Global Data Flow Analysis of Syntax Tree Intermediate Code

A Thesis by : Richard McCabe BSc.

Supervisors : Micheal O'hEigeartaigh

Heather Ruskin

Mervyn Barman

Declaration :

No portion of this work has been submitted in support of an application for another degree or qualification in the Dublin City University or any other University or Institute of Learning.

Global Data Flow Analysis of Syntax Tree Intermediate Code

Abstract

Author : Richard McCabe BSc.

While software developers make every effort to develop correct, easily maintainable and efficient programs, it is always possible to make improvements. These improvements may be made to the program code or to the data which is manipulated by the program. The process of measuring the efficiency of program code can be performed by the use of the many tools which are available today, such as performance analyzers. The processs of detecting inefficiency in data manipulation is a more complex task but can nevertheless be represented as the result of data flow analysis. This is analogous to program flow analysis.

Performing a data flow analysis on a computer program yields data which can be of use in many areas. It can be considered as bringing relevant information from each part of a program to every other part. It becomes possible to make certain optimizations on the code representation. Decisions as to which types of optimizations are best will be made as a result of consulting the data flow analysis results. Tools can also be constructed which will provide the programmer with information about the state of data which is used by the program.

The issue of performing Global Data Flow Analysis on a particular intermediate code representation of a computer program are discussed in this thesis.

Preface

The work carried out in this thesis was done so under certain constraints. The programming language, compiler and intermediate representation used are the property of a software house. No disclosure of the internals of this software is allowed by the company on legal grounds. For this reason what can be shown and referred to in the thesis has certain limitations. To overcome this problem the author has created a hypothetical intermediate language.

Contents

List of Figures

Figure	Page

1 DATA FLOW ANALYSIS OF SYNTAX TREE INTERMEDIATE CODE

1.1 Introduction

Today there are a great number of computer programming languages available to the software developer. It cannot be said, simply, that any one programming language is better than any other, but only that one language may be more suited to the development of a particular application than another. Because of the great diversity in computer applications there are also a great number of programming languages.

The compiler is a fundamental tool in modern computer science. It acts as a translator between a human oriented *programming language* and computer oriented *machine language.* Compilation is a task which must be carried out for all languages which are not *interpreted.* Even some *interpreted* programming languages may be compiled into a pseudo *machine language.* The general structure of most compilers has a great degree of similarity. [Fis 86] [Hec 77].

The first step in the compilation process is that the *source program* is read in. This is stored in a disk file normally. The *source program* is broken into lexical units by a *lexical analysis* phase of the compiler, as it is read in. The next phase is the *semantic analysis* of the tokens which have been read. This imparts a meaning to the tokens which have been read as well as reporting if any grammatical errors have occurred in the input stream. The output of this phase is to store the program in some *intermediate code* form. This form is of a lower level than the *source* language. Many of the high level constructs such as *for* and *case* statements will have been simplified into test and jump statements . A description of intermediate languages is contained in [Aho 86]. There are several popular forms of intermediate code. The most popular forms are 3-op code, quadruples and syntax tree representation.

The computer oriented *machine code* is generated by the *code generation* phase. This maps the *intermediate code* statements on to *machine code* statements. This is a highly architecture dependent phase of compilation. The code generator may produce code to run in native mode on the architecture, or it may cross generate code for another architecture. This phase handles the allocation of the resources of the *target architecture.* Examples of these resources would be machine registers and memory. The final phase of compilation is to take the produced machine code and assemble it to produce the *object code* .The system supplied assembler might often be used for this phase. This *object code* may then be linked with the system library code to form the *executable image* of the program. From this step the program may now be loaded into memory and executed. The following diagram indicates the compilation process.

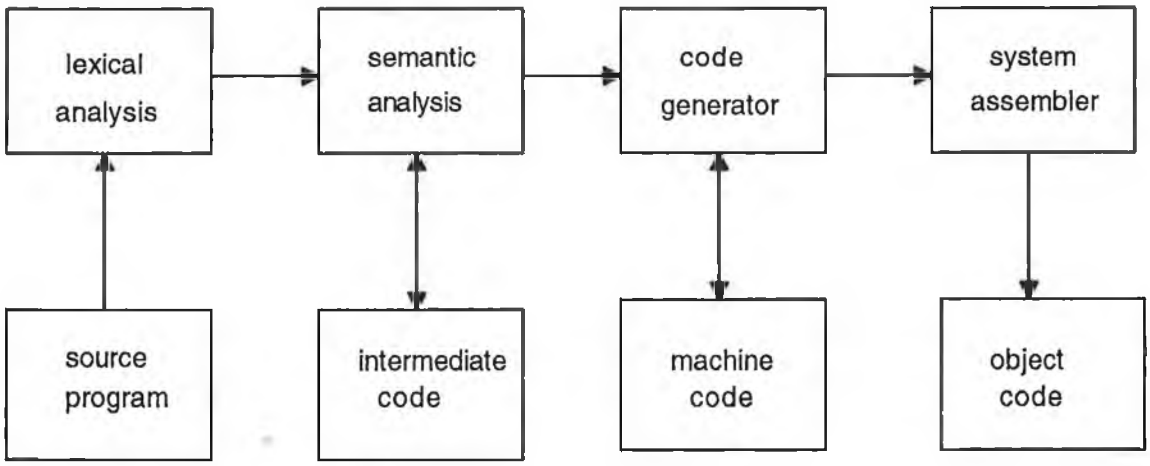

Figure 1.1: The compilation process.

The compilation process as described takes a *source program* and makes an exact translation of every statement which it contains. No attempt is made by the compiler to make any judgement as to the efficiency of the *source* or *intermediate code.* It may become apparent that optimization or improvement may be possible. Optimization means rewriting of a representation of the original source program in a more efficient manner while at the same time fully retaining the functionality of the original program [Aho 77]. A program can be optimized in terms of execution time or in terms of executable code size. Most optimizers will have switches which will allow the user to choose the specific type of optimization desired. Normally a program would be developed with optimizing switched off. This would enable the developer to use a source code debugger on the code under development. When the program has been fully tested and goes into a production run then full optimizing would be switched on.

Sources of optimization can come from two places. In the first place the original source program may have been written in an inefficient manner so that it lends itself to optimization [Aho 86]. Secondly in translation and simplification of the *source program,* as the high level constructs are broken down, certain opportunities for optimization may arise. The translation of the high level source language statements can cause sequences of code to be produced repeatedly in the output *intermediate code,* for example, common subexpressions may occur many times, due to simplification of a sequence of source statements.

Before any optimization can take place some measure of the program's efficiency must be established. This measurement can be performed on isolated program segments or on the entire program itself. The former type of measurement is known as *local* whereas the latter is known as *global.* This type of measurement is known as *data flow analysis.*

Data flow analysis is the examination of how data is affected as the program is executed. This involves studying the possible paths which a program can take in the course of execution and which data items are affected in terms of having their values altered or used [Aho 86] [Fis 88]. The analysis is carried out on the program routines as a whole and the results of these computations are stored in some convenient form. This form should be as compact as possible because of the great amounts of data which may occur. Once this step has been completed the resulting data can be used to determine cases for optimization.

As an example of data flow analysis and optimization consider the following case. A variable A is assigned a constant value 5 at some point in a program. This value of A is used at a second point later on in the program in another assignment statement. If after carrying out data flow analysis, it is observed from the data gathered that the variable is only assigned a value at one point in the program, and that this value is a constant, then there is no reason why the constant value cannot be substituted directly into the expression at the second point. This could save on memory-register transfers required to retrieve the value of A. Furthermore if it is discovered that the value of A is not required anywhere else in the program, then this first assignment may be removed altogether. This would enable a code saving.

1.2 Objectives

This thesis has come about as the result of a study of a commercially available programming language with a view to developing an optimizer for it. It is the intention to optimize the intermediate code in order to provide an architecture independent tool for the may varied and numerous architectures on which this language is available. In order to develop the optimizer a method for data flow analysis would have to be designed. The compiler of this language produces a syntax tree type of intermediate form similar to the hypothetical intermediate code presented in Appendix A.

On carrying out a study of the methods of optimization it is observed that most of the standard algorithms used in the data flow analysis are related to a particular type of intermediate code. A major problem facing this project is that no method of carrying out a data flow analysis for syntax tree intermediate form has been found, even though many of the classical texts and papers on optimization were searched. In order to overcome this problem, the chosen solution is to take the standard data flow analysis algorithms and to modify them to enable them to operate on syntax tree intermediate form. This modification of the algorithms forms the main body of the research work carried out and forms the main objective.

1.3 Thesis overview

The first chapter presents the subject in general terms. It is intended to introduce the reader to the subject matter of the thesis in a high level manner.

The second chapter presents an introduction to the terminology and theory behind global data flow analysis. It starts by presenting the form of intermediate code which is of interest throughout the thesis. We then present a short discussion on optimization. Optimization is the major motivating factor in performing data flow analysis on a program in the first place. Then we consider the techniques and methods employed. This chapter concludes with a discussion on set handling functions.

Chapter three presents the data flow analysis algorithms which are developed. Firstly a presentation of the order of the algorithms is presented followed by the algorithms themselves. The input and output data of these algorithms are described. The algorithms are presented in a pseudo-code form which is self documenting. A short description is also presented.

Chapter four presents an analysis of the algorithms which are developed. The analysis is illustrated by use of an example data flow analysis and a discussion on the efficiency of the algorithms

Chapter five is the concluding chapter. This chapter considers some future directions which this research could take. The development of an optimizer based on this data flow analysis is discussed. Also some tools which could be developed to assist programmers by giving them information on the use of data within programs are discussed.

The appendices contain a description of a hypothetical intermediate code similar to the one used in the development of the code to test out the developed algorithms. The intermediate code is represented as C structures and constant definitions. The principal data structures which are used in development of the data flow analysis algorithms are also included. These data structures are presented more in diagrammatic form to illustrate the principal data of interest. Included here, also, is the set handling code which is used in the course of data flow analysis. Finally some examples are included to illustrate the techniques used and also with a view to tying the thesis together.

