of the node is illustrated in figure 4.5.

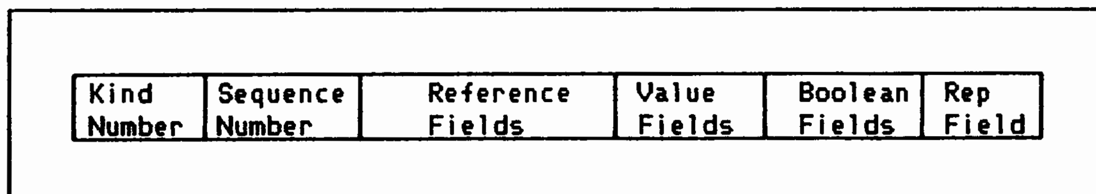


Fig. 4.5: The Tree Node Structure.

What follows is a description of the fields:

- Kind Number: The identification of what kind the node is. This number is used by the code generator to read in the table entries for that node,

- Sequence Number: every node in the tree has a unique sequence number. It is this number which is used by the syntactic and semantic pointers to join the nodes in the tree,

-- Reference Fields: there are eleven available reference fields. These are used to hold syntactic and semantic pointers. If one such field is used for a pointer, and it points to a node, then the sequence number of that node is held in the reference field,

-- Value Fields: there are two of these fields. They are used to carry any numeric attributes which may be attached to the nodes. Note that these fields are of type real,

-- Boolean Fields: there are two of these fields. They carry binary values. In DIANA, they are used to indicate the presence of a number in the value field of the same ordinal number,

-- Rep Field: this is a string field of up to sixteen characters long. It carries any string attributes which may be attached to the nodes.

The structure of the node is as produced by the U.C.T. front end processor. Although this node structure is used to represent DIANA, it should be detailed and flexible enough to carry tree nodes for specifications other than just DIANA.

Without the table definitions, all the information contained in a node cannot be identified and is therefore useless. It is only the table entries for a node which give the fields in the nodes meaning.

Figure 4.6 depicts a possible tree node. The example node has a Kind number of 250. This number matches the Kind number of the example table entries given in figure 4.2

and figure 4.4. Thus those table entries define the tree node given in figure 4.6 and describe what must be done with it.

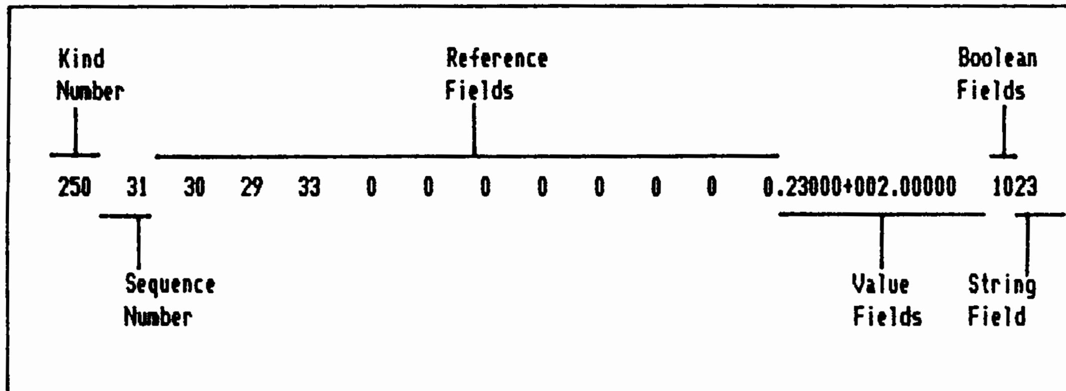


Fig. 4.6: Example of a Node.

Some of the reference fields in the node carry zero values, indicating that they are either unused in this particular node or undefined for this node Kind. Others carry positive non-zero values. The non-zero values in the reference fields refer to other nodes in the (hypothetical) tree from which this node is extracted. According to the tree specification table entry for this node (figure 4.2), only reference fields 2, 3, 4. and 12 are defined. Reference field 2 is a syntactic pointer according to the table entry in figure 4.2. Reference field 2 contains the value 30. Therefore the field represents a syntactic pointer between this node (with a sequence number of 31) and the node with sequence number 30. Similarly, reference fields 3 and 4 contain non-zero values and thus connect this node to other nodes in the tree. However, reference field 12 contains a zero. According to the table entry in figure 4.2, reference field 12 is a link pointer. So the link pointer is unused for this particular occurrence of a node with node Kind number 250.

The first boolean field and the first value field are defined according to the table entry in figure 4.2. The first boolean field in the node carries a non-zero value (that

is, the value 1). In the DIANA trees produced by the U.C.T. front end processor, this indicates that the first value field contains a numeric value. In the example, the first value field carries the numeric value 23 (represented as .23000+002).

Finally, the table entry in figure 4.2 indicates that the string field is defined. There is a string attached to the node ('23'). In this example, the string is the alphanumeric representation of the numeric value in value field 1.

The code template table entry for this node (figure 4.4) informs the code generator what to do when visiting the node. The effects of all the <details> items possible in a code template table entry are covered more fully in Chapter 6 (The Code Generator).

What follows is an example DIANA subtree as produced by the U.C.T. front end processor. The example is the same as that given in Chapter 2 (Background Information). It represents the Ada assignment statement 'A := B * C;'. The diagrammatic representation of the tree is repeated in figure 4.7.

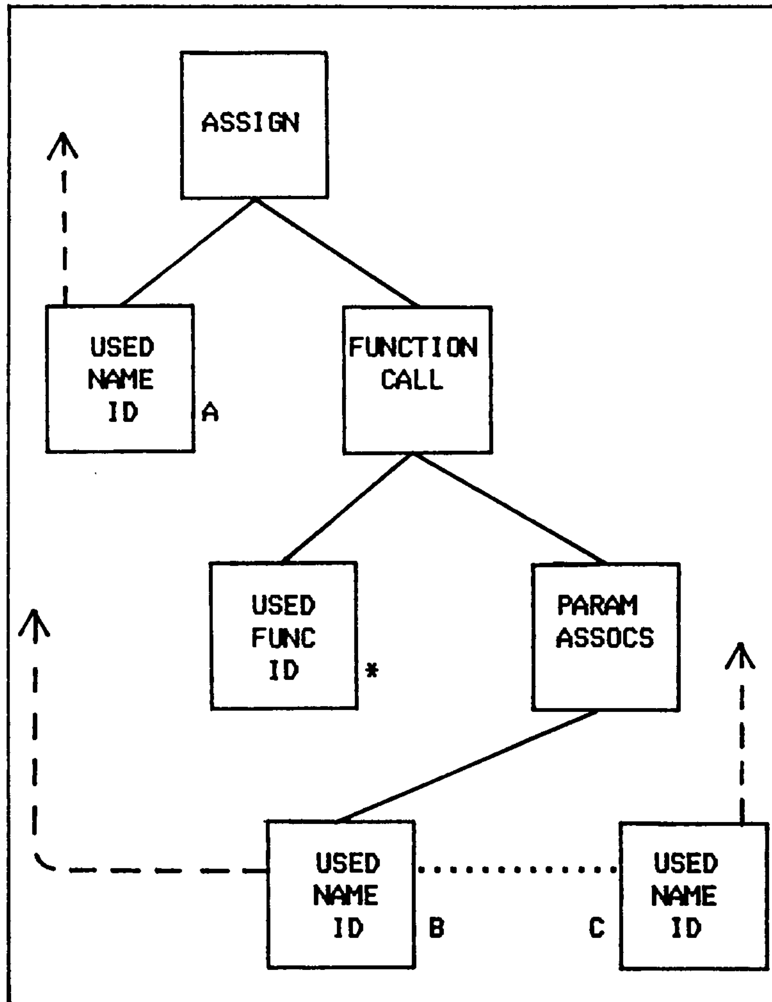


Fig. 4.7: A DIANA Subtree Representing 'A := B * C;'.

The subtree for 'A := B * C;' is output by the front end processor in the form of a sequential text file. The format of that text file is illustrated in figure 4.8.

.
.
.

157	29	19	0	0	0	0	0	0	0	0	0	0.00000	.00000	00A
13	30	29	32	0	0	0	0	0	0	0	0	0.00000	.00000	00
157	31	20	0	0	0	0	0	0	0	0	0	35.00000	.00000	00B
62	32	33	34	0	0	0	0	0	0	0	0	0.00000	.00000	00
138	33	0	0	0	0	0	0	0	0	0	0	0.00000	.00000	00*
109	34	31	0	0	0	0	0	0	0	0	0	0.00000	.00000	00
157	35	21	0	0	0	0	0	0	0	0	0	0.00000	.00000	00C

.
.
.

Fig. 4.8: Example DIANA Subtree as Produced by the U.C.T. Front End Processor.

Figure 4.8 depicts a real DIANA subtree. In the figure there are many examples of the reference field values referring to the sequence numbers of other nodes in the subtree, thus linking the nodes. Some reference fields in the tree carry sequence numbers for which no nodes are depicted. Remember that the subtree is a fragment extracted from a complete tree. Therefore some nodes are not shown, such as those which define the identifiers.

CHAPTER 5

TRANSLATORS AND CONVERTERS

Both the tree specification table and the code template table are in a format which is readable by humans. This facilitates their easy creation, but is not conducive to automatic analysis. So translators are provided to turn the information contained in these user written tables into a machine readable form. These translators take as input the tables in the format described in the last chapter and produce, in sequential output files, machine readable versions of the tables. The converters in turn take these sequential files as input and produce direct file versions of the machine readable tables as output. Also, the tree output produced by Epstein's front end processor is in a sequential file. So a converter is provided to produce direct file versions of the trees. The code generator uses the three direct files as input.

To summarize, the programs described in this chapter prepare the input for the code generator.

5.1 Tree Specification Table Translation

This sub-chapter describes the operation of the translator and converter for the tree specification table.

5.1.1 Input File

The input file for this translator is sequential and contains the tree specification table. Each line in the table must be no longer than seventy nine characters. TAB characters and blank lines may exist in the table.

5.1.2 Translator

The table is processed in a single pass, line-by-line manner. At the start of each node, the translator expects the node name, node kind number, an attribute item and reference item. The node name is discarded as the node kind number is used to identify the definition. But this node name must exist, otherwise an error message is generated and the rest of that node definition is not recognized.

After the first line the translator expects only reference items to exist on each line. The node name and kind number are no longer expected. Attribute items are optional as the last attribute encountered will be assumed for every reference item thereafter until the end of the node or another attribute item in the node definition. The end of the node definition is indicated by a ';' character. Comment lines, identified by a ':' character in the first column in the input table, are treated as blank lines and skipped.

All the item names in this table are unique beyond the first two characters. So the translator takes only the first three characters as significant, skipping over the rest. Therefore, if wished, it is possible to have all item names consisting of only the first three characters. This facility is provided so as to minimize typing effort for users familiar with the system.

All reference items, except for rep, have an attached numeric field. So for each line in the tree specification table there exists three pieces of information, namely the attribute item, the reference item and the reference numeric value (except for rep). There are few enough item types to permit each to be represented by a single alpha character. In addition, in this table no reference number is longer than two digits. It is estimated that the likelihood is very low for there to be more than forty reference

items in a tree specification table node definition. So a reasonable output format to have for each node is a Kind number on a single line followed by three lines, one containing single letters representing the attribute items, one containing single letters representing the reference items and the last containing the two digit numeric reference values. The elements with the same ordinal number in each of the three output lines represent the translated version of the input line with the same ordinal number in the input node definition.

Figure 5.1 gives a diagrammatic representation of the translated node definition structure.

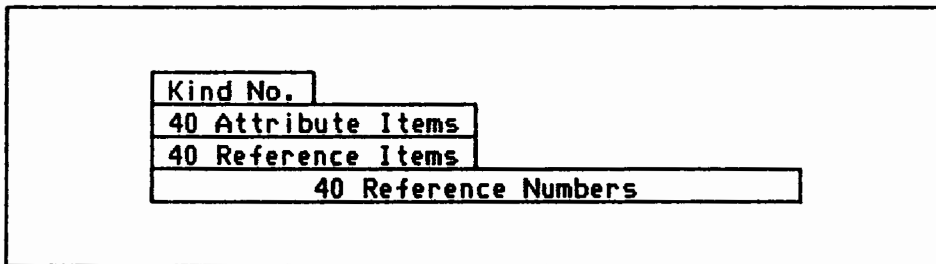


Fig 5.1: Structure of the Translated Tree Specification Table Node.

Remember that the attribute items in the input table need be mentioned only once for an unbroken list of reference items with the same attribute. But in the output lines that attribute item's character representation will occur for each of the aforementioned reference items' character representations. Note that in the case of the rep field, the numeric value is simply ignored. The order of the entries in the output table results directly from the order in the input table. But this does not affect the operation of the code generator, as this table is used only for verification of existence, not order. The order of the entries is important only in the code template table.

Although the lines are hardly ever filled, all remaining space in the attribute and reference lines is padded out with '.' characters. Also, all unused reference number fields contain zero digits. This padding is to allow storage of the lines in a direct file, where the size of stored data structures must be regular. The choice of the '.' character is only to permit easier visual examination of the structures in the output file.

So as to illustrate a translation, the example tree specification table node entry given in Chapter 4 (The Tables and Tree, figure 4.2) is fed to the translator. This example table entry is repeated here in figure 5.2.

```
EXAMPLE 250  as-synpntr  reference(2)
             as-synpntr  reference(3)
             sm-sempntr  reference(4)
             sm-valbool  value(1)
                         boolean(1)
             lx-string   rep
             link        reference(12);
```

Fig. 5.2: Example of a Tree Specification Table Node Definition.

Figure 5.3 shows the output resulting from the translation of the table entry illustrated in figure 5.2.

```
 250
AASSSXL...............   ...
FFFVBPF...............   ...
 2 3 4 1 1 012 0 0 0 0 0 0 0      0 0 0
```

Fig 5.3: Example of a Translated Tree Specification Table Node Entry.

The three output lines are stored internally in arrays. The node number followed by the three arrays are reproduced in the output file only when the end of the associated node definition is reached.

During the scan of the table, an echo of each line is reproduced on the standard output of the computer, along with any error(s) detected for each line. The errors detected are listed underneath the relevant line in the trace, along with a '^' character pointing to each error in that line. Note that as each line of the input table is read in, all TAB characters are converted into the equivalent number of blanks before the scan of that line commences. This is done to avoid confusing the error position detector. Although this TAB conversion is normally not visible, a (Unix) redirected output trace file will occupy significantly more space than the input file if many TABs are used.

The above translation process is repeated until there are no more node definitions in the table.

Before use can be made of the above output table, it must be converted to a direct file format. This is done by a C sequential file to direct file converter program. It simply reads in the Kind number and three lines of information for each table node, calculates a direct file address from the Kind number and the space taken by the above

information, and writes that information into the direct file. Note that the kind number is also written. This is for any future code generator verification of the table information read.

The C language is used for the conversion program as Pascal's file handling capabilities do not (normally) include random access or direct files.

5.2 Code Template Table Translation

This chapter describes the translator and converter for the code template table.

5.2.1 Input File

The input file for this translator is sequential and contains the code template table. Each line in this table must be no longer than seventy nine characters. TAB characters and blank lines may exist in this table.

5.2.2 Translator

As before, this table is processed in a single pass, line-by-line manner. At the start of each node, the translator expects the node name followed by the node kind number and then a details item (either a command word and any associated conditionals, code templates or command words, or a pointer indicator). Like before, the node name is discarded and the node kind number is used as the index for this table. After the first line only details items occur until a ';' character which indicates the end of a node

definition. Comment lines, identified by a ':' character in the first column, are treated as blank lines and skipped.

The code template table is more flexible in layout than the tree specification table. The extra freedom is required because, whereas the specification table describes only what exists in terms of pointers and fields for each tree node, the template table must control the scan of a tree as well as the code produced. In addition, the tree specification table consists of far fewer words in its vocabulary. This freedom manifests itself in the placement of conditionals, command words and code templates within the special command words. Flexibility in the layout results in better control of generating code, especially with reference to the code templates,.

There are many more items in this table than in the tree specification table. So, not surprisingly, many more characters need to be taken as significant before the items can be seen as unique. The translator takes only the first eight characters as significant. The remaining characters are skipped. Although more characters need to be typed than with the tree specification table, it should still prove advantageous for users familiar with the system.

Given the large number of command words in this table vocabulary, a single character representing a command word is inadequate. So character pairs are used. To help when debugging the system, use is made of the non-alphanumeric characters '+', '$' and '&'. These categorize the command words. For example, all the command words with attached numeric values are prefixed by a '+' character. But all this is transparent to the user and does not affect the use of the system.

The nature of the input format of this table precludes an output format like that for the specification table. Instead, the output for this table consists of the kind number of the node definition on a single line, followed by a single line, up to one thousand eight hundred and twenty four characters long. This length is arbitrary and based upon observation of the requirements. In the details line are stored all the symbol pairs representing the command words and link indicators, four digit numeric values, conditionals and code templates.

A diagrammatic representation of the translated node definition is illustrated in figure 5.4.

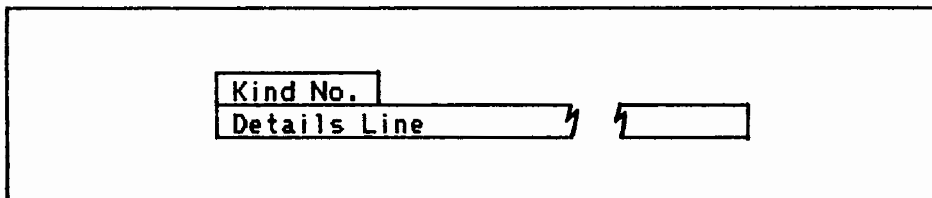


Fig. 5.4: **Structure of the Translated Code Template Table Entry.**

Although the details line is hardly ever filled, the remaining space in the line is padded out with blanks to the maximum length. This padding is to permit storage of the line in a direct file format, where the size of the stored data structures must be regular.

For each command word or pointer indicator encountered, a character pair is generated and put into the details line in the order in which they are encountered in the input table. Any attached numbers follow the associated character pair.

The numeric values in this table can be up to four digits long and carry an (optional) '+' symbol or '-' symbol to indicate the sign of the number. This sign does not take the space of a digit. Very often, the full range possible for a numeric value would be

illegal. But range checks in the code generator prevent incorrect values being used. When put into the details line, the numerals take up four spaces to allow for their maximum size. Any sign takes an additional space just preceding the numerals.

All code templates and conditionals are placed in the details line verbatim from the input table. These are for processing by the code generator and cannot be modified in the translator.

The code generator scans this line from the beginning and executes the instructions in the order encountered. That is, the order in which the code generator encounters the command words and link indicators results directly from the order in which the details items were entered in the input code template table. This means that, unlike the tree specification table, the order of entry in the code template table is of extreme importance in the control of the code generator.

So as to illustrate a translation, the example code template table node entry given in Chapter 4 (The Tables and Tree, figure 4.4) is fed to the translator. This example table entry is repeated here in figure 5.5.

```
EXAMPLE        250      null
   code an unconditional template:
                        SynReference(2)
                        Lflag1Set
                        SynReference(3)
                        LStore2Reset(9)
                        SynReference(2)
                        Lflag1Clear
                        SemReference(4)
   comm ?T12f           Lflag10Set
                        SynReference(2)
                        Lflag10Clear:
   code ?B1t a conditional template:
                        SynReference(12);
```

Fig 5.5: Example of a Code Template Table Node Specification.

Figure 5.6 shows the output resulting from the translation of the table entry illustrated in figure 5.5.

```
0250
+N+C AN UNCONDITIONAL TEMPLATE:+A    2$K
+A    3+T    9+A    2$a+S    4+P?TL2F&m+A
 2&p:+C ?B1T A CONDITIONAL TEMPLATE:+A
 12;
```

Fig. 5.6: Example of a Translated Code Template Table Node Entry.

From the example it can be seen that the characters forming code templates and conditional structures are converted to upper case within the bounds of the 'code' special command word. The same is true for the conditional structures within the 'comm' special command word. This is done to avoid any complications involved with case sensitivity.

As with the specification table translator, a line by line trace of the input is reproduced on the standard output of the computer, along with any errors indicated underneath the relevant line. Also, as before, TAB characters are converted to the equivalent number of spaces.

The scan of the lines in a node definition continues until a ';' character is encountered, indicating the end of that node definition. To indicate the end of the definition in the details line, a ';' character is put at the end of the list of characters thus far entered in the line (but before any padding blanks). The details line is stored internally in an array and is reproduced in the output file, along with the node kind number, only when the end of the associated node definition is reached. The scan then restarts all the above process for the next node definition. This loop continues until there are no more node definitions to process.

As with the other output table, this table needs to be converted to a direct file format before used can be made of it. This is done by another C sequential file to direct file converter program. The process here is the same as before except that the address and size is worked out for the kind number and details line. The kind number is stored in the table for future verification by the code generator.

5.3 Tree Conversion

There is no need for a translator for the tree (the source program has effectively been 'translated' by the front end processor). All that is needed is a sequential file to direct file converter because the trees, as produced by the U.C.T. front end processor, are in sequential files. This converter is similar to the others. It is a C program and

reads the tree, node by node and rewrites these nodes in a direct file. The difference here is that the node sequence number is used as the key instead of the node kind number.

CHAPTER 6

THE CODE GENERATOR

In this chapter, a detailed examination is made of the back end processor's central program. It is this program which can produce an object code representation of an input tree by using the information provided in the tables. Aside from being able to walk around a tree (according to information in a table) it also has a number of built in functions which can be invoked by the use of command words. In addition, it evaluates any conditional structures included in the code template table so as to select the correct pointers to follow, command words to execute and code templates to print.

This is a large chapter, so a short description of what follows is in order:

(1) summary of the input to the code generator,

(2) description of the important data structures,

(3) description of the conditional structures,

(4) description of the pointer indicators,

(5) description of the command words,

(6) description of the code templates,

(7) description of the tree traversal process, followed by a simple example.

6.1 Tree and Table Input

The input to the code generator is taken from the three direct files produced by the sequential file to direct file converters. These files hold the (translated) tree specification table, the (translated) code template table and the tree to be scanned. Most Pascal implementations have little or no direct file handling capability. To bypass this problem, the code generator has special purpose C routines linked in which enable the opening and reading of the direct files.

What follows are short summaries of the input details of the tables and tree.

6.1.1 Tree Specification Table Input

All references made in the code template table to the fields of a tree node are compared with the specification for that node in the tree specification table. This operation highlights any discrepancy in the corresponding table definitions.

In operation, the definition of a node in this table acts as a mask through which the corresponding tree node is viewed, thereby indicating what fields of the node carry information and what the (existing) reference fields represent.

Note that the tree specification table identifies what the reference fields are for each

node, but it indicates only the presence or absence of all the remaining fields (that is, the value, boolean and rep fields). Information about the usage of these other fields is found in the code template table.

The node definitions are drawn from the translated direct file form of the table. The key for each node definition in this table is the kind number. This is also the access key for the node definitions when the table is stored on disk in a direct file format.

6.1.2 Code Template Table Input

The code template table serves four main purposes:

(1) it provides all the syntactic and semantic pointer information necessary to enable the code generator to correctly traverse a tree,

(2) it provides all the code templates necessary for the generation of the output code,

(3) it carries all the command words and conditional structures which provide additional information to the code generator about the nodes encountered during the traversal of the tree,

(4) it contains the initialization node (node 0) which carries all the initialization values and native type attributes.

All references made to tree node fields in the code template table are verified with the information about those nodes in the tree specification table. This highlights any discrepancy between the definitions of the (same) nodes in each table.

If necessary, a syntactic or semantic pointer indicator can exist more than once within the same node definition. Such an ability is essential because in DIANA a pointer often has to be traversed more than once during a visit to the node from which that pointer emanates. Such multiple use of the same pointer does not have to be reflected in the tree specification table. If this was not the case, the tree specification table could be rendered useless for anything other than this code generator.

Like the tree specification table node definitions, the key to the code template table node definitions is the kind number. This number is also the direct disk file access key.

6.1.3 Tree Input

The tree input is the intermediate form of a source program. It is this tree which is scanned to produce the output code representing the tree and hence representing the original source program. It provides all the nodes and associated information necessary for the code generator, guided by the information in the tables, to walk around the tree and produce the equivalent output code.

The nodes are drawn from the converted output of the U.C.T. front end processor. The key for each node is the sequence number. This is also the access key for the nodes when the tree is stored on disk in a direct file format.

6.2 Important Data Structures

There are a number of data structures which are used regularly with the conditional structures and command words. In addition, a simple symbol table exists to hold all the information which cannot be extracted from a tree at any random time during the tree's traversal. It is important to know about these data structures if the operation and use of the code generator is to be properly understood.

To help in the following data structure descriptions, certain command words and conditional structures are briefly mentioned. More detailed descriptions of all the command words and conditional structures, and their effect on the data structures, follow later in this chapter.

6.2.1 Symbol Table

It is desirable to obtain as much information as possible directly from the DIANA tree with minimum use of a symbol table. But there are some items of information which cannot be readily accessed whenever randomly required. The lexical level at which an identifier is defined is a good example of such an item. There is no obvious way to obtain this information from the tree structure at a time after traversing the identifier's definition subtree. The lexical level at which the identifier is defined can be determined only at the time of traversal of its definition subtree. Therefore, the only way that this information can be accessed at some later time is by the use of a simple global symbol table.

The P-code data segment offset value of a data structure is another item of information which cannot be obtained by a random access of the tree. So an integer

field is provided in the symbol table for this. Likewise, the ordinal number of each defined procedure causes problems similar to the data structure. So when referring to a procedure (or function) name, this same field is used to carry the ordinal number of the procedure. Such use of the field for two different items helps to maintain the simplicity and small size of the symbol table. Use of the symbol table is not mandatory. But it would be impossible to generate P-code without it.

The symbol table takes the form of a linked list of nodes. As the lexical level is incremented, so a new node is added to this list. When the lexical level is decremented, so the most recently added node is discarded. The first node in this list is the global level node. Aside from the pointer field which attaches a level node to its neighbor, two fields exist within that node. The one field holds the lexical level of definition of that node. The other field carries a pointer which links to another linked list. Each of the nodes in this other linked list carry the various identifier attributes which cannot be randomly drawn from the tree.

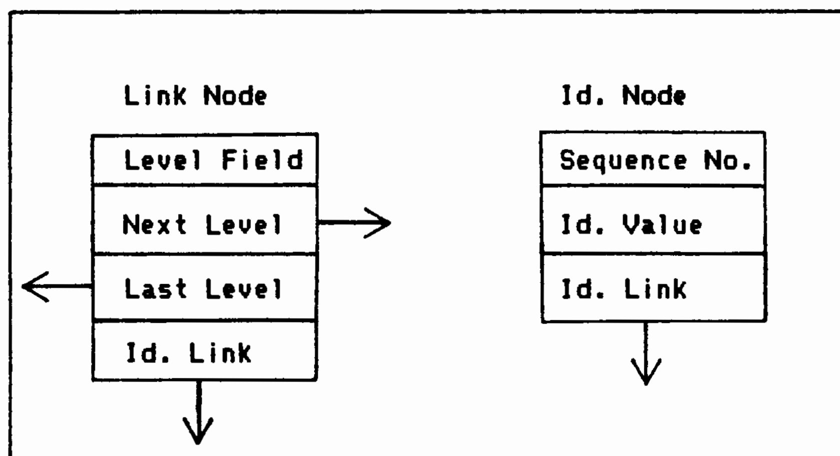


Fig. 6.1: The Symbol Table Nodes.

Each of the attribute linked lists carry the additional attributes for all the identifiers defined at the levels represented by the level nodes to which the lists are attached.

If the attribute node represents a data structure identifier, the value field carries the data segment offset value of that structure. If the identifier is for a procedure (or function), the field carries the ordinal number of that procedure. Another field in this node carries the node sequence number of the identifier node. This is provided to allow the system to find the attribute node of interest. Use of the node sequence number as a key eliminates the need to carry the string which represents that identifier in the symbol table. Use of the string instead would be wasteful of space. It could also result in a limitation because of the searching algorithm that would be necessary to avoid confusion when an identifier name is overloaded. Anyway, the front end processor should already have resolved any scoping problems and overloading definitions (as Epstein's does) and such effort should not be duplicated.

Finally, a field is provided to link the attribute nodes together. The resulting linked list carries all the additional attributes for the identifiers defined at the level represented by the level node to which the list is attached.

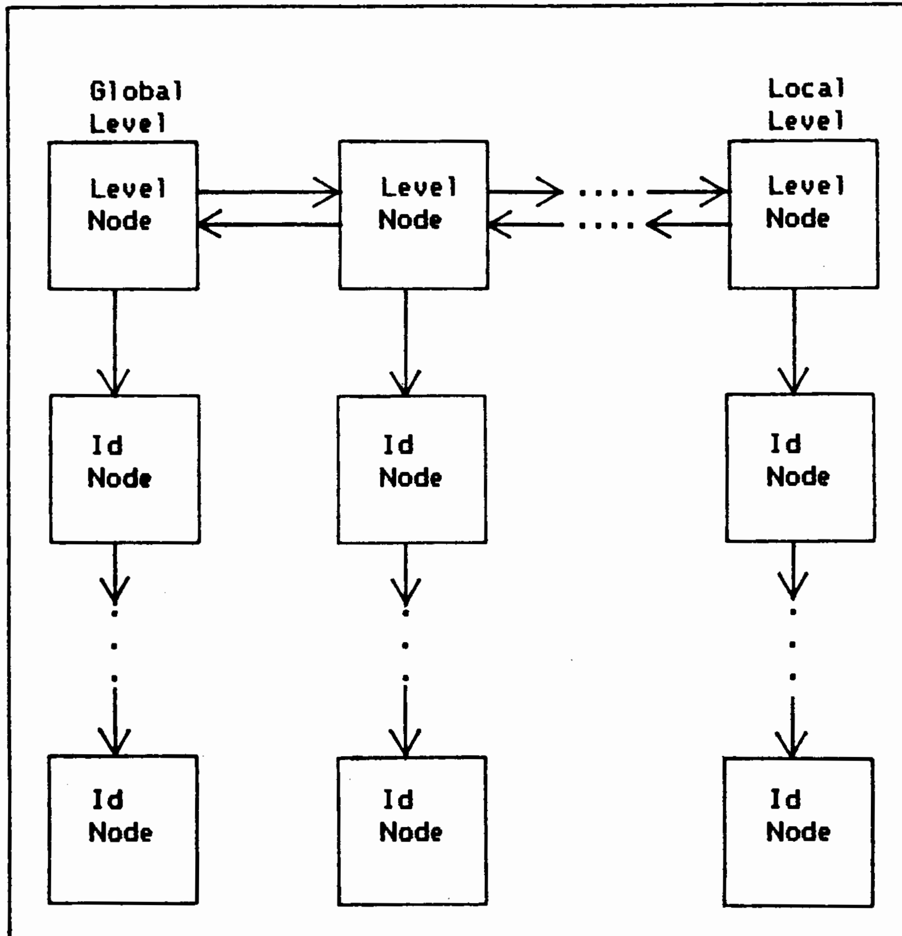


Fig. 6.2: Structure of the Symbol Table.

Whenever the lexical level is decremented, and the associated level node is discarded, then all the identifier attributes for that level are also discarded. This mechanism maintains the necessary scoping levels for all the identifiers. Use of the table in this manner is purely optional. If, for example, everything must be held at one level, then the lexical level is simply neither incremented nor decremented.

6.2.2 Native Type Identifier Store

When a data structure is defined, no matter how complex it is, it must eventually consist of native data types and structures (natives). So information about these

natives must exist within the code generator. But, in keeping with the idea not to tie the system down to a specific implementation of a tree language, this native information is entered via one of the tables. This is achieved by using node definition 0 in the code template table to carry all the information about the natives.

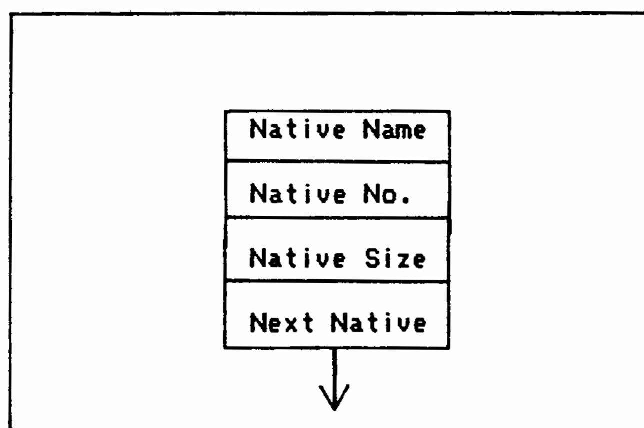


Fig. 6.3: The Native Store Node.

The data structure which holds the native information is in the form of a global linked list, similar to the attribute linked lists which 'dangle' from the level nodes. In addition to holding the name and data segment size of each native identifier (as entered through the 'natives' word in node 0, described later), a unique sequence number is calculated and stored for each native. This number is used by the type conditional and its use is transparent to the user.

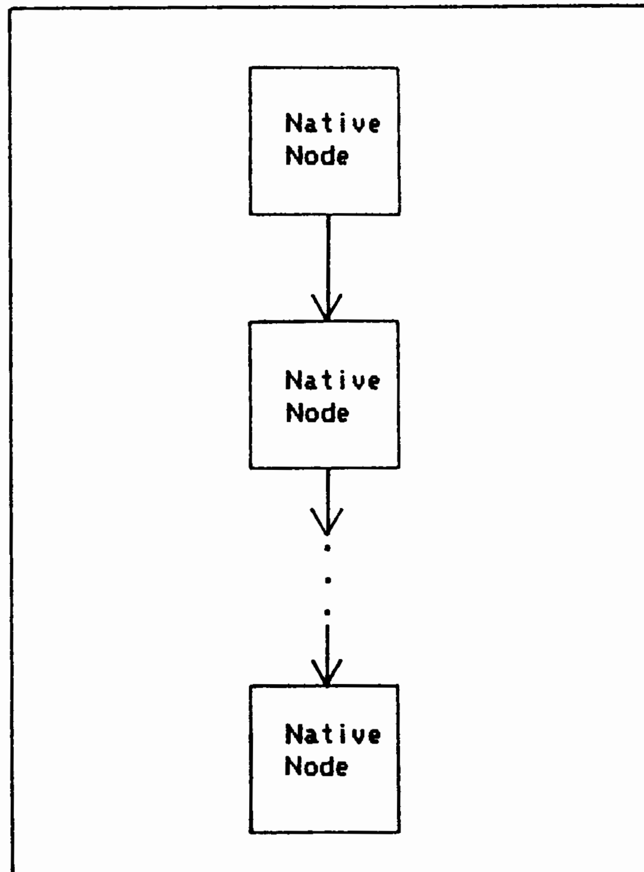


Fig. 6.4: Structure of the Native Store.

6.2.3 Current Identifier Store

Whenever information about an identifier is to be accessed for any reason, all its attributes must be first loaded into a temporary or 'current' store. These attributes consist of all the information contained in the symbol table about the identifier as well as additional information provided by the identifier's base type. An identifier, for which the attributes have been loaded into the current store, is referred to as the current identifier.

The current store is global to the whole tree and takes the form of a record containing

the necessary fields. It is loaded from both the symbol table and the native store. The fields for the information from these sources all exist in the store.

In other words, the store can be seen as having two components. One component carries all the identifier attribute information loaded from the symbol table. The other carries native attribute information loaded from the native store. To make most effective use of the store, the symbol table attributes of the current identifier are matched with the attributes of the most recent native type definition for that identifier. This can be effected by controlling the walk around the tree such that the most recent associated native type is loaded whenever the identifier is loaded. This is, of course, optional. Should there be any reason to load one component without the other, or with a different matching component, it can be done.

To provide more visible control of this store, provision is made to clear it by the use of a command word. This is most useful for the level conditional. If the store is cleared (and optionally followed by a native load without an identifier load) and a level decision based on the attributes in the store is then attempted, a warning is given. The conditional, and all it contains is then skipped. Note that this is not an error condition and actually may be sometimes desired. But the warning message is provided to highlight such use so that the user is aware of the situation.

6.2.4 User Flags

When developing the system, it was soon apparent that DIANA is not the ideal intermediate language to convert to P-code. Take, for example, they way DIANA subtrees representing identifiers are used not only for definition purposes but also during the statement phase. To make the system flexible enough to handle this, a

mechanism must exist to allow arbitrary control over which attribute pointers to follow, code templates to select code for generation and command words to execute. In conjunction with a conditional structure, a set of fifteen general purpose user flags provide such a mechanism. Control is implemented by the user including instructions in the code template table to set and clear these flags. The User Flag conditional structure is then able to decide what to do on the basis of the state of the flags. In order to add to this flexibility, three of the flags are global to the whole tree, but the remaining twelve flags are local only to the subtrees from the node where they were last affected. When the walk around the tree exits from such a subtree, the local flags again take on the value they held before being modified. In the case of the global flags, any change of state is reflected at any point in the traversal of the tree.

The general purpose user flags are sure to be useful in the control of activities for most input tree specifications.

6.2.5 User Stores

There are occasions when it is necessary to hold numeric values for use in the generation of the P-code. One example of such a need is the production of jump labels for unlabeled jumps. For this, values which can be manipulated have to be held in internal stores in order to produce the correct jump labels. To allow for such needs, eight general purpose user stores are provided. These stores are partitioned into four local stores and four global stores, similar in manner to the user flags. But, in this case, the values in the local stores can be transferred to the global stores with the same ordinal number, and vice versa. This facility is necessary so that, when using the stores to hold label numbers, a record of the highest number used to date can be held in the global store. Without this, the highest number used in a subtree would

be lost when that subtree is exited. But the local store must also be used to maintain the number which was current before the subtree was entered.

The manipulation of the values in the stores is implemented by the use of command words. Also, aside from being used to hold values (for any use, not just jump labels), the stores are used in conjunction with the comparison store and a conditional. Both the comparison store and the aforementioned command words are covered later in this chapter.

Like the general purpose user flags, the stores may be used for any purpose and are not isolated to holding jump values. Little in their design is specific to the jump values.

6.2.6 History Array and Index

With this structure, the sequence in which nodes of interest occur can be recorded. With such information, a selected sequence of operations can be executed, or a selected sequence of P-code Instructions can be generated.

The history array, and associated index came about because of the need to record the sequence of parameters as they occur in the parameter definition and noting whether they are IN, OUT or IN-OUT parameters. This information is necessary to generate the correct call-by-reference or call-by-value P-code for the variable identifier parameters in the parameter list of a called procedure. The DIANA trees provided by Epstein's front end processor provide no obvious alternate technique to acquire the necessary information for the generation of such P-code.

As before, an attempt was made to keep the structure and associated operations as uncommitted as possible. There is no specializing of this structure to the parameter list problem. So it can be used for any problem requiring the features provided by the history array and index.

The structure takes the form of a single dimension global character array of 128 elements, with a matching index to control access to the structure. The structure is initialized to contain only blank characters, but each element can be individually accessed and can be set to any alphanumeric character or unreserved non alphanumeric character. A conditional structure is available to test the indexed element and can make decisions based upon the success or failure of the comparison. The index can be controlled by command words.

6.2.7 Comparison Store

There are many instances, when generating object code, where it is useful to compare values with different stores or with node sequence numbers and node kind numbers.

One such case is when wishing to determine the type of a numeric literal in DIANA. Attached to the numeric literal node is a special pointer (the denoted type pointer - covered in Chapter 4) which identifies the literal's type. This value is loaded into the comparison store for testing. Based upon the test, the appropriate P-code can be then produced for the literal.

This structure simply takes the form of a global integer store, similar to one of the global general purpose stores. What makes this store so versatile are the associated command words and conditional structure. The command words cause the loading of

various values to the comparison store. The conditional structure has a number of variations which allow decisions to be made by comparing the value in the comparison store with, among others, the general purpose stores and the current node and Kind numbers.

Like many of the other data structures, this store is not specific to any particular DIANA structure and is general purpose.

6.2.8 Offset Stores

In order to Keep a record of the highest P-code data segment offset value during the definition of a procedure or function, three simple global integer stores are provided. Although one such store could do the job, three are provided so as to allow a separate record to be Kept of the parameter offset and local data structure offset values, as well as the total offset if so wished.

The offset stores are used as the source for the offset value of a data structure currently being defined when adding its identifier to the symbol table. Any of the offset stores can be used for this job.

In order to copy an offset value into the symbol table and to update the offset values, command words are provided. In addition, the values held in the offset stores can be produced in the code output file. They also can be used, in conjunction with the comparison store, in a conditional structure. This conditional makes decisions on the basis of a comparison between the values in the comparison store and one of the offset stores.

There is no obligation to use the offset stores but they are indispensable for generating P-code.

6.2.9 Procedure Count Store

Similar in function to the offset stores, the procedure count store records the maximum number of procedures defined to date. This store is a simple global integer store.

The procedure count store is used as the source for the ordinal number of the procedure (or function) currently being defined when adding its identifier to the symbol table.

Command words are provided to copy the value from the procedure count store to the symbol table and to update the value contained in the procedure count store. In addition, the value held in this store can be produced in the code output file. The store also can be used, in conjunction with the comparison store, by a conditional structure in a manner identical to such usage of the offset stores.

As with the offset stores, this data structure is optional but it would be difficult to produce correct P-code without it.

6.2.10 Control Flags

The control flags are provided to allow control over major aspects of the code generator. The exact function and effect of these control flags is described in more detail later in this chapter. The flags exist in the form of four independent local flags. They are local in a manner similar to the local general purpose user flags.

6.2.11 Stash

At each node encountered in the tree, many operations are often done involving the flags, stores, current identifier and other data structures. But it is not always desirable to keep these changes when such nodes, and the subtrees for which they are roots, are exited. So for each node it is necessary to have a store into which the values of the aforementioned data structures can be put. At some later date before leaving this node and its attached subtree these values can be restored to their appropriate data structures.

It is because of the recursive nature of the walk around a tree that such a data structure can be so easily implemented. It takes the form of a record which consists of all the necessary fields. This record is local to the recursively called procedure in the code generator which does the actual 'walking' around the tree. For each node, a fresh call is made on the recursive procedure. Therefore, at each node, such a record exists which is local only to that node.

There is a set of command words which cause the 'stashing' and restoring of all the data structures which may require saving. Note that these stash commands work in groups. For example, a stash operation on the local flags causes only these flags to be stashed. Similarly for the restore commands, a restore operation on the local flags causes only these flags to be restored.

Except for the symbol table and native type store, all the above data structures may be stashed and subsequently restored.

6.3 Conditional structures

During the development of the code generator it soon became obvious that a simple unconditional walk through a tree, without any form of control, would be too inflexible. The same applies to the unconditional generation of code and the unconditional execution of command words. So it is necessary to include a variety of conditional structures which can control the walk through a tree. These same conditional structures can also control the selection of the code to generate or the selection of a command word to execute from a number of options.

With the conditional structures, decisions can be based upon the following variable parameters:

-- the type of the current identifier,

-- the lexical level of definition of the current identifier,

-- the state of a set of general purpose (user setable) flags,

-- the character contained in an indexed history array element,

-- the value contained in the comparison store,

-- the previous or next syntactic or semantic pointer being a null pointer,

-- the state of the node boolean fields,

-- the string in the rep field.

When a conditional is evaluated and the result is true, the scan continues to whatever is next, be that another conditional, a code template, a pointer indicator or a command word. But If the conditional is evaluated as false, then everything between the point of that conditional, and the next ':' or ';' character is skipped. In this manner, the conditional structure can control the important aspects of the tree traversal because pointers can be followed or skipped, command words can be executed or skipped and code templates can be generated or skipped.

Note that the conditional structures can exist only within the bounds of two special command words. Although detailed descriptions of all the command words follow later in this chapter, it is felt that these special command words should be explained now because of their importance to the conditional structures.

6.3.1 Special Command Words

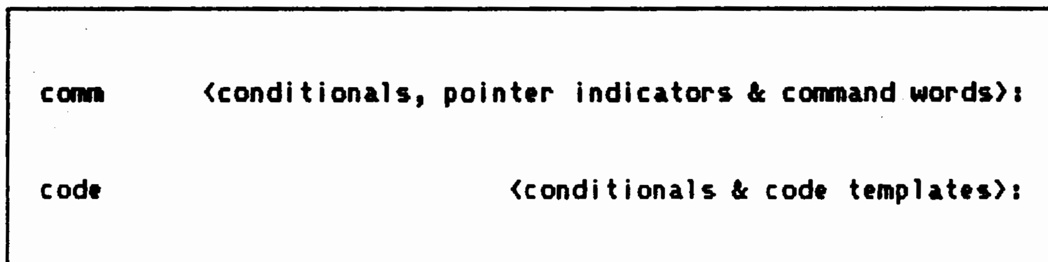
The two special command words are as follows:

-- 'comm' - this word carries the conditional structures controlling the command words and pointers,

-- 'code' - this word carries the code templates and any conditional structures controlling them.

The system must be kept as free as possible from being tied down to any form of output code. As a result, what goes into the templates (aside from the template substitute operators and reserved characters, described later) is of no relevance to the code generator's operation. So some mechanism is included to tell the system what the code templates are. This is done by 'enclosing' the templates within the bounds of the 'code' special command word. Any conditional structures affecting the generation of the P-code also must be included within 'code'. It is mandatory to use 'code' wherever any code templates exist irrespective of whether conditional structures exist or not.

Unlike the templates, the command words are well defined and known to the code generator. In this case, such an identifying mechanism need not be employed. Therefore, in order to conserve space, 'comm' need only be used wherever any conditional structures are used to control command words. Otherwise, the command words can exist on their own in the code template table.



comm <conditionals, pointer indicators & command words>:

code <conditionals & code templates>:

Fig. 6.5: Structure of the 'comm' and 'code' Special Command Words.

If the end of a 'code' or 'comm' special command word coincides with the end of the code template node definition then the ';' character, indicating the end of the node definition, doubles as the end of special command word character. In such a case the ':' character need not be used.

It is not possible to nest 'comm' special command words, or to include 'code' words within 'comm' word bounds. This does not prove to be a disadvantage, as all reasonable conditional combinations can be achieved by the separate use of the 'comm' and 'code' words. Any attempt to nest the above words will be detected and flagged as an error by the code template table translator.

Most of the conditional structures owe their existence to the need to select, from a number of options, the correct P-code. However, all conditional structures can be used within both special command words. Thus, although possible, there is no need at present for some conditional structures to control some command words. But there may be a future requirement for such combinations, hence their inclusion.

