To the Graduate Council:

I am submitting herewith a thesis written by C. Chris Meystrik entitled "An Interactive System for Program Dependence Analysis". I have examined the final copy of this thesis for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Master of Science, with a major in Computer Science.

Micah Beck, Major Professor

We have read this thesis
and recommend its acceptance:

Michael W. Berry

Brad Vander Zanden

Accepted for the Council:

Associate Vice Chancellor
and Dean of the Graduate School

An Interactive System for Program Dependence Analysis

A Thesis

Presented for the

Master of Science Degree

The University of Tennessee, Knoxville

C. Chris Meystrik

December 1996

Dedication

I dedicate this work to the man whom I have admired the most in my life, my father, Charles H. Meystrik. His love through the years brought me joy and happiness. His strength fueled my fires. His devotion and passion ignited my life. Although my father passed away January 23, 1990, he has always and will always be a source of inspiration in my life. I will always tell my children that their grandpa was the greatest man they never knew. Even with new wind beneath my wings the old wind that made me soar will never be forgotten. In loving memory.

I love you dad.

Acknowledgments

I would like to thank my thesis advisor, Dr. Micah Beck, for his support, guidance, and most of all his patience. I also thank Dr. Mike Berry and Dr. Brad Vander Zanden who served on my committee. I would like to thank my mother Terry Meystrik, brother Jeff Meystrik, and my sister Michele Heitman, whose closeness has always been a source of inspiration to me. I would like to thank Dr. Leonid Sagalovsky for his initial encouragement to attend graduate school. I thank my great friend Tim Martin for his words of support and encouragement and his experience in finishing a thesis. I also thank my friend Carolyn Aebischer for her tremendous help in editing the final draft, it was cool to finish with her. A big appreciation goes to my son, Jacob, who even as a pre-schooler still recognized my absence and missed daddy when I was not around. Just the little hints that he wanted me around were enough to give me the desire to finish. I thank my daughter Rebecca, all she has to do is smile. The biggest thanks I save for last. I would like to thank my wife Carianne. Without her support, this would not have been possible. Her sacrifices are numerous, and her patience is amazing; even down to the wire, I could always count on her to put me on the right track again. She is the wind beneath my wings. Finally, I thank the Lord above all else, without whom nothing is possible.

Abstract

Since the introduction of parallel computers system, designers have been trying to make parallel programming easier. Several efforts have taken place in easing the burden of programmers who target supercomputers for their applications: (1) implementation of high level languages, (2) design of tools to aid in programming, and (3) the use of low level programming interfaces specific for to architecture. The desired approach to producing code for a particular architecture is dependent on many factors including the skill level of the programmer and the effort required to produce the correct code.

However, there is also another problem domain. There exists many applications that have been written for the scientific and engineering communities that were originally designed for sequential architectures. Some of this code is still essential to scientific research and the efficiency of that research could be improved if the runtime of the code could be reduced, therefore allowing the scientist to concentrate on analysis. It is, therefore, desirable for some of these applications to be "ported" to some current supercomputer architecture. The porting of this complex scientific code would potentially take many man years, especially when dealing with supercomputers and their programming environments. This problem has inspired an area of research focused around designing *supercompilers* . A *supercompiler* transforms code written in sequential programming languages into optimized code for use on a particular target supercomputer. With the success of *supercompiler* technology the burden of programmers could poten-

tially be eased. Given the idiosyncrasies of supercomputer programming environments, if a programmer could write sequential code and have the compiler do the transformation to the parallel environment these architectures would potentially be more widely used. Minimally, the use of these computers could be spread to programmers who are less familiar with the details of a particular architecture.

One of the most important areas of interest inside of *supercompiler* research is dependence analysis. In dependence analysis the compiler tries to determine if there exists a dependence between two or more expressions, usually inside of a loop, in order to facilitate the production of parallel or vector code for the target machine. Current algorithms used in dependence analysis use a data flow system in a feed-forward manner. In other words, they proceed from the top of the *control flow graph* and propagate information forward. This thesis presents a backward-chaining algorithm as another approach which has the potential to uncover some independent expressions that would otherwise be inaccessible to conventional dependence algorithms. Also presented is an interactive environment for testing the application of the proposed algorithms inside a compiler.

Contents

1 Introduction 1

1.1 Overview 1

1.2 Motivation 3

1.3 Analysis Environment 6

1.4 Overall Architecture 7

1.4.1 Input Language 7

1.4.2 Compiler Analysis 7

1.4.3 Data Dependence Engine 11

1.4.4 Graphical User Interface 13

2 Language Definition 14

2.1 Declarations 16

2.2 Assignments 17

2.3 Expressions 17

2.4 Conditionals 19

2.5 Loops 19

3 Compiler Design **21**

3.1 Parser and Syntax Tree 21

3.2 Control Flow Graph 23

3.3 Prolog Control Graph Code Generation 26

3.3.1 Edges and Nodes 27

3.3.2 Assignments 28

3.3.3 Loops 29

3.3.4 Conditionals 29

3.3.5 Merges 31

3.3.6 Operators 31

3.3.7 Three Address Code 33

4 Graphical Interface **34**

4.1 Compiling Programs 36

4.2 Selecting Statements 36

4.3 Variable Selection 37

4.4 In/Out Selection 37

4.5 Executing Tests 38

4.5.1 Constant Variables 39

4.5.2 Induction Variables 40

4.5.3 Loop Invariance 40

4.5.4 GCD Dependence Test 40

4.5.5 Dependence Test 41

4.6 Internal State Database 41

5 Data Dependence Logic **43**

5.1 Assertional Logic and Propagation 44

5.2 Backward Chaining Algorithm 46

5.3 Assertion Cache 47

5.4 Substitution of Expressions 48

5.5 Loops and their Properties 50

5.6 Induction Variable Recognition 51

5.7 Dependence Tests 53

5.7.1 Conventional problems with Reaching Definitions 53

5.7.2 Dependence test cross-checking 55

6 Future Work **57**

6.1 Implementation of Data Dependence Engine in C 57

6.2 Multi-dimensional Arrays 58

6.3 Implementation of more dependence tests 58

6.4 Code transformation in the compiler 58

6.5 User Assertions from the interface 59

Bibliography **60**

Appendix 63

A Data Dependence Logic 64

B Parse Tree/CFG Data Structure 72

C Tiny Language Grammar 74

Vita 77

List of Tables

2.1 Syntax of declarations 17

2.2 Syntax of assignments 18

2.3 Syntax of expressions 18

2.4 Syntax of conditionals 19

2.5 Syntax of loops 20

3.1 Elements of the syntax tree node. 25

3.2 Elements of the *CFG* node. 25

3.3 Elements of the *CFG* edge. 25

5.1 Logic rules for propagation 45

5.2 Propagation implementation 46

5.3 Substitution implementation 50

List of Figures

1.1 Difficult induction variable 5

1.2 Overall system architecture 8

1.3 Pipe interface to Prolog interpreter 13

2.1 Sample input program 15

3.1 Simple input program for the syntax tree 24

3.2 Internal syntax tree example 24

3.3 Control flow graph example 26

3.4 Flow of input language into data dependence engine 27

3.5 Mapping of input program to Prolog *CFG* 28

3.6 Loop edges 30

3.7 Conditional edges 30

3.8 Merge node usage relative to an input program 32

3.9 Mapping of three-address code to Prolog 33

4.1 Main GUI interface window 35

4.2 Edge direction example 39

5.1 Data Dependence Engine Components 44

5.2 *Backchain*(g) algorithm definition 47

5.3 *Backchain*(g) algorithm with assertion cache 48

5.4 Induction variable discovery 52

5.5 Data dependence in backchaining 54

5.6 Dependence test depiction 56

Chapter 1

Introduction

1.1 Overview

Parallel compilers and compiler environments have taken on many forms in the past few years. Some commercial and educational research tools have surfaced that show us how sequential code can be analyzed and restructured to make use of hidden concurrency. By finding array sub-expressions that are independent of each other one can potentially rearrange code in order to produce more optimal code for use on parallel machines or generate vector code for a vector processor. This thesis explores the idea of backchaining algorithms as an approach to discovering independence in program expressions which are much more difficult to discover using the traditional "feed-forward" data flow analysis approaches. These backchaining algorithms are implemented in an interactive compiler environment.

The *data dependence engine* used in this thesis was originally developed by Dr.

Micah Beck as a method of discovering independent expressions that are hidden by "difficult constants" [Bec94]. The *data dependence engine* has been extend to include the domain of induction variables and further extends the induction algorithm to include the discovery of "difficult induction variables". This demand driven discovery is then used in a dependence test to discover some independences which are more difficult for current data flow algorithms to discover due to performance constraints. The algorithms presented in this thesis are suitable for implementation in an interactive compiler environment because they do not depend on a phasic approach to execution.

The rest of this chapter will present the overall architecture of the system, how queries are performed internally, and the mechanisms that are used for communications between the different parts of the system, including a discussion of how data flows through the system. Then the language definition will be described for our compiler will be described, and the features and design of the compiler discussed. Next, the graphical interface will be presented, along with a description of how it is used to perform queries on the programs through the *data dependence engine* . The next discussion will be on the implementation of the algorithms in the Prolog *data dependence engine* along with examples. Finally, some future directions that are possible with this work along with some improvements which can be made for practical application is a commercial compiler will be presented.

1.2 Motivation

The primary goal of this thesis was to develop a method of data dependence testing using backchaining in order to prove the independence of expressions that would normally be inadmissible to the traditional data flow algorithms. Also, the concept of propagation of induction and dependence is incorporated in order to discover expression independence. This allows there to be a dependence query performed at a particular point in the program *control flow graph* and to have the query propagate back through the *control flow graph* in order for the goal to be proven at the original node. A dependence query is a query from the interface to the *data dependence engine* about a particular program statement or variable. A typical query might ask if some variable, x, is a constant at a given program statement, S_1 (i.e. $constant(x, S_1)$).

There is also an investigation of interactive compilation in this thesis. Consequently, it is required that the dependence tests have the following properties:

1. Demand driven dependence testing.

2. Integration of the different phases of analysis into a single phase.

With these properties the compiler would not have to consider the problems of phase ordering that are encountered in normal data flow analysis. In other words, the compiler would not have to concern itself with the ordering of phases such as constant propagation, loop invariant code discovery, induction variable discovery and substitution and the circular execution of these phases. The algorithms presented here are designed

such that when a specific demand arises (i.e. discovery of a constant is necessary) the algorithm attempts to prove that "sub goal" in order for the original goal to be satisfied.

Our algorithms are designed for the general case and extend the normal data flow algorithms to find what are referred to as "difficult induction variables". Figure 1.1 shows the idea of a "difficult induction variable" in a program fragment. Program assertions are denoted by $\{expr\}$ and apply after the preceding statement executes. Although there is not much magic to the discovery of these particular induction variables the method of discovery is very important to the algorithm presented for dependence testing. What is important to note in the example presented in Figure 1.1 is that, if one wanted to use the assertions about x and y being induction variables, they would only apply within the "scope" of the conditional statements code blocks (i.e. not past statement 11). Furthermore, the induction of variables x and y is different depending on the resulting run time evaluation of ϕ, which is assumed to be unknown at compile time. To apply this to the example in Figure 1.1, if one were to ask about the compile time state of variable z at statement 11, the only conclusion that any compiler could make would be $\{unknown\}$. Furthermore, using conventional techniques, the assertions about the variables x and y are lost after statement 11. Any query (i.e. dependence test) that might be able to use this information for admissibility would be lost. Given this informal example one can see how important it is for dependence testing to be applied backward through the program. With the back propagation of goals the compiler can uncover information that would otherwise not be available to statements from which dependence

tests are applied. The forward data flow techniques have performance problems; as the size of the problem grows the amount of information to preserve grows exponentially. The backchaining algorithms do not preserve state data at individual nodes and thus do not have this problem.

```
1:   j = 3
2:   k = 6
3:   for i = 0 to 10 do
4:      j = j + 6
5:      k = k + 3
6:      if ( φ ) then
7:         y = j
8:         x = k
        else
9:         y = k
10:        x = j
        endif
11:     z = x + y
     endfor
```

(a) Initial program

```
1:   j = 3
2:   k = 6
3:   for i = 0 to 10 do
4:      j = j + 6
        {j = 6 * i + 9}
5:      k = k + 3
        {k = 3 * i + 9}
6:      if ( φ ) then
7:         y = j
           {y = 6 * i + 9}
8:         x = k
           {x = 3 * i + 9}
        else
9:         y = k
           {y = 3 * i + 9}
10:        x = j
           {x = 6 * i + 9}
        endif
11:     z = x + y
     endfor
```

(b) Program assertions

Figure 1.1: Difficult induction variable

The secondary goals were to build a compiler and interface to test the *data dependence engine* . With these secondary goals in mind a compiler was built based on the language developed by Dr. Michael Wolfe [Wol90]. The grammar for the language was taken from Dr. Wolfe's "Tiny Loop Restructuring Tool". Around the compiler there is

a graphical interface to perform the necessary queries to the *data dependence engine* and to gather the results returned by the *data dependence engine* . The interface and the compiler integration to the *data dependence engine* gives us the mechanism to perform tests on the *data dependence engine* from "real" source code. The design of the compiler and interface are considered secondary because they would not be relevant without a dependence engine to integrate with them; however, their design and implementation are still very significant to this work.