2 DATA FLOW ANALYSIS OVERVIEW

Before embarking on a discussion of data flow analysis algorithms it is helpful, at this point, to present the basic material from which a data flow analysis method is constructed. This takes the form of a brief introductory discussion of the terminology which will be met throughout the thesis, illustrated with diagrams wherever it is thought these are helpful in clarifying a point. This terminology is the common language of this subject and can be found in all the standard literature on the subject.

2.1 Intermediate Code Representation

Intermediate code is a representation of a computer program which is created during the compilation process by most compilers. This representation is the result of breaking down and simplifying the original source program by the compiler's syntax analyzer stage. The intermediate representation falls in complexity somewhere between the original source program, understood by the human programmer, and the machine code representation, understood by the computer. Intermediate code representation is rarely referred to by programmers, the prime function of it is to present a more simple construct of the source program for mapping to machine code. It is usual to have a representation where one intermediate code statement maps to exactly one machine instruction, this also makes the allocation of machine registers for the statement an easier task.There are several forms of intermediate code representation, each with its own particular merits and drawbacks. The most common forms are *postfix notation*, *three-address codes* and *syntax tree* representation

Postfix notation is known to mathematics as *polish notation.* In this notation the operators appear after the operands to which they apply, giving a parenthesis-free notation. The main attractions of this notation are the simplicity of translation and the conciseness of representation. This notation lends itself more to the implementation of an *interpreter* because of the stack based characteristics which it possesses. For this same reason also it is not such a useful intermediate language.

Next come the category of representation sometimes known as the *three-address codes.* As the name implies, three addresses and one operator are used in each statement. The addresses refer to data locations where data is read from or results of calculations are stored . Two addresses are used for the storage of the operands and one address is used to store the result. In an intermediate code statement in this language only one operation on the data per statement is carried out. This is convenient when it comes to code generation, as the statement can often map directly on to a machine instruction. This category divides into two sub-categories known as *triples* and *quadruples.* The big difference between these notations and *postfix* notation is the way in which results are referenced. In *postfix* notation the results have to be referenced by means of stack manipulation, whereas *triples* and *quadruples* generate explicit references to results. The difference between *triples* and *quadruples* is that the former reference intermediate values by means of the number of the triple which created them, whereas the latter type uses explicit intermediate value names to reference values. This means that *quadruple* form would lend itself more readily to register allocation.

The *syntax tree* intermediate form gives a hierarchical representation of the source program. Here nodes in the tree represent the elements, such as operators and operands, making up the intermediate language statements. Attributes of the language statement may also be stored in the syntax tree nodes. These attributes would typically contain type information and other such detail, built up as the intermediate code is scanned, and would be of use at code generation time. The advantage of this form of representation is that it is easy to manipulate when it is necessary to restructure the syntax tree in performing optimization [Aho 77]. Also the syntax tree structure can be easily mapped to data structures available in most modern computer high level languages. For example it could be represented as a linked list of records, with node information contained in a record which also contains pointers to the nodes further down the hierarchy. Functions which traverse tree and list structures are generally simple to implement in high level recursive programming languages. These functions can be found in [Wir 76].

As an example consider the following assignment statement :

```
a = b * c + d
```

Figure 2.1: An assignment statement

The syntax tree representation of this statement would be :

```
        =
      /   \
     a     +
          /  \
         *     d
       /   \
      b     c
```

Figure 2.2: Syntax tree representation

The internal nodes of the tree would contain the operators and the leaves would contain the operands. Any attribute information would be stored in the node's extra record fields.

For the purpose of the discussion on the algorithm design a syntax tree representation is included . While this representation is designed for the purpose of illustration, it contains most of the elements which would be found in an actual syntax tree representation. This hypothetical intermediate language is implemented in terms of C structures and constant definitions and is contained in Appendix A.

2.2 Optimization

Optimization is the term given to transforming a program in order to achieve an improvement in the characteristics of the executable program. The main goals of optimizing a program are to produce a reduction in the executable code size and a decrease in the execution time of the program [Fis 88] [Aho 77] [Aho 86]. These conditions are not necessarily connected. Satisfying one does not automatically guarantee satisfying the other. Optimization can only improve the translation of an algorithm, it can never replace a poorly written algorithm with a better one. When dealing with program performance questions it is useful to start by considering whether the programmer could improve any of the algorithms, before considering optimization. A fundamental condition in carrying out any optimization is that the correctness of the program be maintained no matter what modifications are made. It is better to perform only the safe optimizations of a program in order to meet this condition. Many of the valuable optimizations can, in some circumstances, be unsafe. Optimizations guaranteed to always produce the same program results, between the original program and the optimized version, are known as *safe*, those which could cause a difference in results are known as *unsafe*. One of the richest areas of optimization is any improvement which can be made in the performance of tight program loops. As a rule, a lot of programs spend most of their execution time in a very small area of the program code. If these areas can be identified and optimized, then a performance improvement is bound to be made. There is a question of trade off between the amount of effort required to produce a complex optimization and the benefit to the program execution time or size.

One of the more difficult problems in dealing with optimized code is where the program results become incorrect. These situations can be difficult to locate as the optimized code can interfere with the behaviour of debugging tools, and removing the optimization will remove the problem.

Program optimization may take place at various levels such as at the *machine code* level and *intermediate code* level. Optimization of intermediate code has the greater advantage in that it is independent of the target architecture on which the program will be executed. It also means that if the optimization stage of the compiler is written in a portable manner, then it a simple task to re-target it to new architectures.

Optimization of intermediate code means that the syntax tree is rearranged according to certain rules followed by the optimizer. These rules can mean that code or data can be removed from a program altogether, or that they may be moved in order to improve efficiency of the program. The rules used are based upon decisions made by the optimizing program once it has consulted the results of program and code flow analysis. This implies that a first step in writing an optimizer is to perform data flow analysis.

2.3 Data Flow Analysis

Data flow analysis is the collecting of information about the history of the data contained in a program segment [Hec 77] [Muc 81]. This is a static analysis carried out after the *semantic analysis* phase during compile time. The information is gathered into tables within the data flow analyzer following traversals of the program intermediate code. This data collection is carried out prior to any optimization being performed. What is of interest about a data item is where in the program it is defined, where it is modified and where it is used.

2.3.1 Basic Blocks

The first step in performing data flow analysis is the division of the program *intermediate code* into basic blocks. This allows the analysis to be carried out on discrete fragments of the entire program. Blocks are simply sequences of program code which will be executed in their entirety once entered. This means that a block will begin execution at its first statement and end execution at its last statement.

The method of creating basic blocks is straightforward and is defined by the following rules adapted from [Aho 77].

The first statements of basic blocks are first determined.

i) The first program statement is a basic block first statement.

ii) Any statement which is the target of a jump statement is a basic block first statement.

iii) Any statement which follows a conditional jump statement is a basic block first statement.

From these simple rules the intermediate code is partitioned into basic blocks. Basic block information must be ordered, each basic block must contain information telling it which basic blocks are its predecessors and which are its successors. The most suitable structure for basic block information would be a linked list of records. Each record would contain pointers to the start statement and end statement of the block, as well as pointers to the lists of predecessors and successors.

A simple code optimization opportunity can be derived from the basic block information. Consider the case where a basic block, other than the first block, has no predecessor. This would imply that the code can never be entered, and therefore never executed. Such a basic block may be removed as unreachable code.

2.3.2 Flow graphs

A useful representation of the relationships between basic blocks and their predecessors and successors is presented by a directed graph called a *flow graph*. Each node of the graph represents a basic block, and the graph edges are the paths to successor blocks [Aho 77]. One block has the property of being the initial block. This initial node would be the basic block containing the entrance into a function.

As an example of a flow graph, consider the following code fragment :

```
if ( a == 1 )
{
        b = 1;
}
else
{
        c = 1;
}

d = a + b;
```

Figure 2.3: Sample code

The flow graph for this code fragment would be :

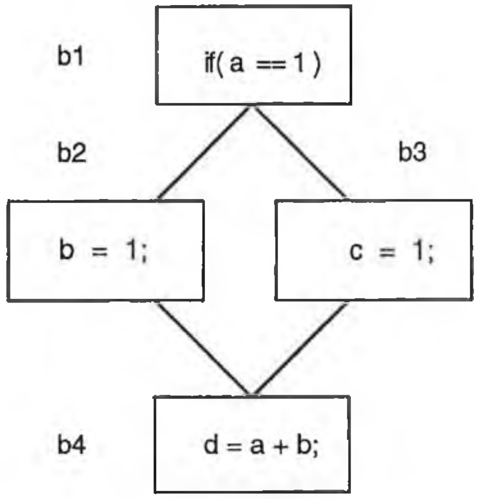

Figure 2.4: A Flow Graph

Here **b1**, **b2**, **b3**, and **b4** are the nodes of the graph representing basic blocks. The control flow is represented by the graph edges joining the nodes. As can be seen, control can pass from block **b1** to either block **b2** or **b3** depending on the value of **a**.

Block **b2** has a predecessor block **b1** and a successor block **b4**. Block **b4** has two predecessors, blocks **b2** and **b3**, and no successors.

2.3.3 Iterative analysis technique

The problem of data flow analysis can be considered as finding the data flow of a program from the control flow structure. The information which is built up by data flow analysis involves the definition and use information for the program variables. This information is constructed to show which definitions of a program variable affect the uses of the variable. The control flow structure is discernible from the ordered basic block information, given that , by definition, these ordered basic blocks map out the program flow.

The data flow questions fall into two classes:

Class 1

Given a particular statement in a program, what happens to the program variables before control reaches that point in the program, (i.e. , what variable definitions can affect computations at that point)?

Class 2.

Given a particular statement in a program, what happens to the program variables after control leaves that point, (i.e. , what variable uses can be affected by definitions at that point)?