To indicate the existence of all conditional structures, except the type select conditional, a '?' character is used. This is followed in turn by a letter which identifies the actual conditional structure. Therefore the '?' is a reserved character and cannot be used in the code templates.

It is possible to nest the conditional structures (except for the type conditional, which may contain no conditional structures). Some structures may be recursively nested. But for conditional structures where such recursive nesting is meaningless, it is not permitted.

What follows are descriptions of the function and syntax of each conditional structure. In these descriptions, a brief reason is given for the inclusion of each conditional. But remember that all these structures can be used within both special command words, even though that is not stated in the description. The word 'option' is used to indicate the presence of:

-- a code template,

-- a syntactic or semantic pointer,

-- a command word (excluding the special command words 'code' and 'comm'),

-- a conditional structure (except for the type conditional).

6.3.2 Type Conditional

It is necessary to be able to produce different P-code for different identifier types. A '/' symbol preceding the first piece of P-code indicates that there follow a number of P-code options, each delineated by a '/' symbol. The structure is depicted in figure 6.6.

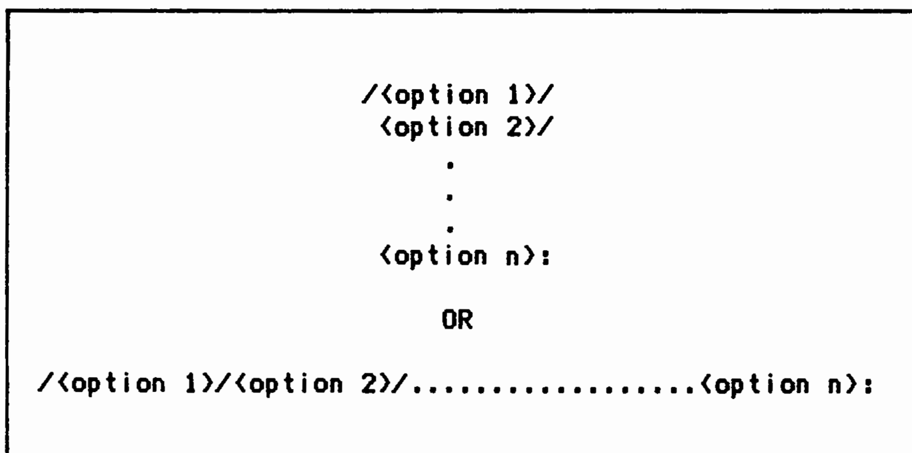


Fig. 6.6: Structure of the Type Conditional.

The number and order of the options must match the number and order of the native types entered in the P-code table, as there is a mapping of the first option to the first native type, second option to the second native type, and so on.

Note that no conditionals may exist within this structure. It was initially tried, but it proved to be untidy and even unreadable for complex options. So such a combination is excluded. This does not restrict the operation of the conditionals, as all reasonable conditional combinations can be produced without this denied ability.

If a type option does not need any code to be generated, command words to be executed or pointers to be followed then a '*' symbol should be substituted in that option's location. Although no character need be inserted there when in the 'code' segment, the '*' symbol suppresses an automatic line feed which otherwise would be printed (in the code output file)when at the end of the segment. Obviously, the line feed suppression is not applicable to the 'comm' special command word.

It is important to remember that this type selection depends solely upon the type of the most recent native type load encountered during the traversal of the tree.

6.3.3 Lexical Level Conditional

In P-code, the instructions used to manipulate an identifier are dependent upon whether that identifier was declared at a local, intermediate or global level. Therefore a conditional structure is included to allow the generation of code dependent upon the aforementioned lexical level comparisons. The structure of this conditional is given in figure 6.7.

```
?Ll                                   <option l>/
?Li                                   <option i>/
?Lg                                   <option g>:
```

Fig. 6.7: Structure of the Level Conditional.

The leading '?L' symbol combination (in the first line above) identifies the structure. The letters 'l','i' and 'g' following the '?L' symbols indicate that the associated code options should be printed (in the code output file) if the current identifier is (respectively) defined at a local, intermediate or global lexical level.

The lexical level at which the current identifier is defined is checked when this conditional is invoked. If this identifier is not global, then its lexical level is compared with the current lexical level to determine if it is defined locally or at an intermediate level.

Remember that the current identifier is the one most recently encountered during the traversal of the tree.

6.3.4 User Flag Conditional

In the course of developing the code generator, it was deemed essential to be able arbitrarily control the generation of code, the execution of command words and the following of pointers during the traversal of a tree. To meet this requirement, a set of general purpose flags is included in the code generator. These flags can be arbitrarily set or cleared at any node in a tree.

A set of fifteen flags and a set associated command words which set and clear each of these flags are provided. Three of the flags are global to the whole tree whereas the other twelve flags are local to the subtrees whose roots contain the command words which set or clear the flags. In other words if, for example, a global flag is set then whenever that flag is tested (during the walk of the tree) it will be read as set until it is cleared (and vice versa). Whereas, with the setting of a local flag at some node, this flag will be read as set only in the subtree rooted at the above node. Once the walk of the tree exits that subtree, the local flag will take on the value it had before entering the subtree.

This structure is recognized by the (leading) '?U' character combination. The structure of this conditional is shown in figure 6.8.

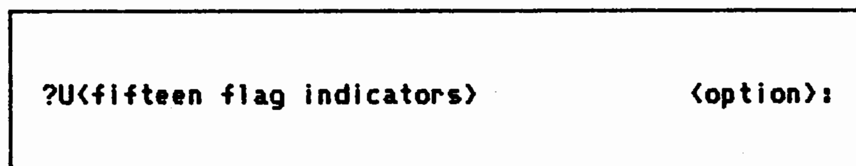


__Fig. 6.8__: **Structure of the User Flag Conditional.**

The first twelve flag indicators refer to local flags one to twelve, and the last three flag indicators refer to global flags one to three. Each of these flag indicators can be one of the following symbols:

-- 't' - if this flag is set then true,

-- 'f' - if this flag is clear then true,

-- 'x' - ignore the value of this flag.

When this structure is encountered, each flag is tested according to its respective indicator. Next a Boolean 'AND' operation is done to the results of all the above tests. If the result of the 'AND' operation is true, then the associated option is selected.

Here are some examples of the structure:

-- txxtxxfxxxxxXXX - if local flags one and four are set and local flag seven is clear, then true,

-- fffffttffffffFFF - if all the global flags are clear and all the local flags are clear, except local flags six and seven, then true,

-- tfftxxxxxxxxXXX - if local flags one and four are set and local flags two and three are clear, then true,

-- xxxxxxxxxxxxTXX - if global flag one is set, then true.

Although, at first sight, this structure may appear ungainly, there is no other obvious method to carry so many conditional options and yet use so little space. There is great flexibility built into the structure, as the conditional can test the value of single flags or any combination of (some or all) the flags, with mixed indicators ('t', 'f' and don't care) and mixed flag groupings (local and global).

6.3.5 History Store Conditional

When loading parameters in response to a procedure (or function) call, it is necessary to decide whether to produce P-code to load onto the stack the value of a data structure, or its address. This decision is based upon whether the associated parameter in the called procedure is a pass-by-reference or pass-by-value parameter. The only way to determine this is to look at the parameter list in the subtree representing the procedure. But DIANA does not give any direct link between the values to be loaded onto the stack and the associated parameters. The only reasonable technique whereby the necessary information can be obtained is to scan through the procedure's parameter definition list, passing each parameter definition subtree and loading up an array in the code generator with an indication as to whether each parameter is a reference or value parameter. Each time an array element is loaded, the array element index is incremented. Having done this and returned to the identifier list attached to the procedure call, a decision can be made as to whether each identifier's address or value must be loaded. This decision is based upon the information in the array.

The data structure used for the above task is called the history array and the associated index is called the history index. To use the history array a conditional structure is supplied and appears as illustrated in figure 6.9.

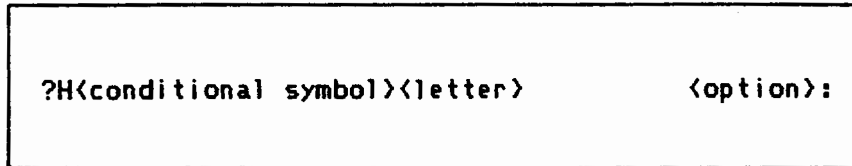

?H<conditional symbol><letter>   <option>:

Fig. 6.9: Structure of the History Conditional.

The conditional symbol can be one of the following:

-- 't' - true if <letter> matches the contents of the indexed element of the history array,

-- 'f' - true if <letter> does not match the contents of the indexed element of the history array.

Note that the <letter> can consist of any alphanumeric character and any unreserved non alphanumeric character. Also remember that anything entered via the tables is converted to upper case. So the comparison is not case sensitive.

Although a specific need spawned this data structure and associated conditional, there is nothing which ties this structure specifically to that need.

Some examples of this conditional structure are as follows:

-- Hta - if the currently indexed history array element contains the letter 'a', then true,

-- Hfd - if the currently indexed history array element does not contain the letter 'd', then true.

6.3.6 Comparison Store Conditional

In order to determine the type of a numeric literal in DIANA, it is necessary to examine the value of a reference field. The value of this field will indicate what the type is, namely integer or real. To do this, a facility is needed to compare values and make decisions on that basis. So a comparison store is provided, along with some associated command words and a conditional structure. The conditional structure has a number of variations which enable comparisons between the comparison store and a number of other stores. The are two different structures for the comparison store conditional, one with a numeric store reference and one without. The structure without the numeric store reference is given in figure 6.10.

```
?T<type><conditional symbol>          <option>:
```

Fig. 6.10: Non-numeric-reference Comparison Store Conditional Structure.

The <type> can be one of the following:

-- 'K' - compare the value in the comparison store with the Kind number of the current node,

-- 'N' - compare the value in the comparison store with the sequence number of the current node,

-- 'H' - compare the value in the comparison store with the value in the history array index,

-- 'P' - compare the value in the comparison store with the value in the procedure count store,

-- 'XC' - compare the value in the comparison store with the current lexical level,

-- 'XI' - compare the value in the comparison store with the lexical level of definition of the current identifier,

-- 'XD' - compare the value in the comparison store with the value resulting from the (positive) difference between the current lexical level and the lexical level of definition of the current identifier.

None of the above stores with which the comparison store is compared have a numeric reference, hence the name of the above variation. The variation, as depicted in figure 6.11, does have this numeric reference.

```
?T<type><number><conditional symbol>    <option>:
```

Fig. 6.11: Numeric-reference Comparison Store Conditional Structure.

The <type> and <number> in this case represent:

-- 'O<number>' - compare the value in the comparison store with the value in the offset store referred to by <number>, where <number> is in the range 1 to 3,

-- 'G<number>' - compare the value in the comparison store with the value in the

global user store referred to by <number>, where <number> is in the range 1 to 4,

-- 'L<number>' - compare the value in the comparison store with the value in the local user store referred to by <number>, where <number> is in the range 1 to 4.

For both the above variations of this conditional structure, the conditional symbol represents:

-- 't' - true if the comparison reveals identical values,

-- 'f' - true if the comparison reveals dissimilar values.

In combination with the command words, this conditional provides many useful decision making abilities. Although many of the possible comparisons are not needed at present, it was not difficult to include them for any future need.

Here are a few examples of this conditional structure:

-- ?Tnt - if the node sequence number of the current node is the same as the number stored in the comparison store, then true,

-- ?Thf - if the value stored in the history array index is not the same as the value stored in the comparison store, then true,

-- ?To1t - if the value stored in offset store 1 is the same as the value stored in the comparison store, then true,

-- ?T12f - if the value stored in local store 2 is not the same as the value stored in the comparison store, then true.

6.3.7 Pointer Conditional

In DIANA, some nodes result in different P-code being produced, dependent upon the presence or absence of a child node (that is, if a syntactic or semantic pointer is null or not). In addition, the only way to determine if a USED NAME ID node represents a native type or not is to examine a semantic pointer emanating from the node. Such a check will determine if that pointer is a null pointer or not, hence determining if the node represents a native type or not. So a conditional is included which makes a decision, dependent upon whether the next or previous syntactic pointer or the next or previous semantic pointer (relative to the location of the conditional structure in a code template table node definition) is null or not.

The structure of this conditional is shown in figure 6.12.

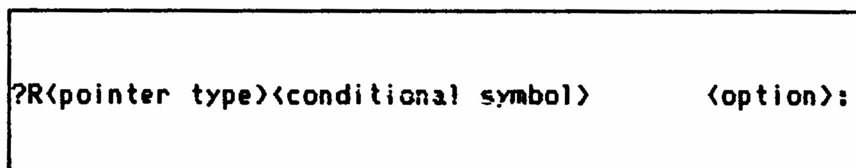


Fig. 6.12: Structure of the Pointer Conditional.

This structure is recognized by a '?R' character combination. This is followed by one of the pointer type indicators listed below:

-- 'a' - indicates that the test is done on the previous or next syntactic pointer,

-- 's' - indicates that the test is done on the previous or next semantic pointer.

The pointer type indicator is, in turn, followed by one of the conditional symbols listed below:

-- 'nt' - true if the next indicated pointer is not null,

-- 'nf' - true if the next indicated pointer is null,

-- 'lt' - true if the last encountered pointer is not null

-- 'lf' - true if the last encountered pointer is null.

Note that if a test is on a next pointer, it will test the next existing pointer, regardless of whether that pointer is within a 'comm' conditional structure or not. But if a test is on a previous pointer, then that test will be on the pointer most recently encountered, which may not be necessarily the nearest previous pointer. This is the most logical arrangement for the previous pointer test as a pointer encountered is of greater relevance than a pointer skipped. This cannot be done with the next pointer because, although it is clear whether a pointer already has been followed, it is not possible to know what the conditions will be when a pointer is encountered. This does not prove to be a restriction and any reasonable combination of conditional structures and pointers can be formed with this arrangement.

A null pointer is a pointer reference field which carries a value less than the threshold value defined in the initialization node for null pointers.

Here are some examples to illustrate this conditional:

- -- ?Rslt - if the last encountered semantic pointer is not null then true,
- -- ?Ralf - if the last encountered syntactic pointer is null then true,
- -- ?Rsnf - if the next existing semantic pointer is null then true,
- -- ?Rant - if the next existing syntactic pointer is not null then true.

6.3.8 Boolean Field Conditional

In the DIANA trees produced by the U.C.T. front end processor, the boolean fields of the nodes are used to indicate the existence of values in the node value fields with the same ordinal number. Therefore, it is necessary to provide a conditional structure which examines these boolean fields and makes decisions based upon the state carried by the fields.

The boolean conditional is recognized by a leading '?B' character combination. Its structure is illustrated in figure 6.13.

```
?B<number><conditional symbol>          <option>:
```

Fig. 6.13: Structure of the Boolean Field Conditional.

The <number> carries the ordinal number of the boolean field being tested. Before any use is made of a boolean field in the code template table, the tree specification table is checked to validate that the indicated boolean field may be used. If not, an error message is generated and the scan skips the current special command word.

The <conditional symbol> is as follows:

-- 't' - if the state carried by the boolean field is true, then the conditional evaluates to true,

-- 'f' - if the state carried by the boolean field is false, then the conditional evaluates to true.

In the nodes produced by the U.C.T. front end processor, the values in the boolean fields represent false with the number 0 and true with the number 1. The code generator converts these values internally to boolean values where the number 0 is converted to false and any other number is converted to true.

Although this conditional structure has a specific use when working with DIANA trees (indicating that a node value field carries a value), it is in no way tied to that use only.

6.3.9 Rep String Conditional

This structure is necessary because, in DIANA, there are certain nodes whose operation depends upon the attached strings in the rep field. Take, for example, the USED BLTN OP node. This node represents expression operators such as arithmetic and relational functions. The string in the rep field represents which of these operators is intended (for example '+' or '>').

The most convenient representation for this conditional was found to be a structure similar to a high level language case statement. Here a '?C' character combination is used to identify the structure as well as to indicate the options. The structure is given in figure 6.14.

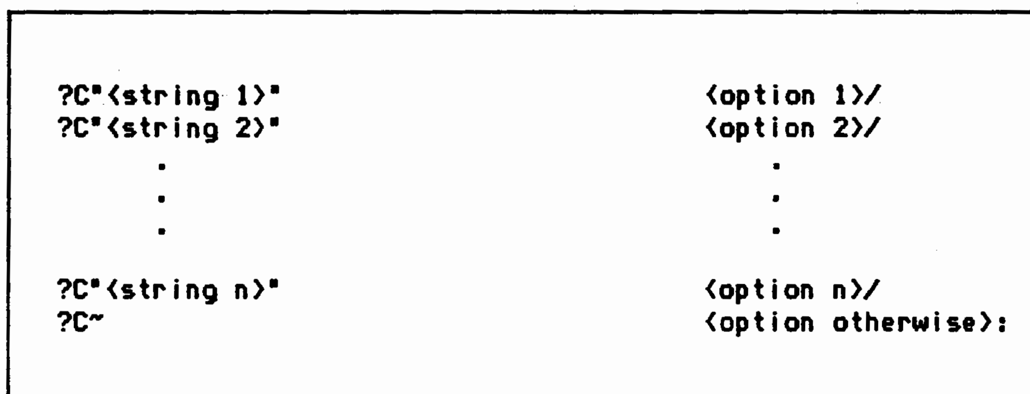


Fig. 6.14: Structure of the String Case Conditional.

The string in the rep field is compared with each string in quotes immediately following the '?C' symbol combination, one after the other, until the two strings match. Once a match is found, the associated option is printed in the code output file.

The '?C~' symbol combination at the end of the structure is an otherwise clause, selected if none of the test strings match the string in the rep field. Although not

mandatory, because sometimes no code exists for an 'otherwise' case, it is advisable always to include it, with the option carrying an error message to be printed out in the code output file.

Before the inclusion of the comparison store, the only way to determine the type of a numeric literal was to examine the literal's associated string. This examination needed to be able to detect the existence of a decimal point or the letter 'E' (indicating an exponent). To fulfil this requirement, it was necessary to include a wild card character search. This variation of the case structure searches for the existence of a single character in the string in the rep field (to search for a '.' or 'E' in the above example).

Although the literal type determination is now done by the use of the comparison store and associated conditional, this wild card variation of the case conditional remains. The structure is depicted in figure 6.15.

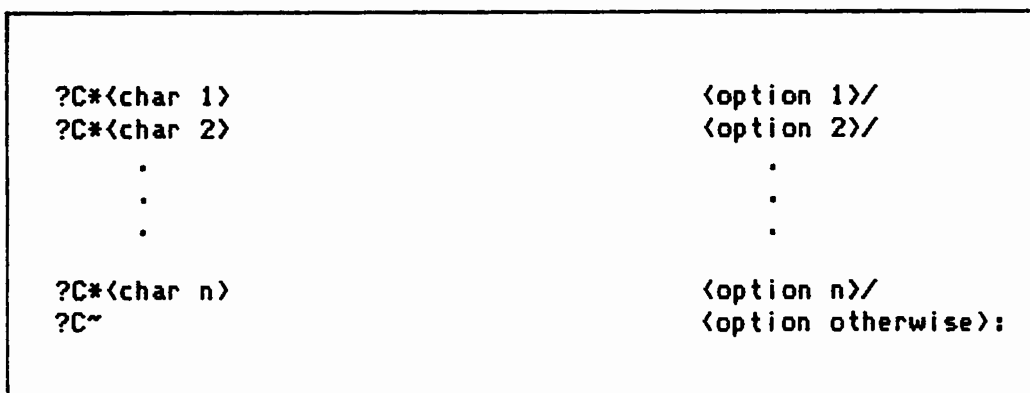


Fig. 6.15: Structure of the Wild Card Case Conditional.

It is permissible to mix wild card character patterns and full string patterns in the same case structure, but bear in mind that the case search is sequential in nature, starting at the beginning of the pattern list. To illustrate a possible problem caused

by this, assume that a string case exists in the structure. But this string case is preceded by a wild card character case, where this character exists in the string case mentioned above. The option associated with that string case will never be reached because the search will always encounter the wild card character case first (whose associated option is therefore reached).

6.4 Pointer Indicators

The pointer indicators inform the code generator that syntactic or semantic pointers can exist at the indicated node reference fields. They take the form of a word, describing whether a pointer is syntactic or semantic and an attached numeric value indicating what node reference field carries the pointer. The format is as follows:

-- SynReference(<number>) - a syntactic pointer may exist at node reference field <number> in the current node, where <number> is in the range 2 to 12,

-- SemReference(<number>) - a semantic pointer may exist at node reference field <number> in the current node, where <number> is in the range 2 to 12.

When such a word is encountered in the code template table node definition for the current node, the following actions occur:

(1) a check is made in the tree specification table definition for the current node to confirm that the indicated field may be a pointer of the appropriate type. If there is a discrepancy in the definitions between the two table entries for the current node, an error message is generated and the code generator aborts the

run. Note that a syntactic pointer is defined in the tree specification table as a reference with a syntactic or link attribute. A semantic pointer is defined as a reference with a semantic attribute,

(2) if the indicated pointer is permitted, then the value contained in the indicated reference field of the current node is read. If this value is not a null pointer, it is used as the key to access the next tree node of that sequence number. Identification for null pointers is entered in the initialization node, covered later. If the pointer is a null pointer, the pointer indicator is simply ignored,

(3) the kind number of the newly accessed tree node is read and used as the key for the node definitions for that kind number in both tables. These node definitions are then read in.

The process is the same for both syntactic pointers and semantic pointers. It is with this technique that the code generator guides itself around the tree being scanned. Like the command words, these pointer indicators can exist within a 'comm' special command word. Thus the walk can be controlled by the use of the conditional structures.

6.5 The Command Words

The command words instruct special purpose, built-in routines to be executed. Along with the conditionals and pointer indicators, they can be regarded as the 'vocabulary' of the code template table. The special purpose routines cause, among other functions, the following:

-- the initialization of the code generator before a run starts,

-- conditions to be set up for the conditional structures in the 'comm' and 'code' special command words,

-- control of syntactic and semantic pointer following, code generation and command word execution,

-- storing, updating and loading of the various identifier attributes in the symbol table,

-- manipulation of stored numeric values.

The command words are broken into two groups, namely those for the initialization node and those for the rest of the tree. The initialization command words are covered first in the following descriptions. Also, the command words are grouped together according the the data structure affected by their operation. If no data structure is affected, the words are then grouped according to function.

6.5.1 Node 0 Natives Special Command Word

This word falls into the same class as the 'comm' and 'code' command words. It is used to initialize the code generator with the native type names which can appear in the tree. Attached to the names are numeric values which indicate the space occupied by identifiers, or their dope vectors, which are defined using the native types. The format of this word is depicted in figure 6.16.

```
natives                    <native name>(<value>)
                           <native name>(<value>)
                                   .
                                   .
                                   .
                           <native name>(<value>):
```

Fig. 6.16: Structure of the 'Natives' Special Command Word.

The structure is recognized by the word 'natives'. The command word includes the following:

-- <native name>: the identifying string of a native type. This string must be the same as the string representing this type in the tree. The strings used are irrelevant to the code generator. They are relevant only to the implemented code. A string may be up to sixteen characters long,

-- <value>: gives the space occupied by an identifier, or its dope vector, defined using the associated native type. The units of the size are irrelevant to the code generator. They are relevant only to the implemented code and could represent, for example, bytes,

The ':' character is normally used to indicate the end of the natives command word. But if the end of this word coincides with the end of the initialization node then the terminating ';' character doubles as the end of the natives command word.

Although 'natives' is a special command word, no conditional structures are allowed within its bounds.

An example of the use of 'natives' is illustrated in figure 6.17.

```
natives                          real(2)
                                 integer(1)
                                 char(1)
                                 boolean(1):
```

Fig. 6.17: Example of Use of the 'Natives' Special Command Word.

The 'natives' special command word may be used only in the initialization node.

6.5.2 Node 0 Offset Initialization

It is useful to be able to initialize the three offset stores to given values before the start of the tree traversal by the code generator. The word which does this is as follows:

-- Offset<number>(<value>): initialize the offset store of ordinal number <number> with <value>, where <number> is in the range 1 to 3 and <value> is in the range -9999 to 9999.

The value in brackets attached to the above command word is a positive or negative numeral of up to four digits (excluding the sign). The value it represents is assigned to the offset store being reset.

Here is an example of the use of this command word:

-- Offset2(5): initialize the second offset store to 5.

The above command word may be used only in the initialization node.

6.5.3 Node 0 Procedure Count Initialization

As with the offset stores, it is useful to initialize the procedure count store to a given value before the start of a tree walk:

-- ProcCount(<value>): initialize the procedure count store with <value>, where <value> is in the range -9999 to 9999.

This command word may be used only in the initialization node.

6.5.4 Node 0 Start Node and Null Pointer Setup

In order to walk around a tree, it is essential to inform the code generator which node must be read in order to start the walk. A command word is included which supplies the sequence number of the starting node:

-- StartNode(<value>): give to the code generator the sequence number <value> of the first node to be read in a tree, where <value> is in the range 1 to 9999.

In addition, it is necessary to inform the code generator what the lower bound of valid

node sequence numbers is. This stops the walk from attempting to follow null pointers, which carry values below this given value:

-- LowerBound(<value>): give to the code generator the lower bound <value> of valid node sequence numbers, where <value> is in the range 1 to 9999.

Note that this lower bound setting affects only the pointers selected for traversal and the null pointer conditional structure. It does not affect any other use of the node sequence numbers and reference fields, either by command words or other conditional structures.

These command words may be used only in the initialization node.

6.5.5 Node 0 Global Level Initialization

The numeric value of the global lexical level is given to the code generator by this command word. When the lexical level is incremented for the first time in a tree walk, the level node produced is given this global lexical level value. Thereafter, every time the lexical level is incremented and a new level node is created, so the value of the previous level node is incremented and given to this new level node. The command word is as follows:

-- GlobalLevel(<value>): give to the code generator the numeric global lexical level of <value>, where <value> is in the range -9999 to 9999.

The lexical level value is used by the lexical level conditional structure, and can be produced in the code output file.

This command word may be used only in the initialization node.

The command words which follow from here are for use in the rest of the code template table and cannot be used in the initialization node.

6.5.6 Symbol Table Commands

There are two different operations for this structure. One increments and decrements the lexical level by creating and deleting a level node. The other adds an identifier to the symbol table by creating an identifier node. Although the two operations are quite different, they both affect the symbol table and are therefore grouped under this heading.

Although the requirement is to use a symbol table as little as possible, a few items cannot be accessed in any other way. The lexical level is one such value. It cannot be randomly extracted from the tree whenever it is required. The sub-chapter discussing data structures covers the symbol table structure. Whenever the lexical level is incremented, a new level node is created and linked onto the previous level node. The lexical level value contained in the previous level node is incremented and stored in this new level node. Whenever the lexical level is decremented, the latest level node is disconnected and discarded, thus leaving the then previous level node as the now latest level node.

-- IncLexicalLevel: add a new level node onto the level node linked list and increment the value contained in the lexical level store,

-- DecLexicalLevel: discard the latest level node form the level node linked list.

The lexical level cannot be decremented past the global level. If this is attempted, an error condition occurs, causing an error message to be printed. The code generator then aborts the run.

When adding a new identifier to the symbol table, a new identifier node is created. This is linked onto the end of the identifier list which 'dangles' from the current level node. If the level node was only recently created and no such identifier list exists yet, the newly created identifier node is then attached to the level node. The identifier list for the current lexical level is started in this manner.

There is a single command word which creates a new identifier node and links it on to the identifier list for the current lexical level in the manner described above:

-- DefineId: create an identifier node and link it onto the end of the identifier list in the symbol table for the current level. Then load the sequence number of the current tree node into the sequence number field of the newly created symbol table identifier node.

The sequence number is used later by another command word (IdInfoGrab) to locate the identifier node in the symbol table.

There is no restriction on the use of this command word. It can be used to add an identifier node to the current level at any time during the traversal of the tree.

6.5.7 Current Identifier Store Commands

This store is used to hold the attributes of an identifier. These attributes are obtained from both the symbol table and the native store. So command words are provided to load these attributes from that table and store:

-- IdInfoGrab: search through the symbol table identifier list at the current level and continue the search down through the levels towards the global level. Stop when a node in the symbol table is found with same sequence number as the current tree node. Then load all the ancillary attributes from this symbol table node to the current identifier store, along with the lexical level of its definition,

-- NativeInfoGrab: search through the native store for a node with the same string as that attached to tree node for which this command word exists. Then load all the native attributes into the current identifier store from the native store node.

The sequence number for every identifier defined in the tree is unique, whereas the identifier string is not necessarily so. Therefore it is more logical to use this sequence number in the symbol table so as to avoid ambiguities. To ensure that the sequence numbers match for an identifier between being defined and being used, the IdInfoGrab command word must exist in the same node definition as the DefineId command word. One node in the tree represents a single identifier at all times.

No sequence numbers for all the native types exist. In addition, there is no ambiguity

as native types cannot be overloaded (by other native types). It is therefore more logical in this case to use the strings attached to the relevant tree nodes.

If the required identifier does not exist in the symbol table, or the required native type does not exist in the native store, an error message is generated and the scan continues.

It is often necessary to be able to load the attributes of a given native type via a command word in the code template table:

-- Type <name>: search through the native store for the native type labeled <name> and load its attribute into the current identifier store.

The <name> string can be up to sixteen characters long and the letters composing it can consist of all alphanumeric characters and unreserved non alphanumeric characters. Remember that everything entered via the tables is converted to upper case, so there is no distinction between the same upper and lower case alpha characters.

If the required native type does not exist in the native store, an error message is generated and the scan continues.

Finally, to make the operation of the level conditional more visible, the current identifier store can be cleared. If a level decision is attempted on a clear store a warning is given and the conditional, with all it contains, is skipped. The command word to clear this store is:

-- ResetIdStore: clear the contents of the current identifier store.

When this is done, all the fields in the current identifier store are cleared.

There is no restriction on the use of this command word. At the start of the code generator's run, the current identifier store is cleared.

6.5.8 User Flag Commands

As mentioned in earlier sub-chapters, there exist fifteen user setable flags, three of which are global to the whole tree and twelve of which are local to the subtrees rooted to nodes where such flags were most recently set or cleared. Command words are provided to set and clear each of these flags. They are as follows:

-- Gflag<number>Set: set the global flag referred to by <number> where <number> is in the range 1 to 3,

-- Gflag<number>Clear: clear the global flag referred to by <number> where <number> is in the range 1 to 3,

-- Lflag<number>Set: set the local flag referred to be <number> where <number> is in the range 1 to 12,

-- Lflag<number>Clear: clear the local flag referred to by <number> where <number> is in the range 1 to 12.

Here are some examples of the use of these command words:

-- Gflag1Set: set the first global flag,

-- Lflag4Clear: clear the fourth local flag.

It is sometimes necessary to clear all the local flags or all the global flags. If this had to be done by writing each of the above clear commands, it would be very wasteful of table storage as well as being tedious to write. So two extra command words are included. One clears all the local flags and the other clears all the global flags. These command words are as follows:

-- ClearGflags: clear all the global flags 1 to 3,

-- ClearLflags: clear all the local flags 1 to 12.

There is no restriction on the use of any of the flag command words. At the start of the code generator's run, all the flags are cleared.

6.5.9 User Store Commands

There are occasions during the traversal of a tree when it is necessary to store, manipulate and produce numbers along with the output code. One such case is with the handling of jump addresses. Here, the need is to generate unique numbers so as to produce labels and jumps to these labels.

To cater for such needs, eight general purpose stores are provided. Four are global to the whole tree and four are local to subtrees rooted to where values contained in these stores were most recently changed.

All these stores can have their contained values incremented and decremented. What follows are the command words to do this:

-- Gstore<number>Increment: increment the value in the global store referred to by <number> where <number> is in the range 1 to 4,

-- Gstore<number>Decrement: decrement the value in the global store referred to by <number> where <number> is in the range 1 to 4,

-- Lstore<number>Increment: increment the value in the local store referred to by <number> where <number> is in the range 1 to 4,

-- Lstore<number>Decrement: decrement the value in the local store referred to by <number> where <number> is in the range 1 to 4.

It was found necessary to be able to transfer values between the local and global stores. So provision is made to enable such transfers between local and global stores with the same ordinal number. The command words which enable this are:

-- Gstore<number>ToLstore: transfer the value contained in the global store referred to by <number> to the local store of the <u>same</u> ordinal number where <number> is in the range 1 to 4,

-- Lstore<number>ToGstore: transfer the value contained in the local store referred to by <number> to the global store of the <u>same</u> ordinal number where <number> is in the range 1 to 4.

Although there is no present need to be able to re-initialize individual stores to given values, command words are included to allow this. They are as follows:

-- Gstore<number>Reset(<value>): reset to <value> the global store referred to by <number> where <number> is in the range 1 to 4 and <value> is in the range -9999 to 9999,

-- Lstore<number>Reset(<value>): reset to <value> the local store referred to by <number> where <number> is in the range 1 to 4 and <value> is in the range -9999 to 9999,

It is useful and economical with table space and typing to be able to use one operation to re-initialize all the local stores or all the global stores to a given value. Two command words are provided to enable this. These words are as follows:

-- ResetGstores(<value>): re-initialize all the global stores to <value> where <value> is in the range -9999 to 9999,

-- ResetLstores(<value>): re-initialize all the local stores to <value> where <value> is in the range -9999 to 9999.

Some examples of the above command words follow:

- -- Gstore1Increment: increment the value contained in the first global store,
- -- Lstore3Decrement: decrement the value contained in the third local store,
- -- Gstore2ToLstore: copy the value contained in the second global store to the second local store,
- -- Lstore4Reset(-15): initialize the fourth local store to -15,
- -- ResetGstores(0): initialize all the global stores to 0.

Note that there is no restriction on the use of any of the store related command words. At the start of the code generator's run, the stores are all initialized to 0.

6.5.10 History and Index Commands

It is sometimes necessary to record the sequence of a list of nodes for later decisions. An array and associated index are provided to do such recording.

The command words for this structure allow for the indexing and loading of elements in the array. The value contained in the index is the ordinal number of the indexed element in the array. In the following text, this indexed element is referred to as the current element.

To load the current element with a character, use is made of the following command word:

-- Hist <letter>: load the current element with <letter>, where letter is an alphanumeric character or an unreserved non-alphanumeric character.

Remember that all text entered via the tables is converted to upper case only. So there is no distinction between lower case and upper case alpha characters for the above <letter>.

Commands are included which increment and decrement the index in order to permit sequential access to the history array elements:

-- IncHistIndex: increment the history array index,

-- DecHistIndex: decrement the history array index.

Whenever one of the above index increment or decrement commands is used, a check is first done to ensure that the index will not go out of bounds and try to index below or above the permitted range. If such a situation occurs then the command is ignored, an error message is generated and the scan continues.

A command word which resets the index to a given value is also provided:

-- ResetHistIndex(<number>): reset the history index to <number>, where <number> is in the range 1 to 128.

A range check is done on <number> to ensure that it falls within the above mentioned range. If it is outside of that range, then an error message is generated and the scan continues.

It is necessary to be able to clear the whole history array with one command by resetting every array element with a given letter. The command word to do this is as follows:

-- ClearHist <letter>: reset every element in the history array to <letter>, where <letter> is an alphanumeric character or an unreserved non-alphanumeric character.

There is a restriction on the use of the history array and associated index. The index must fall in the range 1 to 128. Also, the characters which are to be stored in the array elements must not include any reserved characters. At the start of the code generator's run, every element in the history array is set to the blank character and the index is set to 1.

6.5.11 Comparison Store Commands

The comparison store is provided to enable decisions to be made on the basis of what a node's sequence or kind number is, or what values exist in some of the other stores. A set of command words is provided which load and manipulate the comparison store.

The following command words load the comparison store with the values contained in the node kind number and node sequence number fields of a tree node:

-- KindnoToCompStore: copy the current node kind number into the comparison store,

-- NodenoToCompStore: copy the current node sequence number into the comparison store,

-- RefNodenoToCompstore(<number>): copy the node sequence number of the node pointed to by reference field <number> into the comparison store, where <number> is in the range 2 to 12.

For the third point above, the reference field is first validated by checking in the DIANA specification table. If the field does not exist for that node, then an error message is generated and the scan continues. Remember that although the above word is long, only the first eight characters are significant. It is felt that for the purposes of this work, the longer words help in the understanding of what their functions are.

Aside from just loading the comparison store from various sources, its value can be incremented and decremented. The command words to do this are:

-- IncCompStore: increment the comparison store,

-- DecCompStore: decrement the comparison store.

There is no specific requirement for the inclusion of these words. But it was simple to do and they may be needed at some future date.

As with the other structures, provision is made to reset this store to a given value:

-- CompReset(<value>): reset the comparison store to <value>, where <value> is in the range -9999 to 9999.

There is no restriction on the use of the store's related commands. At the start of the code generator's run, this store is initialized to 0.

6.5.12 Offset Store Commands

The three offset stores are provided to hold the current maximum value representing the offset into a procedure data segment. Three are provided so that, in addition to holding the absolute offset from the beginning of the procedure data segment, the parameter and local variable offsets can be held separately, if so required. These stores are used to give each data structure identifier its offset value in the symbol table.

The following command word is used to assign the value in an offset store to the value field of an identifier entry in the symbol table:

-- Offset<number>Assign: Assign the value contained in the offset store referred to by <number> to the value field in the symbol table of the identifier entry currently being defined, where <number> is in the range 1 to 3.

When so wished, the value contained in an offset store can be updated by the size of the current native type. This size information is obtained from the current identifier store. The command word for this is:

-- Offset<number>Update: add the current native size to the value already held in the offset store referred to by <number>, where <number> is in the range 1 to 3.

If it is necessary to reset the values of these stores, it can be easily done by the use of the following command word:

-- Offset<number>Reset(<value>): Reset to <value> the offset store referred to by <number>, where <number> is in the range 1 to 3 and <value> is in the range -9999 to 9999.

Increment commands are provided in order to allow extra control over the offset stores:

-- Offset<number>Increment: increment the offset store referred to by <number>, where <number> is in the range 1 to 3.

Since an offset value never decreases, there is no need to include a decrement command.

Some examples of the above command words follow:

-- Offset1Assign: assign the value contained in the first offset store to the current symbol table identifier node's value field,

-- Offset2Update: add the size of the current native type to the value contained in the second offset store,

-- Offset3Reset(2): initialize the third offset store to 2,

-- Offset1Increment: increment the value contained in the first offset store.

There is no restriction on the use of the offset stores. At the start of the run, these stores are all initialized to 0, unless there are entries in the initialization node to initialize them to a given value. Note that they can be individually initialized by these entries.

6.5.13 Procedure Count Commands

A simple global integer store is provided in order to hold the count of any procedures (and functions) encountered to date during the scan of a tree. This store is the source of the ordinal numbers attached to the identifiers of procedures in the symbol table.

When adding a procedure to the symbol table, the DefineId command word is used in conjunction with:

-- AssignProcCount: assign the value contained in the procedure count store to the value field in the symbol table of the identifier entry currently being defined.

Note that in the case of a procedure, the value field in the symbol table identifier node is used to hold the procedure's ordinal number. Since the offset value of a data structure and the ordinal number of a procedure are mutually exclusive, this should cause no problem.

When so wished, the procedure count can be updated. Here, the update consists of a simple increment:

-- UpdateProcCount: increment the value contained in the procedure count store.

There is no need to include a decrement command since the procedure count never decreases.

Although there is currently no need to reset this store to a given value, such a facility is provided in case the requirement to do so arises:

-- ResetProcCount(<value>): reset to <value> the procedure count store, where <value> is in the range -9999 to 9999.

There is no restriction on the use of the above command words. At the start of the code generator's run, the procedure count is initialized to 0, unless there is an entry in the initialization node to initialize it to a given value.

6.5.14 Stash and Restore Commands

When it is necessary to modify the values contained in certain data structures, and later to restore the values they contained before modification, use can be made of the stash and restore commands. These commands use the stash data structure.

The commands affect a number of different data structure groups, as is explained in the following:

-- GflagStash and GflagRestore: stash and restore the values contained in all the global user flags,

-- LflagStash and LflagRestore: stash and restore the values contained in all the local user flags,

-- GstoreStash and GstoreRestore: stash and restore the values contained in all the global user stores,

-- LstoreStash and LstoreRestore: stash and restore the values contained in all the local user stores,

-- HistStash and HistRestore: stash and restore the contents of the history array and the value contained in the associated history index,

-- CompStash and CompRestore: stash and restore the value contained in the comparison store,

-- OffsetStash and OffsetRestore: stash and restore the values contained in all the offset stores,

-- ProcStash and ProcRestore: stash and restore the value contained in the procedure count store,

-- ControlStash and ControlRestore: stash and restore the suspend and resume control flags. Note that these words are exempt from command word suspension.

Remember that the stash data structure is local only to each visited tree node. So if a stash operation is executed, the corresponding restore operation then can be done for the aforementioned stashed values during the same visit to that node. Also, if the subtree to that node is traversed, and the walk then returns to the node, the stashed values can be restored. But once the walk leaves that subtree and the associated root node, the stashed values are lost. This is the result of the recursive nature of the walk.

There is no restriction on the use of the stash and restore commands.

6.5.15 Suspend and Resume Commands

Commands which can suspend and later resume the major elements of control have been included so as to provide additional control over all activities during the traversal of a tree. These elements are:

-- following a syntactic pointer,

-- following a semantic pointer,

-- executing a command word,

-- generating code.

For each of the above elements, a recursively local control flag exists. If such a flag is clear then whenever the corresponding element is encountered, it is dealt with

normally. But if the flag is set, then that element is skipped. So in order to control the above elements, the provided command words simply set and clear these flags. The command words are as follows:

-- SynWalkSuspend: set the syntactic walk control flag,

-- SynWalkResume: clear the syntactic walk control flag,

-- SemWalkSuspend: set the semantic walk control flag,

-- SemwalkResume: clear the semantic walk control flag,

-- CommWdSuspend: set the command word control flag,

-- CommWdResume: clear the command word control flag,

-- PcodeSuspend: set the code control flag,

-- PcodeResume: clear the code control flag.

Note that when suspending the command words, the resume command word will not be suspended. If the converse was the case then, once set, the command word control flag could never be cleared! Note also that the command word suspend command does not affect the generation of code. It was found desirable to control this with a separate command, hence the provision of PcodeSuspend and PcodeResume.

Note also that the 'comm' special command word is not affected by the command word suspend command. Instead, all the command words and pointer indicators within the bounds of 'comm' are controlled normally as described above. This is to avoid the undesirable side effect of a command word suspend resulting in any pointer indicators within the bounds of 'comm' being skipped.

When setting or clearing a flag, that condition will hold throughout the subtree rooted to the node in which the flag was modified. When that subtree is exited, the flag will then resume its previous value.

There is no restriction on the use of these command words. At the start of the code generator's run, all the flags are cleared, so that none of the elements are skipped.

6.6 The Code Templates

In order to generate P-code instructions and operands from the code template table, patterns and substitute operators are needed. The patterns are produced in the code output table exactly as they occur in the code template table. Substitute operators result in the values of data structures, in both the code generator and the tree, being produced in place of the operators. These patterns and substitute operators together are known as templates.

6.6.1 Placement of the Templates

All code templates, along with any conditional structures affecting the template selection, must exist within the 'code' special command word. Whenever conditional

structures are used with the templates, they must be placed within the innermost conditional structure used to date in the current 'code' special command word (except for the type conditional structure, already covered). Within this innermost conditional structure, the templates must be placed in the positions marked 'option' as displayed in the conditional structures sub-chapter. When the conditional structures are evaluated, and the result of every conditional is true, then the templates within the selected option are produced in the code output file. Note that more than one template may exist within the option. If this is the case, then the successful selection of an option with more than one template results in these templates being produced in the code output file in exactly the same order as they occur within the option. Of course, the conditional structures are not mandatory. So if they are not needed, then the code templates are located immediately after the special command word 'code'.

6.6.2 The Patterns

The patterns are quite simple in nature. They can consist of all the alphanumeric characters and all the unreserved non-alphanumeric characters. If selected for output, a pattern is produced in the code output file in exactly the same form as it appears in the code template table. There are a few non-alphanumeric symbols which are reserved for specific tasks and may not be used as actual pattern characters. These are:

-- ';' - indicates the end of a node definition (in both tables),

-- ':' - indicates the end of a code, comm or natives special command word (this is

substituted by the end of node symbol ';' if the end of the special command word coincides with the end of the node),

-- '?' - indicates the existence of a conditional structure (excluding the type selection conditional structure),

-- '/' - indicates the existence of a type selection conditional structure and delineates the options in various conditional structures,

-- '^' - indicates the existence of a substitute operator,

-- '*' - causes the suppression of the automatic line feed at the end of a selected option and is also used to indicate no template (or command word or pointer in 'comm') exists for a specific option,

-- '!' - causes the generation of a line feed character.

The '*' and '!' symbols are of special relevance to the patterns (and the templates in general) as they allow control of the placement of the generated code in the code output file.

An example of the need for such placement control of the generated code is the production of a P-code exit jump instruction in the code output file from the scan of a DIANA subtree. First the jump instruction is generated. The actual label name (or, in the case of an unlabeled jump, the label number) is appended to the jump instruction only after following a pointer to a label node. So the production of such an instruction is possible only when code segments generated from templates in separate 'code'

special command words can be concatenated. To enable such concatenation, provision is made to suppress the automatic line feed which is generated at the end of a selected option. To do this, a '*' character is simply appended to the appropriate template. The option is then exited. The next option to be selected will be source to code which is appended to the previous piece of generated code, regardless of which node this template originates from.

By contrast, it is also often necessary to be able to place code segments, which originate from the same option, on separate lines in the code output file. Originally, the system was arranged so that wherever a line feed occurred in a template, then a line feed was produced in the code output file. But such an arrangement was found to be too inflexible and difficult to control. So, in the present system, all line feeds in the templates are ignored. If a line feed is required in the code output file then a '!' character is placed in the corresponding location in the relevant template. With such an arrangement, much greater flexibility is provided over the template layouts. Also, greater control over the code generation is available.

6.6.3 The Substitute Operators

The patterns by themselves would not be nearly complete enough to permit the production of output code. Many 'variables' often need to be reproduced in the code output file. Examples of such variables are the values contained in the value fields of tree nodes, or the values contained in the code generator's general purpose user stores. So provision must be made to allow for the reproduction of the contents of some tree node fields and code generator data structures in the code output file. So as to permit this, some commands have been introduced which enable the printing of these contents. To identify such a command, a '^' character is used. This is followed

by a single letter to identify what data structure is to be accessed. If there is more than one of a particular data structure type, such as in the general purpose user stores, the letter is then followed by a single digit number to identify which is being referred to.

