ABSTRACT

Machine independent and language independent methods of optimizing the execution time of a source program are described and implemented. The approach is based on the topological characteristics of a program. A program partitioning into 'strongly connected' regions is developed which permits modular optimization.

Although most of the existing optimizing transformations have been described , two principal methods are implemented,redundant instruction elimination and code motion from one part of a source program to another.

The final chapter contains examples of programs run on the developed optimizer which embodies the described methods.Timing considerations of these programs and their optimized versions are also given in order to offer a further measure of the improvements,in terms of execution time, made by the optimizer.

A SURVEY OF PROGRAM OPTIMIZATION

AND

THE IMPLEMENTATION OF SOME OPTIMIZING TECHNIQUES

by

Nicholas Alexandrakis

A dissertation presented for the degree of Master of Science
of the University of St. Andrews.

1973

Declaration

I declare that the following thesis is a record of research work carried out by me, that the thesis is my own composition, and that it has not been presented in application for a higher degree previously.

Nicholas Alexandrakis

Certificate

I certify that Nicholas ALEXANDRAKIS, graduate in Mathematics of Athens University, has spent four terms as a research student in the Computing Laboratory of the United College of St. Salvator and St. Leonard in the University of St. Andrews, that he has fulfilled the conditions of Ordinance 51 (St. Andrews), and that he is qualified to submit the accompanying thesis in application for the degree of Master of Science.

J. M. Wilson
Supervisor

ACKNOWLEDGEMENTS

My sincere thanks to:

Professor A. J. Cole who provided the opportunity for this study.

Mr. J. M. Wilson, my thesis advisor whose help, stimulation and encouragement have been invaluable.

Mrs. M. Todd and Mrs. C. Evans-Smith for the thorough typing of this thesis.

CONTENTS

CHAPTER 1

INTRODUCTION

One of the features that promulgated the widespread acceptance of high level languages is the fact that computer programs written in this more convenient form could be translated to machine language, or to a form close to it, by another computer program, the compiler, running on the same or possibly a different machine. Due to the complexity of the compiling process, however, a significant amount of the total time of a job is usually spent during compilation. This loss is particularly heavy in university environments where the number of submitted programs and the proportion of errors are higher than usual. Moreover, debugged programs tend to run in production very rarely in such an environment. Fast compilation is, therefore, one objective of every compiler writer.

On the other hand, production programs written in a high level language should be compiled into object code competitive with 'hand-written' programs, otherwise the advantages of the high level language (elegance, power, etc.) would be somewhat compromised. The new breed of 'languages for implementation of systems' increased the requirements for good optimizing compilers to the extreme.

Experience has shown that fast compilation and production of efficient object code can not be successfully combined into a single compiler, although it has been observed that a compiler with some local optimization often runs faster than one without any, due to the shorter object program produced.

There are numerous fast one-pass compilers in the market today, used in program development and teaching. On the other hand there is a large number of multi-pass compilers performing optimization in various levels. Some of these compilers produce object programs which are almost as efficient as 'hand written' programs, and at far less expense. For example the authors of the OS/360 FORTRAN-H code-optimizing compiler state that at the cost of a 40 percent increase in compilation time they produce code which is 25 percent smaller and which executes in one-third the time of that produced by the FORTRAN-G compiler. But against these impressive measurements the existing optimizing techniques can not be considered satisfactory as yet. They are both time and (especially) space consuming. For example the size of the FORTRAN H compiler is 400 kbytes and it requires a minimum of 256 kbyte storage in order to process 600 to 700 statements!

In this thesis most of the theoretical work done so far in the area of object code optimization is presented and discussed. In addition an algorithm performing two basic optimizing transformations, redundant subexpression elimination and code motion, is described. The reasons for implementing these two special kinds of optimization are:-

(i) They are intimately related. By moving code from one part of a program to a more advantageous part, more redundancies may be exposed for elimination.

(ii) Redundant subexpressions and invariant instructions in frequently executed parts of a problem program are the commonest reasons of program inefficiency.

(iii) They are machine and language independent.

Because of the wide availability of FORTRAN programs, FORTRAN was chosen as the source language for the optimizing algorithm. The algorithm itself was also written in FORTRAN. It is doubtful that the adopted optimization techniques could have been implemented and debugged in a reasonable amount of time if the optimizer had not been written in a high level language. The output generated by the optimizer is also in FORTRAN in order to offer a convenient visual measure of the improvements made in problem programs.

1.1 A Brief History of Optimization

The problem of object code optimization has received a lot of attention since the development of the first FORTRAN compiler in 1958 and there are, therefore, many investigations of this area described in the literature. Hence, we will not attempt to produce a complete catalogue but instead will select those efforts which supplied most of the ideas for the development of this project.

Realising the importance of the program flow analysis in optimization R. T. Prosser (30), in 1959, showed how Boolean matrices can be used efficiently in the analysis of flow diagrams. He also introduced the concept of 'dominance relations' which was used ten years later in the development of the FORTRAN-H compiler.

In August 1965 C. W. Gear (15) summarised some machine independent optimizations and proposed a three pass compilation incorporating these strategies. These optimization processes remain the basis for most of today's investigations.

A significant amount of research into the area of optimization has centred around the work of F. E. Allen (3), J. Cocke (7,8,9) and J. Schwartz (9). Their influence is very evident in the optimizations of the FORTRAN-H compiler which are described by E. Lowry and C. Medlock (26). Much of the work done by Allen and Cocke concerns itself with the processing of the control flow structure of programs and hence contains a considerable amount of graph-theoretic investigation related to control flow representation. The identification of computations which can be moved, or eliminated, can be determined, as J. T. Schwartz indicated, by solving of a set of simultaneous Boolean equations. Since several other optimization problems can be similarly formulated, this technique promises to be of a rather general utility in the future.

1.2 Thesis Outline

This thesis contains five chapters. Chapter II is a survey of various techniques that have been used in the analysis of the control flow of a program. Some optimizing transformations are catalogued and discussed by Chapter III. In Chapter IV the developed optimizing algorithm is described in detail and finally, in Chapter V, a set of examples illustrating the implemented optimization strategies is presented.

CHAPTER 2

CONTROL FLOW ANALYSIS

2.1 Introduction

Any global optimizing procedure requires a knowledge of the data flow of the source program. For example it is very desirable to know which definitions of variables can affect computations at a given point in the program, and which uses can be affected by computations at a given point. Once we have this information we can answer such questions as: if an expression can be removed from a program segment where can it be correctly and profitably placed, what are the consequences of this movement to the data flow of the program? etc.

Clearly the main source of this kind of information is the control flow graph of the program. In addition it is essential to codify these flow relationships in a suitable way for processing by a machine. A few formal approaches to this subject will be presented in this chapter.

In the first section, 'Basic concepts', all relevant information about directed graphs is catalogued.

In the second section, some applications of Boolean matrices to control flow analysis are presented.

In the third section the concept of the strongly connected region is introduced, a procedure is given for its construction and its use in program optimization is outlined.

In the fourth section, 'Dominance relationships' are defined.

In the last section of this chapter, 'Intervals', their properties and their use are discussed. In addition a procedure is given for their construction.

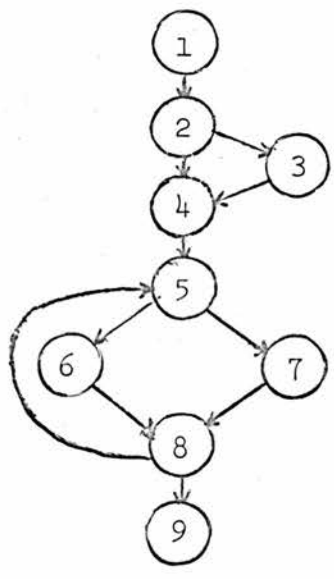
2.2 Basic Concepts

A basic block is a linear sequence of instructions having the property that, if the first instruction is executed all of them will be executed.

A directed graph is a structure consisting of a set of nodes. Generally each node carries some kind of information. In addition it may point to or be pointed at by any other node of the graph. Directed graphs representing the control flow of a program are called control flow graphs. In a control flow graph the nodes represent basic blocks and the pointers represent control flow paths.

More formally, a control flow graph, G, can be denoted by $G=(B,E)$ where B is the set of basic blocks $\{b_1, b_2, \ldots, b_n\}$ in the program and E is the set of pointers (or directed edges) $\{(b_i,b_j) \mid i,j \in \{1,\ldots,n\}\}$. Each directed edge can be denoted by an ordered pair (b_i,b_j) indicating that the flow goes from the ith block to the jth one.

Let us now look at a typical control flow graph:



We can now define a <u>successor</u> <u>function</u>, S, such that: $S(b_i)=\{b_j|(b_i,b_j)\in E\}$. Clearly $S(b_i)$ is the set of the immediate successors of the block b_i. Accordingly the reverse function of S, P, gives the immediate predecessors of the block b_j: $P(b_j)=\{b_i|(b_i,b_j)\in E\}$.

Generally a directed graph is called <u>connected</u> if any node in the graph can be reached by any other node by successive applications of S (or P). Control flow paths are always connected.

A <u>subgraph</u> of a directed graph, $G=(B,E)$, is a directed graph $G'=(B',E')$ in which $B'\subset B$, $E'\subset E$, $G\cap G'=G'$ and $G\cup G'=G$. In addition the successor function S' for G' is now defined as $S'(b_i)=\{b_j|(b_i,b_j)\in E'\}$.

A <u>path</u>, P, in the control flow graph of a program is a sequence of blocks $b_1,\ldots, b_i$ such that for each $b_k\in P$ it follows that $b_{k+1}\in S(b_k)$. A path then represents a control flow sequence in a program.

A block b is said to be a <u>successor</u> of block a if there exists one path $P=\{b_1,\ldots,b_i\}$ for which $b_1=a$ and $b_i=b$. In this case a is said to be a <u>predecessor</u> of b.

A <u>closed</u> <u>path</u> is a path $P=\{b_1,\ldots,b_i\}$ in which $b_1=b_i$. If all basic blocks in P are different from each other the closed path is <u>simple</u>; otherwise it is <u>composite</u>.

The <u>length</u> of a path is the number of edges in the sequence. More formally let us define a distance function, D, such that for any path $P=(b_1,\ldots,b_i)$, $D(P)=i-1$. The shortest path Dmin between two points, a and b, will be defined by: $Dmin=MIN(D(P_1),D(P_2),\ldots)$ for all $P_i=\{a,\ldots,b\}$.

2.3 Codification of the Flow Relationships in a Program

Primarily the control flow of a program can be determined by detailed specifications describing it or it can be given in a convenient geometric representation by means of control flow graphs. Unfortunately, neither of these forms is immediately usable by a machine. A much more machine-oriented representation may be given by means of Boolean matrices. A Boolean matrix is a matrix whose entries may be either zero or one. Assume that the control flow relationships in a program are given in the form of a control flow graph. If n is the number of the nodes in the graph we can construct a $n \times n$ Boolean matrix, $C=(c_{ij})$ called the connectivity matrix associated with the graph, according to the following rule: $c_{ij}=1$ if and only if block j is an immediate successor of block i, it is zero otherwise. Obviously the ith row (column) of this matrix holds all immediate successors (predecessors) of the ith block respectively. It is apparent that this matrix is unique easy to construct and easy to handle. In addition certain elementary operations on the connectivity matrix yield detailed information on the program flow. Given now two $n \times n$ Boolean matrices A and B we define the Boolean sum, $A \vee B$, as that matrix S whose (i,j)th entry is $a_{ij} \vee b_{ij}$. We also define the Boolean product, $A \wedge B$, as that matrix P whose (i,j)th entry is $\bigvee_{k=1}^{n}(a_{ik} \wedge b_{kj})$. Let C^2 be the square of the connectivity matrix C. By the definition it follows that $(c_{ij})^2 = \bigvee_{k=1}^{n}(c_{ik} \wedge c_{kj})$ (1). Assume now that $(c_{ij})^2=1$. By equation (1), $\bigvee_{k=1}^{n}(c_{ik} \wedge c_{kj})=1$ which implies that there is (at least) one k with $c_{ik} \wedge c_{kj}=1$. Therefore $c_{ik}=1$ and $c_{kj}=1$. Hence control goes from block i to block k and from block k to block j. Consequently $(c_{ij})^2=1$ implies that there is a path $p=(b_i,\dots,b_j)$, whose length is equal to two, connecting block i and block j. Conversely, if there is such a path then $(c_{ij})^2=1$. In this case $c_{ik} \wedge c_{kj}=1$ which implies $\overset{n}{\underset{k=1}{\vee}}(c_{ik} \wedge c_{kj})$, so $(c_{ij})^2=1$.

Let us form now the matrix $R=C \vee C^2$. The (i,j)th entry of R will be given by: $r_{ij}=c_{ij} \vee (c_{ij})^2$ (2). By the previous result, the assertion $r_{ij}=1$ implies that there is a path $P=\{b_i,\ldots,b_j\}$whose length is less than or equal to two. Conversely, the existance of such a path implies that $r_{ij}=1$. We shall show that much the same conclusions are true for every power of the connectivity matrix, (30). First a definition.

Definition

Let C be the connectivity matrix. We define $C^m=C \wedge C \wedge \ldots \wedge C$ (m times) and $R_m=R_{m-1} \vee C^m$ with $R_0=0$ and $m \in \{1,2,\ldots,n\}$.

Theorem

The (i,j)th entry of $C^m(Rm)$ is equal to one if and only if there is a path $P=\{b_i,\ldots,b_j\}$ whose length is equal (less than or equal) to m, respectively; it will be zero otherwise.

Proof

Given m, we shall prove both conclusions together by induction on m.

For m=1, the result is immediate. Suppose then that the conclusion holds for m=k, with $k \in \{1,\ldots,n-1\}$, and consider the case for m=k+1.

If the (i,j)th entry of C^{k+1} (R_{k+1}) is equal to one, it follows that $\overset{n}{\underset{r=1}{\vee}}(c_{ir} \wedge (c_{rj})^k)=1$ (1) $((r_{ij})_k \vee (c_{ij})^{k+1}=1$ (2))

By equation (1) it is implied that there is an $r \in \{1,\ldots,n\}$with $c_{ir}=1$ and $(c_{rj})^k=1$. Therefore by the definition of the connectivity matrix and the induction assumption, control goes from block i to block r and there is a path $P=\{b_r,\ldots,b_j\}$ whose length is equal to k. Hence there is a path $P'=\{b_i,\ldots,b_j\}$ whose length is equal to k+1.

By equation (2) it follows that:

either $(r_{ij})_k=1$ (3)

or $(c_{ij})^{k+1}=1$ (4)

By equation (3) and the induction assumption there is a path $P=\{b_i,\ldots,b_j\}$ whose length, L, is less or equal to k. Therefore L is less than or equal to k+1 too.

By equation (4) and the first conclusion it follows that there is a path $P=\{b_i,\ldots,b_j\}$ whose length, L, is equal to k+1. Therefore L is also less than or equal to k+1.

Obviously $(c_{ij})^{k+1}=0$ $((r_{ij})_{k+1}=0)$ implies that there is no such path.

The truth of the converse statement can be proved in a similar way.

This theorem has several striking consequences:

1. The limit, R, of the sequence Rm exists as a Boolean matrix. More specifically, Rm=R for each m greater than or equal to the longest path in the diagram. Because if it was not true we would have R_L not equal to R_{L+1}, where L is the length of the longest path in the diagram. Therefore it would exist an $(r_{ij})_L=0$ with $(r_{ij})_{L+1}=1$

or $\bigvee_{k=1}^{n}((c_{ik})^L \wedge C_{kj}) \vee (r_{ij})_L=1$

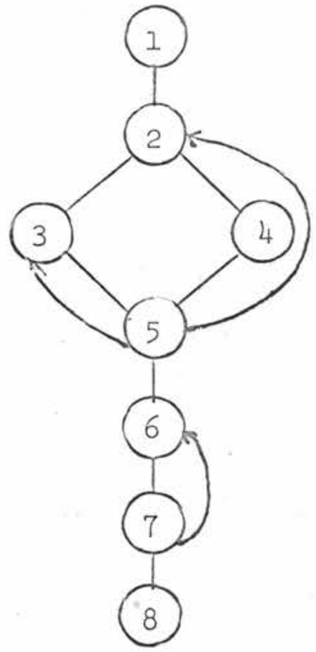
or $\bigvee_{k=1}^{n}((c_{ik})^L \wedge C_{kj})=1$

which implies that there is a k with $(c_{ik})^L \wedge C_{kj}=1$. Hence there is a path whose length is greater than L, a contradiction.

2. The (i,j)th entry of R is 1 if and only if there is a path, $P=\{b_i,\ldots,b_j\}$ of any length connecting blocks i and j.

Let us now illustrate this theory by applying it to a typical graph.

Consider the following control flow graph:



and the connectivity matrix representing it.

$$C = \begin{pmatrix} 0&1&0&0&0&0&0&0 \\ 0&0&1&1&0&0&0&0 \\ 0&0&0&0&1&0&0&0 \\ 0&0&0&0&1&0&0&0 \\ 0&1&1&0&0&1&0&0 \\ 0&0&0&0&0&0&1&0 \\ 0&0&0&0&0&1&0&1 \\ 0&0&0&0&0&0&0&0 \end{pmatrix}$$

Straightforward computation gives:

$$C^2 = \begin{pmatrix} 0&0&1&1&0&0&0&0 \\ 0&0&0&0&1&0&0&0 \\ 0&1&1&0&0&1&0&0 \\ 0&1&1&0&0&1&0&0 \\ 0&0&1&1&1&0&1&0 \\ 0&0&0&0&0&1&0&1 \\ 0&0&0&0&0&0&1&0 \\ 0&0&0&0&0&0&0&0 \end{pmatrix}$$

$$R_2 = \begin{pmatrix} 0&1&1&1&0&0&0&0 \\ 0&0&1&1&1&0&0&0 \\ 0&1&1&0&1&1&0&0 \\ 0&1&1&0&1&1&0&0 \\ 0&1&1&1&1&1&1&0 \\ 0&0&0&0&0&1&1&1 \\ 0&0&0&0&0&1&1&1 \\ 0&0&0&0&0&0&0&0 \end{pmatrix}$$

$$C^3 = \begin{pmatrix} 0&0&0&0&1&0&0&0 \\ 0&1&1&0&0&1&0&0 \\ 0&0&1&1&1&0&1&0 \\ 0&0&1&1&1&0&1&0 \\ 0&1&1&0&1&1&0&1 \\ 0&0&0&0&0&0&1&0 \\ 0&0&0&0&0&1&0&1 \\ 0&0&0&0&0&0&0&0 \end{pmatrix}$$

$$R_3 = \begin{pmatrix} 0&1&1&1&1&0&0&0 \\ 0&1&1&1&1&1&0&0 \\ 0&1&1&1&1&1&1&0 \\ 0&1&1&1&1&1&1&0 \\ 0&1&1&1&1&1&1&1 \\ 0&0&0&0&0&1&1&1 \\ 0&0&0&0&0&1&1&1 \\ 0&0&0&0&0&0&0&0 \end{pmatrix}$$

$$C^4 = \begin{pmatrix} 0&1&1&0&0&1&0&0 \\ 0&0&1&1&1&0&1&0 \\ 0&1&1&0&1&1&0&1 \\ 0&1&1&0&1&1&0&1 \\ 0&1&1&1&1&1&1&0 \\ 0&0&0&0&0&1&0&1 \\ 0&0&0&0&0&0&1&0 \\ 0&0&0&0&0&0&0&0 \end{pmatrix}$$

$$R_4 = \begin{pmatrix} 0&1&1&1&1&1&0&0 \\ 0&1&1&1&1&0&1&0 \\ 0&1&1&1&1&1&1&1 \\ 0&1&1&1&1&1&1&1 \\ 0&1&1&1&1&1&1&1 \\ 0&0&0&0&0&1&1&1 \\ 0&0&0&0&0&1&1&1 \\ 0&0&0&0&0&0&0&0 \end{pmatrix}$$

$$C^5 = \begin{pmatrix} 0&0&1&1&1&0&1&0 \\ 0&1&1&0&1&1&0&1 \\ 0&1&1&1&1&1&1&0 \\ 0&1&1&1&1&1&1&0 \\ 0&1&1&1&1&1&1&1 \\ 0&0&0&0&0&0&1&0 \\ 0&0&0&0&0&1&0&1 \\ 0&0&0&0&0&0&0&0 \end{pmatrix}$$

$$R_5 = \begin{pmatrix} 0&1&1&1&1&1&1&0 \\ 0&1&1&1&1&1&1&1 \\ 0&1&1&1&1&1&1&1 \\ 0&1&1&1&1&1&1&1 \\ 0&1&1&1&1&1&1&1 \\ 0&0&0&0&0&1&1&1 \\ 0&0&0&0&0&1&1&1 \\ 0&0&0&0&0&0&0&0 \end{pmatrix}$$

$$C^6 = \begin{pmatrix} 0&1&1&0&1&1&0&1 \\ 0&1&1&1&1&1&1&0 \\ 0&1&1&1&1&1&1&1 \\ 0&1&1&1&1&1&1&1 \\ 0&1&1&1&1&1&1&1 \\ 0&0&0&0&0&1&0&1 \\ 0&0&0&0&0&0&1&0 \\ 0&0&0&0&0&0&0&0 \end{pmatrix}$$

$$R_6 = \begin{pmatrix} 0&1&1&1&1&1&1&1 \\ 0&1&1&1&1&1&1&1 \\ 0&1&1&1&1&1&1&1 \\ 0&1&1&1&1&1&1&1 \\ 0&1&1&1&1&1&1&1 \\ 0&0&0&0&0&1&1&1 \\ 0&0&0&0&0&1&1&1 \\ 0&0&0&0&0&0&0&0 \end{pmatrix}$$

Since no path in the diagram is longer than 6 edges it follows that R_6 is the limit R. (We exclude loops of course)

From this matrix we verify immediately that block 1 is an entry block ($r_{i1}=0$ for every i) and block 8 is an exit block ($r_{8j}=0$ for every j). In addition blocks 2,3,4,5,6,7 are involved with loops because $r_{22}=r_{33}=r_{44}=r_{55}=r_{66}=r_{77}=1$.