Class 1 questions are known as *forward-flow* problems while class 2 questions are known as *backward-flow* problems.Solutions to both of these problems are possible.

An analysis which spans more than one block is called a *global data flow analysis.* With this type of analysis it is impossible to statically determine in what sequence the basic blocks will be executed. The analysis must consider that all paths are possible. In this way any optimizations based on the analysis are valid no matter what the execution path.

The method of solving flow analysis problems is best described by considering the solution of a simple flow analysis problem. Take, for example, the live variable analysis problem. By definition a live variable is a variable whose current value will be used before it is assigned a new value. In the context of global data flow analysis a variable is considered live if along any possible path it is used before being reassigned a new value.

Let **b** be the index of a basic block. Define In(b) as the set of variables live on entry into basic block b. Define Out(b) as the set of variables live on exit from basic block **b**. Let S(b) be the set of basic blocks which are successors of **b**. It is then the case that the OUT set is the set union of all the IN sets of successor blocks of **b** :

$$Out(b) = \bigcup_{i \in S(b)} In(i)$$

That is to say that a variable is live on exit from a block when it is live on entry to some successor block.

Let Use(b) be a set of variables within basic block **b** which are used before they are re-defined. The definition of a variable is the action of assigning a value to it. The set Use(b) is a constant set whose value depends entirely upon the program statements within basic block b. If $v \in Use(b)$ then $v \in In(b)$.

Let Def(b) be the set of variables that are defined in basic block **b**. Def(b) is also a constant set whose value depends on the statements contained in **b**. The value of Def(b) is obtained by scanning the statements in block **b** and finding where a variable is assigned or read. If a variable is live on exit from block **b**, it is also live on entry to block **b** unless it is defined in the block. That is , $v \in Def(b)$.

This also means that the set In(b) in given by the difference between sets Out(b) - Def(b) which is stated as :

$$In(b) \supseteq Out(b) - Def(b)$$

Careful consideration of live variables shows that the only way a variable can be live on entrance to a block is either for it to be in the set of variables used by the block, Use(b), or for it to be live on exit from the block and not redefined within the block. This is stated by the equation :

$$In(b) = Use(b) \cup (Out(b) - Def(b))$$

This equation, relating IN and OUT sets, exists for each basic block.

It is always possible to solve data flow equations for a flow graph which has a unique starting node, with no predecessors, and one or more terminating nodes, with no successors [Fis 88]. The technique used to solve the live variable analysis problem is to start with the values generated in the basic blocks , the USE sets. These values are then propagated to the set of predecessor blocks. The values killed in each block, the DEF sets, are excluded. This process is iterated until the IN and OUT sets converge. The sets have converged when there are no more changes in the IN and OUT sets for an iteration.

This data flow problem considered, live variable analysis, assumes the property to be true If it could be seen to hold along any path. This is known as an *any-path* type of problem. A solution to an *any- path* problem dos not require that a property must hold, only that it may hold.

Data flow problems can also exist in an *all-path* form. In this case the desired property is stated to hold along all possible paths. In the solution of an *all-path* problem the desired property can be guaranteed to hold along all possible paths.

2.3.4 Data flow problem types

Data flow problems follow a very clean grouping. For each basic block there is an IN and an OUT set. These sets are the definitions live on entry into a basic block and live on leaving a basic block respectively.

In solving *forward-flow* data problems, OUT sets are computed from IN sets within a particular basic block and IN sets are computed from OUT sets across a basic block. For *backward-flow* problems, IN sets are computed from OUT sets within a basic block, and OUT sets are computed from IN sets across a basic block.

Within a basic block the IN and OUT sets are related by equations of the form :

$$In(b) = Use(b) \cup (Out(b) - Kill(b))$$

or

$$Out(b) = Use(b) \cup (In(b) - Kill(b))$$

depending on whether a *backward-flow* problem or a *forward-flow* problem is under consideration.

On an *any-path* problem a union of predecessor (or successor) values is computed; on an *all-paths* problem the intersection of predecessor (or successor) values is computed.

Boundary conditions are that the IN set of the initial basic block in a *forward-flow* problem and the value of the OUT sets of the final basic blocks for a *backward-flow* problem are specified. These boundary condition sets are either empty or contain the set of all possible values depending on the interpretation of the problem.

These equations can be summarized as follows :

	Forward-flow	Backward-flow
Any-path	$Out(b) = Gen(b) \cup (In(b) - Kill(b))$ $In(b) = \bigcup_{i \in P(b)} Out(i)$	$In(b) = Gen(b) \cup (Out(b) - Kill(b))$ $Out(b) = \bigcup_{i \in S(b)} In(i)$
All-path	$Out(b) = Gen(b) \cup (In(b) - Kill(b))$ $In(b) = \bigcap_{i \in P(b)} Out(i)$	$In(b) = Gen(b) \cup (Out(b) - Kill(b))$ $Out(b) = \bigcap_{i \in S(b)} In(i)$

Figure 2.5: Data Flow Equations

2.3.5 Set Operations

In performing data flow analysis bit vectors provide a very convenient and compact form of data representation. A bit vector is a set of bits with the following interpretation : a bit position represents the number of an object in which the observer is interested, and its value, 0 or 1, represents the state of this object. This technique allows large amounts of information to be stored in a compact manner.

As an example, consider the use of a bit vector to represent the GEN variable information in the following basic blocks :

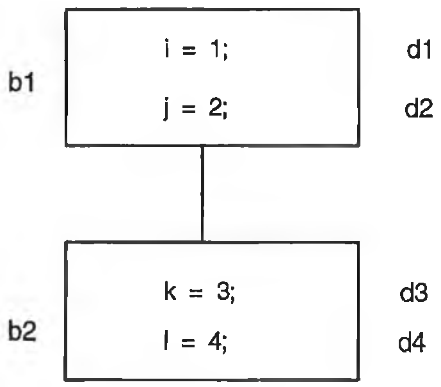

Figure 2.6: Code sample

The bit vectors in this case would be :

Block[b]	Gen[b]	bit vector
b1	{d1, d2}	1100
b2	{d3, d4}	0011

Figure 2.7: Bit vectors

The bit vectors are considered as sets, with a one representing the presence of a quantity, and a zero representing the absence of a quantity. In performing data flow analysis the operations required to be carried out on bit vectors are as follows: the creation and removal of sets, the insertion and removal of a set element, union, intersection of sets, comparison of sets and check for membership of a particular set.

Not all high level programming languages contain set handling functions as part of their standard library, or if present then they may not be capable of handling the large sets required for the purpose of data flow analysis. It is desirable to be able to allocate sets dynamically as the flow analysis proceeds. It is possible to write set handling functions to suit the application. (See Appendix B). The sets are implemented by obtaining a block of memory from the operating system and then performing the required operations by means of set functions. These functions are necessary as the algorithms require set operations to be carried out on data.

3 DATA FLOW ANALYSIS ALGORITHMS

This chapter presents a method and algorithms which are developed to carry out a data flow analysis. The material presented here is the first stage in the development of an optimizer for a commercially available compiler for a high level programming language. Before presenting the algorithms it is of interest to examine existing methods of carrying out data flow analysis, and point out why these methods could not be applied to the problem at hand. This discussion will give some insight as to why these methods were unsuitable and had to be modified.

The subjects of *data flow analysis* and *code optimization* have been under development since the early days of computer science. In the past the necessity for optimization, and hence data flow analysis, was much greater because of the lower power of the processors available, by comparison with today's standards. The main objective of optimization then was to reduce both storage size and execution time of a program. Today the trend is more toward improving execution speed, as the cost and availability of memory is not as restrictive as it once was. Nevertheless the use of data flow analysis in compilers has increased dramatically, both in potential and actual usage, since its inception. Much pioneering work on the subject was carried out in the early to mid seventies.

A wide range of literature is available today on the subject of *data flow analysis* which provide solutions to the problems encountered. From this literature come a large number of standard methods and algorithms which have been implemented and tested over time. Proof of the correctness of these methods is also available. The methods and algorithms presented in the standard literature solve particular *data flow* problems for a particular type of *intermediate code.*Two forms of *intermediate code* are most popular and are widely used in the algorithms contained in the literature which was studied by the author. These forms are commonly known as *three-address code* and *quadruples* respectively. They tend to be a very simple representation of the original *source code* statement. The complexity of the original statement is broken down into multiple simple steps. There is an abundance of material available which deals with the subject of *data flow analysis* for *three-address code* and *quadruples.*

These forms of *intermediate code* have been in use for many years and have several advantages. These representations may have been popular when more strict memory constraints were in force on earlier computers. As the name suggests, the usual format for this representation is two addresses for the operands and one for the result. There is at most one operator per statement. This representation is convenient in that the amount of space required per statement is small. These simple three operand statements map on to most assembler instructions in a convenient manner, making code generation a simple process. Temporary values are created as needed in the code, and these conveniently map to temporary register loads and stores. In fact this type of code simplifies the task of register allocation for a particular machine.

As pointed out earlier the problem being dealt with here was the fact that the requirement was to provide a *data flow analysis* of a certain type of *intermediate code.* The code of interest to this project was of the *syntax tree* type, which is discussed in section 2.1 of this thesis. None of the standard literature contains detail as to how this could be accomplished. As the standard algorithms stand, they are not adequate to apply to this problem.

A point about syntax tree representation is that it is rather expensive in terms of memory usage. This point does not have so much relevance to modern computers with 32 bit addressing and virtual memory. The fact that syntax tree representation is conveniently restructured makes it ideal as a language which has to be optimized.

There are a number of possible solutions which could be applied to solve the problem. Among the most feasible of these is the possibility of carrying out a translation of the intermediate code as it stands, in order to create the form which will be accepted by the standard *flow analysis* algorithms. This is ruled out on the grounds that the code generators will have to be modified or else another translation will have to be made in order to get the code back to its original form after *flow analysis* and *optimization.* The alternative to this is to modify the algorithms so that they will accept the *syntax tree* representation. This approach appears to give a better return on the time invested. It will also greatly reduce the compile time if no translations of the *intermediate code* have to be carried out. For these reasons this is the chosen method.

