RELATIONAL DATABASE STRUCTURE FOR STORAGE AND MANIPULATION OF DEPENDENCY GRAPHS

Sivagnanam Ramasundaram Easwar

Department of Computer Sciences
The University of Texas at Austin
Austin, Texas 78712-1188

TR-88-13 April 1988

To my parents

RELATIONAL DATABASE STRUCTURE FOR STORAGE AND MANIPULATION OF DEPENDENCY GRAPHS

by

SIVAGNANAM RAMASUNDARAM EASWAR, B.E.

THESIS

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

MASTER OF SCIENCE IN ENGINEERING

THE UNIVERSITY OF TEXAS AT AUSTIN

May, 1988

Acknowledgments

I wish to place on record, my sincere thanks to my supervising professor, Dr. J.C.Browne for his guidance, encouragement and support. I would also like to acknowledge Prof. B.F.Womack for co-supervising this thesis. I would also like to thank members of the **Parallel Programming Group**, especially Dr. Ashok Adiga for his valuable guidance and help. I am grateful to my wife, Ruma, and my parents for their encouragement throughout my education.

SIVAGNANAM RAMASUNDARAM EASWAR

The University of Texas at Austin

May, 1988

Abstract

This thesis provides a first step towards resolution of the problem, of converting sequential Fortran programs to parallel, by capturing the potential parallel computation structure of a Fortran program in a Relational Database. Parallel languages are required to fully utilize the Parallel machines that have been developed. Many Man-years of Sequential Programs (in FORTRAN) have already been written. Re-writing these programs in some parallel language would be almost impossible. The Database produced by this thesis can then be used by other programs, to generate specific parallel computation structures, appropriate for given environments.

Table of Contents

List of Figures

Chapter 1

Introduction

The topic of Parallel Processing refers not only to parallel machines and software systems which operate on them, but also to the organization of computations which are to be executed in parallel. The development of computer architecture with powerful parallel processing units has spawned an interest in languages that permit the explicit specifications of parallel operations. Parallel languages have been developed to assist programmers in writing high level programs that fully utilize the (parallel) hardware.

Many millions of lines of Fortran code have been written without the benefit of explicit parallel operations. The question now is, how to efficiently convert the existing code to run on these machines without having to rewrite the entire code in some new parallel language. This thesis provides a first step towards resolution of the problem, by capturing the potential parallel computation structure of a Fortran program in a Relational Database. This Database can then be used by other programs, to generate specific parallel computation structures, appropriate for given environments.

1.1 Outline

1.1.1 Computation Model and Dependency Graph Concepts

Dependency Graphs are a basis for compiler optimizations and recognition of parallelism in programs [KUC 77] [BRO 85]. An extended form of the Dependency graphs is used in this thesis as the basis for capturing the parallel structure inherent in a Fortran program.

The computation model on which this thesis is based, is the one developed by J.C.Browne [BRO 85]. This model consists of a directed graph, where the nodes represent Schedulable Units of Computations (SUCs) and the arcs represent the dependencies between SUCs. Execution of the computation is obtained by traversal of the graph along the paths defined by the dependency relationships associated with the arcs. A SUC is characterized by its functionality and state. It may have one or more initial states, a sequence of active states and a final state.

The granularity of the SUCs may vary from single instructions to subroutines or functions. The granularity chosen here will be such that the time required to create the SUC will be much lesser than the time to execute the SUC.

Dependencies are relations among SUCs. There are different types of dependencies : Data Dependencies, Mutual Exclusion Dependencies and Control Dependencies.

A Data Dependency exists between two SUCS, if one SUC needs a value from another SUC to reach a valid state for execution. Mutual Exclusion Dependencies occur when two SUCs access common data, and can do so in any order as long as their execution does not overlap. Control Dependencies occur when one SUC has to execute only after some other SUC.

One advantage of this model is that it is inherently hierarchical. Any computation can be defined as a single SUC. The SUC can be decomposed into a subgraph to allow specification of finer details. On the other hand, a subgraph can be replaced with a SUC to allow a higher level of abstraction. Refer to section 1.1.2 for more details.

Another advantage of this model is that of portability. There is a clean separation among computations and dependency relations. This clean separation allows each to be separately resolved and mapped.

1.1.2 Dependency Graphs For Fortran Programs

A Fortran program defines a family of dependency graphs at different levels of resolution. Fortran, however, as is the case for other sequential higher level languages, imposes constraints on the execution of the program by adding control dependencies. Some of these control dependencies are not essential to correct execution. Parallel programs are attained by stripping away those control dependencies not required for a correct execution structure.

Individual statements are the smallest units which will be stored in the database and considered as potential SUCs. This level of granularity for SUCs creates an enormous dependency graph and an enormous scheduling problem. It is, however, the level at which restructuring compilers typically work.

The next largest level of granularity is the single-entry/single-exit blocks(SE/SE) of statements [HECHT]. Modules can be decomposed by known algorithms into single-entry/single-exit blocks. Modules (subroutines and functions) are especially important cases of single-entry/single-exit blocks. They are, in effect, single-entry/single-exit blocks which are given global names and can be recognized across the program structure. It may often be useful to compose modules linked by calls into a single executable unit. The largest possible granularity is the total program itself.

The representation of dependency graphs to be captured in the database of this thesis will be able to support formulation of dependency graphs for Fortran programs across all of these levels of granularity.

The dependencies to be captured in the data base are data dependencies where one SUC generates inputs which are required by another SUC, mutual exclusion dependencies, where data is used by several SUCs in no particular order and certain essential control dependencies which cannot be deleted and still maintain correct execution of the original Fortran program. (We will have to keep all control dependencies since we cannot delete them without analysis). These dependencies will be resolved down to the statement level so that they can be utilized in synthesis of schedulable units of execution at higher levels.

1.2 The Problem

Fortran, as is the case for other sequential higher level programming languages, imposes non-essential constraints on the execution of the program by adding control dependencies. Conversion to parallel computation structures (conversion of the sequential form imposed by the total order) of Fortran programs could be easily accomplished by simply deleting the control dependencies from the dependency graph, were it not for the fact that programs are generally written in forms which implicitly require the sequential control structure for correct execution. It is this implicit dependence on a particular order of execution which renders it awkward and difficult to do complete and total restructuring of Fortran programs. Human input and expertise is often required to determine what is essential and what is non-essential.

1.3 The Goal

The goal of this thesis will be to be able to support formation of dependency graphs across several levels of granularity, but with a focus on SUCs whose execution time will be much greater than the overhead of initialization and scheduling of the SUC.

The database will be structured to support the creation of programs which utilize the information stored in the database to create a dependency graph under supervision of a user who understands the program. The SUCs so created will generally be of sufficient size that they are effective subjects for application of the optimizing compilers, which are very effective at lower levels of granularity.

1.4 The Approach

This thesis will specify and implement a system which will capture the potential parallel computation structures of a sequential Fortran program (the full dependency graph), and will store all of the relevant information in a relational database. The database will describe the program on a line-by-line and module-by-module basis. It will explicitly contain the relationships between modules and the relationships between statements within modules. With this information there can be extracted a dependency graph for the entire program at variable levels of granularity.

There will also be defined auxiliary constructs in the database which will be needed by the analysis programs. These include definition of the call graph and definition of the dependency graph which results from application of the restructuring functions.

1.5 Construction of Dependency Graphs

The construction of dependency graphs, although not a direct purpose of the thesis, may proceed as follows.

1. The program dependency graph at module and statement levels will be generated.
2. The call graph, which will be used to guide subsequent steps, will then be generated.
3. The main program will be examined for invocations of modules and for loops which contain invocations of modules.

The expected result of this thesis will be a database which will effectively support the application of these functions to produced dependency graphs and to manipulate dependency graphs.

1.6 Organization of The Thesis

The logical(schema) and physical Database design are presented in *chapter 2*. *chapter 3* describes the Lexer and the Parser and describes how the parser is used to analyze Fortran programs, how data is extracted and mapped to the Database. The final chapter illustrates the ability of the Database to support the functions which have been defined above.

Chapter 2

The Database

The *Relational* data model [COD 70] represents the Database as a collection of tables each of which has a unique name. A row in a table represents a relationship among a set of values. Since a table is a collection of such relationships, there is a close correspondence between the concept of a table and the mathematical concept of a relation, from which the *Relational Data Model* takes its name.

2.1 SunUnify

The database chosen to implement this thesis is SunUnify [SUN]. It is a commercially available Database that is distributed by SUN MICROSYSTEMS. The *SunUnify Database Management System* is a powerful, general purpose package that simplifies record keeping tasks, organizes information, cross references data in ways that would be difficult to do manually. The reason for choosing SunUnify is because it provides a number of convenient facilities to perform a variety of functions. Some of these functions are:

- Ad Hoc user queries and updates in an English like Language.
- Fast data access from programmed applications.
- Logical data integrity checking.
- Database load, dump.

- Use of SUN's window management system to provide different views of the database.

The SunUnify Database package contains several tools. Some of the tools that were used extensively are described below.

2.1.1 Design and Create a New Database

This tool has two different phases:

- Database Design
- Database Create

The design phase includes designing record types and their associated fields. The Database Create phase has the following set of application development options:

- Print a report of the Database Design.
- Create an empty Database using the new design.
- Create data entry screens.
- Create a menu.

2.1.2 Reconfigure Database

This tool is part of the Database Design Utilities. It is used when the Database design is modified, or when the size of the expected number of records in a relation is to be increased beyond a previously defined limit or when B-Tree indexes are added or dropped. Care must be taken to make a backup of the Database in case of a hardware or software failure.

When executed, the tool prompts the user to rebuilt the hash table index. This index has to be rebuild if the total expected number of records in the Database has increased.

2.1.3 SQL

SQL (Structured Query Language) was introduced as the query language for System R [CHA 76]. It is an English-keyword-based query language that is powerful and flexible. SQL uses a combination of *Relational Algebra* and *Relational Calculus* constructs. The basic structure of an SQL expression consists of three clauses:

- **Select** corresponds to the projection operation in relational algebra.
- **From** gives the list of relations to be scanned.
- **Where** corresponds to the selection predicate of relational algebra.

Chapter 4 provides numerous examples where SQL is used to query the Database.

2.1.4 Database Load

dbload is a program for loading data, schema information or B-tree information, in ASCII format, into the Database. The text files need to be in a specific format. Appendix A contains an example with the correct format. The advantage of using dbload instead of SQL to load data into the Database is that it is much faster.

2.2 The Database Schema

The Database schema consists of eight relations. The names of the relations and a brief description of each follows. Refer to figure 2.1 for the Database schema . A detailed description of the schema is included in Appendix A.

2.2.1 MOD_INFO

This relation consists of two fields. The first field contains the module number . This is a unique number assigned to each module as it appears in the program. The values in this field are unique and hence this is the *Key* field. The second field contains the name of each module.

2.2.2 MOD_INPU

MOD_INPU has four fields. The first field contains the module number and the second field, the name of an input parameter. Since a module can have several input parameters, the first field alone cannot make up the key, but the two fields together can guarantee a unique record, and hence make up the *Key* field. Field three gives the variable type (integer, real, etc.) for the parameter in field two and field four provides information as to whether or not the parameter is modified in the module. This information is useful when questions regarding duplication of a SUC arises.