What follows is a list of all the substitute operators and an identification of the data structures they refer to:

-- '^XI' - print the lexical level of definition of the current identifier,

-- '^XC' - print the current lexical level (<u>not</u> the lexical level of the current identifier),

-- '^XD' - print the number representing the (positive) difference between the present lexical level and the lexical level of definition of the current identifier,

-- '^S' - print the native size value in the current identifier store,

-- '^N' - print the offset value of the current identifier (<u>not</u> one of the three global offset stores). If the current identifier is a procedure or function (and the command word 'AssignProcCount' was used in the identifier's definition), then the number printed is the ordinal number of that procedure or function (<u>not</u> the value in the procedure count store),

-- '^L<number>' - print the value contained in the local general purpose user store with ordinal number <number>,

-- '^G<number>' - print the value contained in the global general purpose user store with ordinal number <number>,

-- '^HI' - print the value contained in the history index,

-- '^HE' - print the character contained in the currently indexed history array element,

-- '^C' - print the value contained in the comparison store,

-- '^O<number>' - print the value contained in the (global) offset store with ordinal number <number>,

-- '^P' - print the value contained in the procedure count store,

-- '^VI<number>' - print as an integer the value contained in the value field with ordinal number <number> in the current node,

-- '^VR<number>' - print as a real number the value contained in the value field with ordinal number <number> in the current node,

-- '^R' - print the string contained in the rep field attached to the current node.

Wherever a substitute operator is used with an attached number, a range check is done to ensure that this number does not refer to an invalid data structure. In addition, for the last three points above, a check in the tree specification table is done to ensure

that the node fields referred to actually exist for the current node. If the number is out of range or the field referred to does not exist, then an error message is generated and the scan continues.

6.7 Traversal of a Tree

Before the given tree is scanned by the code generator, node definition 0 (the initialization node) is read from the code template table and all the native type information is loaded. In addition the code generator is set up, according to the other information in node 0, to start scanning the tree.

When a node is read from a tree, the Kind number of the node is extracted and used as the access Key for both the tree specification table definition and code template table definition for that node in the respective (translated and converted) tables.

At this point, a search through the code template table definition is initiated (from the beginning of the definition). If any command words or code segments are encountered, they are dealt with in the order in which they are found.

If the search encounters a syntactic or semantic pointer, the associated reference number is extracted. Remember that this search is started from the beginning of the code template table node definition. So the order in which the pointers (for that node) are entered in that table will determine the order in which the pointers of the current node are traversed. A search for a matching pointer attribute is then commenced in the attribute field store of the tree specification table definition.

If a matching syntactic or semantic attribute is found, its associated reference number is extracted. The two reference numbers are compared. If these numbers are not the same, then the search through the tree specification table definition continues. If the end of the definition is reached and no matching entry is found, then there is a discrepancy between the node definitions in the two tables (for the same node) and an error message is generated.

If completely matching entries are found, then the reference number is used as an index to the node reference fields. The number contained in the indexed field is the pointer. It is extracted and used as the access key for the indicated child node. Analysis is then commenced at the child node, if it exists (that is, if the number contained in the pointer field is not a null pointer - the existence of a pointer does not automatically imply that it points to anything).

If the same pointer is indicated more than once in the code template table definition, then the code generator will follow that pointer each time such an indicator is encountered (subject to any conditionals attached to the individual pointer definitions).

All the command words, code segments and pointers are dealt with in the order in which they are encountered. The above process is repeated with the search and analysis continuing until no further command words, code segments or pointer indicators are encountered in the code template table definition. When the end of the node definition is reached, the tree analysis is returned to the father node which has a pointer to this current node.

During the traversal of the tree, a trace is produced on the standard output of the computer. This trace gives a detailed account of every node encountered and every

command word, pointer indicator, conditional, code and comm statement encountered. The trace is included to help debug any new tree entries and also to illustrate the operation of the code generator.

The operation of the code generator is best illustrated with an example.

What follows is an example designed to give an understanding of the operation of the code generator. After providing the example tables and tree, a walk through the tree, based on the information in the tables, and the actions at each node are described in detail. Afterwards, a summary of all the actions is given as a list of the command words and pointers as they were encountered in the walk.

The following example is based on DIANA. It is not DIANA. A DIANA tree and the associated table entries would be too complex and large to serve as an illustrative example. This example shows the walk of the code generator as it traverses a tree representing the Ada statement 'A := 1.0;'.

To instruct the system what to do with the given tree, a tree specification table and a code template table are needed. Only the table entries relevant to the above statement are included.

The first table presented is the tree specification table relevant to the example:

```
VAR-ID      1     sm-type          reference(2)
                  as-exp           reference(3);

REAL        2     sm-type          reference(2);

USED-ID     3     sm-definition    reference(2)
                  lx-symrep        rep;
NUM-LIT     4     sm-type          reference(2)
                  sm-value         value(1)
                                   boolean(1)
                                   rep;

ASSIGN      5     as-name          reference(2)
                  as-exp           reference(3);
```

The next table listed is the code template table relevant to this example:

```
PCODEINFO  0        natives integer(1)
                            real(2);

VAR-ID     1                null

;-----------------------------------------

: VAR-ID definition entries
: (not needed for this example)

;-----------------------------------------

comm ?UtftxxxxxxxxxXXX      IdInfoGrab
                            SemReference(2):

code ?UtftxxxxxxxxxXXX ?Ll /*/
                            LLA ^N/
                       ?Li /*/
                            LDA ^XD ^N/
                       ?Lg /*/
                            LAO ^N:


code ?UtfxtfxxxxxxxXXX ?Ll /LDL ^N/
                            LDM 2/
                       ?Li /LOD ^XD ^N/
                            LDM 2/
                       ?Lg /LDO ^N/
                            LDM 2:

code ?UtfxftxxxxxxxXXX ?Ll /STL ^N/
                            STM 2/
                       ?Li /STR ^XD ^N/
                            STM 2/
                       ?Lg /SRO ^N/
                            STM 2;
```

```
REAL        2                   type real;


USED-ID     3                   SemReference(2);


NUM-LIT     4                   ResetLstore(4)
                                RefNodeNoToComp(2)

comm                      ?T11t type integer:
comm                      ?T11f type real:

code                      ?T11t LDCI ^R:
code                      ?T11f LDC 2   ^VR1;

 ASSIGN     5                    ClearLflags

                                 Lflag1Set
                                 Lflag3Set
                                 SynReference(2)

                                 IdStash
                                 Lflag4Set
                                 SynReference(3)
                                 IdRestore

                                 Lflag4Clear
                                 Lflag5Set
                                 SynReference(2);
```

Use is made of the user flags in the above code template table. These are set up to control base type access, address loading, value loading and value storing. Whenever there are mutually exclusive operations, the flags are used in a bit pattern where each combination represents a different condition. This is done to conserve flag usage. Any of the flag combinations which are not arranged as mutually exclusive can be combined to have more than one condition set up at a time. This is optional and it is up to the user to decide what conditions to represent, how to represent them and what to do when these conditions occur. What follows is a list of the flag setups for the example:

-- tfxxxxxxxxxxXXX - statement phase,

-- ftxxxxxxxxxxXXX - definition phase,

-- xxtxxxxxxxxxXXX - address load,

-- xxxtfxxxxxxxXXX - value load,

-- xxxftxxxxxxxXXX - value store.

Given the combinations used for the example, it is possible and necessary to have, for example, an address load condition and a value load condition set up at the same time. But it is not possible to have a value load condition and a value store condition set up simultaneously. Such a combination is meaningless. As mentioned before, it is up to the user to decide how to represent any required conditions with the flags. An address load condition set up simultaneously with a value load condition, in this example system, appears as follows:

-- xxttfxxxxxxxXXX - address load and value load.

In this example, two types have been defined in node 0 (the initialization node). These are the types real and integer, where real is defined as occupying a space of two words and integer a space of one word. Remember that labels such as 'real' and 'integer' mean nothing to the code generator. What they mean and how they are used depends only upon the table entries.

Next is the tree structure for the Ada statement 'A := 1.0;'.

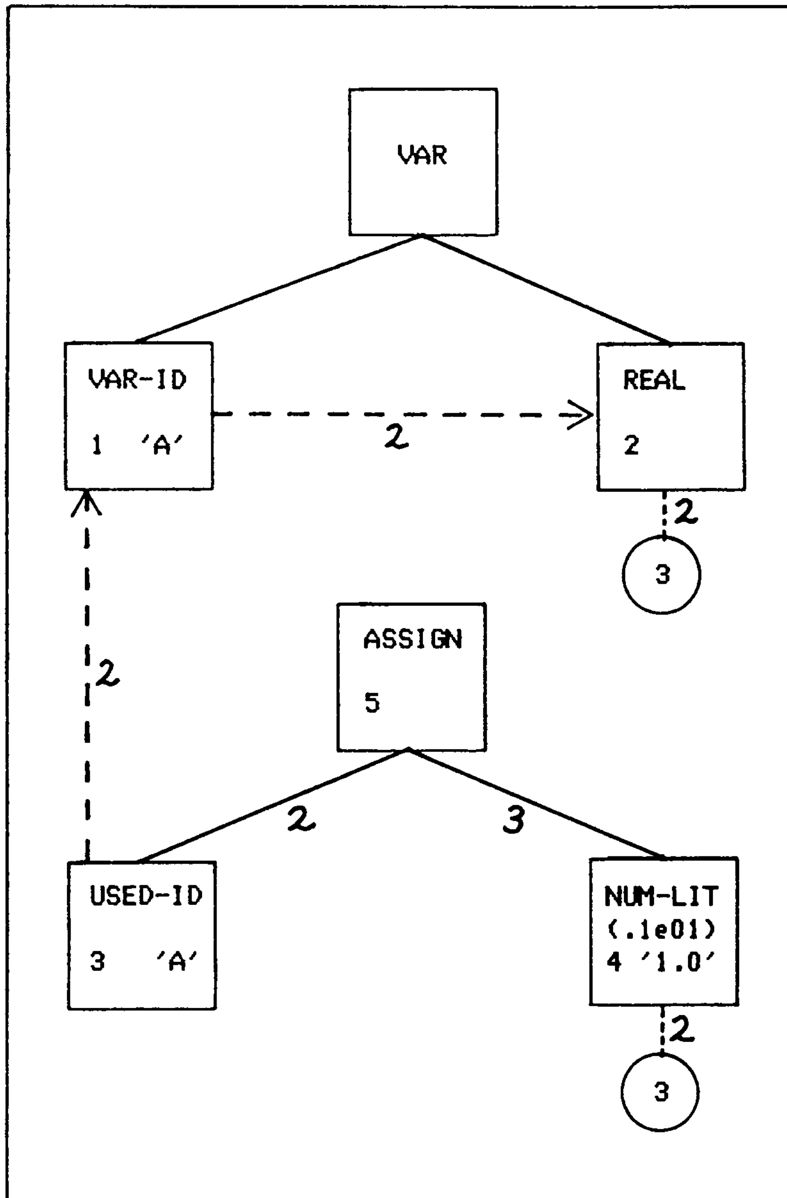


Fig. 6.18: A Simple Tree Example Representing 'A := 1.0;'.

Here are some explanatory points about this example:

(1) In the above example, a 'VAR' node exists which is not defined in the table. It is included in the tree diagram for completeness' sake and does not affect the example,

(2) Any strings and values attached to nodes in the above diagram are included in the node boxes. The strings are identified by being in quotes. The values are included in brackets,

(3) The solid lines indicate syntactic pointers and the dashed lines indicate semantic pointers. The numbers next to these lines indicate the reference field numbers of the pointers,

(4) The numbers in the bottom left corners of the node boxes are the node kind numbers. The node sequence numbers are not included as they would have cluttered the tree diagram,

(5) The code template table entry for the VAR-ID node is incomplete in that the entries which handle an identifier's definition are not included. These entries are not relevant to the example,

(6) For the example, assume that the identifier 'A' has been defined at the global level with an offset value of 8,

(7) To indicate the type of nodes representing numerals or numeric data structures, denoted type pointers are used. From such a node, these pointers resemble attribute pointers. In this example, a denoted type value of 3 represents a real. A denoted type value of 4 represents an integer. Such denoted type pointers are attached to the NUM-LIT and REAL nodes in the tree above,

(8) The traversal of the tree starts at the assign node.

Remember that whenever a node is entered from a parent node, the entries in both tables for the newly entered node are read to obtain the instructions for the node. Also remember that whenever reference is made to any of the tree node fields in the code template table, either by command words or by template substitute operators, the existence of those fields is first verified in the tree specification table.

When node Kind 5 (the ASSIGN node) is first encountered, all the local flags are cleared and local flags one and three are set, indicating (in this example) a statement phase address load condition. Then the syntactic pointer 2 is followed. This pointer leads to node Kind 3 (the USED ID node) which represents the left hand side of the assignment.

Node Kind 3 represents the identifier 'A' on the left hand side of the example assignment. Nothing occurs at this node other than to follow the semantic pointer 2 from here to node Kind 1 (the VAR ID node).

Node Kind 1 carries all the address load, value load and value store code templates for a variable identifier. But before any of these can be selected, the attributes for the identifier have to be loaded. So if the user flags are set up to indicate an address load (as they are during this visit to node Kind 1), the command word and semantic pointer contained in the comm statement are selected. IdInfoGrab is used to obtain the identifier's level of definition and offset value (for the P-code output program). Then the semantic pointer 2 is followed to obtain the identifier's base type. This leads to node Kind 2 (the REAL node).

At node Kind 2 the type real is loaded by the type command word. The attributes for this type are loaded. From this point, there is nothing left to do in this node. So the walk returns to node Kind 1 and continues from the semantic pointer 2 which caused a visit to node Kind 2.

At this point, we have all the information needed about the identifier. Next a code special command word is encountered. Remember that the flags were set up earlier to indicate an address load condition. This flag setup, along with the type information, is now used to select the P-code command(s) to be generated. The first code segment, in this case, has the correct flags set up for an address load.

The scan continues to the next conditional (or template, whichever comes first) within the above mentioned code segment. This is a level conditional which tests the level of definition of the current identifier. The current identifier was defined at a global level, so the conditional '?Lg' is true. Therefore the other two level selectors (and their contained conditionals and templates) are skipped. The scan continues to the next conditional (or template) after the '?Lg' conditional.

The next conditional encountered is the type select conditional. The type of the current identifier is real. Remember that the order of the options for the type conditional must match the order in which the native types are listed. So the first option in this example is for an integer and the second option is for a real. In this case, the type of the current identifier is real. So the first option is skipped and the second is selected. Note that in the options for an address load of an integer, a '*' character exists. This means no template is to be printed. The '*' character is placed here because, for an integer store, a single P-code command exists which does not need

an address on the stack. So if the type of the current identifier was integer, no address load P-code instruction would be produced.

A P-code template is now encountered. This template consists of the pattern 'LAO ' and the substitute operator '^N'. The pattern is reproduced in the code output file, and the substitute operator causes the offset value of the current identifier to be printed. The position of the offset number relative to the P-code instruction in the code output file is the same as that of the substitute operator relative to the pattern. When the ':' is reached, the option is complete and this code segment is exited. A line feed is now generated in the code output file. The other two code segments are not scanned beyond the flag conditionals because they do not match with the current flag setup. At this point the node has no more matching instructions and is therefore exited.

The walk returns to node kind 3, but this too has no further matching instructions and is also exited.

The walk returns to node kind 5 and continues to scan the instructions from the syntactic pointer 2 which caused the whole subtree scan just described. The next command word stashes all the current identifier attributes so that, when returning to this node after descending syntactic pointer 3, these attributes can be restored. Next the local flag is set which indicates a value load condition. Remember that earlier on in this node the flag indicating an address load was set. These conditions are not mutually exclusive, so now both address load and value load conditions are set up. This is necessary for traversing the subtree representing the right hand side of an assignment. Any real variables on the right hand side would have their address load

P-code instructions produced, followed immediately by the actual load instructions. From here syntactic pointer 3 is followed which leads to node kind 4 (the NUM LIT node).

The first instruction encountered in the table for this node is Lstore1Reset(4). This is the value which represents an integer in a denoted type pointer. After this the RefNodeNoToComp(2) instruction loads the value of pointer 2. This is a denoted type pointer, the value of which indicates the type of the literal. Next a comm special command word is encountered. The first comm command is not scanned beyond the comparison conditional, the result of which is false, and is skipped. The next instruction is also a comm command word. In this one, the comparison conditional is true, that is, the value in the comparison store is not equal to the value in the local store 1. This means that the denoted type is not an integer. So, given that the only other possible case is real, the numeric literal represented by this node must be real. The command word executed as a result of this conditional evaluation is the type command, which causes a real native attribute load.

After the two comm commands are two code commands, with exactly the same conditionals and therefore the same results. This causes the P-code template 'LDC 2 ^VR1' to be selected. As with the earlier template, the pattern 'LDC 2 ' is reproduced as it appears here. The substitute operator '^VR1' causes the value contained in the value field 1 of the tree node to be reproduced in the code output file as a real number. No further instructions exist for this node, so a line feed is printed in the code output file and the walk returns to node kind 5.

The scan of the instructions continues in node kind 5 from the syntactic pointer 3. The attributes of the left hand side identifier are restored by the IdRestore command.

Then the flags are set up, this time to indicate only a value store condition. Syntactic pointer 2 is, once again, traversed and leads to node kind 3.

As before, nothing happens at this node other than to continue the walk to node kind 1.

This time, for node kind 1, the comm word controlled instructions selected when the address load condition was set are skipped. Only the code command word with the user flag conditional set to select a value store condition is not skipped. The selection process for the current identifier is the same as before and causes the pattern 'STM 2', followed by a line feed, to be produced. Nothing else occurs in this node and the walk returns to node kind 3.

As before, the walk does nothing in node kind 3 and returns to node kind 5.

No instructions are left for this node and so this tree is exited and the run is finished.

The P-code produced by this run, given the initial conditions stated earlier, is as follows:

```
LAO 8
LDC 2    .1e01
STM 2
```

This is the correct P-code for 'A := 1.0;'. To close this chapter, a list is provided which gives a trace of the command words executed, the pointers followed and the code generated, all in the order of occurrence:

NODE	INSTRUCTION	P-CODE
5	ClearLflags	
	Lflag1Set	
	Lflag3Set	
	SynReference(2)	
3	SemReference(2)	
1	Comm IdInfoGrab	
	SemReference(2)	
2	Type real	
1	Code	LAO 8
	Code <not selected>	
	Code <not selected>	
3	----	
5	IdStash	
	Lflag4Set	
	SynReference(3)	
4	LStore1Reset(4)	
	RefNodeNoToComp(2)	
	Comm <not selected>	
	Comm Type real	
	Code <not selected>	
	Code	LDC 2 .1e01
5	IdRestore	
	Lflag4Clear	
	Lflag5Set	
	SynReference(2)	
3	SemReference(2)	
1	Comm <not selected>	
	Code <not selected>	
	Code <not selected>	
	Code	STM 2
3	----	
5	----	

CHAPTER 7

CONCLUSION

The original purpose of this thesis was to produce a back end processor capable of converting DIANA (produced by the U.C.T. front end processor) to P-code. This goal has been achieved. A major extension on this is the ability for the user to configure the back end processor to handle a variety of input and output specifications simply by supplying the appropriate tables.

Aside from operating with the U.C.T. front end processor, the U.C.T. back end processor should prove to be a useful addition to the Unix supplied routines Lex and YACC. Such a combination should ease the production of many compilers.

In undertaking this project, a greater understanding has been gained about code generation techniques, Ada, DIANA and P-code. Also, much experience has been gained with the Unix operating system. The project provides a useful tool and a good basis for further research.

APPENDIX A

REFERENCES

[Goos 82]: G. Goos, W. Wulf
"DIANA Reference Manual"
Instutut fuer Informatik
Universitaet Karlsruhe and
Carnegie-Mellon University
March 1981

ALSO

A. Evans Jr., K. J. Butler
Tartan Laboratories Incorporated
28 February 1983

[SofTech 81]: G. Anderson, R. Clark, et al
"U.C.S.D. P-System and U.C.S.D. Pascal"
Version IV.0 International Architecture Guide
SofTech Microsystems, Inc.
March 1981

[Apple 80]: "Apple Pascal Operating System Reference Manual"
Apple Computer Inc.
1980

[Epstein 83]: J. Epstein
"The Implementation of a Front End Processor for a Subset of Ada"
M.Sc. Thesis
University of Cape Town
April 1983

[Gries 71]: D. Gries
"Compiler Construction for Digital Computers"
Cornell University
(John Wiley and Sons, Inc.)
1971

[Calingaert 79]: P. Calingaert
"Assemblers, Compilers and Program Translation"
University of North Carolina
(Pitman Publishing Limited)
June 1979

[Bornat 79]: R. Bornat
"Understanding and Writing Compilers"
Queen Mary College
University of London
(The MacMillan Press Ltd.)
1979

[Waite 84]: W. M. Waite, G. Goos
"Compiler Construction"
University of Colorado and
Institut fuer Informatik
Universitaet Karlsruhe
(Springer-Verlag)
1984

[Graham 80]: S. L. Graham
"Table Driven Code Generation"
IEEE "Computer" Journal
August 1980

APPENDIX B

FURTHER READING

G. Persch
"The Use of DIANA in Compilers, Language Transformers, Formatters and Debuggers"
Ada Implementation Group
Documentation Number DIANA-08
Institut fuer Informatik II
Universitaet Karlsruhe
29 June 1983

J. Uhl
"A Formal Definition of DIANA"
Ada Implementation Group
Documentation Number DIANA-FD
Institut fuer Informatik II
Universitaet Karlsruhe
28 June 1983

G. Persch, M. Dausmann
"The Intermediate Language DIANA Design and Implementation"
Ada Implementation Group
Documentation Number DIANA-07
Institut fuer Informatik II
Universitaet Karlsruhe
27 June 1983

M. Ganapathi, C. N. Fischer, J. L. Hennessy
"Retargetable Compiler Code Generation"
ACM "Computing Surveys" Journal
December 1982

"Reference Manual for the Ada Programming Language"
Draft Revised Mil-Std 1815 for the United States Department of Defense
Honeywell Inc., (U.S.A.) and
Alsys (France)
July 1982

T. E. Devine
"Contributions to Automatic Code Generation"
Ph.D. Thesis
University of California, Los Angeles
1982

R. G. G. Cattell
"Automatic Derivation of Code Generators from Machine Descriptions"
ACM "Transactions on Programming Languages and Systems" Journal
April 1980

Christopher Jay Terman
"The Specification of Code Generation Algorithms"
M.Sc. Thesis
Massachusetts Institute of Technology
January 1978

S. L. Graham, R. S. Glanville
"The Use of a Machine Description for Compiler Code Generation"
Computer Science Division - EECS
University of California, Berkeley
Information Technology, J. Moneta (editor)
(North Holland Publishing Company)
1978

K. V. Nori, U. Ammann, et al
"The Pascal <P> Compiler Implementation Notes, Revised Edition"
Institut fuer Informatik
Eidgenoessische Technische Hochschule, Zuerich
July 1976

J. S. Rohl
"An Introduction to Compiler Writing"
University of Manchester
Institute of Science and Technology
(Macdonald - London, American Elsevier Inc. - New York)
1975

F. L. Bauer, J. Eikel (editors)
"Compiler Construction - An Advanced Course"
Lecture Notes in Computer Science number 21
(Springer-Verlag)
1974

E. Humby
"Programs from Decision Tables"
International Computers Limited
(Macdonald - London, American Elsevier - New York)
1973

F. R. A. Hopgood
"Compiling Techniques"
Atlas Computer Laboratory
Science Research Council
(Macdonald - London, American Elsevier - New York)
1972

APPENDIX C

THE IMPLEMENTED P-CODE

An example set of tables is provided in order to demonstrate and verify the operation of the back end processor and to provide greater insight into how the system functions. The tree specification table contains the specification of DIANA, as supplied in Epstein's thesis. The code template table contains a set of P-code templates, command words and pointer indicators.

With the U.C.T. front end processor and the tables provided, the back end processor can produce P-code for a Pascal subset of Ada, with some of the extensions Ada provides over Pascal. The P-code used here is the SofTech version IV.0 - a dialect of U.C.S.D. P-code. The P-code used is kept as similar as possible to U.C.S.D. P-code for two reasons:

(1) by disassembling the P-code produced by the U.C.S.D. Pascal System, it is easy to verify (by comparison) the P-code produced by the Ada compiler for any similar Ada (and hence DIANA) constructs,

(2) it is advisable to produce a code which is kept as similar as possible to a well known (and well debugged) working code. By doing this, the problems associated with developing an assembler/interpreter for the code are minimized.

What follows is a description of the important DIANA constructs and the resulting P-code, as implemented in the tables. There may be better or more efficient layouts for the entries in the code template table. If the user so wishes, he/she may

re-design or add to these entries. Such flexibility in code generation (as well as tree scanning) is, after all, what this project is about!

C.1 The Main Ada-DIANA Definition P-code Structures

Most data structure definitions do not result in P-code being produced. The data structure definitions which do result in generated P-code are:

-- array definitions,

-- record definitions,

-- initialization at definition time.

For a data structure defined as a variable or pass-by-value (IN) parameter, space is assigned for that structure in the activation record of the currently defined procedure. For simple structures, this space is used to hold the value carried by the data structure. For complex structures, the space holds a dope vector (similar to a pointer or access type) which carries the address and size of the structure. This is so done because arrays and records are held on the heap as dynamic structures. For a data structure defined as a pass-by-reference (IN OUT or OUT) parameter, space is also assigned in the activation record of the currently defined procedure. However, in this case, the space is used to hold the address or dope vector of the data structure being passed as an IN OUT or OUT parameter.

In addition to data structure definitions, procedure and function definitions also result in the generation of P-code and ancillary information.

P-code templates and tree scanning controls are included in the code template table so as to allow the correct P-code to be generated for the DIANA constructs which represent the above.

C.1.1 Array Definition

Because of the manner in which DIANA represents an array element access, the resultant P-code is different to that produced by the U.C.S.D. Pascal system. In the U.C.T. system, the array upper and lower bounds (which are used in the bounds checks of indexes, as well as in the array definition phase) result in P-code being produced. Arrays are treated as dynamic data structures which are created only at the run time of the (assembled) P-code object. The advantage of this is that arrays can be fixed in size at run time, which some languages permit. In addition, if the evaluation of static expressions by the front end processor is deficient, evaluation at run time removes the need for a complex and restrictive static expression evaluator in the back end processor.

Unlike records and access types, the array type can be used as an anonymous type in a variable or parameter definition. But this does not complicate the situation. Figure C.1 depicts the DIANA construct for the array type.

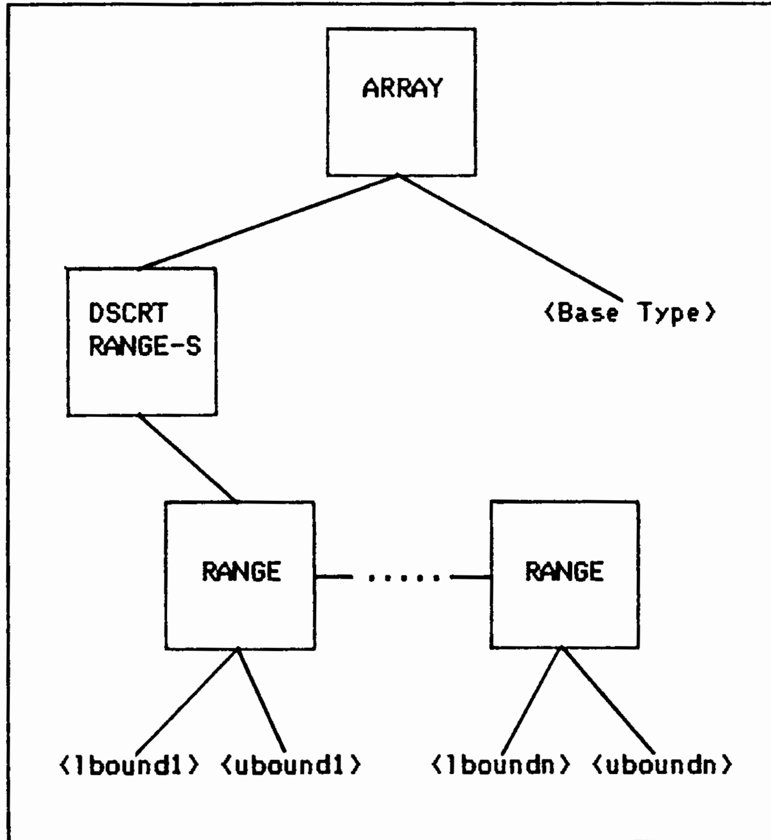


Fig. C.1: The DIANA ARRAY Type construct.

The P-code produced by the above construct must be able to do two things, namely to put onto TOS the size of the structure and to assign space in the heap and procedure activation record. These two operations must be separate. For example, when an array is being created as a variable, then the size must be calculated and the space assigned on the heap. However, only the size of an array is needed when it is a component of a record or the base type of another array definition. The following gives the P-code structures for both size calculation and space assignment (for a multi-dimensional array):

```
SLDC 1 -- start size calculation
<expression> -- upper bound 1
<expression> -- lower bound 1
SBI
DECI
MPI
 .
 .
<expression> -- upper bound n
<expression> -- lower bound n
SBI
DECI
MPI
<load base type size>
MPI -- size left on TOS

--------------------------------

<load address of dope vector>

<calculate size shown as above>

DUP
NEW
STM  2 -- store address and size
       -- of newly created space
       -- into dope vector
```

C.1.2 Record Definition

Records, like arrays, are defined as dynamic structures. This is done to simplify the handling of records when they contain array components. Records are similar to arrays in that dope vectors address the structures on the heap. Figure C.2 shows the tree construct for a record definition.

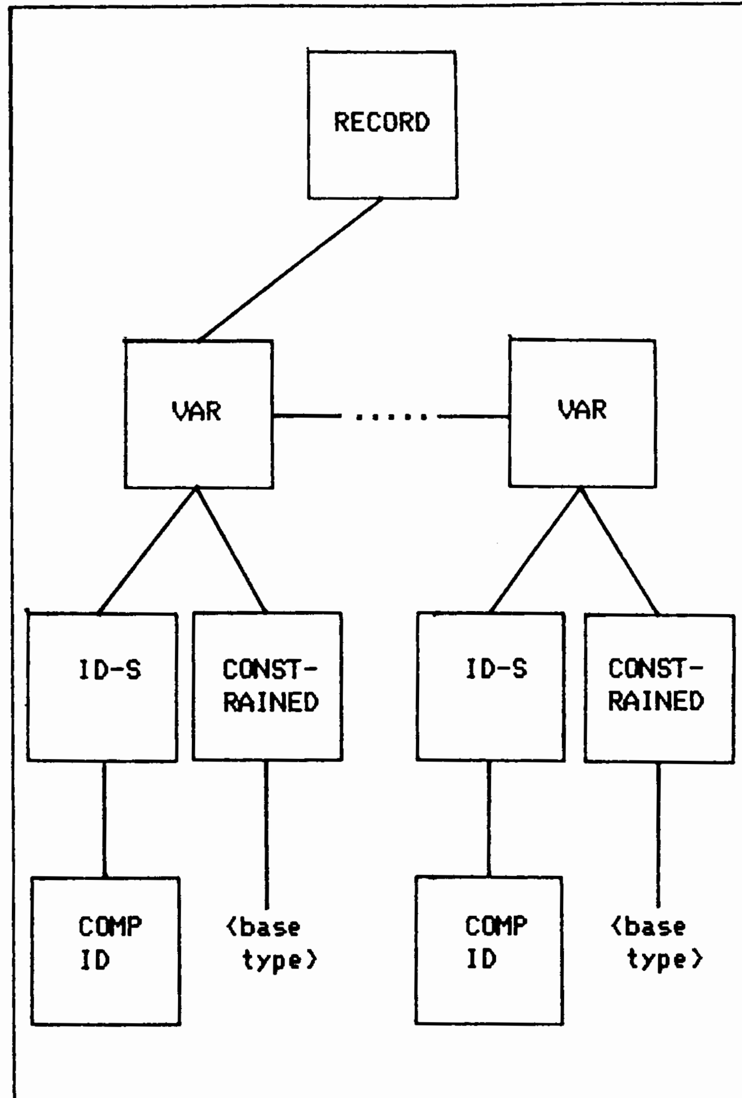


Fig C.2: The DIANA RECORD Type Construct.

When the record construct is first encountered in a type definition, space is assigned in the activation record of the currently defined procedure for every component, as if the components were defined normally as variables. But instead of holding values and dope vectors, the assigned activation record spaces hold the offsets that the components have relative to the start address of the record structure. P-code is generated to calculate these relative offsets and to assign them to the relevant activation record stores.

No space is assigned on the heap for a record structure in a type definition.

P-code is generated to calculate the size and assign space on the heap for a record structure in, for example, a variable or pass-by-value (IN) parameter definition. The component offsets at this stage are already calculated and stored in the activation record by the type definition of the structure.

Finally, the size of the record may be needed when it is a component or base type of another structure. In this case, P-code is produced to calculate the size and no space is assigned.

The first P-code structure given calculates the relative offset values of the components of a record and assigns these values to the relevant offsets:

```
SLDC 0 -- start offset calculation
DUP
<store in activation record component store>
<load size of 1st component>
ADI
  .
  .
DUP
<store in activation record component store>
<load size of nth component>
ADI
POP1 -- dispose of record size
```

The next P-code structure calculates the size of the record, assigns the required space on the heap and stores the address and size of the structure in the relevant activation record location. The same P-code structure, minus the P-code which assigns the heap space and stores the dope vector values in the activation record, is used to calculate the structure's size when only this is needed:

```
SLDC 0 -- start size calculation
<load size of 1st component>
ADI
 .
 .
<load size of nth component>
ADI

--------------------------------

<load address of dope vector>

<calculate size, as above>

DUP
NEW
STM  2 -- store address and size
       -- of newly created space
       -- into dope vector
```

Variant records and discriminants are catered for in the tables. However, the resulting records are not truly variant but 'pseudo' variant. The components of variant parts are handled in a manner identical to normal record components. So all components are assigned their own space in the resulting record structure, regardless of whether they are variant or not. The discriminants are treated as normal data structures. Any limitations to their use are applied by the front end processor.

C.1.3 Initialization at Definition Time

The initialization capabilities provided in the given table entries are somewhat limited compared to what Ada (DIANA) can do in this regard. Here, only block initialization of simple data structures and records is possible, but not of arrays. This is not a limitation of the back end processor, but merely a restriction resulting from the table

entries provided. There is nothing to stop the user from expanding these entries to encompass full Ada initialization.

The P-code structure for initialization is as follows:

```
-- for simple data structures
<load initialization value>
<store in appropriate location>

-------------------------------

-- for records
<load all initialization values>
<load size>
<load address>
INIT
```

In the above P-code, the non P-code instruction INIT exists. This command is regarded as a standard procedure. It takes the size and address of the relevant data structure from TOS and TOS-1 respectively and transfers the appropriate number of elements from the stack to the data structure. The size and address are restored to the stack after the operation. If wished, INIT can be easily implemented as normal in-line P-code. This is described in more detail in sub-chapter C.3 (Additional Standard Procedures) of this appendix.

C.1.4 Procedure Definition

The procedure definitions themselves do not result in much generated P-code. Only at the end of the defined procedure is there any P-code resulting from the procedure definition itself. All P-code generated by defined parameters falls into the defined data structures category. The DIANA construct of the procedure is given in figure C.3.

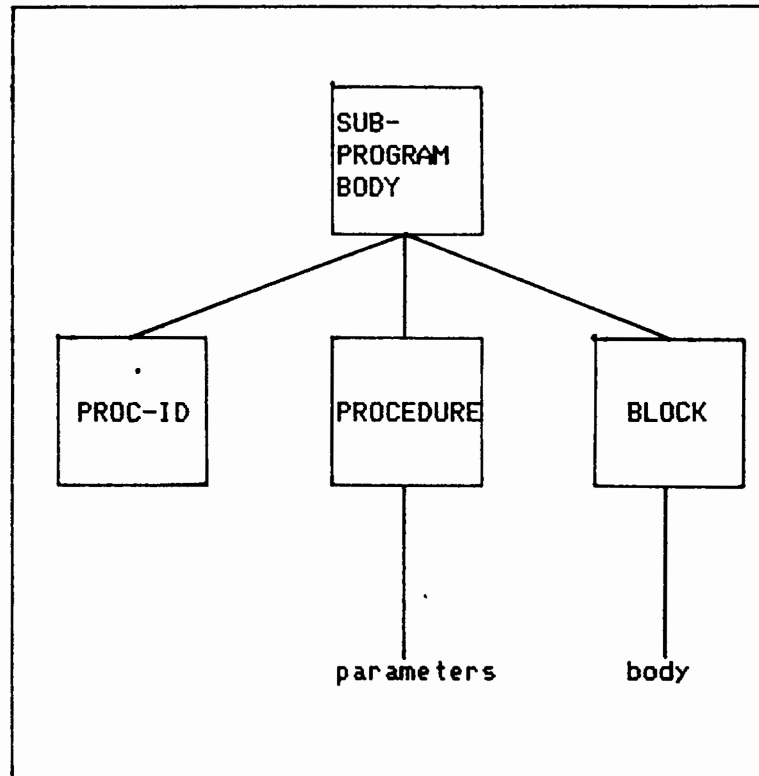


Fig C.3: The DIANA Procedure Definition Construct.

Only one P-code instruction is actually generated for the procedure definition. This is the 'RPU 0' instruction placed at the end of the P-code produced by the contents of the procedure (that is, the procedure parameter and data structure definitions plus the procedure body). The '0' indicates that there are no return values for this subprogram. Only for functions does this number vary.

After the RPU 0 instruction, the various ancillary data for the procedure follows. This consists of:

-- subprogram number,

-- lexical level of definition,

-- number of words needed by parameters,

-- number of words needed by local data structures.

The structure of the procedure definition is as follows:

```
<data structure P-code>
<initialization P-code>
<procedure body P-code>
RPU  0

PROCEDURE NUMBER = <number>
LEVEL OF DEFN.   = <number>
PARAMETERS       = <number>
DATA             = <number>
```

C.1.5 Function Definition

The function definition is very similar to the procedure definition. Figure C.4 gives the DIANA construct for the function definition.

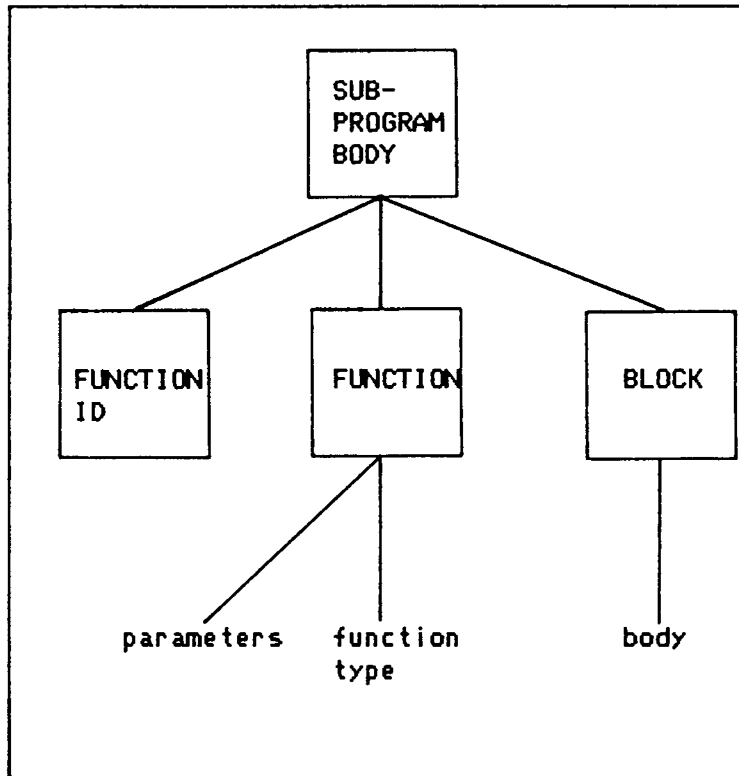


Fig C.4: The DIANA Function Definition Construct.

In the above diagram, it can be seen that the only difference in the tree construct is the function return type indicator. In most P-code dialects, this can be only a simple type which takes no more than two words. This is so because only two words are reserved for the return parameter. The P-code structure for a function definition is as follows:

```
<data structure P-code>
<initialization P-code>
<function body  P-code>
RPU  <number>

FUNCTION NUMBER = <number>
LEVEL OF DEFN.  = <number>
PARAMETERS      = <number>
DATA            = <number>
```

In this case, it can be seen that the 'RPU' instruction carries a <number> with it, where <number> is in the range 0 to 2. This <number> indicates how many words are being returned.

For both the procedure definition and the function definition, it must be remembered that the ancillary information is produced via the tables, just as the P-code is. In no way is this information, or the way it is represented, 'hard wired' in the code generator. If wished, it is a simple matter for the user to change the table entries so as to change what ancillary information is produced (if any) and how it is represented.

C.2 The Main Ada-DIANA Statement P-code Structures

The main Ada statements for which code must be generated are:

-- Expression,

-- Assignment,

-- If Statement,

-- Case Statement,

-- Loop and Exit Statements,

-- Goto Statement,

-- Procedure and Function calls and Return Statement,

-- Blocks.

In addition, there are noteworthy tree constructs in DIANA which represent complex data structure addressing (for storing or loading data). These are:

-- Array Element Addressing,

-- Record Component Addressing,

-- Access (Pointer) Addressing.

P-code templates and tree scanning controls are included in the code template table so as to allow the correct P-code to be generated for the DIANA constructs which represent the above.

C.2.1 Expression

An expression construct in DIANA consists mainly of the FUNCTION CALL node. This node has subtrees connected which represent the function and the parameters (or operands) of the function. The P-code necessary to push the operands onto the stack must be produced first because of the reverse Polish nature of the P-code stack machine. Only then is the P-code instruction(s) representing the function generated.

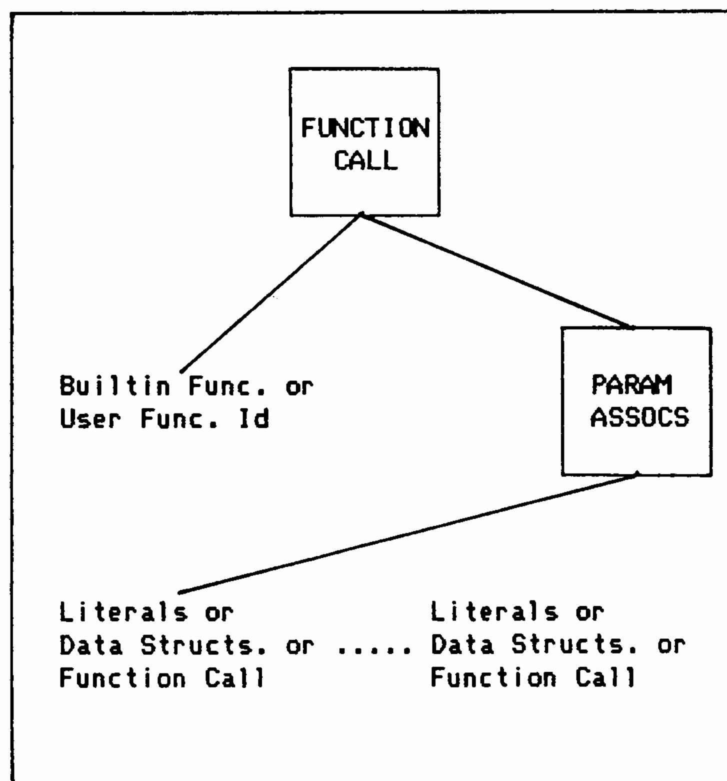


Fig. C.5: The DIANA FUNCTION CALL Construct.

The function can be either an Ada built-in operation (for example, '+', '>=') or a user defined function. The USED BLTN OP node represents the built-in operations, whereas the USED NAME ID node represents a user defined function.

A parameter can be either a literal (such as a numeric literal), a data structure identifier or another function call. In addition, for data structures, the parameter load can be either call-by-reference (IN OUT and OUT) or call-by-value (IN). Since parameter loading is the same for both functions and procedures, this is covered in the procedure call sub-chapter.

The function subtree scanning process described above is recursively repeated when dealing with a nested function call. This nesting of function calls builds up the expression represented by the tree construct (and hence produces a P-code structure representing that construct).

The P-code structure for a function call is as follows:

```
<parameter> --a literal, data structure or function call
     .
     .
     .
<parameter>
<function call> --a builtin operation or a user function
```

C.2.2 Assignment

This statement must produce P-code which evaluates an expression and stores the result in the specified data structure. The data structure may be simple such as an integer variable, or complex such as a component of a record in an array of records. Both the expression and the data structure are represented by subtrees connected to the ASSIGN node.

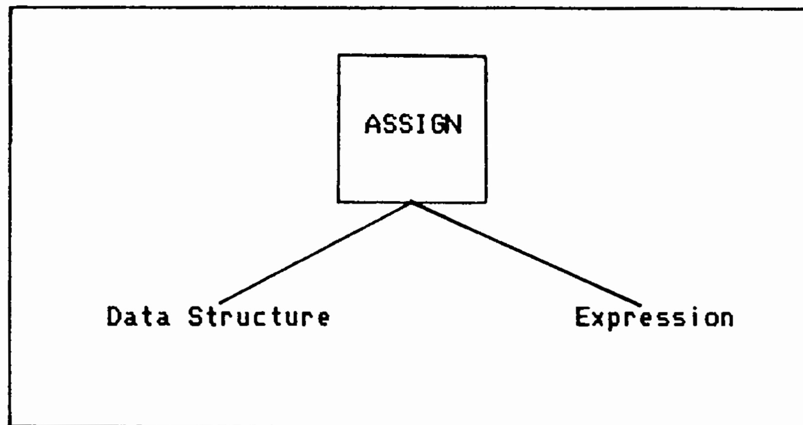


Fig. C.6: The DIANA ASSIGN Construct.

The P-code structure which represents an assignment is :

```
<Load Address of Structure> --only for complex data structures
<Expression>
<Store at immediate or addressed location>
```

When assigning to complex data structures, the address must be loaded onto the stack before the expression is evaluated. Only then is the actual store operation done. This means that the data structure subtree must be visited twice, first to generate the code to load the address (if necessary) and then (after generating the code to evaluate the expression) to generate the code to store the result. Here, loading the address includes any indexing done to access array elements and record components.