It has taken a considerable amount of effort to arrive at the solution to this problem. In the first place it is necessary to be familiar with the compiler and intermediate language which is the subject of this work. A copy of the compiler and intermediate code description has been made available for the project. On becoming familiar with the workings of the compiler and its internals, a wide range of literature on the subject of data flow analysis has been acquired and studied. There are may classic papers on the subject as well as material contained in text books on compilers. Some of the authors whose papers have been studied are [All 71] [All 76] [Ken 72] [Mor 79] [Ull 75]. Also the many years of experience by the author in the area of systems software has proven to be advantageous in the design and implementation of the data flow analyzer.

The first decision is which algorithms are to be implemented, mindful of the fact that the objective of the data flow analysis is to provide information for an optimizer stage. Having decided upon a set of algorithms, it is then necessary to take each of the algorithms and try to prove empirically that the problem can be solved for the type of intermediate language on hand. The correctness of the standard algorithms is assumed to hold. The assumption is also made that the algorithms will still produce correct results if modified for syntax tree intermediate code. Initially examples have been traced through the algorithms by hand and result data checked. Once satisfied with these results, the algorithms can be implemented, one by one, and their output checked against known results.

The material contained in this chapter will be of benefit to anyone who is considering implementing *data flow analysis,* or *code optimization* in two ways. Firstly, as an educational tool, the material presents an implementation of the subject of *data flow analysis,* which would serve to broaden the readers knowledge in the area. Secondly, it would solve the problem being faced by anyone implementing *data flow analysis* if dealing with a *syntax tree* representation of *intermediate code.* The modified algorithms presented here will carry out the task if implemented. The terms *data flow analysis* and *code optimization* are very closely related, since it is not possible to make a thorough job of the latter without implementing the former. Although no optimization is carried out in this work, the more complex task of *data flow analysis* is tackled. The material should also be of interest to students of computer science, since most text books consider the basic principles of *data flow analysis* without going deeply into implementation details.

The algorithms are presented in a pseudo-code form, the language constructs of which should be familiar to most readers. Set handling functions are implemented in the C programming language. These functions could be called from the data flow analyzer program. (See Appendix C). These algorithms are implemented in one such programming language and tested on a particular *intermediate code* form. When these algorithms are implemented then the data collected constitutes a data flow analysis, the results of which can be fed to an optimizing stage of the compiler or can serve as input data to some other form of tool.

The order in which the data flow analysis is carried out [Aho 77] is as follows :

The first step is to break the program intermediate form into basic blocks. The blocks are then ordered in terms of predecessors and successors for each basic block. These lists of predecessors and successors will be used by later algorithms. Rules for deciding the basic block subdivision have been discussed earlier.

The next step is to collect all static information about the program. This information is obtained from both the symbol table information and by a traversal of the intermediate code. Here, what is of interest are the DEF and USE information. The DEF information is a linked list of records which hold information about the points in the program where a local or static variable is assigned or read. Static variables retain their values over repeated calls to the function. Only variables local to a routine are considered, since treating arrays or pointers is a more complex task. This means that each routine has data flow analysis carried out on it in isolation from the other routines making up a program, so that no interprocedural analysis is performed. Global variables would be handled by considering interprocedural data flow analysis.

Likewise the USE information is a linked list containing information about where each local or static variable is used. Since this could occur anywhere in the syntax tree, routines must be written which will traverse an expression tree to any depth and mark all usage points of the variables of interest [Wir 76].

To prepare for the data flow analysis, certain information about the state of variables within each basic block will be necessary. This information is known as the GEN and KILL information. The GEN set information is information about local and static variable definitions which reach the end of the basic block. These would be the definitions or re-definitions of a variable. The result is a bit vector representing the variables live at the end of the block. The KILL set is a set of local or static variable definitions outside of the basic block which also have definitions inside of the block. That is to say that the KILL set marks all those definitions which are destroyed by this latest re-definition.The GEN and KILL information is represented per basic block as a bit vector.

The next step is to build what are known as the IN and OUT sets for each basic block. These are again bit vectors which represent the following variable definition properties. The IN set is the set of all variable definitions which are live before the first statement of the basic block Is executed. The OUT set is the bit vector representing the set of variables which are live after executing the last statement in the basic block. The algorithm involved in the building of these sets is different from the previous algorithms in that the previous algorithms carry out their task by observation of certain rules, whereas the IN-OUT sets are computed by the iterative technique.

The final algorithm is that which builds use-def chain information. The UD chain information is represented by a bit vector for each use of a variable. It is computed as follows. If a usage of a variable is preceded in its basic block by a definition of the same variable, then the UD chain consists only of the last definition of the variable. Otherwise the UD set consists of all definitions of this variable in the IN set.

3.1 BASIC BLOCK DIVISION

ALGORITHM : Divide a routine into its basic blocks.

DATA IN : The routine intermediate code linked list.

DATA OUT : A linked list of basic blocks.

METHOD :

The following rules are used to identify basic blocks in the intermediate code :

i) The first program statement forms the start of the first basic block.

ii) Any statement which is the target of any go to statement is the start of a basic block.

iii) Any statement following a go to statement forms the start of a basic block.

The algorithm first gets the start node of the intermediate code. This by definition i) is the start of the initial basic block. A **basic block record** is created and the basic block pointer, **block_start,** is set . The local basic block counter is initialized to 1, and this value is set in **block_number.** Now the intermediate code is traversed to identify the subsequent basic blocks, searching for occurrences of **BRANCH, COMPARE** and **LABELDEF** nodes. From the rules above these nodes signify the start and end of basic blocks. Each time such a node is encountered the end of the previous basic block is marked, the local block counter is incremented and a new basic block record is created. For label definition nodes, **LABDEF,** a temporary record is created to identify the block which contains the label. This information will be necessary when finding basic block predecessors and successors. This algorithm terminates after reading the last program statement of the last basic block. The last basic block record is completed once the main traversal loop completes. (See example in Appendix D).

```
ALGORITHM START
    get start_pointer to beginning of intermediate code
    block_count = 1
    create the initial basic block record
    initialize record with block_count and block_start pointer
    while NOT last node
    do
        case node_type of
            BRANCH ,
            COMPARE ,
            LABELDEF :
                    set record  block_end pointer
                    link basic block record into basic block list
                    increment block_count
                    create a  new basic block record
                    initialize record with block_count and block_start pointer
                    if ( node_type == LABELDEF )
                        create a label definition record
                        initialize record with label number
                        and owner basic block
                        link this record into label list
                    endif

            default :
        endcase
        next node
    loop
```

set last record block_end pointer

link last basic block into basic block list

for each basic block

do

find all predecessor and successor blocks to this block

store the information in this basic block record

loop

ALGORITHM END

3.2 DEF LIST ALGORITHM

ALGORITHM : Mark all definition points of local variables in the routine.

DATA IN : A linked list of basic block information containing the syntax tree intermediate code.

DATA OUT : A linked list of definition points for the routine.

METHOD :

The **DEF list** is created as auxiliary data which is used to build up the data flow analysis. The previously defined basic block structures are used in this algorithm. Definition of data can occur in the **EVAL** node and also as part of the **CALL** and **CALLPTR** nodes. In this case the argument lists may be assigned values. The intermediate code contained in each basic block is traversed as the definition information needs to know the basic block number. The linked list of basic blocks presents a convenient structure for traversal. The linked list is traversed from the beginning of the list. The intermediate code nodes contained in each basic block are examined to find **EVAL** nodes as well as the **CALL** and **CALLPTR** nodes. These nodes are the points where data are defined. For the **CALL** and **CALLPTR** nodes the **ARGREF** nodes are marked as definition points. Each time one of the above mentioned nodes are encountered in the intermediate code a **DEF record** is created. Information stored in the **DEF record** is the definition number, and the node pointer. The definition number is given by a local definition counter which is incremented each time a new definition is encountered. The counter is set to one for the first definition. The algorithm terminates after the last basic block structure is traversed.

```
ALGORITHM START

    for each basic block

    do

        while NOT last node of  basic block

        do

            case node_type of

            EVAL   :
                        if left_node type  is a  local or static variable

                            increment definition counter

                            create a DEF record

                            set  DEF count to definition counter

                            set  DEF node to this node

                            link this DEF record into definition point list

                        endif

            CALL,

            CALLPTR  :

                        foreach argument of arg list

                        do

                            if argument is an ARGREF

                                Increment definition counter

                                create a DEF record

                                set  DEF count to definition counter

                                set  DEF node to this node

                                link this DEF record into definition point list

                            endif

                        loop

            default  :

            endcase

            next node
```

```
            loop
        loop
    ALGORITHM END
```

3.3 USE LIST ALGORITHM

ALGORITHM :	Mark all use points of local variables in the routine.
DATA IN :	A linked list of basic block information containing the syntax tree intermediate code.
DATA OUT :	A linked list of use points for the routine.

METHOD :

The **USE list** is also auxiliary data list used by data flow analysis, used by the subsequent algorithms. The linked list of basic block information is traversed from start to finish. The syntax tree intermediate code contained in each basic block is traversed. As a use of data can be embedded anywhere in an expression tree, the entire expression tree structure must be traversed. The syntax tree parent nodes can be one of the following types, a **BRANCH** node, a **COMPARE** node, an **EVAL** node or a **RETURN** node. The intermediate code is scanned for occurrences of these node types, as the children of these node types could contain variable uses. Once one of these nodes is encountered the underlying tree must be traversed. This traversal is carried out by the NODE_TRAVERSE function. This is a binary tree traversal function [Wir 76]. On meeting a data reference in the tree traversal, a **USE record** is created. The record is initialized with the block number and a pointer to the tree node. The algorithm terminates after the last basic block structure is traversed.

```
ALGORITHM START

    for each basic block
    do
        while NOT last node of  basic block
        do
            case node_type of

                BRANCH,
                COMPARE,
                EVAL,
                RETURN        : call NODE_TRAVERSE(node)
                default  :
            endcase
            next node
        loop
    loop

    START routine NODE_TRAVERSE(node)

        do a recursive NODE_TRAVERSE( ) of the tree whose root is
        the given node searching for all data references.

        if a data reference
            create a USE record
            set USE node to node
            set USE basic block number
            link USE record into linked list of USE records
        endif
    END routine NODE_TRAVERSE
ALGORITHM END
```

3.4 GEN SET ALGORITHM

ALGORITHM : Build a GEN set for each basic block

DATA IN : A chain of basic blocks, and the definition information for all basic blocks.