Here is now a faster method of obtaining the matrix R due to S. Marshall (31).

1. Set R=C (C is the connectivity matrix)
2. Set j=1
3. Set i=1
4. If $r_{ij}=1$ set $r_{ik}=r_{ik} \vee r_{jk}$ for all $K \in \{1,\dots,n\}$
5. Set i=i+1. If $i \leq n$ go to step 4; otherwise go to step 6.
6. Set j=j+1. If $j \leq n$ go to step 3; otherwise stop.

Obviously the above algorithm is suggested by the following recursive definition of R:

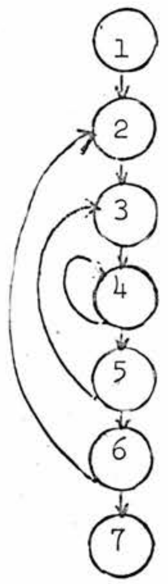
1. $(r_{ij})_o = c_{ij}$
2. $(r_{ij})_{k+1} = (r_{ij})_k \vee ((r_{ik+1})_k \wedge (r_{k+1j})_k)$
3. $r_{ij} = (r_{ij})_n$

Where k has the same meaning as the counter of the outer loop in the above algorithm.

2.4 The Use of Strongly Connected Regions

A strongly connected region of a directed graph is a directed subgraph, $G'=(B',E')$, such that, for any $b_i \epsilon B'$ and $b_j \epsilon B'$ there exists a path $P=\{b_i,\ldots,b_j\}$ connecting them.

Since the above definition applies even when $b_i \equiv b_j \equiv b$ we conclude that every node in a strongly connected region lies on at least one closed path. Consider the following directed graph:



Clearly there are three strongly connected regions here:

R(1)=4

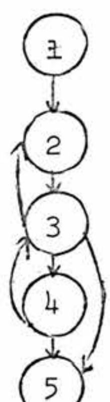
R(2)=3-4-5

R(3)=2-3-4-5-6

Thus, strongly connected regions have the same structure with respect to each other that conventional loops have, i.e. they are nested, or they can have no blocks in common and are thus "parallel". They should not overlap, of course.

The last condition necessitates a stricter selection of strongly connected regions: A <u>properly nested set of strongly connected regions</u>,(3), $S=\{R_1, R_2, \ldots, R_k\}$, is a partially ordered set such that for $i \neq j$ either $R_i \cap R_j = \emptyset$ or $R_i \cap R_j = R_i$; that is either R_i and R_j are disjoint or R_i is a subset of R_j. Since two overlapping regions are not allowed in S which one should be put in then? A reasonable action would be to give preference to the most frequently executed region. Conventional loops are considered generally as the busiest segments in a program. In addition their structure is quite characteristic: there is always a single block in the region, called an entry block of the region, with an edge pointing to it by a single block outside the region, called a predecessor block of the region. Furthermore it can be proved that between two overlapping regions at most one of them has a single predecessor and a single entry block. Thus, if two regions overlap we can give preference to the one with one entry and one predecessor block.

Consider the following control flow graph:



There are three regions involved in this graph:

R(1)=2-3

R(2)=3-4

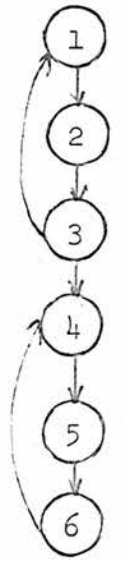
R(3)=2-3-4

Using the above rule we reject the first region because it has two predecessors, blocks 1 and 4, and two entry blocks, blocks 2 and 3. Therefore the region list should now contain the regions R(2) and R(3).

We shall show now how the theory of connectivity matrices described above is applied naturally to the construction of the region list of a graph.

First of all the connectivity matrix, C, associated with the given graph is constructed. Clearly if $c_{ii}=1$, block i forms a closed path itself. Therefore it will be put on the list.

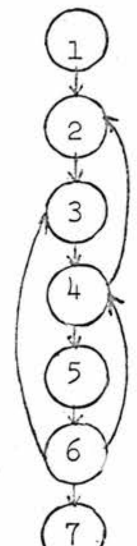
Next the matric C is raised to successive powers, r, $(1<r\leq n)$, where n is the number of the blocks in the given graph. Clearly the relation $(c_{ii})^r=1$ implies that block i belongs to a closed path of length r. But if there exist more than one closed path of the same length r in the graph, they should be separated out in a different way since the diagonal of C^r tells us only which blocks are involved in loops. Consider the following graph:



The closed paths (1,2,3) and (4,5,6) are both of length 3. Therefore it would appear that $(c_{ii})^3=1$ for all $i=1,2,\ldots 6$.

In order to separate these closed paths out, an Integer distance matrix, D, is maintained. The (i,j)th entry of D holds the maximum length of all paths connecting blocks i and j. Whenever $(c_{ii})^r=1$, all other blocks in the graph are tested whether they belong to the closed path initiated by block i or not. A block j belongs to the same closed path with block i iff $(c_{jj})^r=1$, $D_{ij}\neq o$, $D_{ji}\neq o$ and $D_{ij}+D_{ji}\leq r$. Whenever a constructed closed path is unique and complies with the definition of a strongly connected region it is put on the region list.

One might anticipate that a region R(k) generated by the analysis of C^r would be of length r and consequently no sorting had to be applied to the constructed region list. But consider the following graph:



Clearly the analysis of C^3 would produce the regions:

R(1)=2-3-4

R(2)=4-5-6

and R(3)=2-3-4-5-6

because every block in the last region is 3 edges from itself and not 5.

Primarily region R(1) is rejected because it overlaps with R(2) and it has more than one predecessor. Thus the region list is modified to:

R'(1)=4-5-6

R'(2)=2-3-4-5-6

Similarly the analysis of C^4 would produce the region:

R'(3)=3-4-5-6

which is covered by the region 2,3,4,5,6. Hence a time consuming sorting of the resultant list can not be avoided.

2.5 Dominance Relationships

Prosser (30) introduced the concept of dominance relationships in 1959 and Medlock (26) refined and used it in the development of the FORTRAN-H compiler in 1969. Before establishing these relationships, two special kinds of nodes will be defined. A terminal or exit node, b_i, in a directed graph, G, is a node with lack of successors. More formally b_i is an exit node if and only if $S(b_i)=\emptyset$ where S is the successor function defined earlier in this chapter.

Since a program entry node may be the first node of a loop and therefore have a predecessor we can not give an analogous definition for it. But we can introduce an arbitrary block, b_o, into the graph immediately preceeding all entry nodes of the graph. This block is called the <u>initial</u> <u>node</u> of the graph and, apparently, it lacks predecessors. In the remainder of this chapter, any reference to a graph will be to a directed graph with a single entry node, b_o, and a set of exit nodes $XN=\{x_1, x_2, \ldots\}$

A node b_i is said to predominate or back dominate a node, b_k, if b_i is on every path from b_o to b_k. More formally if P is the set of all paths connecting b_o and b_k, then the set of back dominators of b_k,$BD(b_k)$, is given by:

$$BD(b_k)=\{b_i|b_i\neq b_k \text{ and } b_i\epsilon P_i \text{ for all } P_i\epsilon P\}$$

Accordingly the immediate back dominator, b_i, of node b_k is the back dominator which is 'closest' to b_k, that is $b_i\epsilon B\ D(b_k)$ and for all $b_j\epsilon BD(b_k)$:

$$Dmin(b_i,b_K)\leq Dmin(b_j,b_k)$$

The back dominance relation is transitive: if block b_i predominates block b_j and block b_j predominates block b_k, then block b_i predominates block b_k. Further if block b_k is predominated by both blocks b_i and b_j, then block b_i predominates block b_j or vice versa. In addition it can be proved that there is one and only one immediate predominator of a given block $b_k\neq b_o$. For if there were two such blocks say b_i and b_j, we would have:

$$Dmin(b_i,b_k)=Dmin(b_j,b_k)$$

But this can only occur if b_i and b_j are in separate paths or $b_i\equiv b_j$. Since a back dominator is in every path it follows that b_i is equal to b_j. Obviously the set of back dominators of block b_j is completely ordered by the distance function Dmin. To compute the immediate predominator of a block, b_k, Lowry and Medlock (26) laid out some arbitrary non looping path from b_o to b_k. Then , starting from the end of the path, they removed every block which did not comply with the definition of the immediate predominator. The block remaining closest to b_k after repeatedly removing blocks in this way was the immediate predominator of b_k.

The set of all back dominators of block b_j can be readily obtained by the connectivity matrix of the graph.

To test if block b_i predominates block b_j we zero the ith line and ith column of C. Then we raise this modified boolean matrix to the nth power, call it C^*. If $c^*_{ij}=0$ block i predominates block j since there is no path $P=(b_o,\ldots b_j)$ without i.

2.6 Intervals

As a notational convention, if B is a set of basic blocks, S(B) will give the set of all blocks which are successors of blocks in B but which are not in B themselves:

$$S(B)=\left\{\bigcup_{b\in B} S(b)\right\}-B$$

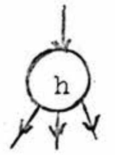
Similarly, the set of the predecessors of a set B is defined by:

$$P(B)=\left\{\bigcup_{b\in B} P(b)\right\}-B$$

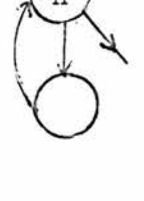
Given a control flow graph, an interval is defined as a set I of nodes of the graph with the following properties:

1. There is a node $h\in I$ called the head of the interval, such that $P(I)\subset P(h)$ and $P(I-\{h\})\subset\{h\}$ (i.e. the head is the only node in I with predecessors outside of I). This property forces I to be a single entry region.
2. For any $b\in I$ there exists a path connecting h and b.
3. $I-\{h\}$ is cycle free. All closed paths in I must include the head.

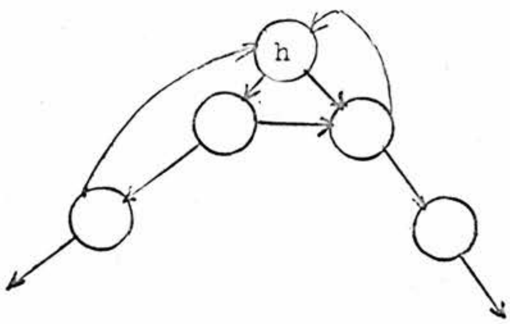
Let us now look at some examples of intervals. The simplest interval is a single block:



A simple loop forms an interval too:



A more complex interval might have the following figure:



Given node h, the maximum interval, MAX(h), whose head is node h can be defined by the following construction:

1. Set MAX(h)={h}

2. If there exists a block $b \in S(MAX(h))$ such that $P(b) \subset MAX(h)$ then set $MAX(h)=MAX(h) \cup \{b\}$ and repeat step 2. Otherwise continue with the next step.

3. If there is not such a block stop. MAX(h) is the required maximum interval.

MAX(h) is an interval. The validity of it can be proved very easily. First of all MAX(h) has only one possible entry node, h. For if there was another node $b \in MAX(h)$, $b \neq h$, which was also an entry node, b would have a predecessor outside the MAX(h) which is impossible since b became a member of the interval only when all of its predecessors became interval members. Hence the only entry node of the interval is the node h.

All closed paths in MAX(h) contain h. Suppose that there is a closed path $P=\{b_1, \ldots, b_k, b_1\}$ which does not contain h. Since only one node at a time can be added to MAX(h) we can assume that the first node of P added to MAX(h) was an arbitrary one, say $b_i \in P$. But since P is a closed path, b_i has a predecessor in P. In addition

b_i should be a successor of a node outside the closed path P. Therefore when b_i was becoming an interval member, b_i had, at least, two predecessors one in MAX(h) and another outside it: a contradiction. Hence P contains h.

Finally the header node of MAX(h) back dominates every node in it. Since, as we proved previously, the only possible entry to MAX(h) is through h, node h must lie in every path from b_o to any block in MAX(h).

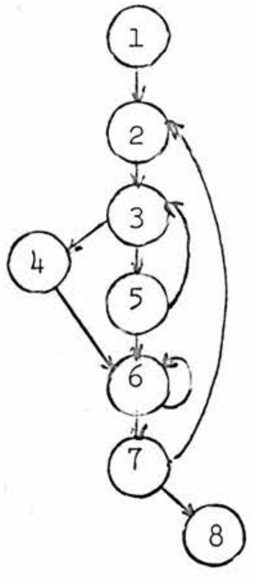
Consequently MAX(h) is an interval. It is also maximal. This follows from step 2 of the procedure since nodes are added to MAX(h) until no more can be.

A partition of a graph G is a set of subgraphs $g_1, g_2, \ldots, g_n$ such that $g_i \subset G, \cup_i g_i = G$, for all $i \neq j$, $g_i \cap g_j = \emptyset$.

By selecting the proper set of header nodes a graph may be partitioned into a unique set of intervals. The following algorithm, due to Allen and Cocke (21) produces the canonical intervalization of the control flow graph C.

1. Establish a list H for header nodes and initialize it to b_o.
2. Pick a block $b \in H$ and set $H=H-\{b\}$.
3. Find MAX(b) and add this to the list of intervals.
4. Set $H=H \cup S(MAX(b))$
5. If H is non-empty, go to step 2; otherwise stop.

Let us now illustrate the partitioning of a typical graph into intervals:



Intervals

I(1)=1

I(2)=2

I(3)=3,4,5

I(4)=6,7,8

We shall discuss now some interesting properties of the intervals generated by the previous procedure. First of all a <u>local successor function</u>, $LS(b_i)$, is defined for I(h):

$$LS(b_i)=\{b_j \mid b_j \in S(b_i) \text{ and } b_j \neq h\}$$

In addition a <u>forward path</u> in an interval is a path $P=\{b_1,\ldots,b_n\}$ where $b_{i+1} \in LS(b_i)$.

1. The nodes in an interval are partially ordered by the local successor function. Given an interval $I(h)=(b_i(=h),b_2,\ldots,b_n)$ if $i<j$ then either b_i and b_j lie in the same forward path or not. Clearly this follows from the construction of I(h).

2. Given an interval $I(h)=(b_i(=h),b_2,\ldots,b_n)$ and a back dominator list $BD(b_k)$ where $b_k \in I(h)$, the relative ordering of the nodes in $BD(b_k)$ and I(h) is the same.

3. If $b_k \in I(h), b_k \neq h$ and $BD(b_k)=\{b_o,\ldots,b_j\}$, then $h=b_r \in BD(b_k)$ and for each $r \leq t \leq k$, $b_t \in I(h)$. This is again a direct consequence of the way we produce the intervals.

4. If there is a strongly connected region, R, in the interval I(h) then $h \in R$, i.e. the strongly connected region contains the interval head. This follows from the fact that all closed paths in I(h) must

contain h. Another direct consequence of this property is that for each $b_i \in R$ and $b_j \in I(h)$ there exists a path, $P=\{b_i,\ldots,b_j\}$. Now let us suppose that the canonical intervalization of the control flow graph C produced the set of intervals G_1. We can define a successor relation S_1 on G_1 in a natural way. Interval I(h) is a successor of interval I(h') if control can transfer out of I(h') directly into I(h).

More formally:

$I(h) \in S_1(I(h'))$ iff $\exists b_i \in S(I(h'))$ such that $b_i \in I(h)$

If I(h) is a successor of I(h') it follows that there is a node, b, in I(h) which is a successor of a node in I(h'). Obviously $b \equiv h$. Using the successor relation, S_1, we can define a new control flow graph $C_1=(G_1,S_1)$ which is called the first derived graph. If we repeat the intervalization algorithm on C_1 we can get yet another derived graph C_2.

The repetitive application of this algorithm produces a sequence of derived graphs:

$$SEQ=\{C_1, C_2, \ldots, C_r, \ldots\}$$

If $NN(C_i)$ denotes the number of the nodes in the graph C_i it follows that the sequence:

$$\{NN(C_1), NN(C_2), \ldots,\}$$

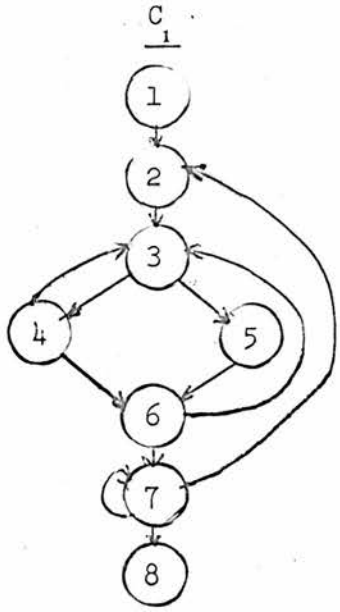
is monotone decreasing. In addition the sequence $NN(C_n)$ is bounded, Consequently there is the limit $\lim_{n\to\infty} C_n = L$. Moreover there exists $C_t \in SEQ$ such that $C_t \equiv L$.

If $NN(L)=1$ the initial graph C is called fully reducible.

If $NN(L)>1$ C is called irreducible.

We shall give one example for each case.

Consider the control flow graph:



The first application of the algorithm produces the following list of intervals:

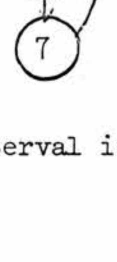
I(1)=1

I(2)=2

I(3)=3,4,5,6

I(4)=7,8.

Considering each of the above intervals as a node, C_1 can be reduced to:



where every multinode interval is replaced by the number of its head.

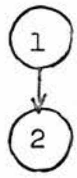
Similarly C_2 produces the list:

I(1)=1

I(2)=2,3,7

and the graph:

$\underline{C_3}$



A final application of the algorithm derives the list.

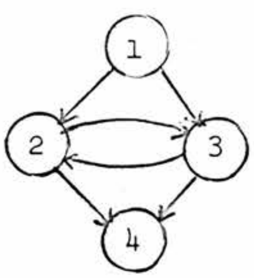
I(1)=1

and the graph:

Hence C_1 is fully reducible.

Let us now consider the control flow graph:

$\underline{C}$



and its corresponding list of intervals:

I(1)=1

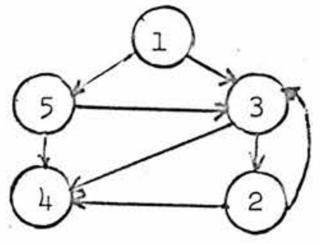
I(2)=2

I(3)=3

I(4)=4

Clearly this graph can not be reduced because of the cycle (2,3,2). The trouble is that both constituents of this cycle are immediate successors of the head. By duplicating node 2 we can obtain a graph C' where node 2 is not an immediate successor of the head, i.e.