Use is made of the flag conditionals to control such code generation. This is done by selecting local flag combinations to indicate the address load condition and the value store condition. When descending the data structure subtree on the first visit, the flags are set up only for the address load condition. The P-code templates for the address load in the node(s) representing the data structure are enabled. If the type of the data structure requires an address load, then the necessary P-code is generated.

When the data structure subtree is traversed the second time, the flags are set up to enable the generation of the value store command in the same manner as described above.

C.2.3 If Statement

The code generated for this statement must select a block of P-code to execute dependent upon which controlling expression is true. The 'if' statement in Ada (and hence DIANA) can represent 'if', 'if...else', 'if elseif...elseif' and 'if elseif...elseif else' conditionals.

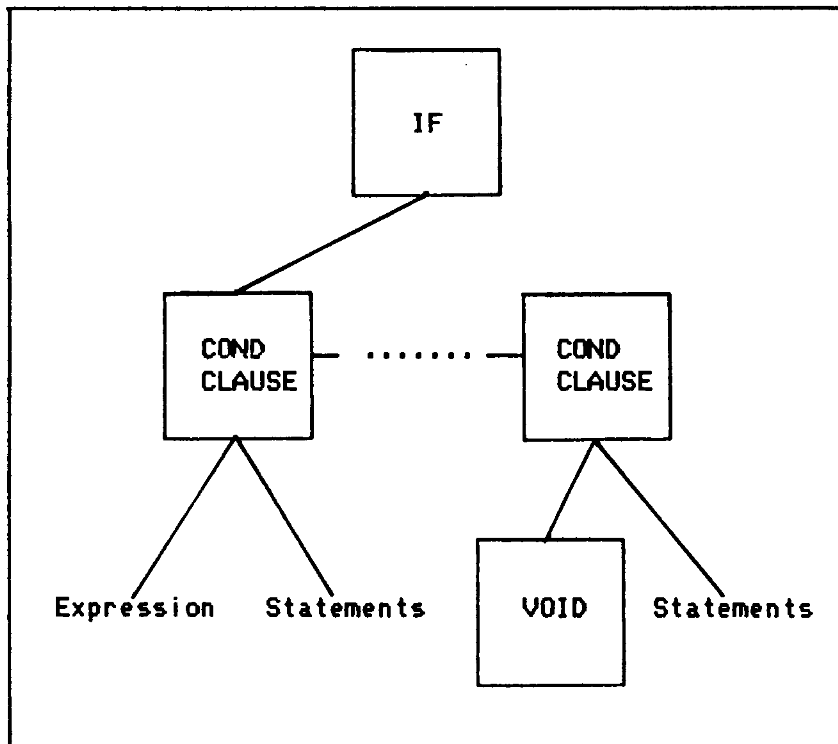


Fig. C.7: The DIANA IF Construct.

When scanning this construct, it must be determined when the COND CLAUSE node represents an 'else' clause, instead of an 'if' or 'elseif' clause. This is achieved by detecting the VOID node which substitutes for the conditional expression when the

'else' clause is represented. Each time the expression subtree is descended, a global flag is cleared. But in the VOID node, there is an instruction which unconditionally sets that flag. So, after descending the expression subtree, we can generate code according to whether the global flag is still clear (indicating that no VOID node was encountered - which means that the clause is an 'if' or 'elseif' clause), or whether the global flag is set (indicating that a VOID node was encountered - which means that the clause in an 'else' clause).

To eliminate any side effects whenever setting and clearing global flags, it is always advisable first to do a Stash operation on the flags. Then, if the values they currently contain are not needed, do a blanket clear on them before use . At the end of the node definition for the node kind concerned, a Restore operation on the global flags returns their original values before entry into the node was made.

In addition to flags, use is made of the local and global stores 2. The local store is used to hold the current jump or label number (for use in the generated P-code). The global store is used to hold the highest jump label number currently encountered.

All 'if' statement labels are prefixed by the string "IF" so as to avoid conflicts with the label numbers used by the 'loop' structures. Therefore the P-code assembler must be able to deal with with alphanumeric labels. This has the two advantages:

(1) the generation of the P-code is isolated from the machine upon which the P-code assembler/interpreter is implemented, implying greater portability,

(2) the P-code templates are simpler to produce.

The P-code structure for the 'if' statement is as follows:

```
<Expression>
FJP no. --if false, jump to the label number no.
<Statements>
UJP no. + 1 --skip to the end of the if statement
LBL no.
<Expression>
FJP no. + 2 --if false, jump to label number no. + 2
     .
     .
LBL no. + n
<Statements>
LBL no. + 1
LBL no. + 3
     .
     .
LBL no. + n + 1
```

Although it may seem inefficient to produce so many destination labels at the same place, they all reduce to a single address when the P-code is assembled. All the jumps are then to that same address.

C.2.4 Case Statement

The P-code generated by this system for representing 'case' statements is somewhat different to that produced by U.C.S.D. Pascal. Among other reasons, this is caused by the way DIANA represents multiple 'case' labels for each choice. There is an error in the U.C.T. front end processor regarding this. The aforementioned multiple 'case' label nodes (subtrees) are normally chained together by the link pointers (held in reference field 12 of the nodes). However, in trees produced by the U.C.T. front end processor, these nodes exist but are not connected. For the purposes of verifying this

thesis, the situation is manually corrected by 'editing' the relevant trees, thus allowing the U.C.T. back end processor to generate the correct P-code.

The correct DIANA construct for the 'case' statement is depicted in figure C.8.

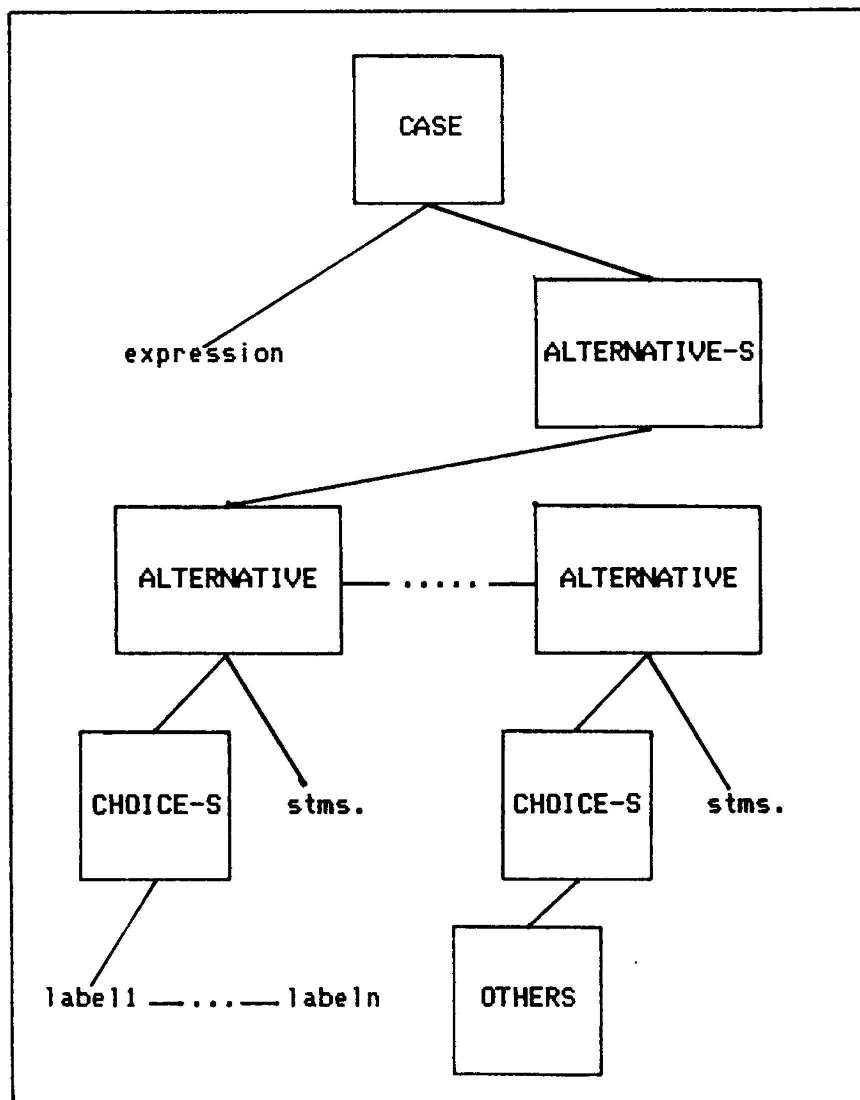


Fig. C.8: The DIANA CASE Construct.

Note the way multiple 'case' labels are simply chained together. For each alternative, this necessitates loading all such labels onto the stack before testing them. In addition, a 'case' selector store must be assigned within the activation record of the

current procedure so as to hold the result of the selector expression. The activation record location of the 'case' selector store is held and updated in local and global stores 4.

The OTHERS node in the above tree indicates the 'others' case in Ada. This case is selected only if all the explicit choices in the 'case' statement fail.

Local and global stores 3 are used to hold the 'case' label jump numbers. In order to avoid confusion with the 'loop' and 'if' labels, all 'case' label numbers are prefixed with yCASE, where y is from A to D (thereby distinguishing between the different labels within the 'case' statement).

The P-code structure for a 'case' statement is as follows:

```
<expression> -- the selector
<store in case variable>
  .
  .        -- cases 1 to x-1
  .
<load choice x1> -- list for case x
<load choice x2>
  .
  .
<load choice xm>
LDCI m+1
LBL  ACASEx
SWAP
<load from case variable>
<equal test>
TJP  BCASEx
DECI
DUP1
SLDC 1
EQUI
FJP  ACASEx
POP1
UJP  CCASEx
LBL  BCASEx
<statements> --attached to case x
UJP  DCASEx
LBL  CCASEx
 .
 .         --cases x+1 to n
 .
<statements> --others case
LBL  DCASEn
 .
 .
LBL  DCASEx
 .
 .
LBL  DCASE1
```

As with other structures, all the multiple label occurrences are reduced to a single address at assembly time.

C.2.5 Loop and Exit Statements

This statement has a number of variations. These are:

(1) a 'while loop', with an expression controlling the loop,

(2) a 'for loop', with a control variable,

(3) an unconditional loop, containing an 'exit' statement (enabling a controlled exit from the loop).

The 'exit' construct is handled separately.

All the above variations have the same basic structure, with different control subtrees hanging off the LOOP node (or, in the case of the unconditional loop, no control subtree).

As with the 'if' construct, the 'loop' construct uses the general purpose stores. In this case, the construct uses both local and global stores 1. All the labels in the P-code produced by this construct are prefixed by the string "LOOP".

The first construct to be explained is the 'while loop'. In DIANA, this takes the form of a LOOP node with a 'while' subtree and a statements subtree connected to it.

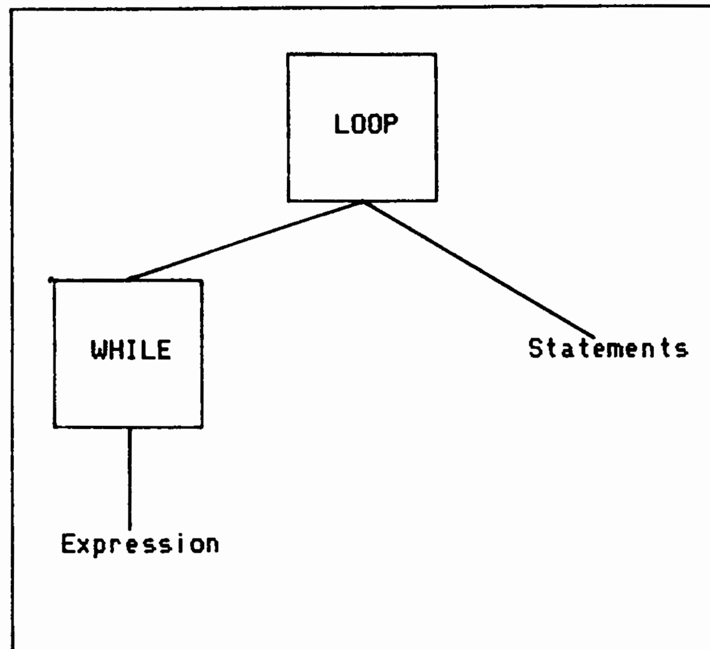


Fig. C.9: The DIANA WHILE LOOP Construct.

The P-code structure for this DIANA construct is as follows:

```
LBL no.
<Expression>
FJP no. + 1
<Statements>
UJP no.
LBL no. + 1
```

The 'for loop' DIANA construct is similar to that of the 'while loop'. In this case, the one subtree carries the 'for' conditional information and the other subtree carries the statements controlled by the loop.

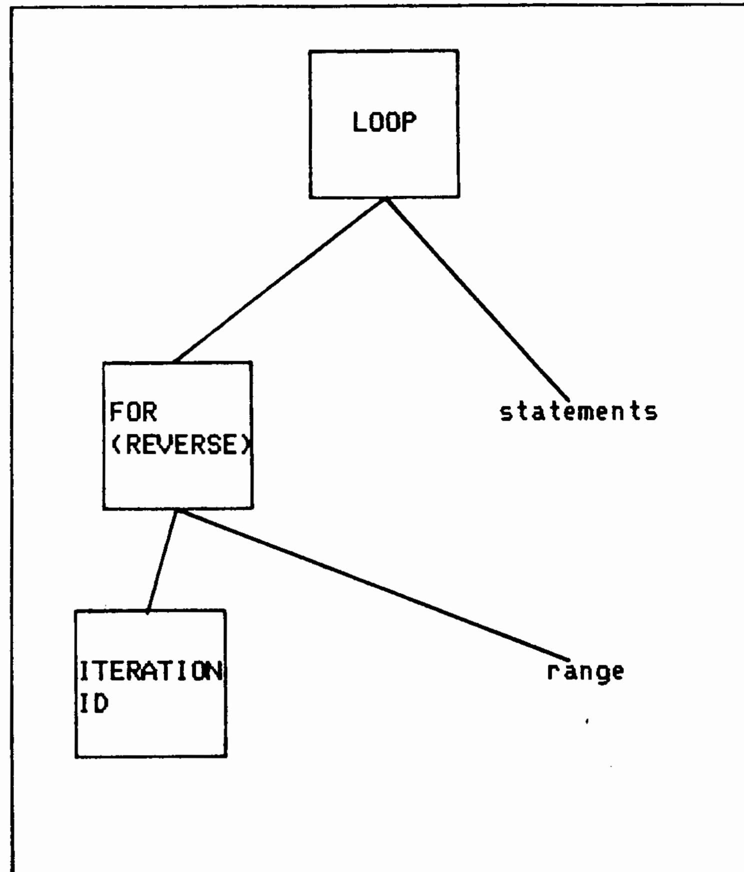


Fig C.10: The DIANA FOR LOOP Construct.

There is a closely related DIANA construct, namely the 'reverse' construct. This is identical to the 'for' construct, except that the FOR node is replaced by the REVERSE node. This simply indicates a 'for loop' with a reverse count.

The P-code structure for a 'for (reverse) loop' is as follows:

```
<load lower (upper) bound>
DECI (INCI)
<store in control variable>

LBL  LOOPn
<load control variable>
INCI (DECI)
DUP
<store in control variable>
<load upper (lower) bound>
LEQI (GEQI)
FJP  LOOPn+1

<statements>

UJP  LOOPn
LBL  LOOPn+1
```

The unconditional loop is similar in structure to the 'while loop' and 'for loop', except that the WHILE node (and its associated expression) or the FOR node is replaced by the VOID node. To exit this loop, use is made of the 'exit' statement. This is a separate statement and has no direct structural connection with the loop. First the 'loop' structure is described.

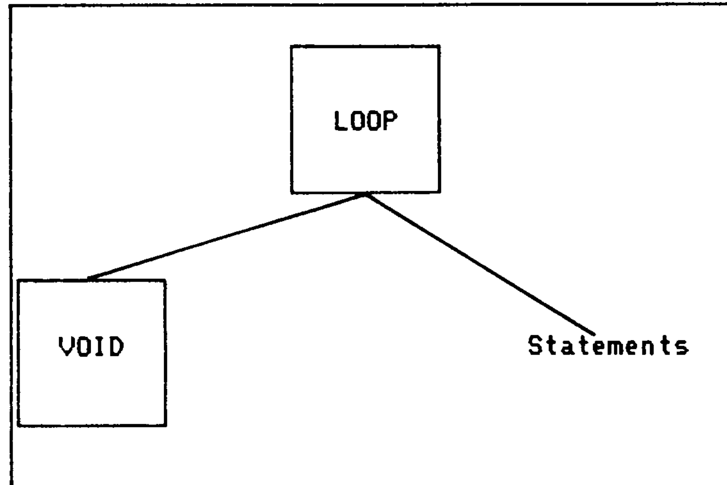


Fig. C.11: The DIANA Unconditional LOOP construct.

The P-code structure produced by the above DIANA construct is as follows:

```
LBL no.
<statements>
UJP no.
LBL no. + 1
```

Note the label at the end of the P-code above (LBL no. + 1). Although there is no explicit exit condition caused by the unconditional 'loop' structure, this label is used by the 'exit' statement.

To exit a loop in Ada, use can be made of the 'exit' statement. This causes an exit jump from any of the different 'loop' structures (not only the unconditional loop). It can take the form either of an unconditional exit or a conditional exit (with the addition of a 'when' clause). Also, the exit can be from the most recently entered loop (an unnamed exit) or from a specified loop (a named exit).

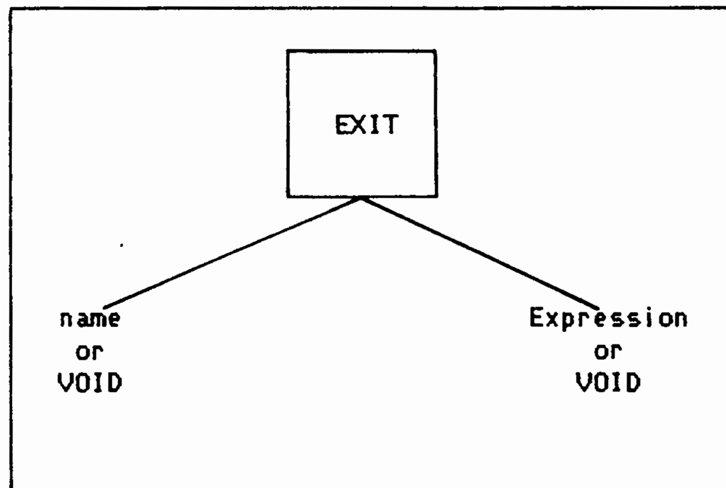


Fig. C.12: The DIANA EXIT Construct.

As with the 'if' structure, detection of the VOID node by the use of a global flag is employed to determine whether the exit is named or not and whether it is conditional or not.

In the case of a named exit, the LOOP node is preceded by a NAMED STM node. The NAMED STM node has the 'loop' structure as one of its subtrees. It also has another subtree which carries the name of the loop (in the LABEL ID node). When this structure is encountered, the 'loop' subtree is traversed first. Then the LABEL ID node is scanned. The traversal of the subtrees in this order results in the label being placed after the P-code representing the 'loop' structure. The label is generated with the loop's name as the label identifier. So if a named exit is encountered (in the P-code program), it causes a jump to the specified label, which is (in this case) a 'loop' structure's label. Hence it causes a jump to the end of the indicated 'loop' structure.

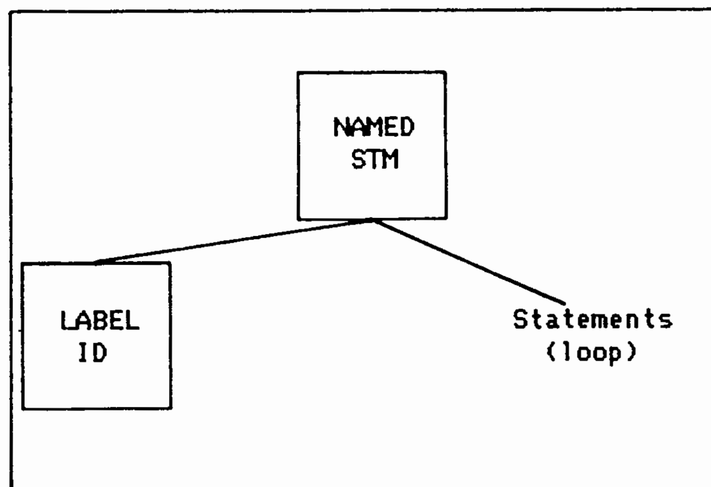


Fig. C.13: The DIANA NAMED STM Construct.

The P-code construct for a named 'loop' statement with an embedded 'exit' statement is as follows:

```
LBL no.
<Statements>
      .
      .
      .
<Expression> --part of the Exit statement
TJP --either to no. + 1 or to <name>
      .
      .
      .
UJP no.
LBL no. + 1
LBL <name>
```

There are two exit labels at the same location in the case of the named 'loop' statement. As mentioned before, such a situation is sorted out at assembly time because both labels are then reduced to a single destination address (for any jumps to these labels).

C.2.6 The Goto Statement

The 'goto' statement is very similar to the 'exit' statement. It comprises two parts, namely the label and the jump.

The label part is almost the same as the named statement part of the 'exit' statement except that the label(s) is put in front of the statement block in the P-code program. The DIANA construct for the labeled statement is shown in figure C.14.

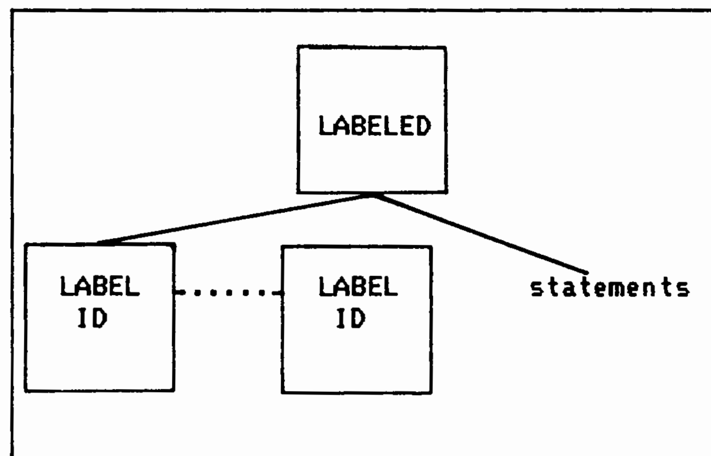


Fig. C.14: The DIANA LABELED Statement Construct.

The DIANA GOTO construct is given in figure C.15.

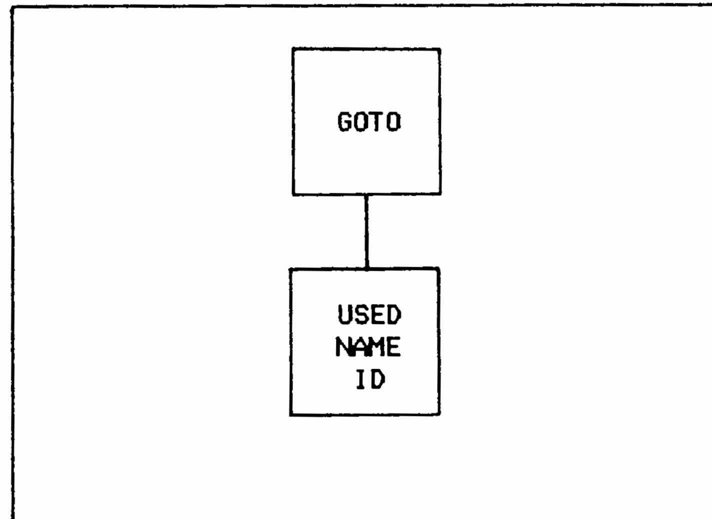


Fig. C.15: The DIANA GOTO Construct.

The table entries for the nodes in the above constructs simply produce a P-code instruction for an unconditional jump to the specified label.

C.2.7 Procedure and Function Calls and Return Statement

The DIANA function call construct is already covered in the expression sub-chapter. The procedure call construct is almost identical and is illustrated in figure C.16.

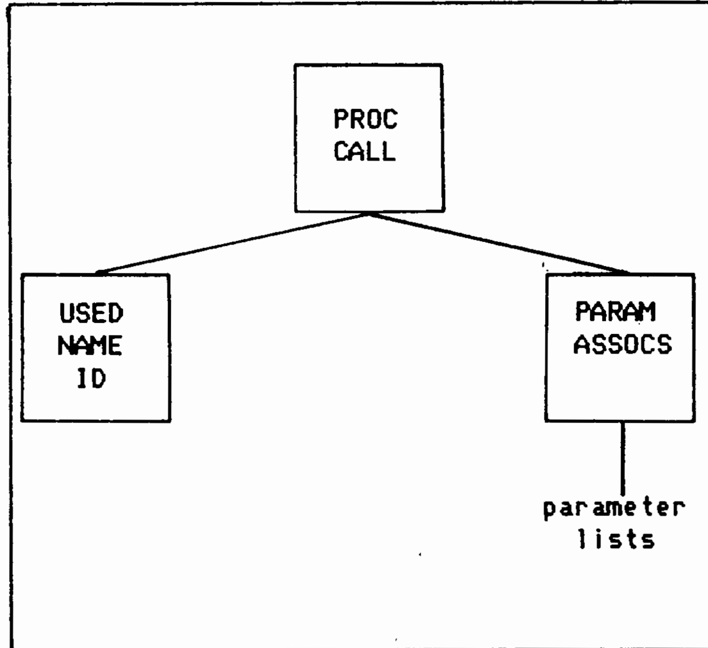


Fig. C.16: The DIANA PROCEDURE CALL Construct.

For both procedure calls and function calls, the relevant subprogram's parameter definition list is scanned so as to determine which parameters are call-by-reference (OUT and IN OUT) and which are call-by-value (IN). This is essential information because, for static data structures, call-by-reference parameters have their activation record addresses loaded onto the stack before the subprogram call. But call-by-value parameters have their contained values loaded onto the stack. For dynamic data structures, call-by-reference parameters have their dope vectors loaded onto the stack. However, call-by-value parameters are more complex. First space is assigned on the heap for the new structure. The size of this space is the same as that for the data structure being passed as a parameter. Then, using an additional standard procedure DXFR (see sub-chapter C.3), the parameter's contained data on the heap is copied into the newly assigned space. Finally, the dope vector of the new data structure is loaded into the activation record of the called subprogram. The new

dynamic data structure is indistinguishable from a local structure in the called subprogram - except that it already contains data.

The first two words of a function's activation record are used as the return value location. When a function is called, two integer zeros are pushed onto the stack after all the parameters are pushed. The U.C.S.D. P-machine needs these zero values to clear the two return words.

The DIANA construct which enables the return value to be loaded is the 'return' construct and is shown in figure C.17.

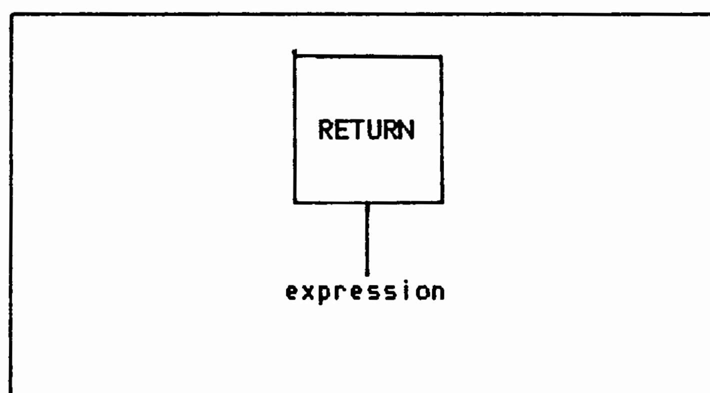


Fig. C.17: The DIANA RETURN Construct.

This construct simply produces a store instruction for the return value type. The return value is stored in activation record offset location 0 (and 1, if the return value is two words long).

C.2.8 Blocks

Blocks are handled in a manner resembling both that of a procedure and a statement list. Like a procedure, a block has an optional declarative part (data structure

definitions) and a statement list. However, it has no parameters and cannot be called. As such, its appearance in a P-code program is simply that of in-line code.

The data structure definitions for blocks are handled in a manner identical to that of procedures and functions. Space is assigned for a block's data structures in the activation record of the current procedure (or function) in which that block resides.

C.2.9 Array Element Addressing

As arrays are dynamically defined, array manipulation is somewhat different to that of normal Pascal produced P-code. Whenever the lower and upper bound values are loaded, they can result in P-code being produced to evaluate the lower and upper bound expressions. The conventional P-code array indexing command (IXA) is not powerful enough to handle the dynamic array indexing, so this is also handled in a different manner.

The structures for both an array element load and an array element store are the same. In both cases the method of addressing the element is identical. Only the final instruction specifies whether the access is a load or a store operation.

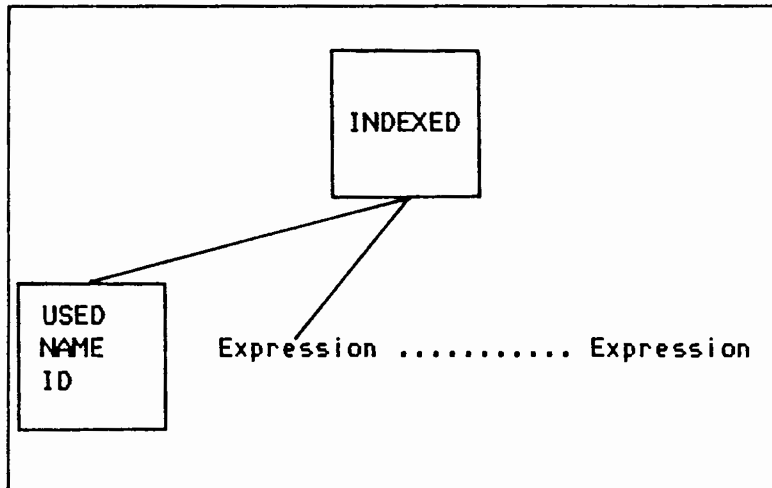


Fig. C.18: The DIANA Construct for a Simple Array Element Access.

A variation to the above simple array access is the nested array access. Here the base type of an array definition is also an array type. In this case the USED NAME ID node is replaced by another INDEXED node, along with its associated subtrees.

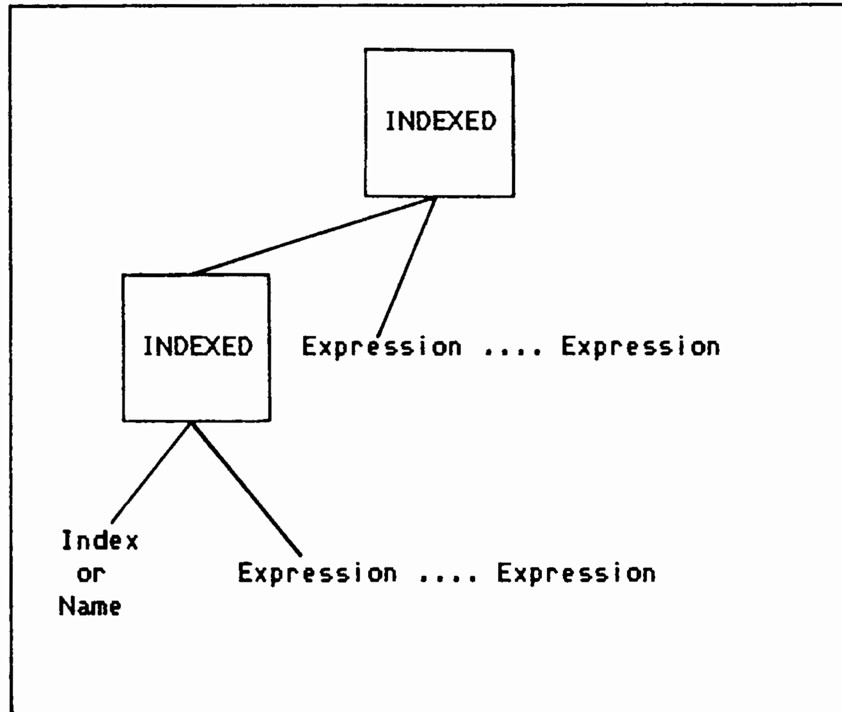


Fig. C.19: The DIANA Construct for a Nested Array Element Access.

The P-code structure produced by the DIANA simple multi-dimensional array access construct is as follows:

```
<expression> --index n
<expression> --index n-1
      .
      .
      .
<expression> --index 1
<load array start address>
<load base type size>
LDCI 0 --start of indexing accumulator
LDCI 1 --start of range multiplier

DUP
SWAP51

DUP --this checks index range
<expression> --lower bound 1
<expression> --upper bound 1
CHK

<expression> --lower bound 1
SBI
MPI
SWAP21
ADI
SWAP

<expression> --upper bound 1
<expression> --lower bound 1
SBI
LDCI 1
ADI
MPI
  .
  .
  .
```

```
DUP
SWAP51

DUP
<expression> --lower bound n
<expression> --upper bound n
CHK

SBI
MPI
SWAP21
ADI
SWAP

POP1
MPI
ADI
```

The above P-code reduces a simple multi-dimensional array and leaves the start address of the base type element on TOS. If the base type is an array and this too is indexed, then the tree scan continues to reduce the base type array in the manner just described. The walk is arranged such that all combinations of array accesses result in the indexes being pushed onto the stack in the correct order before any indexing is started. So the remaining indexes are on the stack ready and waiting. This holds also for combinations of record and array accesses.

The SWAP21 and SWAP51 instructions are additional standard procedures. They can be easily represented as normal in-line P-code if so wished. Sub-chapter C.3 (Additional Standard Procedures) covers this in more detail.

C.2.10 Record Component Addressing

As with arrays, record manipulation in this system is somewhat different to that of normal Pascal produced P-code. Like arrays, records are dynamically defined. For each component in the record, there exists an entry in the relevant procedure's

activation record which holds the offset value of the component relative to the start of the record. This allows for easy component start address calculation.

The DIANA construct for addressing a component of a record is the same for both a load and a store operation, and is illustrated in figure C.20.

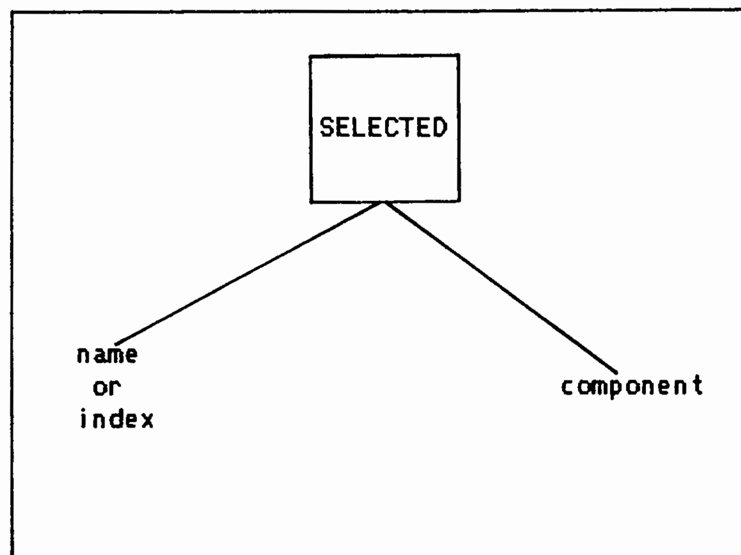


Fig. C.20: the DIANA for a Record Component Access.

In the above construct, the record name can be either a name or an index. This is so listed to cover the possibility of the record being an element of an array. The component also can be a complex data structure such as an array or record.

The P-code structure for a record component access is as follows:

```
<load base address of record>
<load component offset value>
ADI --add above to component address
```

C.2.11 Access Addressing

Before being able to address a dynamic structure, it first must be created. The DIANA construct for this is given in figure C.21.

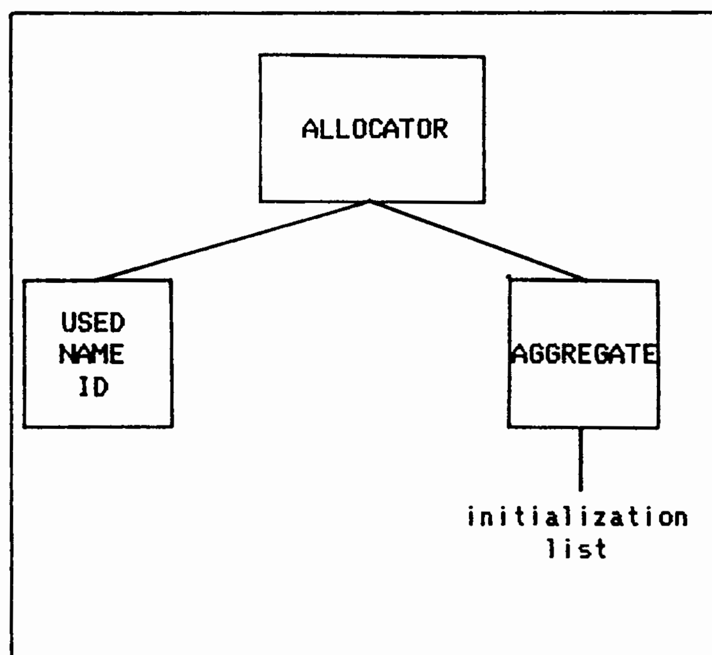


Fig. C.21: The DIANA NEW Construct.

The P-code structure which achieves the above is as follows:

```
<load any initialization values>
<calculate size of accessed data
 structure and load result>
DUP1
NEW
INIT -- if initialization values exist
```

Once a dynamic structure is created, the resulting dope vector which contains that structure's start address and length must be stored in the activation record. This is done in Ada (and DIANA) by using, for example, an assignment statement or parameter passing. The data structure which receives the dope vector is of type 'access' in Ada (DIANA).

If no initialization list exists, then the AGGREGATE subtree is replaced by the VOID node.

The actual addressing of the accessed structure is very similar to that of the record and array. In fact, the addressing of a normal record component and that of an accessed record component is nearly identical in both the DIANA construct and the resulting P-code.

In general, whenever a dynamic structure is addressed, the start address of that structure is simply loaded from the dope vector onto the stack.

C.3 Additional Standard Procedures

Included in the P-code table entries are references to three additional standard procedures, namely INIT, DXFR and SWAPmn. These routines perform mundane tasks,

as described below. If so wished, the function of all three standard procedures can be performed by equivalent conventional in-line P-code. This is not done here for two reasons:

(1) the example P-code listings supplied are already complex, so using the in-line versions of the standard procedures would reduce the readability of the examples,

(2) the three functions all need some scratch pad space in the activation record, so it is logical to create them as standard procedures where they would be autonomous units. They would then not need to have space assigned in the activation records of the P-code programs.

In the descriptions which follow, the in-line P-code versions of the additional standard procedures are included to demonstrate that they can be composed of normal P-code.

C.3.1 The INIT Standard Procedure

This procedure is used when initializing data structures. The data structure's dope vector (size and address) is taken from TOS. The initializing values are then transferred from the stack to the data structure. Afterwards, the dope vector is restored to TOS. The initializing values and the data structure's dope vector first must be put onto the stack. This is easily arranged with the scan of the appropriate subtrees in the DIANA tree.

The elements are loaded from the stack to the data structure, from the end of that structure to its beginning. This is done because the initialization values are loaded

onto the the stack in a forward order. A forward load into the data structure would result in the initialization stream being reversed because of the last-in-first-out nature of the stack. An alternative to this method is loading the initialization values onto the stack in a reverse order and then loading them into the data structure in a forward order. The reverse order stack load can be easily done by scanning the initialization nodes in the DIANA tree in a reverse order.

The in-line P-code for this operation requires some scratch pad space. This can be assigned in the relevant procedure's activation record. One possible in-line P-code program for INIT is as follows:

```
DUP1
STL <size>
SWAP
DUP1
STL <addr>
ADI
INC1
STL <temp>
LBL SYS*INIT<number>
LDL <temp>
DEC1
DUP1
STL <temp>
STM 1
LDL <temp>
LDL <addr>
EQUI
NFJ SYS*INIT<number>
LDL <addr>
LDL <size>
```

Note the choice of the label SYS*INIT. Such a label name is illegal in Ada source code. Therefore, no confusing duplication of the label name can occur at the P-code level. The <number> after the label is incremented at each occurrence of the above piece of code in the program. This avoids jump label confusion between each occurring INIT routine.

C.3.2 The DXFR Standard Procedure

This standard procedure is used whenever the contents of one complex data structure are to be assigned to another (similarly sized) complex data structure. As such, it is used mostly when a block assignment is done or when a block parameter pass is made. It transfers TOS-2 words from the address in TOS to the address in TOS-1. Unlike INIT, these size and address values are not restored to the stack after the operation. This is in keeping with the operation of the normal P-code store commands. However, like INIT, DXFR can be represented using conventional in-line P-code. The in-line P-code requires some scratch pad activation record space. The in-line P-code for DXFR is as follows:

```
SWAP21
LBL SYS*DXFR<number>
STL <size>
DUP1
SWAP21
DUP1
SWAP21
MOV 0,1
INC1
SWAP
INC1
SWAP
LDL <size>
DEC1
DUP1
LDCI 0
EQUI
NFJ SYS*DXFR<number>
POP1
POP1
```

As with INIT, the choice of label name (SYS*DXFR) is designed to avoid confusion with Ada source program labels. Also, the number after the label is incremented for each occurrence of the above piece of code in the P-code program.

C.3.3 The SWAPmn Standard Procedure

The in-line code for the DXFR instruction uses the new standard procedure SWAP21. There are a number of variations of this procedure, each indicated by the number pair attached. Each variation is, in fact, a separate routine. The instruction for SWAPmn swaps m words at TOS with n words at TOS-m (effectively a rotate). As with the other additional standard procedures, SWAPmn can be implemented in equivalent conventional in-line P-code. The in-line P-code for SWAPmn requires some scratch pad activation record space. The in-line P-code for SWAP21 (as used in DXFR) is:

```
STL <temp>
STL <temp+1>
STL <temp+2>

LDL <temp+1>
LDL <temp>
LDL <temp+2>
```

If the order on the stack before the operation was A - B - C then, after SWAP21, the order would be C - A - B.

Similarly, other swap operations can be implemented such as SWAP22, SWAP13 and so on. SWAP51 and SWAP21 are used in array indexing.

It is not necessary to implement a SWAP11 because such a swap already exists in the version of P-code used here. It is simply named SWAP and it swaps TOS with TOS-1.

APPENDIX D

TABLE LISTINGS

This appendix contains listings of the code template table and tree specification table. These demonstration tables enable the U.C.T. back end processor to take DIANA trees as input and produce P-code text, ready for assembly, as output. The DIANA trees are produced by the U.C.T. Ada subset front end processor. The P-code produced is SofTech version IV.0 - a U.C.S.D. dialect.

The above tables are used in the production of the examples given in Appendix E. With these examples, the tables are supplied to demonstrate and verify the operation of the back end processor system. Also, with these tables, the requirement that the back end processor operates with the U.C.T. front end processor is met.