2.2.3 CAL_INFO

CAL_INFO consists of five fields. They are, the Calling Module, Calling Line, Called Module, Parameter number, Parameter. The first four

MOD_INFO

Mod_number	Mod_name

MOD_INPU

Mod_number	Input_param	Param_type	Read_write

CAL_INFO

Calling_mod	Calling_lin	Called_mod	Param_num	Parameter

SUCC_LIN

Mod_number	Line_number	Succ_mod	Succ_line

LINE_DES

Mod_number	Line_number	Line_descr

COM_BLOK

Mod_number	Common_name	Var_name	Read_write

VAR_INFO

Mod_number	Line_number	Var_name	Var_type	Read_write

DO_LOOPS

Mod_number	Start_line	End_line	Label

Figure 2.1: The Database Schema

fields together make up the *Key*. This relation contains information such as the number of calls made to a module from another module, or the calls made from within Do Loops, or the Parameters passed to a module, etc. To generate a **Call Graph** for the program, SQL can be used to query this relation and generate unique values for fields one and three.

2.2.4 SUCC_LIN

This relation consists of four fields, all of which together make up the *Key*. The four fields are Module number, Line number, Successor module number and Successor line number. This relation generates the *Control Flow* graph for the entire program. For any module number and line number, the successor module number(s) and line number(s) are provided. In addition to this, information is also available concerning the predecessor module(s) and line(s) . This can be achieved by using SQL to generate all the records of this relation for particular values in fields three and four.

2.2.5 LINE_DES

LINE_DES is made up of three fields, the first two of which make up the *Key*. The fields are Module number, Line number and Line description. The line numbers in the Database correspond to all the non-comment and non-blank lines in the program. The reason for doing this is because of speed and memory constraints. A detailed explanation of this is included in the Parser section of Chapter 3.

2.2.6 COM_BLOK

COM_BLOK consists of four fields, the first three of which make up the *Key*. Field one generates the module number in which the common statement appears. Field two gives the name of the common block. In a labeled common statement, this field contains the actual name of the block and in an unlabeled common statement, the name field contains *_COMMON*. The third field contains the name of each variable as it appears in the common statement. Field four provides information as to if any of the variables are modified in the module.

2.2.7 VAR_INFO

VAR_INFO consists of five fields, the first three of which make up the *Key*. Field one contains the module number, field two has the line number and field three has the name of the variable. Field four has the variable type and field five describes whether the variable was modified or not in this occurrence. This relation is used to create the **Dependency Graph** at the statement level.

2.2.8 DO_LOOPS

This relation contains four fields. Field one contains the module number. Field two and field three contain the starting and ending line numbers of a *Do Loop*. Fields one, two and three together make up the *Key*. Field four contains the label that the Do loop references. This relation, when joined with CAL_INFO, generates information about modules that are invoked from within loops.

2.3 Physical Database Design

SunUnify supports four different access methods. They are:

1. Hashing
2. Explicit relationships
3. B-Trees
4. Buffered sequential access

Each of these access methods is designed for a different kind of data retrieval operation. Hashing is used when records are to be accessed in a random fashion by supplying an exact key. Explicit relationships are used when there is a need to join tables that were split apart as a result of normalization. B-Trees are used when the queries concern ranges of values, or partial, inexact matching. Buffered sequential access is most efficient when all the records of a given table need to be accessed, starting at the first one and proceeding one-by-one to the last.

2.3.1 B-Trees

The access method chosen to implement this thesis is the B-Tree method. B-Trees are always balanced, so every search takes the same amount of time. Also, the number of Disk accesses, and hence the search time, required to find an entry rises by a factor of $Log\ N$ as the index gets larger. Finally, B-Trees reorganize themselves dynamically, so their performance stays constant even after many additions and deletions.

The advantages of using B-Trees are as follows:

- B-Trees permit ordered access to all records of a given type, based on the value of the indexed field. This thesis requires large numbers of records be accessed very rapidly in sorted order. B-Trees are ideal for this application.
- B-Trees can be added or dropped without reconfiguring the Database.
- B-Trees can be used on any field to create a secondary field. This feature has been used extensively .

Chapter 3

The NAG TOOLPACK

The Lexer and Parser used to implement this thesis are part of the NAG(National Algorithms Group) Toolpack/1, which is a collection of software tools to perform various types of analysis on Fortran programs [COH 84]. The Fortran Source Code is passed through the Lexer, and the output from here is sent to the Parser. The Parser produces a *symbol table* and a *parsed tree* of the program. An interface program was written that picked up information from these files and stored it in the Database. Refer to figure 3.1.

3.1 The Lexer

ISTLX [ILES] is a Fortran 77 scanner that converts Fortran 77 source text to a token stream and detects and reports lexical errors. The scanner has been mechanically generated from a specification of the Fortran 77 language. The target language accepted, and a definition of the grammar, are given in Appendix A. ISTLX reads Fortran 77 source text from the source file (parameter:1). The different parameters are listed on page 18. The resulting token stream is placed in the token file (parameter:3) and the comments are placed in the comment file (parameter:4). Any errors discovered are reported to the optional list file (parameter:2) and an attempt is made to continue scanning by deleting or adding tokens. During operation the scanner

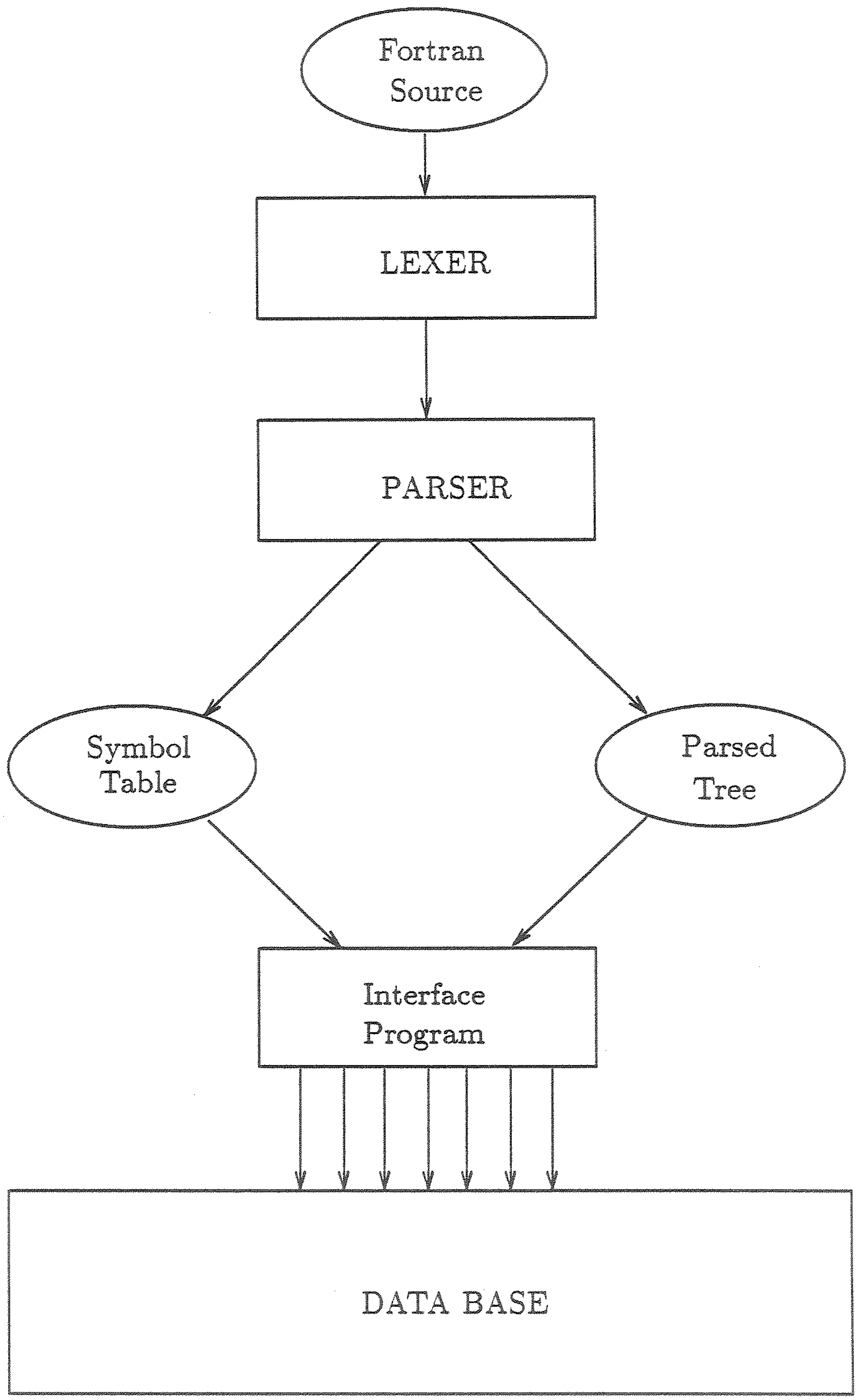

Figure 3.1: Schematic layout of the Thesis

optionally produces a list file which contains the input source text preceded by the token number of the first token for each statement. If no list file is required (producing a list file does slow the scanner down) then parameter 2 should be set to - .

Parameter 1: Name of Source File.

Parameter 2: Name of List File.

Parameter 3: Name of Token File or Files.

Parameter 4: Name of Comment File or Files.

The scanner may be instructed to place the tokens and comments for each program unit in a separate file. To do this the token and comment file names should each be placed in parentheses. If either the token or comment file name is in parentheses then both must be. The file name in parentheses is used as a base for a set of file names, one per program unit. The scanner accepts Fortran 77 standard conforming software. All errors are reported to the list file. The statement and token number when the error occurred are also reported. This can be related back to the source code using the token numbers given in the list file. The values at the start of each statement in the list file are the statement number and the number of the first token in that statement. Errors are as follows:

1. Token too long.
2. Error in token.
3. Error in token to be screened.
4. Unprocessed text remaining in token to be screened.
5. Screen ended in error.

6. Scan ended in error.

7. Screened token ends unexpectedly.

Fatal errors are reported separately.

3.2 The Parser

ISTYP parses a Fortran-77 program. It takes as its input a token stream produced by ISTLX and produces a parse tree, symbol table and comment index. ISTYP is a table-driven parser generated using the YACC [JOH 78] parser-generator. All error and warning messages produced by ISTYP are written both to the standard error channel and the symbol table file. When a tool which uses the symbol table is executed, these warning and error messages are displayed again. As many error conditions render at least part of the symbol table or parse tree information invalid, it is important that the user is aware of the possibility that further processing may be completely useless.

Parameter 1: Name of token stream file

Parameter 2: Name of comment stream file

Parameter 3: Name of parse tree file

Parameter 4: Name of symbol table file

Parameter 5: Name of comment index file.

ISTYP parses the standard Fortran-77 language with the Hollerith extension and some additional data types including DOUBLE COMPLEX. It will accept all legal Fortran-77 programs and reject most syntactically incorrect programs. The semantic routines which produce the symbol table do a modest amount of semantic checking, but were designed primarily to generate correct symbol information for correct programs, not for checking a program's correctness. This means that even when ISTYP detects an inconsistency in the use of a symbol it may not produce a very informative error message.

3.2.1 The Parse Tree

The parse tree is organized recursively as a list of lists. All the subnodes of a node are grouped into a doubly-linked linear list with owner pointers.

Thus each node in the parse tree has four pointers: Up, Down, Next and Prev. The up pointer of the root of the tree points to itself; an up pointer is only zero when a node is a "deleted" node, or orphan. Orphan nodes only exist temporarily within the parse tree during the building operation or during modification; the parser always links them into the parse tree. The Next pointers form a chain of subnodes of a single node, from the first to the last. The Next pointer of the last node in the chain is zero. The Prev pointers form a circular list of the subnodes of a single parent node, the last node in the chain can be simply found be going "prev" from the first node. The Down pointer of a node points to the first subnode in its subnode list. A leaf node has either a zero Down pointer, or a negative Down pointer. A negative Down pointer is a pointer into the symbol table (for N_NAME, N_CBLK_NAME, N_LABEL and N_LABELREF nodes) or into the string table (for other leaf nodes N_ICONST).