1.3 Analysis Environment

The analysis environment was fully developed by the author. The compiler was designed and written in C and is currently ported to the Sun Microsystems SunOS 4.1.x operating system and Linux. It converts the "Tiny" language into the Prolog *control flow graph* . The graphical interface, which was developed under X Windows using the OSF/Motif widget set. The *data dependence engine* is written in Prolog and currently runs under SWI Prolog 2.1 [Wie95]. The interface program and the Prolog interpreter are coupled together by an inter-process communications mechanism to facilitate the queries between the compiler and the *data dependence engine* . An overview of this environment is given in the following sections and details of each piece are described in the following chapters.

1.4 Overall Architecture

The overall architecture of the system is shown in Figure 1.2. This figure shows the general flow of data and control throughout the system. It also pictures the graphical interface, shown as the **Compiler/Data Dependence Engine Interface**, and how it is connected to the system. Each element of the system is discussed in the following sections of this chapter.

1.4.1 Input Language

The input language for the compiler is just like any other programming language, the source is written into a flat text file in the filesystem, and that file is read into the compilers lexical analyzer. Figure 1.2 shows the input language being read by the compiler. The details of the language are given in Chapter 2 and the full language grammar is in Appendix C.

1.4.2 Compiler Analysis

Compiler analysis is made up of four parts: lexical analysis, token parsing, *syntax tree* creation, and *control flow graph* creation. Each part of compiler analysis acts much like a filter; it takes input and converts it to some output stream or data structure. However, compiler analysis is only the front end of the compiler. The compiler has two other separate phases, which are discussed in Chapter 3: Prolog control flow graph generation and code optimization. When looking at the architecture from a control and

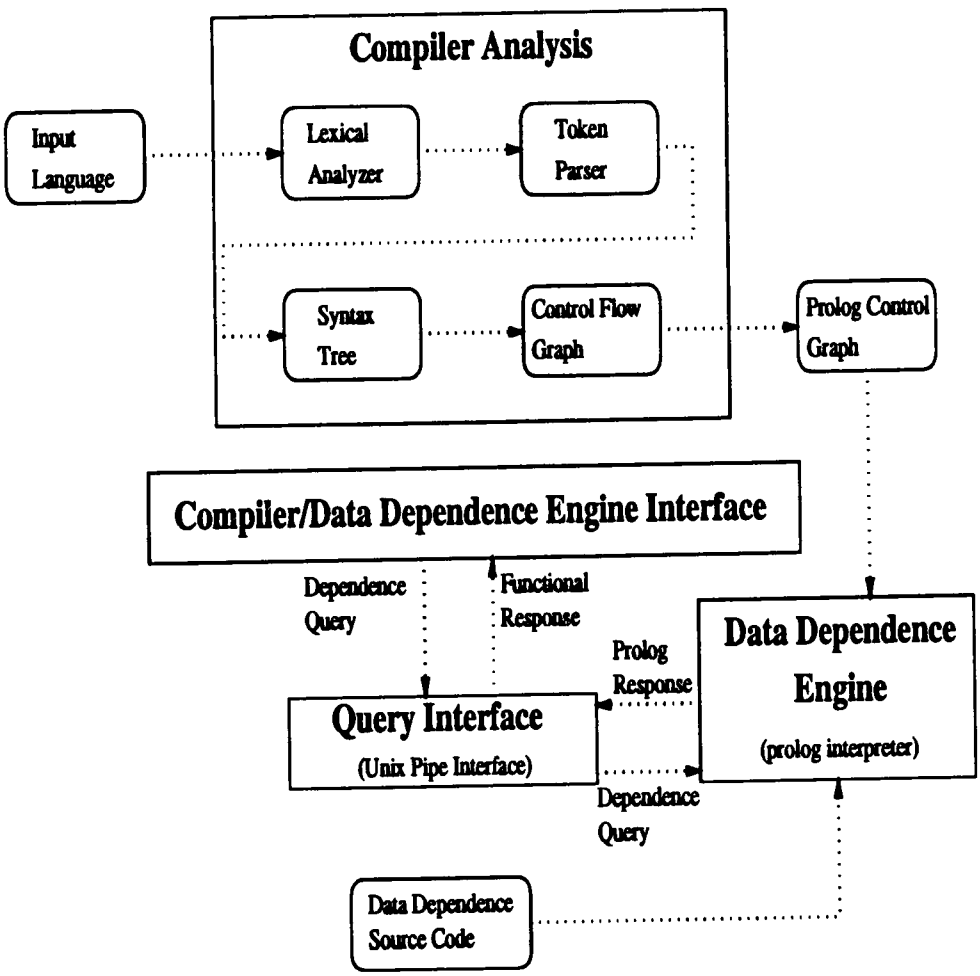


Figure 1.2: Overall system architecture

data flow perspective these two phases are not completely obvious. Compiler analysis is a further break down of the first two phases (discussed in Chapter 3) into the four parts listed above. Each element of compiler analysis is discussed in the following subsections.

Lexical Analyzer

The lexical analyzer takes the input language as a stream of characters. This input stream is broken into tokens and fed to the parser, which analyzes the language for correctness. The main objective of the lexical analyzer is the conversion of the character input stream into a token stream for the parser. A token is a sequence of characters having a collective meaning. The lexical analyzer was written using lex , a Unix tool for generating lexical analyzers. The lexical analyzer is based on the code developed for the Tiny project [Wol90].

Parser

The parser uses a token stream as input and groups the tokens into grammatical phrases. These grammatical phrases describe the syntactical relationship of each token. The syntactical relationship of the tokens is given by a context-free grammar (see Appendix C) [ARJ86]. This grammar is compiled into a set of tables using yacc , which executes an LR(1) parsing algorithm on the input token stream [ARJ86]. The grammar for the parser is based on the code developed for the Tiny project [Wol90]. During the parsing of the input program the compiler also performs another very important step, creation of a symbol table. The symbol table is a representation of each variable in the

input language its name along with its type, value (if any), and dimensions (arrays). Although the symbol table is not discussed it is a critical part of the success of the control flow analysis and optimization phases.

Syntax Tree

The *syntax tree* is an intermediate representation of the input program. This intermediate representation is an internal data structure which depicts a syntactical description of the input program. The *syntax tree* is discussed in Section 3.1. After the *syntax tree* is generated, the source program can be checked for semantic correctness.

Control Flow Graph Creation

The input to the control flow graph creation module is the head node to the *syntax tree* . From the *syntax tree* the control flow graph module evaluates the language and determines how control flows between the particular statements (i.e. nodes) of the input program. The *control flow graph* module generates a *control flow graph* data structure (Section 3.2). This data structure describes all possible control paths through the input program. A control path is an ordered sequence of statements (i.e. instructions) that are to be scheduled during execution of the input program. Any control path may be altered during the code optimization phase of the compiler provided that the effect of the original code is preserved. Given a *control flow graph* , the compiler can then determine, for a given variable, the flow of data through the input program (i.e. Data Flow Analysis) in terms of definition-use and use-definition chains [ZC91]. However,

this step is not necessary in our implementation and is one of the phases of compiler optimization that is eliminated.

1.4.3 Data Dependence Engine

The *data dependence engine* is made up of three parts: the Prolog control flow graph, the data dependence logic, and the dependence query interface. Both the Prolog control flow graph and the data dependence logic are written in Prolog and require a Prolog interpreter for their execution. The dependence query interface is a C module linked in with the compiler and graphical interface. The *data dependence engine* is responsible for accepting dependence queries and returning a result appropriate for the query.

Prolog Control Flow Graph

The Prolog control flow graph is a representation of the *control flow graph* in the Prolog language suitable for the *data dependence engine* . Based on the *control flow graph* the compiler generates Prolog source code that describes the control flow of the input program. This source code is the interface between the *data dependence engine* and the input program. Without this representation of the program, the *data dependence engine* would not understand the makeup of the program it was interpreting. The Prolog control flow graph is discussed in detail in Section 3.3.

Data Dependence Logic

The data dependence logic controls the *data dependence engine* . This Prolog source code encodes the logic rules by which the *data dependence engine* determines the results of the dependence queries. The data dependence logic is responsible for the handling of dependence tests, query propagation (back-chaining), control flow, and substitution among other things. The logic for the source is completely described in Chapter 5 and the entire Prolog source code is given in Appendix A.

Dependence Query Interface

The dependence query interface is the mechanism by which dependence queries are sent to the *data dependence engine* . This interface is comprised of two levels of interface code: the dependence test interface and the communications interface. The dependence test interface is a series of C subroutines that are linked into the compiler that hide the details of how the query is actually being resolved. The communications interface is only necessary because the current implementation of the *data dependence engine* is in Prolog and there must exist a mechanism for bidirectional communications with the Prolog interpreter.

The communications between the *data dependence engine* and the interface is accomplished through the use of an Interprocess Communications mechanism called pipes [Ste92]. More precisely the mechanism is known as pty pipes. The Unix terminal facility is used to accomplish unbuffered communications between the Prolog interpreter and the in-

terface. Unbuffered communications is required because the Prolog interpreter puts out a set of prompt tokens that delimit the output of each query. The prompt tokens are not followed by a newline character, so without unbuffered communications the prompt tokens would never be flushed to the interface side of the pipe. This interface is depicted in Figure 1.3.

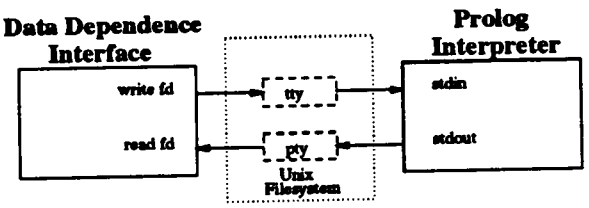


Figure 1.3: Pipe interface to Prolog interpreter

1.4.4 Graphical User Interface

A graphical user interface (GUI) has been developed to allow users easy access to the interactive environment. This interface allows users to load input programs, compile programs, and run dependence queries on selected statements and variables within the programs. Figure 1.2 only shows the communication between the GUI and the dependence query interface. This depiction is correct, but does not show the ability to load input programs and run the compiler analysis on these programs to generate the Prolog control flow graph. The interface will also automatically load the *data dependence engine* each time a program is compiled. A more detailed description of the interface functionality is given in Chapter 4.

Chapter 2

Language Definition

The language that is used for the input programs to this compiler is derived from the "Tiny Loop Restructuring Tool" [Wol90] with further extensions given by the Petit project [KMP+95]. These are excellent general purpose compiler research tools that accomplish a great deal of optimizations on sequential languages. The context-free grammar or BFN (Backus-Naur Form) [ARJ86] notation for the language is given in Appendix C. The language is similar to FORTRAN in syntax with some of the awkward logical operators removed (e.g. .LE. has become <=). A sample program of the language is given in Figure 2.1. This example program shows looping structures, conditionals, and declarations of both arrays and scalars.

The grammar of the language provides for vector and parallel constructs. These constructs were built in to provide not only for future work in processing these types of statements, but also for producing code for some parallel or vector machines given

```
   ! This is a sample program
1: integer i, j, q, w = 3, x, y, z
2: integer a(0 : 100)
3: if ( x ) then
4:     y = 2
5:     x = 4
   else
6:     x = 2
7:     y = 4
   endif
8: w = y + x
9: x = 2
10: y = 4
11: z = 3
12: for i = 0 to 10 do
13:     for j = 0 to 10 do
14:         a(x * i + x * j) = z
15:         q = a(y * i - w * j + z)
        endfor
    endfor
```

Figure 2.1: Sample input program

some optimizations to the scalar code. These constructs are not used in the language except for being parsed by the compiler.

Comments for the language are initiated by the '!' character. Any text following the comment character on a given line is ignored. Comments can appear either at the start of the line, thus making the entire line a comment, or after a program statement on a given line of text.

The following sections describe the different aspects of the language in more detail. The tables giving the syntax of a particular program construct use small portions of the grammar from Appendix C. Any production label that begins with **st** defines an actual program statement. All other productions are used for representation of smaller program constructs (i.e. sub-statement level). When a terminal symbol appears in all caps it represents a variable token that has been passed to the parser from the lexical analyzer (e.g. an integer number INT). All other non-variable tokens (e.g. **do**, +, etc.) are given as string literals in the grammar.

2.1 Declarations

Declarations are defined by the grammar in Table 2.1. A declaration can appear anywhere in the program text and may be initialized according to its type. A program variable must be declared before its use in the program text. This rule, however, does not apply to loop variables that can be assumed to be **integer scalars**. In these cases, a warning message will be generated by the compiler but compilation will proceed.