The first listing is the code template table. The second listing is the tree specification table. The tables are input to the respective translators in the form exactly as listed in this appendix.

```
:*********************************************************************:
:THE CODE TEMPLATE TABLE                                              :
:=======================                                              :
:This table contains all the DIANA node usage and P-code template     :
:information necessary to enable the generation of P-code programs    :
:from DIANA trees.  What follows are the meanings of the user flag    :
:combinations used for this P-code implementation:                    :
:                                                                     :
:     fttxxxxxxxxxXXX    (Defn Phase) Define identifier               :
:     ftxtxxxxxxxxXXX    (Defn Phase) Supress space creation          :
:     ftxxtxxxxxxxXXX    (Defn Phase) Base type access                :
:     ftxxxtxxxxxxXXX    (Defn Phase) Base identifier access          :
:     ftxxxxtxxxxxXXX    (Defn Phase) Supress record/array space      :
:     ftxxxxxtxxxxXXX    (Defn Phase) Supress record component scan   :
:     ftxxxxxxtfxxXXX    (Defn Phase) Supress subprogram code         :
:     ftxxxxxxftxxXXX    (Defn Phase) Supress data struct & body code :
:     ftxxxxxxxxtxXXX    (Defn Phase) Variant part scan               :
:     ------------                                                    :
:     tfftxxxxxxxxXXX    (Stat Phase) Data segment address load       :
:     tfftfxxxxxxxXXX    (Stat Phase) Normal address load             :
:     tffttxxxxxxxXXX    (Stat Phase) Structure address load          :
:     tftffxxxxxxxXXX    (Stat Phase) Structure size load             :
:     tftftxxxxxxxXXX    (Stat Phase) Base type access                :
:     tfttfxxxxxxxXXX    (Stat Phase) Load lower bound                :
:     tftttxxxxxxxXXX    (Stat Phase) Load upper bound                :
:     tfxxxfftxxxxXXX    (Stat Phase) Normal load                     :
:     tfxxxftfxxxxXXX    (Stat Phase) Normal store                    :
:     tfxxxfttxxxxXXX    (Stat Phase) Address on stack load           :
:     tfxxxtffxxxxXXX    (Stat Phase) Address on stack store          :
:     tfxxxtftxxxxXXX    (Stat Phase) Print label with LBL prefix     :
:     tfxxxttfxxxxXXX    (Stat Phase) Parameter list scan             :
:     tfxxxtttxxxxXXX    (Stat Phase) Print subprogram statistics     :
:     tfxxxxxxtxxxXXX    (Stat Phase) Parameter load                  :
:     tfxxxxxxxtxxXXX    (Stat Phase) Reverse identifier list scan    :
:     tfxxxxxxxxtxXXX    (Stat Phase) Print label without LBL prefix  :
:     tfxxxxxxxxxtXXX    (Stat Phase) Generate initialization code    :
:     ------------                                                    :
:     tttxxxxxxxxxXXX    Generate a normal range check                :
:     ------------                                                    :
:     xxxxxxxxxxxxTXX    Void node encountered                        :
:     xxxxxxxxxxxxXTX    Initialization encountered                   :
:     xxxxxxxxxxxxXXT    All node encountered                         :
:                                                                     :
:Note that these flag combinations can be configured in any manner the :
:user requires, to indicate any meaning the user intends.  The above  :
:flag combinations could therefore be improved upon if so wished.     :
:                                                                     :
:Note also the 'convention' of putting the global flags in upper case. :
:This is, of course, not mandatory.  But it is felt that it helps to  :
:improve the readability of the table.                                :
:                                                                     :
:The general purpose stores are used for the following:               :
:                                                                     :
:    Local and Global stores 1 : loop statement jump labels,          :
:    Local and Global stores 2 : if statement jump labels,            :
:    Local and Global Stores 3 : case statement jump labels,          :
```

```
:    Local and Global Stores 4 : case statement variable locations.    :
:                                                                      :
:In addition, local store 1 is used, in conjunction with the           :
:comparison store, in the NUMERIC LITERAL node and the RANGE node.     :
:This is so done because no unpleasant interaction with the loop labels:
:results and therefore store usage is minimized.                       :
:**********************************************************************:

PCODEINFO            0 natives           float(2)
                                         integer(1)
                                         character(1)
                                         boolean(1)
                                         enumerated(1)
                                         access(2)
                                         array(2)
                                         record(2):
                                         offset1(2)
                                         offset2(0)
                                         offset3(0)
                                         proccount(0)
                                         lowerbound(11)
                                         startnode(11)
                                         globallevel(1);

ABORT                1                   unimplemented;

ACCEPT               2                   unimplemented;

ACCESS               3                   type access
: if global flag 3 is set, then an ALL node was encountered in an accessed
: record reference.   So return a record type instead of an access type.
: This will cause a block record reference instead of a reference to
: the pointer.
        comm ?UxxxxxxxxxxxXXT            type record;

ADDRESS              4                   unimplemented;

AGGREGATE            5                   SynReference(2)
                                         SynReference(12);

ALL                  6                   GflagStash
                                         ClearGflags
                                         Gflag3Set
                                         SynReference(2)
                                         GflagRestore
                                         SynReference(12);

ALLOCATOR            7                   LflagStash
                                         GflagStash

: set up the flags for a normal address and value load condition and traverse
: the list of initial values (if any) to generate the P-code necessary to
: load the values onto the stack.
                                         ClearLflags
                                         ClearGflags
                                         Lflag1Set
```

```
                                        Lflag4Set
                                        Lflag8Set
                                        SynReference(3)

: If syntactic link 3 carries only a VOID node, go to the defined data
: structure with the initialization flag conditions set up so as to
: generate any P-code to initialize the structure at definition time.
      comm ?UxxxxxxxxxxxxTXX            ClearLflags
                                        Lflag1Set
                                        Lflag12Set
                                        SynReference(2):

: set up the conditions necessary for loading the attributes of the base type
: of the identifier down syntactic link 2 and generate P-code to create
: the space in the heap for the size of this base type.
                                        ClearLflags
                                        Lflag1Set
                                        Lflag3Set
                                        SynReference(2)
      code                             /SLDC 2/
                                        SLDC 1/
                                        SLDC 1/
                                        SLDC 1/
                                        SLDC 1/
                                        SLDC 2/
                                        */
                                        *:
      code                              DUP1!
                                        NEW:

: If syntactic link 3 does not carry VOID, transfer the initial values to
: the newly pointed-to structure.   INIT is a standard procedure which
: stores the top 2 words on the stack in temporary variables and uses that
: information to copy the values on the stack into the required location.
: When this is done, these two words are replaced on the stack.
      code ?UxxxxxxxxxxxxFXX            INIT:

: If syntactic link 3 carries a VOID node, and the scan of the data
: structure type reveals initialization, then generate the INIT instruction
: as before.
      code ?UxxxxxxxxxxxxTTX            INIT:

                                        GflagRestore
                                        LflagRestore
                                        type access
                                        SynReference(12);

ALTERNATIVE          8                  Gstore3ToLstore3

                                        LflagStash
                                        GflagStash
                                        HistStash

: first set up the conditions necessary to load the test cases for
: this particular option.   Note the use of the parameter loading
: condition in order to allow all the case elements to be counted
```

```
: for this particular option.
                                                ClearGflags
                                                ClearLflags
                                                ClearHist v
                                                ResetHistIndex(1)
                                                Lflag1Set
                                                Lflag4Set
                                                Lflag8Set
                                                Lflag9Set
                                                SynReference(2)
                                                LflagRestore

: generate the code which goes through all the test cases and compares
: them with the case control element stored in the case temporary
: location.  This piece of code causes an iteration which tests every
: case for this particular option.  It then goes down syntactic link 3
: to generate the code for these cases.  Note that if this is an
: others case, then no testing code is generated.
        code ?UxxxxxxxxxxxxFXX              LDCI ^HI!
                                                LBL  ACASE^L3!
                                                SWAP:
        comm ?UxxxxxxxxxxxxFXX              CompStash
                                                CompReset(1):
        code ?UxxxxxxxxxxxxFXX      ?Txct /LAO  ^L4!
                                                LDM  2/
                                                LDO  ^L4/
                                                LDO  ^L4/
                                                LDO  ^L4/
                                                LDO  ^L4/
                                                */
                                                */
                                                *:
        code ?UxxxxxxxxxxxxFXX      ?Txcf /LLA  ^L4!
                                                LDM  2/
                                                LDL  ^L4/
                                                LDL  ^L4/
                                                LDL  ^L4/
                                                LDL  ^L4/
                                                */
                                                */
                                                *:
        code ?UxxxxxxxxxxxxFXX           /EQU  2/
                                                EQUI/
                                                EQUI/
                                                EQUI/
                                                EQUI/
                                                */
                                                */
                                                *:
        comm ?UxxxxxxxxxxxxFXX              CompRestore:
        code ?UxxxxxxxxxxxxFXX              TJP  BCASE^L3!
                                                DECI!
                                                DUP1!
                                                SLDC 1!
                                                EQUI!
                                                FJP  ACASE^L3!
```

```
                                            POP1!
                                            UJP  CCASE^L3!
                                            LBL  BCASE^L3:

                                            ClearGflags
                                            Gstore3Inc
                                            Lstore3Inc
                                            SynReference(3)
                                            Lstore3Dec
                                            GflagRestore

      code ?UxxxxxxxxxxxxFXX                UJP  DCASE^L3:
      code ?UxxxxxxxxxxxxFXX                LBL  CCASE^L3:

: finally, go to the next case option, if it exists.
                                            Lstore3Inc
                                            LflagRestore
                                            SynReference(12)
                                            Lstore3Dec
      code ?UxxxxxxxxxxxxFXX                LBL  DCASE^L3:

                                            HistRestore
                                            GflagRestore
                                            LflagRestore;


ALTERNATIVE-S          9                    SynReference(2);

AND-THEN               10                   null;

ARGUMENT-ID            11                   unimplemented;

ARRAY                  12                   type array

: The following table entries produce the P-code which works out the
: size of the array, assigns space for it in the dynamic heap and
: stores the size and start address in the data segment of the procedure.

: First P-code is generated to load the newly defined dope vector address
: of the array being defined.  The SLDC 1 constant load is needed for
: the P-code which calculates of the total number of elements in this
: array, as calculated from the range(s), which are obtained by following
: the syntactic link 2.
      code ?UfttxxxfxxxxxXXX          ?L1 LLA  ^N/
                                      ?Li ERR/
                                      ?Lg LAO  ^N:
      code ?UfttxxxxxxxxxXXX              SLDC 1:

      comm ?UfttxxxxxxxxxXXX              IdStash
                                          SynReference(2)
: This finds the attributes for the base type identifier which is necessary
: for producing the P-code to load its size.   Note tha[illegible] are set
: up for a definition basetype AND baseid search.
                                          ResetIdStore
                                          LflagStash
                                          ClearLflags
```

```
                                        Lflag2Set
                                        Lflag5Set
                                        Lflag6Set
                                        SynReference(3)
                                        LflagRestore:

: if the base type of the array is a record or array, set up the flags
: to load only the size and go back to the base type to generate the
: P-code to load the size.
      comm ?UfttxxxxxxxxxXXX          /*/
                                        */
                                        */
                                        */
                                        */
                                        */
                                        Lflag7Set
                                        SynReference(3)
                                        LflagRestore/
                                        Lflag7Set
                                        Lflag8Set
                                        SynReference(3)
                                        LflagRestore:

      code ?UfttxxxxxxxxxXXX          /SLDC 2/
                                        SLDC 1/
                                        SLDC 1/
                                        SLDC 1/
                                        SLDC 1/
                                        SLDC 2/
                                        */
                                        *:
      code ?UfttxxxxxxxxxXXX          MPI:

: After producing the P-code to load the base size, the original attributes
: are restored and the P-code is generated which works out the structure
: size, creates the new dynamic structure and stores the address and size
: in the data segment.  Note that if local flag 7 is set, then the space
: will not be created.  This is because this array definition will be for
: a record component.
      comm ?UfttxxxxxxxxxXXX          IdRestore:
      code ?UfttxxxfxxxxxXXX          DUP1!
                                        NEW!
                                        STM  2:

: The following table entries produce the P-code necessary to access an
: array element.  To calculate the actual indexing, the ranges are
: accessed to calculate the multiplying offset, as well as to check
: the ranges of the indices.  If this is one componemt of a nested
: array, then the walk will continue from this node to repeat the process
: on the next array node.  All these calculated sub offsets are organised
: such that the final offset will result naturally.

      comm ?UtffttxxxxxxxXXX          LflagStash
                                        Lflag3Set
                                        Lflag4Clear
                                        Lflag5Set
```

```
                                                  SynReference(3)
                                                  LflagRestore:

: if the base type of the array is a record or array, set up the flags
: to load only the size and go back to the base type to generate the
: P-code to load the size.
        comm ?UtffttxxxxxxxXXX                   /*/
                                                  */
                                                  */
                                                  */
                                                  */
                                                  */
                                                  LflagStash
                                                  ClearLflags
                                                  Lflag2Set
                                                  Lflag3Set
                                                  Lflag7Set
                                                  SynReference(3)
                                                  LflagRestore/
                                                  LflagStash
                                                  ClearLflags
                                                  Lflag2Set
                                                  Lflag3Set
                                                  Lflag7Set
                                                  Lflag8Set
                                                  SynReference(3)
                                                  LflagRestore:

        code ?UtffttxxxxxxxXXX                   /SLDC 2/
                                                  SLDC 1/
                                                  SLDC 1/
                                                  SLDC 1/
                                                  SLDC 1/
                                                  SLDC 2/
                                                  */
                                                  *:

        code ?UtffttxxxxxxxXXX                    SLDC 0!
                                                  SLDC 1:

        comm ?UtffttxxxxxxxXXX                    SynReference(2):

        comm ?UtffttxxxxxxxXXX                    SynReference(3);

ASSIGN                  13                        null
: Before doing anything, stash the current state of the flags.  Their
: condition can then be restored at the end of this assignment subtree.
                                                  LflagStash
                                                  GflagStash

: Set the flags for a normal address load condition.  Then follow syntactic
: reference 2 to generate the address/index/selected P-code and to obtain
: the final base type identifier attributes and associated type information
: for the LHS identifier.
                                                  Lflag3Clear
                                                  Lflag4Set
```

```
                                         Lflag5Clear
                                         SynReference(2)

: Stash the the freshly loaded base type identifier attributes and its
: associated type information.  This is necessary for generating the store
: command later in this sequence.  Set the flags for a normal load and
: a value load condition.  Then follow syntactic reference 3 to generate
: the P-code for the RHS expression.
                                         IdStash
                                         Lflag6Clear
                                         Lflag7Clear
                                         Lflag8Set
                                         SynReference(3)

: Set the flags for a store condition and restore the aforementioned base
: type information.  Then follow syntactic reference 2 once again to
: generate the store command for that base type.
                                         Lflag3Clear
                                         Lflag4Clear
                                         Lflag5Clear
                                         Lflag6Clear
                                         Lflag7Set
                                         Lflag8Clear
                                         IdRestore
                                         SynReference(2)

: Finally, reset the flags to their original status before leaving this
: assignment subtree.
                                         GflagRestore
                                         LflagRestore
                                         SynReference(12);

ASSOC                  14                SynReference(2)
                                         SynReference(3)
                                         SynReference(12);

ATTR-ID                15                unimplemented;

ATTRIBUTE              16                unimplemented;

ATTRIBUTE-CALL         17                unimplemented;

BINARY                 18                SynReference(2)
                                         SynReference(3)
                                         SynReference(4)
                                         SynReference(12);

BLOCK                  19                null

: if encountered during a statement phase in an in-line Ada block
: statement, set up the flag conditions to indicate a definition
: phase and descend syntactic link 2 to define any identifiers
: attached to this block.  The rest of the entries here are for
: normal sub-program blocks.
        comm ?UtfxxxxxxxxxXXX            LflagStash
                                         ClearLflags
```

```
                                        Lflag2Set
                                        SynReference(2)
                                        LflagRestore:

      comm ?UftxxxxxxxxxxXXX            SynReference(2):

      comm ?UftxxxxxxftxxXXX            ControlStash
                                        SynWalkSuspend
                                        CommWdSuspend:

                                        LflagStash
                                        ClearLflags
                                        Lflag1Set
                                        Lflag2Clear
                                        SynReference(3)
                                        LflagRestore

      comm ?UftxxxxxxftxxXXX            ControlRestore:

                                        SynReference(4)
                                        SynReference(12);

BOX                  20                 unimplemented;

CASE                 21                 GflagStash
                                        LflagStash

: first generate the P-code which evaluates the case selector expression.
                                        ClearLflags
                                        Lflag1Set
                                        Lflag4Set
                                        Lflag8Set
                                        SynReference(2)
                                        LflagRestore

: assign to local store 4 the location that the temporary case variable
: is going to use.
                                        Gstore4ToLstore4

: update the offset and case variable location counts accordingly.  This
: reserves the required space in the data segment.
                                        Offset1Inc
                                        Offset3Inc
                                        Gstore4Inc
      comm                             /Offset1Inc
                                        Offset3Inc
                                        Gstore4Inc/
                                        */
                                        */
                                        */
                                        */
                                        */
                                        */
                                        *:

: this tests the current level and produces an appropriate store
```

```
: operation for the case expression.   For every case statement,
: space is made available in the data segment of the current
: procedure for the case selector expression.
                                            CompStash
                                            CompReset(1)
        code                        ?Txct  /LAO  ^L4!
                                            STM  2/
                                            SRO  ^L4/
                                            SRO  ^L4/
                                            SRO  ^L4/
                                            SRO  ^L4/
                                            */
                                            */
                                            *:
        code                        ?Txcf  /LDL  ^L4!
                                            STM  2/
                                            STL  ^L4/
                                            STL  ^L4/
                                            STL  ^L4/
                                            STL  ^L4/
                                            */
                                            */
                                            *:
                                            CompRestore

                                            SynReference(3)

                                            GflagRestore

                                            SynReference(12);

CHOICE-S                22                  SynReference(2);

CODE                    23                  SynReference(2);

COMP-ID                 24                  null

: the following adds the identifier to the symbol table.
        comm ?Ufttxxxxfxxxx XXX    ?Rsnt   DefineId
                                            Offset1Assign
                                            IdInfoGrab:
        comm ?UfttxxxxxxxxXXX               LflagStash
                                            ClearLflags
                                            Lflag2Set
                                            Lflag5Set
                                            SemReference(2)
                                            LflagRestore:
        comm ?UfttxxxxfxxxxXXX              Offset1Update
                                            Offset3Update
                                            Gstore4Inc:
: update the case variable location count (Gstore4Inc above and below).
        comm ?UfttxxxxfxxxxXXX             /Gstore4Inc/
                                            */
                                            */
                                            */
                                            */
```

```
                                        Gstore4Inc/
                                        Gstore4Inc/
                                        Gstore4Inc:

: when defining a component, it is necessary to load the offset this
: component has relative to the start address into the component's
: data segment location.  This offset location, for a component, does
: not hold a value.  Instead, it holds the offset of the component
: from the start of the record.  The actual value is held in the
: dynamic heap.
      code ?Ufttxxxxfxxxx XXX ?Rslt ?Ll DUP1!
                                        STL  ^N/
                                    ?Li ERR/
                                    ?Lg DUP1!
                                        SRO  ^N:

: if the base type of this compenent is an array or record, set up the
: flags to load only the size and return to the base type to generate
: the P-code to load the size.
      comm ?UfttxxxxxxxxxXXX           /*/
                                        */
                                        */
                                        */
                                        */
                                        */
                                        Lflag7Set
                                        SemReference(2)
                                        LflagRestore/
                                        Lflag7Set
                                        Lflag8Set
                                        SemReference(2)
                                        LflagRestore:

      code ?UfttxxxxxxxxxXXX   ?Rslt /SLDC 2/
                                        SLDC 1/
                                        SLDC 1/
                                        SLDC 1/
                                        SLDC 1/
                                        SLDC 2/
                                        */
                                        *:

      code ?UfttxxxxxxxxxXXX   ?Rslt  ADI:

      comm ?UtffttxxxxxxxXXX           LflagStash
                                        Lflag3Set
                                        Lflag4Clear
                                        Lflag5Set
                                        SemReference(2)
                                        LflagRestore
                                        IdInfoGrab:

: the following generates the  P-code for the operation indicated by the
: flag conditions.  This is for one of the address or value store
: or load operations.
      code ?UtffttxxxxxxxXXX      ?Ll LDL  ^N/
```

```
                                        ?Li LOD  ^XD ^N/
                                        ?Lg LDO  ^N:

       code ?UtffttxxxxxxxXXX               ADI:

       comm ?UtffttxxxxxxxXXX              /*/
                                            */
                                            */
                                            */
                                            */
                                            */
                                            SemReference(2)/
                                            *:

: This code is to detect and load any component initialization upon request.
: Global flag 2 is used to indicate if the link to the initialization exists.
       comm ?UtfxxxxxxxxxtXXX          ?Rsnt Gflag2Set:
       comm ?UtfxxxxxxxxxtXTX               LflagStash
                                            GflagStash
                                            ClearLflags
                                            ClearGflags
                                            Lflag1Set
                                            Lflag4Set
                                            Lflag8Set
                                            SemReference(3)
                                            GflagRestore
                                            LflagRestore
                                            SynReference(12):

       comm ?UftxxxxxxxxxxXXX               SynReference(12);

COMP-REP             25                     unimplemented;

COMP-REP-S           26                     unimplemented;

COMP-UNIT            27                     IncLexicalLevel
                                            SynReference(2)
                                            SynReference(3)
                                            ClearLflags
                                            SynReference(4)
                                            SynReference(12);

COMPILATION          28                     SynReference(2);

COND-CLAUSE          29                     Gstore2ToLstore2

                                            GflagStash
                                            ClearGflags
                                            LflagStash
                                            ClearLflags

: this code first descends syntactic link 2 to load the P-code necessary
: to evaluate the conditional clause expression, if it exists.
                                            Lflag1Set
                                            Lflag4Set
                                            Lflag8Set
```

```
                                                SynReference(2)
                                                LflagRestore

: if the expression exists (determined by non detection of the VOID node)
: then generate the necessary P-code tests, jumps and labels along with the
: statements code.  If the conditional does not exist, then generate the
: statements and labels for an else case.
        code ?UxxxxxxxxxxxxFXX              FJP  IF^L2:
                                                Lstore2Inc
                                                Gstore2Inc
                                                Lstore2Inc
                                                Gstore2Inc

                                                SynReference(3)

                                                Lstore2Dec
        code ?UxxxxxxxxxxxxFXX          ?Rant UJP  IF^L2:
                                                Lstore2Dec
        code ?UxxxxxxxxxxxxFXX              LBL  IF^L2:

                                                SynReference(12)

                                                Lstore2Inc
        code ?UxxxxxxxxxxxxFXX          ?Ralt LBL  IF^L2:

                                                Lstore2Inc
                                                GflagRestore;

COND-ENTRY              30                      unimplemented;

CONST-ID                31                      SynReference(12);

CONSTANT                32                      SynReference(2)
                                                SynReference(3)
                                                SynReference(4)
                                                SynReference(12);

CONSTRAINED             33                      null

: These table entries follow the base type semantic link 5 to wherevever
: the base type is defined for a definition run.
        comm ?UftxxxxxxxxxxXXX              SemReference(5):

: If there is a base type link then the syntactic link 2 is ignored.   If
: there is no semantic link to any base type, then the USED NAME ID node must
: itself represent a native type and so this syntactic link 2 is followed,
: on a base type definition run.
        comm ?UftxxtxxxxxxxXXX          ?Rslf SynReference(2):

: When the size of the base type is calculated, it is necessary to obtain,
: from the symbol table, the attributes of the base type.   This information
: is necessary when wishing to load the size of arrays and records.   Such
: information is kept in the data segment.   When such a load is required,
: it is necessary to go down to the USED NAME ID node and follow the semantic
: link (if it exists) from there to the  definition node for that base type,
: and then to load the attributes into the current id store there.
```

```
        comm ?UftxxxxtxxxxxXXX          SynReference(2):

: When in the statement mode, we may wish to follow the semantic link 5 for
: a base type access, or to generate code for a structure or range check.
: So the link is followed for any statement condition.
        comm ?UtfxxxxxxxxxXXX           SemReference(5):

: If on a base type run in the statement phase, and semantic link 5 is null,
: the base type must then be represented by the USED NAME ID at the end of
: syntactic link 2.
        comm ?UtftftxxxxxxXXX     ?Rslf SynReference(2):

: when on a range check run, first examine syntactic reference 3 to see
: if any constraints are attached.   If not, then follow semantic link 6
: to the next nearest possible constraint (if any).
        comm ?UtttxxxxxxxxXXX           SynReference(3):
        comm ?UtttxxxxxxxxXXX     ?Ralf SemReference(6);

CONTEXT               34                SynReference(2);

CONVERSION            35                null
        comm ?UtfxxxxxxtxxXXX           SynReference(12):

                                        SynReference(3)

                                        GflagStash
                                        ClearGflags
                                        LflagStash
                                        ClearLflags
                                        Lflag1Set
                                        Lflag3Set
                                        Lflag5Set
                                        SynReference(2)
        code                           /FLT/
                                        RND/
                                        */
                                        */
                                        */
                                        */
                                        */
                                        *:

                                        LflagRestore
                                        GflagRestore

        comm ?UtfxxxxxxfxxXXX           SynReference(12);

DECL-REP-S            36                unimplemented;

DECL-S                37                SynReference(2);

DEF-CHAR              38                null;

DEF-OP                39                null;

DELAY                 40                unimplemented;
```

```
DERIVED              41                    SynReference(2);

DISCRIMANT-AGGREGATE 42                    SynReference(2);

DISCRMT-ID           43                    null
: these table entries create the space in the table, make the
: identifier current by loading in the current id store and
: update the global offset value.
             comm ?UfttxxxxxxxxxXXX ?Rsnt DefineId
                                           Offset1Assign
                                           IdInfoGrab
                                           LflagStash
                                           Lflag3Clear
                                           Lflag5Set
                                           Semreference(2)
                                           LflagRestore
                                           Offset1Update
                                           Offset3Update
                                           Gstore4Inc:
: Gstore4Incs above and below update the case variable location count
             comm ?UfttxxxxxxxxxXXX       /Gstore4Inc/
                                           */
                                           */
                                           */
                                           */
                                           Gstore4Inc/
                                           Gstore4Inc
                                           SemReference(2)/
                                           Gstore4Inc
                                           Lflag8Set
                                           SemReference(2)
                                           LflagRestore:

: The following handles the correct generation of any initialization at
: definition time which may exist.
             comm ?UfttxxxxxxxxxXXX          GflagStash:
             comm ?UfttxxxxxxxxxXXX ?Rsnt Gflag2Set:
             code ?UfttxxxxxxxxxXTX ?Ll /LLA  ^N/
                                           */
                                           */
                                           */
                                           */
                                           LLA  ^N/
                                           */
                                           */
                                      ?Li  ERR/
                                      ?Lg /LA0  ^N/
                                           */
                                           */
                                           */
                                           */
                                           LA0  ^N/
                                           */
                                           *:
             comm ?UfttxxxxxxxxxXTX        LflagStash
```

```
                                    IdStash
                                    Lflag1Set
                                    Lflag4Set
                                    Lflag8Set
                                    SemReference(3)
                                    IdRestore
                                    LflagRestore:
         code ?UfttxxxxxxxxxXTX ?Ll /STM  2/
                                    STL  ^N/
                                    STL  ^N/
                                    STL  ^N/
                                    STL  ^N/
                                    STM  2/
                                    */
                                    */
                               ?Li  ERR/
                               ?Lg /STM  2/
                                    STR  ^N/
                                    STR  ^N/
                                    STR  ^N/
                                    STR  ^N/
                                    STM  2/
                                    */
                                    *:

         comm ?UfttxxxxxxxxxXFX    /*/
                                    */
                                    */
                                    */
                                    */
                                    */
                                    */
                                    LflagStash
                                    ClearLflags
                                    Lflag1Set
                                    Lflag12Set
                                    SemReference(2)
                                    LflagRestore:

         comm ?UfttxxxxxxxxxXXX     GflagRestore:

: This entry is only for when a base type attribute search reaches this
: node.  It simply causes the attributes to be loaded from the table
: for the id that this node represents.
         comm ?UftxxxxtxxxxxXXX     IdInfoGrab:


: These table entries first stash the local flags, clears them and then
: sets the local flags to indicate a base type search.  By clearing all
: the flags, all other activities with no specific reference to the base type
: search are ignored.  Then follow semantic reference 2 to find the base type.
: After returning, restore the local flags to their original condition and
: access the symbol table information for this identifier.
                                    LflagStash
         comm ?UtffftxxxxxxxXXX     ClearLflags
                                    Lflag1Set
```

```
                                    Lflag3Set
                                    Lflag5Set:
         comm ?UtfftfxxxxxxxXXX     ClearLflags
                                    Lflag1Set
                                    Lflag3Set
                                    Lflag5Set:
         comm ?UtffttxxxxxxxXXX     ClearLflags
                                    Lflag1Set
                                    Lflag3Set
                                    Lflag5Set:
         comm ?UtftftxxxxxxxXXX     SemReference(2)
                                    IdInfoGrab:
                                    LflagRestore

: having found the attributes of the identifier (above), generate the P-code
: for address loads, value loads, value stores, etc. according to the
: conditions represented by the flags.
         code ?UtfftfxxxxxxxXXX ?Ll /LLA  ^N/
                                    */
                                    */
                                    */
                                    */
                                    LLA  ^N/
                                    LLA  ^N!
                                    LDM  2/
                                    LLA  ^N!
                                    LDM  2/
                               ?Li /LDA  ^XD ^N/
                                    */
                                    */
                                    */
                                    */
                                    LDA  ^XD ^N/
                                    LDA  ^XD ^N!
                                    LDM  2/
                                    LDA  ^XD ^N!
                                    LDM  2/
                               ?Lg /LAO  ^N/
                                    */
                                    */
                                    */
                                    */
                                    LAO  ^N/
                                    LAO  ^N!
                                    LDM  2/
                                    LAO  ^N!
                                    LDM  2:

         code ?UtffttxxxxxxxXXX ?Ll LLA  ^N!
                                    LDM  1/
                                ?Li LDA  ^XD ^N!
                                    LDM  1/
                                ?Lg LAO  ^N!
                                    LDM  1:

         comm ?UtffttxxxxxxxXXX    /*/
```

```
                                 */
                                 */
                                 */
                                 */
                                 */
                                 SemReference(2)/
                                 *:

code ?UtffftxxxxxxxXXX ?Ll LLA   ^N/
                       ?Li LDA   ^XD ^N/
                       ?Lg LAO   ^N:

code ?UtfxxxfftxxxxXXX ?Ll /LDM  2/
                           LDL   ^N/
                           LDL   ^N/
                           LDL   ^N/
                           LDL   ^N/
                           LDM   2/
                           */
                           */
                      ?Li /LDM   2/
                           LOD   ^XD ^N/
                           LOD   ^XD ^N/
                           LOD   ^XD ^N/
                           LOD   ^XD ^N/
                           LDM   2/
                           */
                           */
                      ?Lg /LDM   2/
                           LDO   ^N/
                           LDO   ^N/
                           LDO   ^N/
                           LDO   ^N/
                           LDM   2/
                           */
                           *:

code ?UtfxxxftfxxxxXXX ?Ll /STM  2/
                           STL   ^N/
                           STL   ^N/
                           STL   ^N/
                           STL   ^N/
                           STM   2/
                           DXFR/
                           DXFR/
                      ?Li /STM   2/
                           STR   ^XD ^N/
                           STR   ^XD ^N/
                           STR   ^XD ^N/
                           STR   ^XD ^N/
                           STM   2/
                           DXFR/
                           DXFR/
                      ?Lg /STM   2/
                           SRO   ^N/
                           SRO   ^N/
```

```
                                                SRO  ^N/
                                                SRO  ^N/
                                                STM  2/
                                                DXFR/
                                                DXFR:

                code ?UtfxxxfttxxxxXXX /LDM  2/
                                                LDM  1/
                                                LDM  1/
                                                LDM  1/
                                                LDM  1/
                                                LDM  2/
                                                */
                                                *:

: this sets up the range check load condition only for value loads.
                comm ?UtfxxxfxtxxxxXXX  LflagStash
                                                ClearLflags
                                                Lflag1Set
                                                Lflag2Set
                                                Lflag3Set:
                comm ?UtttxxxxxxxxxXXX /*/
                                                SemReference(2)/
                                                SemReference(2)/
                                                SemReference(2)/
                                                SemReference(2)/
                                                */
                                                */
                                                *:
                comm ?UtttxxxxxxxxxXXX  LflagRestore:

                code ?UtfxxxtffxxxxXXX /STM  2/
                                                STM  1/
                                                STM  1/
                                                STM  1/
                                                STM  1/
                                                STM  2/
                                                DXFR/
                                                DXFR:

                comm ?UfttxxxxxxxxxXXX  SynReference(12);

DSCRT-RANGE-S           44                      SynReference(2);

ENTRY                   45                      unimplemented;

ENTRY-CALL              46                      unimplemented;

ENTRY-ID                47                      null;

ENUM-ID                 48                      type enumerated

: print out the current value only.
        code ?UtfxxxfftxxxxXXX              LDCI ^VI1:

: print the lower bound and return.
```

```
        code ?UtfttfxxxxxxxXXX          LDCI ^VI1:

: find and print the upper bound.
        comm ?UtftttxxxxxxxXXX          SynReference(12):
        code ?UtftttxxxxxxxXXX     ?Ralf LDCI ^VI1;

ENUM-LITERAL-S          49              type enumerated
        comm ?UtfxxxxxxxxxxXXX          SynReference(2);

EXCEPTION               50              unimplemented;

EXCEPTION-ID            51              null;

EXIT                    52              null
: This saves the current flag setting and sets up a normal address and value
: load condition before descending syntactic link (3) to the conditional
: expression (if it exists).
                                        GflagStash
                                        LflagStash
                                        IdStash
                                        ClearLflags
                                        ClearGflags
                                        Lflag1Set
                                        Lflag4Set
                                        Lflag8Set
                                        SynReference(3)

: if a VOID node exists for the expression, then print out an
: unconditional jump to the label, otherwise print a conditional jump
: dependent upon what the expression is.  Note that the label will be
: appended to the jump instruction.
        code ?UxxxxxxxxxxxxTXX          UJP  *:
        code ?UxxxxxxxxxxxxFXX          TJP  *:

: set up the conditions to print out the label to append it to the
: just printed P-code jumps.  If this too has a VOID node then
: simply print the current LOOP exit number.
                                        Gflag1Clear
                                        ClearLflags
                                        Lflag1Set
                                        Lflag11Set
                                        SynReference(2)
        comm ?UxxxxxxxxxxxxTXX          Lstore1Dec:
        code ?UxxxxxxxxxxxxTXX          LOOP^L1:
        comm ?UxxxxxxxxxxxxTXX          Lstore1Inc:

                                        GflagRestore
                                        LflagRestore
                                        IdRestore;

EXP-S                   53              null
: This saves the current flag setting and sets up the condition necessary
: for a normal address and value load before decending the link (2) to the
: waiting expression list.  Note that local flag 10 can be used to indicate
: a reverse scan of this list, so do NOT clear it.  Note also that the
: parameter test flag 9 is cleared to stop this list interfering with a
```

```
: parameter load, as this list is used only for indexes etc.
                                                LflagStash
                                                GflagStash
                                                IdStash
                                                ClearGflags
                                                Lflag3Clear
                                                Lflag4Set
                                                Lflag5Clear
                                                Lflag6Clear
                                                Lflag7Clear
                                                Lflag8Set
                                                Lflag9Clear
                                                SynReference(2)
                                                IdRestore
                                                GflagRestore
                                                LflagRestore;

FIXED                   54                      type float
                                                SynReference(2)
        comm ?UtfxxxxxxxxxxXXX                  SynReference(3);

FLOAT                   55                      type float
                                                SynReference(2)
        comm ?UtfxxxxxxxxxxXXX                  SynReference(3);

FOR                     56                      LflagStash
: this node is arrived at from the LOOP node.   It is this node which
: turns the loop into a for loop.   Note that space is created in the
: data segment of the current procedure (or function) for the
: iteration identifier.   See also the REVERSE node.

: descend syntactic link 2 to define the iteration identifier.
                                                ClearLflags
                                                Lflag2Set
                                                Lflag3Set
                                                SynReference(2)
                                                IdStash

: descend syntactic link 3 to generate the P-code necessary to
: give the upper and lower bounds for the for loop.
                                                ClearLflags
                                                Lflag1Set
                                                Lflag3Set
                                                Lflag4Set
                                                SynReference(3)

        code                                    DECI:

                                                ClearLflags
                                                Lflag1Set
                                                Lflag7Set
                                                IdRestore
                                                SynReference(2)

        code                                    LBL LOOP^L1:
                                                Lstore1Inc
```

```
                                          ClearLflags
                                          Lflag1Set
                                          Lflag8Set
                                          IdRestore
                                          SynReference(2)

      code                                INCI!
                                          DUP1:

                                          ClearLflags
                                          Lflag1Set
                                          Lflag7Set
                                          IdRestore
                                          SynReference(2)

                                          ClearLflags
                                          Lflag1Set
                                          Lflag3Set
                                          Lflag4Set
                                          Lflag5Set
                                          SynReference(3)

      code                                LEQI!
                                          FJP  LOOP^L1;

FORMAL-DSCRT          57                  unimplemented;

FORMAL-FIXED          58                  unimplemented;

FORMAL-FLOAT          59                  unimplemented;

FORMAL-INTEGER        60                  unimplemented;

FUNCTION              61                  null
      comm ?Uftxxxxxxtfxx XXX              SynReference(2)
      comm ?Uftxxxxxxftxx XXX              SynReference(2)
      comm ?Utfxxxttfxxxx XXX              SynReference(2)
      comm ?Utfxxxtttxxxx XXX              SynReference(3);

FUNCTION-CALL         62                  null
      comm ?Utfxxxxxxxtxx XXX              SynReference(12):

                                          LflagStash
                                          HistStash
: stop side effects of local flags 9 and 10.
                                          Lflag9Clear
                                          Lflag10Clear

: set up the history index and go down to scan any parameter definitions.
: If the call is not to a user defined function, or there are no parameters,
: then set up the flags so that the loads are all normal, otherwise set up
: the flags so that they use the information stored in the history index to
: decide whether to load by reference or by value.  To detect if there are
: any parameter definitions, 'x' is stored in the first element of the index.
: If, after coming back from the parameter scan, that element still holds
```

```
: 'x', no parameters exist, or the function is a built in one.
                                        ResetHistIndex(1)
                                        Hist x

: the next Hist x is used for detection of unary operators by USED BLTN OP.
                                        IncHistIndex
                                        Hist x
                                        DecHistIndex

                                        ClearLflags
                                        Lflag1Set
                                        Lflag6Set
                                        Lflag7Set
                                        ControlStash
                                        PcodeSuspend
                                        SynReference(2)
                                        ControlRestore
                                        LflagRestore
: stop side effects of local flags 9 and 10.
                                        Lflag9Clear
                                        Lflag10Clear

: if there are any parameters, then set local flag 9, otherwise do nothing.
                                        ResetHistIndex(1)
        comm                       ?Hfx Lflag9Set:
                                        SynReference(3)

                                        Lflag9Clear

: set up a normal load condition which causes the function call code
: to be generated.
                                        Lflag6Clear
                                        Lflag7Clear
                                        Lflag8Set
                                        SynReference(2)

                                        HistRestore
                                        LflagRestore

        comm ?UtfxxxxxxtxxxXXX          IncHistIndex:
        comm ?UtfxxxxxxxfxxXXX          SynReference(12);

FUNCTION-ID           63                null

        comm ?UftxxxxxxtfxxXXX          ControlStash
                                        PcodeSuspend:

: if a parameter list scan arrives at this node then continue
: down semantic link 2
        comm ?UtfxxxttfxxxxXXX          SemReference(2):

: at definition time assign a P-code numeric identifier for this function
: and put it into the symbol table entry for this function.
        comm ?UfttxxxxxxxxxXXX          DefineId
                                        UpdateProcCount
                                        AssignProcCount:
```

```
: generate the P-code function call instruction preceded by two 0 words.
: the 0 words simply clear and reserve the first two words in the data
: segment of the called procedure in preparation for the function return
: value (if one exists).
        comm ?UtfxxxfftxxxxXXX          IdInfoGrab:
        code ?UtfxxxfftxxxxXXX          SLDC 0!
                                        SLDC 0:
        code ?UtfxxxfftxxxxXXX      ?Ll CLP  ^N/
                                    ?Li CIP  ^N/
                                    ?Lg CGP  ^N:

: descend semantic link 2 to determine the function return type.
: this is needed to determine whether one or two words are to be
: returned.  Then generate the return instruction with the
: appropriate return size parameter.
        comm ?UtfxxxtttxxxxXXX          LflagStash
                                        ClearLflags
                                        Lflag1Set
                                        Lflag3Set
                                        Lflag5Set
                                        SemReference(2)
                                        LflagRestore:
        code ?UtfxxxtttxxxxXXX          RPU  *:
        code ?UtfxxxtttxxxxXXX         /2!/
                                        1!/
                                        1!/
                                        1!/
                                        1!/
                                        2!/
                                        ERR!/
                                        ERR!:

: print out the P-code function data.  The format here was chosen
: for demonstration purposes.  However, any format can be chosen
: which may be most convenient.
        comm ?UtfxxxtttxxxxXXX          IdInfoGrab:
        code ?UtfxxxtttxxxxXXX          Function Number = ^N!
                                        Level of Defn.  = ^XI!
                                        Parameters      = ^02!
                                        Data            = ^03!!!:

        comm ?UftxxxxxxtfxxXXX          ControlRestore;

GENERIC               64                unimplemented;

GENERIC-ASSOC-S       65                unimplemented;

GENERIC-ID            66                null;

GENERIC-OP            67                unimplemented;

GENERIC-PARAM-S       68                unimplemented;

GOTO                  69                LflagStash
```

```
: print an unconditional jump instruction in preparation for the label,
: which will be appended to this instruction.
        code                                UJP  *:

: set up the conditions to print out the label to append it to the
: just printed P-code jump.
                                            ClearLflags
                                            Lflag1Set
                                            Lflag11Set
                                            SynReference(2)

                                            LflagRestore

                                            SynReference(12);

ID-S                    70                  SynReference(2);

IF                      71                  SynReference(2)
: examine the COND CLAUSE node for more if statement entries
                                            SynReference(12);

IN                      72                  null

        comm ?Uftxxxxxxftxx XXX             ControlStash
                                            PcodeSuspend:

                                            GflagStash
                                            LflagStash
                                            IdStash

        comm ?UftxxxxxxxxxxXXX              Lflag3Set:
                                            SynReference(2)
        comm ?UfttxxxxxxxxxXXX              Lflag3Clear:

                                            IdRestore
                                            LflagRestore
                                            GflagRestore

        comm ?UftxxxxxxftxxXXX              ControlRestore:

                                            SynReference(12);

IN-ID                   73                  null
: if this node is encountered during a parameter scan, load up the current
: element of the history index with the character representing pass-by-
: value and increment the history index.
              comm ?UtfxxxttfxxxxXXX        hist v
                                            IncHistIndex:

: define the identifier on the data segment, load its attributes into
: the current identifier store and access its base type by setting up
: the required conditions and going up the necessary link.
              comm ?UfttxxxxxxxxxXXX ?Rsnt DefineId
                                            Offset1Assign
                                            IdInfoGrab
                                            LflagStash
```

```
                                        Lflag1Clear
                                        Lflag5Set
                                        SemReference(2)
                                        LflagRestore
                                        Offset1Update
                                        Offset2Update
                                        Gstore4Inc:

: load start address of record or array parameter onto stack for later
: copy of the contents of this structure into the new local in parameter
: structure.
          code ?UfttxxxxxxxxxXXX ?Rslt ?L1 /*/
                                        */
                                        */
                                        */
                                        */
                                        */
                                        LLA  ^N!
                                        LDM  1/
                                        LLA  ^N!
                                        LDM  1/
                                    ?Li ERR/
                                    ?Lg /*/
                                        */
                                        */
                                        */
                                        */
                                        */
                                        LAO  ^N!
                                        LDM  1/
                                        LAO  ^N!
                                        LDM  1:
: for records and arrays, return to defintion of base type to produce code
: which defines these new local parameter structures.   Additional
: Gstore4Incs used to update case variable location count (also see above
: Gstore4Inc in this node definition).
          comm ?UfttxxxxxxxxxXXX    /Gstore4Inc/
                                        */
                                        */
                                        */
                                        */
                                        Gstore4Inc/
                                        Gstore4Inc
                                        SemReference(2)/
                                        Gstore4Inc
                                        SemReference(2):
: load the address and size of the newly created array or record structures
: and copy the contents of the structure in the parameter list to this newly
: created structure.
          code ?UfttxxxxxxxxxXXX ?Rslt ?L1 /*/
                                        */
                                        */
                                        */
                                        */
                                        */
                                        LLA  ^N/
```

```
                                LLA  ^N/
                           ?Li  ERR/
                           ?Lg /*/
                                */
                                */
                                */
                                */
                                */
                                LAO  ^N/
                                LAO  ^N:
          code ?UfttxxxxxxxxxXXX ?Rslt /*/
                                */
                                */
                                */
                                */
                                */
                                LDM  2!
                                SWAP21!
                                DXFR/
                                LDM  2!
                                SWAP21!
                                DXFR:

: This entry is only for when a base type attribute search reaches this
: node.  It simply causes the attributes to be loaded from the table
: for the id that this node represents.
          comm ?UftxxxxtxxxxxXXX     IdInfoGrab:

: These table entries first stash the local flags, clears them and then
: sets the local flags to indicate a base type search.  By clearing all
: the flags, all other activities with no specific reference to the base type
: search are ignored.  Then follow semantic reference 2 to find the base type.
: After returning, restore the local flags to their original condition and
: access the symbol table information for this identifier.
                                LflagStash
          comm ?UtffftxxxxxxxXXX     ClearLflags
                                Lflag1Set
                                Lflag3Set
                                Lflag5Set:
          comm ?UtfftfxxxxxxxXXX     ClearLflags
                                Lflag1Set
                                Lflag3Set
                                Lflag5Set:
          comm ?UtffttxxxxxxxXXX     ClearLflags
                                Lflag1Set
                                Lflag3Set
                                Lflag5Set:
          comm ?UtftftxxxxxxxXXX     SemReference(2)
                                IdInfoGrab:
                                LflagRestore

: having found the attributes of the identifier (above), generate the P-code
: for address loads, value loads, value stores, etc. according to the
: conditions represented by the flags.
          code ?UtfftfxxxxxxxXXX ?Ll /LLA  ^N/
                                */
```

```
                              */
                              */
                              */
                              LLA   ^N/
                              LLA   ^N!
                              LDM   2/
                              LLA   ^N!
                              LDM   2/
                         ?Li /LDA   ^XD ^N/
                              */
                              */
                              */
                              */
                              LDA   ^XD ^N/
                              LDA   ^XD ^N!
                              LDM   2/
                              LDA   ^XD ^N!
                              LDM   2/
                         ?Lg /LAO   ^N/
                              */
                              */
                              */
                              */
                              LAO   ^N/
                              LAO   ^N!
                              LDM   2/
                              LAO   ^N!
                              LDM   2:

code ?UtffttxxxxxxxXXX ?L1 LLA   ^N!
                              LDM   1/
                          ?Li LDA   ^XD ^N!
                              LDM   1/
                          ?Lg LAO   ^N!
                              LDM   1:

comm ?UtffttxxxxxxxXXX      /*/
                              */
                              */
                              */
                              */
                              */
                              SemReference(2)/
                              *:

code ?UtffftxxxxxxxXXX ?L1 LLA   ^N/
                          ?Li LDA   ^XD ^N/
                          ?Lg LAO   ^N:
code ?UtffftxxxxxxxXXX /*/
                        */
                        */
                        */
                        */
                        */
                        LDM   2/
                        LDM   2:
```

```
code ?UtfxxxfftxxxxXXX ?L1 /LDM  2/
                           LDL  ^N/
                           LDL  ^N/
                           LDL  ^N/
                           LDL  ^N/
                           LDM  2/
                           */
                           */
                       ?Li /LDM  2/
                           LOD  ^XD ^N/
                           LOD  ^XD ^N/
                           LOD  ^XD ^N/
                           LOD  ^XD ^N/
                           LDM  2/
                           */
                           */
                       ?Lg /LDM  2/
                           LDO  ^N/
                           LDO  ^N/
                           LDO  ^N/
                           LDO  ^N/
                           LDM  2/
                           */
                           *:

code ?UtfxxxftfxxxxXXX ?L1 /STM  2/
                           STL  ^N/
                           STL  ^N/
                           STL  ^N/
                           STL  ^N/
                           STM  2/
                           DXFR/
                           DXFR/
                       ?Li /STM  2/
                           STR  ^XD ^N/
                           STR  ^XD ^N/
                           STR  ^XD ^N/
                           STR  ^XD ^N/
                           STM  2/
                           DXFR/
                           DXFR/
                       ?Lg /STM  2/
                           SRO  ^N/
                           SRO  ^N/
                           SRO  ^N/
                           SRO  ^N/
                           STM  2/
                           DXFR/
                           DXFR:

   code ?UtfxxxfttxxxxXXX /LDM  2/
                           LDM  1/
                           LDM  1/
                           LDM  1/
                           LDM  1/
```

```
                                        LDM  2/
                                        */
                                        *:

: this sets up the range check load condition only for value loads.
                comm ?UtfxxxfxtxxxxXXX  LflagStash
                                        ClearLflags
                                        Lflag1Set
                                        Lflag2Set
                                        Lflag3Set:
                comm ?UtttxxxxxxxxxXXX /*/
                                        SemReference(2)/
                                        SemReference(2)/
                                        SemReference(2)/
                                        SemReference(2)/
                                        */
                                        */
                                        *:
                comm ?UtttxxxxxxxxxXXX  LflagRestore:

                code ?UtfxxxtffxxxxXXX /STM  2/
                                        STM  1/
                                        STM  1/
                                        STM  1/
                                        STM  1/
                                        STM  2/
                                        DXFR/
                                        DXFR:

                comm ?UtfxxxttfxxxxXXX  SynReference(12):
                comm ?UfttxxxxxxxxxXXX  SynReference(12);

IN-OP                74                 null;

IN-OUT               75                 null

        comm ?UftxxxxxxftxxXXX          ControlStash
                                        PcodeSuspend:

                                        GflagStash
                                        LflagStash
                                        IdStash

        comm ?UftxxxxxxxxxxXXX          Lflag3Set:
                                        SynReference(2)
        comm ?UfttxxxxxxxxxXXX          Lflag3Clear:

                                        IdRestore
                                        LflagRestore
                                        GflagRestore

        comm ?UftxxxxxxftxxXXX          ControlRestore:

                                        SynReference(12);

IN-OUT-ID            76                 null
```

```
: if this node is encountered during a parameter scan, load up the current
: element of the history index with the character representing pass-by
: reference and increment the history index.
            comm ?Utfxxxttfxxxx XXX     hist r
                                        IncHistIndex:

: define the identifier on the data segment, load its attributes into
: the current identifier store and access its base type by setting up
: the required conditions and going up the necessary link.
            comm ?UfttxxxxxxxxXXX    ?Rsnt DefineId
                                        Offset1Assign
                                        IdInfoGrab
                                        LflagStash
                                        Lflag1Clear
                                        Lflag5Set
                                        SemReference(2)
                                        LflagRestore:

            comm ?UfttxxxxxxxxXXX    ?Rslt Offset1Inc
                                        Offset2Inc
                                        Gstore4Inc:

: This entry is only for when a base type attribute search reaches this
: node.  It simply causes the attributes to be loaded from the table
: for the id that this node represents.
            comm ?UftxxxxtxxxxxXXX      IdInfoGrab:

: These table entries first stash the local flags, clears them and then
: sets the local flags to indicate a base type search.  By clearing all
: the flags, all other activities with no specific reference to the base type
: search are ignored.  Then follow semantic reference 2 to find the base type.
: After returning, restore the local flags to their original condition and
: access the symbol table information for this identifier.
                                        LflagStash
            comm ?UtffftxxxxxxxXXX      ClearLflags
                                        Lflag1Set
                                        Lflag3Set
                                        Lflag5Set:
            comm ?UtfftfxxxxxxxXXX      ClearLflags
                                        Lflag1Set
                                        Lflag3Set
                                        Lflag5Set:
            comm ?UtffttxxxxxxxXXX      ClearLflags
                                        Lflag1Set
                                        Lflag3Set
                                        Lflag5Set:
            comm ?UtftftxxxxxxxXXX      SemReference(2)
                                        IdInfoGrab:
                                        LflagRestore

: having found the attributes of the identifier (above), generate the P-code
: for address loads, value loads, value stores, etc. according to the
: conditions represented by the flags.
            code ?UtfftfxxxxxxxXXX ?Ll LDL  ^N/
                                   ?Li LOD  ^N/
                                   ?Lg LDO  ^N:
```

```
code ?UtfftfxxxxxxxXXX    /*/
                           */
                           */
                           */
                           */
                           */
                           LDM  2/
                           LDM  2:

code ?UtffttxxxxxxxXXX ?Ll LDL  ^N!
                           LDM  1/
                       ?Li LOD  ^XD ^N!
                           LDM  1/
                       ?Lg LDO  ^N!
                           LDM  1:

comm ?UtffttxxxxxxxXXX    /*/
                           */
                           */
                           */
                           */
                           */
                           SemReference(2)/
                           *:

code ?UtffftxxxxxxxXXX ?Ll LDL  ^N/
                       ?Li LOD  ^XD ^N/
                       ?Lg LDO  ^N:

   code ?UtfxxxfftxxxxXXX /LDM  2/
                           LDM  1/
                           LDM  1/
                           LDM  1/
                           LDM  1/
                           LDM  2/
                           */
                           *:

   code ?UtfxxxftfxxxxXXX /STM  2/
                           STM  1/
                           STM  1/
                           STM  1/
                           STM  1/
                           STM  2/
                           DXFR/
                           DXFR:

   code ?UtfxxxfttxxxxXXX /LDM  2/
                           LDM  1/
                           LDM  1/
                           LDM  1/
                           LDM  1/
                           LDM  2/
                           */
                           *:
```

```
: this sets up the range check load condition only for value loads.
                comm ?UtfxxxfxtxxxxXXX  LflagStash
                                        ClearLflags
                                        Lflag1Set
                                        Lflag2Set
                                        Lflag3Set:
                comm ?UtttxxxxxxxxxXXX /*/
                                        SemReference(2)/
                                        SemReference(2)/
                                        SemReference(2)/
                                        SemReference(2)/
                                        */
                                        */
                                        *:
                comm ?UtttxxxxxxxxxXXX  LflagRestore:

                code ?UtfxxxtffxxxxXXX /STM  2/
                                        STM  1/
                                        STM  1/
                                        STM  1/
                                        STM  1/
                                        STM  2/
                                        DXFR/
                                        DXFR:

                comm ?UtfxxxttfxxxxXXX  SynReference(12):
                comm ?UfttxxxxxxxxxXXX  SynReference(12);

INDEX                   77              SynReference(2)
                                        SynReference(12);

INDEXED                 78              null
        comm ?UtfxxxxxxxtxxXXX          SynReference(12):

                                        LflagStash
                                        Lflag10Clear

: convert a normal address load condition into an indexed address load
: condition
        comm ?UtfftfxxxxxxxXXX          Lflag3Clear
                                        Lflag4Set
                                        Lflag5Set:

: convert a normal value load condition into an address on stack load
: condition
        comm ?UtfxxxfftxxxxXXX          Lflag6Clear
                                        Lflag7Set
                                        Lflag8Set:

: convert a normal value store condition into an address on stack store
: condition
        comm ?UtfxxxftfxxxxXXX          Lflag6Set
                                        Lflag7Clear
                                        Lflag8Clear:
: reverse identifier list scan and descend syntactic reference 3 to
: index expressions.  Any side effects of the above flags are eliminated
```

```
: by the EXP-S node which connects to the parameter expressions.
        comm ?UtffttxxxxxxxXXX          Lflag10Set
                                        SynReference(3)
                                        Lflag10Clear:

                                        SynReference(2)

                                        LflagRestore

        comm ?UtfxxxxxxxfxxXXX          SynReference(12);

INNER-RECORD            79              LflagStash
                                        Lflag11Clear
                                        SynReference(2)
                                        LflagRestore;

INSTANTIATION           80              unimplemented;

INTEGER                 81              type integer
        comm ?UtfxxxxxxxxxxXXX          SynReference(2);

ITEM-S                  82              SynReference(2);

ITERATION-ID            83              null
: this defines and controls access to the iteration identifier
: needed for the for and reverse-for loops.  It is similar
: to normal variable definition and use, except simpler.
: Space is reserved for this identifier in the normal manner
: on the data segment of the current procedure (function).
        comm ?UfttxxxxxxxxxXXX          DefineId
                                        Offset1Assign
                                        IdInfoGrab
                                        LflagStash
                                        Lflag3Clear
                                        Lflag5Set
                                        SemReference(2)
                                        LflagRestore
                                        Offset1Update
                                        Offset3Update
                                        Gstore4Inc:
: update case variable location count (see Gstore4Inc above and below)
        comm ?UfttxxxxxxxxxXXX         /Gstore4Inc/
                                        */
                                        */
                                        */
                                        */
                                        Gstore4Inc/
                                        Gstore4Inc/
                                        Gstore4Inc:

                                        LflagStash
        comm ?UtffftxxxxxxxXXX          ClearLflags
                                        Lflag1Set
                                        Lflag3Set
                                        Lflag5Set:
        comm ?UtfftfxxxxxxxXXX          ClearLflags
```

```
                                                Lflag1Set
                                                Lflag3Set
                                                Lflag5Set:
        comm ?UtftftxxxxxxxXXX                  SemReference(2)
                                                IdInfoGrab:
                                                LflagRestore

        code ?UtffftxxxxxxxXXX              ?Ll LLA  ^N/
                                            ?Li LDA  ^XD ^N/
                                            ?Lg LAO  ^N:

        code ?UtfxxxfftxxxxXXX              ?Ll LDL  ^N/
                                            ?Li LOD  ^XD ^N/
                                            ?Lg LDO  ^N:

        code ?UtfxxxftfxxxxXXX              ?Ll STL  ^N/
                                            ?Li STR  ^XD ^N/
                                            ?Lg SRO  ^N:

        code ?UtfxxxfttxxxxXXX                  LDM  1:

: this sets up the range check load condition only for value loads.
        comm ?UtfxxxfxtxxxxXXX                  LflagStash
                                                ClearLflags
                                                Lflag1Set
                                                Lflag2Set
                                                Lflag3Set:
        comm ?UtttxxxxxxxxxXXX                 /*/
                                                SemReference(2)/
                                                SemReference(2)/
                                                SemReference(2)/
                                                SemReference(2)/
                                                */
                                                */
                                                *:
        comm ?UtttxxxxxxxxxXXX                  LflagRestore;

L-PRIVATE               84                      unimplemented;

L-PRIVATE-TYPE-ID       85                      unimplemented;

LABEL-ID                86                      null
        code ?UtfxxxtftxxxxXXX                  LBL  ^R:
        code ?UtfxxxxxxxxtxXXX                  ^R:
        comm ?UtfxxxtftxxxxXXX                  SynReference(12);

LABELLED                87                      LflagStash

: set up the flag conditions to cause the actual label to be printed
: at the beginning of the statements and follow syntactic link 2.
                                                ClearLflags
                                                Lflag1Set
                                                Lflag6Set
                                                Lflag8Set
                                                SynReference(2)
                                                LflagRestore
```

```
: then follow syntactic link 3 to the statements.
                                        SynReference(3)

                                        SynReference(12);

LOOP                  88                GstorelToLstorel

: go down syntactic link 2 to determine what kind of loop this is
: and generate the appropriate code.  The choice is between a
: while loop, a for loop and an unconditional loop.  The unconditional
: loop is identified by a VOID node being connected to link 2.
: Down syntactic link 3 are the statements controlled by the loop.
                                        GflagStash
                                        ClearGflags
                                        SynReference(2)
        code ?UxxxxxxxxxxxTXX           LBL  LOOP^L1:
                                        GflagRestore

                                        LstorelInc
                                        GstorelInc
                                        LstorelInc
                                        GstorelInc
                                        SynReference(3)
                                        LstorelDec
                                        LstorelDec

        code                            UJP  LOOP^L1:
                                        LstorelInc
        code                            LBL  LOOP^L1:

                                        SynReference(12);

MEMBERSHIP            89                SynReference(2)
                                        SynReference(3)
                                        SynReference(4)
                                        SynReference(12);

NAME-S                90                SynReference(2);

NAMED                 91                SynReference(2)
                                        SynReference(3)
                                        SynReference(12);

NAMED-STM             92                LflagStash

: first follow the syntactic link 3 to the statements.
                                        SynReference(3)

: then set up the flag conditions to cause the label to be printed
: at the and of the statements and follow syntacic link 2.
                                        ClearLflags
                                        Lflag1Set
                                        Lflag6Set
                                        Lflag8Set
                                        SynReference(2)
```

```
                                                LflagRestore

                                                SynReference(12);

NO-DEFAULT              93                      unimplemented;

NOT-IN                  94                      null;

NULL-ACCESS             95                      type access
        code                                    NULL:
                                                SynReference(12);

NULL-COMP               96                      null;

NULL-STM                97                      SynReference(12);

NUMBER                  98                      null

        comm ?Uftxxxxxxftxx XXX                 ControlStash
                                                PcodeSuspend:

        comm ?UftxxxxxxxxxxXXX                  GflagStash
                                                LflagStash
                                                IdStash:

        comm ?UftxxxxxxxxxxXXX                  Lflag3Set:
                                                SynReference(2)
        comm ?UfttxxxxxxxxxXXX                  Lflag3Clear:

        comm ?UftxxxxxxxxxxXXX                  IdRestore
                                                LflagRestore
                                                GflagRestore:

        comm ?UftxxxxxxftxxXXX                  ControlRestore:

                                                SynReference(12);

NUMBER-ID               99                      null
        comm ?UtfxxxfftxxxxXXX                  SemReference(3);

NUMERIC-LITERAL         100                     ResetIdStore

        comm ?UtfxxxxxxxtxxXXX                  SynReference(12):

: These table entries check to see if the semantic link 2 points
: to a node sequence no. 3.   If this is the case, then the
: literal this node represents is a float, otherwise it is an integer.
                                                SemReference(2)
                                                CompStash
                                                Lstore1Reset(3)
                                                RefNodeNoToCompStore(2)
        code                        ?T11t       LDC  2 ^VR1:
        comm                        ?T11t       type float:
        code                        ?T11f       LDCI ^R:
        comm                        ?T11f       type integer:
```

```
        comm ?Utfxxxxxxtxxx XXX          IncHistIndex:
        comm ?Utfxxxxxxxfxx XXX          SynReference(12);

OR-ELSE            101                   null;

OTHERS             102                   Gflag1Set
                                         SynReference(12);

OUT                103                   null

        comm ?Uftxxxxxxftxx XXX          ControlStash
                                         PcodeSuspend:

                                         GflagStash
                                         LflagStash
                                         IdStash

        comm ?Uftxxxxxxxxxx XXX          Lflag3Set:
                                         SynReference(2)
        comm ?Ufttxxxxxxxxx XXX          Lflag3Clear:

                                         IdRestore
                                         LflagRestore
                                         GflagRestore

        comm ?Uftxxxxxxftxx XXX          ControlRestore:

                                         SynReference(12);

OUT-ID             104                   null
: if this node is encountered during a parameter scan, load up the current
: element of the history index with the character representing pass-by
: reference and increment the history index.
            comm ?Utfxxxttfxxxx XXX      hist r
                                         IncHistIndex:

: define the identifier on the data segment, load its attributes into
: the current identifier store and access its base type by setting up
: the required conditions and going up the necessary link.
            comm ?Ufttxxxxxxxxx XXX     ?Rsnt DefineId
                                         Offset1Assign
                                         IdInfoGrab
                                         LflagStash
                                         Lflag1Clear
                                         Lflag5Set
                                         SemReference(2)
                                         LflagRestore:

            comm ?Ufttxxxxxxxxx XXX     ?Rslt Offset1Inc
                                         Offset2Inc
                                         Gstore4Inc:

: This entry is only for when a base type attribute search reaches this
: node.   It simply causes the attributes to be loaded from the table
: for the id that this node represents.
```

```
            comm ?UftxxxxtxxxxxXXX     IdInfoGrab:

: These table entries first stash the local flags, clears them and then
: sets the local flags to indicate a base type search.  By clearing all
: the flags, all other activities with no specific reference to the base type
: search are ignored.  Then follow semantic reference 2 to find the base type.
: After returning, restore the local flags to their original condition and
: access the symbol table information for this identifier.
                                       LflagStash
            comm ?UtffftxxxxxxxXXX     ClearLflags
                                       Lflag1Set
                                       Lflag3Set
                                       Lflag5Set:
            comm ?UtfftfxxxxxxxXXX     ClearLflags
                                       Lflag1Set
                                       Lflag3Set
                                       Lflag5Set:
            comm ?UtffttxxxxxxxXXX     ClearLflags
                                       Lflag1Set
                                       Lflag3Set
                                       Lflag5Set:
            comm ?UtftfxxxxxxxXXX      SemReference(2)
                                       IdInfoGrab:
                                       LflagRestore

: having found the attributes of the identifier (above), generate the P-code
: for address loads, value loads, value stores, etc. according to the
: conditions represented by the flags.
            code ?UtfftfxxxxxxxXXX ?Ll LDL  ^N/
                                   ?Li LOD  ^N/
                                   ?Lg LDO  ^N:
            code ?UtfftfxxxxxxxXXX    /*/
                                       */
                                       */
                                       */
                                       */
                                       */
                                       LDM  2/
                                       LDM  2:

            code ?UtffttxxxxxxxXXX ?Ll LDL  ^N!
                                       LDM  1/
                                   ?Li LOD  ^XD ^N!
                                       LDM  1/
                                   ?Lg LDO  ^N!
                                       LDM  1:

            comm ?UtffttxxxxxxxXXX    /*/
                                       */
                                       */
                                       */
                                       */
                                       */
                                       */
                                       SemReference(2)/
                                       *:
```

```
          code ?UtffftxxxxxxxXXX ?Ll LDL  ^N/
                                 ?Li LOD  ^XD ^N/
                                 ?Lg LDO  ^N:

             code ?UtfxxxfftxxxxXXX /LDM  2/
                                    LDM  1/
                                    LDM  1/
                                    LDM  1/
                                    LDM  1/
                                    LDM  2/
                                    */
                                    *:

             code ?UtfxxxftfxxxxXXX /STM  2/
                                    STM  1/
                                    STM  1/
                                    STM  1/
                                    STM  1/
                                    STM  2/
                                    DXFR/
                                    DXFR:

             code ?UtfxxxfttxxxxXXX /LDM  2/
                                    LDM  1/
                                    LDM  1/
                                    LDM  1/
                                    LDM  1/
                                    LDM  2/
                                    */
                                    *:

: this sets up the range check load condition only for value loads.
             comm ?UtfxxxfxtxxxxXXX  LflagStash
                                    ClearLflags
                                    Lflag1Set
                                    Lflag2Set
                                    Lflag3Set:
             comm ?UtttxxxxxxxxxXXX /*/
                                    SemReference(2)/
                                    SemReference(2)/
                                    SemReference(2)/
                                    SemReference(2)/
                                    */
                                    */
                                    *:
             comm ?UtttxxxxxxxxxXXX  LflagRestore:

             code ?UtfxxxtffxxxxXXX /STM  2/
                                    STM  1/
                                    STM  1/
                                    STM  1/
                                    STM  1/
                                    STM  2/
                                    DXFR/
                                    DXFR:
```

```
                 comm ?UtfxxxttfxxxxXXX  SynReference(12):
                 comm ?UfttxxxxxxxxxXXX  SynReference(12);

PACKAGE-BODY          105                unimplemented;

PACKAGE-DECL          106                unimplemented;

PACKAGE-ID            107                null;

PACKAGE-SPEC          108                unimplemented;

PARAM-ASSOC-S         109                Lflag3Clear
                                         Lflag4Set
                                         Lflag5Clear
                                         Lflag6Clear
                                         Lflag7Clear
                                         Lflag8Set
                                         SynReference(2);

PARAM-S               110                SynReference(2);

PARENTHESISED         111                SynReference(2)
                                         SynReference(12);

PRAGMA                112                SynReference(2)
                                         SynReference(3)
                                         SynReference(12);

PRAGMA-ID             113                SynReference(2);

PRAGMA-S              114                SynReference(2);

PRIVATE               115                unimplemented;

PRIVATE-TYPE-ID       116                unimplemented;

PROC-ID               117                null

        comm ?UftxxxxxxtfxxXXX           ControlStash
                                         PcodeSuspend:

: if a parameter list scan arrives at this node then continue
: down semantic link 2.
        comm ?UtfxxxttfxxxxXXX           SemReference(2):

: at definition time assign a P-code numeric identifier for this
: procedure and put it into the symbol table entry for this procedure.
        comm ?UfttxxxxxxxxxXXX           DefineId
                                         UpdateProcCount
                                         AssignProcCount:

: generate the P-code procedure call instruction.
        comm ?UtfxxxfftxxxxXXX           IdInfoGrab:
        code ?UtfxxxfftxxxxXXX      ?Ll  CLP  ^N/
                                    ?Li  CIP  ^N/
                                    ?Lg  CGP  ^N:
```

```
        code ?UtfxxxtttxxxxXXX          RPU  0!:

: print out the P-code procedure data.   The format here was chosen
: for demonstration purposes.   However any format can be chosen
: which may be most convenient.
        comm ?UtfxxxtttxxxxXXX          IdInfoGrab:
        code ?UtfxxxtttxxxxXXX          Procedure Number = ^N!
                                        Level of Defn.   = ^XI!
                                        Parameters       = ^02!
                                        Data             = ^03!!!:

        comm ?UftxxxxxxtfxxXXX          ControlRestore;

PROCEDURE              118              null
        comm ?UftxxxxxxtfxxXXX          SynReference(2):
        comm ?UftxxxxxxftxxXXX          SynReference(2):
        comm ?UtfxxxttfxxxxXXX          SynReference(2);

PROCEDURE-CALL         119              null

                                        LflagStash
                                        HistStash
: stop side effects of local flags 9 and 10.
                                        Lflag9Clear
                                        Lflag10Clear

: set up the history index and go down to scan any parameter definitions.
: If there are no parameters, then set up the flags so that the loads
: are all normal, otherwise set up the flags so that they use the
: information stored in the history index to decide whether to load by
: reference or by value.   To detect if there are any parameter definitions,
: 'x' is stored in the first element of the index.   If, after
: coming back from the parameter scan, that element still holds 'x', no
: parameters exist.
                                        ResetHistIndex(1)
                                        Hist x
                                        ClearLflags
                                        Lflag1Set
                                        Lflag6Set
                                        Lflag7Set
                                        ControlStash
                                        PcodeSuspend
                                        SynReference(2)
                                        ControlRestore
                                        LflagRestore
: stop side effects of local flags 9 and 10.
                                        Lflag9Clear
                                        Lflag10Clear

: if there are any parameters, then set local flag 9, otherwise do nothing.
                                        ResetHistIndex(1)
        comm                       ?Hfx Lflag9Set:
                                        SynReference(3)

                                        Lflag9Clear
```

```
: set up a normal load condition which causes the procedure call code
: to be generated.
                                          Lflag6Clear
                                          Lflag7Clear
                                          Lflag8Set
                                          SynReference(2)

                                          HistRestore
                                          LflagRestore

                                          SynReference(12);


QUALIFIED            120                  SynReference(2)
                                          SynReference(3)
                                          SynReference(12);

RAISE                121                  unimplemented;

RANGE                122                  null

: The first entries are for array size calculations.  Note the way in which
: the flags are set up for a normal address and value load situation before
: the expression links are entered.  This allows normal dynamic expressions
: to exist as lower and upper bounds.
        comm ?UfttxxxxxxxxxXXX           /*/
                                          */
                                          */
                                          */
                                          */
                                          */
                                          LflagStash
                                          Idstash
                                          ClearLflags
                                          Lflag1Set
                                          Lflag4Set
                                          Lflag8Set
                                          SynReference(3)
                                          SynReference(2)
                                          IdRestore
                                          LflagRestore/
                                          *:

        code ?UfttxxxxxxxxxXXX           /*/
                                          */
                                          */
                                          */
                                          */
                                          */
                                          SBI!
                                          INCI!
                                          MPI/
                                          *:

: all this is to give the array bounds calculations.
```

```
code ?UtffttxxxxxxxXXX          DUP1!
                                SWAP51:

comm ?UtffttxxxxxxxXXX          LflagStash
                                ClearLflags
                                Lflag1Set
                                Lflag4Set
                                Lflag8Set
                                IdStash
                                SynReference(2)
                                SynReference(3)
                                IdRestore
                                LflagRestore:
code ?UtffttxxxxxxxXXX          CHK:

comm ?UtffttxxxxxxxXXX          LflagStash
                                ClearLflags
                                Lflag1Set
                                Lflag4Set
                                Lflag8Set
                                IdStash
                                SynReference(2)
                                IdRestore
                                LflagRestore:
code ?UtffttxxxxxxxXXX          SBI!
                                MPI!
                                SWAP21!
                                ADI!
                                SWAP:

comm ?UtffttxxxxxxxXXX          CompStash
                                RefNodeNoToCompStore(12)
                                Lstore1Reset(0):
comm ?UtffttxxxxxxxXXX    ?T11f LflagStash
                                ClearLflags
                                Lflag1Set
                                Lflag4Set
                                Lflag8Set
                                IdStash
                                SynReference(3)
                                SynReference(2)
                                IdRestore
                                LflagRestore:

code ?UtffttxxxxxxxXXX    ?T11f SBI!
                                INCI!
                                MPI:

code ?UtffttxxxxxxxXXX    ?T11t POP1!
                                MPI!
                                ADI:

: this is to give the lower bound upon request (not to do with arrays).
comm ?UtfttfxxxxxxxXXX          LflagStash
                                ClearLflags
                                Lflag1Set
```

```
                                        Lflag4Set
                                        Lflag8Set
                                        SynReference(2)
                                        LflagRestore:

: this is to give the upper bound upon request (not to do with arrays).
      comm ?UtftttxxxxxxXXX             LflagStash
                                        ClearLflags
                                        Lflag1Set
                                        Lflag4Set
                                        Lflag8Set
                                        SynReference(3)
                                        LflagRestore:

: this generates a normal range check (for single words only - a
: restriction of U.C.S.D. P-code).
      comm ?UtttxxxxxxxxXXX             LflagStash
                                        ClearLflags
                                        Lflag1Set
                                        Lflag4Set
                                        Lflag8Set
                                        SynReference(2)
                                        SynReference(3)
                                        LflagRestore:
      code ?UtttxxxxxxxxXXX             CHK:

      comm ?UtfftxxxxxxXXX              CompRestore:

                                        SynReference(12);

RECORD               123                type record

: if the record is part of a variant, then simply follow syntactic
: link 2 down to the var components.  Local flag 11 is cleared while
: down syntactic link 2 so as to prevent any side effects.
      comm ?UftxxxxxxxxtxXXX            Lflag11Clear
                                        SynReference(2)
                                        Lflag11Set:

: the following table entries produce the P-code which works out the
: size of the record, assigns space for it in the dynamic heap and
: stores the size and start address in the data segment of the procedure.
      code ?UfttxxxfxxxfxXXX        ?Ll LLA  ^N/
                                    ?Li LDA  ^XD ^N/
                                    ?Lg LAO  ^N:
      code ?UfttxxxxxxxfxXXX            SLDC 0:
      comm ?UfttxxxxxxxfxXXX            IdStash
                                        SynReference(2)
                                        IdRestore:
      code ?UfttxxxtfxxfxXXX            POP1:
      code ?UfttxxxfxxxfxXXX            DUP1!
                                        NEW!
                                        STM  2:

: link for entry to components for initialization.
      comm ?UtfxxxxxxxxtXXX             Synreference(2);
```

```
RECORD-REP          124                 unimplemented;

RENAME              125                 unimplemented;

RETURN              126                 LflagStash

: set up condition for normal address and value load and clear local flag 9
: to prevent any parameter load side effects.  The return value is stored
: in data segment location 0 (and 1 for 2-word values) of the current
: procedure (or function).
                                        Lflag3Clear
                                        Lflag4Set
                                        Lflag5Clear
                                        Lflag6Clear
                                        Lflag7Clear
                                        Lflag8Set
                                        Lflag9Clear
                                        SynReference(2)
      code                             /LLA  0!
                                        STM  2/
                                        STL  0/
                                        STL  0/
                                        STL  0/
                                        STL  0/
                                        LLA  0!
                                        STM  2/
                                        */
                                        *:

                                        LflagRestore

                                        SynReference(12);

REVERSE             127                 LflagStash
: this code is arrived at from the LOOP node.  It is this node which
: turns the loop into a reverse for loop.  Note that space is created
: in the data segment of the current procedure (or function) for the
: iteration identifier.  See also the FOR node.

: descend syntactic link 2 to define the iteration identifier.
                                        ClearLflags
                                        Lflag2Set
                                        Lflag3Set
                                        SynReference(2)
                                        IdStash

: descend syntactic link 3 to generate the P-code necessary to
: give the lower and upper bounds for the loop.
                                        ClearLflags
                                        Lflag1Set
                                        Lflag3Set
                                        Lflag4Set
                                        SynReference(3)

      code                              INCI:
```

```
                                    ClearLflags
                                    Lflag1Set
                                    Lflag7Set
                                    IdRestore
                                    SynReference(2)

        code                        LBL LOOP^L1:
                                    Lstore1Inc

                                    ClearLflags
                                    Lflag1Set
                                    Lflag8Set
                                    IdRestore
                                    SynReference(2)

        code                        DECI!
                                    DUP1:

                                    ClearLflags
                                    Lflag1Set
                                    Lflag7Set
                                    IdRestore
                                    SynReference(2)

                                    ClearLflags
                                    Lflag1Set
                                    Lflag3Set
                                    Lflag4Set
                                    Lflag5Set
                                    SynReference(3)

        code                        GEQI!
                                    FJP  LOOP^L1;

SELECT              128             unimplemented;

SELECT-CLAUSE       129             unimplemented;

SELECT-CLAUSE-S     130             unimplemented;

SELECTED            131             null
        comm ?UtfxxxxxxxtxxXXX      SynReference(12):

                                    LflagStash
                                    Lflag10Clear

: convert a normal address load condition into a structured address
: load condition, stop any side effects of the value flags and
: descend syntactic link 2 followed by syntactic link 3 to
: calculate the start address of the selectd component.
        comm ?UtfftfxxxxxxxXXX      Lflag3Clear
                                    Lflag4Set
                                    Lflag5Set
                                    Lflag6Clear
                                    Lflag7Clear
```

```
                                        Lflag8Clear
                                        Lflag9Clear
                                        SynReference(2)
                                        SynReference(3)
                                        LflagRestore
                                        Lflag10Clear:

: stop any side effects of the address flags affecting value
: loads and stores.
                                        Lflag3Clear
                                        Lflag4Clear
                                        Lflag5Clear

: convert a normal value load condition into an address on stack
: value load condition.
      comm ?UtfxxxfftxxxxXXX            Lflag6Clear
                                        Lflag7Set
                                        Lflag8Set
                                        SynReference(2):

: convert a normal value store condition into an address on stack
: value store condition.
      comm ?UtfxxxftfxxxxXXX            Lflag6Set
                                        Lflag7Clear
                                        Lflag8Clear
                                        SynReference(2):

                                        LflagRestore

      comm ?UtfxxxxxxxfxxXXX            SynReference(12);

SIMPLE-REP            132               unimplemented;

SLICE                 133               SynReference(2)
                                        SynReference(3)
                                        SynReference(12);

STM-S                 134               Lflag1Set
                                        Lflag2Clear
                                        SynReference(2);

STRING-LITERAL        135               null
                                        type character
      code ?Rant                        STG  ^R*:
      code ?Ranf                        STG  ^R:
                                        SynReference(11)
                                        SynReference(12);

STUB                  136               unimplemented;

SUBPROGRAM-BODY       137               LflagStash

      comm ?UftxxxxxxtfxxXXX            ControlStash
                                        PcodeSuspend
                                        ClearLflags
                                        Lflag2Set
```

```
                                        Lflag2Set
                                        Lflag9Set
                                        SynReference(2)
                                        SynWalkSuspend
                                        SemWalkSuspend
                                        CommWdSuspend:

: so as to correctly order any initial code for data structures and any
: procedures and functions which may be defined in this subprogram, the
: following table entries exist.  They cause all the code for the
: procedures and functions to be generated first.  Only then is the
: data structure initializing code produced, followed immediatly by
: the body code for the subprogram.

: the first run through the procedure or function is set up to generate
: P-code only for the procedures and functions within this subprogram.
: Although all the identifiers are defined normally (as is necessary for
: any defined procedures and functions) NO initialization code is
: generated.  At the end of this, all the defined ids for the current
: lexical level are discarded by decrementing the level.  This is in
: preparation for the next table entries so as to stop duplicate symbol
: table definitions from occuring.
                                        ClearLflags
                                        Lflag2Set
                                        Lflag3Set
                                        Lflag10Set
                                        SynReference(2)
                                        Lflag3Clear
                                        IncLexicalLevel
                                        OffsetStash
                                        ProcStash
                                        Offset1Reset(2)
                                        Offset2Reset(2)
                                        Offset3Reset(0)
                                        Gstore4Reset(2)
                                        SynReference(3)
                                        SynReference(4)
                                        ProcRestore
                                        OffsetRestore
                                        DecLexicalLevel

: in this part, the opposite to above occurs.  Here the flags are set to
: suppress the P-code pertaining to the procedures and functions.  Only
: the initializing code for the identifiers and the body code for this
: subprogram will be produced.  Although no code is produced for the
: procedures and functions in this subprogram they are all defined
: normally  in the symbol table.  Note that this subprogram has already
: deen defined, so don't do that again, hence the absence of link 2.
                                        ClearLflags
                                        Lflag2Set
                                        Lflag9Set
                                        IncLexicalLevel
                                        OffsetStash
                                        Offset1Reset(2)
                                        Offset2Reset(2)
                                        Offset3Reset(0)
```

```
                                        Gstore4Reset(2)
                                        SynReference(3)
                                        SynReference(4)

: finally, print out the RPU instruction followed by the statistics
: for this subprogram.  This is done by setting up the correct flag
: conditions and following syntactic link 2.
                                        ClearLflags
                                        Lflag1Set
                                        Lflag6Set
                                        Lflag7Set
                                        Lflag8Set
                                        SynReference(2)
                                        OffsetRestore
                                        DecLexicalLevel

                                        LflagRestore

      comm ?UftxxxxxxtfxxXXX            ControlRestore:

                                        SynReference(12);

USED-BLTN-OP          138               null
: this node definition produces the P-code instructions for the
: various built in functions.  The problem here was to determine
: if an operator is unary or not (especially in the case of '+'
: and '-', which have both unary and binary versions).  This is
 done, in condition with the FUNCTION CALL node, by checking if
: there is one parameter attached to the function call.  If so
: then the operator is unary, otherwise not.

                                        ClearLflags
                                        HistStash

: use the values left in the history index by the FUNCTION CALL node's
: parameter scan to determine how many parameters there are.  If there
: is only one parameter, then the function is unary.  Since there are
: no links to be followed from this node and the local flags do not
: pass information to here, anything can be done with those flags
: without any unpleasant side effects.  That is why they are used here
: to indicate the type of the operator (unary or binary).
                                        ResetHistIndex(1)
      comm ?Htx                         Lflag1Set:
                                        IncHistIndex
      comm ?Hfx                         Lflag1Set:

: the following code is for the unary operators.  Note that there is no
: code for a unary '+'.  This operation does not affect the operand, so
: it is simply ignored.
      code ?UfxxxxxxxxxxXXX ?C"-"      /NGR/
                                        NGI/
                                        */
                                        */
                                        */
                                        */
                                        */
```

```
                                       */
                             ?C"NOT" /*/
                                       LNOT/
                                       LNOT/
                                       LNOT/
                                       LNOT/
                                       */
                                       */
                                       *:

: the following code is for the rest of the (non unary) built in operators.
        code ?UtxxxxxxxxxxxXXX ?C"+"   /ADR/
                                       ADI/
                                       */
                                       */
                                       */
                                       */
                                       */
                                       */
                             ?C"-"    /SBR/
                                       SBI/
                                       */
                                       */
                                       */
                                       */
                                       */
                                       */
        code                 ?C"*"    /MPR/
                                       MPI/
                                       */
                                       */
                                       */
                                       */
                                       */
                                       */
                             ?C"/"    /DVR/
                                       DVI/
                                       */
                                       */
                                       */
                                       */
                                       */
                                       */
                             ?C"**"   /PRR/
                                       PRI/
                                       */
                                       */
                                       */
                                       */
                                       */
                                       */
                             ?C"ABS" /ABR/
                                       ABI/
                                       */
                                       */
                                       */
```

```
              */
              */
              */
?C"="       /EQU   2/
              EQUI/
              EQUI/
              EQUI/
              EQUI/
              EQU   2/
              */
              */
?C"/="      /NEQ   2/
              NEQI/
              NEQI/
              NEQI/
              NEQI/
              NEQ   2/
              */
              */
?C"<="      /LEQ   2/
              LEQI/
              LEQI/
              LEQI/
              LEQI/
              LEQ   2/
              */
              */
?C"<"       /LES   2/
              LESI/
              LESI/
              LESI/
              LESI/
              LES   2/
              */
              */
?C">="      /GEQ   2/
              GEQI/
              GEQI/
              GEQI/
              GEQI/
              GEQ   2/
              */
              */
?C">"       /GRT   2/
              GRTI/
              GRTI/
              GRTI/
              GRTI/
              GRT   2/
              */
              */
?C"AND"     /*/
              LAND/
              LAND/
              LAND/
              LAND/
```

```
                                                  */
                                                  */
                                                  */
                                         ?C"OR"  /*/
                                                  LOR/
                                                  LOR/
                                                  LOR/
                                                  LOR/
                                                  */
                                                  */
                                                  *:

                                                  HistRestore;

SUBPROGRAM-DECL       139                         unimplemented;

SUBTYPE               140                         null

        comm ?Uftxxxxxxftxx XXX                   ControlStash
                                                  PcodeSuspend:

        comm ?UftxxxxxxxxxXXX                     GflagStash
                                                  LflagStash
                                                  IdStash:

        comm ?UftxxxxxxxxxXXX                     Lflag3Set:
                                                  SynReference(2)
        comm ?UfttxxxxxxxxXXX                     Lflag3Clear:

        comm ?UftxxxxxxxxxXXX                     IdRestore
                                                  LflagRestore
                                                  GflagRestore:

        comm ?UftxxxxxxftxxXXX                    ControlRestore:

                                                  SynReference(12);

SUBTYPE-ID            141                         null

: these table entries create the space in the table, make the
: identifier current by loading into the current id store.
        comm ?UfttxxxxxxxxXXX        ?Rsnt        DefineId
                                                  Offset1Assign
                                                  IdInfoGrab
                                                  LflagStash
                                                  Lflag3Clear
                                                  Lflag2Set
                                                  SemReference(2)
                                                  LflagRestore:

        comm ?UfttxxxxxxxxXXX                    /*/
                                                  */
                                                  */
                                                  */
                                                  */
                                                  */
```

```
                                                 */
                                                 Lflag7Set
                                                 Lflag8Clear
                                                 SemReference(2)
                                                 LflagRestore:

       comm ?UftxxxxtxxxxxXXX                    IdInfoGrab:

: this is used for creating a new heap structure for a pointer (access) type.
: It sets up the conditions for defining an identifier, supresses any space
: creation and heads for the type.
       comm ?UtftffxxxxxxxXXX                    IdInfoGrab
                                                 LflagStash
                                                 ClearLflags
                                                 Lflag2Set
                                                 Lflag3Set
                                                 Lflag7Set
                                                 Lflag8Set
                                                 SemReference(2)
                                                 LflagRestore:

: this is for any statement time base type accesses.
       comm ?UtftftxxxxxxxXXX                    SemReference(2)
                                                 IdInfoGrab:

: this is to allow any initialization runs to reach the appropriate data
: structure (in this case, a record).
       comm ?UtfxxxxxxxxxtXXX                    SemReference(2):

: this is to allow range check runs to pass on to the appropriate subtree.
       comm ?UtttxxxxxxxxxXXX                    SemReference(2);


SUBUNIT                 142                      unimplemented;

TASK-BODY               143                      unimplemented;

TASK-BODY-ID            144                      null;

TERMINATE               147                      unimplemented;

TIMED-ENTRY             148                      unimplemented;

TYPE                    149                      null

: any discriminants which exist are defined down syntactic link 3
                                                 SynReference(3)

       comm ?UftxxxxxxftxxXXX                    ControlStash
                                                 PcodeSuspend:

       comm ?UftxxxxxxxxxxXXX                    GflagStash
                                                 LflagStash
                                                 IdStash:

       comm ?UftxxxxxxxxxxXXX                    Lflag3Set:
```

```
                                        SynReference(2)
      comm ?Ufttxxxxxxxxxxxxx           Lflag3Clear:

      comm ?Uftxxxxxxxxxxxxx            IdRestore
                                        LflagRestore
                                        GflagRestore:

      comm ?UftxxxxxxftxxXXX            ControlRestore:

                                        SynReference(12);

TYPE-ID             150                 null

: these table entries create the space in the table and make the
: identifier current by loading into the current id store.
      comm ?UfttxxxxxxxxxXXX    ?Rsnt   DefineId
                                        Offset1Assign
                                        IdInfoGrab
                                        LflagStash
                                        Lflag3Clear
                                        Lflag2Set
                                        SemReference(2)
                                        LflagRestore:

      comm ?UfttxxxxxxxxxXXX           /*/
                                        */
                                        */
                                        */
                                        */
                                        */
                                        */
                                        Lflag7Set
                                        Lflag8Clear
                                        SemReference(2)
                                        LflagRestore:

      comm ?UftxxxxtxxxxxXXX            IdInfoGrab:

: this is used for creating a new heap structure for a pointer (access) type.
: It sets up the conditions for defining an identifier, supresses any space
: creation and heads for the type.
      comm ?UtftffxxxxxxxXXX            IdInfoGrab
                                        LflagStash
                                        ClearLflags
                                        Lflag2Set
                                        Lflag3Set
                                        Lflag7Set
                                        Lflag8Set
                                        SemReference(2)
                                        LflagRestore:

: this is for any statement time base type accesses.
      comm ?UtftftxxxxxxxXXX            SemReference(2)
                                        IdInfoGrab:

: this is to allow any initialization runs to reach the appropriate data
```

```
: structure (in this case, a record).
       comm ?UtfxxxxxxxxxtXXX          SemReference(2):

: this is to allow range check runs to pass on to the appropriate subtree.
       comm ?UtttxxxxxxxxxXXX          SemReference(2);

UNIVERSAL-FIXED       151                 type float;

UNIVERSAL-INTEGER     152                 type integer;

UNIVERSAL-REAL        153                 type float;

USE                   154                 SynReference(2)
                                          SynReference(12);

USED-BLT-ID           155                 null;

USED-CHAR             156                 null;

USED-NAME-ID          157                 null
: this node serves three purposes.  It acts as a bridge to
: the definition of user id, produces native id information and
: handles the procedure calls for procedures which are neither built
: in nor user defined (e.g. the standard procedures PUT and GET).
       comm ?UtfxxxxxxxtxxXXX          SynReference(12):

: if encountered during a definition phase, head for the defintion of
: this id (if any exists) down semantic link 2.   If no definitions
: exist, then this node must represent a native type id, so retrieve
: the native id information.
       comm ?UftxxxxtxxxxxXXX          SemReference(2):
       comm ?UftxxtxxxxxxxXXX     ?Rslf NativeInfoGrab:

       comm ?UtfxxxxxxxxxxXXX          LflagStash
                                       Lflag10Clear:

: if encountered during a parameter load, determine the type of the
: parameter (whether IN, OUT or IN-OUT), set up the appropriate load
: conditions and head for the id definitions down semantic link 2.
       comm ?UtfxxxxxxtxxxXXX     ?Htr Lflag3Clear
                                       Lflag4Clear
                                       Lflag5Set
                                       Lflag6Clear
                                       Lflag7Clear
                                       Lflag8Clear
                                       Lflag9Clear
                                       SemReference(2)
                                       LflagRestore
                                       Lflag10Clear:

       comm ?UtfxxxxxxtxxxXXX     ?Htv Lflag9Clear
                                       SemReference(2)
                                       Lflag9Set:

: if not a parameter load, then simply head for the id definitions.
       comm ?UtfxxxxxxfxxxXXX          SemReference(2):
```

```
: for a base type access if no id definitions exist for the id in
: question (hence a native type id).
        comm ?Utftftxxxxxxxx XXX    ?Rslf NativeInfoGrab:

: the following handles standard procedure calls for GET and PUT.
        comm ?UtfxxxttfxxxxXXX ?Rslf ?C"PUT" hist v
                                             IncHistIndex/
                                     ?C"GET" hist r
                                             IncHistIndex:
        code ?UtfxxxxxxfxxxXXX ?Rslf ?C"PUT" /PUTR/
                                              PUTI/
                                              PUTC/
                                              PUTB/
                                              */
                                              */
                                              */
                                              */
                                     ?C"GET" /GET2/
                                              GET1/
                                              GET1/
                                              GET1/
                                              */
                                              */
                                              */
                                              *:

        comm ?UtfxxxxxxxxxxXXX          LflagRestore:

        comm ?UtfxxxxxxtxxxXXX          IncHistIndex:
        comm ?UtfxxxxxxxfxxXXX          SynReference(12);

USED-OBJECT-ID        158               ResetIdStore

        comm ?Tllt ?UtfxxxxxxxtxxXXX    SynReference(12):

: if the object is a boolean literal, then handle it as such.
: Otherwise it is a user defined object, in which case head
: for the id defintions down semantic link 2.
        code ?Rsnf              ?C"FALSE" SLDC 0/
                                ?C"TRUE"  SLDC 1:

        comm ?Rsnf              ?C"FALSE" type boolean/
                                ?C"TRUE"  type boolean:

        comm ?Rsnt                        LflagStash
                                          ClearLflags
                                          Lflag1Set
                                          Lflag4Set
                                          Lflag8Set
                                          SemReference(2)
                                          LflagRestore:

        comm ?UtfxxxxxxtxxxXXX            IncHistIndex:
        comm ?UtfxxxxxxxfxxXXX            SynReference(12);
```

```
USED-OP             159                 SynReference(12);

VAR                 160                 null

        comm ?UftxxxxxxftxxXXX          ControlStash
                                        PcodeSuspend:

        comm ?UftxxxxxxxxxxXXX          GflagStash
                                        LflagStash
                                        IdStash:

        comm ?UftxxxxxxxxxxXXX          Lflag3Set:
                                        SynReference(2)
        comm ?UfttxxxxxxxxxXXX          Lflag3Clear:

        comm ?UftxxxxxxxxxxXXX          IdRestore
                                        LflagRestore
                                        GflagRestore:

        comm ?UftxxxxxxftxxXXX          ControlRestore:

                                        SynReference(12);

VAR-ID              161                 null

: these table entries create the space in the table, make the
: identifier current by loading in the current id store and
: update the global offset value.
             comm ?UfttxxxxxxxxxXXX ?Rsnt DefineId
                                        Offset1Assign
                                        IdInfoGrab
                                        LflagStash
                                        Lflag3Clear
                                        Lflag5Set
                                        Semreference(2)
                                        LflagRestore
                                        Offset1Update
                                        Offset3Update
                                        Gstore4Inc:
: Gstore4Incs above and below update the case variable location count
             comm ?UfttxxxxxxxxxXXX    /Gstore4Inc/
                                        */
                                        */
                                        */
                                        */
                                        Gstore4Inc/
                                        Gstore4Inc
                                        SemReference(2)/
                                        Gstore4Inc
                                        Lflag8Set
                                        SemReference(2)
                                        LflagRestore:

: The following handles the correct generation of any initialization at
: definition time which may exist.
             comm ?UfttxxxxxxxxxXXX       GflagStash:
```

```
comm ?UfttxxxxxxxxxXXX ?Rsnt Gflag2Set:
code ?UfttxxxxxxxxxXTX ?Ll /LLA  ^N/
                           */
                           */
                           */
                           */
                           LLA  ^N/
                           */
                           */
                      ?Li  ERR/
                      ?Lg /LAO  ^N/
                           */
                           */
                           */
                           */
                           LAO  ^N/
                           */
                           *:
comm ?UfttxxxxxxxxxXTX     LflagStash
                           IdStash
                           Lflag1Set
                           Lflag4Set
                           Lflag8Set
                           SemReference(3)
                           IdRestore
                           LflagRestore:
code ?UfttxxxxxxxxxXTX ?Ll /STM  2/
                           STL  ^N/
                           STL  ^N/
                           STL  ^N/
                           STL  ^N/
                           STM  2/
                           */
                           */
                      ?Li  ERR/
                      ?Lg /STM  2/
                           STR  ^N/
                           STR  ^N/
                           STR  ^N/
                           STR  ^N/
                           STM  2/
                           */
                           *:

comm ?UfttxxxxxxxxxXFX    /*/
                           */
                           */
                           */
                           */
                           */
                           */
                           LflagStash
                           ClearLflags
                           Lflag1Set
                           Lflag12Set
                           SemReference(2)
```

```
                                  LflagRestore:

            comm ?UfttxxxxxxxxxXXX    GflagRestore:

: This entry is only for when a base type attribute search reaches this
: node.  It simply causes the attributes to be loaded from the table
: for the id that this node represents.
            comm ?UftxxxxtxxxxxXXX    IdInfoGrab:


: These table entries first stash the local flags, clears them and then
: sets the local flags to indicate a base type search.  By clearing all
: the flags, all other activities with no specific reference to the base type
: search are ignored.  Then follow semantic reference 2 to find the base type.
: After returning, restore the local flags to their original condition and
: access the symbol table information for this identifier.
                                  LflagStash
            comm ?UtffftxxxxxxxXXX    ClearLflags
                                  Lflag1Set
                                  Lflag3Set
                                  Lflag5Set:
            comm ?UtfftfxxxxxxxXXX    ClearLflags
                                  Lflag1Set
                                  Lflag3Set
                                  Lflag5Set:
            comm ?UtffttxxxxxxxXXX    ClearLflags
                                  Lflag1Set
                                  Lflag3Set
                                  Lflag5Set:
            comm ?UtftftxxxxxxxXXX    SemReference(2)
                                  IdInfoGrab:
                                  LflagRestore

: having found the attributes of the identifier (above), generate the P-code
: for address loads, value loads, value stores, etc. according to the
: conditions represented by the flags.
            code ?UtfftfxxxxxxxXXX ?L1 /LLA  ^N/
                                  */
                                  */
                                  */
                                  */
                                  LLA  ^N/
                                  LLA  ^N!
                                  LDM  2/
                                  LLA  ^N!
                                  LDM  2/
                              ?Li /LDA  ^XD ^N/
                                  */
                                  */
                                  */
                                  */
                                  LDA  ^XD ^N/
                                  LDA  ^XD ^N!
                                  LDM  2/
                                  LDA  ^XD ^N!
                                  LDM  2/
```

```
                    ?Lg /LAO  ^N/
                        */
                        */
                        */
                        */
                        LAO  ^N/
                        LAO  ^N!
                        LDM  2/
                        LAO  ^N!
                        LDM  2:

code ?UtffttxxxxxxxXXX ?Ll LLA  ^N!
                        LDM  1/
                    ?Li LDA  ^XD ^N!
                        LDM  1/
                    ?Lg LAO  ^N!
                        LDM  1:

comm ?UtffttxxxxxxxXXX    /*/
                        */
                        */
                        */
                        */
                        */
                        SemReference(2)/
                        *:

code ?UtffftxxxxxxxXXX ?Ll LLA  ^N/
                    ?Li LDA  ^XD ^N/
                    ?Lg LAO  ^N:

code ?UtfxxxfftxxxxXXX ?Ll /LDM  2/
                        LDL  ^N/
                        LDL  ^N/
                        LDL  ^N/
                        LDL  ^N/
                        LDM  2/
                        */
                        */
                    ?Li /LDM  2/
                        LOD  ^XD ^N/
                        LOD  ^XD ^N/
                        LOD  ^XD ^N/
                        LOD  ^XD ^N/
                        LDM  2/
                        */
                        */
                    ?Lg /LDM  2/
                        LDO  ^N/
                        LDO  ^N/
                        LDO  ^N/
                        LDO  ^N/
                        LDM  2/
                        */
                        *:
```

```
code ?Utfxxxftfxxxx XXX ?Ll /STM  2/
                           STL  ^N/
                           STL  ^N/
                           STL  ^N/
                           STL  ^N/
                           STM  2/
                           DXFR/
                           DXFR/
                      ?Li /STM  2/
                           STR  ^XD ^N/
                           STR  ^XD ^N/
                           STR  ^XD ^N/
                           STR  ^XD ^N/
                           STM  2/
                           DXFR/
                           DXFR/
                      ?Lg /STM  2/
                           SRO  ^N/
                           SRO  ^N/
                           SRO  ^N/
                           SRO  ^N/
                           STM  2/
                           DXFR/
                           DXFR:

   code ?UtfxxxfttxxxxXXX /LDM  2/
                           LDM  1/
                           LDM  1/
                           LDM  1/
                           LDM  1/
                           LDM  2/
                           */
                           *:

: this sets up the range check load condition only for value loads.
   comm ?UtfxxxfxtxxxxXXX  LflagStash
                           ClearLflags
                           Lflag1Set
                           Lflag2Set
                           Lflag3Set:
   comm ?UtttxxxxxxxxxXXX /*/
                           SemReference(2)/
                           SemReference(2)/
                           SemReference(2)/
                           SemReference(2)/
                           */
                           */
                           *:
   comm ?UtttxxxxxxxxxXXX  LflagRestore:

   code ?UtfxxxtffxxxxXXX /STM  2/
                           STM  1/
                           STM  1/
                           STM  1/
                           STM  1/
                           STM  2/
```

```
                                            DXFR/
                                            DXFR:

                 comm ?UfttxxxxxxxxxXXX     SynReference(12);

VAR-S                 162                   SynReference(2)
                                            SynReference(3);

VARIANT               163                   SynReference(3)
                                            SynReference(12);

VARIANT-PART          164                   LflagStash
: variant components are catered for as pseudo variants.   they are
: defined as normal record components.   Note that discriminants
: are also catered for.   See the TYPE node definition and the
: DISCRMT ID node definition.   Set local flag 11 to indicate to
: the RECORD node further down that it is part of a variant part.
                                            Lflag11Set
                                            SynReference(3)
                                            LflagRestore
                                            SynReference(12);

VARIANT-S             165                   SynReference(2);

VOID                  166                   Gflag1Set;

WHILE                 167                   null
: this node is arrived at from the LOOP node.   It is this node
: which turns the loop into a while loop.   Descend syntactic link 2
: to evaluate the expression, then generate the appropriate P-code.
        code                                LBL  LOOP^L1:
                                            Lstore1Inc

                                            ClearLflags
                                            Lflag1Set
                                            Lflag4Set
                                            Lflag8Set
                                            SynReference(2)
        code                                FJP  LOOP^L1;

WITH                  168                   SynReference(2)
                                            SynReference(12);

CONTINUATION          169                   null
: if there is another CONTINUATION node attached via syntactic link 11,
: print the string without a carriage return at the end, otherwise print
: it with a carriage return.
        code ?Rant                          ^R*:
        code ?Ranf                          ^R:
                                            SynReference(11);
```

```
:***********************************************************************:
:THE TREE SPECIFICATION TABLE                                           :
:============================                                           :
:This table contains the complete DIANA specification, as given in     :
:Epstein's thesis.  It is used as a check to maintain the accuracy of  :
:usage of the nodes' components in the code template table.  If there  :
:is a discrepancy between the tables in the use of some node component :
:when that node is encountered in a tree, then an error message is     :
:generated by the code generator.                                      :
:                                                                      :
:Note the similarity between this table and that given by Epstein.     :
:There are, in fact, only three small differences.  The first          :
:difference is in the use of a semicolon to indicate the end of a node :
:definition.  The second difference is in the slightly altered format  :
:used for the attribute node definitions (node kind numbers 151, 152 and:
:153).  Finally the rep field representation here is slightly different:
:to that used in Epstein's thesis.                                     :
:***********************************************************************:

ABORT               1                                 unimplemented;

ACCEPT              2                                 unimplemented;

ACCESS              3      as-constrained             reference(2)
                           sm-size                    reference(3)
                           sm-storage-size            reference(4)
                           lx-srcpos                  reference(5)
                           cd-impl-size               value(1)
                                                      boolean(1)
                           cd-alignment               value(2)
                                                      boolean(2)
                           sm-controlled              boolean(5);

ADDRESS             4                                 unimplemented;

AGGREGATE           5      as-list                    reference(2)
                           sm-exp-type                reference(3)
                           sm-constraint              reference(4)
                           lx-srcpos                  reference(5)
                           link                       reference(12)
                           sm-value                   value(1)
                                                      boolean(1);

ALL                 6      as-name                    reference(2)
                           sm-exp-type                reference(2)
                           lx-srcpos                  reference(5)
                           link                       reference(12)
                           sm-value                   value(1)
                                                      boolean(1);

ALLOCATOR           7      as-name                    reference(2)
                           as-access-constraint       reference(3)
                           sm-exp-type                reference(4)
                           lx-srcpos                  reference(5)
                           link                       reference(12)
                           sm-value                   value(1)
```