C'



Node5≡Node 2

Obviously the graph C' represents a program which is equivalent to that represented by the previous graph. Moreover repetitive application of the algorithm reduces it to an one node graph:

I(1)=1,5

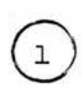
I(2)=3,2

I(3)=4

C_2'

I(1)=1,3,4

C_3'



The method described above is known as node splitting and it can transform every irreducible graph to a reducible one. However, studies have shown that approximately 90% of all FORTRAN programs are reducible.

Let us now prove that the set of intervals

$$J=\{I(h_1),I(h_2),...\}$$

generated by the procedure forms a unique partition of any graph G.

First of all we shall prove that J covers G. Suppose that J does not cover G or equivalently there is $b\epsilon G$ which is not an interval member in any $I(h)\epsilon J$. Since G is a connected directed graph, b must either be b_o (the initial node) or have, at least, one predecessor.

If $b=b_o$ then $b\epsilon I(b_o)$, the first interval constructed. If $b\neq b_o$, then, node b would have at least one immediate predecessor, b'. But node b' would not be in any I(h) because if it did then b would become a member of the same interval or the head of another interval. Applying the same reasoning recursively we shall find that b_o does not exist in any I(h), a contradiction. Hence J covers G.

The elements of J are disjoint, that is for any I(h) and I(h') in J, $I(h)\cap I(h')=\emptyset$. Primarily $h\neq h'$, because every node in the list H of the algorithm can appear only once. In addition a node in H can be processed only once. If $h\neq h'$ but $h\epsilon I(h')$ then all immediate predecessors of h must be in I(h'). But since h is an interval head it follows that all if its immediate predecessors must belong to an interval I(h''). Consider now an immediate predecessor of h, say b, with $b\epsilon I(h'')\cap I(h')$. Clearly node b is back dominated by both h' and h'' and since the back dominators of a block are strictly ordered either h' predominates h'' or vice versa. But since I(h'') and I(h) form different intervals then h'' can not back dominate h. Thus h'' can not back dominate h'. Consequently h' back dominates h'' and thus $h''\epsilon I(h')$. Similarly we can prove that there exist $I(h''')$ such that $h''' \epsilon I(h')$ with h' back dominating h'''. Proceeding inductively some h will become eventually equal to h': a contradiction. Let us now suppose that there exists $b \epsilon I(h) \cap I(h')$. Clearly node b is not a head. Therefore a number of nodes in the list of back dominators of b must belong to $I(h) \cap I(h')$ and thus an interval header will belong to $I(h) \cap I(h')$: a contradiction.

Hence J forms a partition of G. Furthermore it can be proved that the list J is unique.

CHAPTER 3

OPTIMIZING TRANSFORMATIONS

3.1 Introduction

Program optimization refers to the process of transforming a given program, A, to an equivalent program, B, whose execution time is expected to be less than that of A.

Since the program equivalency problem is recursively unsolvable (30) it is quite clear that no general theory producing a completely optimum program can be developed. However, there are some ad hoc techniques for performing a partial optimization. Most of these techniques apply iteratively a certain set of transformations to improve the execution time of a given program. Generally, some attention is paid to execution space but no attention is paid to optimizing compile time or to the more general questions of total job or system optimization.

Not all optimizing transformations will result always in an improvement to a program. The correctness of some transformations is another problem. It would be very desirable to prove that the application of any optimizing transformation on a given program, does not affect the results of that program. This problem has not been solved completely yet.

A number of optimizing transformations will be presented in this chapter. Transformations involved with efficient subroutine linkage are presented first. Transformations which are most conveniently applied on the source language level are presented next, followed by the machine independent optimizations which can be performed on any machine and the machine dependent optimizations whose application is dependent on the hardware of a specific computer.

3.2 Procedure Integration

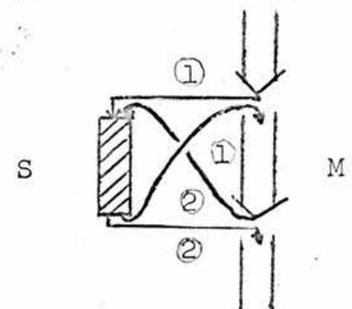
Subprograms are extremely useful structural components of programs. A subprogram is a block of code which is executed as a unit and which performs some part of the total processing of the program. Despite their convenience, subprograms do bring with them considerable overheads in both space and execution time, especially when they are not used cautiously. In addition, most optimizing compilers are unable to combine subprograms in a single compilation. A direct consequence of this is that during optimization a compiler always assumes the worst case (e.g. All variables in common are set and used in all subroutines etc.).

What is desired, therefore, is to provide more global program units for optimization by merging a carefully selected set of subprograms into their calling routines.

If subprogram M calls subprogram S the following linkages may be established:

1. Closed. This is a standard linkage. There is only one copy of S in storage and every time it is called control passes to it. On exit, control passes back to the locus subsequent to the point of call in M.

This linkage can be illustrated by the following figure:

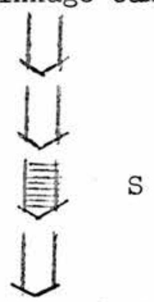


For instance with specific reference to FORTRAN, routine S is known to the system loader but no information about S is available during the compilation of M. The closed subroutine linkage is inefficient because the contents of the machine registers must be saved on entry and restored on exit. In addition, specific registers must be allocated to specific functions.

This may cause delays at execution time on closed sequential calls in pipelined CPU's. Moreover, the parameters and global variables used (or set) in S are not known during the compilation of M. During the compilation of M it is assumed that all such parameters are set and used in S. The first uses of these parameters after the CALL statement cause main storage to be accessed. Another inefficiency of the closed subroutine linkage is that in most cases, all information must be passed in storage and that no intersubprogram optimization can be performed.

2. Open. An open linkage is not really a linkage at all, but more like a macro expansion. The code body of S is inserted into the calling program's code at each point that the procedure identifier is met. Its parameters are replaced, of course, by the corresponding arguments on the call statement. Thus the program S is completely integrated in M and is not known to the system.

The open subroutine linkage can be depicted as follows:



There are many advantages of this kind of linkage. The most important are that there is no linkage overhead and that both M and S can be optimized together. (Constant arguments can be folded, invariant instructions can be moved etc.). Clearly, therefore, an opening of every closed subroutine by an optimizer would be very desirable. But let us now consider a few disadvantages of this suggestion. First of all, the size of the program M might become extremely large. In addition, irreducible subprograms require special handling. (An irreducible subprogram is one which has a history, contains I/O operations, can return different function values for identical argument values or it has multiple return points.) Furthermore, the required space during compilation of M is increased since a high-level version of S should be always available and, finally, an update of S would obsolete all object modules into which it had been merged, necessitating recompilation of all of them.

3. Semi-open. A semi-open linkage is a non-standard linkage in which routines S and M are compiled together. In this case S and M become part of the same object module. If during optimization it is recognized that actual parameter locations are not needed, sub-programs M and S may become indistinguishable by replacing all CALL's to S by suitable branching statements to a location internal to the module.

The main advantage of this form of linkage is that M and S can be optimized together as a unit. In addition, the branching is much faster than a standard linkage.

The semi-open form of linkage has almost the same disadvantages as the open form.

4. Semi-closed. The semi-closed linkage is a non-standard linkage in which routines M and S are compiled as separate modules. The called routine, S, is compiled first. During the compilation of S the linkage registers and the parameter passing conventions are determined to be used during the compilation of M when S is called. The other information about S which could be collected during its compilation might include names of global variables set or used in S and the names of registers whose contents are altered by S.

The advantages of this form of linkage are the following: the compilation of S is not constrained by fixed register assignment at entry and exit points. Registers unused in S need not be saved during the linkage time. (The unused registers can be active in M during this period.) Finally, since M knows the way that global variables are used in S, it can avoid unnecessary memory references during CALL's of S and it can also carry information in the registers.

Obviously an update of S obsoletes M and this counts as a disadvantage of this kind of linkage. Many compilers in the market expand all reducible subprograms in-line; for example, the mathematical functions SIN, COS, SQRT and EXP, the implicit functions FLOAT and IFIX and some manipulative functions such as AMAXO, AMINO and MOD are all opened up by the compiler. However, there is no commercially available compiler performing procedure integration. The difficulty lies in the organization of all existing compilers. It is well known that if subprograms S_1 and S_2 are submitted together they will be compiled separately by the compiler. In order to perform the optimization discussed in this section, the compiler has to have access to both subprograms at the same time. In addition, some standard criteria for determining the type of linkage which should be performed must be established. Clearly, subroutines which are called only once or whose size is very small can be opened up by the compiler. Beyond these simple considerations the general problem has not been solved yet.

3.3 Loop Transformations

Since a large proportion of the program's time is usually spent in loops, special attention is given to them. There are several transformations which can be applied to program loops. The transformations presented in this section are usually performed on a language level which is close to the source language.

Three types of loop transformations will be considered here: Loop Unrolling, Loop Fusion and Unswitching.

1. Loop Unrolling Every iteration of a loop requires incrementation and testing of the loop variable. This overhead may be reduced at the expense of additional instructions by the technique called loop unrolling. We say that loop A is completely unrolled to the set of statements B if the successive computations implied by A appear sequentially in B. For example, the following Do-loop:

```
      DO  1  I=1,4,1
  1   A(I)=I
```

becomes when unrolled:

```
A(1)=1
A(2)=2
A(3)=3
A(4)=4
```

But the same Do-loop can be partially unrolled to produce the following Do-loop:

```
      DO  1  I=1,4,2
      A(I)=I
    1 A(I+1)=I+1
```

In this case, we say that the initial loop is unrolled by 2. Obviously a DO-loop can be unrolled by any number n provided that the number n is between the initial value and the test value of the Do-variable. Clearly the reduction in the execution time is proportional to n. In addition, by unrolling a loop more instructions are exposed for parallel execution.

The major disadvantage of loop unrolling is the increase of the required instruction space. For this reason, before unrolling a loop its size and relative frequency should be considered. Other factors should be the available space and the form of the loop itself. For example, if the parameters of the loop are variables, loop unrolling does not seem very promising because a few additional tests must be inserted for end conditions. Consider for example the following DO-loop.

```
      DO  1 I=J,K,L
    1 A(I)=B(I)+C(I)
```

which should become when unrolled by 2:

```
      L1=2*L
      DO 1 I=J,K,L1
      A(I)=B(I)+C(I)
      IF(I+L1.GT.K) GOTO2
    1 A(I+L1)= B(I+L1)+C(I+L1)
    2
```

2. Jamming or Loop Fusion. In this transformation the ranges of two loops are combined to form the range of a single one. Consider the following example:

```
          :
      DO 1 I=1,100
    1 A(I)=0
      DO  2J=1,100
    2 B(J)=0
          :
```

which becomes after jamming:

```
          :
      DO 10 I=1,100
      A(I)=0
   10 B(I)=0
          :
```

Clearly the loop overhead and code space are reduced. In addition more instructions are exposed for parallel execution and for local optimization. Generally two loops will be fused if they satisfy the following criteria:

(a) When one loop is executed the other one is also.

(b) They are independent; that is, the computations in either loop do not depend upon the computations of the other.

(c) Their ranges are executed the same number of times. The generation of code for the end conditions can eliminate this requirement.

This transformation is particularly important for some mathematical languages which have array or vector operations. (e.g. APL).

3. Unswitching. This transformation is the opposite of jamming. If a loop contains an invariant test, the loop may be replaced by two loops with that test executed outside and selecting which of the two loops will be executed.

Consider the following example:

```
      DO 1  I=1,100
      IF(K.GT.9)GOTO2
      A(I)=B(I)+C(I)
      GO TO 1
    2 A(I)= B(I) - C(I)
    1 CONTINUE
```

which becomes:

```
      IF(K.GT.9) GOTO 2
      DO 1 I=1,100
    1 A(I)=B(I)+C(I)
      GOTO 4
    2 DO 3 I=1,100
    3 A(I)=B(I)-C(I)
    4    .
         .
         .
```

Clearly execution time is reduced but more instruction space is required.

The machine independent and language independent transformations are considered next. They are called machine independent because their application to a problem program will cause it to run faster on many different types of machines.

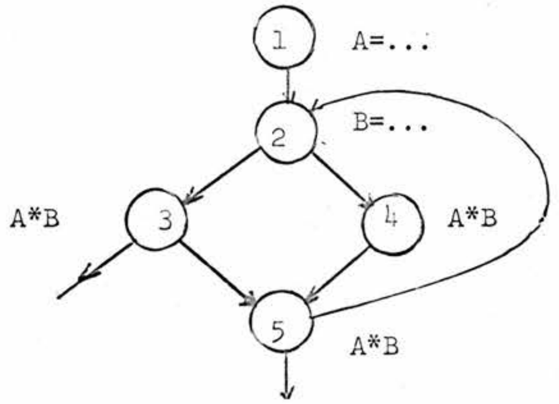
They are also called language independent because they are applicable to a variety of high level languages.

3.4 Machine Independent Transformations

In the following sections, the control flow relationships between the basic blocks in a program will be expressed by means of directed graphs.

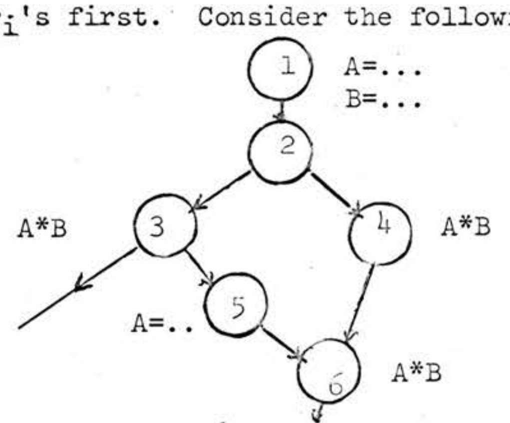
3.4.1 Redundant Subexpression Elimination

This transformation detects and eliminates those computations whose values are already available. In the following example the instruction A*B in block 5 is redundant and can be eliminated.



The computation A*B is redundant in block 5 because there is an identical computation in all paths leading to block 5.

It probably would be desirable to require that every path from a program entry block to a given computation should contain at least one definition of each variable operand of the computation. (A variable can be defined explicitly by an assignment statement or implicitly as a subroutine parameter, as a variable in COMMON etc.) Obviously such a requirement would not be necessary since the existance of a path does not necessarily mean that it will be traversed at execution time. However, if an instruction r is redundant because of the instructions r_1, r_2,..., then it is essential to require that there does not exist a flow path from a definition of any of r's operands to r which does not go through one of the r_i's first. Consider the following example:



The computation in block 6 is not redundant because of the definition of A in block 5.

There are two forms of analysis for redundant subexpression elimination according to the way they identify redundant subexpressions. The first form of analysis depends upon the existance of identical instructions. The other form is based upon a value number algorithm. This algorithm does not depend upon explicit formal identities for the identification of a redundant calculation. Consider for example the following program segment where all variables used are of real type.

```
X = A+B
Y = A
R = Y+B
```

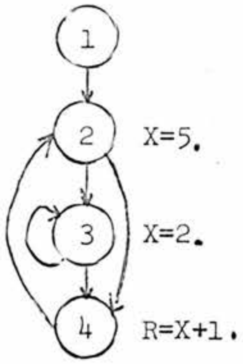
Clearly Y+B is not formally identical with A+B but computes the same value so it is redundant. Such redundancies can only be found by the value number method. But in some cases, existing versions of the value number algorithm would have failed to recognize redundancies which could be readily detected as formal identities. The major advantages of this transformation is that execution time is reduced and code space is saved. The disadvantage is that register usage is increased.

Some of the developed algorithms for redundant subexpression elimination perform this operation on a block basis, that is only these redundant subexpressions occurring in the same block are eliminated. These algorithms are very realistic in terms of time and they can become quite efficient if they are combined with a suitable code moving algorithm. In this case instructions from many different blocks may end up in the same block and, if they are redundant, be eliminated.

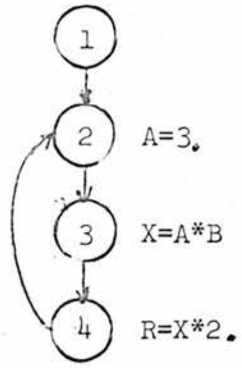
3.4.2 Code Motion

Code motion refers to the process of moving suitable instructions from frequently executed areas of the program to less frequently executed areas. An instruction can be moved if its movement neither interferes in an existing definition-use link nor severs a link between the instruction and a use of its result.

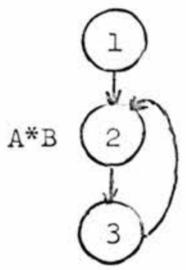
Consider the following examples:



and

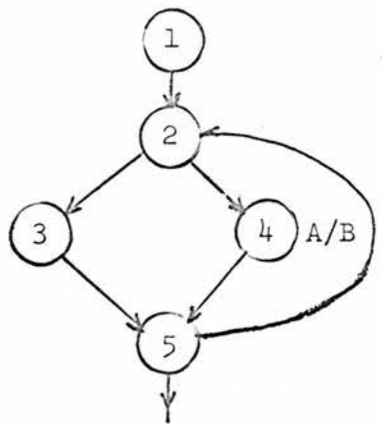


In the first example the assignment X=2. can not be moved anywhere. Similarly the instruction A*B in the second example can not be moved because of the definition of A in the second block. An additional requirement for the movability of an instruction is that such a movement be "safe". An instruction can be safely moved if its movement does not cause side effects to occur which would not have occured if the instruction had remained in its original position. Consider the following example:



The subexpression A*B can be safely moved into node 1, provided that neither A nor B are defined in nodes 2 and 3. This movement is safe because it can not create any side effects by itself. Of course, if the sub-expression A*B caused an overflow in the original program then the same overflow would occur after the movement in block 1. Clearly the number of times that this possible overflow might occur is probably altered since blocks 2 and 3 are generally more frequently executed than block 1, but this is not so important if the optimized program is to run under a system which takes some kind of action in such cases.

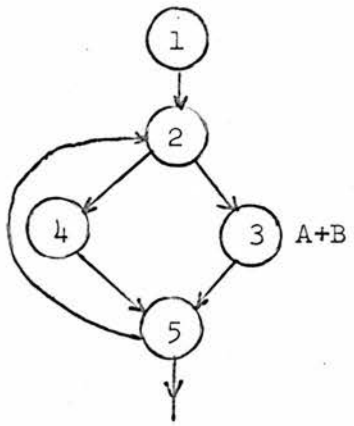
Consider now the following example:



The subexpression A/B can not be safely moved into block 1 because a divide check error could possibly occur which would not occur in the program as given. (The variable B could be set to zero in block 1 and the path {1,2,4,5} might never be traversed during execution time.)

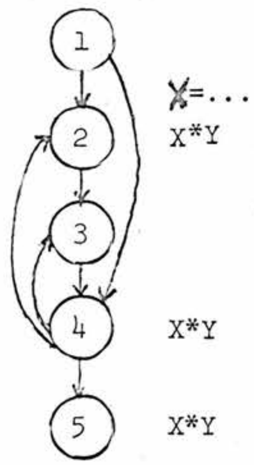
So far the necessary requirements to preserve the correctness of this transformation have been stated. Clearly the profit of a movement should be also considered. But it is not always easy to determine whether a movement results in an improvement or not because the relative execution frequencies of various parts of a program are not always available.

Consider the following example:

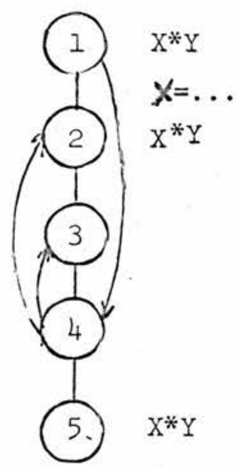


The subexpression A+B can be correctly moved into block 1 but it is not apparent that this improves the initial program since the path 2-3-5 might never be traversed during execution time.