The program

PROGRAM MAIN

K = 5 + 6

STOP

END

generates a tree as shown in figure 3.2. The numbers in each node, are listed in Appendix B, under the section YNODES.

The structure of the parse tree is detailed by listing the possible nodes which may be subnodes of any particular node type. For example, when traversing the parse tree, if a Node of type *Do* is reached, it will have children of type N_LABELREF and N_DOSPEC. N_LABELREF is a leaf node, with a pointer into the symbol table and N_DOSPEC has children of type N_NAME and three arithmetic expressions. This information is specified in the listing below. Node types have the form "N_XXXX", where XXXX consists of uppercase alphabetic characters and underlines. Macros for these node types are defined in the macro file YNODES.

```
In the following listing:
- parentheses indicate grouping,
- vertical lines indicate alternatives,
- asterisks indicate closure (i.e. the previous item occurs
  zero or more times),
- plus  signs  indicate  positive closure (i.e. the previous
  item occurs one or more times)
- question marks indicate the previous item is optional,
- /* and */ delimit comments.  Token types are  those  listed
```

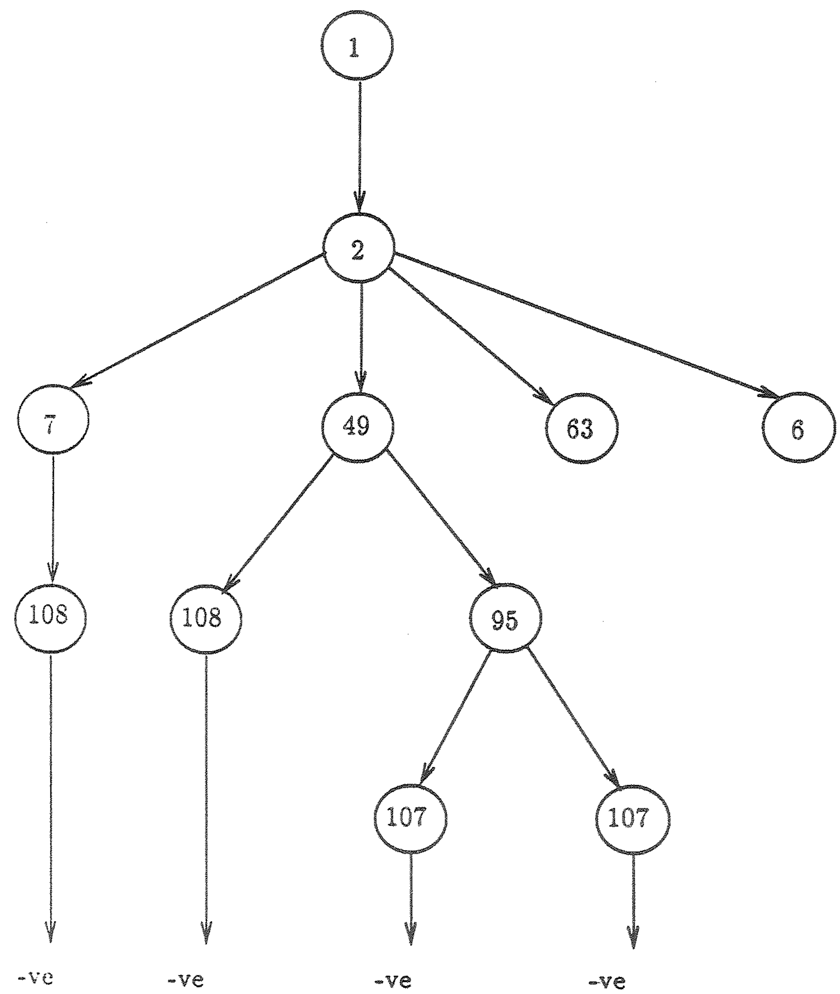

Figure 3.2: Sample Parse Tree

in the ISTLX documentation, and have the form TXXXXX where XXXXX is up to five letters in upper case.

```
N_ROOT       : (N_MAIN | N_F_SUBP | N_S_SUBP | N_BD_SUBP)+

N_MAIN       : N_PROGRAM? Statement* N_END
                /* Main program */
N_F_SUBP     : N_FUNCTION Statement* N_END

N_S_SUBP     : N_SUBROUTINE Statement* N_END

N_BD_SUBP    : N_BLOCKDATA Statement* N_END

N_PROGRAM    : N_NAME

N_FUNCTION   : Datatype? N_NAME N_LIST?

N_SUBROUTINE: N_NAME N_LIST?

N_BLOCKDATA : N_NAME?

N_LIST       : N_NAME+
/* function */
N_LIST       : (N_NAME | N_ASTERISK)+
/* subroutine */
N_END        : N_LABEL?
```

```
Datatype     : N_INTEGER | N_REAL | N_DOUBLE_P | N_COMPLEX |
               N_LOGICAL | N_CHARACTER | N_DCMPLX

N_DOUBLE_P, N_DCMPLX /* leaf nodes with no information */

N_CHARACTER : (Arithmetic_expression | N_ASTERISK)?

N_INTEGER, N_REAL, N_COMPLEX, N_LOGICAL: N_ICONST?

N_NAME /* leaf node, pointer into symbol table */

N_LABEL /* leaf node, pointer into symbol table */

Statement : N_FORMAT | N_ENTRY | N_PARAMETER | N_IMPLICIT |
            N_DATA | N_DIMENSION | N_EQUIV | N_COMMON |
            N_TYPE | N_EXTERNAL | N_INTRINSIC | N_SAVE |
            N_DO | N_LOG_IF | N_BLOCKIF | N_ELSE |
    N_ELSEIF | N_ENDIF | N_ARITHIF | N_ASGN |
    N_ASSIGN | N_STMT_FN | N_GOTO | N_STOP |
    N_PAUSE | N_READ | N_WRITE | N_PRINT | N_REWIND |
    N_BACKSPACE | N_ENDFILE | N_OPEN | N_CLOSE |
    N_INQUIRE | N_CALL | N_RETURN

N_DIMENSION: N_ARR_DECL+

N_ARR_DECL : N_NAME (N_ARDIM+ N_DARDIM? | N_DARDIM)
```

```
N_ARDIM    : Arithmetic_expression? Arithmetic_expression

N_DARDIM   : Arithmetic_expression?

N_EQUIV    : N_EQVSET+

N_EQVSET   : (N_NAME | N_ARELM | N_SUBSTR)+

N_COMMON   : (N_BLNKCM | N_LBLDCM)+

N_BLNKCM   : N_CBITEMS

N_LBLDCM   : N_CBLK_NAME N_CBITEMS

N_CBITEMS  : (N_NAME | N_ARR_DECL)+

N_TYPE     : Datatype (N_NAME | N_ARR_DECL | N_CHAR_LEN)+

N_CHAR_LEN : (N_NAME | N_ARR_DECL) (Arithmetic_expression |
     N_ASTERISK)

N_IMPLICIT : N_IMPL_DECL+

N_IMPL_DECL: Datatype N_CHRRNG+

N_CHRRNG   : N_IMPCHAR N_IMPCHAR?
```

```
N_IMPCHAR /* leaf node with pointer into string table */

N_PARAMETER: N_PARA_DECL+

N_PARA_DECL: N_NAME expression

N_EXTERNAL : N_NAME+

N_INTRINSIC: N_NAME+

N_SAVE     : (N_NAME | N_CBLK_NAME)+

N_CBLK_NAME /* leaf node with pointer into symbol table */

N_DATA     : N_DATA_DECL+

N_DATA_DECL: N_DATA_ITEMS N_DATA_VALS

N_DATA_ITEMS: (N_NAME | N_ARELM | N_SUBSTR | N_DATA_IMPDO)+

N_DATA_VALS: (N_MULT_VAL | N_NEG | Data_constant)+

N_MULT_VAL : (N_NAME | N_ICONST) (N_NEG | Data_constant)

Data_constant : N_ICONST | N_RCONST | N_DPCONST | N_SCONST |
                N_LCONST | N_HCONST
```

```
N_ARELM     : N_NAME expression+

N_SUBSTR    : (N_NAME | N_ARELM) N_SSSPEC

N_SSSPEC    : (N_DEFAULT | Arithmetic_expression)
              (N_DEFAULT | Arithmetic_expression)

N_DEFAULT /* leaf node */

N_DATA_IMPDO: (N_ARELM | N_DATA_IMPDO)+ N_DOSPEC

N_DOSPEC : N_NAME Arithmetic_expression Arithmetic_expression
           Arithmetic_expression?

N_ENTRY : N_NAME N_LIST?

N_ASGN : (N_ARELM | N_SUBSTR | N_NAME) expression

N_ASSIGN : N_LABELREF N_NAME

N_LABELREF /* leaf node with pointer into symbol table */

N_STMT_FN : N_NAME N_LIST expression

N_LIST (statement function) : expression+
```

```
N_GOTO : N_LABELREF

N_CMGOTO : N_LABELLIST Arithmetic_expression

N_ASGOTO : N_NAME N_LABELLIST?

N_LABELLIST : N_LABELREF+

N_ARITHIF : expression N_LABELREF N_LABELREF N_LABELREF

N_LOG_IF : expression Statement
/* this occurrence of "Statement" will never have a label */

N_BLOCKIF : expression

N_ELSEIF : expression

N_ELSE, N_ENDIF /* leaf nodes */

N_DO : N_LABELREF N_DOSPEC

N_CONTINUE /* leaf node */

N_STOP, N_PAUSE : (N_ICONST | N_SCONST)?

N_WRITE : N_CILIST (expression | N_IOIMDL)*
```

```
N_IOIMDL /* write and print */ : (expression | N_IOIMDL)+
 N_DQSPEC

N_CILIST : N_UNITID? (N_FMTID | N_CIITEM)*

N_UNITID : expression | N_ASTERISK

N_FMTID : N_LABELREF | N_ASTERISK | expression

N_CIITEM : (N_IOKW (expression | N_ASTERISK)) |
           ((N_ERRKW | N_ENDKW) (expression | N_ASTERISK |
   N_LABELREF))

N_READ : ((N_FMTID | N_CILIST) (N_NAME | N_ARELM |
   N_IOIMDL)*) | N_AMBIGUOUS

N_IOIMDL /* read */ : (N_NAME | N_ARELM | N_IOIMDL)+
N_DOSPEC

N_PRINT : N_FMTID (expression | N_IOIMDL)*

N_OPEN, N_CLOSE, N_INQUIRE : N_CILIST

N_CILIST /* open close inquire */ : (N_UNITID | N_CIITEM)
 N_CIITEM*

N_BACKSPACE, N_ENDFILE, N_REWIND : N_UNITID | N_CILIST
```

```
N_CILIST : (N_UNITID | N_CIITEM) N_CIITEM*

N_FORMAT : (N_FMTFLD | N_SCONST | N_HCONST | N_SLASH |
    N_SUBFMT | N_COLON | N_REPEAT | N_SCALE)*

N_SUBFMT : ( /* same as for N_FORMAT */ )+

N_FMTFLD, N_SCALE  /* leaf nodes with text pointers */

N_CALL : N_NAME (expression | N_LABELREF)*

N_RETURN : Arithmetic_expression?

expression : N_EQV | N_NEQV | N_OR | N_AND | N_NOT | N_LT |
     N_LE | N_GT | N_GE | N_EQ | N_NE | N_CONCAT |
     N_SCONST | N_HCONST | N_LCONST | N_SUBSTR |
     Arithmetic_expression

Arithmetic_expression : N_PLUS | N_MINUS | N_POS | N_NEG |
N_MULTIPLY | N_DIVIDE | N_EXPONT |
N_NAME | N_ARELM | N_FUNREF | N_SPAREN
| N_ICONST | N_RCONST | N_DPCONST |
N_CCONST

N_EQV, N_NEQV, N_OR, N_AND, N_CONCAT : expression expression
```

```
N_NOT : expression

N_LT, N_LE, N_GT, N_GE, N_EQ, N_NE : Arithmetic_expression
                                     Arithmetic_expression

N_POS, N_NEG : Arithmetic_expression

N_PLUS, N_MINUS, N_MULTIPLY, N_DIVIDE, N_EXPONT :
        Arithmetic_expression Arithmetic_expression

N_SPAREN : expression
    /* This is a parenthesised expression */

N_FUNREF : N_NAME expression*

N_ICONST, N_RCONST, N_LCONST, N_DPCONST, N_SCONST, N_HCONST
    /* leaf nodes with pointers into the string table */

N_CCONST : (expression | N_IOIMDL) (N_NEG | N_RCONST |
    N_ICONST | N_DPCONST)

N_NEG : N_RCONST | N_ICONST | N_DPCONST
```