DATA OUT : A GEN set for each basic block.

METHOD :

This is a set of data which is created to identify which data definitions are made within a basic block. An element of the **GEN** set represents a definition of a variable made within a basic block which reaches the end of the block. If there are several definitions of the same variable made in a basic block, then the **GEN** definition is the last such definition. The set handling functions are used to create and enter data into the sets. The sets are stored by storing their pointers in the basic block data structure. The intermediate code contained in the basic block structures is traversed from start to finish. For every variable definition the block is searched in a forward direction to find the last definition of the variable. Last definitions are then stored in a temporary data structure. This process is repeated until the entire block has been traversed. The **GEN** set is then built up from the temporary **GEN** list by traversing it and entering each definition into the set. Finally the **GEN** list can be deallocated in order to recycle memory.

```
ALGORITHM START

    for each basic block
    do
        for each variable definition in this basic block
        do
            find last definition of each variable
        loop
        for each GEN  record in list
        do
            if  this definition NOT found
                create a GEN record and enter this variable
                link this record into GEN list
            endif
        loop
    loop

    for each basic block
    do
        if  GEN list exists for this block
             allocate a set
            for every item in GEN list
            do
                 insert each member of  GEN list into GEN set
                deallocate GEN record
            loop
        endif
    loop
ALGORITHM END
```

3.5 KILL SET ALGORITHM

ALGORITHM : Build a KILL set for each basic block.

DATA IN : A chain of basic blocks, and the definition information for all basic blocks.

DATA OUT : A KILL set for each basic block.

METHOD :

The **KILL** set represents those variables defined outside the basic block which also have definitions inside the block. Those definitions outside the block are destroyed by the redefinitions inside the block. These sets of information will be used later in the flow analysis. Again the **KILL** sets are economically represented by bit vectors. The sets are manipulated by making calls to the supplied set handling functions. Temporary records are created to gather the information prior to building the sets. This temporary storage is released on termination. The global linked list of definitions is scanned to find all those definitions of a variable which are contained in the block as well as outside it. The definitions outside the block are considered to have been destroyed. For each of these instances a **KILL** record is created. This record is initialized with the block number, variable name and the value of the definition number. This record is then linked into a linked list. On completion the basic block list is once again scanned and for each block containing a **KILL** list a set is allocated and the killed definitions are transferred to the set by looping over the list of **KILL** records. The temporary storage is relinquished at this point.

```
ALGORITHM START

    for each basic block
    do
        for each variable definition outside this basic  block
        do
            for each GEN  record in this basic block defining the same variable
            do
                create a KILL record
                enter this variable
                link into temp KILL list
            loop
        loop
    loop

    for each basic block
    do
        if  a temp KILL list exists for this block
             allocate a set
            for every item in KILL list
            do
                insert each member of KILL list into KILL set
                deallocate KILL record
            loop
        endif
    loop

ALGORITHM END
```

3.6 IN-OUT SET ALGORITHM

ALGORITHM : Build the IN and OUT sets for each basic block.

DATA IN : A chain of basic blocks, for which the GEN and KILL sets have been computed.

DATA OUT : The IN and OUT sets for each basic block.

METHOD :

This algorithm computes the **IN** and **OUT** sets. These sets represent data which reach a point just before the first statement of a basic block and those definitions which reach a point just after the last statement of a basic block respectively. The **iterative** method, described earlier, is used to compute these sets. The equations relating **IN** and **OUT** sets are :

$$Out(b) = In(b) - Kill(b) \cup Gen(b)$$

and

$$In(b) = \bigcup_{p \in P(b)} Out(p)$$

The initial condition is that the **IN** set is NULL. The algorithm starts by allocating **IN** and **OUT** sets for every basic block, and by assigning each **GEN** set to each **OUT** set. A boolean flag CHANGE is used to mark any changes in the **IN** sets. For each basic block the set of predecessors is traversed and the set union is taken of all the predecessors. The list of predecessors was already computed by the Basic Block Division Algorithm. If the **IN** set has changed the CHANGE flag is set. This propagation of definitions is carried as far as it can go without being killed.

```
ALGORITHM START

    for each basic block
    do
        allocate  the IN set
        allocate the OUT set
        initialize the IN set to zero
        copy the GEN set to the OUT set for this block
    loop

    allocate a NEWIN set
    CHANGE = TRUE

    while CHANGE == TRUE
    do
        CHANGE = FALSE
        for each basic block b
        do
            for each basic block a which is a predecessor of block b
            do
                NEWIN = set union of NEWIN and OUT[a]
            loop
            if NEWIN != IN[b]
                CHANGE = TRUE
            endif
            copy the NEWIN set to the IN[b] set
            OUT[b] =  set union of set complement of IN[b] and KILL[b] with GEN[b]
        loop
    loop
ALGORITHM END
```

3.7 USE-DEF SET ALGORITHM

ALGORITHM :	Build the USE-DEF chain for each basic block.
DATA IN :	Basic block list, DEF list, USE list and IN set
DATA OUT :	The UD set for each USE point.

METHOD :

The **UD** set is a set of information which tells where particular uses of a variable within a basic block are defined. The USE-DEF set is computed from the **DEF** and **USE** lists. By definition the ud-chain is a set of information with the following meaning. If a variable is preceded by a definition within a basic block, then only the last definition forms the **UD** chain. If a use of the variable has no definitions within the basic block then only those definitions contained in the **IN** set form the **UD** chain. From these definitions the set is computed as follows. The **USE** points of each basic block are examined to find definitions within the block.The *intermediate code* is traversed from the end of a block to the beginning looking for definitions as only the last definition is of interest. When a definition occurs a **UD** set is created and the definition is entered into it. If no definitions are found in the block the definitions of the variable in the in the **IN** set form the **UD** set.

ALGORITHM START

```
for each basic block
do
    for each use point  in this basic block
    do
        get pointer to use node
        while NOT start of basic block
        do
            for each DEF record in list
            do
                if DEF point == node
                    make UD set
                    insert this definition in the UD set
                    FOUND = TRUE
                    leave
                endif
            loop
            if FOUND == TRUE
                leave
            endif
            get previous node
        loop
        if FOUND == FALSE
            make UD set
            for each DEF  of this USE variable
            do
                if DEF is a member of IN set
                    insert this definition in the UD set
                endif
```

```
                loop
            endif
        loop
    loop
ALGORITHM END
```

4 ANALYSIS

4.1 Test Of Flow Analysis

All algorithms and methods of *data flow analysis* discussed in this thesis are fully implemented in a high level programming language and have been tested by the author. The set handling code, presented in Appendix C, is also implemented, tested and incorporated into the analyzer. This C code proves to be functionally correct. To verify that the entire implementation is correct a large number of function tests have been carried out on each algorithm. To carry out tests on the correctness of the analyzer, the test programs are written in the source language producing the intermediate code. Some of these tests include test material from [Aho 77] which provide intermediate test results. All the documented intermediate results are compared with actual test results. A hard copy of the full implementation of the *data flow analyzer* and test results are retained by the author.

4.2 A Detailed Example

The best method of describing the complexities of the computations in data flow analysis is by means of taking a simple example. The output of each step of the data flow analysis is explained with reference to the particular algorithms which produce the output. In this example the algorithms which compute the IN and OUT sets for an example piece of source code are examined. Although the piece of code in itself is meaningless, it will yield several basic blocks and definitions as well as uses of variables. The output from a test run by the analyzer is included in Appendix D. The example code is as follows :

```
main

define I,  J as integer variables

'L1'
        I = 2
        J = I + 1
'L2'
        I = 1
        if J = 999
                go to L1
        endif

'L3'
        J = J + 1
        if J = 999
                go to L4
        endif
        J = J - 4
'L4'
        go to L2

end
```

Figure 4.1: Example code.

The flow graph for this code is as follows :

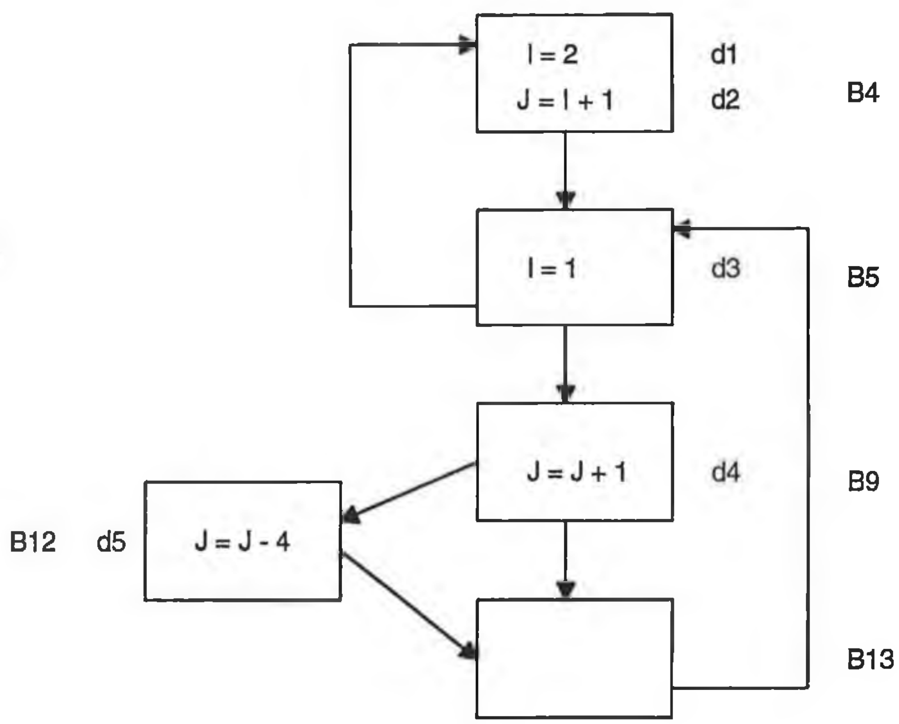

Figure 4.2: Flow Graph

Many of the basic blocks are omitted fom the flow graph as these form the system prologue and epilogue and are of no interest to the example. The basic blocks are B4, B5 B9, B12 and B13, the program variables are I and J and the definitions of these variables are marked d1, d2, d3, d4, and d5..