Code motion can be very profitably combined with redundant subexpression elimination. Code which seems unmovable by pure code motion techniques can be completely eliminated by use of certain techniques and the redundant subexpression elimination algorithm. A few optimizing procedures <u>insert</u> code at some privileged parts of the program in order to expose more redundancies for optimization. Consider the following example:



Clearly the computation X*Y in block 4 considered as belonging to the region 3-4 can be correctly moved into the predecessor blocks of the region 2 and 1. If this action were taken the previous graph would become:



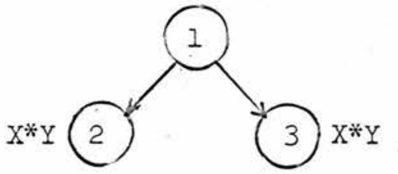
Let us now assume that the value of the variable X on entry to the block 1 causes an overflow to the computation X*Y but the value assigned to X in block 2 does not. In this case we would have an overflow condition in the derived graph. But this side effect would not occur in the initial graph if the flow went from block 1 to 2 and not to 4. Hence the computation X*Y in block 4 can not be safely moved. In addition the computation X*Y in block 5 can not profitably be moved anywhere.

However, if an X*Y were placed in block 1, then both of them could be eliminated. The primary advantage of the code motion transformation is the reduction of the number of instructions executed. The disadvantage is that register usage is increased.

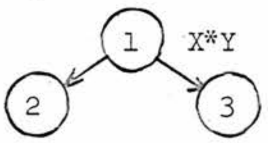
3.4.3. Hoisting

This transformation moves instructions which can not be further moved by the pure code motion transformation discussed in the previous section because they either can not be examined by it as not belonging to the range of a loop or do not conform with the requirements for code motion stated before.

More specifically, in this transformation instructions are moved as close to the entry block as possible, with the hope that more redundancies will be exposed for redundant subexpression elimination. Consider the following example:



which becomes after hoisting:

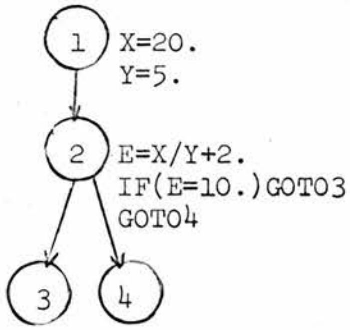


Clearly hoisting does not necessarily reduce the number of instructions executed but it does save instruction space.

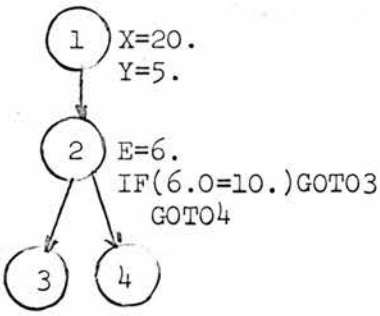
3.4.4 Constant Folding

Constant folding is the process of replacing uses of variables by their given constant value and of performing operations with constant operands at compile time. (Other terms for this process are constant propagation and sumsuption.)

Consider the following example:



which would become after folding:



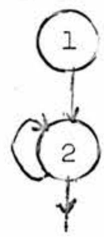
Constant folding has many advantages but no disadvantages. It is particularly important when it is combined with other optimizing transformations. Since arguments to subprograms are very frequently constants,constant folding should be applied after the procedure integration transformation discussed earlier in this chapter. When constants replace the parameters of the subprogram, many transformations can be made to it. Furthermore, folding has been proved very profitable when it is applied after the code motion transformation.

3.4.5 Dead Code Elimination

An instruction is considered dead when it can not be executed because either it is in an area of the program which can not be reached or its result is never used.

The existance of dead code in a program is not always a result of programmer error or carelessness. The strength reduction of code (see next section) involving recursively defined variables and the replacement of tests which depend on these variables frequently expose recursive definitions which are unused anywhere except in the recursive computation. In addition, the elimination of dead recursive definitions and their uses, allows, very frequently, other non-recursive definitions of the same variables to be eliminated.

Consider the following example:




```
I=1

T1=I*K1
A=B(T1)

I=I+K2
IF(I≤K5)GOTO2
```

which would become after strength reduction and test replacement:


```
I=1
I2=K1*I
K3=K1*K2
K6=K1*K5

T1=I2
A=B(T1)
I=I+K2
I2=I2+K3
IF(I2≤K6)GOTO2
```

Assuming that variable I is not used anywhere in block 2, clearly, the assignment statement I=I+K2 is dead and it can be eliminated.

3.4.6. Strength Reduction

The strength reduction optimization reduces certain computations using recursively defined variables to recursive definitions.

A variable, J, is recursively defined if its definition is a function of J. The recursive definitions important to optimization have the form: J=J+cst. Where cst. is a signed constant. Similarly, uses of a recursively defined variable, J, which are of some interest in optimization, are:

1. J*cst. and 2. J±cst. or cst±J

Consider that a strongly connected region contains n recursive definitions of a recursive variable J:

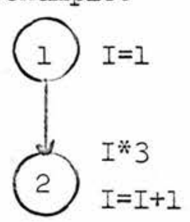
J=J+cst.(1), J=J+cst.(2), ..., J=J+cst.(n)

Then the use J*cst. can be reduced to a recursive definition by the following procedure.

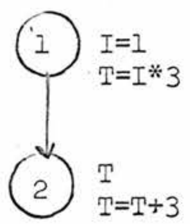
(a) A new variable, T, is introduced and set equal to the value of the expression J*cst. This definition is inserted in all entries of the region.

(b) Each recursive definition J=J+cst.(K) is paired by the T=T+C, where C=cst.(K)*cst.

Consider the following example:



which becomes after strength reduction:

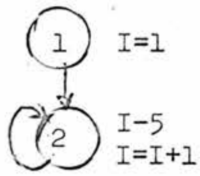


Similarly, the use J±cst. or cst.±J can be reduced to a recursive definition by:

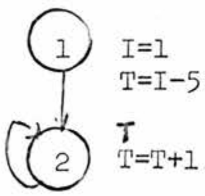
(a) Putting the definition T=J±cst. or T=cst.±J on all entries of the region; and

(b) Putting T=T+cst.(K) or T=T±cst.(K) with J=J+cst.(K)

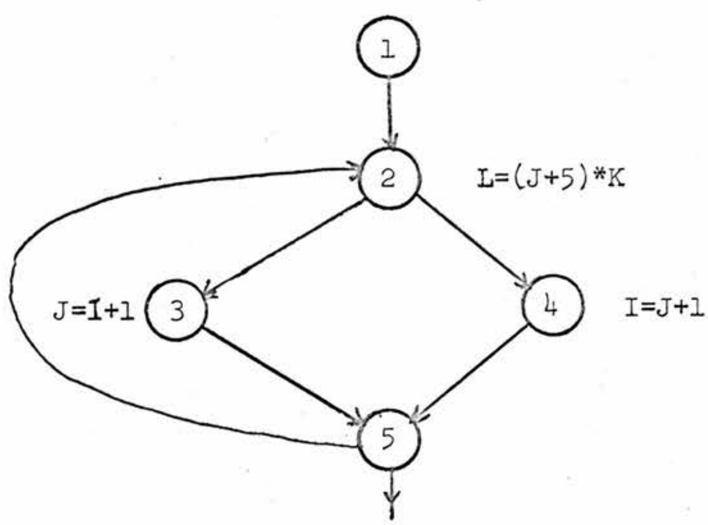
Consider the example:



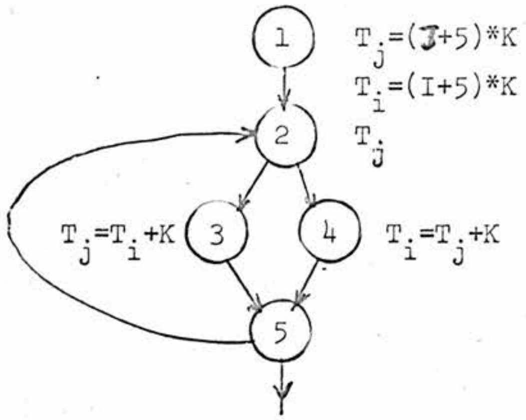
which becomes:



It now becomes obvious that the intent of these transformations is the replacement of subscript calculations involving the DO loop induction variable by index register increments. A generalization of these transformations is exemplified by the following:



which becomes:

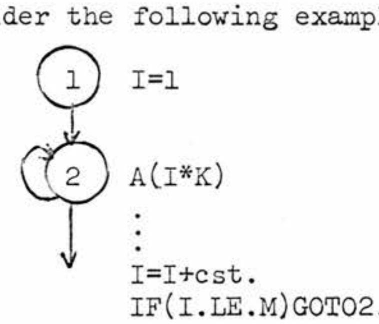


Where the variables I and J were recursively defined in terms of each other.

The advantage of this transformation is that faster computations are used. (The new recursively defined variables may end up in index registers and be updated by register increments.)

3.4.7 Test Replacement

After the application of the strength reduction optimization and the introduction of new recursively defined variables, it very frequently appears that the only use of the original recursively defined variable is in a test controlling a loop. If this test can be replaced by another one which depends upon a new recursively defined variable, then the initialization and incrementation of the original recursive variable might become dead. Consider the following example:



which after strength reduction would become:

```
(1)  I=1
     T=I*K
     K1=cst.*K

(2)  A(T)
     I=I+cst.
     T=T+K1
     IF(I.LE.M)GOTO2
```

Clearly the last test in the derived program could be replaced by the IF(T.LE.M*K)GOTO2 making immediately the statement I=I+cst. dead. In addition, the application of the constant folding transformation makes the first assignment statement dead. Finally after a few other minor modifications, the derived program would become:

```
(1)  T=K
     K1=cst.*K

(2)  A(T)
     .
     .
     .
     T=T+K1
     IF(T.LE.K1)GOTO2
```

3.5 Machine Dependent Transformations

3.5.1 Instruction Ordering

In this transformation, reordering of instructions within each block is performed to maximize the opportunities for parallel execution. This optimization is best used of course when the computer has pipelined units. Consider the following example:

```
R1=A+B
R2=R1-C
R3=R2+D
E=R3
R4=X/Y
F=R4
```

In this case the instructions are not conveniently ordered for parallel execution since during the execution of the slow division instruction, no other instruction remains in order to be executed simultaneously with it. The above segment after the reordering of its instructions becomes:

```
R4=X/Y
R1=A+B
R2=R1-C
R3=R2+D
E=R3
F=R4
```

which might almost halve execution time.

This transformation reduces execution time at the expense of register usage.

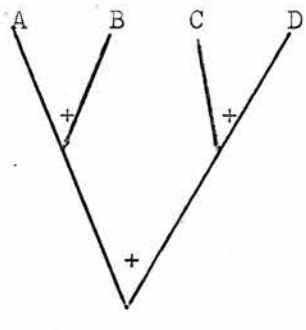
3.5.2 The Minimum Depth Parse

Several parsing methods have been developed to improve the quality of the generated code. Of special interest in optimization is the minimum depth parse. The intent of this method is to minimise instruction dependencies. For example the expression:

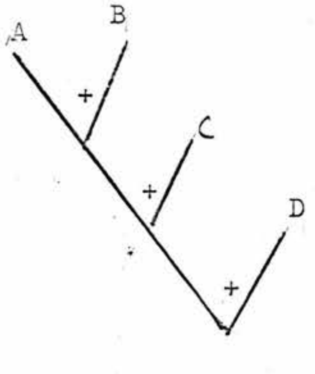
A+B+C+D

would be parsed as if it had been written as (A+B)+(C+D) rather than (((A+B)+C)+D).

Depicted in tree form, this is:



instead of



The major advantages of the minimum depth parse are that:

(a) more instructions are generated for parallel computation in computers with pipelined units, and

(b) more independent instructions are exposed for global optimization. In the above example the sub-expression C+D is exposed as independent of A and B and therefore it can be moved and transformed independently.

The disadvantage of this method is that register usage is extended.

3.5.3 Register Allocation

Effective register allocation is very important in optimization especially in computers where many registers are available.

Register allocation is generally separated from register assignment. The allocation of a register, R, for a data item, i, implies the decision that i is to reside in some virtual register R without specifying which real one. On the other hand register assignment determines which of the actual registers will be used for each allocated register.

The procedure for allocating or assigning registers can be local or global. The former process ignores the control flow of the program and it is therefore simpler than the latter one which considers the program as a whole. An optimizing allocation normally consists of both local and global allocation.

3.6 Some Miscellaneous Optimizations

3.6.1 Anchor Pointing

The intent of this optimization is to minimize the number of logical tests performed in a Boolean expression before branching.
For example the statement:

```
IF(A.OR.B.OR.C) COTOIO
```

should be broken down into the following equivalent set of statements

```
IF(A)GO TO 10
IF(B)GO TO 10
IF(C)GO TO 10
```

Clearly the ordering of the generated statements should not be at random. It should be chosen so that control leaves the sequence of IF's as soon as possible. In the above example, it was assumed that the possibility of truth was decreasing from condition A to C.

3.62 Special case code generation

In some special situations a great amount of execution time can be saved by a few very local considerations. Since no control flow analysis is required for them, they can be implemented in every compiler without serious degradation in compiler time. On the contrary, experience has shown that a compiler with some optimization will often run faster than one without any, due to the shorter object program it produces.

A list of some of these special transformations follows:

(a) Elimination of unnecessary operations such as I*1, I+0, I*0 etc. Although these situations occur very rarely in the source program, they often arise from subscript evaluations.

(b) Conversion of a floating point division by a constant to a faster multiplication. Specifically the division A/constant should be changed to A*(1/constant) <u>if</u> it is safe. Clearly, this transformation is safe if (1/constant)*constant=1.0.

(c) Expansion of X**n where 'n' is an integer constant into in-line code.

(d) Simplification of logical expressions using DeMorgan's theorem etc., etc.

3.6.3 Peephole Optimization

In this optimization, which is also called window optimization, the final code from the compiler is examined through a window of, for example, 10 instructions for possible transformations.

CHAPTER 4

THE IMPLEMENTATION OVERVIEW

An algorithm developed for optimizing the computation of arithmetic expressions of a FORTRAN program is described in this chapter. The objectives of the processing are (1) to eliminate redundant subexpressions, and (2) to move invariant subexpressions out of frequently executed areas of a program. In order to perform these optimizing transformations, the algorithm analyses the control flow of the source program. Most of the ideas used for the development of the flow analyser derive from two classical papers on this subject by Prosser (30) and Allen (3).

The procedure to be described can be divided into three stages. In stage one, the FORTRAN input program is transformed into an intermediate text convenient for optimization. In stage two program flow analysis and optimization are performed and finally, in stage three, the resultant intermediate text is translated into FORTRAN again. The intermediate text is an integral part of the procedure, so a discussion on its structure before the description of the algorithm seems worthwhile.

4.1 The Intermediate Text

What kind of internal form is most advantageous for optimization? A direct answer to this question is not easy and there are not enough published details of optimizing compilers for comparisons and conclusions to be drawn. However, triples, indirect triples and quadruples are mentioned frequently in the literature. All of these internal structures represent basic operations. Triples generally have the form:

ARG1 ARG2 OP

where ARG1 and ARG2 are the operands and OP is the operator associated with them. The major advantage of the triple representation is the flexibility of the structure. For example, the individual elements of a program segment can be easily reordered. However, optimization requires quite a large amount of sorting and rearranging of triples. The same flexibility but greater ease of handling (from the optimization point of view) derives from the use of indirect triples. They are also coded in triple form but their linear order is given by means of a table whose entries point to them. Clearly, it is faster to manipulate pointers to triples than the triples themselves. Therefore, indirect triples are adopted for internal representation in this thesis. Further advantages of this internal form will become apparent during the description of the optimizing algorithms later on. The last candidates, quadruples, do not seem so convenient for our purposes because they always need a description of each temporary value. Indirect triples, however, do cause some processing problems, as we shall see.

The intermediate text used in this thesis has, therefore, the following constituents:

1. An <u>Instruction Table</u>, INSTR, containing the unique instructions in the program. Each instruction is coded as a triple. The first two entries hold the arguments and the last holds the operator. Each triple has the general form:

A1 A2 OP

where A1 and A2 are pointing either to other triples or to the symbol table, which is described in this section. In addition, OP points to the operator table which is a fixed size table where all possible operators are kept in character form. If A1, or A2, is greater than 1000 it points implicitly to the result of the triple numbered A1-1000(or A2-1000).

If A1, or A2, is greater than 100 and less than 1000 it points to the entry numbered A1-100, or A2-100, of the operand table. The value of OP is always less than 25. If the operator is not binary (unary minus, logical.NOT.) the first entry of the triple is zero. Obviously, the numbers 100 and 1000, mentioned above, could be changed. They are simply used for discrimination between operators, operands and references to other triples. The instruction table is used merely as a "pool" of instruction types. The actual sequence of these instructions is given in the sequence table in the form of pointers to the individual types.

2. The <u>sequence table</u> defines the linear order of the instructions in a block. Every entry in the sequence table points to a triple in the instruction table. The beginning of every block is marked with a zero entry. The use of this entry will be discussed later.

The sequence table saves storage and time; it saves storage because common triples can occupy one single entry in the instruction table and it saves time because it facilitates instruction manipulation.

3. The <u>symbol table</u>, mentioned above, contains programmer variables, constants and, after optimization, generated names, all in character form.

4. The <u>flow table</u> defines the basic blocks in the program and their limits in the sequence table. For example, the fourth entry of this table points to the beginning of the fourth block in the sequence table. Moreover, the fifth entry points to the end of the fourth block and to the beginning of the fifth.

Clearly, the limits of each block of a program can be accessed directly through the flow table.

5. The <u>statement number table</u> contains the statement number associated with the first statement of every block.

6. Some subsidiary tables are kept for purposes to be discussed later.

The following example shows in symbolic form the contents of the tables described so far, from the translation of the FORTRAN statements:

```
10   A=E(I)+B
     C=E(I)+C
     R=A**C
```

where E is the name of an array of arbitrary size.

Entries	Instruction table	Sequence table	Flow table	Stat.num.table
1	E I ↓	0	1	10
2	(1) B +	1		
3	A (2) =	2		
4	(1) C +	3		
5	C (4) =	1		
6	A C ↑	4		
7	R (6) =	5		
8		6		
9		7		

As shown in the above example, subscript calculations do not appear in the intermediate text. Although subscript calculations provide many opportunities for improvement to an object program, they were ignored because of the level of the generated code. So the array reference A(I,J) would generate the following instructions:

```
I J ,
A (1) ↓
```

Defining operations are those which assign a value to a variable (e.g. I=5, K=(1) etc.). In an "explicit" definition like a FORTRAN assignment statement, the first argument of the generated instruction which indicates the assignment is not always a simple variable. For example, the assignment statement A(I)=5 would generate the following instructions:

```
A I ↓
(1) 5 =
```

This is the only case where the first argument of an assignment instruction is a reference to another instruction. Whenever an array element is defined, the optimizer assumes that the definition applies to the entire array. Consider the following straight-line code:

```
C=A(I)+1
A(1)= ...
D=A(I)+1
```

In this example the subexpression A(I)+1 in the third statement can not be eliminated as redundant because of the definition of the first element of the array A.

There are some other ways of course to define a variable. For example the statement READ(J,1)A assigns a value to the variable A. A subroutine CALL or the use of a function subprogram are also considered as definitions of all actual parameters. In addition, a DO statement or an I/O operation containing an implied DO-loop are considered definitions of the counter. (The term definition is used in a rather loose way here, since the counter is officially undefined after exit of the loop). During the translation of the source program into the intermediate text each time a variable, X, is defined by a statement which is not an assignment statement, a new pseudo-instruction

```
X, 0, 25
```

is initiated. Thus, each definition is given explicitly in the instruction table at the point where it occurs.

Generally, every statement except an assignment statement causes the initiation of at least one instruction of the following form:

```
INF , PTR , CDN
```

where CDN is a code number for identification, PTR is a pointer to a work table where the source language version of the whole statement (or part of it) resides and INF holds useful information for the code generator.

4.2 Construction of the Intermediate Text

In this section, the translation from the source program to the intermediate text is described. The translation process is performed, conveniently, in two passes over each input statement.

During the first pass the input statement is recognized and some general information is collected in order to be used later on.

In the second pass, the input statement is translated, according to its identification number and relevant information collected so far, into the intermediate text.

During these two passes the source program is also segmented into basic blocks.

4.2.1 Pass one

Each statement of the source program is read (in A1 FORMAT) and processed individually.

Unquoted blanks embedded in the input string are eliminated first.