Sample declarations are given in Figure 2.1 at Statements 1 and 2.

Table 2.1: Syntax of declarations

stdecl	→	class type decllist
decllist	→	decl | decllist ',' decl
decl	→	newid init |
		newid '(' boundlist ')' init
init	→	'=' expr | ∅

2.2 Assignments

Assignments are shown in the grammar presented in Table 2.2. A general assignment statement is simply given by a **lhs** operator and the terminal symbol '=' followed by a **rhs** operator. The **rhs** operator may be either a constant or an expression. The language also supports C–like assignment operators, assignment statements that have an implied operator associated with them as in '+ ='. Before a variable may appear as a **lhs** operator it must be declared as described in Section 2.1; however, there is no provision in the compiler that checks for the definition of a variable (i.e. **lhs**) before it is used (i.e. **rhs**) in an assignment or expression.

2.3 Expressions

The syntax of an expression is given in Table 2.3. Expressions may appear in several places in a program text aside from their usual appearance as **rhs** operators to assign-

Table 2.2: Syntax of assignments

stasgn	→	lhs '=' expr
stasgnoper	→	lhs asgnoper expr
asgnoper	→	'+ =' | '* =' |
		'***max*** =' | '***min*** ='

ments. For example, an expression may be used as a bound for an array index or as the upper-bound to a loop structure. The complete use of expressions can be obtained from the language grammar in Appendix C. The associativity and precedence of operators is in standard arithmetic direction and order, respectively.

Table 2.3: Syntax of expressions

expr	→	oldid | oldid '(' list ')' |
		INT | FLOAT | '(' expr ')' |
		'+' expr | '−' expr |
		expr '+' expr | expr '−' expr |
		expr '*' expr | expr '/' expr |
		expr '∧' expr | expr '<' expr |
		expr '<=' expr | expr '==' expr |
		expr '! =' expr | expr '>=' expr |
		expr '>' expr | expr '%' expr |
		expr '***max***' expr | expr '***min***' expr |

2.4 Conditionals

Logical flow control is described in the syntax in Table 2.4. For conditionals the programmer is provided with a simple **if-then-else** construct for logic flow. The language does not support an **else-if** construct. It is possible, however, to embed **if** conditions in the statement list of another conditional. Logical operators are provided by the expression grammar given in Table 2.3. The conditional expression evaluates to **false** if the expression evaluates to 0 and **true** otherwise. This is typical of most programming languages.

Table 2.4: Syntax of conditionals

stif	→	**'if'** expr **'then'** stlist elsepart **'endif'**
elsepart	→	**'else'** stlist | ∅

2.5 Loops

There are three non-parallel, non-vector looping structures supported by the language, which are described in Table 2.5. The parallel/vector loops are not described in this section since they are not currently handled by the compiler except for parsing. For information on the parallel/vector loops see Appendix C for the productions **stforall**, **stparfor**, and **stdoany**. The three other loops that are defined provide adequate support for the investigation and research in this thesis and are fairly complete otherwise.

All of the loops have the standard lower-bound, upper-bound and step associated with them. The first **'do'** loop provides support for loops that contain only one loop statement. The next two loops are semantically identical, only the syntax is different. The first provides for a **for-do-endfor** type of loop while the second supports a **do-{}** syntax. Both provide for an "unlimited" number of loop statements in the body of the loop. The syntactical representation of the lower-bound, upper-bound and step of the loop has a varied syntax also, but all of them are semantically equivalent.

Table 2.5: Syntax of loops

stfor	→	**'do'** doid '=' expr forto expr optforby ';' stmt |
		'do' doid '=' expr forto expr optforby '{' stmtlist '}' |
		'for' doid '=' expr forto expr optforby **'do'** stmtlist **'endfor'** |
optforby	→	forby expr | ∅
forto	→	**'to'** | ',' | ':'
forby	→	**'by'** | ',' | ':'

Chapter 3

Compiler Design

The compiler is broken down into four distinct phases. First, the input file is read and a syntax tree is generated. Second, a *control flow graph* is created based on the generated syntax tree and the corresponding edges are linked to each node of the *control flow graph* . Third, once the *control flow graph* is created the Prolog control flow graph code is generated. The Prolog control flow graph code is generated from a combination of the *control flow graph* and the *syntax tree* . Finally, the compiler enters the optimization phase where different scalar tests are run on the source code at the request of the user.

3.1 Parser and Syntax Tree

As with any compiler the parser is responsible for both creating an internal representation of the syntax of the program and error checking the input program. In this implementation the parser checks for errors while simultaneously converting the pro-

gram into its internal representation.

There are two "basic" types of internal representations used: **abstract syntax trees** and **parse trees** [ARJ86]. These internal structures have been implemented in many different ways in many different compilers, and ours is no exception. In an abstract syntax tree all the leaf nodes represent operands while the interior nodes represent the operators. Evaluating program expressions into an abstract syntax tree makes compile time evaluation, for things like data dependence testing, much simpler. A parse tree, by contrast, is much easier to create and does not discard superficial elements of the language grammar from its nodal structure. For example, if the grammar has a production called, **optforby**, the parse tree would contain a node that represented the execution of that production.

The internal structure used is actually a combination of a parse tree and an abstract syntax tree. In the upper levels of our structure, some of the superficial elements of our language grammar are found. Using a variation of a parse tree for the upper statement nodes of the tree allows for easier manipulation of the actual language during compilation; for example, creating the *control flow graph* . In the lower levels of the tree (i.e. expressions) the definition of an abstract syntax tree is adhered to. It is necessary, in scalar and dependence analysis, to perform compile time computations of expression. Having expressions in the form of an abstract syntax tree makes these computations much simpler. This lower level abstract syntax tree design also aided us in the generation of three address code for the Prolog *data dependence engine* . The

internal representation will be referred to simply as a *syntax tree* .

The *syntax tree* is created as productions of the grammar are executed. After the evaluation of each production set in the grammar, a node is created and linked into the *syntax tree* . The *syntax tree* node format is described in Table 3.1. Given an input program, the *syntax tree* is created with a root node called the start node. From the start node a list of children is generated, each of which is of class *statement.* [1] Any *syntax tree* node that is of class *statement* is given a statement number in its *syntax tree* node. Any statements that are made up of more sub-statements, such as conditionals and loop structures, are give a node called a *statement list*, which is the parent node of the sub-statements for that statement node. The sub-tree of any particular *statement* node is relative to that node. In other words, each subclass of a statement has a particular format of its own. For example, given the input program in Figure 3.1 the generated *syntax tree* would look like Figure 3.2. This example shows the two distinct levels of the *syntax tree* that are used by the compiler. The *syntax tree* data structure is given in Appendix B.

3.2 Control Flow Graph

Once the *syntax tree* has been created and the program has been determined to be correct by the parser the compiler moves on and creates a *Control Flow Graph* (CFG). A *CFG* is used to model the control flow of a program [ZC91]. A *CFG* is made up of

[1] Each production in the grammar (see Appendix C) which begins with "st" is of class *statement*

```
1:  integer x,y,z
2:  x = 1
3:  y = 3 + x * 4
4:  if( x < y ) then
5:      z = y - x * 5
    else
6:      z = x - y * 5
    endif
7:  x = z + y
```

Figure 3.1: Simple input program for the syntax tree

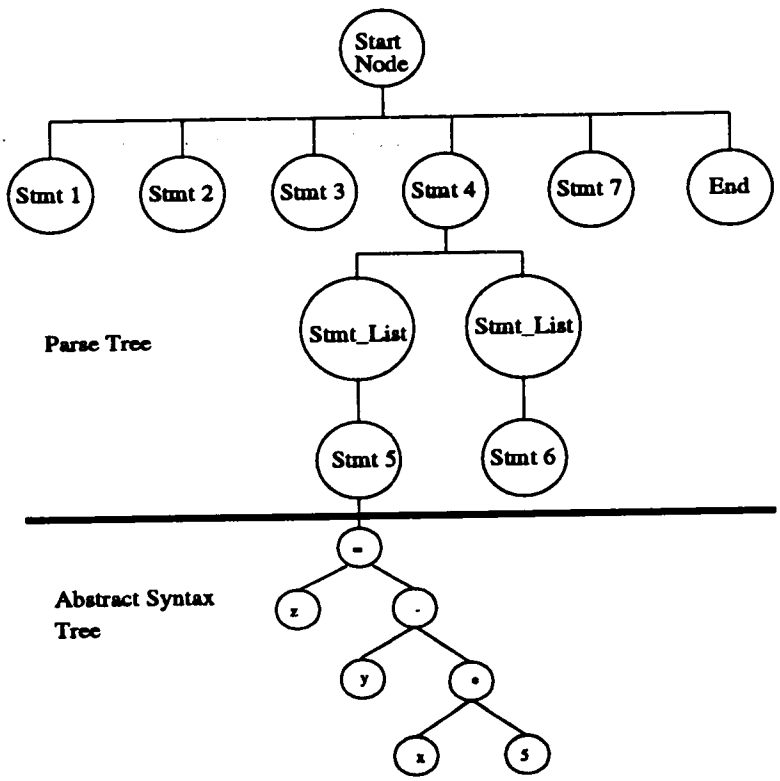


Figure 3.2: Internal syntax tree example

Table 3.1: Elements of the syntax tree node.

Class	main type of node that this represents
Subclass	subclass of the class
Stmt_number	the sequential number of the statement
Exprvar	Prolog expression that this represents
Value	the value of the node (if applicable)
Parent *	the parent node of this node
Children *	list of the children of this node
Next_stmt	the next statement node in the program

nodes and edges. A node corresponds to a single program statement. Edges represent the transfer of control between two nodes in the graph. Both the edges and the control nodes are depicted with separate data structures and each is assigned a unique number. The structure of a *CFG* node and a *CFG* edge are described in Tables 3.2 and 3.3 respectively. The *CFG* generated from the code in Figure 3.1 is illustrated in Figure 3.3.

Table 3.2: Elements of the *CFG* node.

Num Predecessors	Number of predecessor edges
Predecessors	List of edges that point to predecessor nodes
Num Successors	Number of successor edges
Successors	List of edges that point to successor nodes

Table 3.3: Elements of the *CFG* edge.

Source node	Node from which this edge originated
Destination node	Node where the edge terminates
Edge Name	Name of the edge for Prolog
Edge Number	Number of the edge

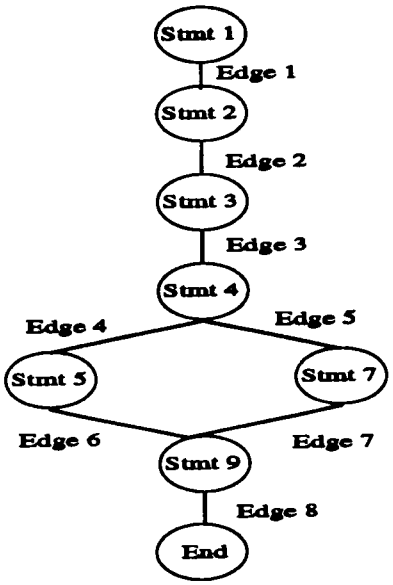


Figure 3.3: Control flow graph example

3.3 Prolog Control Graph Code Generation

The interface to the Prolog *data dependence engine* is made up of three parts: the *data dependence engine* source code, the Prolog control flow graph, and the query interface. At this point the discussion moves to the generation of the Prolog control flow graph, which will be referred to as the *Prolog control graph* .

The *Prolog control graph* is a series of Prolog predicates that when put together form a control graph. These predicates describe the input program to the *data dependence engine* as shown in Figure 3.4. The predicates that describe the input program in the *Prolog control graph* are considered to be facts about the program. These predicates are different than those in the *data dependence engine* as they require no proof to be true, they are simply asserted. The purpose of the *Prolog control graph* is to describe

the facts about the program being input to the *data dependence engine* . From this description answers to queries are derived by the *data dependence engine* .