First consider the computation of the GEN and KILL sets. These computations are carried out by Algorithms 3.4 and 3.5 respectively. Prior to running these algorithms the basic blocks are formed and scanned to create lists of all DEF and USE points. From the GEN set algorithm, (Algorithm 3.4) each of the basic blocks are scanned to find the last definition of a variable within them. Once found these definitions are temporarily saved. On scanning all basic blocks the temporary data is transferred to a set. For the KILL set generation, (Algorithm 3.5), the definitions inside the basic block kill all definitions outside it. All such definitions are temporarily stored and finally transferred to sets on termination of the algorithm. For the example this yields the following GEN and KILL sets :

In B4 the definitions d1 and d2 of I and J respectively kill all definitions outside of B4 giving KILL[B4] = { d3, d4, d5 } , and also as d1 and d2 are the last definitions of I and J in the block B4, GEN[B4] = { d1, d2 }. For block B5 the definition d3 of I kills all definitions which exist outside it, so that KILL[B5] = { d1 } and also GEN[B5] = { d3 }. This process is repeated for the remaining blocks so that in summary :

Block	GEN Set	Bit Vector	KILL Set	Bit Vector
B4	{ d1, d2 }	11000	{ d3, d4, d5 }	00111
B5	{ d3 }	00100	{ d1 }	10000
B9	{ d4 }	00010	{ d2, d5 }	01001
B12	{ d5 }	00001	{ d2, d4 }	01010
B13	NULL	00000	NULL	00000

Figure 4.3: GEN and KILL sets.

Having computed the GEN and KILL sets the IN and OUT sets can be generated. Considering Algorithm 3.6, the IN and OUT sets are allocated and initialized for all basic blocks. The IN set is initialized to the null set and the OUT set is initialized to the contents of the GEN set, or in other words a set copy is made. A temporary work set, NEWIN, is also allocated. Now consider the iterations over the basic blocks. For block B4 the algorithm gives NEWIN = OUT[B5], as this is the only predecessor in the chain. This value of NEWIN at this point is GEN[B5] = 00100, from Figure 4.3. The test of NEWIN = IN[B4] fails and the boolean variable CHANGE is set to TRUE. The computation of OUT[B4] is :

From the equation

$$Out[B] = In[B] - Kill[B] \cup Gen[B]$$

we have

OUT[B4]	=	00100 - 00111 + 11000
	=	11000

Next consider the case for block B5. Here the computation yields

NEWIN	=	OUT[B4] + OUT[B13]
	=	11000 + 00000
	=	11000

This is the value for IN[B5]

OUT[B5]	=	IN[B5] - KILL[B5] + GEN[B5]
OUT[B5]	=	11000 - 10000 + 00100
	=	01100

For the next block, B9, the computation is

IN[B9]	=	OUT[B 5]
	=	01100

and also

OUT[B9]	=	IN[B9] - KILL[B9] + GEN[B9]
	=	01100 - 01001 + 00010
	=	00110

Next for block, B12 we have

IN[B12] = OUT[B9]

= 00110

and for OUT B12

OUT[B12] = IN[B12] - KILL[B12] + GEN[B12]

= 00110 - 01010 + 00001

= 00101

And finally for block B13

IN[B13] = OUT[B9] + OUT[B12]

= 00110 + 00101

= 00111

Then

OUT[B13] = IN[B13] - KILL[B13] + GEN[B13]

= 00111 - 00000 + 00000

= 00111

To summarize tables are given to demonstrate the values Initially and over the iterations of the algorithm.

Initially the values of the IN and OUT sets are :

Block	IN[B]	OUT[B]
B4	00000	11000
B5	00000	00100
B9	00000	00010
B12	00000	00001
B13	00000	

After Iteration 1 :

Block	IN[B]	OUT[B]
B4	00100	11000
B5	11000	01100
B9	01100	00110
B12	00110	00101
B13	00111	00111

After Iteration 2 :

Block	IN[B]	OUT[B]
B4	01100	11000
B5	11111	01111
B9	01111	00110
B12	00110	00101
B13	00111	00111

After Iteration 3 :

Block	IN[B]	OUT[B]
B4	01111	11000
B5	11111	01111
B9	01111	00110
B12	00110	00101
B13	00111	00111

Any iterations after the third will produce the same results for the IN and OUT sets, so that the algorithm terminates since there are no more changes. Note that the solution was arrived at after only three iterations.

4.3 Performance Factors

The performance of the flow analysis, in terms of program time is not really treated as an issue here. What is of interest in the algorithms is how long it takes the sets to converge. Although no formal proofs of these algorithms are presented, the bibliography contains references to texts where such proofs can be found.

If the IN- OUT algorithm (Algorithm 3.6) is considered it can be seen that the iteration will eventually halt. An induction on the number of times the algorithm statement

NEWIN = set union of NEWIN and OUT[**a**]

will show that IN[b] is always a subset of NEWIN. The IN's are always increasing and as they are subsets of a finite set they cannot increase in size in an indefinite manner. In the iterative loop there must come a pass where NEWIN = IN[b] for every block. This situation will cause termination of the algorithm, since the flag CHANGE will remain FALSE.

Theory shows that the upper bound on the number of iterations around the WHILE loop is the number of nodes in the flow graph. Intuitively, the reason is that if a definition reaches a point, it can do so along a cycle-free path, and the number of nodes in a flow graph is an upper bound in the number of nodes in a cycle-free path. So that each time around the while-loop the definition progresses by at least one node along the path in question. For a properly ordered set of basic blocks there is empirical evidence to suggest that the number of iterations may be under 5.

4.4 Summary

As the motivating factor behind this work is to study the subject of optimizing a *programming language*, an important decision to be made is which approach should be used to achieve this aim. One of the first decisions is that any design carried out will be *architecture independent.* This will avoid the great cost and futility of repeating the effort for other architectures. Also this decision fits in with the software engineering principle of implementing code which is reusable. In order to port the tool developed it will be a simple matter of compiling and linking the tool on the new platform.

The main body of work is in the choice of an appropriate method of performing data flow analysis. The criterion to be satisfied here is the assessment of how well a particular method will suit the type of language with which we have to work. This means that the author had to become familiar with current methods of data flow analysis. It is found from this study that there are no immediate *off the shelf* methods which will suit the abstract syntax tree representation. The natural question then arises : "Is it possible to take a standard method and modify it to suit the problem on hand ?" The reasoning behind choosing this approach is that all algorithms presented solve sets of data flow equations and appear to be independent of the form of *intermediate code* employed. If this path is followed then it will be possible to develop data flow analysis for the language at hand. Although no formal proof that this is the case is presented here this conclusion is arrived at by empirical means. Proof of the solutions to data flow equations are available in the standard texts. (See Bibliography). Firstly a pencil and paper model of the flow analysis algorithms are constructed and followed through by means of tracing with a pencil. Once satisfied that these are functionally correct , the algorithms are coded. To test these algorithms simple test examples are written and then worked examples from text books are analysed and the results checked [Aho 77]. As well as modifying data flow algorithms it is also necessary to provide set handling functionality.

A debugging routine is also implemented as part of the flow analysis which dumps the *intermediate code* in ASCII form as well as the basic block divisions and the contents of all sets and lists of data collected. This function allows the testing of the analysis results. (See Appendix D).

5 CONCLUSION

The goal of this thesis is to propose algorithms which will solve the problem of data flow analysis for an intermediate code representation of the abstract syntax tree type. The author would like to provide some insight into the reasons for performing a data flow analysis, as well as point out future directions which this work could take. The usefulness of data flow analysis results are also discussed.

Data flow analysis provides a wealth of information about a program's data usage. This information could never be constructed by a reader of a program because of the complex data paths which are possible. Data flow analysis results can be utilized in a number of ways. The main category of application is the construction of optimizing tools. It is fundamental to the development of an intelligent optimizer to perform data flow analysis. Only a limited local *peephole optimization* is possible without adequate information of a type provided by data flow analysis. This type of optimization is usually highly *architecture dependent* as it is performed on a small window of *machine code*.To illustrate this type of optimization, consider the elimination of the store instruction as an example in the following piece of code :

```
LOD    R1, LOC1
STO    R1, LOC1
```

An optimizer is implemented as a program which reads the intermediate code statements and follows built in optimization rules. The decision as to whether a certain rule applies, or not, is made by consulting the results of the data flow analysis. The amount of program optimization is therefore dependent on the level of data flow analysis.

There is another category of application which utilizes the results of data flow analysis. This category encompasses the tools which provide information about the structure of a computer program to the user. These tools can be implemented to provide a rich set of information. Examples of such information are unreachable code, unused variables, unused parameters and uninitialized variables. Such tools could be gainfully employed in the quality control area of software development.

For example, rules as to what constitutes an acceptable program could be drawn up and these tools would tell the user if any of these rules are being broken.