3.2.2 The Symbol Table

The symbol table consists of two parts: *the string table*, which contains the text of a symbol, and *the symbol table* proper [ISTYP]. Refer to Appendix B for a sample *Symbol Table*. Constants do not have a symbol associated with them; these are simply stored in the string table, and a pointer to the string table is stored in the node for these items in the parse tree. A symbol consists of three fixed fields, and up to five additional fields. The additional fields are called attributes, and vary according to the type of the symbol. The three fixed fields uniquely identify each symbol, and are:

- SYMBOL_TYPE. This field contains the type of symbol, e.g. common block name, label, variable, etc.
- SYMBOL_NAME. This field contains a pointer into the string table to the textual representation of the symbol.
- SYMBOL_PUN . This field contains the program unit number within the file in which the symbol appears.

The next five fields depend on the Symbol types used. The different Symbol types are S_LABEL, S_COMMON, S_NAME, S_PU, S_VAR, S_PARAM, S_PROC, S_SF, S_ENTRY.

The symbol type S_LABEL has these attributes:

1. LABEL_DEFN. This field contains a pointer to the top node of the statement which is labelled with this label.
2. LABEL_CF_REF. This field contains the number of control-flow references to that label.

3. LABEL_DO_REF. This field contains the number of DO-loops (ASSIGN statements) which reference the label.

4. LABEL_IO_REF. This field contains the number of i/o-statements which reference this label as a format-identifier.

5. LABEL_SCOPE. This field contains the node number of the innermost enclosing DO, IF-THEN, ELSEIF, or ELSE statement which contains the label. If the label is referenced but not defined, this field will contain the node number where the label was first referenced.

The symbol type S_COMMON has one attribute.

1. COMMON_DEFN. This field contains a pointer to the N_LBLDCM or N_BLNKCM node which has the first occurrence of that common block. For blank (unlabelled) common, the symbol name is $COMMON.

The symbol type S_NAME is a temporary symbol type which is usually changed to another type once the full meaning of the symbol is known. If it has not been changed, it means that the symbol has not been referenced in the program-unit apart from its defining occurrence in a type statement. All the following symbol types include the attributes of this symbol.

1. NAME_DTYPE. This field contains a small integer which specifies the base data type of the name. The possible values are listed in the appendix.

2. NAME_CHRLEN. This field contains a value which specifies the length of the character string for character data types. It is zero for all other data types.

3. NAME_STATUS. This field contains a number of status bits which describe the specific occurrences of the symbol in the program-unit. The bits which may be set by ISTYP are detailed below.

   - DECL_EXTERNL: The name appears in an EXTERNAL statement.
   - DECL_INTRINS: The name appears in an INTRINSIC statement.
   - FORMAL_PARAM: The name is a formal parameter (dummy argument) of the program unit.
   - EXPLICIT_TYP: The name appears in a type statement, or if it is a function subprogram name, has the type specified in the FUNCTION statement.
   - IN_ASSIGN: The name appears in an ASSIGN statement.
   - ASSIGNED_TO: The name appears on the left-hand side of an assignment statement.
   - IN_READ_LIST: The name appears in the input-list of a READ statement.
   - IN_DATA_STMT: The name appears in a DATA statement.
   - STMT_FN_PARA: The name is a formal parameter (dummy argument) of a statement function.
   - IN_EQUIV: The name appears in an EQUIVALENCE statement.
   - IN_COMMON: The name appears in a COMMON statement.
   - USED_AS_ARG: The name is used as the actual argument to a called function or subroutine.
   - STD_INTRINSIC: The name is that of a standard intrinsic function.
   - FUN_CALLED: The name is called as a function.

- IN_EXPR: The name appears in an expression.
- SUB_CALLED: The name is called as a subroutine.
- DOLOOP_INDEX: The name is used as the controlling variable in a DO statement or implicit DO-loop.
- USE_BITS: This macro is actually the inclusive or of the bits: formal_param, in_ASSIGN, assigned_to, in_READ_list, in_DATA_st, stmt_fn_para, in_EQUIV, used_as_arg, fun_called, in_expr, sub_called and doloop_index.

The S_PU symbol type is for the program-unit itself. There is always exactly one S_PU symbol for each program-unit. If the program-unit is an unnamed main program, then the string pointer for the symbol will point to the string $MAIN. If it is an unnamed block data subprogram the string pointer will point to the string $BLOCKDATA.

There are no additional attributes for this symbol type beyond those of the S_NAME symbol type.

The S_VAR symbol type includes local, common and argument variables. There is one additional attribute:

1. VAR_ARR_DECL. This attribute is zero for a simple variable, and a pointer to the defining N_ARR_DECL (array_declarator) node for an array variable.

The S_PARAM symbol type has one additional attribute:

1. PARAMETER_DF. This attribute contains a pointer to the expression which defines the value of the parameter.

The S_PROC symbol type covers external functions, external subroutines and intrinsic functions. It has no additional attributes.

The S_SF symbol type has one addition attribute:

1. STMT_FN_DEFN. This contains a pointer to the N_STMT_FN node which defines the statement function.

The S_ENTRY symbol type has no additional attributes.

The symbol type S_PROC (subroutines and functions) is treated differently from other symbol types due to the complexity of deciding what data type it has. The attribute bits used to determine the data type are: fun_called, decl_externl decl_intrins, formal_param and used_as_arg.

3.3 Parser Interface to the Database

The *interface* program has three stages. In the first stage, the *string table*, the *symbol table* and the *parse tree* are read into memory. The relation MOD_INFO can be generated from the information present in the *symbol table*.

The second stage consists of assigning module numbers and line numbers to each node of the tree. This is done as follows: The *parse tree* is constructed in a manner such that the nodes that are present, one level below the root node of the tree, correspond to the modules in the program. For example, if a program has three modules in it, the root of the tree will have three children nodes.

The nodes that are present, one level below these, correspond to the individual line numbers of the modules. Refer to figure 3.3.

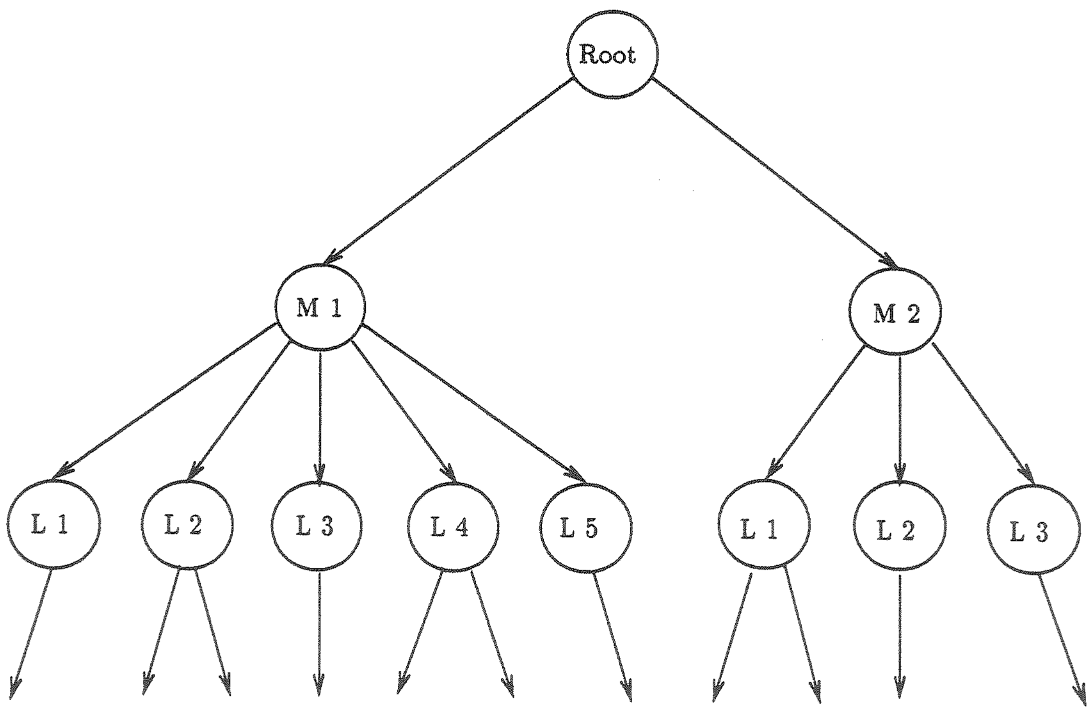

Figure 3.3: Module and Line numbers in the Parse Tree

Here M stands for module and L stands for line. The program in the figure contains two modules, containing five and three lines respectively.

The third and final stage consists of traversing the tree and extracting information to map to the Database. Two methods were used to access the data. To generate information for the relations DO_LOOPS, SUCC_LIN, MOD_INFO, and CAL_INFO, the tree was scanned as a flat file. When, for example, a node of type *DO* was accessed, the sub-tree under *DO* was processed to get the label value and the index used.

To generate information for the remainder of the relations, a pre-order traversal was done on the tree, till the occurrence of a particular node type. The sub-tree was then processed for the required information.

Chapter 4

Example Queries on the Database

This chapter presents a sample session with the Database. Several queries are presented here. Refer to Appendix B, for a sample program. The parse tree and the symbol table for this program are also provided. In addition to this, the data, that the interface program extracts from the tree, to map on to the Database is also shown.

4.1 Generating the Database

A shell program has been provided, that accepts a *FORTRAN* program as its input. The *FORTRAN* program is passed through the Lexer and the Parser, and the output files from the parser are fed to the interface program. The interface program then writes out a file called *Final.* This file is in the appropriate format to be loaded into the Database.

The shell program copies the file (*Final*) to the Database directory, creates and loads the data into a new Database, and then invokes SunUnify. The user can now bring up, either Databrowse, to view the data in the Database, or SQL, to query the Database.

The Parser, writes out warnings and error messages to the Symbol table file. These warning messages might appear for correct programs. *These messages must be removed, before the interface program can be called.*

4.2 Queries

The following queries first extract the *CALL GRAPH* from the program. Working with this information, a module level Dependency Graph can be obtained. Example 1 shows how a *call graph* can be extracted from the Database. The *call graph* shows that module one calls modules two, three and four at lines 8, 9 and 10 respectively.