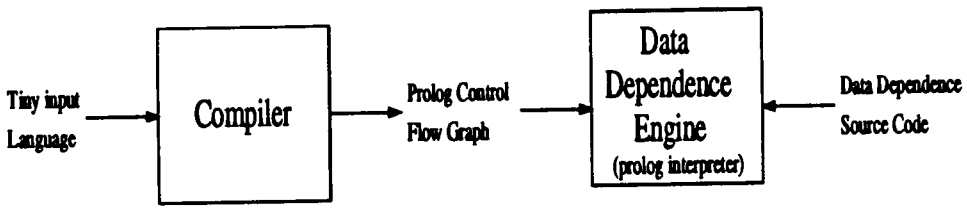


Figure 3.4: Flow of input language into data dependence engine

3.3.1 Edges and Nodes

Both edges and nodes are equally important in the understanding of the *Prolog control graph* . Nodes of the *Prolog control graph* are represented by Prolog predicates, for example:

$$\mathbf{asg}(\boldsymbol{in}, \boldsymbol{out}, \boldsymbol{lhs}, \boldsymbol{rhs})$$

In this example, the **asg** predicate is a single node in the *Prolog control graph* . Edges of the *Prolog control graph* are represented implicitly by the parameters to the nodal predicates. The ***in*** and ***out*** parameters to the **asg** predicate represent the in and out edges of the node, respectively. By tracing through a series of Prolog predicates, given in order, one can distinguish the connecting edges of the *Prolog control graph* , given as parameters to the Prolog predicate. Figure 3.5 shows a mapping of a simple input program to a *Prolog control graph* . In this example, the simple program has two assignment nodes (see Section 3.3.2), each assignment node has an in edge and an out edge. Nodes are designated as n_x for this example. Edges are show as e_n where for

each value of n, e is a unique edge. One would follow the control flow by starting with the initial in edge e_1 which connects the implied start state with node 1. Then the out edge of node 1, e_2, connects node 1 with node 2. Finally, the out edge, e_3, of node 2 connects node 2 with an implied stop node. This is a simple example of how the nodes and edges in the *Prolog control graph* are represented. In the following sections each predicate, its parameters, and the parameters meanings to the programs *Prolog control graph* will be described.

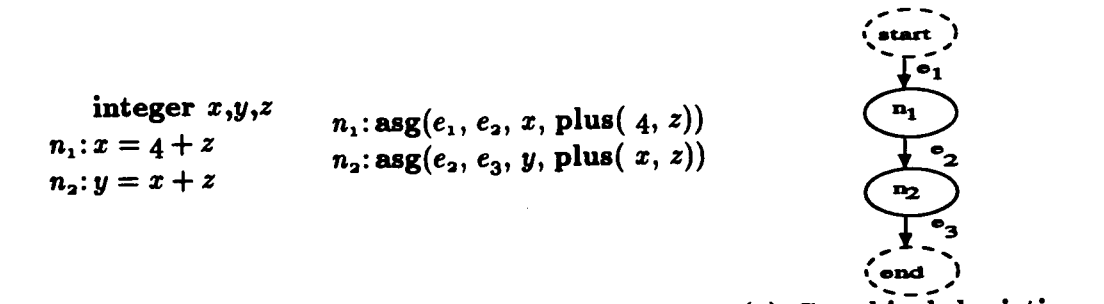


Figure 3.5: Mapping of input program to Prolog *CFG*

3.3.2 Assignments

An assignment is the simplest form of a Prolog control node. An assignment predicate merely states a fact about the assignment of a value to a variable. The predicate takes the form:

$$\mathbf{asg}(in, out, lhs, rhs)$$

The **asg** predicate is of arity 4 [2] (written **asg/4**) and hence takes four parameters: the in edge name (in), the out edge name (out), the left hand side variable (lhs) and the right hand side variable (rhs), which could also be an operator (see Section 3.3.6).

3.3.3 Loops

The **loop** predicate differs from the **asg/4** in that only control branching in a loop clause is represented. The loop index variable and the ranges are not defined in the Prolog context. Loops defined in the input language are **assumed** to be normalized. A loop is considered to be normalized if and only if its lower bound is the constant 0 and the loop step is equal to +1. A **loop/5** predicate has the following form:

$$\textbf{loop}(\textit{lnum}, \textit{in}, \textit{into}, \textit{back}, \textit{out})$$

The Prolog **loop/5** predicate takes as its parameters: a loop number (lnum), an in edge (in), the loop into edge (into), the back edge (back), and an out edge (out). These edges are depicted in Figure 3.6.

3.3.4 Conditionals

A conditional is the construct that allows for branching based on a given logical expression. The conditional that is use in the *Prolog control graph* is referred to as a switch statement and has the following form:

$$\textbf{sw}(\textit{in}, \textit{true}, \textit{false}, \textit{cond})$$

[2] Arity in Prolog is the number of parameters

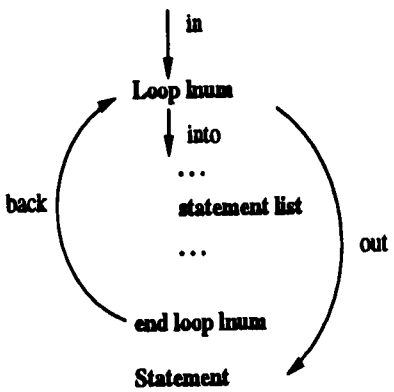


Figure 3.6: Loop edges

The Prolog **sw/4** predicate takes as its parameters: an in edge (in), a true edge (true), a false edge (false) and a conditional (cond). The meaning of these edges are given in Figure 3.7.

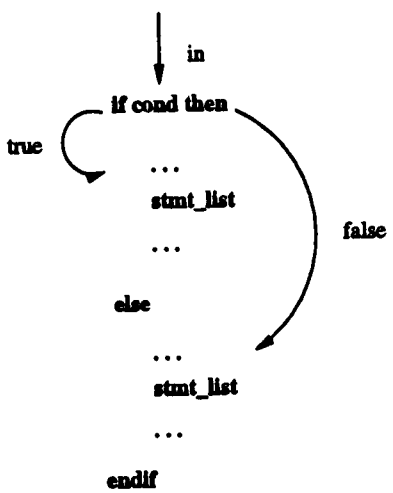


Figure 3.7: Conditional edges

3.3.5 Merges

Merge nodes are used when a node is determined to have two in edges coming to it. Merge nodes are used commonly on the statements which immediately follow conditionals. A merge node has the following form:

$$\mathbf{mg}(in_1, in_2, out)$$

The Prolog **mg/3** clause has two in edges (in_1 and in_2) and a single out edge (out) as its parameters. A merge node is not used as a programming construct but is used only in the internal control flow of the program. In other words, there is no statement in the input program that directly maps to a merge node in the *Prolog control graph* . Figure 3.8 shows the relation of a merge node to an input program. The edge depicted by m_n is the nth generated merge edge.

3.3.6 Operators

There are several operators that are supported with the **asg/4** and **sw/3** predicates. These operators define such things as binary mathematical operators and logical operators. The following is a list of the supported operators:

- **plus**(x, y) defines addition of two numbers
- **mul**(x, y) defines multiplication of two numbers
- **sub**(x, y) defines subtraction of two numbers
- **equal**(x, y) defines logical equality

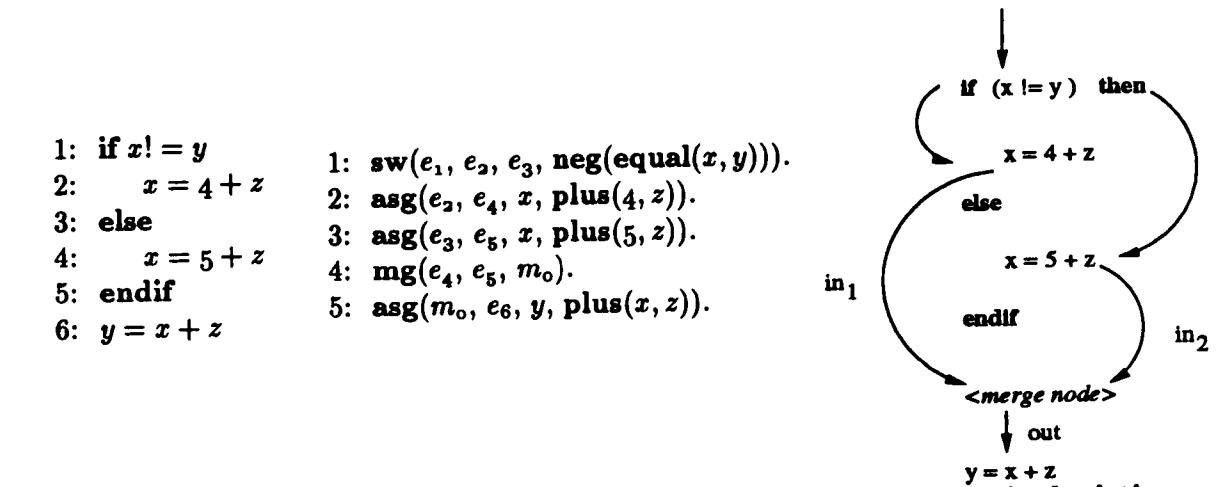


Figure 3.8: Merge node usage relative to an input program

- **neg(x)** defines unary negation

Figure 3.8 shows the usage of several of these operators. The *data dependence engine* does not, by default, have associated with it the normal operators that are present in most programming languages. Every operator that is to be used must be defined in the *data dependence engine* so that flow and substitution can be defined for the operator. Different tests may need to handle operators differently. For example, the induction test handles subtraction differently than the constant propagation test. This will be discussed in more detail in later chapters.

3.3.7 Three Address Code

All expressions that are presented to the *data dependence engine* must be represented in *three-address code. Three-address code* consists of a sequence of instructions, each of which has at most three operands [ARJ86]. From this definition of *three-address code* it is also apparent that each expression has at most one operator in addition to the assignment. This mapping of "complex" expression into three address code is handled by the compiler when the *Prolog control graph* is generated. An example of the mapping of a "complex" expression into three address code in presented in Figure 3.9. Temporary variables are denoted by tmp_n where for every value of n, tmp is a unique temporary variable. Also, temporary edges or "fake" edges are denoted by t_n where for each value of n, t is a unique edge. Notice in Figure 3.9 that aside from the need to create temporary storage variables, there is also need to create "fake", edges, but the main in edge and out edge of the original expression are preserved (i.e. e_1 and e_2).

(a) Input program	(b) Input three-address	(c) Prolog three-address
$x = 4 + z * y - x$	$tmp_1 = z * y$	**asg**(e_1, t_1, tmp_1, **mul**(z,y)).
	$tmp_2 = 4 + tmp_1$	**asg**(t_1, t_2, tmp_2, **plus**(4,tmp_1)).
	$tmp_3 = tmp_2 - x$	**asg**(t_2, t_3, tmp_3, **sub**(tmp_2,x)).
	$x = tmp_3$	**asg**(t_3, e_2, x, tmp_3).

Figure 3.9: Mapping of three-address code to Prolog

Chapter 4

Graphical Interface

An X windows interface has been developed for the *data dependence engine* . The interface was written using the OSF/Motif widget set. Figure 4.1 shows the main screen of the interface. The interface allows users to load programs, compile them and perform the various tests using the *data dependence engine* . The interface is point–and–click and is based on statements of the input program thus hiding the implementation details of edges and three-address code from the user. The user simply has to select the statement(s) upon which he wishes perform the test. Next, he selects the variables from within these statement(s). Once the variables and statements are selected the users executes the desired tests. An internal database is kept for each statement. This database contains the facts that have been discovered about the statement.