Next, the resultant string is examined by the Recognizer. Since most of the key words in FORTRAN are not reserved, it is not safe to identify the input statement through them. For example, the following are all legal FORTRAN assignment statements:

```
READ(5,1) = 10
DO 1 J= T
IF(I) = N
```

For this reason a given statement must be tested first to see if it is an assignment statement. If this test is negative, then the FORTRAN statement may be recognized by its initial characters. More specifically, the recognizer tests first the source statement for a zero level equal sign, that is an equal sign not enclosed by apostrophes or parentheses. If this test is positive, the source statement may be an assignment or DO or IF(xxx)X=Y statement. A DO statement is recognized by the presence of a zero level comma. An IF(xxx)X=Y statement is recognized by the presence of a right parenthesis followed immediately by an alphameric character. If the statement is neither a DO nor an IF(xxx)X=Y statement, it will be an assignment statement.

If no zero level '=' is found then the source statement is recognized by its initial characters. Two characters, stored in a nx2 array C, from every key word are enough for this purpose. Each C(K,1) holds the first character of the Kth key word. Similarly, the C(K,2) corresponds to the f(K)th character of the same key word. Where f is a simple mapping function. If the recognized statement is a logical IF statement, then part of the procedure recurs for the recognition of the dependent statement.

Logical and arithmetic IF statements are also examined by the recognizer for optimization suitability. Since our main objective is the optimization of arithmetic expressions, we consider an IF statement as optimizable <u>iff</u> there is at least one arithmetic operator in the arithmetic or logical expression of the IF.

The statement number test follows. If there is a statement number between columns 1 and 5 of the source statement, then the current block is closed and a new one is initiated which is considered as a successor of the previous one. In addition, the entry of the statement number table corresponding to the new block is initialised to the numeric value of the statement number. Finally, the same number is compared with the last entry of the DO-stack. The <u>DO-stack</u> is an Nx2 array which holds in the first column the statement number of the last statement of the range of a DO-loop. The second column accommodates the number of the block where the DO-statement was encountered. If a matching between the statement number found and the (M,1) entry of the DO-stack occurs, the current block is flagged in order to be closed after the processing of the current statement, considering as successors the succeeding block and the block whose number is held in DO(M,2). This process recurs until either the DO-stack is empty or there is no matching.

Consider the following example:

```
   DO  1  I=1,100        |DO  1  I=1,100|      block no. n
   DO  1  J=1,100        |DO  1  J=1,100|      block no. n+1
          :                     :  (maybe more than one block
          :                     :    in this area)
 1  CONTINUE
          :            1 |CONTINUE      |      block no. n+K
                                :
                                :
```

When the analysis reaches the statement numbered 1 in the example, the last two entries of the DO-stack will be:

1 , n

1 , n+1

<u>4.2.2 Pass Two</u>

During this pass the source statement is translated into the intermediate text. Generally, the way a statement is treated depends upon its ID number. Description of the action taken in each case follows:

1. Arithmetic IF Statements. (ID=1).

If the arithmetic expression of the IF is optimizable, then it must be transformed into triples. All operators and all operands of the statement are replaced first by numbers pointing to appropriate tables. In FORTRAN, because of the relatively poor character set used, some characters are used in many different syntactic positions. The analyser converts the different uses of the same character into distinct characters. For example, the parentheses used for enclosing subscript expressions are kept in a different entry in the operator table from the parentheses used for grouping arithmetic expressions; the binary operator substruct (-) is also separated from the unary operator negate, etc.

As stated above, two tables are used by the analyser for the accommodation of the basic items of an arithmetic (or logical) expression: the operand and the operator table. The operand table was described in the section 4.1. The operator table is a fixed 24-entry table holding all operators used. The operator table with the corresponding dilimiter hierarchy table are given below. (19)

Entry Number	Hierarchy Number	Operator	USE
1	0	(	Grouping
2	0	<	Beginning of Statement
3	0	↓	Subscript operator
4	1	)	Grouping
5	1	>	End of Statement
6	2	.OR.	Logical operator or
7	3	.AND.	" " AND
8	4	.NOT.	" " NOT
9	5	.EQ.	Relational operator: equal to
10	5	.NE.	" " not equal to
11	5	=	Assignment operator
12	5	.LT.	Relational operator less than
13	5	.LE.	" " less than or equal to
14	5	.GT.	" " greater than
15	5	.GE.	" " greater than or eq. to
16	6	,	Subscript separator.
17	6	)	Enclose subscripts
18	7	-	Arith. operator minus
19	7	+	" plus
20	8	*	" multiply
21	8	/	" divide
22	8	~	" negation
23	9	**	" exponentiation
24	10	(	Enclose subscripts

The conversion process can be exemplified by the following example:

The FORTRAN statement: X=(-Y+Z(I,J)**5)*X becomes:

101,11,1,22,102,19,103,24,104,16,105,17,23,106,4,20,101

where each pointer to the operand table:

Entries	Names
1	X
2	Y
3	Z
4	I
5	J
6	5

is incremented by the constant 100. Each number which is less than 100 is pointing to the operator table. During the above conversion, identical names (the variable X in the example) are represented by identical pointers. Next, the resultant string is translated to early operator reverse Polish form. The usual stack compilation techniques are used here. As we shall see later, identical subexpressions are detected during the translation process and they are expressed as such in the intermediate language. In order to generate formal identities for equivalent statements, one must take advantage of the commutativity of some operations. For example, the operations X+Y and Y+X although equivalent are not formal identities and therefore they can not be detected as redundant. An ordering of the X and Y in lexicographic order before the generation of the intermediate text solves the problem. To do this systematically, we should order all the operands of an n-element addition, (or a n n-element multiplication). Primarily, an algorithm for this general sorting was developed but it turned out very time consuming. Therefore a simpler sorting, performed with a part of the original algorithm, was adopted. Thus, not all equivalent subexpressions are recognized as redundant. Consider the following statements:

X=A+B+C

Y=C+A+B

In this case no formal identities will be exposed because they will become after sorting:

X=A+B+C

Y=A+C+B

However, a complete sorting does not solve the whole problem because there are some cases where a complete sorting would destroy existing redundancies. For example, the statement:

X=A+B+A+B

would become after sorting:

X=A+A+B+B

Finally the resultant Polish string is transformed into triples. Although the generation of the triples is straightforward, their placing in the instruction table is quite interesting.

A reasonable suggestion might be to start searching the instruction table, every time a new triple was generated, for a match with the new triple. If the match occured, the first available entry of the sequence table would accommodate the pointer to the matched entry in the instruction table. Otherwise, the new triple would be added to the instruction table and its associated pointer to the sequence table. Thus, the statements:

X=A+B

Y=A+B

would generate the code:

Entries	Instruction Table	Sequence Table
1	A B +	1
2	X(1)=	2
3	Y(1)=	1
4		3

Clearly, formal identities would be exposed in the sequence table as desired. Assuming that the data dependencies problem was solved, in some way, say by assigning a level number to each pointer in the sequence table, the way would be open for redundant subexpression elimination. But, the application of the same method to the statements:

```
X=A+B
Y=A+B+C
A=5
R=A+B+C
```

would generate the code:

Entries	Instruction Table	Sequence Table
1	A B +	1
2	X(1)=	2
3	(1)C +	1
4	Y(3)=	3
5	A(5)=	4
6	R(3)=	5
7		1
8		3
9		6

Apparently, the instruction A B + would be found redundant in the second assignment statement but not in the fourth one because of the presence of the definition of A. The generation of a new variable, say T1, to be assigned the result of the redundant instruction A B + would necessitate the replacement of every reference to this instruction by T1. Thus, the third instruction in the instruction table would become T1 C +. But this instruction belongs to the second and the fourth assignment statements. Consequently, the elimination of the redundant instruction A B + in the first two statements would cause an uncontrolled elimination of the same instruction everywhere in the block. For this reason, whenever a new instruction is to be added to the instruction table, the translator scans the instruction table from its end for a matching with the new instruction. It stops scanning when a definition of one of the two operands of the new instruction is encountered. Applying this algorithm to the statements of the previous example we have:

Entries	Instruction Table	Sequence Table
1	A B +	1
2	X(1)=	2
3	(1)C+	1
4	Y(3)=	3
5	A 5 =	4
6	A B +	5
7	(6)C+	6
8	R(7)=	7
9		8

Although the required instruction space is increased, it is much easier to perform redundant subexpression elimination under this scheme since identical pointers in the sequence table do not only imply that the corresponding instructions are the same but in addition, that neither of their operands are defined between them. Consequently, redundant subexpressions can be found and eliminated safely in the sequence table without any further analysis.

Uses and especially definitions of array elements need some more attention. Specifically because array elements generate at least one instruction they should be protected from elimination when they are defined. Consider the following examples:

```
C(I)=C(I)+1     and     X=C(I)
                          :
                        C(I)=5
```

In neither case will the instruction which defines C(I) appear in the sequence table between the occurrencies of the instruction C I ↓ necessitating therefore a look at the Polish string. A detailed description of this algorithm will be given in the next section. After processing the arithmetic expression of the IF, the translator informs the predecessor-successor table that the blocks, whose labels appear as the transfer addresses in the arithmetic IF statement are to be considered as successors of the current block. The predecessor-successor table holds the immediate successors of each block in the program. Because this table should hold block numbers and not source statement numbers the negated values of the three transfer addresses are inserted where the corresponding block numbers should be. At the end of the translation process, when the relationships between block numbers and statement numbers will be given in the statement number table, a fairly simple lock up will replace the negative statement numbers by their corresponding block numbers.

Next the string of three numbers representing the statement numbers is stored (in character form) in a work table and the instruction:

1 PTR 31 (a),

is added to the instruction table. The number 1 in (a) implies that the arithmetic expression has been expanded into triples, PTR points to the beginning of the stored string in the work table and 31 is the ID number of the statement increased by 30 in order to avoid confusion with the pointers of the normal operators whose range is from 1 to 24.

If the arithmetic statement is not suitable for optimization, it will not be expanded into triples but it will be stored directly into the work table (in character form). Every other action mentioned above recurs except that the value of the first entry of (a) is now 0 and not 1.

2. <u>GO TO Statements</u> (ID=2).

Primarily, the GO TO statement is stored in the work table and a new instruction is initiated to indicate the presence of the GO TO and the place in the work table where it is stored. Next, the current block is closed and the negative of the transfer address is passed to the predecessor-successor table.

3. <u>Logical IF Statements</u> (ID=3).

The logical expression is treated in the same manner as the arithmetic expression of the arithmetic IF statements.

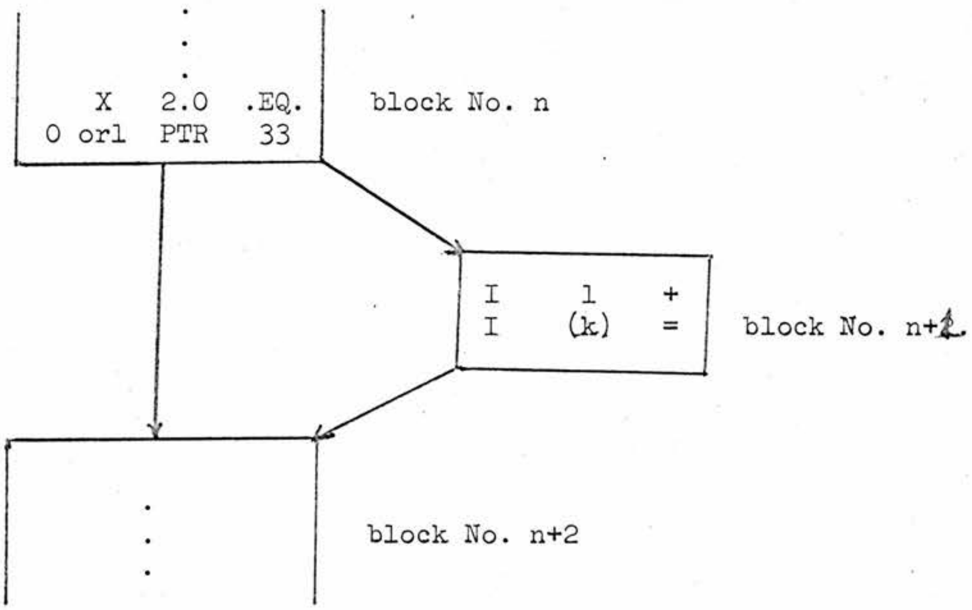
A logical IF statement generally produces more than one block. So after the insertion of the usual ID instruction the current block, say n, is closed and the two consecutive blocks n+1 and n+2, following it are considered as successory to it.

The block n+1 will accommodate the dependent expression of the IF statement. The dependent expression will be processed separately but after its processing, the block n+1 will be closed considering the next block, n+2, as a successor to it, except if the dependent statement is a control statement where block n+2 is not considered as a successor of block n+1.

For example the statement:

IF(X.EQ.2.0)I=I+1

generates the code:



4. <u>Computed GO TO statements</u> (ID=4).

A Computed GO TO statement may cause more successors to the current block than a simple GO TO statement but this is actually the only difference between them.

5. <u>Subroutine CALLS and READ statements</u>

They do not cause any change in the program flow. The reason they are treated separately is that they may define variables.

A routine developed for this purpose, detects the defined variables and generates an instruction for each of the form:

VAR 0 25

where VAR is the defined variable as entered in the operand table and 25 is the code number indicating a definition.

6. DO statements

The presence of a DO statement causes the initiation of a new block. Next, the usual ID triple is inserted in the instruction table. In addition, the definition of the counter causes, through the routine mentioned in the CALL-READ case, the initiation of a definition instruction where the first entry holds the counter. Finally, DO information passes to the DO-stack.

7. Assignment statements

An assignment statement is entirely translated into triples. The relevant procedure is the same as the one described in the translation of the arithmetic expression of an arithmetic IF statement.

8. All other statements cause just the initiation of an ID triple.

After the processing of all statements of the source program the negative numbers in the predecessor successor table are replaced, through the statement number table, by their corresponding block numbers.

4.3 Optimization Process

Machine independent and language independent methods of improving the execution time performance of the source program are implemented.

In this section the implemented optimizing transformations and several general schemes for performing them are discussed first, followed by the description of their implementation.

4.3.1 The Implemented Transformations

Two optimizing transformations are performed by the optimizer: redundant subexpression elimination and code motion.

Redundant subexpressions occur quite frequently in problem programs for two reasons:

1. The natural expression of a problem in a high level language

frequently involves redundant subexpressions. For example, to find the roots of a quadratic equation, one might more transparently write:

```
R1=(-B+SQRT(B**2-4*A*C))/(2*A)
R2=(-B-SQRT(B**2-4*A*C))/(2*A)
```

than the equivalent, more efficient, but more obscure:

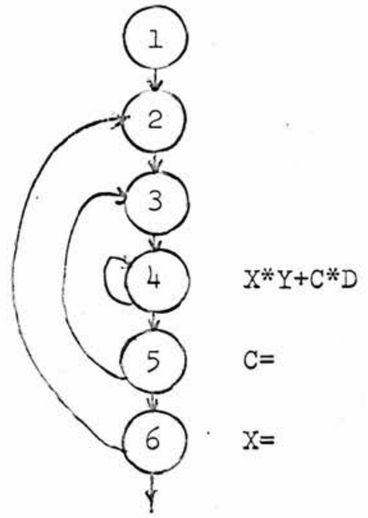
```
X=2*A
Y=SQRT(B**2-4*A*C)
R1=(-B+Y)/X
R2=(-B-Y)/X
```

2. Hidden subscript computations quite frequently involve redundant instructions, e.g. A(I,J)=B(I,J)+C(I,J). These redundancies are clearly beyond the programmer's control. Unfortunately, they are also beyond our control because the output is handled interpretively by run time procedures. Clearly, the best optimization procedures should be embedded in the compiler to handle such hidden computations. However, as we shall see later, a use of an array element generates at least one instruction and as such it may be eliminated as redundant.

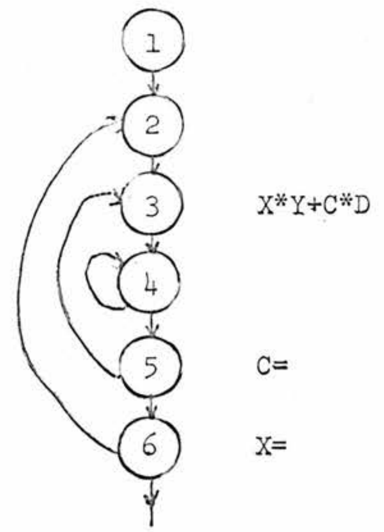
Invariant instructions occur very often in programming too. On the one hand, because of the programmer's carelessness and on the other hand because they add clarity to the program. As stated in section 3.2, these transformations require information concerning both flow of control and data interference. In addition, since the goal of the code motion transformation is to move instructions from frequently executed areas to less frequently executed areas, a structure determining the relative execution frequency of a program area should be established. Clearly, such a structure could correspond to the loop, since it may be assumed that loops are themselves frequently executed areas and that inner loops are, in general, executed more frequently than outer loops.

There are two important properties which should be satisfied by a loop-like construct. First, there must be a reasonable likelihood that code within the loop will be executed more than once, if the loop is entered at all. Second, given two loops, they must not overlap, that is they must be either disjoint or the one be contained in the other.

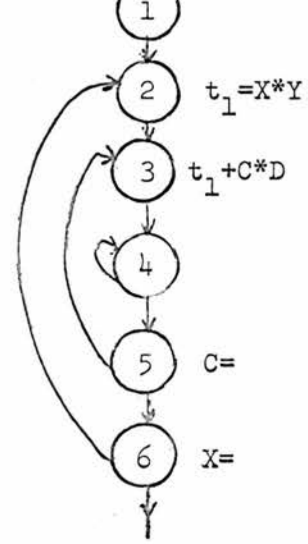
Clearly, loops satisfying the above criteria can be ordered, and processed, in an inner-to-outer basis. In this ordering the most deeply nested loops are processed first and as many instructions as possible are moved out of them to the next outer loops, where they may be considered again as candidates for removal. The same process applies to the next outer loop and so on. Thus, instructions may be moved as far as possible out of the frequently executed areas. For example, consider the following control flow graph.



According to the general optimization plan stated before, the code motion transformation applies to the fourth block first:



Next, blocks 3-4-5 are examined for movable instructions:



And finally, the outer loop 2-3-4-5-6 is processed without any change in the last graph. Loop determination can be accomplished in a variety of ways. A simple method for example is to consider only the explicit source language loop construction. (e.g. DO, for etc.). These loops have, obviously, the required properties. In addition, the single initialization block of such a loop can be used conveniently to accommodate the instructions which will be removed from the loop body. When the source language iteration construction is used as a basis of loop determination only the instructions contained in these loops can be moved.

More sophisticated techniques exist, however, with various other types of loop constructs. In some cases the loop construct is chosen so that a single predecessor block exists in order to receive the movable code of the loop. Although it is not absolutely necessary that a single block be established as a back target, it is obviously very desirable. In loop constructs, like the strongly connected regions, where more than one predecessors of a given loop construct may exist, it is quite possible to copy the movable code in each predecessor of the loop. This method may, however, increase considerably the size of the program and is usually avoided. Another alternative to this problem is to optimize only the regions with a single entry point. The same strategy is adopted in this implementation.

Another interesting loop construct is the interval. The interval (2.7) is defined in such a way as to guarantee the uniqueness of the back target.

In this implementation, the notion of the strongly connected region is used as the basic structuring device.

The general optimization plan is:

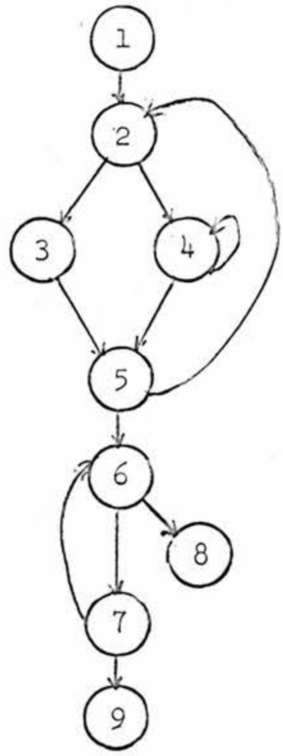
(a) Every block in the program is examined for redundant instructions.

(b) Every strongly connected region, in term, is examined for invariant instructions. If any invariant instructions are moved, the predecessor block which received the code is examined again for redundant instructions.