4.2.1 Example: 1

```
sql> select unique Mod_number, Line_number, Called_mod
sql> from CAL_INFO/
recognized query!

Mod_number|Line_number|Called_mod
----------------------------------
         1|          8|         2
         1|          9|         3
         1|         10|         4
```

Example 2 shows the Parameters that are passed from module one to module two and Example 3 shows what module two does to its input Parameters. Two of the four parameters (P1 and M) are used to read data. This shows two input *Data Dependencies* into module two. The other two Parameters (E and SIZE) are modified in module two. This shows two output *Data Dependencies* from module two. Similarly, Examples 4–7 extract the *Data Dependencies* from the other modules.

4.2.2 Example: 2

```
sql> select Mod_number,Called_mod,Parameter_number,
sql> Parameter_passed from CAL_INFO where
sql> Mod_number = 1 and Line_number = 8/
recognized query!

Mod_number|Called_mod|Parameter_number|Parameter_passed
-------------------------------------------------------
         1|         2|               1|P1
         1|         2|               2|I
         1|         2|               3|E
         1|         2|               4|ESIZE
```

4.2.3 Example: 3

```
sql> select Mod_number,Input_param,Read_written
sql> from MOD_INPU where Mod_number = 2/
recognized query!

Mod_number|Input_param        |Read_written
---------------------------------------------
         2|P1                 |READ
         2|M                  |READ
         2|E                  |WRITE
         2|SIZE               |WRITE
```

4.2.4 Example: 4

```
sql> select Mod_number,Called_mod,Parameter_number,
sql> Parameter_passed from CAL_INFO where
sql> Mod_number = 1 and Line_number = 9/
recognized query!

Mod_number|Called_mod|Parameter_number|Parameter_passed
-------------------------------------------------------
         1|         3|               1|P2
         1|         3|               2|I
         1|         3|               3|O
         1|         3|               4|OSIZE
```

4.2.5 Example: 5

```
sql> select Mod_number,Input_param,Read_written
sql> from MOD_INPU where Mod_number = 3/
recognized query!

Mod_number|Input_param        |Read_written
---------------------------------------------
         3|P2                 |READ
         3|M                  |READ
         3|O                  |WRITE
         3|SIZE               |WRITE
```

4.2.6 Example: 6

```
sql> select Mod_number,Called_mod,Parameter_number,
sql> Parameter_passed from CAL_INFO where
sql> Mod_number = 1 and Line_number = 10/
recognized query!

Mod_number|Called_mod|Parameter_number|Parameter_passed
-------------------------------------------------------
         1|         4|               1|E
         1|         4|               2|O
         1|         4|               3|ESIZE
         1|         4|               4|OSIZE
```

4.2.7 Example: 7

```
sql> select Mod_number,Input_param,Read_written
sql> from MOD_INPU where Mod_number = 4/
recognized query!

Mod_number|Input_param         |Read_written
---------------------------------------------
         4|E                   |READ
         4|O                   |READ
         4|ESIZE               |READ
         4|OSIZE               |READ
```

Example 8 generates the *control flow* graph for module one. By using the information here, along with standard algorithms, any module can be split up into single-entry/single-exit blocks.

4.2.8 Example: 8

```
sql> select * from SUCC_LIN
sql> where Mod_number = 1
sql> order by Mod_number,Line_number asc /
recognized query!

Mod_number|Line_number|Successor_mod|Successor_line
----------------------------------------------------
         1|          1|            1|             2
         1|          2|            1|             3
         1|          3|            1|             4
         1|          4|            1|             5
         1|          4|            1|             8
         1|          5|            1|             6
         1|          6|            1|             7
         1|          7|            1|             4
         1|          8|            1|             9
         1|          8|            2|             1
         1|          9|            1|            10
         1|          9|            3|             1
         1|         10|            1|            11
         1|         10|            4|             1
         1|         11|            1|            12
```

The next example provides information as to what modules are available, their names and types (program, subroutines, functions, etc.). This information can be very useful, because, assumptions need to be made about those modules that are unavailable. This query provides the necessary information.

4.2.9 Example: 9

```
sql> select Mod_number,Mod_name,Line_description,
sql> Mod_avail_or_not from
sql> MOD_INFO,LINE_DES
sql> where
sql> MOD_INFO.Mod_number = LINE_DES.Mod_number
sql> and LINE_DES.Line_number = 1
sql> order by Mod_number asc /
recognized query!

Mod_number|Mod_name|Line_description|Mod_avail_or_not
-----------------------------------------------------

         1|GEN     |N_PROGRAM       |AVAILABLE
         2|EVEN    |N_SUBROUTINE    |AVAILABLE
         3|ODD     |N_SUBROUTINE    |AVAILABLE
         4|PRNT    |N_SUBROUTINE    |AVAILABLE
```

Example 10 shows all the variables that occur in the program, and the modules they occur in. Once the program is split up into single-entry/single-exit blocks, the information provided by this query, along with the Read-Write information, can be used to illustrate the *Data Dependencies* between the different blocks.

4.2.10 Example: 10

```
sql> select unique Mod_number,Variable_name
sql> from VAR_INFO
sql> /
recognized query!

Mod_number|Variable_name
--------------------------------
         1|I
         1|P1
         1|P2
         2|E
         2|I
         2|J
         2|P1
         2|SIZE
         2|TEMP
         2|TEMP1
         3|I
         3|J
         3|O
         3|P2
         3|SIZE
```

```
3|TEMP
3|TEMP1
```

The tool was also used on other larger programs. A 4000 line *FORTRAN* program was used as input to the tool. The time taken to parse the program and apply the information to the Database was approximately 20 minutes. The examples that follow, query the new Database. The program was too cumbersome to include here, but a copy (long_sample.f) is kept in the Database directory.

Example 11 provides information to extract the *Mutual Exclusion* dependencies from this program.

4.2.11 Example: 11

```
sql> select unique Mod_number,Common_name,Read_written
sql> from COM_BLOK
sql> where Mod_number < 25 /
recognized query!

Mod_number|Common_name         |Read_written
---------------------------------------------
         1|DEBUGC              |READ
         1|UBEAC               |READ
         1|USUBC               |READ
         4|IBEAC               |WRITE
         4|UBEAC               |READ
         4|USUBC               |READ
         4|USUBC               |WRITE
```

5	UBEAC	READ
5	UBEAC	WRITE
5	USUBC	READ
5	USUBC	WRITE
6	IBEAC	READ
6	IBEAC	WRITE
6	ISUBC	READ
6	UBEAC	READ
6	UBEAC	WRITE
6	USUBC	READ
12	DEBUGC	READ
13	DEBUGC	READ
14	DEBUGC	READ
14	REPLFC	READ
15	DEBUG	READ
24	DEBUGC	READ

The next example finds out the *Do Loops* in a module. This information will be used in Example 13 to find all *Calls* to modules from within a *Do Loop* .

4.2.12 Example: 12

```
sql> select * from DO_LOOPS
sql> where Mod_number = 1/
recognized query!

Mod_number|Line_num_start|Line_num_end|Index_used
-------------------------------------------------
         1|            59|         136|IYT
         1|            73|          75|I
```

4.2.13 Example: 13

```
sql> select unique Mod_number,Line_number,Called_mod,
sql> Parameter_passed,from CAL_INFO,DO_LOOPS
sql> where
sql> CAL_INFO.Mod_number = DO_LOOPS.Mod_number
sql> and CAL_INFO.Mod_number = 1
sql> and CAL_INFO.Line_number between
sql> (DO_LOOPS.Line_num_start + 1) and
sql> (DO_LOOPS.Line_num_end - 1) /
recognized query!
```

Mod_number	Line_number	Called_mod	Parameter_passed
1	66	64	MODE
1	66	64	_I_CONSTANT
1	75	22	IY
1	79	20	LUOUT
1	79	20	NX
1	79	20	X
1	79	20	_S_CONSTANT
1	85	4	BESTOP
1	85	4	BEX
1	85	4	IFLAG
1	85	4	ISTAB
1	85	4	IWORK
1	85	4	MAXNFE
1	85	4	MODE
1	85	4	NFE
1	85	4	NORMX
1	85	4	NORMY
1	85	4	NX
1	85	4	NY
1	85	4	SCALE
1	85	4	TOL
1	85	4	X
1	85	4	XBOUND
1	114	20	NX

```
 1|      114|       20|X
 1|      114|       20|_I_CONSTANT
 1|      114|       20|_S_CONSTANT
 1|      117|       20|FXSTAT
 1|      117|       20|_I_CONSTANT
 1|      117|       20|_S_CONSTANT
 1|      119|       20|EYSTAT
 1|      119|       20|_I_CONSTANT
 1|      119|       20|_S_CONSTANT
 1|      121|       20|CSTAT
 1|      121|       20|_I_CONSTANT
 1|      121|       20|_S_CONSTANT
 1|      127|       20|FXSTAT
 1|      127|       20|_I_CONSTANT
 1|      127|       20|_S_CONSTANT
 1|      129|       20|EYSTAT
 1|      129|       20|_I_CONSTANT
 1|      129|       20|_S_CONSTANT
 1|      131|       20|CSTAT
 1|      131|       20|_I_CONSTANT
 1|      131|       20|_S_CONSTANT
```

4.3 Interface to a Grahical Display

The *Call Graph* of the program (example 1), can be displayed and manipulated using IDeA [SRI 88]. IDeA (Interactive Dependency Graph Analyzer) is a general purpose graphical tool, used for the display and manipulation of the dependency graphs. Refer to figure 4.1 for the *Call Graph* of the sample program.

A statement level *Control Flow Graph* was also extracted from the Database, by quering the relations LINE_DES and SUCC_LIN. Refer to figure 4.2 for the *Control Flow Graph*.