Future work in this area would be to complete the *data flow analysis*. This work proves the point that it is possible to modify and implement algorithms to carry out data flow analysis to suit the type of *intermediate code* on hand. The analysis has been implemented to the point of collecting data which will be of use in certain types of optimization. There is a wider range of data which could be collected, such as def-use chain information, and very busy expressions. In order to implement this it will simply be a matter of building on the work which has already been carried out by the author. Once the data flow analysis is brought to its conclusion it will be possible to implement some of the applications which utilize it.

An obvious application would be to implement an optimizer from the results of the data flow analysis. The *intermediate code* should be studied to find which cases of optimization would prove most fruitful. Only those cases yielding an appreciable gain in performance or reduction in size should be considered.

Secondly the analysis results can be used to construct tools which will Indicate the quality of the program code under observation. As mentioned previously these tools will point out poor use of data such as non-use or failure to initialize them. Dead code regions might point to coding errors. Tables can be drawn up to give statistics on the use of data.

Finally it is intended that the material contained in this thesis should provide a student reader with an introduction to the field of data flow analysis and code optimization. It should be pointed out that the material contained here is concerned with the practical rather than to the theoretical aspect of dealing with a problem. This introduction is by means of demonstrating a practical solution to the problems encountered in developing a complex piece of software. It should demonstrate to the student the type of problem encountered in software engineering.

In conclusion, the author believes that the original goal of implementing a data flow analysis for abstract syntax tree representation has been reached.

APPENDIX A SYNTAX TREE INTERMEDIATE FORM

The *intermediate code* is presented in this appendix. This form is a simplified version of a real case, for the purpose of illustration. The fundamental layout is a record containing the *intermediate code* node. Here the record is presented as a C struct. These nodes are dynamically created in memory as part of the *semantic analysis* phase of the compiler front end. The *intermediate code* takes the shape of a binary tree. Each node may have one or two children, or it may be a terminating node. Each node has the ability to point to its children. The node type is held in the node_type field. Constants defining the node types are held in the #define table. Any text which a node may own is also stored as a pointer. Such text could be identifier names, function names and textual representation of constants. Miscellaneous flags are also stored.

```
/*  intermediate code node definition              */

    struct tnode
    {
                    struct tnode    *p_node;
                    struct tnode    *s_node;
                    struct tnode    *left_node;
                    struct tnode    *right_node;
                    int             node_type;
                    int             node_flags;
                    char            *text;
    };
```

Figure A1: Intermediate Code Node

```
/*          node types #define table                                        */

#define EOF             1       /*      end of file marker                  */
#define ENDFUNC1        2       /*      end of function 1                   */
#define ENDFUNC2        3       /*      end of function 2                   */
#define LINE            4       /*      line number                         */
#define LABELDEF        5       /*      label reference                     */
#define ARG             6       /*      function argument                   */
#define LOCDEF          7       /*      local variable definition           */
#define STATICDEF       8       /*      static variable definition          */
#define GLOBALDEF       9       /*      global storage                      */
#define EVAL            10      /*      assignment statement                */
#define BRANCH          11      /*      control flow statement              */
#define COMPARE         12      /*      comparison statement                */
#define ERROR           13      /*      error code                          */
#define RETURN          14      /*      return statement                    */
#define CONSTANT        15      /*      constant  information               */
#define NULL            16      /*      empty statement                     */
#define LABELREF        17      /*      label reference                     */
#define LOCALARG        18      /*      local argument                      */
#define LOCALVAR        19      /*      local variable                      */
#define STATICVAR       20      /*      static variable                     */
#define GLOBALVAR       21      /*      global variable                     */
#define ARRAYREF        22      /*      array reference                     */
#define RECORDREF       23      /*      record reference                    */
#define NEGATE          24      /*      negate statement                    */
#define ABSOLUTE        25      /*      absolute value                      */
#define CONVINT         26      /*      convert to integer                  */
#define CONVFLOAT       27      /*      convert to float                    */
#define ADD             28      /*      addition  operator                  */
```

```
#define SUBTRACT    29    /*    subtraction operator          */
#define MULTIPLY    30    /*    multiplication  operator      */
#define DIVIDE      31    /*    division operator             */
#define POWER       32    /*    power operator                */
#define MOD         33    /*    remainder operator            */
#define MAX         34    /*    maximum function              */
#define MIN         35    /*    minimum function              */
#define ARGVAL      36    /*    argument value                */
#define ARGREF      37    /*    argument reference            */
#define FUNCVALUE   38    /*    function value                */
#define TEXTERASE   39    /*    erase text                    */
#define CALL        40    /*    function call                 */
#define CALLPTR     41    /*    indirect function call        */
#define RESERVE     42    /*    reserve array storage         */
#define TRUNCATE    43    /*    truncate float                */
#define CREATE      44    /*    allocate dynamic record       */
```

Figure A2: Node Type Table

APPENDIX B DATA STRUCTURES

The following data structures are referred to in the flow analysis algorithms. Very general structures are presented which would ideally be records in any high level language, without any attempt to define types for the fields. The fields can be considered as 32 bit fields for the purpose of discussion, since this would allow the storing of integers, pointers and string information. The intention is to give a guide-line as to what data may be required in reality.

B.1 Basic block information

next_block_ptr	linked list pointer to next block
block_number	block id
block_start	pointer to block start node
block_end	pointer to block end node
gen_set	generated definitions bit vector
kill _set	killed definitions bit vector
in_set	definitions live on entry to basic block
out_set	definitions live on basic block exit
gen_list	temporary linked list of generated definitions
kill_list	temporary linked list of killed definitions
block_pred_list	basic block predecessors
block_succ_list	basic block successors

Figure B.1: Basic block information

B.2 Definition points

def_node	definition node
def_pnode	parent node
def_variable	definition variable
def_number	definition number
def_bb	owner basic block
next_def	pointer to next record

Figure B.2: Definition points

B.3 Use points

use _node	use node
use_pnode	parent node
use_variable	use variable
ud_set	use-def set pointer
use_bb	owner basic block
next_use	pointer to next use record

Figure B.3: Use points

B.4 Label Definition

label_number	label identifier
block_number	owner basic block
next _label	pointer to next record

Figure B.4: Label Definition

APPENDIX C SET MANIPULATION FUNCTIONS

```
/*
   GENOPT -  Set handling routines
*/

#define  TRUE   1
#define  FALSE  0
#include <malloc.h>
#define SETSIZE(x)  (x+32)/32

/*
   GENOPT -  Set handling routines
   Create a dynamic set
*/
long *MakeSet(Size)
long Size;
{
     long *SetPtr;
     SetPtr = (long *)calloc((int)(SETSIZE(Size)+1), sizeof(long));
     *SetPtr = Size;
     return(SetPtr);
 }
```

```
/*
   GENOPT -  Set handling routines
    Compare two similar sized sets.
   Return 0 if sets are equal.
   Return 1 otherwise.
*/
long CompareSet(Set1, Set2)
 unsigned long *Set1, *Set2;
 {
     int i,       Size;
     if((Set1 == 0L)&(Set2 == 0L))
               return(1L);
     Size = SETSIZE(*Set1);
     for(i = 0; i <= Size; i++)    {
               If(*Set1 != *Set2)
                         return(1L);
               Set1++; Set2++;
     }
     return(0L);
 }
```

```
/*
    GENOPT -  Set handling routines
    Perform the complement of two sets.
*/
long *ComplementSet(Set1, Set2)
unsigned long *Set1, *Set2;
{
      int i;
      long *RetPtr,         *ReturnSet,          Size;
      if(Set1 == 0)                    /* NULL set          */
                  return(0L);
      if(Set2 == 0){                   /* return the A set   */
                  ReturnSet = MakeSet(*Set1);
                  CopySet(ReturnSet, Set1);
                   return(ReturnSet);
      }
      if(*Set1 == *Set2)              /* make correct size    */
                  Size=*Set1;
      else
                  if(*Set1>*Set2)
                              Size = *Set1;
                  else
                              Size = *Set2;
      RetPtr = ReturnSet = MakeSet(Size);
      for(i = 0; i < SETSIZE(Size); i++) {
                  Set1++; Set2++; ReturnSet++;
                  *ReturnSet = *Set1 & ~*Set2;
      }
```

```
    return(RetPtr);
}
/*
   GENOPT -  Set handling routines
   Copy a set.
*/
CopySet(SrcSet, DestSet);
unsigned long *SrcSet, *DestSet;
{
    int i;
    long Size;
    Size=*SrcSet;
    if(*DestSet<Size)              /* make correct size   */
    DestSet = MakeSet(Size);
    for(i = 0; i < SETSIZE(Size); i++)
    {
            SrcSet++; DestSet++;
            *DestSet = *SrcSet;
    }
}
```

```
/*
   GENOPT -  Set handling routines
   Perform the intersection of sets
*/
IntersectSet( set1, set2, set3 )
unsigned long *set1, *set2, *set3;
{
      int size, group;
      size = *set1;
      set1++; set2++; set3++;
      *set3 = *set1 & *set2;
      for( group = 1; group < size; group++)  {
                  set1++; set2++; set3++;
                  *set3 = *set1 & *set2;
      }
}
/*
   GENOPT -  Set handling routines
   Perform the union of sets
*/
long *UnionSet(Set1, Set2)
unsigned long *Set1, *Set2;
{
      int i;
      long *RetPtr,        *ReturnSet,         Size;
      if((Set1 == 0L)&(Set2 == 0L))         /* both NULL          */
                  return(0L);
      if(Set2 == 0L){
                  ReturnSet = MakeSet(*Set1);
```

```
        CopySet(ReturnSet, Set1);
        return(ReturnSet);
    }
    if(Set1 == 0L){
        ReturnSet = MakeSet(*Set2);
        CopySet(ReturnSet, Set2);
        return(ReturnSet);
    }
    if(*Set1 == *Set2)          /* make correct size    */
        Size=*Set1;
    else
        if(*Set1>*Set2)
            Size = *Set1;
        else
            Size = *Set2;
    RetPtr = ReturnSet = MakeSet(Size);
    for(i = 0; i < SETSIZE(Size); i++) {
        Set1++; Set2++; ReturnSet++;
        *ReturnSet = *Set1 | *Set2;
    }
    return(RetPtr);
}
```

```
/*
   GENOPT -  Set handling routines
   Insert a member into a set
*/
long InsSet(Set, Entry)
 *Set; long  Entry;
{
      int Stop, Group;
      Set++;
      Entry = Entry - 1;
      Stop = Entry/32;
      for(Group = 1; Group <= Stop; Group++)    {
                 Set++;       Entry = Entry - 32;
                 *Set = *Set | ( 1L << Entry );
      }
}
```

```
/*
   GENOPT -  Set handling routines
   Set debugging routine
*/
long WriteSet(Set)
long *Set;
{
    int Size, i, j;
    Size = SETSIZE(*Set);
    Set++;
    for(j = 1; j <= Size; j++)  {
            for(i = 0; i <= 31; i++)  {
                    if(*Set & ( 1L << i))
                            printf("1");
                    else
                            printf("0");
            }
            Set++;
            printf("  ");
    }
    printf("\n");
}
```

```
/*
   GENOPT -  Set handling routines
   Remove a set member - First see if it's there
*/
long RemSet(Set, Entry)
long *Set; long  Entry;
{
     int Stop, Group;
     Set++;
     Entry = Entry - 1;
     Stop = Entry/32;
     for(Group = 1; Group <= Stop; Group++)    {
               Set++;
               Entry = Entry - 32;
     }
     if(*Set&(1L << Entry))    {
               *Set = *Set^(1L << Entry);
     }
}
```

```
/*
   GENOPT -  Set handling routines
   Check for set membership.
*/
long MemberSet(Set, Entry)
long *Set; long Entry;
{
     int Stop, Group, Tmp;
     Set++;
     Tmp = Entry - 1;
     Stop = Tmp/32;
     for(Group = 1; Group <= Stop; Group++)    {
               Set++;
               Tmp = Tmp - 32;
     }
     if(*Set & ( 1L << Tmp ))
               return(TRUE);
     else
               return(FALSE);
}
```