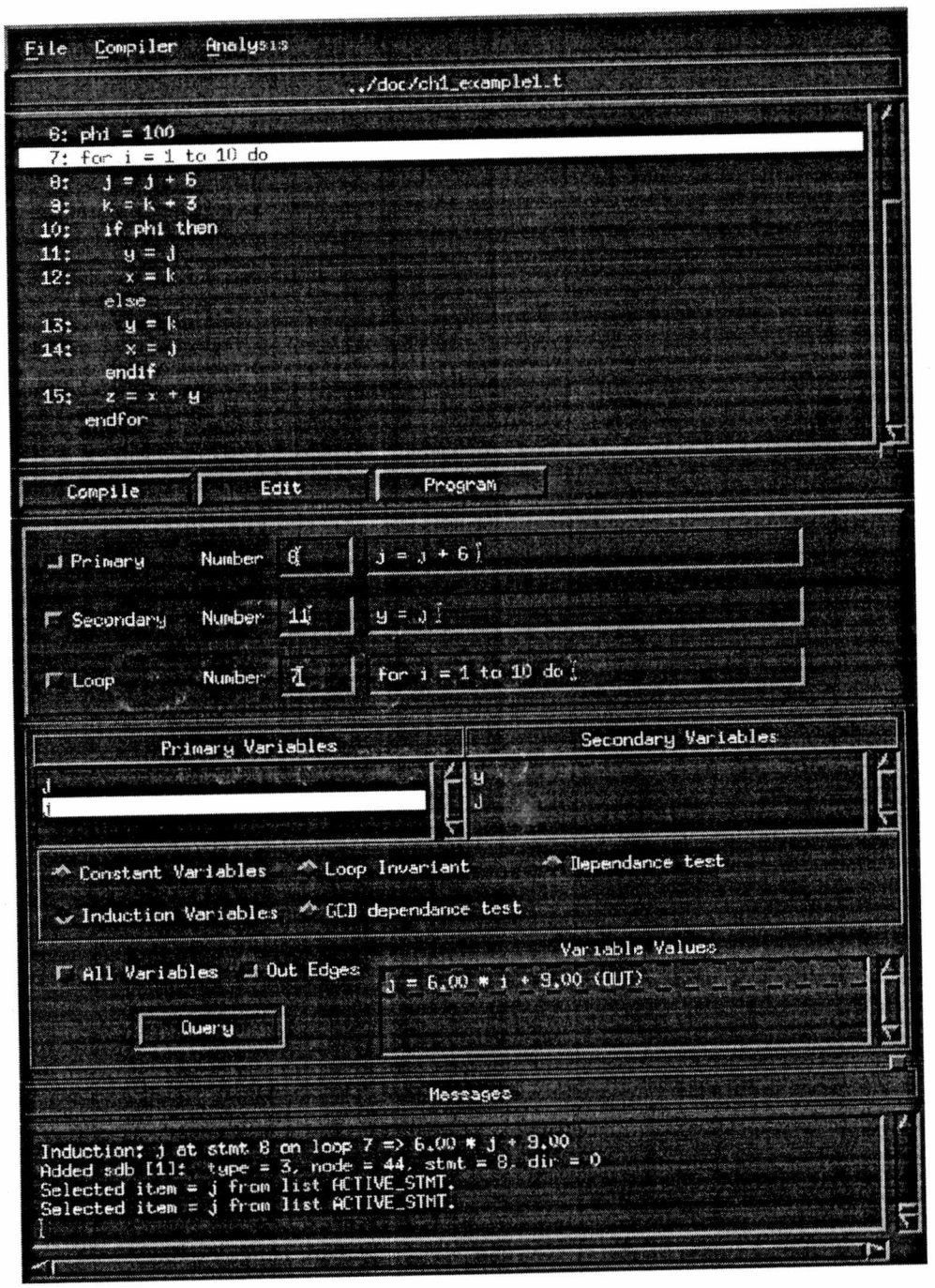


Figure 4.1: Main GUI interface window

4.1 Compiling Programs

Once the program is loaded into the interface, the program must next be compiled. The currently loaded program file is listed in the text box above the top scrolled window. The user may compile the program via the *"Compile"* button which is located just below the scrolled program window. The compiler may also be invoked from the *"File"* option of the Main Menu. After compilation, an internal representation of the program will be displayed in the program window which should very closely match the original text. The display of this internal representation will have blank lines removed and there will be statement numbers inserted to the left of the statements.

The current program may be viewed by selecting the *"Compiler"* option from the main menu bar and then selecting *"Current Program Text"* . Alternatively, the user may push the *"Program"* button under the top scrolled program window.

4.2 Selecting Statements

In order to perform tests on the program, a statement must first be selected. Some tests require the selection of two statements in addition to a loop statement (see Section 4.5). Statements are selected from the statement editor, located just below the first row of buttons. Once the statement editor contains multiple statements, there are several ways to select a particular statement from within the statement editor. One such method is to simply depress the appropriate button directly left of the statement window. For example, if the user wished to select the primary statement, he would push

the *"Primary"* button and then double–click on the desired statement. Alternatively, the *"Edit"* button may be pushed, thus placing the selected statement into the statement editor. Another method is to type the number of the statement that you wish to edit into the appropriate *"Number"* box. Finally, for tests that require the selection of two or more statement an initial statement must first be selected by clicking on it, then while holding down the control key additional statements, even a loop statement, may be selected. Once these statements have been selected, the *"Edit"* button will place them into the statement editor. The program will determine which statement is the loop statement and put it into the appropriate location.

4.3 Variable Selection

Once a statement is selected, the variables for the statement will appear in the appropriate *"Variables"* windows. The user may either have just the variables for the selected statement appear in the *"Variables"* window or the *"All Variables"* toggle button may be selected to include every variable in the program into the *"Variables"* windows. One variable may be chosen at a time and variables are selected by simply clicking on the name of the variable in the appropriate window.

4.4 In/Out Selection

Tests are performed based on an edge or a direction. The user may decide in which direction the test should be performed. By default it is assumed that the test will be

performed based on the in–edges to the currently selected statement, thus performing the test prior to the execution of the currently selected statement. Conversely, if the user selects the *"Out Edges"* toggle button the tests will be performed on the out–edges from the currently selected statement, thus performing the test after the execution of the currently selected statement. The selection of this option may have a very different effect on the outcome of a given test. For example, consider Figure 4.2. In this example the variable x is defined when the predicate p is either true or false. When p evaluates to true, x is defined to be A and when p evaluates to false x is defined to be B. It is assumed for this example that the state of p cannot be asserted by the compiler. If one evaluates the test for constants at the statement $x = C$, then the evaluation will be performed on edge e_3, by default. Since at edges e_1 and e_2 the value of x is A and B, respectively, x cannot be proven to be a constant at e_3. If the test were run on the out edge e_4 of the statement $x = C$, then the previous definitions of x do not reach this edge, because x is redefined to be C. Hence, it can be proven that $x = C$ at edge e_4. This example shows how choosing the correct edge to perform the test will have an effect on the outcome of the test.

4.5 Executing Tests

Tests are executed in the interface by selecting the appropriate statements and variables, selecting the desired test and then hitting the *"Query"* button. Any messages that the test needs to give to the user will be displayed in the *"Messages"* window. A dialog

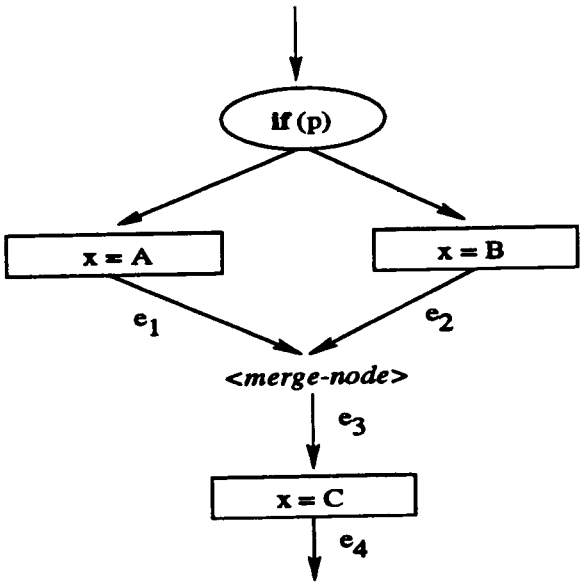


Figure 4.2: Edge direction example

box will appear with the results of the query and the *"Variable Values"* window will then display the appropriate information. The discussion of how to setup each test in the interface will be presented in the following sections. The details of each test are saved for discussion in Chapter 5.

4.5.1 Constant Variables

Constant propagation is a test that is performed on a single statement with a single variable selection. For this test the user must select one of the *"Primary"* or *"Secondary"* statements and a variable that is in the appropriate variable window. Then select the *"Constant Variables"* test.

4.5.2 Induction Variables

The induction variables test is performed on a single variable within a loop statement. With this test, the user must select one of the *"Primary"* or *"Secondary"* statements and a variable from the appropriate variable window. The user must also select a loop statement that contains the statement and variable and then the *"Induction Variables"* test must be selected.

4.5.3 Loop Invariance

The loop invariance test is much like the induction variable test in that it requires the selection of a statement, a variable from that statement and a loop statement. After this, the user must select the *"Loop Invariant"* toggle button which performs the query.

4.5.4 GCD Dependence Test

The Greatest Common Denominator dependence [ZC91] test is one of the tests in the interface that is written in the compiler interface with queries to the Prolog engine. The Prolog engine determines particular facts about the expressions in question. The test itself is performed in the compiler and not in the *data dependence engine* . When the test approaches a point where it needs some information about the expression, it will ask the *data dependence engine* for that information. For example, the test may need to determine if the bounds of a loop are constant. To this end, it will retrieve this information via a query to the *data dependence engine* . For this test the user is

required to select two statements and a loop. From the statements the user must select an expression or variable from each and the select the *"GCD dependence test"* toggle button from the test list.

4.5.5 Dependence Test

The dependence test is written completely in the *data dependence engine* and is an implementation of the GCD dependence test. Data dependence information is propagated through the CFG in this test whereas in the previous GCD dependence test only the information that is gathered from the *data dependence engine* is propagated through the CFG. This test will be discussed in more detail in Section 5.7. For this test to be performed from the interface, the user must select the same information as in the GCD dependence test: two statements, a loop, and a variable or an expression from each statement. Then the user selects the *"Dependence test"* toggle button to perform the test.

4.6 Internal State Database

Every node in the CFG has associated with it an internal database of facts that have been discovered about that particular statement. These facts are displayed in the *"Variable Values"* windows near the bottom of the interface. This internal database is not related to the assertion cache that is maintained by the *data dependence engine* (Section 5.3). The internal state database that is maintained by the interface is strictly for

display purposes. This database is not consulted before the execution of a dependence query (i.e. it is not a cache). Each selection of a statement displays the facts that are currently known about that statement. This display could include facts about the value of a constant that has been discovered, facts about any independences that are known about this statement, as well as any other relevant fact that have been discovered.

Chapter 5

Data Dependence Logic

The data dependence logic, in this context, is the portion of the *data dependence engine* that describes the logic rules for the dependence tests that are performed from the dependence query interface on the Prolog control graph predicates (described in Section 3.3). Remember that the *data dependence engine* is made up of three distinct parts as shown if Figure 5.1. Each portion of the *data dependence engine* is read into the Prolog interpreter, except for the interface, and the combined elements makeup the *data dependence engine* . In Figure 5.1 the portion that is discussed in this chapter is labeled **Data Dependence Source**; this will be referred to as the *data dependence logic* for the remainder of this chapter.

The *data dependence logic* make use of a couple properties in its operation:

- Assertional logic
- Backward chaining

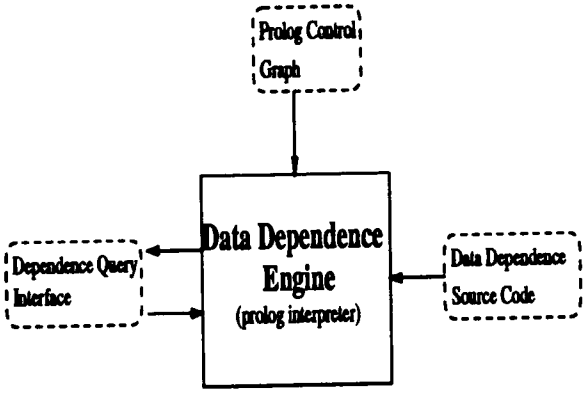


Figure 5.1: Data Dependence Engine Components

The description of the assertional logic and the use of that assertional logic in the development of a backward chaining algorithm is essential to the understanding of the *data dependence logic* .

5.1 Assertional Logic and Propagation

Assertional logic rules are formulated as program assertions of the form:

$$\phi \textbf{ on } E$$

where ϕ is a predicate on program variable values and E is an edge in the CFG. The assertion ϕ **on** E holds if the predicate ϕ holds on E when program control reaches E on any input to the program [Bec94]. This describes how predicates on a particular edge E of the CFG are evaluated. The assertion ϕ **on** E is propagated by the logic described in Table 5.1.

Given these assertional logic statements, some logic that is defined on a CFG G is

now formulated. A CFG is defined to be a triple $G = \langle N, E, s \rangle$ where N is the set of all nodes, $s \in N$ is the *start node*, and E is the set of all edges. There is a path from s to every node $n \in N$ [ZC91]. The algorithm is computed with the configuration $\langle E, \sigma \rangle$ where $E \in G$ is an edge and σ is a mapping from program variables to their values. It is assume that the CFG has an initial set of variable states σ_0. $[G]$ of a CFG G is defined to be the mapping from an initial state to the set of all reachable configurations of G. Finally, for a set S of input configurations and a CFG G the following definitions are formulated:

$$S \models \phi \text{ on } E \iff \forall \langle E, \sigma \rangle \in S. \sigma \models \phi$$

$$G \models \phi \text{ on } E \iff \forall \sigma_0. [G](\sigma_0) \models \phi \text{ on } E$$

Informally, given a set of configurations S one can say that S satisfies the the predicate ϕ at edge E if one can prove that for all the pairs $\langle E, \sigma \rangle$ in the set S the variable value state σ satisfies the predicate ϕ. The implementation of these rules is given in Table 5.2

Table 5.1: Logic rules for propagation

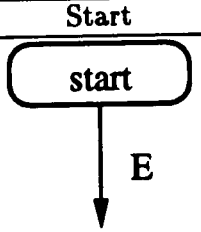
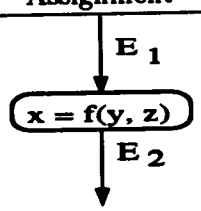
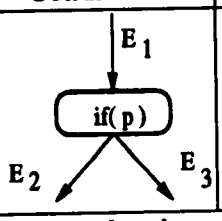
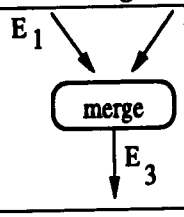
Start	Assignment	Conditional	Merge
			