4.3.2 The Flow Analyser

The flow analyser uses the information collected as described above in order to produce the list of strongly connected regions. If the source program has been partitioned into n basic blocks the region list is developed as follows:

1. A nxn Boolean connectivity matrix, C, is constructed. If j is a successor of i then $C_{ij}=1$; otherwise $C_{ij}=0$. Consider the directed graph:



The connectivity matrix of this graph would be:

	1	2	3	4	5	6	7	8	9
1	0	1	0	0	0	0	0	0	0
2	0	0	1	1	0	0	0	0	0
3	0	0	0	0	1	0	0	0	0
4	0	0	0	1	1	0	0	0	0
5	0	1	0	0	0	1	0	0	0
6	0	0	0	0	0	0	1	1	0
7	0	0	0	0	0	1	0	0	1
8	0	0	0	0	0	0	0	0	0
9	0	0	0	0	0	0	0	0	0

2. A list, UCP, of unique closed paths of the source program is now developed. Each entry on the list is a Boolean vector expressing the blocks which are on the same closed path. So $UCP_{ij}=1$ if block j is on a closed path

which happens to be the ith entry of the UCP list. The real ordering between the closed paths in UCP is kept in another table, LST, whose entries point to the entries of the UCP list. So it is easy and fast to reject one path or to exchange two paths on the UCP list. The UCP list is developed as follows:

(a) If $C_{ii}=1$ then the block i is an immediate successor of itself. An entry is made on the UCP list. For the above example the UCP list would now have one entry:

	1	2	3	4	5	6	7	8	9
UCP_1:	0	0	0	1	0	0	0	0	0

(b) The connectivity matrix is raised to successive powers 1 ($1 \geq L \leq n$). Obviously the C^{L+1} will be constructed by the C^L according to the formula:

$$(C_{ij})^{L+1} = \bigvee_{k=1}^{n} (C_{ik} \wedge (C_{kj})^L)$$

So C^{L+1} can be obtained by the following simple method:

(b0) The C^{L+1} matrix is initialized to zero.

(b1) We set i=1.

(b2) $\forall_j \in [1,n] : C_{ij}=1$ the $C_i^{L+1}=C_i^{L+1} \vee C_j^{L}$ is formed.

(b3) i=i+1. If $i \leq n$ go to step b2; otherwise stop.

Apparently, the access of rows in C^L and C^{L+1} is direct. The ORing between them is fast too. In addition, the number of operations involved increases linearly with the number of branches in the program. If $(C_{ii})^L=1$ then the basic block i belongs to a closed path of length L. But this closed path is not necessarily unique; for example if $(C_{ii})^L=1$ then $(C_{ii})^{L+L}=1$ too, because $C^{L+L}=C^L \wedge C^L$ and therefore $(C_{ii})^{L+L}=\bigvee_{k=1}^{n} ((C_{ik})^L \wedge (Cki)^L)$ and for k=i, $(C_{ii})^{L+L}=1$. It is apparent now that $(C_{ii})^L=1$ implies that $(C_{ii})^r=1 \ \forall \ r=mxL$ where $m \in [2,n/L]$, $m \in N$.

In addition, if block j is in the same closed path with i then $(C_{jj})^L=1$ too.

(c) In order to separate out the closed paths imbedded in a given C^L, an integer distance matrix, D, is kept. D_{ij} holds the length of the shortest path from block i to block j. Whenever $(C_{ij})^L=1$ and $D_{ij}=0$, D_{ij} is set to L.

(d) Whenever $(C_{ii})^L=1$ and $\mathrm{mod}(L/D_{ii}) \neq 0$ a candidate closed path, P, is constructed by :

(1) setting $P_i=1$ and

(2) $\forall\ j\neq i\ (1\leq j\leq n)$ if $D_{ij}\neq 0$ and $D_{ji}\neq 0$ and $D_{ij}+D_{ji}\leq L$, $P_j=1$.

(e) P is added to the list if it is unique. Finally, the resultant closed paths are sorted and the list of the strongly connected regions is developed.

4.3.3 Eliminating Redundant Subexpressions

As we said in section 4.2.1, formal identities are recognized during the development of the intermediate text. Whenever a new instruction, NEW, is to be inserted on the instruction table, INSTR, a search is made, through the Sequence Table (SEQ), in the previous instructions of the current block for a matching instruction. The current block is scanned backward; if a matching occurs, between NEW and some INSTR(SEQ(K)), the first empty entry in SEQ receives the pointer to the matched instruction in INSTR and the search terminates. If a definition of one of the operands in NEW or the beginning of the block is encountered, the instruction NEW is entered in the first available entry in INSTR and its pointer is assigned to a similar entry in the table SEQ.

An instruction INSTR(SEQ(I)) is a definition if its operator is equal to 11 or 25. The defined item is given, of course, by the first entry of the INSTR(SEQ(I)). If this entry is a variable, it is compared directly with the operands of NEW. If the first entry of INSTR(SEQ(I)) is a reference to the result of another instruction then the name of the defined array (as entered in the operand table) is given by the entry:

INSTR(INSTR(SEQ(I)),1)

For example, consider the assignment statement:

R(I1,I2,I3)=6

and its equivalent code, (in symbolic form):

(1) I1 I2 ,

(2) (1) I3 ,

(3) R (2) ↓

(4) (3) 6 =

where the name of the defined array is given in the entry: INSTR(INSTR(4,1),1). There is no need, of course, to find out the name of the defined array if the operator in NEW is not the "take" operator :'↓'. But if it is so, the defined array name must be found and compared with the operands of NEW.

In the actual implementation of the above algorithm some other special cases are also considered. For example, as we mentioned before in this chapter, there are cases involved with the definition of an array element where the defining instructions:

(K) 6 =

does not always appear between the Kth instruction and another identical (to the Kth) instruction which constructs the same array element for later use. For example, the recursive definition:

C(I)=C(I)+10

should generate the code:

Entries	Instruction Table	Sequence Table
1	C I ↓	1
2	C I ↓	2
3	(2) 10 +	3
4	(1) (3) =	4

where no definition appear between the first and the second instruction.

During redundant subexpression elimination new variables must be introduced in order to replace the discovered redundant subexpressions. However, care must be taken to introduce as few new variables as possible. For example the statements:

```
X=(A+B+C+D)*E
Y=(A+B+C+D)*F
```

should become:

```
T1=A+B+C+D
X=T1*E
Y=T1*F
```

instead of:

```
T1=A+B
T2=T1+C
T3=T2+D
X=T3*E
Y=£3*F
```

The generation of new variables is controlled by the following algorithm.

(0) Initialize the pointer, PTR, of a stack, TRCTN, to zero. In addition, assume that variables I1,I2 are pointing to the beginning and end of the current block respectively and set I=I1-1.

(1) Set I=I+1, if I<I2 continue with the next step; otherwise stop.

(2) If the instruction INSTR(SEQ(I)) is not referencing the results of other instructions (e.g. A B +) set STATUS=1 and go to step 6; otherwise continue with the next step.

(3) If one and only one entry of INSTR(SEQ(I)) points to another instruction (e.g. (1) B*) go to step 4; otherwise go to step 5.

(4) If PTR≠ 0 and TRCTN(PTR) is also pointing to the referenced instruction set STATUS=2 and go to step 6; otherwise go to step 8.

(5) Apparently, both operands of the instruction INSTR(SEQ(I)) are pointing to other instructions, e.g. (1) (2) +. If PTR≠0 and TRCTN(PTR), TRCTN(PTR-1) are pointing to the first and second referenced instruction respectively set STATUS=3 and continue with the next step; otherwise go to step 8.

(6) If the instruction INSTR(SEQ(I)) is not identical with at least one of the instructions following it in the current block go to step 8; otherwise continue with the next step.

(7) According to the status number perform one of the following operations; after that go to step 1.

(a) If STATUS=1 set TRCTN(PTR+1)=SEQ(I);

(b) If STATUS=2 set TRCTN(PTR)=SEQ(I);

(c) If STATUS=3 set TRCTN(PTR-1)=SEQ(I).

(8) If PTR=0 go to step 1; otherwise generate a total of q (q=value of PTR) new variables, Tn, ne[1,PTR], and insert q new assignments of the form:

TK TRCTN(K) =

In addition, replace every reference to the instructions pointed at by the entries of TRCTN by the corresponding new variables, set PTR=0 and go to step 1.

Multiple dimensional array references need some special attention during this process.

The optimizer treats the subscript separator as a break - character, that is every time an instruction containing it is encountered, control is transferred directly to the step 8. This action guarantees that commas will be always "protected" by take operators (↓). Consequently, the assignment statements:

```
X=A(I+J,K**2,L)
Y=A(I+J,K**2,N)
```

become:

```
T1=I+J
T2=K**2
X=A(T1,T2,L)
Y=A(T1,T2,N)
```

But this action creates a new problem. As it is apparent from the description of the previous algorithm, when the stack TRCTN is empty the procedure starts searching for redundancies from the first independent instruction.

Consider the following statements:

```
X=R(A,B)+1
Y=R(A,B)+1
```

and their equivalent internal representation:

Entries	Instruction Table	Sequence Table
1	A B ,	1
2	R (1) ↓	2
3	(2) 1 +	3
4	X (3) =	4
5	Y (3) =	1
6		2
7		3
8		5

The first instruction contains a breaking character and since the stack TRCTN is empty we proceed to the second which depends on the first, therefore we go to the third etc.

Clearly, we failed to recognize the expression R(A,B)+1 as redundant because of the presence of the array element R(A,B). A simple solution to this problem would be to accept the "take" operator as a candidate unconditionally. Furthermore, we could reject array elements with arithmetic expressions as subscripts. The presence of an arithmetic expression as a subscript in a multiple dimensional array reference indicates that the arithmetic expression was not found redundant when examined (because if it were, it would be replaced by a new variable) which implies, in turn, that the entire array element can not be redundant. So a simple test could save us from a useless scanning. For example, consider the array element:

A(I1,I2,I3,I4)

and its internal form:

```
I1   I2   ,
(1)  I3   ,
(2)  I4   ,
 A   (3)  ↓
```

Clearly, when the array element has single variables, or constants, as subscripts the sytax reflects it: all referenced instructions have a comma as an operator and the first instruction is the only independent one. In addition, dependencies, after the "take" operation, occur only in the first operand. Every violation of this syntax permits us to skip the entire array element.

Besides the subscript separator, all logical operators, all relational operators and the equal sign, are also considered as break - characters.

In addition, the sixth step in the previously described algorithm is executed on condition that the number of the identical instructions per instruction remains constant. Every fluctuation of this number causes the creation of a new entry on the stack TRCTN. Consider the following example:

```
X=A+B+C+D
Y=A+B+F
Z=A+B+C+G
```

Apparently, the instruction A B + is redundant in the second and the third statement but the instruction (1) C + is redundant only, in the third statement. This causes the following output:

```
I1=A+B
T2=T1+C
X=T2+D
Y=T1+F
Z=T2+G
```

Clearly, the algorithm described in this section does not recognize the occurrence of redundant instructions in different blocks. Only those redundant instructions occurring in the same block are eliminated. This does not mean that the instructions had to be in the same block initially. As a result of moving code to less frequently executed program segments, instructions from many different blocks may end up in the same block and, if they are redundant, be eliminated.

4.3.4 Moving Invariant Subexpressions

This transformation is performed on a region basis. Each strongly connected region is examined for movable instructions and, if there are any, they are moved into the predecessor block of the region. (Only those regions with a unique predecessor block which has in term one single successor are processed).

More specifically, assuming that regions:

$$R(1),R(2),\ldots,R(K-1)$$

have already been optimized, region R(K) is selected from the region list.

An instruction of a block, b, in R(K) is movable if its variable operands are not defined in R(K). So if block b has already been examined as belonging to a region $R(n) \subseteq R(K)$, it should not be examined again since an unmovable instruction in R(n) is also unmovable in R(K). Therefore, considering the regions:

$$R(1),R(2),\ldots,R(K)$$

as Boolean vectors in the sense that if block n belongs to the region R(I) then the nth entry of the vector R(I) will be 1; otherwise it will be 0, the blocks to be examined on behalf of R(K) are given by the following expression:

$$R(K)\ \left(\bigwedge_{j=1}^{k-1} \overline{R(J)} \right)$$

During the construction of the sequence table the optimizer left an initial zero-value entry per block. After the processing of the corresponding block the value of this heading entry becomes one. Consequently before examining block b for removal of invariant instructions, a simple test of the heading entry informs us if b has been examined for the same purpose before or not. (The value of the heading entry of b is given directly by:

SEQ(FLOW(L))).

The only global information collected so far is in the form of a table of Boolean vectors. Table DEF holds the definition vectors for each programmer's variable; these were constructed when the redundant instructions were eliminated. DEF(I,J)=1 if variable I ($I \in \{1,2,\ldots\}$), as entered to the symbol table, is defined in block J, otherwise it is zero.

A new pseudo block is initiated into which the code moved backwards out of R(K) is collected and which after optimization of R(K) will be appended to the end of the predecessor block of R(K). To determine the movability of an instruction in R(K) we ask if each variable operand of the instruction is defined in R(K) or not. From every other aspect the procedure is almost the same as the elimination of redundant instructions described earlier. Each invariant instruction is placed, of course, on the pseudo block and the definition of the generated variable is placed there too.

Multiple dimensional arrays caused problems again. As we said before, a basic feature of the intermediate text is that every referenced instruction in a block, L, must belong to L. Therefore when we find an invariant instruction which is forming an element of a multiple dimensional array we must know the addresses of all instructions dependent (directly or indirectly) on it, in order to place them first in the pseudo block.

After all blocks in R(K) have been examined, then, if the pseudo block is not empty it is appended to the end of the predecessor block of R(K). Next the predecessor block is examined for redundant instruction elimination and so the process continues until all blocks have been dealt with.

4.3.5 Code Generation

At this stage the intermediate text is translated into FORTRAN statements.

Initially the specification statements are printed out. Each active entry on the specification table points to the first character of the stored string in the work table. The end of the string is recognized by the number 82 ehich is not the EBCDIC form of any character. The statement is copied character by character into a buffer, and upon recognition of the last character of the string the contents of the buffer are printed out.

The generation of the executable part of the program is performed in a block basis. All basic blocks in the program are processed in order of their ID number. During this analysis constructed parts of statements pass, in character form, into the buffer.When the contents of the buffer form a complete statement they are printed out. Obviously, the number of the FORTRAN statements generated by the processing of one block may vary from a relatively high value to zero. It could be zero since, as stated in section 4.2, a logical IF statement is generally accommodated in two consecutive blocks.

Before printing each constructed statement we must find out if there is any statement number to be printed in the first five columns of the line or not. The definition of a basic block implies that one and only one statement, in a given block, can be labelled; the very first. Therefore, if the current statement is the first one in its block and if this block has been labelled initially by the programmer, then the current statement will be printed out with the label in front of it; otherwise no label will be printed. Consider for example the following program segment:

```
              .
              .
              .
1         X=A+B*C
          Y=D+B*C          block K
              .
              .
              .
```

and its optimized version:

```
                .
                .
                .
  1       T1=B*C
          X=A+T1
          Y=D+T1          block K
                .
                .
                .
```

But consider also the following Do-loop:

```
                .
                .
                .
        DO 1 I=J,100             block K-1

  1     A(I)=I**2+(I+1)/I**2     block K
```

Obviously, if we had labelled the optimized code generated from the above DO-loop in the same manner we would have produced the following output:

```
        DO  1  I=J,100        block K-1

  1     T1=I**2
        A(I)=T1+(I+1)/T1      block K
```

Clearly, the labelling of the block K is in error. This problem was solved by discriminating the statement numbers which define the end of the range of a DO-loop from all others.

FORTRAN statements are classified according to the action required for their generation into three main categories:

(1) The statements which are entirely stored in the work table. (e.g. READ, WRITE, CALL, etc.).

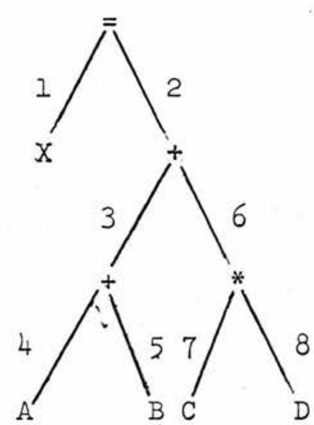
(2) The statements which are entirely translated into triples (e.g. Assignment statements).

(3) All others. (Logical and arithmetic IF statements).

The first type of statement did not pose any problems. They are simply fetched from the work table into the buffer and printed out. But the generation of arithmetic and logical expressions is more interesting. Consider the assignment statement:

X=A+B+C*D

and its corresponding tree structure:



Where each node carries an operator and two pointers (left and right) which point either to terminal identifiers or "lower" nodes of the tree. The above tree structure suggests the following algorithm:

(0) Associate each node with a counter, CNTR and set all counters initially to zero.

(1) Start scanning the tree structure from its root obeying the following rules:

(a) Each time a node is reached increment its corresponding counter by 1.

If CNTR is equal to 1 then follow its left pointer.

If CNTR is equal to 2 pass the operator situated in the

node to the buffer and follow its right pointer.
If CNTR is equal to 3 return to the higher level node which is pointing to the current node.

(b) If a leaf of the tree is met pass the corresponding operand to the buffer and return to the current node.

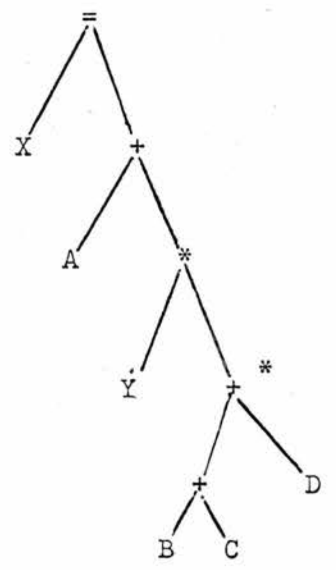
(c) If the counter of the root becomes equal to 3 we are done.

A pictorial representation of this algorithm is given below. (For easy reference, each pointer in the previous tree structure has been associated with a unique number).

Pointers Traversed	Buffer
1	X
1,1	X=
1,1,2	X=
1,1,2,3,4	X=A
1,1,2,3,4,4	X=A+
1,1,2,3,4,4,5	X=A+B
1,1,2,3,4,4,5,5,3	X=A+B+
1,1,2,3,4,4,5,5,3,6,7	X=A+B+C
1,1,2,3,4,4,5,5,3,6,7,7	X=A+B+C*
1,1,2,3,4,4,5,5,3,6,7,7,8	X=A+B+C*D
L,L,2,3,4,4,5,5,3,6,7,7,8,8,6,2	X=A+B+C*D

So far we have described the generation of simple arithmetic statements where the order of computation was determined by the hierarchy values of the operators. But it is well know that one can change this order by the use of parentheses.

If operations OP1, OP2 are executed sequentially and the hierarchy number of the first operator is less than the one of the second it is implied that both its operands are to be enclosed in parentheses. (Where each operand could be another node of course). For example, consider the following tree structure where each flagged node is marked by a star:



and the generated statement:

X=A+Y*(B+C+D)

The "take" operator (↓) is represented in the operator table by a left parenthesis. In addition, the node accommodating it, is flagged unconditionally after the processing of the first operand.