			boolean(1);
ALTERNATIVE	8	as-choice-s	reference(2)
		as-stm-s	reference(3)
		lx-srcpos	reference(4)
		link	reference(12);
ALTERNATIVE-S	9	as-list	reference(2)
		lx-srcpos	reference(3);
AND-THEN	10	lx-srcpos	reference(2);
ARGUMENT-ID	11		unimplemented;
ARRAY	12	as-dscrt-range-s	reference(2)
		as-constrained	reference(3)
		sm-size	reference(4)
		lx-srcpos	reference(5)
		cd-impl-size	value(1)
			boolean(1)
		cd-alignment	value(2)
			boolean(2)
		sm-packing	boolean(5);
ASSIGN	13	as-name	reference(2)
		as-exp	reference(3)
		lx-srcpos	reference(4)
		link	reference(12);
ASSOC	14	as-id	reference(2)
		as-actual	reference(3)
		lx-srcpos	reference(4)
		link	reference(12);
ATTR-ID	15		unimplemented;
ATTRIBUTE	16		unimplemented;
ATTRIBUTE-CALL	17		unimplemented;
BINARY	18	as-exp1	reference(2)
		as-binary-op	reference(3)
		as-exp2	reference(4)
		sm-exp-type	reference(5)
		lx-srcpos	reference(6)
		link	reference(12)
		sm-value	value(1)
			boolean(1);
BLOCK	19	as-item-s	reference(2)
		as-stm-s	reference(3)
		as-alternative-s	reference(4)
		lx-srcpos	reference(5)
		link	reference(12);
BOX	20		unimplemented;