(a) Graphic of CFG control nodes			
$\frac{\phi \text{ on } E}{failure}$	$\frac{\phi(f(y,z)) \text{ on } E_1}{\phi(z) \text{ on } E_2}$	$\frac{\phi \text{ on } E_1}{\phi \text{ on } E_2 \wedge \phi \text{ on } E_3}$	$\frac{\phi \text{ on } E_1 \wedge \phi \text{ on } E_2}{\phi \text{ on } E_3}$
(b) Assertional logic on CFG nodes			

Table 5.2: Propagation implementation

```
% Start
on1(Q, false, e1) :- !, fail.
% Assignment
on1(Q, P, E2) :-
    asg(E1, E2, X, Y), subst(P, Y, X, P1), on(Q, P1, E1).
% Conditionals
on1(Q, P, E2) :- sw(E1, E2, E3, equal(X, Y)),
    subst(P, Y, X, P1), on(Q, P1, E1).
on1(Q, P, E2) :- sw(E1, E2, E3, equal(X, Y)),
    subst(P, X, Y, P1), on(Q, P1, E1).
% Merges
on1(Q, P, E3) :- mg(E1, E2, E3), on(Q, P, E1), on(Q, false, E2).
on1(Q, P, E3) :- mg(E1, E2, E3), on(Q, P, E2), on(Q, false, E1).
```

5.2 Backward Chaining Algorithm

Given the assertional logic defined in Section 5.1 the backchaining algorithm ***Backchain***(g) can now be defined. This algorithm is defined as a subroutine with one parameter, which is the goal that is to be proved. ***Backchain***(g) is defined as a recursive routine. The rules are written as $\frac{A}{C}$ such that A is the list of *antecedents* and C is the list of *consequences*. Given a goal (i.e. *consequence*), the algorithm will match all the *consequences* and attempt to prove the goal. It proves the goal by attempting to prove the *antecedents* of the *consequence*. A *consequence* without an *antecedent* is a fact and is always true. By this algorithm the proof of the goal g proceeds backwards through the CFG of the program, which is defined as a list of facts (Section 3.3). The ***Backchain***(g) algorithm is listed in Figure 5.2.

1. For each *consequence* that matches goal g
   (a) If *consequence* has a list of *antecedents*:
      i. Execute *Backchain*() on *antecedent* a as $Backchain(a)$
      ii. If $Backchain(a) = true$ return *success* else *failure*
   (b) Otherwise:
      i. Return *true*
2. If all *antecedents* return *true* then return *success* else *failure*

Figure 5.2: $Backchain(g)$ algorithm definition

5.3 Assertion Cache

The $\boldsymbol{Backchain(g)}$ algorithm from Section 5.2 is optimized by adding an *assertion cache* to the *data dependence logic* . The *assertion cache* is defined by a set of facts about the program that are accumulated by the rules of the *data dependence logic* . Informally, as the rules of the *data dependence logic* are executed, facts about the program, both successful and unsuccessful, are being discovered. As these facts are discovered, they are added to the *assertion cache* for future executions of $\boldsymbol{Backchain(g)}$.

Formally, the *assertion cache* χ is defined by a set of facts φ. Each *assertion cache* entry is defined as $\varphi(\phi, E)$, where ϕ is a predicate and $E \in G$ is an edge in the CFG. The notation $\varphi(\phi, E)$ denotes that a fact ϕ is true on edge E while $\boldsymbol{failure}(\varphi(\phi, E))$ denotes a fact in the contrary about the program. φ by itself if assumed to mean $\varphi(\phi, E)$ and the shorthand $\boldsymbol{failure}(\varphi)$. Each entry $\varphi \in \chi$ is uniquely defined. It is further defined that $\varphi < \boldsymbol{failure}(\varphi)$. When an entry is defined as a success and failure, the maximum of the entries is taken. The *assertion cache* is updated with the

following update function:

$$\chi + \varphi = \chi \cup \{\varphi\}$$

A similar function is used for a $\boldsymbol{failure}(\phi)$ update. Given the presence of the *assertion cache*, the $\boldsymbol{Backchain}(g)$ algorithm is now modified. The new algorithm is defined in Figure 5.3 [Bec94].

1. If g is mentioned in the cache
   (a) if $g \in \chi$ then return success
   (b) if $failure(g) \in \chi$ then return failure
2. Otherwise
   (a) Match g against the rules to find a matching *consequence*
   (b) If there is no matching *consequence*, let $\chi = \chi + failure(g)$ and return failure
   (c) Else, foreach rule whose *consequence* matches g
      i. If *consequence* has no *antecedents*
         A. Let $\chi = \chi + g$
         B. Return true
      ii. Else
         A. Execute *Backchain*() on *antecedent a* as *Backchain*(a)
         B. If *Backchain*(a) = *true*, let $\chi = \chi + a$ return *success*
   (d) If all *antecedents* return *true* then, let $\chi = \chi + g$ and return *success*
   (e) Else, let $\chi = \chi + failure(g)$ and return *failure*

Figure 5.3: *Backchain*(g) algorithm with assertion cache

5.4 Substitution of Expressions

In the backward propagation of a predicate ϕ, certain substitutions must be made in order to prove ϕ **on** E. This is especially true when a predicate is propagated back

through an assignment node. In order to prove ϕ for a particular variable x, the assignment of x to another expression e must be considered. The notation $\phi[e|x]$ is used to denote the resulting substitution of expression e for every occurance of x in ϕ. With this and given the graphic for assignment in Table 5.1 one can use the following additional assertional logic rule, where $e = f(y,z)$:

$$\frac{\phi[f(y,z)|x] \text{ on } E_1}{\phi \text{ on } E_2}$$

If x does not occur in ϕ, then the resulting substitution rule will have no effect on ϕ. For example, lets say one is trying to prove $const(x)$ and in the back propagation of this predicate one encounters a statement:

$$x = y + z$$

and the in edge to this statement is E_1 and the out edge is E_2. In order to prove $const(x)$ on E_2 one must now prove $const(y+z)$ on E_1 and thus must substitute into our predicate the value $y + z$. Table 5.3 gives a sample of the Prolog substitution code that is implemented in the *data dependence logic* . The Prolog predicate ***subst*** is of arity 4 and in its simplest form is written $subst(Z, X, Y, Z)$, which means in Z find all occurances of X and replace them with Y, the final Z will be the result. So to continue with the example above one would try to prove the goal $on_1(all, const(x, C), E_2)$, which matches the *antecedent* $on_1(Q, P, E_2)$ in Table 5.3. The *consequence* of this *antecedent* is to find an assignment that has an out edge E_2. One can then make the

Table 5.3: Substitution implementation

```
subst(Y, X, Y, X) :- !.
subst(Z, X, Y, Z) :- number(Z), !.
subst(val(N), X, Y, val(N)) :- !.
subst(Z, X, Y, Z) :- atom(Z), \+ Y = Z.
subst(const(X1, Y1), X, Y, const(X2, Y1)) :- subst(X1, X, Y, X2).
on1(Q, P, E2) :-
    asg(E1, E2, X, Y), subst(P, Y, X, P1), on(Q, P1, E1).
```

substitution using

$$\boldsymbol{subst}(\boldsymbol{const}(\boldsymbol{x}, \boldsymbol{C}), \boldsymbol{x}, \boldsymbol{y}+\boldsymbol{z}, \boldsymbol{const}(\boldsymbol{X2}, \boldsymbol{C})),$$

which transforms the ***subst*** into its simple form. For any $\boldsymbol{x}$, one can replace $\boldsymbol{x}$ with $\boldsymbol{y}+\boldsymbol{z}$ and store the result in $\boldsymbol{X2}$; hence in the original ***subst*** there will be $\boldsymbol{const}(\boldsymbol{y}+\boldsymbol{z}, \boldsymbol{C})$. After the substitution one can try to prove the new predicate $\boldsymbol{const}(\boldsymbol{y}+\boldsymbol{z})$ on the in edge $\boldsymbol{E_1}$.

5.5 Loops and their Properties

The concept of loops in the *data dependence logic* will now be formally introduced. A loop is formally described by the five-tuple $\langle \boldsymbol{n}, \boldsymbol{A}, \boldsymbol{B}, \boldsymbol{C}, \boldsymbol{D} \rangle$, where $\boldsymbol{n}$ is the loop number, $\boldsymbol{A}$ is the *in* edge, $\boldsymbol{B}$ is the *into* edge, $\boldsymbol{C}$ is the *back* edge, and $\boldsymbol{D}$ is the *out* edge. For a graphical depiction of the meaning of these edges in a loop see Figure 3.6.

Given this notion of a loop, some quantifiers that allow one to express assertions on a particular iteration of a loop can be defined. The quantifiers $\mathbf{first_n}$, $\mathbf{this_n}$, and

$\mathbf{next_n}$ are defined to describe assertions on the *first*, *current*, and *next* iteration of the loop n. The following assertional logic rules describe the quantifiers meaning on a loop $\langle n, A, B, C, D \rangle$:

$$\frac{\phi \textbf{ on } A}{first_n\ \phi \textbf{ on } B} \qquad \frac{this_n\ \phi \textbf{ on } C}{next_n\ \phi \textbf{ on } B}$$

These quantifiers are used in the formulation of the induction variable algorithm for the *data dependence logic* . Informally, when the quantifiers are used as above, they express, for example, that the assertional logic statement ϕ **on** E is true on a particular iteration of loop n (i.e. $\mathbf{first_n}$).

5.6 Induction Variable Recognition

An Induction variable is a variable v in a loop L such that the expression of the variable can be rewritten in linear terms of the loops do-variable and have the form $v = c_0 + c_1 * i$. Where i is the loops do-variable and c_0 and c_1 are constants. Figure 5.4 gives an example of an induction variable in a program fragment. The discovery of such variables is important to the forthcoming dependence problem. In the mathematical formulation of the dependence problem in compilers there are a number of conditions that must be met in order for an expression to be admissible to the dependence test. One such condition is the **linearity condition** [ZC91], which states

$$f(I) = a_0 + \sum_{1 \le j \le n} a_j\, I_j,$$

```
j = 3                          j = 3
for i = 0 to 10 do             for i = 0 to 10 do
   ...                            ...
   j = j + 6                      j = 6 * i + 9
   ...                            ...
endfor                         endfor
```

(a) Initial program (b) Induction variable

Figure 5.4: Induction variable discovery

where $\mathrm{I} = (I_1, I_2, I_3, \ldots, I_n)$ is a vector of do-variables and a_0 and a_j are constants. This condition supposes that consideration has been taken to handle multi-dimensional arrays. Only one-dimensional arrays will be considered in the dependence test, hence $n = 1$. Given this condition that is put on the admissibility of expressions that are even considered for dependence testing, it is easy to see the importance of induction variable recognition and substitution.

The induction variable function has the form $induct(n, x, E)$, where n is a loop number, x is a program variable enclosed in loop n, and E is an edge in the CFG. The output of the function is either the values c_0 and c_1 such that $x = c_0 + c_1 * i_n$, where i_n is the do-variable for loop n or *failure*. The function in the assertional logic is expressed as follows:

$$induct(n, x, E) = \frac{first_n\ x = c_0\ \mathbf{on}\ E \wedge this_n\ x + c_1 = t\ \mathbf{on}\ E}{next_n\ x = t\ \mathbf{on}\ E},$$

where t is a temporary value to prove the induction of x.

5.7 Dependence Tests

There are several different algorithms for determination of dependence between two expressions (or vectors of expressions): the Exact test, Separability test, Banerjee test, GCD test and more. All of these test have certain properties, such as complexity, efficiency, and limitations that make one desirable over another. In most compilers these test are grouped into a function for checking dependence and they are applied in order of efficiency to attempt to find a dependence as quickly as possible. This function would have the following form:

$$dep_test(n, S, S', v, v')$$

where n is a loop, S and S' are statements enclosed in the loop n, and v, v' are subscripted variables in S, S' [ZC91]. The GCD test (Greatest Common Divisor) is used in this thesis. The extension of this *data dependence logic* would simply be a matter of implementation of additional algorithms (not under estimating their complexity).