Having described some interesting parts of the implementation of the adopted optimizing strategies, a listing of the developed algorithm is now given.

```
C                        ******************
C                        * INITIALISATICN *
C                        ******************
C
C
C***************************************************************************
C     LISTING OF VARIABLES USED
C     -------------------------
C
C     RES          WORK TABLE
C     IRES         POINTER TO THE WORK TABLE
C     DECL         SPECIFICATION TABLE
C     IDECL        POINTER TO THE DECL TABLE
C     TRIPLE       INSTRUCTION TABLE
C     ITRPLE       POINTER TO THE TRIPLE TABLE
C     SEQ          SEQUENCE TABLE
C     ISEQ         POINTER TO THE SEQUENCE TABLE
C     OPRND        OPERAND TABLE
C     N3           POINTER TO THE OPRND TABLE.
C     FL           FLOW TABLE
C     IFL          POINTER TO THE FL TABLE
C     LBL          TABLE OF INSTRUCTION NUMBERS
C     ILB          POINTER TO LBL
C     SUC          SUCCESSOR-PREDECESSOR TABLE
C     ISUC         POINTER TO THE SUC TABLE
C     OPRTR        OPERATOR TABLE
C     HRCH         HIERARCHY TABLE
C     DISTCE       DISTANCE TABLE
C***************************************************************************
C
C
      BLOCK DATA
      COMMCN/A3/IRES,RES/A4/DECL,IDECL/A5/TRIPLE,ITRPLE,SEQ,ISEQ
     1/A6/OPRND,N3/A7/FL,IFL/A8/ILBL,SUC,ISUC/A9/OPRTR/A10/HRCH/A13/DISTCE
```

```
     2CE,RESULT,CCPY
      INTEGER*2 IRES,DECL(20),IDECL,TRIPLE(3, 600),ITRPLE,SEQ( 900),ISEQ
     1,FL( 80),IFL,ILBL,SUC( 600),ISUC,RES(1600),HRCH(24)/3*0,2*1,2,3,4
     2,7*5,2*6,2*7,3*8,9,10/,DISTCE(80,80)/6400*0/
      REAL*8 OPRTR(24)/'(','<','(',')','>','.OR.','.AND.','.NOT.','.EQ.'
     1,'.NE.','=','.LT.','.LE.','.GT.','.GE.',',','|','-','+','*','/',
     2'-','**','%'/,OPRND(80)
      LOGICAL*1 RESULT(30,80)/2400*F/,COPY(80,80)/6400*F/
      DATA IRES,IDECL,ITRPLE,ISEQ,IFL,ILBL,ISUC/1,0,1,2,2,1,2/,FL(1)/1/,SUC(1)/1
     1SUC(1)/1/,SEQ(1)/0/
      END
C           ****************************************
C           * MAIN PROGRAM FOR PROGRAM OPTIMIZATION *
C           ****************************************
C
C     AUTHOR
C     ------
C     NICHOLAS P. ALEXANDRAKIS
C
C     SESSION  1972-1973
C
C     DESCRIPTION
C     -----------
C     THE FUNCTIONS OF THIS PROGRAM ARE:
C    1)TRANSLATION OF THE SOURCE PROGRAM INTO THE INTERMEDIATE TEXT.
C    2)CONTROL FLOW ANALYSIS.
C    3)OPTIMIZATION.
C    4)CODE GENERATION.
C
C     SUBROUTINE AND FUNCTION SUBPROGRAMS REQUIRED:
C     PUT
C     TRSLTE
C     HELP
C     FNDCLR
```

```
C       NUMBER
C       DFEC
C       ID
C       REDUNT
C***********************************************************************
C
C
C
C
        INTEGER*4 SK,WR
        INTEGER*2 LBL( 80),SUC( 600),FL( 80),SEQ( 900),TRIPLE(3, 600),INPU
       1T(72),START/'<'/,END/'>'/,FI/'F'/,DO(2,60),IDO/0/,IKLB/0/,BLANK/' '/,ILB
       2'/,ILBL,ISUC,IFL,ISEQ,ITRPLE,RPR/')'/,COMA/','/,IRES,ZERO/'0'/,DECL(20),ID
       3L(20),IDECL,IZ/0/,RES(1600),HRCH(24),OUTPUT(72),IOUT,MRK/Z5240/
       4,IWTA/'I'/,LPR/'('/,PUSHDN(20),APST/''''/,DISTCE(80,80),LENGTH/1/
       5,BO(30),LST(30),ILST/0/,PSB(20),IPSB
        LOGICAL*1 PAR(20),USED(20),CNTY(80,80)/6400*F/,RESULT(30,80),COPY(80,80),T
       180,80),TMP
        REAL*8 OPRND(80)
        COMMON/A1/INPUT/A3/IRES,RES/A4/DECL,IDECL/A5/TRIPLE,ITRPLE,SEQ,ISEQ/A6/OPR
       1Q/A6/OPRND,N3/A7/FL,IFL/A8/ILBL,SUC,ISUC/A10/HRCH/A11/OUTPUT
       2/A12/PSB,IPSB/A13/DISTCE,RESULT,COPY
        WRITE(6,34)
 34     FORMAT('1',6X,'INPUT PROGRAM'//)
        IOP=0
        N3=0
C*****READ ONE LINE
 1      READ(5,2,END=31)INPUT
 2      FORMAT(72A1)
C*****OUTPUT IT FOR COMPARISON WITH THE OUTPUT GENERATED BY THE OPTIMIZER
        WRITE(6,36)INPUT
 36     FORMAT(' ',72A1)
C*****IS THIS STATEMENT THE LAST ONE OF A RANGE OF A DO?
        IF(IKLB)38,38,37
```

```
C*****YES.CHANGE BLOCK.
37    IF(FL(IFL-1).NE.ISEQ-1)CALL HELP(0,0,0,0,0,2)
      IKLB=0
C******ELIMINATION OF EMBEDED BLANKS FROM COLUMN 7 TO THE END OF THE STRING.
C*****INITIALIZATION CF IB,IS.
 38   IB=7
      IS=1
      DC 3 I=7,72
C*****IF WE ARE IN A REGION ENCLOSEC IN APCSTRCPHES THEN RESPECT BLANKS.
      IF(IS+1.EQ.0)GO TO 5
C*****IF THERE IS A BLANK INCREASE I BUT NOT IB.
      IF(INPUT(I).EQ.BLANK)GC TC 3
C*****IF THERE IS AN APOSTROPHE CHANGE THE SIGN CF IS.
5     IF(INPUT(I).EQ.APST)IS=-IS
C*****IF NC BLANK FOUNC INCREASE IB.
      IF(I.EQ.IB)GO TO 4
C*****THERE ARE EMBEDED BLANKS,MOVE INPUT(I) TO THE RIGHT POSITICN.
      INPUT(IB)=INPUT(I)
      INPUT(I)=BLANK
4     IB=IB+1
3     CCNTINUE
C*****CALL SUBROUTINE ID TO RECCGNIZE THE INPUT LINE.
      CALL ID(ID1,ID2)
C*****IF IT IS A SPECIFICATION STATEMENT GOTO READ ANCTHER LINE.
      IF(ID1.EQ.0)GC TC 1
C££££LABEL TEST
C*****IF IT IS A FORMAT STATEMENT IGNORE THE STATEMENT NUMBER.
      IF(ID1.EQ.9.AND.INPUT(7).EQ.FI)GO TO 12
C*****SEARCH THE FIRST FIVE ENTRIES.
      DO 6 I=1,5
      K=6-I
C*****IS THERE ANY NUMBER?
      IF(INPUT(K).NE.BLANK)GO TO 7
 6    CONTINUE
```

```
      GO TO 9
C*****THERE IS ONE CHANGE BLOCK.
 7    IF(FL(IFL-1).NE.ISEQ-1)CALL HELP(0,0,0,0,0,2)
      I=K+1
C*****TRANSFORM THE NUMBER FOUND INTO NUMERICAL FORM AND INFORM LBL.
      LBL(ILBL)=NUMBER(I)
C*****BY THE WAY,ARE THERE ANY UNCLOSED DO-LOOPS?IDO IS THE SUBSCRIPT
C***** OF THE STACK 'DO'POINTING TO THE LAST ITEM INSERTED.
C*****THE STACK 'DO' HOLDS DO PARAMETER INFORMATION.
 8    IF(IDO.EQ.0)GO TO 404
C*****IF THE INSTRUCTION NUMBER FOUND IS NOT EQUAL WITH THE LAST ENTRY IN THE DO
C*****STCK GET OUT.
      IF(LBL(ILBL).NE.DO(1,IDO))GO TO 404
C*****IF IT IS EQUAL SET IKLB=1 TO CHANGE BLOCK AFTER THE PROCESSING
C*****OF THIS LINE
      IKLB=1
C*****INFORM SUC TOO.
      SUC(ISUC)=DO(2,IDO)
      ISUC=ISUC+1
C*****PREPARE THE DO STACK FOR THE NEXT TEST.THE STATEMENT COULD
C*****BE POSSIBLY THE END OF THE RANGE OF MORE THAN ONE DOS.
      IDO=IDO-1
      GO TO 8
 404  IF(IKLB.EQ.0)GO TO 9
      LBL(ILBL)=-LBL(ILBL)
 9    I=7
C*****IS PARAMETER ID1 NEGATIVE?
 10   IF(ID1)11,11,12
 11   IOP=1
      ID1=-ID1
 12   GO TO(13,18,19,22,24,24,25,26,29,30),ID1
C££££ARITHMETIC IF
 13   J=I
C*****OPTIMIZABLE?
```

```
      IF(IOP)14,14,15
C*****IF NCT FIND THE END OF THE STRING.
 14   CALL FNDDLR(J,I2,IZ,IZ,BLANK)
      GO TO 16
C*****IF YES FIND THE END OF THE ARITHMETIC EXPRESSICN AS WELL.
 15   CALL FNCDLR(J,I2,RPR,IZ,BLANK)
      K=I+2
C*****TRANSFORM THE ARITHMETIC EXPRESSION INTO TRIPLES.
      INPUT(K)=START
      INPUT(J)=END
      CALL TRSLTE(K,J)
      J=J+1
 16   K=I2+1
C*****INFCRM SUC.
 17   SUC(ISUC)=-NUMBER(K)
      ISUC=ISUC+1
      IF(INPUT(K).EQ.CCMA)GC TC 17
      SK=IRES
C*****INSERT LAST TRIPLE,LOAD LINE(OR PART)TO THE RES TABLE,AND CHANGE
C*****BLCCK.
      CALL HELP(ICP,SK,31,J,I2,1)
      IOP=0
      GC TC 1
C££££GO TO
 18   J=I
C*****FINC THE LAST CHARACTER OF THE STRING.
      CALL FNCDLR(J,I2,IZ,IZ,BLANK)
C*****2NFCRM SUC TABLE.
      K=I2+1
      SUC(ISUC)=-NUMBER(K)
      ISUC=ISUC+1
      SK=IRES
C*****INSERT LAST TRIPLE,LOAD LINE TO THE RES TABLE AND CHANGE BLCCK.
      CALL HELP(0,SK,32,I,I2,1)
```

```
      GO TO 1
C££££££LOGICAL IF
 19   J=I
C*****FIND THE END OF LOGICAL EXPRESSION.
      CALL FNDDLR(J,I2,RPR,IZ,IZ)
C*****IS IT OPTIMIZABLE?
      IF(IOP.EQ.0)GO TO 20
C*****YES.TRANSFORM LOGICAL EXPRESSION INTO TRIPLES.
      K=I+2
      INPUT(K)=START
      INPUT(J)=END
      CALL TRSLTE(K,J)
      CALL HELP(ID2,0,33,0,I2,3)
C*****PREPARE PARAMETERS FOR PROCESSING OF THE NEAR BY STATEMENT.
 21   I=J+1
      ID1=ID2
C*****CLOSE BLOCK AFTER THAT.
      IKLB=1
      ICP=0
      GO TO 10
C*****INSERT TRIPLE,LOAD LINE AND CHANGE BLOCK WITH CODE 3.(SEE DESCRIPTION
C*****OF SUBROUTINE HELP).
 20   SK=IRES
      CALL HELP(ID2,SK,33,I,J,3)
      GO TO 21
C££££££COMPUTED GO TO
 22   J=I
      CALL FNDDLR(J,I2,RPR,COMA,BLANK)
C*****INFORM SUC
 23   SUC(ISUC)=-NUMBER(J)
      ISUC=ISUC+1
      IF(INPUT(J).EQ.COMA)GO TO 23
      SK=IRES
C*****CHANGE BLOCK.
```

```
      CALL HELP(0,SK,34,I,I2,1)
      GO TO 1
C££££CALL,READ
 24   J=I
      CALL FNDDLR(J,I2,IZ,IZ,BLANK)
      WR=ID1+30
      SK=IRES
      CALL HELP(0,SK,WR,I,I2,0)
C*****FIND DEFINED VARIABLES IN THESE STATEMENTS AND INFORM THE
C*****INSTRUCTION TABLE FOR THAT.
      CALL DFED(ID1,I)
      GO TO 1
C££££STOP
 25   J=I+3
      SK=IRES
C*****CHANGE BLOCK BUT DO NOT LINK WITH THE NEXT.
      CALL HELP(0,SK,37,I,J,1)
      GO TO 1
C££££DO
C*****CHANGE BLOCK.
26    IF(FL(IFL-1).NE.ISEQ-1)CALL HELP(0,0,0,0,0,2)
      J=I+2
C*****FIND THE INSTRUCTION NUMBER OF THE LAST STATEMENT OF THE RANGE OF
C*****THE DO.
      DO 27 K=J,72
      IF(INPUT(K).LT.ZERO)GO TO 28
 27   CONTINUE
 28   IDO=IDO+1
C*****INFORM THE DO TABLE.
      DO(1,IDO)=NUMBER(K)
      DO(2,IDO)=IFL-1
      CALL FNDDLR(J,I2,IZ,IZ,BLANK)
C*****FIND THE DEFINED VARIABLE AND INFORM THE INSTRUCTION TABLE.
      CALL DFED(ID1,7)
```

```
      SK=IRES
C*****ADD LAST TRIPLE AND LOAD LINE TO THE WORK TABLE.
      CALL HELP(0,SK,38,I,I2,0)
      GO TO 1
C£££££ALL CTHERS
C*****IS IT A FORMAT STATEMENT?
 29   IF(INPUT(7).NE.FI)GO TO 100
      I=1
      I2=71
      GO TO 101
100   CALL FNDDLR(I,I2,IZ,IZ,BLANK)
 101  SK=IRES
      CALL HELP(0,SK,39,I,I2,0)
      GO TO 1
C£££££ASSIGN
 30   CALL FNDDLR(I,I2,IZ,IZ,BLANK)
C*****TRANSFORM IT INTO TRIPLES.
      I=I-1
      I2=I2+1
      INPUT(I)=START
      INPUT(I2)=END
      CALL TRSLTE(I,I2)
      GO TO 1
 31   IFL=IFL-1
C*****CHANGE THE NEGATIVE ENTRIES OF SUC WITH THE CORRESPONDING BLOCK
C*****NUMBERS.
      ISUC=ISUC-2
      SUC(ISUC+1)=0
      ILBL=ILBL-1
      DO 32 I=1,ISUC
      IF(SUC(I).GE.0)GO TO 32
      DO 33 J=1,ILBL
      K=LBL(J)
      IF(K.LT.0)K=-K
```

```
      IF(K+SUC(I).NE.0)GO TO 33
      SUC(I)=J
      GO TO 32
 33   CONTINUE
 32   CONTINUE
      LAE=N3+100
      WRITE(6,35)
 35   FORMAT('1',40X,'FLOW ANALYSIS',//)
C
C
C***********************************************************************
C
C
C                  ************************
C                  * CONTROL FLOW ANALYSIS *
C                  ************************
C
C
C     THREE MAIN MATRICES ARE USED IN THIS SEGMENT:THE DISTCE,THE CNTY
C     AND THE COPY.
C
C***********************************************************************
C
C*****INFORM CNTY AND DISTCE MATRICES WITH THE INFORMATION HELD IN SUC.
      K=1
      IND=1
 203  K=K+1
      IF(SUC(K).EQ.0)GO TO 204
      CNTY(IND,SUC(K))=.TRUE.
      DISTCE(IND,SUC(K))=1
      GO TO 203
 204  IF(SUC(K+1).EQ.0)GO TO 205
      K=K+1
      IND=SUC(K)
```

```
      GO TO 203
C*****OUTPUT PREDECESSORS AND SUCCESSORS FOR CONFIRMATION.
 205  WRITE(6,290)
 290  FORMAT(///,4X,'BLOCK',4X,'SUCCESSCRS')
C*****VARIABLE I HOLDS THE NUMBER OF THE PREDECESSOR BLOCK.
      DC 291 I=1,IND
      IBO=C
C*****VARIABLE J HOLDS THE NUMBER OF A SUCCESSOR BLOCK.
      DC 292 J=1,INC
      IF(.NOT.CNTY(I,J))GC TO 292
      IBO=IBO+1
      BC(IBC)=J
 292  CONTINUE
C*****ARE THERE SUCCESSORS OR NOT ?
      IF(IBO.EQ.0)GO TO 293
C*****IF THERE ARE PRINT THEM.
      WRITE(6,294)I,(BO(K),K=1,IBO)
 294  FORMAT(8X,I3,6X,20(I3,','))
      GC TC 291
C*****IF THERE ARE NOT IT IS AN EXIT NODE.
 293  WRITE(6,295)I
 295  FCRMAT(8X,I3,' IS AN EXIT NODE')
 291  CONTINUE
C*****LET'S FIND THE UNIQUE CLOSED PATHS.
      IR=0
      WRITE(6,261)
 261  FORMAT(///' THE UNIQUE CLOSED PATHS OF LENGTH LESS THAN OR EQUAL
     1TO L ARE')
 219  WRITE(6,262)LENGTH
 262  FORMAT(2X,'FOR L=',I3)
C*****SEARCHING OF THE DIAGONAL ENTRIES.
      K1=0
      IS=C
 206  K1=K1+1
```

```
      IF(K1.GT.IND)GO TO 214
      IF(.NOT.CNTY(K1,K1))GO TO 206
C*****IF THE DISTANCE FROM ITSELF IS EQUAL TO LENGTH SKIP NEXT TEST,
      IF(LENGTH.EQ.DISTCE(K1,K1))GO TO 253
C*****DOES DISTANCE DIVIDE LENGTH?IF YES REJECT IT.
      IF(LENGTH.EQ.LENGTH/DISTCE(K1,K1)*DISTCE(K1,K1))GO TO 206
 253  IR=IR+1
      ILST=ILST+1
C*****THE ENTRIES OF ARRAY LIST POINT TO THE BOOLEAN VECTORS RESULT.
      LST(ILST)=IR
      RESULT(IR,K1)=.TRUE.
C*****LET'S FIND THE PARTNERS OF K1.
      DO 208 J=1,IND
      IF(K1.EQ.J)GO TO 208
      IF(.NOT.CNTY(J,J))GO TO 208
 254  IF(DISTCE(K1,J).EQ.0)GO TO 208
      IF(DISTCE(J,K1).EQ.0)GO TO 208
      IF(DISTCE(K1,J)+DISTCE(J,K1).GT.LENGTH)GO TO 208
      RESULT(IR,J)=.TRUE.
 208  CONTINUE
C*****IS RESULTED CLOSED PATH UNIQUE?
      IF(IR.EQ.1)GO TO 266
      K2=IR-1
      DO 209 J=1,K2
      DO 210 L=1,IND
      IF(RESULT(J,L).AND..NOT.RESULT(IR,L))GO TO 209
      IF(.NOT.RESULT(J,L).AND.RESULT(IR,L))GO TO 209
 210  CONTINUE
C*****NO.REJECT IT.
      ILST=ILST-1
      DO 211 N=1,IND
 211  RESULT(IR,N)=.FALSE.
      IR=IR-1
      GO TO 206
```

```
  209  CONTINUE
C*****WHEN YOU FINISH WITH CURRENT VALUE OF LENGTH OUTPUT THE CLOSED PATHS
C*****FOUND SO FAR.
  266  IS=1
       IBO=0
       DO 263 J=1,IND
       IF(.NOT.RESULT(IR,J))GO TO 263
       IBO=IBO+1
       BO(IBO)=J
  263  CONTINUE
       WRITE(6,264)(BO(J),J=1,IBO)
  264  FORMAT(6X,25(I3,'-'))
       GO TO 206
  214  LENGTH=LENGTH+1
       IF(IS.EQ.0)WRITE(6,265)
  265  FORMAT(4X,'NONE')