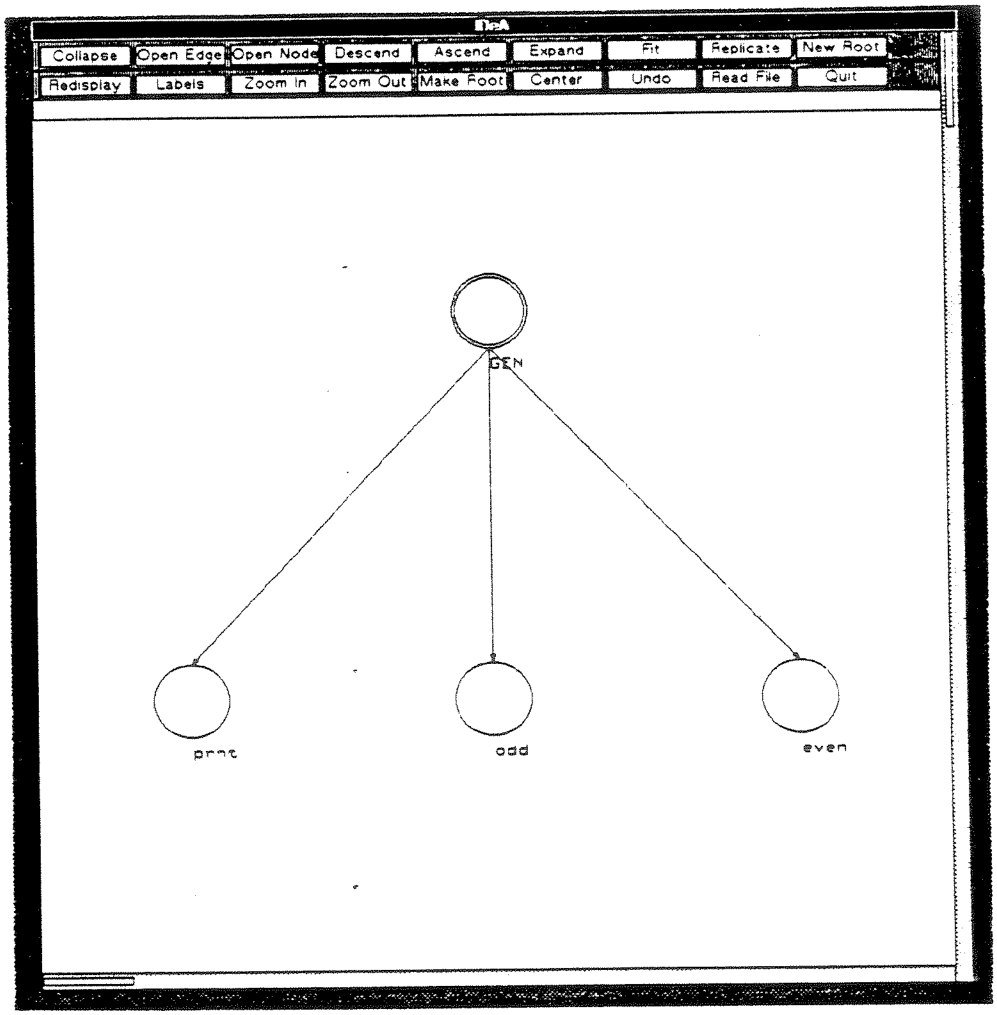

Figure 4.1: Call Graph of the Sample Program

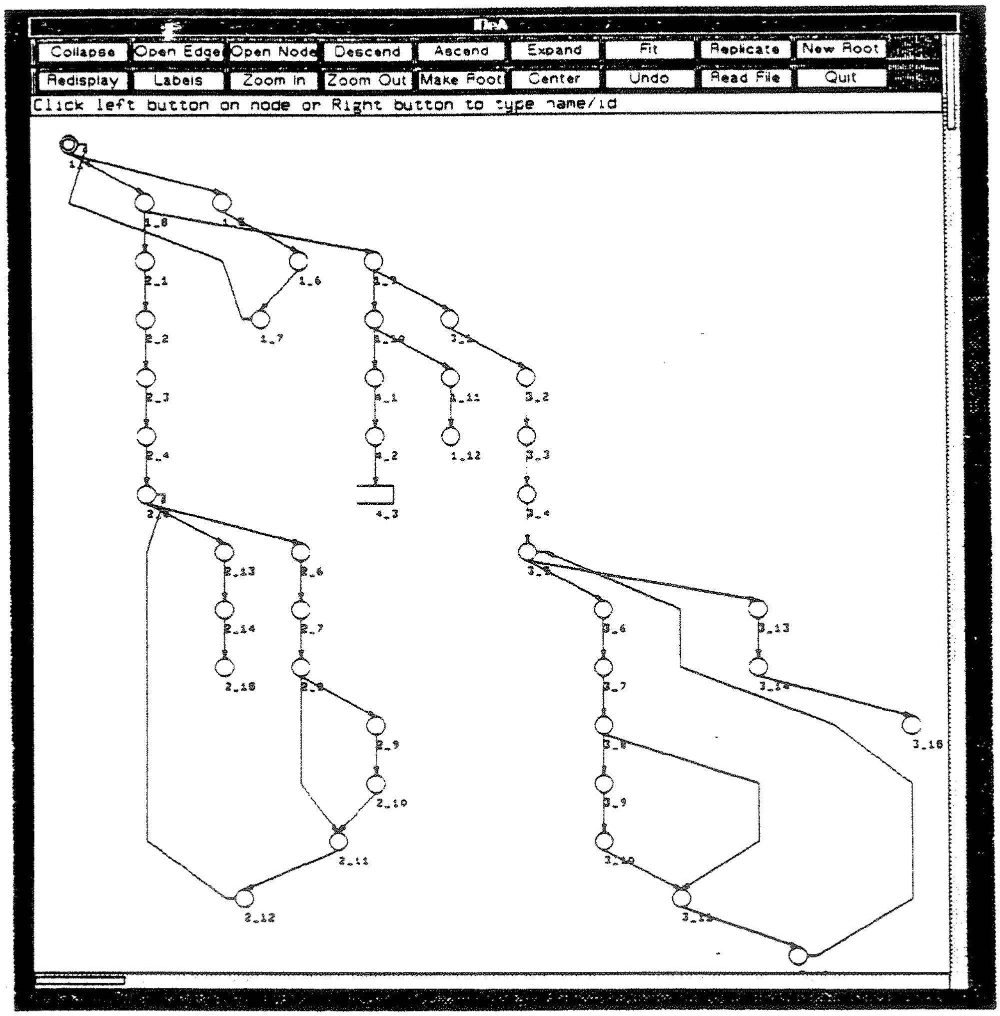

Figure 4.2: Control Graph of the Sample Program

Chapter 5

Conclusion

It has been established that the necessary elements for construction of full hierarchical dependency graphs for large Fortran programs can be captured and put in usable form through the use of standard commercial software elements. This thesis has utilized a lexer/parser combination taken from Toolpack and combined this with a commercial Relational Database system, the Unify system for SUN workstations, to capture the statement-level and module level dependency graph for Fortran programs.

This hierarchical dependency graph has been demonstrated to be an effective basis for analysis and understanding of the parallel structure implicit in programs in sequential languages. The database can serve as a basis for converting Fortran programs to parallel computational structures. One element of the conversion process, interface to a graphical display of control flow graphs, has been demonstrated.

Appendix A

A.1 Database Schema

```
{SunUNIFY}

MOD_INFO  MODULE_INFO         500

   *Mod_number            MO_mod                    Num     4

    Mod_name              MO_name                   Str    30

    Mod_avail_or_not      MO_avail                  Str    20



LINE_DES  LINE_DESCRIPTION    10000

   *Line_key              LI_key                    Comb   --

       ^Mod_number            LI_mod                Num     4

       ^Line_number           LI_line               Num     9

    Line_description      LI_desc                   Str    30
```

```
MOD_INPU  MODULE_INPUT        5000
   *Mod_inpu_key          MI_key                  Comb   --
       ^Mod_number            MI_mod              Num     4
       ^Input_param           MI_var              Str    20
    Var_type              MI_type                 Str    20
    Read_written          MI_r_w                  Str     5
```

```
CAL_INFO  CALL_GRAPH_INFO     5000
   *Call_key              CA_key                  Comb   --
       ^Mod_number            CA_mod              Num     4
       ^Line_number           CA_line             Num     9
       ^Called_mod            CA_cmod             Num     4
       ^Parameter_number      CA_pnum             Num     3
    Parameter_passed      CA_par                  Str    20
```

COM_BLOb	COMMON_BLOCK_INF	10000	
*Common_key	CO_key	Comb	--
^Mod_number	CO_mod	Num	4
^Common_name	CO_name	Str	20
^Variable_name	CO_var	Str	20
Read_written	CO_r_w	Str	5

VAR_INFO	VARIABLE_INFO	10000	
*Variable_key	VA_key	Comb	--
^Mod_number	VA_mod	Num	4
^Line_number	VA_line	Num	9
^Variable_name	VA_name	Str	20
^Read_written	VA_r_w	Str	5
Variable_type	VA_type	Str	10

```
SUCC_LIN  SUCCESSOR_INFO      10000
   *Successor_key       SU_key                 Comb    --
       ^Mod_number          SU_mod             Num      4
       ^Line_number         SU_line            Num      9
       ^Successor_mod       SU_smod            Num      4
       ^Successor_line      SU_slin            Num      9
```

```
DO_LOOPS  DO_LOOP_INFO        5000
   *DO_key              DO_key                 Comb    --
       ^Mod_number          DO_mod             Num      4
       ^Line_num_start      DO_start           Num      9
       ^Line_num_end        DO_end             Num      9
    Index_used          DO_index               Str     20
```

A.2 B-Tree indices

In this section, the row containing Y is the name of the *record* for which a B-Tree index has been assigned and the row containing A's are the different fields the are used as indices.

```
1 MOD_INFO Y
Mod_number A



2 MOD_INPU Y
Mod_number A



3 MOD_INPU Y
Input_param A



4 CAL_INFO Y
Mod_number A
Line_number A
```

5 CAL_INFO Y

Called_mod A

6 VAR_INFO Y

Mod_number A

Line_number A

7 VAR_INFO Y

Variable_name A

8 COM_BLOK Y

Mod_number A

9 COM_BLOK Y

Common_name A

10 LINE_DES Y

Mod_number A

Line_number A

11 SUCC_LIN Y

Mod_number A

Line_number A

12 SUCC_LIN Y

Successor_mod A

Successor_line A

13 DO_LOOPS Y

Mod_number A

Line_num_start A

Line_num_end A

A.3 Parser Node Types

0 - N_ERROR

1 - N_ROOT

2 - N_MAIN

3 - N_F_SUBP

4 - N_S_SUBP

5 - N_BD_SUBP

6 - N_END

7 - N_PROGRAM

8 - N_FUNCTION

9 - N_INTEGER

10 - N_REAL

11 - N_DOUBLE_P

12 - N_COMPLEX

13 - N_LOGICAL

14 - N_CHARACTER

15 - N_LIST

16 - N_SUBROUTINE

17 - N_ASTERISK

18 - N_ENTRY

19 - N_BLOCKDATA

20 - N_DIMENSION

21 - N_ARR_DECLR

22 - N_ARDIM

23 - N_DARDIM

24 - N_EQUIV

25 - N_EQVSET

26 - N_COMMON

27 - N_BLNKCM

28 - N_LBLDCM

29 - N_CBITEMS

30 - N_TYPE

31 - N_CHAR_LEN

32 - N_IMPLICIT

33 - N_IMPL_DECL

34 - N_CHAR_RANGE

35 - N_PARAMETER

36 - N_PARAM_DECL

37 - N_EXTERNAL

38 - N_INTRINSIC

39 - N_SAVE

40 - N_CBLK_NAME

41 - N_DATA

42 - N_DATA_DECL

43 - N_DATA_ITEMS

44 - N_DATA_VALS

45 - N_MULT_VAL

46 - N_NEG

47 - N_DATA_IMPDO

48 - N_DOSPEC

49 - N_ASGN

50 - N_ASSIGN

51 - N_GOTO

52 - N_CMGOTO

53 - N_ASGOTO

54 - N_LABELLIST

55 - N_ARITHIF

56 - N_LOG_IF

57 - N_BLOCKIF

58 - N_ELSEIF

59 - N_ELSE

60 - N_ENDIF

61 - N_DO

62 - N_CONTINUE

63 - N_STOP

64 - N_PAUSE

65 - N_WRITE

66 - N_READ

67 - N_PRINT

68 - N_CILIST

69 - N_CIITEM

70 - N_CONCAT

71 - N_IOIMDL

72 - N_OPEN

73 - N_CLOSE

74 - N_INQUIRE

75 - N_BACKSPACE

76 - N_ENDFILE

77 - N_REWIND

78 - N_FORMAT

79 - N_REPEAT

80 - N_SLASH

81 - N_COLON

82 - N_CALL

83 - N_RETURN

84 - N_EQV

85 - N_NEQV

86 - N_OR

87 - N_AND

88 - N_NOT

89 - N_LT

90 - N_LE

91 - N_EQ

92 - N_NE

93 - N_GT

94 - N_GE

95 - N_PLUS

96 - N_MINUS

97 - N_POS

98 - N_MULTIPLY

99 - N_DIVIDE

100 - N_EXPONT

101 - N_SPAREN

102 - N_CCONST

103 - N_SUBSTR

104 - N_ARELM

105 - N_SSSPEC

106 - N_DEFAULT

107 - N_ICONST

108 - N_NAME

109 - N_LCONST

110 - N_RCONST

111 - N_DPCONST

112 - N_FMTFLD

113 - N_HCONST

114 - N_SCONST

115 - N_LABEL

116 - N_LABELREF

117 - N_SUBFMT

118 - N_IOKW

119 - N_FUNREF

120 - N_IMPCHAR

121 - N_STMT_FN

122 - N_UNITID

123 - N_FMTID

124 - N_AMBIGUOUS

125 - N_DCMPLX

126 - N_SCALE

127 - N_INCLEQV

128 - N_INCLDATA

129 - N_INCLCOMM

130 - N_INCLSAVE

131 - N_COMMENT

A.4 Symbol types

```
15000 - max_nodes
 1009 - max_strings     Other values: 1723,2111,3121,
      3557,4111,5003
 7500 - string_area
20000 - pt_ptr_size     This must be at least max_nodes
 1319 - max_symbols     This is like max_strings
    8 - symbol_size     This is the width of the symbol table
  199 - nesting_size    Depth of DO/IF nesting stack

    1 - S_LABEL         Symbol types
    2 - S_COMMON
    3 - S_NAME
    4 - S_PU
    5 - S_VAR
    6 - S_PARAM
    7 - S_PROC
    8 - S_SF
    9 - S_ENTRY

    1 - symbol_type     Symbol table field names
    2 - symbol_name
    3 - symbol_pun
```