5.7.1 Conventional problems with Reaching Definitions

The extensions and improvements made to the current algorithm do not actually change the fundamental makeup of the algorithm, nor do they represent the generation of a new algorithm. Rather they exploit the characteristics of existing algorithms in the framework of a backchaining algorithm. Consider the program fragment presented in Figure 5.5 and the application of the dependence test to statements S_{16} and S_{17} in loop L_5 and the variables x and y in S_{16} and S_{17}, respectively. The first thing a compiler

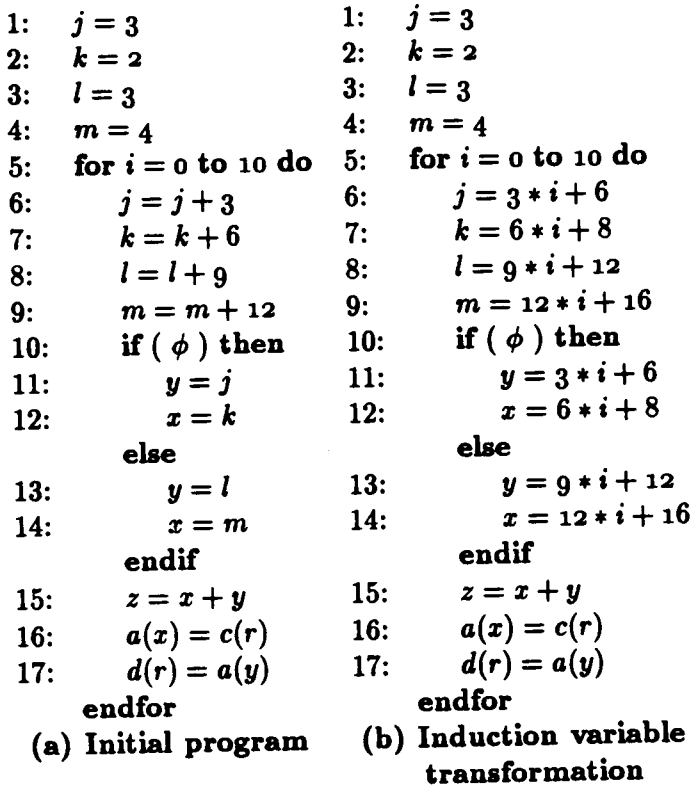


Figure 5.5: Data dependence in backchaining

is going to have to do is prove "something" about the variables x and y. In particular, the compiler will have to show either that x and y are constants or that they are linear functions of the do-variable i for L_5. It is very clear from a simple test that x and y are not constants. The only hope is to recognize x and y as variables that can be expressed in linear terms of i (i.e. induction variables). Given the current data flow analysis systems that exist and considering the specific problem of reaching definitions [ARJ86], the assertion that either x or y are induction variables is "killed" at statement S_{15}, since they would produce different values for c_0 and c_1 at the merge of the if-condition (see Section 5.6). Since neither x nor y can be expressed in linear terms of i, the variables would not be admissible to the data dependence test.

5.7.2 Dependence test cross-checking

Through application of the backchaining algorithm one could eliminate the concept of a reaching definition and propagate the dependence test back through the program until it can be proven that x does not depend on y. Informally, it can easily be shown that if you take the values of x and y and consider each branch of the conditional at S_{10} independently x and y are not dependent on each other. This is a sufficient test to show a loop independent independence. Now consider the merge of the expressions at S_{15}, if one would simply cross-check the dependence at the merge node one would successfully prove the independence of x and y through any loop carried dependence. The idea of dependence test cross-checking is depicted in Figure 5.6. In this example it is shown that one can disprove any loop carried dependence by a simple cross check

of the variables at the merge node. The initial check for loop independent dependence is done by the dependence tests denoted by A and B. If the tests A and B show a dependence, then the test is complete and the compiler moves on; however, if A and B cannot prove a dependence then tests C, D, and E are performed at the merge node to further attempt to prove independence.

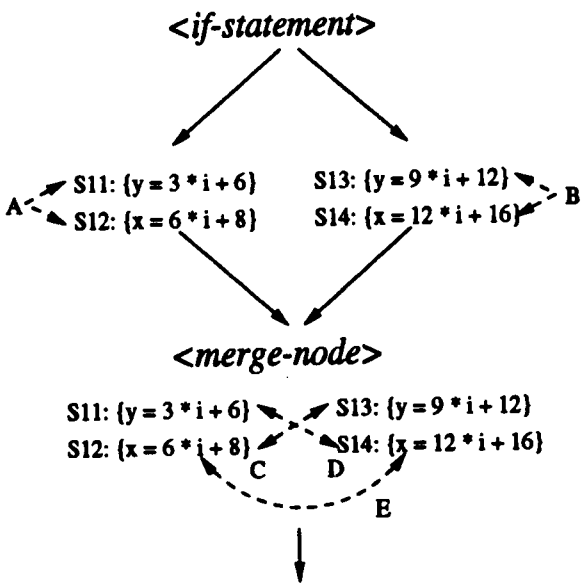


Figure 5.6: Dependence test depiction

Chapter 6

Future Work

There are a number of enhancements that could be added to the compiler environment and algorithms presented here. The following sections outline some interesting possible areas of future study.

6.1 Implementation of Data Dependence Engine in C

The current implementation has the *data dependence engine* , written in Prolog, being accessed through Unix pipes. Although the use of Prolog in the research and proof stages of this thesis offered a quick development platform, this is very inefficient. Controlling an interpreter over a Unix pipe is not practical for a real compiler. Implementing a Backchaining *data dependence engine* in a language such a C would offer greater flexibility and efficiency to the compiler. This would also expose the practical aspects of this thesis.

6.2 Multi-dimensional Arrays

Expanding the algorithms in the current *data dependence engine* to include multi-dimensional arrays would not be difficult. In fact, this extension could occur outside of the actual *data dependence engine* and be included in a higher level of the compiler. Multi-dimensional arrays can be handled by performing the dependence test on each dimension separately and then combining the results. A dependence exists only if it can be show that for a given admissible vector of do-variable values a dependence was shown in every dimension.

6.3 Implementation of more dependence tests

It would be advantageous to show how effective the use of backchaining is with a complete set of dependence tests. A good minimum set to start with would be those presented in [ZC91]: separability test, GCD test, and Banerjee test.

6.4 Code transformation in the compiler

The compiler should be extended to include code transformations into a language that is suitable for parallel or vector machines. This would give the user/programmer a version of the code to inspect after the optimizations are complete. Currently the compiler makes no attempt to transform the input program.

6.5 User Assertions from the interface

It would be nice to provide an interface to the compiler where the user/programmer could provide compile time assertions through the interface to the compiler for enhancing optimization. The compiler might generate some assertion that it needs to prove in order for a dependence test to succeed. In the event that the compiler cannot prove or disprove that assertion the compiler should ask the user for help and record the assertion made by the user in the program as a user assertion.

Bibliography

Bibliography

[ARJ86] Aho A.V., Sethi R., and Ullman J.D. *Compilers, Principles, Techniques, and Tools.* Addison-Wesley, Massachusetts, 1986.

[Bec94] Micah Beck. Deductive Data Flow Analysis. 1994.

[KMP+95] Wayne Kelly, Vadim Maslov, William Pugh, Evan Rosser, Titiana Shpeisman, and David Wonnacott. New User Interface for Petit and Other Extensions. 1995.

[Ste92] W. Richard Stevens. *Advanced Programming in the Unix Environment.* Addison Wesley, Massachusetts, 1992.

[Wie95] Jan Wielemaker. *SWI-Prolog 2.0 Reference Manual.* University of Amsterdam, Amsterdam, The Netherlands, 1995.

[Wol90] Michael Wolfe. *TINY: A Loop Restructuring Research Tool.* Oregon Graduate Institute of Science and Technology, Beaverton, Oregon, 1990.

[ZC91] Hans Zima and Barbara Chapman. *Supercompilers for Parallel and Vector Computers.* ACM Press, New York, 1991.