APPENDIX D EXAMPLES

```
main

define I,  J as integer variables

'L1'
     I = 2
     J = I + 1
'L2'
      I = 1
     if J = 999
               go to L1
     endif

'L3'
     J = J + 1
     if J = 999
               go to L4
     endif
      J = J - 4
'L4'
     go to L2

end
```

Figure D.1: Code example.

```
-------- BB.NO  #1 -------

SUCC list of block   1
SUCC :   3

IN.SET of block   1 :        00000000000000000000000000000000

OUT.SET of block   1 :       00000000000000000000000000000000

CODE :

system prologue 1

-------- BB.NO  #2 -------

SUCC list of block   2
SUCC :   3

IN.SET of block   2 :        00000000000000000000000000000000

OUT.SET of block   2 :       00000000000000000000000000000000

CODE :

system prologue 2

-------- BB.NO  #3 -------

PRED list of block   3
PRED :   1
PRED :   2

SUCC list of block   3
BLOCK:   4

IN.SET of block   3 :        00000000000000000000000000000000

OUT.SET of block   3 :       00000000000000000000000000000000

CODE :

system prologue 3
```

-------- BB.NO #4 -------

PRED list of block 4
PRED : 3
PRED : 6

SUCC list of block 4
BLOCK: 5

GEN.SET of block 4 : 11000000000000000000000000000000

GEN.FILE of block 4
Var : I Def : 1
Var : J Def : 2

KILL.SET of block 4 : 00111000000000000000000000000000

KILL.FILE of block 4
Var : I Def : 3 BB : 5
Var : J Def : 4 BB : 9
Var : J Def : 5 BB : 12

IN.SET of block 4 : 01111000000000000000000000000000

OUT.SET of block 4 : 11000000000000000000000000000000

CODE :

I = 2

J = I + 1

' L2 '
-------- BB.NO #5 -------

PRED list of block 5
PRED : 4
PRED : 13

SUCC list of block 5
SUCC : 8
SUCC : 6

GEN.SET of block 5 : 00100000000000000000000000000000

GEN.FILE of block 5
Var : I Def : 3

KILL.SET of block 5 : 10000000000000000000000000000000

KILL.FILE of block 5
Var : I Def : 1 BB : 4

IN.SET of block 5 : 11111000000000000000000000000000

OUT.SET of block 5 : 01111000000000000000000000000000

CODE :

I = 1

IF J = 999

-------- BB.NO #6 -------

PRED list of block   6
PRED :   5

SUCC list of block   6
SUCC :   4

IN.SET of block   6 :   01111000000000000000000000000000

OUT.SET of block   6 :   01111000000000000000000000000000

CODE :

GO TO L1

-------- BB.NO #7 -------

SUCC list of block   7
SUCC :   8

IN.SET of block   7 :   00000000000000000000000000000000

OUT.SET of block   7 :   00000000000000000000000000000000

CODE :

ENDIF

-------- BB.NO #8 -------

PRED list of block   8
PRED :   5
PRED :   7

SUCC list of block   8
SUCC :   9

IN.SET of block   8 :   01111000000000000000000000000000

OUT.SET of block   8 :   01111000000000000000000000000000

CODE :

' L3 '

-------- BB.NO #9 -------

PRED list of block 9
PRED : 8

SUCC list of block 9
SUCC : 12
SUCC : 10

GEN.SET of block 9 : 00010000000000000000000000000000

GEN.FILE of block 9
Var : J Def : 4

KILL.SET of block 9 : 01001000000000000000000000000000

KILL.FILE of block 9
Var : J Def : 2 BB : 4
Var : J Def : 5 BB : 12

IN.SET of block 9 : 01111000000000000000000000000000

OUT.SET of block 9 : 00110000000000000000000000000000

CODE :

J = J + 1

IF J = 999

-------- BB.NO #10 -------

PRED list of block 10
PRED : 9

SUCC list of block 10
SUCC : 13

IN.SET of block 10 : 00110000000000000000000000000000

OUT.SET of block 10 : 00110000000000000000000000000000

CODE :

GO TO L4

-------- BB.NO #11 -------

SUCC list of block 11
SUCC : 12

IN.SET of block 11 : 00000000000000000000000000000000

OUT.SET of block 11 : 00000000000000000000000000000000

CODE :

ENDIF

-------- BB.NO #12 -------

PRED list of block 12
PRED : 9
PRED : 11

SUCC list of block 12
SUCC : 13

GEN.SET of block 12 : 00001000000000000000000000000000

GEN.FILE of block 12
Var : J Def : 5

KILL.SET of block 12 : 01010000000000000000000000000000

KILL.FILE of block 12
Var : J Def : 2 BB : 4
Var : J Def : 4 BB : 9

IN.SET of block 12 : 00110000000000000000000000000000

OUT.SET of block 12 : 00101000000000000000000000000000

CODE :

J = J - 4

' L4 '

-------- BB.NO #13 -------

PRED list of block 13
PRED : 10
PRED : 12

SUCC list of block 13
SUCC : 5

IN.SET of block 13 : 00111000000000000000000000000000

OUT.SET of block 13 : 00111000000000000000000000000000

CODE :

GO TO L2

-------- BB.NO #14 -------

SUCC list of block 14
SUCC : 15

IN.SET of block 14 : 00000000000000000000000000000000

OUT.SET of block 14 : 00000000000000000000000000000000

CODE :

END

-------- BB.NO #15 -------

PRED list of block 15
PRED : 14

IN.SET of block 15 : 00000000000000000000000000000000

OUT.SET of block 15 : 00000000000000000000000000000000

CODE :

system epilog

```
------ Program Data ------

Definition points for this block

Var : I    BB :   4 Def :  1
Var : J    BB :   4 Def :  2
Var : I    BB :   5 Def :  3
Var : J    BB :   9 Def :  4
Var : J    BB :  12 Def :  5

Use points for and UD set these blocks

Var : I    BB :   4

UD.SET(4)          10000000000000000000000000000000

Var : J    BB :   9

UD.SET(9)          01011000000000000000000000000000

Var : J    BB :  12

UD.SET(12)         00010000000000000000000000000000
```

Figure D.2: Test result.

Bibliography

Aho 77 Aho, A. V. and J. D. Ullman [1977]. *Principles of Compiler Design,* Addison-Wesley, Reading, Mass.

Aho 86 Aho, A. V. , Ravi Sethi and J. D. Ullman [1986]. *Compilers : Principles, Techniques and Tools,* Addison-Wesley, Reading, Mass.

All 71 Allen, F. E. and J. Cocke [1971]. *A Catalogue of Optimizing Transformations,* IBM Research, RC 3548, Yorktown Heights, New York.

All 76 Allen, F. E. and J. Cocke [1976]. *A Program Data Flow Analysis Procedure,* Communications of the ACM, Volume19, Number 3, March 1976.

Fis 88 Fischer, C. N. and R. J. LeBlanc [1988]. *Crafting a Compiler,* Benjamin / Cummings, Menlo Park, California.

Hec 77 Hecht, M.S. [1977]. *Flow Analysis of Computer Programs,* North-Holland, New York.

Ken 72 Kennedy, K. [1972]. *Safety of Code Motion,* Computer Math, Gordon and Breach, Science Publishers Ltd.

Mor 79 Morel, E. and C. Renvose [1979] . *Global Optmization by Supression of Partial Redundancies,* Communications of the ACM, Volume 22, Number 2, February 1979.

Muc 81 Muchnick, S. S. and N. D. Jones [1981]. *Program Flow Analysis : Theory and Applications,* Prentice-Hall, Englewood Cliffs, N. J.

Ull 75 Ullman, J. D. [1975]. *A Survey of Data Flow Analysis Techniques,* Second USA-JAPAN Computer Conference.

Wir 76 Wirth, N. [1976]. *Algorithms + Data Structures = Programs,* Prentice-Hall, Englewood Cliffs, N. J.