4 - label_defn

5 - label_cf_ref

6 - label_DO_ref

7 - label_io_ref

8 - label_scope

4 - common_defn

4 - name_dtype

5 - name_chrlen

6 - name_status

7 - var_arr_decl

7 - parameter_df

7 - stmt_fn_defn

1 - decl_externl   Symbol table status bits (values)

2 - decl_intrins

4 - formal_param

8 - explicit_typ

16 - in_ASSIGN

32 - assigned_to

64 - in_READ_list

128 - in_DATA_stmt

256 - stmt_fn_para

```
   512 - in_EQUIV
  1024 - in_COMMON
  2048 - used_as_arg
  4096 - std_intrinsic
  8192 - fun_called
 16384 - in_expr
 32768 - sub_called
 65536 - doloop_index
 125936 - use_bits

     1 - type_integer    Names for data types
     2 - type_real
     3 - type_logical
     4 - type_complex
     5 - type_dblprec
     6 - type_char
     7 - type_generic
    -1 - type_routine
    -2 - type_bd
```

Appendix B

B.1 Sample Program

```
      program GEN
      integer P1(20), P2(20)
      integer E(20) , O(20), esize, osize
      do 10 i = 1,20
          P1(i) = i
          P2(i) = i
 10   continue
      call even(P1,i,E,esize)
      call odd (P2,i,O,osize)
      call prnt(E,O,esize,osize)
      stop
      end


      subroutine even(P1,m,E,size)
      integer P1(m) , E(m), size
      integer temp, temp1
      j = 1
      do 20 i = 1,m
 temp  = P1(i) / 2
```

```
        temp1 = (P1(i) + 1) /2
if(temp .eq. temp1) then
  E(j) = P1(i)
  j = j + 1
        endif
 20   continue
      size = j - 1
      return
      end



      subroutine odd(P2,m,O,size)
      integer P2(m) , O(m), size
      integer temp, temp1
      j = 1
      do 30 i = 1,m
         temp  = P2(i) / 2
         temp1 = (P2(i) + 1) /2
         if(temp .lt. temp1)  then
           O(j) = P2(i)
           j = j + 1
         endif
 30   continue
      size = j - 1
      return
      end
```

```
      subroutine prnt(E,O,esize,osize)
      integer esize, osize, E(esize),O(osize)

      print *, '----- EVEN -----'
      do 40 i = 1,esize
print *,E(i)
 40   continue

      print *, '----- ODD -----'
      do 50 i = 1,osize
         print *,O(i)
 50   continue

      return
      end
```

B.2 The Parse Tree

```
64 272
108 -1 0 1 2 0 7 1 12 62 63 0 9 0 7 11 12 0
108 -2 6 6 7 0 107 -8 0 5 6 0 22 5 0 4 7 0
21 4 11 3 12 0 108 -3 10 10 11 0 107 -8 0 9 10 0
22 9 0 8 11 0 21 8 0 7 12 0 30 3 24 2 63 0
9 0 17 23 24 0 108 -4 16 16 17 0 107 -8 0 15 16 0
22 15 0 14 17 0 21 14 21 13 24 0 108 -5 20 20 21 0
107 -8 0 19 20 0 22 19 0 18 21 0 21 18 22 17 24 0
108 -6 23 21 24 0 108 -7 0 22 24 0 30 13 30 12 63 0
116 -8 29 29 30 0 108 -9 27 28 29 0 107 -35 28 26 29 0
107 -8 0 27 29 0 48 26 0 25 30 0 61 25 35 24 63 0
108 -2 32 32 33 0 108 -9 0 31 33 0 104 31 34 34 35 0
108 -9 0 33 35 0 49 33 40 30 63 0 108 -3 37 37 38 0
108 -9 0 36 38 0 104 36 39 39 40 0 108 -9 0 38 40 0
49 38 42 35 63 0 115 -8 0 41 42 0 62 41 48 40 63 0
108 -10 44 47 48 0 108 -2 45 43 48 0 108 -9 46 44 48 0
108 -4 47 45 48 0 108 -6 0 46 48 0 82 43 54 42 63 0
108 -11 50 53 54 0 108 -3 51 49 54 0 108 -9 52 50 54 0
108 -5 53 51 54 0 108 -7 0 52 54 0 82 49 60 48 63 0
108 -12 56 59 60 0 108 -4 57 55 60 0 108 -5 58 56 60 0
108 -6 59 57 60 0 108 -7 0 58 60 0 82 55 61 54 63 0
63 0 62 60 63 0 6 0 0 61 63 0 2 2 139 272 64 0
1 63 0 64 64 0 108 -13 67 67 71 0 108 -14 68 70 67 0
15 66 0 65 71 0 108 -15 69 66 67 0 108 -16 70 68 67 0
```

108 -17 0 69 67 0 16 65 82 138 139 0 9 0 76 81 82 0
108 -14 75 75 76 0 108 -15 0 74 75 0 22 74 0 73 76 0
21 73 80 72 82 0 108 -16 79 79 80 0 108 -15 0 78 79 0
22 78 0 77 80 0 21 77 81 76 82 0 108 -17 0 80 82 0
30 72 86 71 139 0 9 0 84 85 86 0 108 -18 85 83 86 0
108 -19 0 84 86 0 30 83 89 82 139 0 108 -20 88 88 89 0
107 -35 0 87 89 0 49 87 95 86 139 0 116 -21 94 94 95 0
108 -22 92 93 94 0 107 -35 93 91 94 0 108 -15 0 92 94 0
48 91 0 90 95 0 61 90 102 89 139 0 108 -18 101 101 102 0
108 -14 98 98 99 0 108 -22 0 97 99 0 104 97 100 100 101 0
107 -71 0 99 101 0 99 99 0 96 102 0 49 96 112 95 139 0
108 -19 111 111 112 0 108 -14 105 105 106 0 108 -22 0 104 106 0
104 104 107 107 108 0 107 -35 0 106 108 0 95 106 0 108 109 0
101 108 110 110 111 0 107 -71 0 109 111 0 99 109 0 103 112 0
49 103 116 102 139 0 108 -18 114 114 115 0 108 -19 0 113 115 0
91 113 0 115 116 0 57 115 123 112 139 0 108 -16 118 118 119 0
108 -20 0 117 119 0 104 117 122 122 123 0 108 -14 121 121 122 0
108 -22 0 120 122 0 104 120 0 119 123 0 49 119 128 116 139 0
108 -20 127 127 128 0 108 -20 126 126 127 0 107 -35 0 125 127 0
95 125 0 124 128 0 49 124 129 123 139 0 60 0 131 128 139 0
115 -21 0 130 131 0 62 130 136 129 139 0 108 -17 135 135 136 0
108 -20 134 134 135 0 107 -35 0 133 135 0 96 133 0 132 136 0
49 132 137 131 139 0 83 0 138 136 139 0 6 0 0 137 139 0
4 71 214 63 64 0 108 -23 142 142 146 0 108 -24 143 145 142 0
15 141 0 140 146 0 108 -25 144 141 142 0 108 -26 145 143 142 0
108 -27 0 144 142 0 16 140 157 213 214 0 9 0 151 156 157 0
108 -24 150 150 151 0 108 -25 0 149 150 0 22 149 0 148 151 0

21 148 155 147 157 0 108 -26 154 154 155 0 108 -25 0 153 154 0
22 153 0 152 155 0 21 152 156 151 157 0 108 -27 0 155 157 0
30 147 161 146 214 0 9 0 159 160 161 0 108 -28 160 158 161 0
108 -29 0 159 161 0 30 158 164 157 214 0 108 -30 163 163 164 0
107 -35 0 162 164 0 49 162 170 161 214 0 116 -31 169 169 170 0
108 -32 167 168 169 0 107 -35 168 166 169 0 108 -25 0 167 169 0
48 166 0 165 170 0 61 165 177 164 214 0 108 -28 176 176 177 0
108 -24 173 173 174 0 108 -32 0 172 174 0 104 172 175 175 176 0
107 -71 0 174 176 0 99 174 0 171 177 0 49 171 187 170 214 0
108 -29 186 186 187 0 108 -24 180 180 181 0 108 -32 0 179 181 0
104 179 182 182 183 0 107 -35 0 181 183 0 95 181 0 183 184 0
101 183 185 185 186 0 107 -71 0 184 186 0 99 184 0 178 187 0
49 178 191 177 214 0 108 -28 189 189 190 0 108 -29 0 188 190 0
89 188 0 190 191 0 57 190 198 187 214 0 108 -26 193 193 194 0
108 -30 0 192 194 0 104 192 197 197 198 0 108 -24 196 196 197 0
108 -32 0 195 197 0 104 195 0 194 198 0 49 194 203 191 214 0
108 -30 202 202 203 0 108 -30 201 201 202 0 107 -35 0 200 202 0
95 200 0 199 203 0 49 199 204 198 214 0 60 0 206 203 214 0
115 -31 0 205 206 0 62 205 211 204 214 0 108 -27 210 210 211 0
108 -30 209 209 210 0 107 -35 0 208 210 0 96 208 0 207 211 0
49 207 212 206 214 0 83 0 213 211 214 0 6 0 0 212 214 0
4 146 272 139 64 0 108 -33 217 217 221 0 108 -34 218 220 217 0
15 216 0 215 221 0 108 -35 219 216 217 0 108 -36 220 218 217 0
108 -37 0 219 217 0 16 215 233 271 272 0 9 0 223 232 233 0
108 -36 224 222 233 0 108 -37 228 223 233 0 108 -34 227 227 228
108 -36 0 226 227 0 22 226 0 225 228 0 21 225 232 224 233 0
108 -35 231 231 232 0 108 -37 0 230 231 0 22 230 0 229 232 0