Appendix

Appendix A

Data Dependence Logic

```
%-----------------------------------------------------------
% Tautologies
%-----------------------------------------------------------
on(Q, P, E) :- failure(on(Q, P, E)), !, fail.
on(Q, P, E) :- P, !.

%-----------------------------------------------------------
% Dependance Test
%-----------------------------------------------------------
independant(X1, X2, L, E) :- gcd_test(X1, X2, L, E).

%-----------------------------------------------------------
% GCD test
%-----------------------------------------------------------
gcd_test(X1, X2, L, E) :- on(dep(L), indep(X,Y,L), E ).

on(dep(L), indep(X, Y, L), e1) :- !, fail.
on(dep(L), indep(X, Y, L), E) :-
        on(all, const(X, C1), E),
        on(all, const(Y, C2), E),
        !, neg(equal(C1,C2)).

on1(dep(L), indep(X, Y, L), E) :- gcd_dep( X, Y, L, E).

gcd_dep(X1, X2, L, E) :-
        atom(X1), atom(X2),
          write('gcd_dep('),write(X1),write(','),
          write(X2),write(','),write(E),write(')'), nl,
```

```
        on(circuit(L), induct(X1, A1, A0), E),
          write(X1), write(': A1 - '), write(A1),
          write('; A0 - '), write(A0), nl,
        on(circuit(L), induct(X2, B1, B0), E),
          write(X2), write(': B1 - '), write(B1),
          write('; B0 - '), write(B0), nl,
        gcd( A1, B1, G),
        !,
        write('GCD - '), write(G), nl,
        (( B0 == A0 ) -> write('fail'), nl, fail;
           (ANS is G mod (B0 - A0),
            write('Mod - '), write(ANS), nl,
            (((ANS =\= 0), (G =\= 1)) ->  write('true'), nl, true);
                                          write('fail'), nl, fail)).


%-----------------------------------------------------------------
% Induction Variables
%-----------------------------------------------------------------
induction(X, L, C1, C0, E) :- on(circuit(L), induct(X, C1, C0), E).

on(circuit(L), induct(X, C1, C0), e1) :- !, fail.
on(circuit(L), induct(X, C1, C0), E) :- induct_var(X, L, C1, C0, E), !.
on(circuit(L), induct(X, C1, C0), E) :- on(all, const(X, C0), E), !, C1 = 0.
%on(circuit(L), induct(X, C1, C0), E) :- inductexpr(X, C1, C0).

on1(circuit(L), induct(Z, C1, C0), E2) :-
     loop(L, ENTER, IN, BACK, OUT ),
     asg( E1, E2, Z, plus(X, Y)),
     on(circuit(L), induct(X, A1, A0), E1),
     on(circuit(L), induct(Y, B1, B0), E1),
     !, C1 is A1 + B1, C0 is A0 + B0.

on1(circuit(L), induct(Z, C1, C0), E2) :-
     loop(L, ENTER, IN, BACK, OUT ),
     asg( E1, E2, Z, sub(X, Y)),
     on(circuit(L), induct(X, A1, A0), E1),
     on(circuit(L), induct(Y, B1, B0), E1),
     !, C1 is A1 - B1, C0 is A0 - B0.

on1(circuit(L), induct(Z, C1, C0), E2) :-
```

```
    loop(L, ENTER, IN, BACK, OUT ),
    asg(E1, E2, Z, mul(X, Y)),
    on(circuit(L), induct(X, A1, A0), E1),
    on(circuit(L), induct(Y, B1, B0), E1),
    !, (((A1 == 0); (B1 == 0)) ->                          % If
            (C1 is (A0 * B1)+(A1 * B0), C0 is A0 * B0);    % Then
            fail).                                         % Else

on1(Q, induct(X, C1, C0), E2) :-
    asg(E1, E2, X, Y),
    on(Q, induct(Y, C1, C0), E1).

induct_var(X, L, C1, C0, E) :- loop(L, E1, E2, E3, E4),
    on(first(L), const(X, C0), E),
    prasserta(on(this(L), equal(X, t), E)),
    on(next(L), equal(X, plus(t, val(I))), E),
    constant(I, C1, E),
    prretract(on(this(L), equal(X, t), E)).

on1(first(L), P, E2) :- loop(L, E1, E2, E3, E4), !, on(all, P, E1).
on1(next(L), P, E2) :- loop(L, E1, E2, E3, E4), !, on(this(L), P, E3).

%-----------------------------------------------------------
% Loop Invariance
%-----------------------------------------------------------
invariant(X, L, E) :- constant(X, C, E).

invariant(X, L, E) :-
        prasserta(on(this(L), equal(X, t), E)),
        on(next(L), equal(X, t), E).

%-----------------------------------------------------------
% Constant Propagation
%-----------------------------------------------------------
constant(X, C, E) :- on(all, const(X, C), E).

on(Q, const(X, C), E) :- constexpr(X, C).
on(Q, const(X, C), e1) :- !, fail.
on(Q, const(X, Y), E) :- neg(equal(X, Y)), !, fail.

constexpr(C, C) :- number(C), !.
```

```
constexpr(plus(X1, X2), C) :- constexpr(X1, C1), constexpr(X2, C2),
        C is C1 + C2.
constexpr(mul(X1, X2), C) :- constexpr(X1, C1), constexpr(X2, C2),
        C is C1 * C2.
constexpr(sub(X1, X2), C) :- constexpr(X1, C1), constexpr(X2, C2),
        C is C1 - C2.

on1(Q, const(Z, C), E2) :-
    asg(E1, E2, Z, plus(X, Y)),
    on(Q, const(X, C1), E1), on(Q, const(Y, C2), E1),
    !, number(C1), number(C2), C is C1 + C2.
on1(Q, const(Z, C), E2) :-
    asg(E1, E2, Z, mul(X, Y)),
    on(Q, const(X, C1), E1), on(Q, const(Y, C2), E1),
    !, number(C1), number(C2), C is C1 * C2.
on1(Q, const(Z, C), E2) :-
    asg(E1, E2, Z, sub(X, Y)),
    on(Q, const(X, C1), E1), on(Q, const(Y, C2), E1),
    !, number(C1), number(C2), C is C1 - C2.


%-----------------------------------------------------------
% General Rules
%-----------------------------------------------------------

% Assignment
on1(Q, P, E2) :-
    asg(E1, E2, X, Y), subst(P, Y, X, P1), on(Q, P1, E1).

% Conditionals
on1(Q, P, E2) :- sw(E1, E2, E3, equal(X, Y)),
    subst(P, Y, X, P1), on(Q, P1, E1).
on1(Q, P, E2) :- sw(E1, E2, E3, equal(X, Y)),
    subst(P, X, Y, P1), on(Q, P1, E1).
on1(Q, P, E3) :- sw(E1, E2, E3, neg(equal(X, Y))),
    subst(P, Y, X, P1), on(Q, P1, E1).
on1(Q, P, E3) :- sw(E1, E2, E3, neg(equal(X, Y))),
    subst(P, X, Y, P1), on(Q, P1, E1).

on1(Q, P, E2) :- sw(E1, E2, E3, P).
on1(Q, neg(P), E3) :- sw(E1, E2, E3, P).
```

```
on1(Q, P, E3) :- sw(E1, E2, E3, neg(P)).
on1(Q, neg(P), E3) :- sw(E1, E2, E3, neg(P)).

on1(Q, false, E2) :- sw(E1, E2, E3, P), on(Q, neg(P), E1).
on1(Q, false, E3) :- sw(E1, E2, E3, P), on(Q, P, E1).

on1(Q, P, E2) :- sw(E1, E2, E3, PP), on(Q, P, E1).
on1(Q, P, E3) :- sw(E1, E2, E3, PP), on(Q, P, E1).

% Merges
on1(Q, const(X, C), E3) :-
    mg(E1, E2, E3), !,
    on(Q, const(X, C1), E1), on(Q, const(X, C2), E2),
    C1 == C2, C = C1.
on1(Q, induct(X, C1, C0), E3) :-
    mg(E1, E2, E3), !,
    on(Q, induct(X, A1, A0), E1), on(Q, induct(X, B1, B0), E2),
    A1 == B1, A0 == B0, C1 = A1, C0 = A0.
on1(Q, indep(X, Y, L), E3) :-
    mg(E1, E2, E3),
    on(Q, indep(X, Y, L), E1), on(Q, indep(X, Y, L), E2).
on1(Q, neg(P), E3) :-
    mg(E1, E2, E3),
    on(Q, neg(P), E1), on(Q, neg(P), E2).
on1(Q, equal(P1, P2), E3) :-
    mg(E1, E2, E3),
    on(Q, equal(P1, P2), E1), on(Q, equal(P1, P2), E2).

on1(Q, P, E3) :- mg(E1, E2, E3), on(Q, P, E1), on(Q, false, E2).
on1(Q, P, E3) :- mg(E1, E2, E3), on(Q, P, E2), on(Q, false, E1).

% Loops
on1(Q, P, E4) :-
    loop(L, E1, E2, E3, E4),
    on(Q, P, E3).

on1(Q, P, E2) :-
    loop(L, E1, E2, E3, E4),
    on(Q, P, E1),
    prasserta(on(maybe(L), P, E2)),
    on(maybe(L), P, E3).
```

```
% Equality
on1(Q, neg(equal(X, Y)), E) :-
    on(Q, const(X, C1), E), on(Q, const(Y, C2), E), !, C1 =\= C2.
on1(Q, equal(X, Y), E) :-
    on(Q, const(X, C), E), on(Q, const(Y, C), E).
on1(Q, equal(plus(X1, Y1), plus(X2, Y2)), E) :-
    on(Q, equal(X1, X2), E), on(Q, equal(Y1, Y2), E).

%-----------------------------------------------------------
% Equality Propagation
%-----------------------------------------------------------
on1(Q, equal(Z, plus(X1, Y1)), E2) :-
    asg(E1, E2, Z, plus(X, Y)),
    on(Q, equal(X, X1), E1), on(Q, equal(Y, Y1), E1).

on1(Q, equal(Z, mul(X1, Y1)), E2) :-
    asg(E1, E2, Z, mul(X, Y)),
    on(Q, equal(X, X1), E1), on(Q, equal(Y, Y1), E1).

on1(Q, equal(Z, sub(X1, Y1)), E2) :-
    asg(E1, E2, Z, sub(X, Y)),
    on(Q, equal(X, X1), E1), on(Q, equal(Y, Y1), E1).

% Start
on1(Q, false, e1) :- !, fail.

%-----------------------------------------------------------
% Assertion Caching
%-----------------------------------------------------------


on1(Q, P, E) :-
    PP = on(Q, P, E), \+ clause(P, P),
    prasserta(failure(on(Q, P, E))), fail.

on(Q, P, E) :- !, once(on1(Q, P, E)), prasserta(on(Q, P, E)).
on(Q, P, E) :- !, once(on1(Q, P, E)).

prasserta(P) :-
```

```
%    print(P), nl,
     asserta(P).

prretract(P) :-
%    print(P), nl,
     retract(P).

%-----------------------------------------------------------
% Equality
%-----------------------------------------------------------
equal(X, X) :- !.
equal(C1, C2) :- number(C1), number(C2), !, C1 =\= C2, fail.
neg(equal(C1, C2)) :- number(C1), number(C2), !, C1 =\= C2.

equal(plus(C1, C2), C3) :-
      number(C1), number(C2), number(C3), !, C3 is C1 + C2.
equal(C3, plus(C1, C2)) :-
     number(C1), number(C2), number(C3), !, C3 is C1 + C2.

equal(mul(C1, C2), C3) :-
      number(C1), number(C2), number(C3), !, C3 is C1 * C2.
equal(C3, mul(C1, C2)) :-
     number(C1), number(C2), number(C3), !, C3 is C1 * C2.

equal(sub(C1, C2), C3) :-
      number(C1), number(C2), number(C3), !, C3 is C1 - C2.
equal(C3, sub(C1, C2)) :-
     number(C1), number(C2), number(C3), !, C3 is C1 - C2.

equal(val(N), N) :- number(N).
equal(N, val(N)) :- number(N).

%-----------------------------------------------------------
% Substitution
%-----------------------------------------------------------
subst(Y, X, Y, X) :- !.
subst(Z, X, Y, Z) :- number(Z), !.
subst(val(N), X, Y, val(N)) :- !.
subst(Z, X, Y, Z) :- atom(Z), \+ Y = Z.

subst(plus(X1, Y1), X, Y, plus(X2, Y2)) :-
```

```
    atom(X), subst(X1, X, Y, X2), subst(Y1, X, Y, Y2).
subst(mul(X1, Y1), X, Y, mul(X2, Y2)) :-
    atom(X), subst(X1, X, Y, X2), subst(Y1, X, Y, Y2).
subst(sub(X1, Y1), X, Y, sub(X2, Y2)) :-
    atom(X), subst(X1, X, Y, X2), subst(Y1, X, Y, Y2).

subst(neg(X1), X, Y, neg(X2)) :-
    subst(X1, X, Y, X2).

subst(indep(X1, Y1, L), X, Y, indep(X2, Y2, L)) :-
     subst(X1, X, Y, X2), subst(Y1, X, Y, Y2).

subst(induct(X1, A1, A0), X, Y, induct(Y1, A1, A0)) :-
     subst(X1, X, Y, Y1).

subst(const(X1, Y1), X, Y, const(X2, Y1)) :- subst(X1, X, Y, X2).

subst(equal(X1, Y1), X, Y, equal(X2, Y2)) :-
     subst(X1, X, Y, X2), subst(Y1, X, Y, Y2).

subst(invar(X1, L), X, Y, invar(X2, L)) :- subst(X1, X, Y, X2).

%-----------------------------------------------------------
% Greatest Common Denomonator.
%-----------------------------------------------------------
gcd(X, Y, C) :-
        (   Y == 0 ) -> fail ;
        number(X), number(Y), !,
        TMP = Y, TMP1 is X mod Y,
        gcd(TMP, TMP1, C).

gcd(X, Y, C) :- ( Y == 0 ) ->  C = X, true.
```

Appendix B

Parse Tree/CFG Data Structure

```
struct parse_tree_node {

        /*
         *  Class and subclass
         */
        st_class      class;
        st_subclass   subclass;

        /*
         *  Important unique tag numbers
         */
        int           stmtnumber;
        int           nodenumber;
        char          paren;

        /*
         *  Interface structure elements.
         */
        int           listlocation;

        /*
         *  Prolog specific structure elements.
         */
        char          exprvar[TMPNAME_LEN];

        /*
         *  Value for variables and constants
         */
```

```
        union {
                char    cval;
                char    *stval;
                int     ival;
                double  rval;
                stnode  *syval;
        } value;


        /*
         * Syntax tree node pointers
         */
        int     num_children;
        struct parse_tree_node *parent;
        struct parse_tree_node **children;
        struct parse_tree_node *next_stmt;

        /*
         * Control flow graph pointers
         */
        int     num_pred;
        int     num_succ;
        int     mergenum;
        int     loopmergenum;
        edge    **pred;
        edge_t  *pred_type;
        edge    **succ;
        edge_t  *succ_type;

        /*
         *  Symbol value database to keep track of the symbols
         *  and their current known values.
         */
        int     num_sdb;
        sym_db  **sdb;
};
```

Appendix C

Tiny Language Grammar

```
pgm         →  stlist
stlist      →  stmt | stlist ';' stmt
stmt        →  stassert | stcomment | stasgn |
               stconst | stfor | stforall |
               stif | stparfor | stdoany |
               stdecl | streturn | ∅
streturn    →  'return' | 'return' expr
stassert    →  'assert' '(' expr ')'
stcomment   →  COMMENT
stasgn      →  lhs '=' expr
stasgnoper  →  lhs asgnoper expr
asgnoper    →  '+=' | '*=' |
               'max=' | 'min='
lhs         →  ID |
               ID '(' list ')'
stconst     →  'const' constlist
constlist   →  constdecl | constlist ',' constdecl
constdecl   →  newid '=' cexpr
stfor       →  'do' doid '=' expr forto expr optforby ';' stmt |
               'do' doid '=' expr forto expr optforby '{' stmtlist '}' |
               'for' doid '=' expr forto expr optforby 'do' stmtlist 'endfor' |
optforby    →  forby expr | ∅
stforall    →  'forall' doif '=' expr forto expr optforby 'do' stlist 'endfor'
stparfor    →  'parfor' doid '=' expr forto expr optforby 'do' stlist 'endfor'
stdoany     →  'doany' doid '=' expr forto expr optforby 'do' stlist 'endfor'
forto       →  'to' | ',' | ':'
forby       →  'by' | ',' | ':'
stif        →  'if' expr 'then' stlist elsepart 'endif'
elsepart    →  'else' stlist | ∅
stdecl      →  class type decllist
class       →  'auto' | 'common' | 'static' |
               'formal' | 'in' | 'out' |
               'inout' | 'builtin' | 'private' |
               ∅
type        →  'integer' | 'real'
decllist    →  decl | decllist ',' decl
decl        →  newid init |
               newid '(' boundlist ')' init
init        →  '=' expr | ∅
boundlist   →  bound | boundlist ',' bound
bound       →  expr | expr ':' expr
```

```
newid     →  ID
doid      →  ID
cexpr     →  ID |
             INT |
             cexpr '+' cexpr |
             cexpr '-' cexpr |
             '-' cexpr |
             '+' cexpr |
             cexpr '*' cexpr |
             '(' cexpr ')' |
oldid     →  ID
expr      →  oldid | oldid '(' list ')' |
             INT | FLOAT | '(' expr ')' |
             '+' expr | '-' expr |
             expr '+' expr | expr '-' expr |
             expr '*' expr | expr '/' expr |
             expr '^' expr | expr '<' expr |
             expr '<=' expr | expr '==' expr |
             expr '!=' expr | expr '>=' expr |
             expr '>' expr | expr '%' expr |
             expr 'max' expr | expr 'min' expr |
list      →  expr | list ',' expr
```

Vita

C. Chris Meystrik was born August 20, 1968 in St. Louis, Missouri. He graduated from Christian Brothers College High School in 1986 and attended Tarkio College, Tarkio Missouri, where he received a Bachelor of Science degree in Mathematics and Computer Science in 1990. From Tarkio he accepted a research position with Argonne National Laboratory in Chicago Illinois doing analysis on particle accelerator control systems, he was advised by Leonid Sagalovsky. This work prompted him to work toward a Masters of Science degree in Computer Science. He is married to Carianne Meystrik, whom he met at Tarkio College in 1988, and has two children Jacob (4) and Rebecca (10 mos). He was awarded The Master of Science degree in Computer Science in December 1996.