CASE	21	as-exp	reference(2)
		as-alternative-s	reference(3)
		lx-srcpos	reference(3)
		link	reference(12);
CHOICE-S	22	as-list-s	reference(2)
		lx-srcpos	reference(3);
CODE	23	as-exp	reference(2)
		lx-srcpos	reference(3);
COMP-ID	24	sm-obj-type	reference(2)
		sm-init-exp	reference(3)
		lx-srcpos	reference(4)
		link	reference(12)
		cd-position	value(1)
			boolean(1)
		cd-first-bit	value(2)
			boolean(2)
		cd-last-bit	value(3)
			boolean(3)
		lx-symrep	rep;
COMP-REP	25		unimplemented;
COMP-REP-S	26		unimplemented;
COMP-UNIT	27	as-pragma-a	reference(2)
		as-context	reference(3)
		as-unit-body	reference(4)
		lx-srcpos	reference(5)
		link	reference(12);
COMPILATION	28	as-list	reference(2)
		lx-srcpos	reference(3);
COND-CLAUSE	29	as-exp-void	reference(2)
		as-stm-s	reference(3)
		lx-srcpos	reference(4)
		link	reference(12);
COND-ENTRY	30		unimplemented;
CONST-ID	31	sm-obj-type	reference(2)
		sm-obj-def	reference(3)
		sm-address	reference(4)
		lx-srcpos	reference(5)
		link	reference(12)
		lx-symrep	rep;
CONSTANT	32	as-id-s	reference(2)
		as-object-def	reference(3)
		as-type-spec	reference(4)
		lx-srcpos	reference(5)
		link	reference(12);

CONSTRAINED	33	as-name	reference(2)
		as-constraint	reference(3)
		sm-type-struct	reference(4)
		sm-base-type	reference(5)
		sm-constraint	reference(6)
		lx-srcpos	reference(7)
		link	reference(12)
		cd-impl-size	value(1)
			boolean(1)
		cd-alignment	value(2)
			boolean(2);
CONTEXT	34	as-list	reference(2)
		lx-srcpos	reference(3);
CONVERSION	35	as-name	reference(2)
		as-exp	reference(3)
		sm-exp-type	reference(4)
		lx-srcpos	reference(5)
		link	reference(12)
		sm-value	value(1)
			boolean(1);
DECL-REP-S	36		unimplemented;
DECL-S	37	as-list	reference(2)
		lx-srcpos	reference(3);
DEF-CHAR	38	sm-obj-type	reference(2)
		lx-srcpos	reference(3)
		sm-pos	value(1)
			boolean(1)
		sm-rep	value(2)
			boolean(2)
		lx-symrep	rep;
DEF-OP	39	sm-spec	reference(2)
		sm-body	reference(3)
		sm-location	reference(4)
		lx-srcpos	reference(5)
		lx-symrep	rep;
DELAY	40		unimplemented;
DERIVED	41	as-constrained	reference(2)
		sm-size	reference(3)
		sm-storage-size	reference(4)
		lx-srcpos	reference(5)
		cd-impl-size	value(1)
			boolean(1)
		cd-alignment	value(2)
			boolean(2)
		sm-actual-delta	value(3)
			boolean(3)
		sm-packing	boolean(5)

		sm-controlled	boolean(6);
DISCRIMANT-AGGREGATE	42	as-list	reference(2)
		lx-srcpos	reference(3);
DISCRMT-ID	43	sm-obj-type	reference(2)
		sm-init-exp	reference(3)
		lx-srcpos	reference(4)
		cd-position	value(1)
			boolean(1)
		cd-first-bit	value(2)
			boolean(2)
		cd-last-bit	value(3)
			boolean(3)
		lx-symrep	rep;
DSCRT-RANGE-S	44	as-list	reference(2)
		lx-srcpos	reference(3);
ENTRY	45		unimplemented;
ENTRY-CALL	46		unimplemented;
ENTRY-ID	47	sm-spec	reference(2)
		sm-address	reference(3)
		lx-srcpos	reference(4)
		lx-symrep	rep;
ENUM-ID	48	sm-obj-type	reference(2)
		lx-srcpos	reference(3)
		sm-pos	value(1)
			boolean(1)
		sm-rep	value(2)
			boolean(2)
		lx-symrep	rep
		link	reference(12);
ENUM-LITERAL-S	49	as-list	reference(2)
		sm-size	reference(3)
		lx-src-pos	reference(4)
		cd-impl-size	value(1)
			boolean(1)
		cd-alignment	value(2)
			boolean(2);
EXCEPTION	50		unimplemented;
EXCEPTION-ID	51	sm-exception-def	reference(2)
		lx-srcpos	reference(3)
		lx-symrep	rep;
EXIT	52	as-exp-void	reference(2)
		as-name-void	reference(3)
		sm-stm	reference(4)
		lx-srcpos	reference(5);

EXP-S	53	as-list	reference(2)
		lx-srcpos	reference(3);
FIXED	54	as-exp	reference(2)
		as-range-void	reference(3)
		sm-size	reference(4)
		lx-srcpos	reference(5)
		cd-impl-size	value(1)
			boolean(1)
		cd-alignment	value(2)
			boolean(2)
		sm-actual-delta	value(3)
			boolean(3)
		sm-bits	value(4)
			boolean(4);
FLOAT	55	as-exp	reference(2)
		as-range-void	reference(3)
		sm-type-struct	reference(4)
		sm-size	reference(5)
		lx-srcpos	reference(6)
		cd-impl-size	value(1)
			boolean(1)
		cd-alignment	value(2)
			boolean(2);
FOR	56	as-id	reference(2)
		as-dscrt-range	reference(3)
		lx-srcpos	reference(4);
FORMAL-DSCRT	57		unimplemented;
FORMAL-FIXED	58		unimplemented;
FORMAL-FLOAT	59		unimplemented;
FORMAL-INTEGER	60		unimplemented;
FUNCTION	61	as-param-s	reference(2)
		as-constrained-void	reference(3)
		lx-srcpos	reference(4);
FUNCTION-CALL	62	as-name	reference(2)
		as-param-assoc-s	reference(3)
		sm-exp-type	reference(5)
		lx-srcpos	reference(6)
		link	reference(12)
		sm-value	value(1)
			boolean(1);
FUNCTION-ID	63	sm-spec	reference(2)
		sm-body	reference(3)
		sm-location	reference(4)
		lx-srcpos	reference(5)
		lx-symrep	rep;

GENERIC	64		unimplemented;
GENERIC-ASSOC-S	65		unimplemented;
GENERIC-ID	66	sm-generic-param-s	reference(2)
		sm-spec	reference(3)
		sm-body	reference(4)
		lx-srcpos	reference(5)
		lx-symrep	rep;
GENERIC-OP	67		unimplemented;
GENERIC-PARAM-S	68		unimplemented;
GOTO	69	as-name	reference(2)
		lx-srcpos	reference(3)
		link	reference(12);
ID-S	70	as-list	reference(2)
		lx-srcpos	reference(3);
IF	71	as-list	reference(2)
		lx-srcpos	reference(3)
		link	reference(12);
IN	72	as-id-s	reference(2)
		as-type-spec	reference(3)
		as-exp-void	reference(4)
		lx-srcpos	reference(5)
		link	reference(12);
IN-ID	73	sm-obj-type	reference(2)
		sm-init-exp	reference(3)
		lx-srcpos	reference(5)
		link	reference(12)
		lx-symrep	rep;
IN-OP	74	lx-srcpos	reference(2);
IN-OUT	75	as-id-s	reference(2)
		as-type-spec	reference(3)
		as-exp-void	reference(4)
		lx-srcpos	reference(5)
		link	reference(12);
IN-OUT-ID	76	sm-obj-type	reference(2)
		lx-srcpos	reference(3)
		link	reference(12)
		lx-symrep	rep;
INDEX	77	as-name	reference(2)
		lx-srcpos	reference(3)
		link	reference(12);
INDEXED	78	as-name	reference(2)
		as-exp-s	reference(3)

		sm-exp-type	reference(4)
		link	reference(12)
		sm-value	value(1)
			boolean(1);
INNER-RECORD	79	as-list	reference(2)
		lx-srcpos	reference(3);
INSTANTIATION	80		unimplemented;
INTEGER	81	as-range	reference(2)
		sm-type-struct	reference(3)
		sm-size	reference(4)
		lx-srcpos	reference(5)
		cd-impl-size	value(1)
			boolean(1)
		cd-alignment	value(2)
			boolean(2);
ITEM-S	82	as-list	reference(2)
		lx-srcpos	reference(3);
ITERATION-ID	83	sm-obj-type	reference(2)
		lx-srcpos	reference(3)
		lx-symrep	rep;
L-PRIVATE	84		unimplemented;
L-PRIVATE-TYPE-ID	85		unimplemented;
LABEL-ID	86	sm-stm	reference(2)
		lx-srcpos	reference(3)
		link	reference(12)
		lx-symrep	rep;
LABELLED	87	as-id-s	reference(2)
		as-stm	reference(3)
		lx-srcpos	reference(4)
		link	reference(12);
LOOP	88	as-iteration	reference(2)
		as-stm-s	reference(3)
		lx-srcpos	reference(4)
		link	reference(12);
MEMBERSHIP	89	as-exp	reference(2)
		as-membership-op	reference(3)
		as-type-range	reference(4)
		sm-exp-type	reference(5)
		lx-srcpos	reference(6)
		link	reference(12)
		sm-value	value(1)
			boolean(1);
NAME-S	90	as-list	reference(2)
		lx-srcpos	reference(3);

NAMED	91	as-choice-s	reference(2)
		as-exp	reference(3)
		lx-srcpos	reference(4)
		link	reference(12);
NAMED-STM	92	as-id	reference(3)
		as-stm	reference(2)
		lx-srcpos	reference(4)
		link	reference(12);
NO-DEFAULT	93		unimplemented;
NOT-IN	94	lx-srcpos	reference(2);
NULL-ACCESS	95	sm-exp-type	reference(2)
		lx-srcpos	reference(3)
		sm-value	value(1)
			boolean(1)
		link	reference(12);
NULL-COMP	96	lx-srcpos	reference(2);
NULL-STM	97	lx-srcpos	reference(2)
		link	reference(12);
NUMBER	98	as-id-s	reference(2)
		as-exp	reference(3)
		lx-srcpos	reference(5)
		link	reference(12);
NUMBER-ID	99	sm-obj-type	reference(2)
		sm-init-exp	reference(3)
		lx-srcpos	reference(4)
		link	reference(12)
		lx-symrep	rep;
NUMERIC-LITERAL	100	sm-exp-type	reference(2)
		lx-srcpos	reference(3)
		link	reference(12)
		sm-value	value(1)
			boolean(1)
		lx-numrep	rep;
OR-ELSE	101	lx-srcpos	reference(2);
OTHERS	102	lx-srcpos	reference(2)
		link	reference(12);
OUT	103	as-id-s	reference(2)
		as-type-spec	reference(3)
		as-exp-void	reference(4)
		lx-srcpos	reference(5)
		link	reference(12);
OUT-ID	104	sm-obj-type	reference(2)

		lx-srcpos	reference(3)
		link	reference(12)
		lx-symrep	rep;
PACKAGE-BODY	105		unimplemented;
PACKAGE-DECL	106		unimplemented;
PACKAGE-ID	107	sm-spec	reference(2)
		sm-body	reference(3)
		sm-address	reference(4)
		lx-srcpos	reference(5)
		lx-symrep	rep;
PACKAGE-SPEC	108		unimplemented;
PARAM-ASSOC-S	109	as-list	reference(2)
		lx-srcpos	reference(3);
PARAM-S	110	as-list	reference(2)
		lx-srcpos	reference(3);
PARENTHESISED	111	as-exp	reference(2)
		sm-exp-type	reference(3)
		lx-srcpos	reference(4)
		link	reference(12)
		sm-value	value(1)
			boolean(1);
PRAGMA	112	as-id	reference(2)
		as-param-assoc-s	reference(3)
		lx-srcpos	reference(4)
		link	reference(12);
PRAGMA-ID	113	as-list	reference(2)
		lx-srcpos	reference(3);
PRAGMA-S	114	as-list	reference(2)
		lx-srcpos	reference(3);
PRIVATE	115		unimplemented;
PRIVATE-TYPE-ID	116		unimplemented;
PROC-ID	117	sm-spec	reference(2)
		sm-body	reference(3)
		sm-location	reference(4)
		lx-srcpos	reference(5)
		lx-symrep	rep;
PROCEDURE	118	as-param-s	reference(2)
		lx-srcpos	reference(3);
PROCEDURE-CALL	119	as-name	reference(2)
		as-param-assoc-s	reference(3)
		lx-srcpos	reference(4)

		link	reference(12);
QUALIFIED	120	as-name	reference(2)
		as-exp	reference(3)
		sm-exp-type	reference(4)
		lx-srcpos	reference(5)
		link	reference(12)
		sm-value	value(1)
			boolean(1);
RAISE	121		unimplemented;
RANGE	122	as-exp1	reference(2)
		as-exp2	reference(3)
		sm-base-type	reference(4)
		lx-srcpos	reference(5)
		link	reference(12);
RECORD	123	as-list	reference(2)
		sm-discriminants	reference(3)
		sm-size	reference(4)
		lx-srcpos	reference(5)
		cd-impl-size	value(1)
			boolean(1)
		cd-alignment	value(2)
			boolean(2)
		sm-packing	boolean(5);
RECORD-REP	124		unimplemented;
RENAME	125		unimplemented;
RETURN	126	as-exp-void	reference(2)
		lx-srcpos	reference(3)
		link	reference(12);
REVERSE	127	as-id	reference(2)
		as-dscrt-range	reference(3)
		lx-srcpos	reference(4);
SELECT	128		unimplemented;
SELECT-CLAUSE	129		unimplemented;
SELECT-CLAUSE-S	130		unimplemented;
SELECTED	131	as-name	reference(2)
		as-designator	reference(3)
		sm-exp-type	reference(4)
		lx-srcpos	reference(5)
		link	reference(12)
		sm-value	value(1)
			boolean(1);
SIMPLE-REP	132		unimplemented;

SLICE	133	as-name	reference(2)
		as-dscrt-range	reference(3)
		sm-exp-type	reference(4)
		sm-constraint	reference(5)
		lx-srcpos	reference(6)
		link	reference(12)
		sm-value	value(1)
			boolean(1);
STM-S	134	as-list	reference(2)
		lx-srcpos	reference(3);
STRING-LITERAL	135	sm-exp-type	reference(2)
		sm-constraint	reference(3)
		lx-srcpos	reference(4)
		link	reference(11)
		link	reference(12)
		sm-value	value(1)
			boolean(1)
		lx-symrep	rep;
STUB	136		unimplemented;
SUBPROGRAM-BODY	137	as-designator	reference(2)
		as-header	reference(3)
		as-block-stub	reference(4)
		lx-srcpos	reference(5)
		link	reference(12);
USED-BLTN-OP	138	sm-operator	reference(2)
		lx-srcpos	reference(3)
		lx-symrep	rep;
SUBPROGRAM-DECL	139		unimplemented;
SUBTYPE	140	as-id	reference(2)
		as-constrained	reference(3)
		lx-srcpos	reference(4)
		link	reference(12);
SUBTYPE-ID	141	sm-type-spec	reference(2)
		lx-srcpos	reference(3)
		lx-symrep	rep;
SUBUNIT	142		unimplemented;
TASK-BODY	143		unimplemented;
TASK-BODY-ID	144	sm-type-spec	reference(2)
		sm-body	reference(3)
		lx-srcpos	reference(4)
		lx-symrep	rep;
TASK-DECL	145		unimplemented;
TASK-SPEC	146		unimplemented;

TERMINATE	147		unimplemented;
TIMED-ENTRY	148		unimplemented;
TYPE	149	as-id	reference(2)
		as-var-s	reference(3)
		as-type-spec	reference(4)
		lx-srcpos	reference(5)
		link	reference(12);
TYPE-ID	150	sm-type-spec	reference(2)
		lx-srcpos	reference(3)
		lx-symrep	rep;
UNIVERSAL-FIXED	151	denoted	attribute(2);
UNIVERSAL-INTEGER	152	denoted	attribute(4);
UNIVERSAL-REAL	153	denoted	attribute(3);
USE	154	as-list	reference(2)
		lx-srcpos	reference(3)
		link	reference(12);
USED-BLT-ID	155	sm-operator	reference(2)
		lx-srcpos	reference(3)
		lx-symrep	rep;
USED-CHAR	156	sm-defn	reference(2)
		sm-exp-type	reference(3)
		lx-srcpos	reference(4)
		sm-value	value(1)
			boolean(1);
USED-NAME-ID	157	sm-defn	reference(2)
		lx-srcpos	reference(3)
		link	reference(12)
		lx-symrep	rep;
USED-OBJECT-ID	158	sm-defn	reference(2)
		sm-exp-type	reference(3)
		lx-srcpos	reference(4)
		link	reference(12)
		sm-value	value(1)
			boolean(1)
		lx-symrep	rep;
USED-OP	159	sm-defn	reference(2)
		link	reference(12)
		lx-symrep	rep;
VAR	160	as-id-s	reference(2)
		as-object-def	reference(3)
		as-type-spec	reference(4)
		lx-srcpos	reference(5)

		link	reference(12);
VAR-ID	161	sm-obj-type	reference(2)
		sm-obj-def	reference(3)
		sm-address	reference(4)
		lx-srcpos	reference(5)
		link	reference(12)
		lx-symrep	rep;
VAR-S	162	as-list	reference(2)
		lx-srcpos	reference(3);
VARIANT	163	as-choice-s	reference(2)
		as-record	reference(3)
		lx-srcpos	reference(4);
VARIANT-PART	164	as-name	reference(2)
		as-variant-s	reference(3)
		lx-srcpos	reference(4)
		link	reference(12);
VARIANT-S	165	as-list	reference(2)
		lx-srcpos	reference(3);
VOID	166		unimplemented;
WHILE	167	as-exp	reference(2)
		lx-srcpos	reference(3);
WITH	168	as-list	reference(2)
		lx-srcpos	reference(3)
		link	reference(12);
CONTINUATION	169	link	reference(11)
		lx-symrep	rep;

APPENDIX E

THE EXAMPLES

This appendix contains a number of examples which demonstrate the operation of the back end processor system. The first four examples are drawn from those in Epstein's thesis which generate DIANA trees. The fifth example is derived from a "hand scanned" demonstration included in Epstein's thesis. These five examples are given here to show that the back end processor can operate with the U.C.T. front end processor (with the appropriate tables).

The above five Epstein examples do not demonstrate all the facilities of the system. Also, because of the complexity of the Epstein examples, many of the construct conversions (from Ada to P-code) are difficult to examine. So a number of additional examples are supplied, each focusing on a particular aspect of the Ada - DIANA - P-code conversion.

All the examples are given here with their original Ada listings, as input to the U.C.T. front end processor. The resulting DIANA tree listings are given with all the additional examples as these listings are quite short. However, DIANA tree listings for the first four Epstein examples are not given. These listings are very long and would add needlessly to the bulk of this thesis because they are already available in Epstein's thesis.

```
--EXAMPLE 1
FUNCTION FACT(I : INTEGER) RETURN INTEGER IS
BEGIN
 IF I = 1 THEN
  RETURN 1;
 ELSE
  RETURN I * FACT(I - 1);
 END IF;
END FACT;
```

```
LDL  2
LDCI 1
EQUI
FJP  IF0
LDCI 1
STL  0
UJP  IF1
LBL  IF0
LDL  2
LDL  2
LDCI 1
SBI
SLDC 0
SLDC 0
CGP  1
MPI
STL  0
LBL  IF1
RPU  1

FUNCTION NUMBER = 1
LEVEL OF DEFN.  = 1
PARAMETERS      = 3
DATA            = 0
```

```
  --EXAMPLE 2
  PROCEDURE TSORT IS --TOPOLOGICAL SORT
  TYPE LEADER;
  TYPE TRAILER;
  TYPE LREF IS ACCESS LEADER;
  TYPE TREF IS ACCESS TRAILER;
  TYPE LEADER IS
   RECORD
     KEY      : INTEGER   := 0;
     COUNT    : INTEGER   := 0;
     TRAIL    : TREF      := NULL;
     NXT      : LREF      := NULL;
   END RECORD;
  TYPE TRAILER IS
   RECORD
     ID       : LREF;
     NXT      : TREF;
   END RECORD;
  HEAD, TAIL, P, Q : LREF;
  T                : TREF;
  Z                : INTEGER;
  X,Y              : INTEGER;
FUNCTION L(W:INTEGER) RETURN LREF IS
  H : LREF;
  BEGIN
    H := HEAD;
    TAIL.KEY := W;
    WHILE TAIL.KEY /= W LOOP
      H := H.NXT;
    END LOOP;
    IF H = TAIL
    THEN
       TAIL := NEW LEADER;
       Z := Z + 1;
       H.COUNT := 0;
       H.TRAIL := NULL;
       H.NXT   := TAIL;
    END  IF;
RETURN H;
END L;
BEGIN
  HEAD := NEW LEADER;
  TAIL := HEAD;
  Z := 0;
  GET(X);
  WHILE X /= 0 LOOP
    GET(Y);
    PUT(X); PUT(Y);
    P := L(X);
    Q := L(Y);
    T := NEW TRAILER(NULL,NULL);
    T.ID := Q;
    T.NXT := P.TRAIL;
    P.TRAIL := T;
    Q.COUNT := Q.COUNT + 1;
    GET(X);
```

```
  END LOOP;
  P := HEAD;
  HEAD := NULL;
   WHILE P /= TAIL LOOP
     Q := P;
     P := P.NXT;
     IF Q.COUNT = 0 THEN
      Q.NXT := HEAD;
      HEAD := Q;
     END IF;
   END LOOP;
   Q := HEAD;
 OUTER : WHILE Q /= NULL LOOP
           PUT(Q.KEY);
           Z:= Z - 1;
           T := Q.TRAIL;
           Q := Q.NXT;
    INNER : WHILE T /= NULL LOOP
              P := T.ID;
              P.COUNT:= P.COUNT - 1;
              IF P.COUNT = 0 THEN
                 P.NXT := Q;
                 Q := P;
              END IF;
              T := T.NXT;
           END LOOP INNER;
        END LOOP OUTER;
   IF Z = 0 THEN
   PUT("THIS SET IS NOT PARTIALLY ORDERED.");
   END IF;
END TSORT;
```

```
LLA  3
LDA  1 12
LDM  2
STM  2
LDA  1 14
LDM  1
LOD  1 2
ADI
LDL  2
STM  1
LBL  LOOP0
LDA  1 14
LDM  1
LOD  1 2
ADI
LDM  1
LDL  2
NEQI
FJP  LOOP1
LLA  3
LLA  3
LDM  1
LOD  1 6
ADI
LDM  2
STM  2
UJP  LOOP0
LBL  LOOP1
LLA  3
LDM  2
LDA  1 14
LDM  2
EQU  2
FJP  IF0
LDA  1 14
LDCI 0
LDCI 0
NULL
NULL
SLDC 0
SLDC 1
ADI
SLDC 1
ADI
SLDC 2
ADI
SLDC 2
ADI
DUP1
NEW
INIT
STM  2
LOD  1 22
LDCI 1
ADI
STR  1 22
```

```
LLA   3
LDM   1
LOD   1  3
ADI
LDCI  0
STM   1
LLA   3
LDM   1
LOD   1  4
ADI
NULL
STM   2
LLA   3
LDM   1
LOD   1  6
ADI
LDA   1  14
LDM   2
STM   2
LBL   IF0
LLA   3
LDM   2
LLA   0
STM   2
RPU   2

FUNCTION NUMBER = 2
LEVEL OF DEFN.  = 2
PARAMETERS      = 3
DATA            = 2


SLDC 0
DUP1
STL   2
SLDC 1
ADI
DUP1
STL   3
SLDC 1
ADI
DUP1
STL   4
SLDC 2
ADI
DUP1
STL   6
SLDC 2
ADI
POP1
SLDC 0
DUP1
STL   8
SLDC 2
ADI
```

```
DUP1
STL  10
SLDC 2
ADI
POP1
LLA  12
LDCI 0
LDCI 0
NULL
NULL
SLDC 0
SLDC 1
ADI
SLDC 1
ADI
SLDC 2
ADI
SLDC 2
ADI
DUP1
NEW
INIT
STM  2
LLA  14
LLA  12
LDM  2
STM  2
LDCI 0
STL  22
LLA  23
GET1
LBL  LOOP2
LDL  23
LDCI 0
NEQI
FJP  LOOP3
LLA  24
GET1
LDL  23
PUTI
LDL  24
PUTI
LLA  16
LDL  23
SLDC 0
SLDC 0
CLP  2
STM  2
LLA  18
LDL  24
SLDC 0
SLDC 0
CLP  2
STM  2
LLA  20
NULL
```

```
NULL
SLDC 0
SLDC 2
ADI
SLDC 2
ADI
DUP1
NEW
INIT
STM  2
LLA  20
LDM  1
LDL  8
ADI
LLA  18
LDM  2
STM  2
LLA  20
LDM  1
LDL  10
ADI
LLA  16
LDM  1
LDL  4
ADI
LDM  2
STM  2
LLA  16
LDM  1
LDL  4
ADI
LLA  20
LDM  2
STM  2
LLA  18
LDM  1
LDL  3
ADI
LLA  18
LDM  1
LDL  3
ADI
LDM  1
LDCI 1
ADI
STM  1
LLA  23
GET1
UJP  LOOP2
LBL  LOOP3
LLA  16
LLA  12
LDM  2
STM  2
LLA  12
NULL
```

```
STM  2
LBL  LOOP4
LLA  16
LDM  2
LLA  14
LDM  2
NEQ  2
FJP  LOOP5
LLA  18
LLA  16
LDM  2
STM  2
LLA  16
LLA  16
LDM  1
LDL  6
ADI
LDM  2
STM  2
LLA  18
LDM  1
LDL  3
ADI
LDM  1
LDCI 0
EQUI
FJP  IF2
LLA  18
LDM  1
LDL  6
ADI
LLA  12
LDM  2
STM  2
LLA  12
LLA  18
LDM  2
STM  2
LBL  IF2
UJP  LOOP4
LBL  LOOP5
LLA  18
LLA  12
LDM  2
STM  2
LBL  LOOP6
LLA  18
LDM  2
NULL
NEQ  2
FJP  LOOP7
LLA  18
LDM  1
LDL  2
ADI
LDM  1
```

```
PUTI
LDL  22
LDCI 1
SBI
STL  22
LLA  20
LLA  18
LDM  1
LDL  4
ADI
LDM  2
STM  2
LLA  18
LLA  18
LDM  1
LDL  6
ADI
LDM  2
STM  2
LBL  LOOP8
LLA  20
LDM  2
NULL
NEQ  2
FJP  LOOP9
LLA  16
LLA  20
LDM  1
LDL  8
ADI
LDM  2
STM  2
LLA  16
LDM  1
LDL  3
ADI
LLA  16
LDM  1
LDL  3
ADI
LDM  1
LDCI 1
SBI
STM  1
LLA  16
LDM  1
LDL  3
ADI
LDM  1
LDCI 0
EQUI
FJP  IF4
LLA  16
LDM  1
LDL  6
ADI
```

```
LLA  18
LDM  2
STM  2
LLA  18
LLA  16
LDM  2
STM  2
LBL  IF4
LLA  20
LLA  20
LDM  1
LDL  10
ADI
LDM  2
STM  2
UJP  LOOP8
LBL  LOOP9
LBL  INNER
UJP  LOOP6
LBL  LOOP7
LBL  OUTER
LDL  22
LDCI 0
EQUI
FJP  IF6
STG  "THIS SET IS NOT PARTIALLY ORDERED."
PUTC
LBL  IF6
RPU  0

PROCEDURE NUMBER = 1
LEVEL OF DEFN.   = 1
PARAMETERS       = 2
DATA             = 23
```

```
--EXAMPLE 3
PROCEDURE INSERT IS  --LINEAR LIST INSERTION.
TYPE NODE;
TYPE REF IS ACCESS NODE;
TYPE NODE IS
 RECORD
  KEY : INTEGER;
  COUNT : INTEGER;
  NXT   : REF;
 END RECORD;
K : INTEGER;
ROOT : REF := NEW NODE(0,0,NULL);
PROCEDURE SEARCH(X : INTEGER;
                 BASE : IN OUT REF) IS
 W   : REF;
 B : BOOLEAN;
 BEGIN
  W   := BASE;
  B   := TRUE;
  WHILE W /= NULL AND B LOOP
   IF W.KEY = X THEN B := FALSE;
   ELSE W   := W.NXT;
   END IF;
  END LOOP;
  IF B THEN --NEW ENTRY
   W :=  BASE;
   BASE := NEW NODE(0,0,NULL);
   BASE.KEY :=  X;
   BASE.COUNT:= 1;
  ELSE
   W.COUNT := W.COUNT  + 1;
  END IF;
END SEARCH;
PROCEDURE PRINTLIST(V:REF) IS
 BEGIN
  WHILE V /= NULL LOOP
   PUT(V.KEY);
   PUT(V.COUNT);
   V := V.NXT;
  END LOOP;
END PRINTLIST;
BEGIN --MAIN PROGRAM
 GET(K);
 WHILE K /= 0 LOOP
  SEARCH(K,ROOT);
  GET(K);
 END LOOP;
 PRINTLIST(ROOT);
END INSERT;
```

```
LLA  4
LDL  3
LDM  2
STM  2
SLDC 1
STL  6
LBL  LOOP0
LLA  4
LDM  2
NULL
NEQ  2
LDL  6
LAND
FJP  LOOP1
LLA  4
LDM  1
LOD  1 2
ADI
LDM  1
LDL  2
EQUI
FJP  IF0
SLDC 0
STL  6
UJP  IF1
LBL  IF0
LLA  4
LLA  4
LDM  1
LOD  1 4
ADI
LDM  2
STM  2
LBL  IF1
UJP  LOOP0
LBL  LOOP1
LDL  6
FJP  IF4
LLA  4
LDL  3
LDM  2
STM  2
LDL  3
LDCI 0
LDCI 0
NULL
SLDC 0
SLDC 1
ADI
SLDC 1
ADI
SLDC 2
ADI
DUP1
NEW
INIT
```

```
STM  2
LDL  3
LDM  1
LOD  1 2
ADI
LDL  2
STM  1
LDL  3
LDM  1
LOD  1 3
ADI
LDCI 1
STM  1
UJP  IF5
LBL  IF4
LLA  4
LDM  1
LOD  1 3
ADI
LLA  4
LDM  1
LOD  1 3
ADI
LDM  1
LDCI 1
ADI
STM  1
LBL  IF5
RPU  0

PROCEDURE NUMBER = 2
LEVEL OF DEFN.   = 2
PARAMETERS       = 4
DATA             = 3


LBL  LOOP2
LLA  2
LDM  2
NULL
NEQ  2
FJP  LOOP3
LLA  2
LDM  1
LOD  1 2
ADI
LDM  1
PUTI
LLA  2
LDM  1
LOD  1 3
ADI
LDM  1
PUTI
LLA  2
```

```
LLA  2
LDM  1
LOD  1 4
ADI
LDM  2
STM  2
UJP  LOOP2
LBL  LOOP3
RPU  0
```

```
PROCEDURE NUMBER = 3
LEVEL OF DEFN.   = 2
PARAMETERS       = 4
DATA             = 0
```

```
SLDC 0
DUP1
STL  2
SLDC 1
ADI
DUP1
STL  3
SLDC 1
ADI
DUP1
STL  4
SLDC 2
ADI
POP1
LLA  7
LDCI 0
LDCI 0
NULL
SLDC 0
SLDC 1
ADI
SLDC 1
ADI
SLDC 2
ADI
DUP1
NEW
INIT
STM  2
LLA  6
GET1
LBL  LOOP4
LDL  6
LDCI 0
NEQI
FJP  LOOP5
LDL  6
LLA  7
CLP  2
```

```
LLA  6
GET1
UJP  LOOP4
LBL  LOOP5
LLA  7
LDM  2
CLP  3
RPU  0

PROCEDURE NUMBER = 1
LEVEL OF DEFN.   = 1
PARAMETERS       = 2
DATA             = 7
```

```
--EXAMPLE 4
PROCEDURE V IS
TYPE INTEGER IS RANGE 1..50;
TYPE COLOUR IS (RED,ORANGE,YELLOW,GREEN);
ROBOT : COLOUR;
VAR2, VAR1 : INTEGER;
BEGIN
<<LAB1>> VAR1 := 2;
<<LAB2>><<LAB3>> VAR2  := VAR1;
VAR2 := 10;
NULL;
GOTO LAB2;
GOTO LAB1;
GOTO LAB3;
<<LAB4>> NAMED1 :  LOOP
         CASE VAR1 IS
         WHEN 1|2 =>
                     DECLARE
                     VAR4 : FLOAT;
                     BEGIN
                       VAR4 := 4.5 * 3.3;
                       VAR4 := VAR4 ** 2;
                     END;
         WHEN 3   => LOOP
                       VAR1 := VAR1 + 1;
                       EXIT WHEN VAR1 = 10;
                     END LOOP;
         WHEN OTHERS => NULL;
         END CASE;
         VAR1 := VAR2;
         VAR2 := 20;
         EXIT NAMED1;
         END LOOP NAMED1;
NAMED2 : FOR I IN COLOUR LOOP
         VAR1 := VAR1 + 1;
         VAR2 := VAR2 + VAR1;
         EXIT NAMED2 WHEN VAR1 = 3;
         END LOOP;
NAMED3 : DECLARE
         VAR3 : FLOAT;
         BEGIN
           VAR3 := 3.3;
           IF VAR1 = VAR2
           THEN
              VAR3 := 6.6;
           END IF;
         END;
WHILE VAR1 < 10 LOOP
                IF VAR2 = 10
                THEN
                   VAR1 := 5;
                ELSIF VAR2 = 9
                THEN
                   VAR1 := 7;
                ELSIF VAR2 = 7
                THEN
```

```
                VAR1 := 5;
             ELSE
                VAR1 := 3;
             END IF;
             EXIT;
             END LOOP;
FOR J IN REVERSE GREEN .. RED LOOP
                              ROBOT := J;
                              RETURN;
                              RETURN VAR1;
                              NULL;
                              END LOOP;
ROBOT := GREEN;
END;
```

```
LBL  LAB1
LDCI 2
STL  4
LBL  LAB2
LBL  LAB3
LDL  4
LDCI 1
LDCI 50
CHK
STL  3
LDCI 10
STL  3
UJP  LAB2
UJP  LAB1
UJP  LAB3
LBL  LAB4
LBL  LOOP0
LDL  4
LDCI 1
LDCI 50
CHK
STL  5
LDCI 1
LDCI 2
LDCI 3
LBL  ACASE0
SWAP
LDL  5
EQUI
TJP  BCASE0
DECI
DUP1
SLDC 1
EQUI
FJP  ACASE0
POP1
UJP  CCASE0
LBL  BCASE0
LLA  6
LDC  2  4.5000E+00
LDC  2  3.3000E+00
MPR
STM  2
LLA  6
LLA  6
LDM  2
LDCI 2
PRR
STM  2
UJP  DCASE0
LBL  CCASE0
LDCI 3
LDCI 2
LBL  ACASE1
SWAP
LDL  5
```

```
EQUI
TJP   BCASE1
DECI
DUP1
SLDC 1
EQUI
FJP   ACASE1
POP1
UJP   CCASE1
LBL   BCASE1
LBL   LOOP2
LDL   4
LDCI  1
LDCI  50
CHK
LDCI  1
ADI
STL   4
LDL   4
LDCI  1
LDCI  50
CHK
LDCI  10
EQUI
TJP   LOOP3
UJP   LOOP2
LBL   LOOP3
UJP   DCASE1
LBL   CCASE1
UJP   DCASE2
LBL   CCASE2
LBL   DCASE2
LBL   DCASE1
LBL   DCASE0
LDL   3
LDCI  1
LDCI  50
CHK
STL   4
LDCI  20
STL   3
UJP   NAMED1
UJP   LOOP0
LBL   LOOP1
LBL   NAMED1
LDCI  0
DECI
STL   8
LBL LOOP4
LDL   8
INCI
DUP1
STL   8
LDCI  3
LEQI
FJP   LOOP5
```

```
LDL  4
LDCI 1
LDCI 50
CHK
LDCI 1
ADI
STL  4
LDL  3
LDCI 1
LDCI 50
CHK
LDL  4
LDCI 1
LDCI 50
CHK
ADI
STL  3
LDL  4
LDCI 1
LDCI 50
CHK
LDCI 3
EQUI
TJP  NAMED2
UJP  LOOP4
LBL  LOOP5
LBL  NAMED2
LLA  9
LDC  2  3.3000E+00
STM  2
LDL  4
LDCI 1
LDCI 50
CHK
LDL  3
LDCI 1
LDCI 50
CHK
EQUI
FJP  IF0
LLA  9
LDC  2  6.6000E+00
STM  2
LBL  IF0
LBL  NAMED3
LBL  LOOP6
LDL  4
LDCI 1
LDCI 50
CHK
LDCI 10
LESI
FJP  LOOP7
LDL  3
LDCI 1
LDCI 50
```

```
CHK
LDCI 10
EQUI
FJP  IF2
LDCI 5
STL  4
UJP  IF3
LBL  IF2
LDL  3
LDCI 1
LDCI 50
CHK
LDCI 9
EQUI
FJP  IF4
LDCI 7
STL  4
UJP  IF5
LBL  IF4
LDL  3
LDCI 1
LDCI 50
CHK
LDCI 7
EQUI
FJP  IF6
LDCI 5
STL  4
UJP  IF7
LBL  IF6
LDCI 3
STL  4
LBL  IF7
LBL  IF5
LBL  IF3
UJP  LOOP7
UJP  LOOP6
LBL  LOOP7
LDCI 3
INCI
STL  11
LBL LOOP8
LDL  11
DECI
DUP1
STL  11
LDCI 0
GEQI
FJP  LOOP9
LDL  11
STL  2
STL  0
LDL  4
LDCI 1
LDCI 50
CHK
```

```
STL  0
UJP  LOOP8
LBL  LOOP9
LDCI 3
STL  2
RPU  0

PROCEDURE NUMBER = 1
LEVEL OF DEFN.   = 1
PARAMETERS       = 2
DATA             = 10
```

```
--EXAMPLE 5
PROCEDURE ARRTEST IS --TEST ARRAY AND RECORD ACCESS.
TYPE E IS RECORD
 D : INTEGER;
END RECORD;
C : E;
A,B : INTEGER;
F :  ARRAY(1..10) OF INTEGER;
BEGIN
 A := 3;
 B := 0;
 WHILE B < 10 LOOP
  F(B) := (4 * 3 + B)/A;
  B  := B + A;
 END LOOP;
 C.D := B;
END ARRTEST;
```