21 229 0 228 233 0 30 222 237 221 272 0 17 0 0 234 235 0
123 234 236 236 237 0 114 -76 0 235 237 0 67 235 243 233 272 0
116 -38 242 242 243 0 108 -39 240 241 242 0 107 -35 241 239 242
108 -36 0 240 242 0 48 239 0 238 243 0 61 238 249 237 272 0
17 0 0 244 245 0 123 244 248 248 249 0 108 -34 247 247 248 0
108 -39 0 246 248 0 104 246 0 245 249 0 67 245 251 243 272 0
115 -38 0 250 251 0 62 250 255 249 272 0 17 0 0 252 253 0
123 252 254 254 255 0 114 -96 0 253 255 0 67 253 261 251 272 0
116 -40 260 260 261 0 108 -39 258 259 260 0 107 -35 259 257 260
108 -37 0 258 260 0 48 257 0 256 261 0 61 256 267 255 272 0
17 0 0 262 263 0 123 262 266 266 267 0 108 -35 265 265 266 0
108 -39 0 264 266 0 104 264 0 263 267 0 67 263 269 261 272 0
115 -40 0 268 269 0 62 268 270 267 272 0 83 0 271 269 272 0
6 0 0 270 272 0 4 221 0 214 64 0

B.3 The Symbol Table

```
25 115
GEN'P1'20'P2'E'O'ESIZE'OSIZE'10'I'1'EVEN'ODD'PRNT'M'SIZE'
TEMP'TEMP1'J'2'30'----- EVEN -----'40'----- ODD -----'50'
40 4 250 0
4 1 1 -3 0 0 0 0 5 5 1 1 0 18472 6 0
5 11 1 1 0 18472 10 0 5 14 1 1 0 18440 16 0
5 16 1 1 0 18440 20 0 5 18 1 1 0 18440 0 0
5 24 1 1 0 18440 0 0 1 30 1 42 0 1 0 30
5 33 1 1 0 83968 0 0 7 37 1 -1 0 32768 0 0
7 42 1 -1 0 32768 0 0 7 46 1 -1 0 32768 0 0
4 37 2 -1 0 0 0 0 5 5 2 1 0 16396 75 0
5 51 2 1 0 16388 0 0 5 14 2 1 0 44 79 0
5 53 2 1 0 44 0 0 5 58 2 1 0 16424 0 0
5 63 2 1 0 16424 0 0 5 69 2 1 0 16416 0 0
1 8 2 131 0 1 0 95 5 33 2 1 0 81920 0 0
4 42 3 -1 0 0 0 0 5 11 3 1 0 16396 150 0
5 51 3 1 0 16388 0 0 5 16 3 1 0 44 154 0
5 53 3 1 0 44 0 0 5 58 3 1 0 16424 0 0
5 63 3 1 0 16424 0 0 5 69 3 1 0 16416 0 0
1 73 3 206 0 1 0 170 5 33 3 1 0 81920 0 0
4 46 4 -1 0 0 0 0 5 14 4 1 0 16396 227 0
5 16 4 1 0 16396 231 0 5 18 4 1 0 16396 0 0
5 24 4 1 0 16396 0 0 1 93 4 251 0 1 0 243
5 33 4 1 0 81920 0 0 1 112 4 269 0 1 0 261
1 13 23 33
```

B.4 Dbload Format

```
[MOD_INFO]
|1|GEN|AVAILABLE|
|2|EVEN|AVAILABLE|
|3|ODD|AVAILABLE|
|4|PRNT|AVAILABLE|

[MOD_INPU]
|1|_NOTHING|N_A|N_A|
|2|P1|INT|READ|
|2|M|INT|READ|
|2|E|INT|WRITE|
|2|SIZE|INT|WRITE|
|3|P2|INT|READ|
|3|M|INT|READ|
|3|O|INT|WRITE|
|3|SIZE|INT|WRITE|
|4|E|INT|READ|
|4|O|INT|READ|
|4|ESIZE|INT|READ|
|4|OSIZE|INT|READ|
```

[SUCC_LIN]

|1|1|1|2|

|1|2|1|3|

|1|3|1|4|

|1|4|1|5|

|1|5|1|6|

|1|6|1|7|

|1|8|1|9|

|1|9|1|10|

|1|10|1|11|

|1|11|1|12|

|2|1|2|2|

|2|2|2|3|

|2|3|2|4|

|2|4|2|5|

|2|5|2|6|

|2|6|2|7|

|2|7|2|8|

|2|8|2|9|

|2|9|2|10|

|2|10|2|11|

|2|11|2|12|

|2|13|2|14|

|2|14|2|15|

|3|1|3|2|

|3|2|3|3|

|3|3|3|4|

|3|4|3|5|

|3|5|3|6|

|3|6|3|7|

|3|7|3|8|

|3|8|3|9|

|3|9|3|10|

|3|10|3|11|

|3|11|3|12|

|3|13|3|14|

|3|14|3|15|

|4|1|4|2|

|4|2|4|3|

|4|3|4|4|

|4|4|4|5|

|4|5|4|6|

|4|7|4|8|

|4|8|4|9|

|4|9|4|10|

|4|11|4|12|

|1|7|1|4|

|1|4|1|8|

|2|12|2|5|

|2|5|2|13|

|2|8|2|11|

|3|12|3|5|

```
|3|5|3|13|
|3|8|3|11|
|4|6|4|4|
|4|4|4|7|
|4|10|4|8|
|4|8|4|11|

[VAR_INFO]
|1|5|P1|WRITE|INT|
|1|5|I|WRITE|INT|
|1|5|I|READ|INT|
|1|6|P2|WRITE|INT|
|1|6|I|WRITE|INT|
|1|6|I|READ|INT|
|2|4|J|WRITE|INT|
|2|6|TEMP|WRITE|INT|
|2|6|P1|READ|INT|
|2|6|I|READ|INT|
|2|7|TEMP1|WRITE|INT|
|2|7|P1|READ|INT|
|2|7|I|READ|INT|
|2|9|E|WRITE|INT|
|2|9|J|WRITE|INT|
|2|9|P1|READ|INT|
|2|9|I|READ|INT|
|2|10|J|WRITE|INT|
|2|10|J|READ|INT|
```

```
|2|13|SIZE|WRITE|INT|
|2|13|J|READ|INT|
|3|4|J|WRITE|INT|
|3|6|TEMP|WRITE|INT|
|3|6|P2|READ|INT|
|3|6|I|READ|INT|
|3|7|TEMP1|WRITE|INT|
|3|7|P2|READ|INT|
|3|7|I|READ|INT|
|3|9|O|WRITE|INT|
|3|9|J|WRITE|INT|
|3|9|P2|READ|INT|
|3|9|I|READ|INT|
|3|10|J|WRITE|INT|
|3|10|J|READ|INT|
|3|13|SIZE|WRITE|INT|
|3|13|J|READ|INT|

[COM_BLOK]

[LINE_DES]
|1|1|N_PROGRAM|
|1|2|N_TYPE|
|1|3|N_TYPE|
|1|4|N_DO|
|1|5|N_ASGN|
|1|6|N_ASGN|
```

|1|7|N_CONTINUE|

|1|8|N_CALL|

|1|9|N_CALL|

|1|10|N_CALL|

|1|11|N_STOP|

|1|12|N_END|

|2|1|N_SUBROUTINE|

|2|2|N_TYPE|

|2|3|N_TYPE|

|2|4|N_ASGN|

|2|5|N_DO|

|2|6|N_ASGN|

|2|7|N_ASGN|

|2|8|N_BLOCKIF|

|2|9|N_ASGN|

|2|10|N_ASGN|

|2|11|N_ENDIF|

|2|12|N_CONTINUE|

|2|13|N_ASGN|

|2|14|N_RETURN|

|2|15|N_END|

|3|1|N_SUBROUTINE|

|3|2|N_TYPE|

|3|3|N_TYPE|

|3|4|N_ASGN|

|3|5|N_DO|

|3|6|N_ASGN|

|3|7|N_ASGN|

|3|8|N_BLOCKIF|

|3|9|N_ASGN|

|3|10|N_ASGN|

|3|11|N_ENDIF|

|3|12|N_CONTINUE|

|3|13|N_ASGN|

|3|14|N_RETURN|

|3|15|N_END|

|4|1|N_SUBROUTINE|

|4|2|N_TYPE|

|4|3|N_PRINT|

|4|4|N_DO|

|4|5|N_PRINT|

|4|6|N_CONTINUE|

|4|7|N_PRINT|

|4|8|N_DO|

|4|9|N_PRINT|

|4|10|N_CONTINUE|

|4|11|N_RETURN|

|4|12|N_END|

[CAL_INFO]

|1|8|2|1|P1|

|1|8|2|2|I|

|1|8|2|3|E|

|1|8|2|4|ESIZE|

```
|1|9|3|1|P2|

|1|9|3|2|I|

|1|9|3|3|O|

|1|9|3|4|OSIZE|

|1|10|4|1|E|

|1|10|4|2|O|

|1|10|4|3|ESIZE|

|1|10|4|4|OSIZE|


[DO_LOOPS]

|1|4|7|I|

|2|5|12|I|

|3|5|12|I|

|4|4|6|I|

|4|8|10|I|
```

BIBLIOGRAPHY

[ALL 83] Allen, J.R., Kennedy, K., Porterfield, C., and Warren, J. : *A Conversion of Control Dependencies to Data Dependence* , Proc 10th Annual ACM Symp on Princ of Prog Lang , pp 177-189, Jan 1983.

[BRO 85] Browne, J.C. : *Formulation and Programming of Parallel Computations : A Unified Approach* ,pp.624-631, ICPP (1985).

[BRO 86] Browne, J.C. : *Hierarchical Dependency Graphs for Fortran Programs* , unpublished report.

[CHA 76] Chamberlin, D.D., Astrahan, M.M., Eswaran, K.P., Griffith, P.P., Lorie, R.A., Mehl, J.W., Reisner, P., and Wade, B.W. : *SEQUEL 2 : A unified approach to Data Definition, Manipulation, and Control* , IBM Journal of Research and Developement; Volume 20,November 6,1976, pp. 560-575.

[COD 70] Codd, E.F., *A Relational Model of data for large shared data banks* , Commun, ACM13,6 (June), 377-387.

[COH 84] Toolpack/1 *Target Fortran 77; Toolpack/1* , Version: 2.1, NAG Publication: NP1313.

[HECHT] HECHT, MATTHEW S. : *Flow Analysis for Computer Programs* , Text Book.

[ILES] ILES R; *ISTLX User Guide* , Toolpack/1, Version: 2.1, NAG Publication(1984) : NP1289.

[ISTYP] Cohen, M., *ISTYP User Guide* , NAG Publication(1984) : N-P1300.

[JOH 78] Yacc: *Yet Another Compiler-Compiler* , UNIX(TM) PROGRAMMER'S MANUAL, Seventh Edition Volume 2B, 1978 ; Bell Telephone Laboratories, Incorporated, New Jersey.

[KUC 77] Kuck, D.I. : *A Survey of Parallel Machine Organization and Programming* , Computing Surveys 9, pp 29-60 (1977).

[SRI 88] Sriram, R. : *A Facility for the Display and Manipulation of Dependency Graphs* , M.S. Thesis, Dept. of Elec. Engg., UT Austin, (May 1988).

[SUN] *SunUnify Reference Manual.*

VITA

Sivagnanam Ramasundaram Easwar was born in Madras, India on April 24th 1962. He finished his Matriculation from Don Bosco Matric School, Madras, in 1978. He received his B.E degree in Electronics and Communication Engineering from the University of Madras, India in June 1984. He has been in the Graduate School at the University of Texas at Austin since January 1985. He expects to obtain his Master of Science degree in Electrical and Computer Engineering in May 1988.

Permanent address: B-6
Anna Nagar
Madras, India 600102

This thesis was typeset[1] with LaTeX by Ruma Easwar and the author.

[1] LaTeX document preparation system was developed by Leslie Lamport as a special version of Donald Knuth's TeX program for computer typesetting. TeX is a trademark of the American Mathematical Society. The LaTeX macro package for The University of Texas at Austin thesis format was written by Khe-Sing The.