28	11	12	0	0	0	0	0	0	0	0	0	0.00000	.00000	00
118	13	0	0	0	0	0	0	0	0	0	0	0.00000	.00000	00
117	14	13	16	0	0	0	0	0	0	0	0	0.00000	.00000	00ARRTEST
137	15	14	13	16	0	0	0	0	0	0	0	0.00000	.00000	00
82	17	18	0	0	0	0	0	0	0	0	0	0.00000	.00000	00
123	20	21	0	0	0	0	0	0	0	0	0	0.00000	.00000	00
70	22	23	0	0	0	0	0	0	0	0	0	0.00000	.00000	00
24	23	24	0	0	0	0	0	0	0	0	0	0.00000	.00000	00D
33	24	25	0	4	4	0	0	0	0	0	0	0.00000	.00000	00
157	25	4	0	0	0	0	0	0	0	0	0	0.00000	.00000	00INTEGER
160	21	22	0	24	0	0	0	0	0	0	0	0.00000	.00000	00
149	18	19	0	20	0	0	0	0	0	0	0	28.00000	.00000	00
150	19	20	0	0	0	0	0	0	0	0	0	0.00000	.00000	00E
70	26	27	0	0	0	0	0	0	0	0	0	0.00000	.00000	00
161	27	29	0	0	0	0	0	0	0	0	0	0.00000	.00000	00C
160	28	26	0	29	0	0	0	0	0	0	0	34.00000	.00000	00
33	29	30	0	20	20	0	0	0	0	0	0	0.00000	.00000	00
157	30	19	0	0	0	0	0	0	0	0	0	0.00000	.00000	00E
70	31	32	0	0	0	0	0	0	0	0	0	0.00000	.00000	00
161	32	35	0	0	0	0	0	0	0	0	0	33.00000	.00000	00A
161	33	35	0	0	0	0	0	0	0	0	0	0.00000	.00000	00B
160	34	31	0	35	0	0	0	0	0	0	0	39.00000	.00000	00
33	35	36	0	4	4	0	0	0	0	0	0	0.00000	.00000	00
157	36	4	0	0	0	0	0	0	0	0	0	0.00000	.00000	00INTEGER
44	41	42	0	0	0	0	0	0	0	0	0	0.00000	.00000	00
70	37	38	0	0	0	0	0	0	0	0	0	0.00000	.00000	00
161	38	40	0	0	0	0	0	0	0	0	0	0.00000	.00000	00F
134	47	49	0	0	0	0	0	0	0	0	0	0.00000	.00000	00
157	48	32	0	0	0	0	0	0	0	0	0	0.00000	.00000	00A
13	49	48	50	0	0	0	0	0	0	0	0	52.00000	.00000	00
100	50	4	0	0	0	0	0	0	0	0	0	0.30000+001	.00000	103
157	51	33	0	0	0	0	0	0	0	0	0	0.00000	.00000	00B
13	52	51	53	0	0	0	0	0	0	0	0	54.00000	.00000	00
100	53	4	0	0	0	0	0	0	0	0	0	0.00000	.00000	100
53	64	65	0	0	0	0	0	0	0	0	0	0.00000	.00000	00
134	61	66	0	0	0	0	0	0	0	0	0	0.00000	.00000	00
157	62	38	0	0	0	0	0	0	0	0	0	0.00000	.00000	00F
78	63	62	64	0	0	0	0	0	0	0	0	0.00000	.00000	00
157	65	33	0	0	0	0	0	0	0	0	0	0.00000	.00000	00B
100	70	4	0	0	0	0	0	0	0	0	0	74.40000+001	.00000	104
62	71	72	73	0	0	0	0	0	0	0	0	78.00000	.00000	00
138	72	0	0	0	0	0	0	0	0	0	0	0.00000	.00000	00*
109	73	70	0	0	0	0	0	0	0	0	0	0.00000	.00000	00
100	74	4	0	0	0	0	0	0	0	0	0	0.30000+001	.00000	103
62	75	76	77	0	0	0	0	0	0	0	0	0.00000	.00000	00
138	76	0	0	0	0	0	0	0	0	0	0	0.00000	.00000	00+
109	77	71	0	0	0	0	0	0	0	0	0	0.00000	.00000	00
157	78	33	0	0	0	0	0	0	0	0	0	0.00000	.00000	00B
13	66	63	79	0	0	0	0	0	0	0	0	84.00000	.00000	00
111	67	75	0	0	0	0	0	0	0	0	0	82.00000	.00000	00
62	79	80	81	0	0	0	0	0	0	0	0	0.00000	.00000	00
138	80	0	0	0	0	0	0	0	0	0	0	0.00000	.00000	00/
109	81	67	0	0	0	0	0	0	0	0	0	0.00000	.00000	00
157	82	32	0	0	0	0	0	0	0	0	0	0.00000	.00000	00A
157	83	33	0	0	0	0	0	0	0	0	0	0.00000	.00000	00B
13	84	83	86	0	0	0	0	0	0	0	0	0.00000	.00000	00

157	85	33	0	0	0	0	0	0	0	0	0	B9.00000	.00000	00B
62	86	87	88	0	0	0	0	0	0	0	0	0.00000	.00000	00
138	87	0	0	0	0	0	0	0	0	0	0	0.00000	.00000	00+
109	88	85	0	0	0	0	0	0	0	0	0	0.00000	.00000	00
157	89	32	0	0	0	0	0	0	0	0	0	0.00000	.00000	00A
100	60	4	0	0	0	0	0	0	0	0	0	0.10000+002.00000		1010
157	90	27	0	0	0	0	0	0	0	0	0	0.00000	.00000	00C
131	91	90	92	20	0	0	0	0	0	0	0	0.00000	.00000	00
157	92	23	0	0	0	0	0	0	0	0	0	0.00000	.00000	00D
88	54	55	61	0	0	0	0	0	0	0	0	93.00000	.00000	00
167	55	57	0	0	0	0	0	0	0	0	0	0.00000	.00000	00
157	56	33	0	0	0	0	0	0	0	0	0	60.00000	.00000	00B
62	57	58	59	0	0	0	0	0	0	0	0	0.00000	.00000	00
138	58	0	0	0	0	0	0	0	0	0	0	0.00000	.00000	00<
109	59	56	0	0	0	0	0	0	0	0	0	0.00000	.00000	00
13	93	91	94	0	0	0	0	0	0	0	0	0.00000	.00000	00
157	94	33	0	0	0	0	0	0	0	0	0	0.00000	.00000	00B
19	16	17	47	0	0	0	0	0	0	0	0	0.00000	.00000	00
160	39	37	0	40	0	0	0	0	0	0	0	0.00000	.00000	00
12	40	41	45	0	0	0	0	0	0	0	0	0.00000	.00000	00
122	42	43	44	4	0	0	0	0	0	0	0	0.00000	.00000	00
100	43	4	0	0	0	0	0	0	0	0	0	0.10000+001.00000		101
100	44	4	0	0	0	0	0	0	0	0	0	0.10000+002.00000		1010
33	45	46	0	4	4	0	0	0	0	0	0	0.00000	.00000	00
157	46	4	0	0	0	0	0	0	0	0	0	0.00000	.00000	00INTEGER
27	12	0	0	15	0	0	0	0	0	0	0	0.00000	.00000	00

```
SLDC 0
DUP1
STL  2
SLDC 1
ADI
POP1
LLA  3
SLDC 0
SLDC 1
ADI
DUP1
NEW
STM  2
LLA  7
SLDC 1
LDCI 10
LDCI 1
SBI
INCI
MPI
SLDC 1
MPI
DUP1
NEW
STM  2
LDCI 3
STL  5
LDCI 0
STL  6
LBL  LOOP0
LDL  6
LDCI 10
LESI
FJP  LOOP1
LDL  6
LLA  7
LDM  1
SLDC 1
SLDC 0
SLDC 1
DUP1
SWAP51
LDCI 1
LDCI 10
CHK
LDCI 1
SBI
MPI
SWAP21
ADI
SWAP
POP1
MPI
ADI
LDCI 4
LDCI 3
```

```
MPI
LDL  6
ADI
LDL  5
DVI
STM  1
LDL  6
LDL  5
ADI
STL  6
UJP  LOOP0
LBL  LOOP1
LLA  3
LDM  1
LDL  2
ADI
LDL  6
STM  1
RPU  0

PROCEDURE NUMBER = 1
LEVEL OF DEFN.   = 1
PARAMETERS       = 2
DATA             = 7
```

```
--EXAMPLE 6
PROCEDURE TEST IS --SUBTYPE AND CONSTANT TEST
TYPE DAY IS (MON,TUE,WED,THU,FRI,SAT,SUN);
SUBTYPE WEEKDAY IS DAY RANGE (MON .. FRI);
PI : CONSTANT := 3.14159;
A : WEEKDAY;
B : FLOAT;
BEGIN
   A := TUE;
   B := PI;
END TEST;
```

28	11	12	0	0	0	0	0	0	0	0	0	0.00000	.00000	00
118	13	0	0	0	0	0	0	0	0	0	0	0.00000	.00000	00
117	14	13	16	0	0	0	0	0	0	0	0	0.00000	.00000	00TEST
137	15	14	13	16	0	0	0	0	0	0	0	0.00000	.00000	00
82	17	18	0	0	0	0	0	0	0	0	0	0.00000	.00000	00
149	18	19	0	20	0	0	0	0	0	0	0	28.00000	.00000	00
150	19	20	0	0	0	0	0	0	0	0	0	0.00000	.00000	00DAY
49	20	21	0	0	0	0	0	0	0	0	0	0.00000	.00000	00
48	21	20	0	0	0	0	0	0	0	0	0	22.00000	.00000	10MON
48	22	20	0	0	0	0	0	0	0	0	0	23.10000+001	.00000	10TUE
48	23	20	0	0	0	0	0	0	0	0	0	24.20000+001	.00000	10WED
48	24	20	0	0	0	0	0	0	0	0	0	25.30000+001	.00000	10THU
48	25	20	0	0	0	0	0	0	0	0	0	26.40000+001	.00000	10FRI
48	26	20	0	0	0	0	0	0	0	0	0	27.50000+001	.00000	10SAT
48	27	20	0	0	0	0	0	0	0	0	0	0.60000+001	.00000	10SUN
140	28	29	30	0	0	0	0	0	0	0	0	37.00000	.00000	00
141	29	30	0	0	0	0	0	0	0	0	0	0.00000	.00000	00WEEKDAY
33	30	31	32	20	20	32	0	0	0	0	0	0.00000	.00000	00
157	31	19	0	0	0	0	0	0	0	0	0	0.00000	.00000	00DAY
122	32	33	34	20	0	0	0	0	0	0	0	0.00000	.00000	00
158	33	21	20	0	0	0	0	0	0	0	0	0.00000	.00000	10MON
158	34	25	20	0	0	0	0	0	0	0	0	0.00000	.00000	10FRI
70	35	36	0	0	0	0	0	0	0	0	0	0.00000	.00000	00
99	36	3	38	0	0	0	0	0	0	0	0	0.00000	.00000	00PI
98	37	35	38	0	0	0	0	0	0	0	0	41.00000	.00000	00
100	38	3	0	0	0	0	0	0	0	0	0	0.31416+001	.00000	103.14159
70	39	40	0	0	0	0	0	0	0	0	0	0.00000	.00000	00
161	40	42	0	0	0	0	0	0	0	0	0	0.00000	.00000	00A
160	41	39	0	42	0	0	0	0	0	0	0	46.00000	.00000	00
33	42	43	0	20	20	32	0	0	0	0	0	0.00000	.00000	00
157	43	29	0	0	0	0	0	0	0	0	0	0.00000	.00000	00WEEKDAY
70	44	45	0	0	0	0	0	0	0	0	0	0.00000	.00000	00
161	45	47	0	0	0	0	0	0	0	0	0	0.00000	.00000	00B
134	49	51	0	0	0	0	0	0	0	0	0	0.00000	.00000	00
157	50	40	0	0	0	0	0	0	0	0	0	0.00000	.00000	00A
13	51	50	52	0	0	0	0	0	0	0	0	54.00000	.00000	00
158	52	22	20	0	0	0	0	0	0	0	0	0.00000	.00000	10TUE
157	53	45	0	0	0	0	0	0	0	0	0	0.00000	.00000	00B
13	54	53	55	0	0	0	0	0	0	0	0	0.00000	.00000	00
158	55	36	3	0	0	0	0	0	0	0	0	0.31416+001	.00000	10PI
19	16	17	49	0	0	0	0	0	0	0	0	0.00000	.00000	00
160	46	44	0	47	0	0	0	0	0	0	0	0.00000	.00000	00
33	47	48	0	3	3	0	0	0	0	0	0	0.00000	.00000	00
157	48	3	0	0	0	0	0	0	0	0	0	0.00000	.00000	00FLOAT
27	12	0	0	15	0	0	0	0	0	0	0	0.00000	.00000	00

```
LDCI 1
STL  2
LLA  3
LDC  2  3.1416E+00
STM  2
RPU  0

PROCEDURE NUMBER = 1
LEVEL OF DEFN.   = 1
PARAMETERS       = 2
DATA             = 3
```

```
--EXAMPLE 7
PROCEDURE TEST IS
TYPE A IS ARRAY(1..10) OF FLOAT;
B : ARRAY(1..5) OF A;
BEGIN
 B(2)(1) := 1.0;
END;
```

28	11	12	0	0	0	0	0	0	0	0	0	0.00000	.00000	00
118	13	0	0	0	0	0	0	0	0	0	0	0.00000	.00000	00
117	14	13	16	0	0	0	0	0	0	0	0	0.00000	.00000	00TEST
137	15	14	13	16	0	0	0	0	0	0	0	0.00000	.00000	00
82	17	18	0	0	0	0	0	0	0	0	0	0.00000	.00000	00
44	21	22	0	0	0	0	0	0	0	0	0	0.00000	.00000	00
149	18	19	0	20	0	0	0	0	0	0	0	29.00000	.00000	00
150	19	20	0	0	0	0	0	0	0	0	0	0.00000	.00000	00A
12	20	21	25	0	0	0	0	0	0	0	0	0.00000	.00000	00
122	22	23	24	4	0	0	0	0	0	0	0	0.00000	.00000	00
100	23	4	0	0	0	0	0	0	0	0	0	0.10000+001	.00000	101
100	24	4	0	0	0	0	0	0	0	0	0	0.10000+002	.00000	1010
33	25	26	0	3	3	0	0	0	0	0	0	0.00000	.00000	00
157	26	3	0	0	0	0	0	0	0	0	0	0.00000	.00000	00FLOAT
44	31	32	0	0	0	0	0	0	0	0	0	0.00000	.00000	00
70	27	28	0	0	0	0	0	0	0	0	0	0.00000	.00000	00
161	28	30	0	0	0	0	0	0	0	0	0	0.00000	.00000	00B
53	40	41	0	0	0	0	0	0	0	0	0	0.00000	.00000	00
53	43	44	0	0	0	0	0	0	0	0	0	0.00000	.00000	00
134	37	45	0	0	0	0	0	0	0	0	0	0.00000	.00000	00
157	38	28	0	0	0	0	0	0	0	0	0	0.00000	.00000	00B
78	39	38	40	0	0	0	0	0	0	0	0	0.00000	.00000	00
100	41	4	0	0	0	0	0	0	0	0	0	0.20000+001	.00000	102
78	42	39	43	0	0	0	0	0	0	0	0	0.00000	.00000	00
100	44	4	0	0	0	0	0	0	0	0	0	0.10000+001	.00000	101
13	45	42	46	0	0	0	0	0	0	0	0	0.00000	.00000	00
100	46	3	0	0	0	0	0	0	0	0	0	0.10000+001	.00000	101.0
19	16	17	37	0	0	0	0	0	0	0	0	0.00000	.00000	00
160	29	27	0	30	0	0	0	0	0	0	0	0.00000	.00000	00
12	30	31	35	0	0	0	0	0	0	0	0	0.00000	.00000	00
122	32	33	34	4	0	0	0	0	0	0	0	0.00000	.00000	00
100	33	4	0	0	0	0	0	0	0	0	0	0.10000+001	.00000	101
100	34	4	0	0	0	0	0	0	0	0	0	0.50000+001	.00000	105
33	35	36	0	3	20	0	0	0	0	0	0	0.00000	.00000	00
157	36	19	0	0	0	0	0	0	0	0	0	0.00000	.00000	00A
27	12	0	0	15	0	0	0	0	0	0	0	0.00000	.00000	00

```
LLA  2
SLDC 1
LDCI 5
LDCI 1
SBI
INCI
MPI
SLDC 1
LDCI 10
LDCI 1
SBI
INCI
MPI
SLDC 2
MPI
MPI
DUP1
NEW
STM  2
LDCI 1
LDCI 2
LLA  2
LDM  1
SLDC 1
LDCI 10
LDCI 1
SBI
INCI
MPI
SLDC 2
MPI
SLDC 0
SLDC 1
DUP1
SWAP51
LDCI 1
LDCI 5
CHK
LDCI 1
SBI
MPI
SWAP21
ADI
SWAP
POP1
MPI
ADI
SLDC 2
SLDC 0
SLDC 1
DUP1
SWAP51
LDCI 1
LDCI 10
CHK
LDCI 1
```

```
SBI
MPI
SWAP21
ADI
SWAP
POP1
MPI
ADI
LDC  2  1.0000E+00
STM  2
RPU  0

PROCEDURE NUMBER = 1
LEVEL OF DEFN.   = 1
PARAMETERS       = 2
DATA             = 2
```

```
--EXAMPLE 8
PROCEDURE TEST IS
TYPE A IS ARRAY(1..10,1..5) OF FLOAT;
B : A;
BEGIN
B(2,1) := 1.0;
END;
```

28	11	12	0	0	0	0	0	0	0	0	0	0.00000	.00000	00
118	13	0	0	0	0	0	0	0	0	0	0	0.00000	.00000	00
117	14	13	16	0	0	0	0	0	0	0	0	0.00000	.00000	00TEST
137	15	14	13	16	0	0	0	0	0	0	0	0.00000	.00000	00
82	17	18	0	0	0	0	0	0	0	0	0	0.00000	.00000	00
44	21	22	0	0	0	0	0	0	0	0	0	0.00000	.00000	00
122	22	23	24	4	0	0	0	0	0	0	0	25.00000	.00000	00
149	18	19	0	20	0	0	0	0	0	0	0	32.00000	.00000	00
150	19	20	0	0	0	0	0	0	0	0	0	0.00000	.00000	00A
12	20	21	28	0	0	0	0	0	0	0	0	0.00000	.00000	00
100	23	4	0	0	0	0	0	0	0	0	0	0.10000+001	.00000	101
100	24	4	0	0	0	0	0	0	0	0	0	0.10000+002	.00000	1010
122	25	26	27	4	0	0	0	0	0	0	0	0.00000	.00000	00
100	26	4	0	0	0	0	0	0	0	0	0	0.10000+001	.00000	101
100	27	4	0	0	0	0	0	0	0	0	0	0.50000+001	.00000	105
33	28	29	0	3	3	0	0	0	0	0	0	0.00000	.00000	00
157	29	3	0	0	0	0	0	0	0	0	0	0.00000	.00000	00FLOAT
70	30	31	0	0	0	0	0	0	0	0	0	0.00000	.00000	00
161	31	33	0	0	0	0	0	0	0	0	0	0.00000	.00000	00B
53	38	39	0	0	0	0	0	0	0	0	0	0.00000	.00000	00
100	39	4	0	0	0	0	0	0	0	0	0	40.20000+001	.00000	102
134	35	41	0	0	0	0	0	0	0	0	0	0.00000	.00000	00
157	36	31	0	0	0	0	0	0	0	0	0	0.00000	.00000	00B
78	37	36	38	3	0	0	0	0	0	0	0	0.00000	.00000	00
100	40	4	0	0	0	0	0	0	0	0	0	0.10000+001	.00000	101
13	41	37	42	0	0	0	0	0	0	0	0	0.00000	.00000	00
100	42	3	0	0	0	0	0	0	0	0	0	0.10000+001	.00000	101.0
19	16	17	35	0	0	0	0	0	0	0	0	0.00000	.00000	00
160	32	30	0	33	0	0	0	0	0	0	0	0.00000	.00000	00
33	33	34	0	3	20	0	0	0	0	0	0	0.00000	.00000	00
157	34	19	0	0	0	0	0	0	0	0	0	0.00000	.00000	00A
27	12	0	0	15	0	0	0	0	0	0	0	0.00000	.00000	00

```
LLA  2
SLDC 1
LDCI 10
LDCI 1
SBI
INCI
MPI
LDCI 5
LDCI 1
SBI
INCI
MPI
SLDC 2
MPI
DUP1
NEW
STM  2
LDCI 1
LDCI 2
LLA  2
LDM  1
SLDC 2
SLDC 0
SLDC 1
DUP1
SWAP51
LDCI 1
LDCI 10
CHK
LDCI 1
SBI
MPI
SWAP21
ADI
SWAP
LDCI 10
LDCI 1
SBI
INCI
MPI
DUP1
SWAP51
LDCI 1
LDCI 5
CHK
LDCI 1
SBI
MPI
SWAP21
ADI
SWAP
POP1
MPI
ADI
LDC  2  1.0000E+00
STM  2
```

RPU 0

```
PROCEDURE NUMBER = 1
LEVEL OF DEFN.   = 1
PARAMETERS       = 2
DATA             = 2
```

```
--EXAMPLE 9
PROCEDURE TEST IS
A : INTEGER;
BEGIN
 SCAN : LOOP
  A := A+1;
  EXIT SCAN WHEN A > 10;
 END LOOP SCAN;
END TEST;
```

28	11	12	0	0	0	0	0	0	0	0	0	0.00000	.00000	00
118	13	0	0	0	0	0	0	0	0	0	0	0.00000	.00000	00
117	14	13	16	0	0	0	0	0	0	0	0	0.00000	.00000	00TEST
137	15	14	13	16	0	0	0	0	0	0	0	0.00000	.00000	00
82	17	20	0	0	0	0	0	0	0	0	0	0.00000	.00000	00
70	18	19	0	0	0	0	0	0	0	0	0	0.00000	.00000	00
161	19	21	0	0	0	0	0	0	0	0	0	0.00000	.00000	00A
134	23	25	0	0	0	0	0	0	0	0	0	0.00000	.00000	00
134	28	30	0	0	0	0	0	0	0	0	0	0.00000	.00000	00
157	29	19	0	0	0	0	0	0	0	0	0	0.00000	.00000	00A
13	30	29	32	0	0	0	0	0	0	0	0	36.00000	.00000	00
157	31	19	0	0	0	0	0	0	0	0	0	35.00000	.00000	00A
62	32	33	34	0	0	0	0	0	0	0	0	0.00000	.00000	00
138	33	0	0	0	0	0	0	0	0	0	0	0.00000	.00000	00+
109	34	31	0	0	0	0	0	0	0	0	0	0.00000	.00000	00
100	35	4	0	0	0	0	0	0	0	0	0	0.10000+001	.00000	101
52	36	37	39	26	0	0	0	0	0	0	0	0.00000	.00000	00
157	37	24	0	0	0	0	0	0	0	0	0	0.00000	.00000	00SCAN
157	38	19	0	0	0	0	0	0	0	0	0	42.00000	.00000	00A
62	39	40	41	0	0	0	0	0	0	0	0	0.00000	.00000	00
138	40	0	0	0	0	0	0	0	0	0	0	0.00000	.00000	00>
109	41	38	0	0	0	0	0	0	0	0	0	0.00000	.00000	00
100	42	4	0	0	0	0	0	0	0	0	0	0.10000+002	.00000	1010
86	24	26	0	0	0	0	0	0	0	0	0	0.00000	.00000	00SCAN
92	25.	24	26	0	0	0	0	0	0	0	0	0.00000	.00000	00
88	26	27	28	0	0	0	0	0	0	0	0	0.00000	.00000	00
166	27	0	0	0	0	0	0	0	0	0	0	0.00000	.00000	00
19	16	17	23	0	0	0	0	0	0	0	0	0.00000	.00000	00
160	20	18	0	21	0	0	0	0	0	0	0	0.00000	.00000	00
33	21	22	0	4	4	0	0	0	0	0	0	0.00000	.00000	00
157	22	4	0	0	0	0	0	0	0	0	0	0.00000	.00000	00INTEGER
27	12	0	0	15	0	0	0	0	0	0	0	0.00000	.00000	00

```
LBL  LOOP0
LDL  2
LDCI 1
ADI
STL  2
LDL  2
LDCI 10
GRTI
TJP  SCAN
UJP  LOOP0
LBL  LOOP1
LBL  SCAN
RPU  0

PROCEDURE NUMBER = 1
LEVEL OF DEFN.   = 1
PARAMETERS       = 2
DATA             = 1
```

```
--EXAMPLE 10
PROCEDURE TEST IS
A : INTEGER;
BEGIN
   A := 23;
   A := A * A + 1;
END TEST;
```

28	11	12	0	0	0	0	0	0	0	0	0	0.00000	.00000	00
118	13	0	0	0	0	0	0	0	0	0	0	0.00000	.00000	00
117	14	13	16	0	0	0	0	0	0	0	0	0.00000	.00000	00TEST
137	15	14	13	16	0	0	0	0	0	0	0	0.00000	.00000	00
82	17	20	0	0	0	0	0	0	0	0	0	0.00000	.00000	00
70	18	19	0	0	0	0	0	0	0	0	0	0.00000	.00000	00
161	19	21	0	0	0	0	0	0	0	0	0	0.00000	.00000	00A
134	23	25	0	0	0	0	0	0	0	0	0	0.00000	.00000	00
157	24	19	0	0	0	0	0	0	0	0	0	0.00000	.00000	00A
13	25	24	26	0	0	0	0	0	0	0	0	28.00000	.00000	00
100	26	4	0	0	0	0	0	0	0	0	0	0.23000+002	.00000	1023
157	27	19	0	0	0	0	0	0	0	0	0	0.00000	.00000	00A
13	28	27	34	0	0	0	0	0	0	0	0	0.00000	.00000	00
157	29	19	0	0	0	0	0	0	0	0	0	33.00000	.00000	00A
62	30	31	32	0	0	0	0	0	0	0	0	37.00000	.00000	00
138	31	0	0	0	0	0	0	0	0	0	0	0.00000	.00000	00*
109	32	29	0	0	0	0	0	0	0	0	0	0.00000	.00000	00
157	33	19	0	0	0	0	0	0	0	0	0	0.00000	.00000	00A
62	34	35	36	0	0	0	0	0	0	0	0	0.00000	.00000	00
138	35	0	0	0	0	0	0	0	0	0	0	0.00000	.00000	00+
109	36	30	0	0	0	0	0	0	0	0	0	0.00000	.00000	00
100	37	4	0	0	0	0	0	0	0	0	0	0.10000+001	.00000	101
19	16	17	23	0	0	0	0	0	0	0	0	0.00000	.00000	00
160	20	18	0	21	0	0	0	0	0	0	0	0.00000	.00000	00
33	21	22	0	4	4	0	0	0	0	0	0	0.00000	.00000	00
157	22	4	0	0	0	0	0	0	0	0	0	0.00000	.00000	00INTEGER
27	12	0	0	15	0	0	0	0	0	0	0	0.00000	.00000	00

```
LDCI 23
STL  2
LDL  2
LDL  2
MPI
LDCI 1
ADI
STL  2
RPU  0

PROCEDURE NUMBER = 1
LEVEL OF DEFN.   = 1
PARAMETERS       = 2
DATA             = 1
```

```
--EXAMPLE 11
PROCEDURE TEST IS
A,B : INTEGER := 0;
CONDIT : BOOLEAN := TRUE;
BEGIN
  WHILE CONDIT LOOP
    A := A+1;
    B := A*2;
    IF A > 10 THEN
     CONDIT := FALSE;
    END IF;
  END LOOP;
END TEST;
```

28	11	12	0	0	0	0	0	0	0	0	0	0.00000	.00000	00
118	13	0	0	0	0	0	0	0	0	0	0	0.00000	.00000	00
117	14	13	16	0	0	0	0	0	0	0	0	0.00000	.00000	00TEST
137	15	14	13	16	0	0	0	0	0	0	0	0.00000	.00000	00
82	17	21	0	0	0	0	0	0	0	0	0	0.00000	.00000	00
70	18	19	0	0	0	0	0	0	0	0	0	0.00000	.00000	00
161	19	22	24	0	0	0	0	0	0	0	0	20.00000	.00000	00A
161	20	22	24	0	0	0	0	0	0	0	0	0.00000	.00000	00B
160	21	18	24	22	0	0	0	0	0	0	0	27.00000	.00000	00
33	22	23	0	4	4	0	0	0	0	0	0	0.00000	.00000	00
157	23	4	0	0	0	0	0	0	0	0	0	0.00000	.00000	00INTEGER
100	24	4	0	0	0	0	0	0	0	0	0	0.00000	.00000	100
70	25	26	0	0	0	0	0	0	0	0	0	0.00000	.00000	00
161	26	28	30	0	0	0	0	0	0	0	0	0.00000	.00000	00CONDIT
134	31	32	0	0	0	0	0	0	0	0	0	0.00000	.00000	00
134	35	37	0	0	0	0	0	0	0	0	0	0.00000	.00000	00
157	36	19	0	0	0	0	0	0	0	0	0	0.00000	.00000	00A
13	37	36	39	0	0	0	0	0	0	0	0	44.00000	.00000	00
157	38	19	0	0	0	0	0	0	0	0	0	42.00000	.00000	00A
62	39	40	41	0	0	0	0	0	0	0	0	0.00000	.00000	00
138	40	0	0	0	0	0	0	0	0	0	0	0.00000	.00000	00+
109	41	38	0	0	0	0	0	0	0	0	0	0.00000	.00000	00
100	42	4	0	0	0	0	0	0	0	0	0	0.10000+001	.00000	101
157	43	20	0	0	0	0	0	0	0	0	0	0.00000	.00000	00B
13	44	43	46	0	0	0	0	0	0	0	0	50.00000	.00000	00
157	45	19	0	0	0	0	0	0	0	0	0	49.00000	.00000	00A
62	46	47	48	0	0	0	0	0	0	0	0	0.00000	.00000	00
138	47	0	0	0	0	0	0	0	0	0	0	0.00000	.00000	00*
109	48	45	0	0	0	0	0	0	0	0	0	0.00000	.00000	00
100	49	4	0	0	0	0	0	0	0	0	0	0.20000+001	.00000	102
134	57	59	0	0	0	0	0	0	0	0	0	0.00000	.00000	00
157	58	26	0	0	0	0	0	0	0	0	0	0.00000	.00000	00CONDIT
13	59	58	60	0	0	0	0	0	0	0	0	0.00000	.00000	00
158	60	0	0	0	0	0	0	0	0	0	0	0.00000	.00000	10FALSE
71	50	51	0	0	0	0	0	0	0	0	0	0.00000	.00000	00
29	51	53	57	0	0	0	0	0	0	0	0	0.00000	.00000	00
157	52	19	0	0	0	0	0	0	0	0	0	56.00000	.00000	00A
62	53	54	55	0	0	0	0	0	0	0	0	0.00000	.00000	00
138	54	0	0	0	0	0	0	0	0	0	0	0.00000	.00000	00)
109	55	52	0	0	0	0	0	0	0	0	0	0.00000	.00000	00
100	56	4	0	0	0	0	0	0	0	0	0	0.10000+002	.00000	1010
88	32	33	35	0	0	0	0	0	0	0	0	0.00000	.00000	00
167	33	34	0	0	0	0	0	0	0	0	0	0.00000	.00000	00
157	34	26	0	0	0	0	0	0	0	0	0	0.00000	.00000	00CONDIT
19	16	17	31	0	0	0	0	0	0	0	0	0.00000	.00000	00
160	27	25	30	28	0	0	0	0	0	0	0	0.00000	.00000	00
33	28	29	0	0	0	0	0	0	0	0	0	0.00000	.00000	00
157	29	0	0	0	0	0	0	0	0	0	0	0.00000	.00000	00BOOLEAN
158	30	0	0	0	0	0	0	0	0	0	0	0.00000	.00000	10TRUE
27	12	0	0	15	0	0	0	0	0	0	0	0.00000	.00000	00

```
LDCI 0
STL  2
LDCI 0
STL  3
SLDC 1
STL  4
LBL  LOOP0
LDL  4
FJP  LOOP1
LDL  2
LDCI 1
ADI
STL  2
LDL  2
LDCI 2
MPI
STL  3
LDL  2
LDCI 10
GRTI
FJP  IF0
SLDC 0
STL  4
LBL  IF0
UJP  LOOP0
LBL  LOOP1
RPU  0

PROCEDURE NUMBER = 1
LEVEL OF DEFN.   = 1
PARAMETERS       = 2
DATA             = 3
```

APPENDIX F

ERROR MESSAGES

The first two groups of error messages listed here (sub-chapters F.1 and F.2) are produced by the tree specification table translator and code template table translator respectively. When an error occurs, an error message is printed underneath the offending line in the output listing of the table being scanned. Also, to highlight the problem, the "^" symbol is printed underneath the location of the error in the line (between the offending line and the error message).

The third group of error messages listed here (sub-chapter F.3) is produced by the code generator. These error messages are printed in the output trace listing (whenever errors occur). The error messages are quite detailed. Such detail helps to locate any run time errors in the tables. Each error message tells at which node the error occurs in the tree being scanned, in which node definition the error exists and what the error is. Wherever possible, the scan continues after the error message is generated. However, a few errors make any further scanning of a tree meaningless. When this happens, the code generator stops. The error message informs which kind of error has occurred.

F.1 Tree Specification Table Translator Error Messages

****** LINE TOO LONG

****** PREMATURE LINE ENDING

****** KIND NUMBER IS TOO LARGE

****** KIND FIELD MUST BE NUMERIC

****** WORD IS TOO BIG

****** OPEN BRACKET IS MISSING

****** CLOSE BRACKET IS MISSING

****** UNABLE TO RECOGNIZE THIS WORD

****** NUMBER IS TOO LARGE

****** TOO MANY REFERENCE FIELDS FOR THIS NODE TYPE

****** TOO MANY ERRORS ON THIS LINE TO LIST OUT

F.2 Code Template Table Translator Error Messages

****** LINE TOO LONG

****** PREMATURE LINE ENDING

****** KIND NUMBER IS TOO LARGE

****** KIND FIELD MUST BE NUMERIC

****** WORD IS TOO BIG

****** OPEN BRACKET IS MISSING

****** CLOSE BRACKET IS MISSING

****** UNABLE TO RECOGNIZE THIS WORD

****** NUMBER IS TOO LARGE

****** TOO MANY REFERENCE FIELDS FOR THIS NODE TYPE

****** OUTPUT-LINE OVERFLOW (REDUCE TEXT FOR THIS NODE)

****** CASE SELECTOR IN COMM STRUCTURE IS TOO LONG

****** HIST LETTER IS MISSING

****** NATIVE TYPE WORD IS MISSING

****** TOO MANY ERRORS ON THIS LINE TO LIST OUT

F.3 The Code Generator Error Messages

```
****ERROR AT NODE SEQUENCE NO. <number>:
AN UNRECOGNIZED SUBSTITUTE OPERATOR
EXISTS IN NODE KIND NO. <number>.
CONTINUING THE SCAN......
```

```
****ERROR AT NODE SEQUENCE NO. <number>:
MULTIPLE USE OF SOME CONDITIONAL STRUCTURES IS
NOT PERMITTED.   CHECK NODE KIND NO <number>
IN THE TEMPLATE TABLE FOR THE PROBLEM CONDITIONAL
IN A "CODE" STRUCTURE.
CONTINUING THE SCAN......
```

```
****ERROR AT NODE SEQUENCE NO. <number>:
AN ATTEMPT WAS MADE TO FORCE AN ATTRIBUTE LOAD
OF A NON-EXISTENT NATIVE TYPE IN THE
TEMPLATE TABLE DEFINITION OF NODE KIND NO. <number>.
CHECK THE UNTRANSLATED TEMPLATE TABLE TO SEE
IF THIS FORCED TYPE (AND ITS SPELLING) AGREES
WITH THE TYPE ENTRIES IN THE INFO NODE (NODE 0).
CONTINUING THE SCAN......
```

```
****ERROR AT NODE SEQUENCE NO. <number>:
NO NODE EXISTS IN THE SYMBOL TABLE FOR THIS
NODE SEQUENCE NUMBER.   THE COMMAND WORD
"IDINFOGRAB" MUST BE IN THE SAME NODE
DEFINITION AS THE COMMAND WORD "DEFINEID"
CONTINUING THE SCAN......
```

```
****ERROR AT NODE SEQUENCE NO. <number>:
NO NODE EXISTS IN THE SYMBOL TABLE
FOR THE NATIVE TYPE WORD "<word>".
CONTINUING THE SCAN......
```

```
****ERROR AT NODE SEQUENCE NO. <number>:
VALUE FIELD DEFINITIONS IN THE TWO TABLES
FOR NODE KIND NO. <number> DO NOT MATCH
IN EXISTENCE.
CONTINUING THE SCAN......
```

```
****ERROR AT NODE SEQUENCE NO. <number>:
BOOLEAN DEFINITIONS IN THE TWO TABLES FOR
NODE KIND NO. <number> DO NOT MATCH
IN EXISTENCE.
CONTINUING THE SCAN......


****ERROR AT NODE SEQUENCE NO. <number>:
REP FIELD DEFINITIONS IN THE TWO TABLES
FOR NODE KIND NO. <number> DO NOT MATCH
IN EXISTENCE.
CONTINUING THE SCAN......


****ERROR AT NODE SEQUENCE NO. <number>:
OFFSET STORE <number> DOES NOT
EXIST.   CHECK NODE KIND NO. <number>
IN THE TEMPLATE TABLE FOR THE PROBLEM.
CONTINUING THE SCAN......


****ERROR AT NODE SEQUENCE NO. <number>:
LOCAL STORE <number> DOES NOT
EXIST.   CHECK NODE KIND NO. <number>
IN THE TEMPLATE TABLE FOR THE PROBLEM.
CONTINUING THE SCAN......


****ERROR AT NODE SEQUENCE NO. <number>:
GLOBAL STORE <number> DOES NOT
EXIST.   CHECK NODE KIND NO. <number>
IN THE TEMPLATE TABLE FOR THE PROBLEM.
CONTINUING THE SCAN......


****ERROR AT NODE SEQUENCE NO. <number>:
AN ATTEMPT WAS MADE TO INCREMENT THE HISTORY
ARRAY INDEX BEYOND THE LAST ELEMENT OF THE ARRAY.
CHECK THE DEFINITION FOR NODE KIND NO. <number>
IN THE TRANSLATED TEMPLATE TABLE.
CONTINUING THE SCAN......
```

```
****ERROR AT NODE SEQUENCE NO. <number>:
AN ATTEMPT WAS MADE TO DECREMENT THE HISTORY
ARRAY INDEX PAST THE FIRST ELEMENT OF THE ARRAY.
CHECK THE DEFINITION FOR NODE KIND NO. <number>
IN THE TRANSLATED TEMPLATE TABLE.
CONTINUING THE SCAN......
```

```
****ERROR AT NODE SEQUENCE NO. <number>:
THE RESET VALUE FOR THE HISTORY INDEX IS OUT
OF BOUNDS.   IT SHOULD BE IN THE RANGE 1 TO
<histlength>.   CHECK NODE KIND NO. <number>
IN THE TEMPLATE TABLE.
CONTINUING THE SCAN......
```

```
****ERROR AT NODE SEQUENCE NO. <number>:
THE REFERENCE INDEX FOR THE COMPARISON STORE
LOAD OF THE REFERENCED NODE'S NUMBER IS OUT
OF BOUNDS.   IT SHOULD BE IN THE RANGE 2 TO
<endofrefs>.   CHECK NODE KIND NO. <number>
IN THE TEMPLATE TABLE.
CONTINUING THE SCAN......
```

```
****ERROR AT NODE SEQUENCE NO. <number>:
AN UNKNOWN COMMAND WORD SYMBOL WAS FOUND IN THE
TEMPLATE TABLE DEFINITION OF NODE KIND NO. <number>.
CHECK THE TRANSLATED TEMPLATE TABLE TO SEE
WHAT "+" SYMBOLS OCCUR FOR THIS NODE KIND.
THE BAD COMMAND SYMBOL IS "+<letter>".
CONTINUING THE SCAN......
```

```
****ERROR AT NODE SEQUENCE NO. <number>:
AN UNKNOWN COMMAND WORD SYMBOL WAS FOUND IN THE
TEMPLATE TABLE DEFINITION OF NODE KIND NO. <number>.
CHECK THE TRANSLATED TEMPLATE TABLE TO SEE
WHAT "$" SYMBOLS OCCUR FOR THIS NODE KIND.
THE BAD SYMBOL IS "$<letter>".
CONTINUING THE SCAN......
```

****ERROR AT NODE SEQUENCE NO. <number>:
AN UNKNOWN COMMAND WORD SYMBOL WAS FOUND IN THE
TEMPLATE TABLE DEFINITION OF NODE KIND NO. <number>.
CHECK THE TRANSLATED TEMPLATE TABLE TO SEE
WHAT "&" SYMBOLS OCCUR FOR THIS NODE KIND.
THE BAD COMMAND SYMBOL IS "&<letter>".
CONTINUING THE SCAN......

****ERROR IN NODE TYPE NUMBER 0 (THE INFO NODE):
AN UNKNOWN COMMAND WORD SYMBOL WAS FOUND IN THE
TEMPLATE TABLE DEFINITION OF NODE KIND NO. 0.
CHECK THE TRANSLATED TEMPLATE TABLE TO SEE
WHAT "+" SYMBOLS OCCUR IN THE INFO NODE.
THE BAD COMMAND SYMBOL IS "+<letter>".
ABORTING THIS RUN.
BYE......

****ERROR AT NODE SEQUENCE NO. <number>:
SYNTACTIC LINK DEFINITIONS IN THE TWO TABLES
FOR NODE KIND NO. <number> DO NOT MATCH
IN EXISTENCE.
ABORTING THIS RUN.
BYE......

****ERROR AT NODE SEQUENCE NO. <number>:
SEMANTIC LINK DEFINITIONS IN THE TWO TABLES
FOR NODE KIND NO. <number> DO NOT MATCH
IN EXISTENCE.
ABORTING THIS RUN.
BYE......

****ERROR AT NODE SEQUENCE NO. <number>:
MULTIPLE USE OF SOME CONDITIONAL STRUCTURES IS
NOT PERMITTED.   CHECK NODE KIND NO. <number>
IN THE TEMPLATE TABLE FOR THE PROBLEM CONDITIONAL
IN A "COMM" STRUCTURE.
ABORTING THIS RUN.
BYE......

****ERROR AT NODE SEQUENCE NO. <number>:
AN ATTEMPT WAS MADE TO DECREMENT THE LEXICAL LEVEL
PAST THE GLOBAL LEVEL.   CHECK THE TEMPLATE TABLE
DEFINITION FOR NODE KIND NO. <number>.
ABORTING THIS RUN.
BYE......

WARNING AT NODE SEQUENCE NO. <number>:
LEVEL SELECT CONDITIONAL IN NODE KIND NO. <number>
IS SKIPPED TO THE END OF THE SPECIAL COMMAND
WORD BECAUSE NO IDENTIFIER ATTRIBUTE WAS
LOADED SINCE THE LAST CURRENT ID STORE RESET
(HENCE NO LEVEL INFORMATION EXISTS).

WARNING AT NODE SEQUENCE NO. <number>:
NODE KIND NO. <number> IS LISTED IN THE
TABLES AS UNIMPLEMENTED.

24 AUG 1987