C*****DID WE FINISH.?
       IF(LENGTH.GT.IND)GO TO 218
C*****IF NOT COPY THE CNTY MATRIX TO COPY AND RISE CNTY TO THE NEXT POWER.
       DO 215 I=1,IND
       DO 215 J=1,IND
       COPY(I,J)=CNTY(I,J)
  215  CNTY(I,J)=.FALSE.
       K=1
       I=1
  216  K=K+1
       IF(SUC(K).EQ.0)GO TO 300
       J=SUC(K)
       DO 217 IK=1,IND
       CNTY(I,IK)=CNTY(I,IK).OR.COPY(J,IK)
       IF(.NOT.CNTY(I,IK))GO TO 217
       IF(DISTCE(I,IK).NE.0)GO TO 217
       DISTCE(I,IK)=LENGTH
  217  CONTINUE
```

```
      GO TO 216
 300  IF(SUC(K+1).EQ.0)GO TO 219
      K=K+1
      I=SUC(K)
      GO TO 216
 218  WRITE(6,267)
 267  FORMAT(////,3X,'THE OPTIMIZABLE STRONGLY-CONNECTED REGIONS ARE')
C*****INITIALIZE COPY.
      DO 305 I=1,IND
      DO 305 J=1,IND
 305  COPY(I,J)=.FALSE.
C*****REJECT CLOSED PATHS WHICH HAVE MORE THAN ONE PREDECESSORS.
C*****REJECT ALSO THE CLOSED PATHS WHOSE PREDECESSOR HAS MORE THAN ONE
C*****SUCCESSOR.
      K=1
      IND=1
 301  K=K+1
      IF(SUC(K).EQ.0)GO TO 302
      COPY(IND,SUC(K))=.TRUE.
      GO TO 301
 302  IF(SUC(K+1).EQ.0)GO TO 303
      K=K+1
      IND=SUC(K)
      GO TO 301
 303  DO 284 I=1,IND
      DISTCE(I,1)=0
 284  CNTY(I,1)=.FALSE.
      IS=0
      II=0
C*****EXAMINE EVERY STRONGLY CONNECTED REGION.
      DO 274 I=1,IR
      II=II+1
      IF(IS.EQ.0)GO TO 278
      DO 286 K=1,IND
```

```
 286  CNTY(K,1)=.FALSE.
      IS=C
 278  DO 275 J=1,IND
      IF(.NOT.RESULT(I,J))GO TO 275
      IF(IS.EQ.1)GO TC 276
      IS=1
      DO 280 K=1,IND
 280  CNTY(K,1)=COPY(K,J)
      GO TO 275
C*****FIND THE PREDECESSORS.
 276  DO 279 K=1,IND
 279  CNTY(K,1)=CNTY(K,1).OR.COPY(K,J)
 275  CONTINUE
      IF(IS.EQ.0)GO TO 274
      IC=C
      DO 285 J=1,IND
      IF(.NOT.CNTY(J,1))GO TO 285
      IF(RESULT(I,J))GO TO 285
C*****IF MORE THAN ONE PREDECESSORS REJECT THE REGION.
      IC=IC+1
      IF(IC.EQ.2)GO TO 288
      IT=C
      DO 287 K=1,IND
      IF(.NOT.COPY(J,K))GO TO 287
C*****IF MCRE THAN ONE SUCCESSORS REJECT THE REGION.
      IT=IT+1
      IF(IT.EQ.2)GO TO 288
      IKA=J
 287  CONTINUE
 285  CONTINUE
C*****PUT THE NUMBER OF THE PREDECESSOR BLOCK INTO THE APPROPRIATE ENTRY
C*****OF THE FIRST COLUMN IN THE DISTCE TABLE.
      IF(IC.EQ.0)GO TO 288
      DISTCE(I,1)=IKA
```

```
      GO TO 274
 288  ILST=ILST-1
      DO 289 J=II,ILST
 289  LST(J)=LST(J+1)
      II=II-1
 274  CONTINUE
C*****IN THIS SEGMENT WE SHORT THE STRONGLY CONNECTED REGIONS AND
C*****WE REJECT THE OVERLAPPING ONES.
      K=ILST
 222  K=K-1
      IF(K.LT.1)GO TO 227
      I=1
 304  I1=LST(I)
      I2=LST(I+1)
      M=0
      IP=0
      IN=0
      IZ=0
      DO 224 N=1,IND
      IF(.NOT.RESULT(I1,N).OR.RESULT(I2,N))GO TO 268
C*****IP = 1 IFF THERE IS N : RESULT(I1,N)=F AND RESULT(I2,N)=T
C*****OTHERWISE IP = 0.
      IP=1
      GO TO 224
 268  IF(RESULT(I1,N).OR..NOT.RESULT(I2,N))GO TO 269
C*****IN = 1 IFF THERE IS AT LEAST ONE N:RESULT(I1,N)=T AND RESULT(I2,N)=F
C*****OTHERWISE IN=0.
      IN=1
      GO TO 224
 269  IZ=1
C*****IZ = 1 MEANS THAT WE HAVE F-F OR T-T.
      IF(RESULT(I1,N).AND.RESULT(I2,N))M=1
C*****M = 1 MEANS T-T.
 224  CONTINUE
```

```
C*****IF IP+IN+M=3 THE REGIONS ARE OVERLAPPING.
      IF(IP+IN+M.NE.3)GO TO 270
C*****REJECT ONE.
      ILST=ILST-1
      K=K-1
      M=I+1
      DO 226 J=M,ILST
  226 LST(J)=LST(J+1)
      GC TC 304
C*****IF IP+IZ=2 AND IN=0 THE FIRST REGICN IS A SUBSET OF THE SECOND.INTERCHANGE
C*****THEIR POINTERS IN THE LIST TABLE.
 270  IF(IP+IZ.NE.2.OR.IN.NE.0)GO TO 223
      LST(I+1)=I1
      LST(I)=I2
 223  I=I+1
      IF(I.LE.K)GO TO 304
      GO TO 222
C*****WRITE THEM DOWN.
 227  DC 272 I=1,ILST
      IBO=0
      I1=LST(I)
      DO 273 J=1,IND
      IF(.NCT.RESULT(I1,J))GO TO 273
      IBO=IBO+1
      BO(IBO)=J
 273  CCNTINUE
      WRITE(6,264)(BO(IA),IA=1,IBO)
 272  CONTINUE
      WRITE(6,277)
 277  FORMAT(////,1X,'THE IMMEDIATE PREDECESSORS OF THE ABOVE REGIONS ARE')
     1E')
C*****PRINT THE PREDECESSORS OF THESE REGIONS IN THE SAME ORDER.
      DC 296 I=1,ILST
      I1=LST(I)
```

```
      WRITE(6,297)DISTCE(I1,1)
 297  FORMAT(' ',I3)
 296  CONTINUE
C
C
C*****************************************************************************
C
C
C                     ***************
C                     * OPTIMIZATION *
C                     ***************
C     EVERY BLOCK IS EXAMINED FOR REDUNDUNT INSTRUCTION ELIMINATION.
C     IN THE SAME PASS THE MATRIX COPY IS INFORMED.EVERY LINE IN THIS
C     MATRIX CORRESPONDS TO A VARIABLE AND EVERY COLUMN TO A BLOCK.
C     SO IF THE VARIABLE K IS DEFINED IN THE BLOCK NO 2 THE SECOND ENTRY
C     OF THE CORRESPONDING BOOLEAN VECTOR WILL HAVE THE VALUE TRUE.
C     AFTER THIS STAGE EVERY STRONGLY CONNECTED REGION IS EXAMINED FOR
C     REMOVAL OF INVARIANT INSTRUCTIONS.
C     ARRAY PSB HOLDS THE POINTERS OF THE REMOVED INSTRUCTIONS.IF THERE
C     ARE INSTRUCTIONS REMOVED WE DEPOSE THEM IN THE BOTTOM OF THE
C     PREDECESSOR BLOCK OF THE REGION AND WE EXAMINE THE PREDECESSOR FOR
C     REDUNDANT INSTRUCTIONS.
C
C
C*****************************************************************************
C
C
C*****REDUNDANT INSTRUCTION ELIMINATION.
C*****ARRAY COPY IS GOING TO HOLD THE DEFINITION OF EVERY VARIABLE
C*****LET'S INITIALIZE IT.
      DO 312 I=1,80
      DO 312 J=1,IND
 312  COPY(I,J)=.FALSE.
      I=IFL-1
```

```
      INV=0
      ILBL=0
      DO 313 J=1,I
  313 CALL REDUNT(J,1)
C*****IF THERE ARE NO OPTIMIZABLE STRONGLY CONNECTED REGIONS SKIP THE
C*****SEGMENT BELLOW.
      IF(ILST.EQ.0)GO TO 306
      DO 307 I=1,ILST
      IPSB=0
C*****THE VARIABLE ISUC IS USELESS NOW AND IT IS IN COMMON ,LET'S USE IT
      ISUC=LST(I)
      K=ISUC
C*****REMOVAL OF INVARIANT INSTRUCTIONS.
      DO 308 J=1,IND
      IF(.NOT.RESULT(K,J))GO TO 308
      IF(SEQ(FL(J)).EQ.1)GO TO 308
      CALL REDUNT(J,2)
      SEQ(FL(J))=1
  308 CONTINUE
C*****DID WE FIND ANYTHING?
      IF(IPSB.EQ.0)GO TO 307
      INV=INV+IPSB
C*****DEPOSITION OF THE PSEUDO-BLOCK.
      M=FL(DISTCE(K,1)+1)
      L=ISEQ-1
  309 SEQ(L+IPSB)=SEQ(L)
      L=L-1
      IF(L.GE.M)GO TO 309
      IPR=M
      I3=FL(DISTCE(K,1))
      DO 310 I1=1,IPSB
      I2=FSB(I1)
      IF(TRIPLE(3,I2).EQ.11)GO TO 503
      I4=IPR
```

```
  408  I4=I4-1
       IF(I4.EQ.I3)GO TO 503
       I5=SEQ(I4)
       IF(I5+1.EQ.0)GO TO 408
       DO 409 I6=1,3
       IF(TRIPLE(I6,I5).NE.TRIPLE(I6,I2))GO TO 5C1
  409  CONTINUE
       SEQ(IPR)=I5
       I7= I1+1
       DC 500 I6=I7,IPSB
       IF(TRIPLE(1,PSB(I6)).EQ.I2+1000)TRIPLE(1,PSB(I6))=I5+1000
  500  IF(TRIPLE(2,PSB(I6)).EQ.I2+10C0)TRIPLE(2,PSB(I6))=I5+1C00
       GC TC 310
  501  I6=TRIPLE(3,I5)
       IF(I6.NE.11.AND.I6.NE.25)GO TO 408
       I7=TRIPLE(1,I5)
       IF(I7.LT.1000)GC TC 502
       IF(TRIPLE(3,I2).NE.3)GO TC 408
       I7=TRIPLE(1,I7-1000)
       IF(I7.EC.TRIPLE(1,I2))GO TC 503
       GO TO 408
  502  IF(I7.EQ.TRIPLE(1,I2))GO TO 5C3
       IF(I7.NE.TRIPLE(2,I2))GO TO 408
  503  SEQ(IPR)=I2
  310  IPR=IPR+1
       M=DISTCE(K,1)+1
       DO 311 J=M,IFL
  311  FL(J)=FL(J)+IPSB
       ISEQ=ISEQ+IPSB
C*****FIND THE REDUNDANT INSTRUCTIONS OF THE BLOCK WHICH RECEIVED THE CODE.
       M=DISTCE(K,1)
       CALL REDUNT(M,1)
  307  CCNTINUE
       IDO=0
```

```
      DC 504  I=1,IND
      I1=FL(I)+1
      I2=FL(I+1)-1
      DO 504  J=I1,I2
      K=SEQ(J)
      IF(K+1.EQ.0)GO TO 504
      IF(IDO.EQ.0)GO TO 505
      DC 506 IU=1,IDO
      IF(TRIPLE(1,K).EQ.DO(1,IU))TRIPLE(1,K)=DO(2,IU)
  506 IF(TRIPLE(2,K).EQ.DO(1,IU))TRIPLE(2,K)=DO(2,IU)
  505 IF(TRIPLE(3,K).NE.11)GO TO 504
      IF(TRIPLE(1,K).LE.LAE)GO TO 504
      IF(TRIPLE(1,K).GT.1000)GO TO 504
      IF(TRIPLE(2,K).LE.LAE)GO TO 504
      IF(TRIPLE(2,K).GT.1000)GO TO 504
      IDO=IDO+1
      DO(1,IDO)=TRIPLE(1,K)
      DO(2,IDO)=TRIPLE(2,K)
      SEQ(J)=-1
  504 CONTINUE
C
C
C*******************************************************************************
C
C
C                    ******************
C                    * CODE GENERATICN *
C                    ******************
C     WE TRANSLATE NOW THE RESULTED INTERMEDIATE TEXT INTO FORTRAN.
C     BECAUSE RECURSICN IS NOT AVAILABLE IN FORTRAN WE HAVE TO MAINTAIN
C     A PUSH-DOWN LIST(PUSHDN).LOGICAL ARRAY PAR INFORMS US IF THE
C     CORRESPONDING INSTRUCTION IS TO BE ENCLOSED IN PARENTHESES OR NOT.
C     LOGICAL ARRAY USED INFORMS US IF AN INSTRUCTION WAS USED BEFORE.
C
```

```
C
C*******************************************************************************
C
C
 306  IFCR=0
      WRITE(6,400)
 400  FORMAT('1',6X,'OUTPUT PROGRAM'//)
C*****OUTPUT THE SPECIFICATION STATEMENTS.
C*****ARE THERE ANY SPECIFICATION STATEMENTS?
      IF(IDECL.EQ.0)GO TO 43
      DO 41 I=1,IDECL
      J=DECL(I)
      K=6
 40   K=K+1
      OUTPUT(K)=RES(J)
      J=J+1
C*****THERE IS A SPECIAL MARK AT THE END OF EVERY DEPCSED STRING.
      IF(RES(J).NE.MRK)GO TO 40
 41   WRITE(6,42)(OUTPUT(N),N=7,K)
 42   FORMAT(7X,72A1)
 43   IOUT=0
C*****OUTPUT IS THE ARRAY TO BE PRINTED.
      I1=IFL
C*****VARIABLE I4 HOLDS THE NUMBER CF THE BLOCK UNDER CONSIDERATION.
      I4=1
 44   IS=0
      WRITE(6,402)I4
 402  FORMAT(' C BLOCK ',I2)
C*****LET'S FIND THE LIMITS OF THE BLOCK.
      I2=FL(I4)
      I3=FL(I4+1)
C*****VARIABLE I2 POINTS AN INSTRUCTION IN SEQ.
 45   I2=I2+1
      IF(I2.EQ.I3)GO TO 58
```

```
      M1=SEQ(I2)
      IF(M1+1.EQ.0)GO TO 45
      M=TRIPLE(3,M1)
C*****COMPAIRE THE THIRD ENTRY OF CURRENT TRIPLE WITH 30.
      IF(M.LT.30)GO TO 56
C*****IF IT IS A STATEMENT ,INCREASE IS.
      IS=IS+1
C*****IS IT AN ARITHMETIC IF?
      IF(M.NE.31)GO TO 54
C*****IF YES HAVE WE GENERATED ITS ARITHMETIC EXPRESSION OR NOT?
      IF(TRIPLE(1,M1).EQ.1)GO TO 52
C*****IF NOT LOAD THE STATEMENT INTO OUTPUT AND PRINT IT.
 53   K1=TRIPLE(2,M1)
 46   IOUT=IOUT+1
      OUTPUT(IOUT)=RES(K1)
      K1=K1+1
      IF(RES(K1).NE.MRK)GO TO 46
      IF(IFOR.EQ.1)GO TO 59
      IF(M.EQ.33)GO TO 58
 47   IF(LBL(I4))405,50,407
 405  IF(I2+1.NE.I3)GO TO 50
      LBL(I4)=-LBL(I4)
 406  WRITE(6,48)LBL(I4),(OUTPUT(J),J=1,IOUT)
 48   FORMAT(' ',I5,1X,72A1)
 49   IOUT=0
      GO TO 45
 407  IF(IS.EQ.1)GO TO 406
 50   WRITE(6,51)(OUTPUT(J),J=1,IOUT)
 51   FORMAT(7X,72A1)
      GO TO 49
 59   WRITE(6,60)(OUTPUT(J),J=1,IOUT)
 60   FORMAT(' ',72A1)
      IFOR=0
C*****WE HAVE A LOGICAL OR ARITHMETIC IF.
```

```
      GO TO 49
 52   OUTPUT(ICUT+1)=IWTA
      OUTPUT(IOUT+2)=FI
      OUTPUT(IOUT+3)=LPR
      ICUT=ICUT+3
 57   K=I2-1
C*****LOGICAL ARRAY PAR INDICATES IN CASE OF TRUE THAT THE CCRESPONDING
C     INSTRUCTICN IS TO BE ENCLOSED IN PARENTHESES.
 61   PAR(1)=.FALSE.
C*****IPD IS THE INDEX OF THE PUSH-DOWN LIST.
      IPD=0
      I=SEQ(K)
75    IPD=IPD+1
      PUSHDN(IPD)=I
C*****LOGICAL ARRAY USED IS USED TO INDICATE IF WE HAVE ALREADY PASS FROM THE
C     TRIPLE.
      USED(IPD)=.FALSE.
C*****IF PARENTHESES ARE NEEDED PUT THE LEFT ONE IN.
      IF(.NOT.PAR(IPD))GO TO 76
      IOUT=IOUT+1
      OUTPUT(IOUT)=LPR
      GO TO 77
76    IF(TRIPLE(3,I).NE.3)GO TO 77
C*****IF TRIPLE(3,I) IS THE UNARY  OP-OR MINUS IGNORE FIRST CPERAND.
      PAR(IPD)=.TRUE.
77    IF(TRIPLE(3,I).EQ.22)GO TO 95
      IF(TRIPLE(3,I).EQ.8)GO TO 95
C*****DO WE HAVE A VARIABLE OR REFFERENCE TO ANOTHER TRIPLE?
      IF(TRIPLE(1,I).LT.1000)GO TO 78
C*****THE VALUE OF PARAMETER K1 IS THE INDEX CF THE REFFERENCED TRIPLE.
      K1=TRIPLE(1,I)-1000
C*****THE VALUE OF PARAMETER K2 IS THE OPERATOR OF THE REFFERENCED TRIPLE.
      K2=TRIPLE(3,K1)
C*****THE VALUE OF PARAMETER K3 IS THE OPERATOR OF CURRENT TRIPLE.
```

```
      K3=TRIPLE(3,I)
C*****CLEAR IPD+1 ENTRY OF PAR.
      PAR(IPD+1)=.FALSE.
C*****IF REFFERENCED OPERATION IS UNARY,PARENTHESIZE ITEXCEPT IF UNARY
C     MINUS IS AFTER AN '=' SIGN.
      IF(K2.NE.22)GO TO 80
      IF(K3.EQ.11)GO TO 80
      GO TO 96
80    IF(K2.EQ.3)GO TO 79
C*****TEST FOR PARENTHESES.
      IF(HRCH(K2).GE.HRCH(K3))GO TO 79
96    PAR(IPD+1)=.TRUE.
79    I=K1
      GO TO 75
C*****PUT THE FIRST OPERAND IN THE OUTPUT LINE.
 78   CALL PUT(IOUT,TRIPLE(1,I))
C*****PUT OPERATOR IN THE OUTPUT LINE.
 95   CALL PUT(IOUT,TRIPLE(3,I))
C*****IF THE SECOND OPERAND IS A REFFERENCE TO ANOTHER TRIPLE INFORM USED TABLE
C     THAT BOTH OPERANDS OF THIS TRIPLE HAVE BEEN USED AND FIND K1,K2,K3
C     AS ABOVE.
      IF(TRIPLE(2,I).LT.1000)GO TO 81
82    K1=TRIPLE(2,I)-1000
      USED(IPD)=.TRUE.
      K2=TRIPLE(3,K1)
      K3=TRIPLE(3,I)
C*****CLEAR THE ENTRY OF THE PAR TABLE CORRESPONDING TO THE REFFERENCED TRIPLE.
      PAR(IPD+1)=.FALSE.
      IF(K2.NE.22)GO TO 97
      IF(K3.EQ.11)GO TO 97
      GO TO 96
C*****PRELIMINARY TEST FOR PARENTHESES WHEN THE REFFERENCE IS FROM THE SECOND
C     OPERAND.
97    IF(HRCH(K2).NE.HRCH(K3))GO TO 80
```

```
      IF(K2.EQ.16)GO TO 79
      IF(K2.EQ.3)GO TO 79
      PAR(IPD+1)=.TRUE.
      GO TO 79
C*****PUT SECOND OPERAND IN THE OUTPUT LINE.
 81   CALL PUT(IOUT,TRIPLE(2,I))
      IF(.NOT.PAR(IPD))GO TO 83
C*****IF PARENTHESIS IS NEEDED PUT IT IN.
      IOUT=IOUT+1
      OUTPUT(IOUT)=RPR
C*****RETURN TO THE PREVIOUS INSTRUCTION.
C*****WE START NOW CLIMBING UP THE PUSH-DOWN LIST STEP BY STEP.
83    IPD=IPD-1
C*****IF WE FINISHED GO TO REPLACE POINTERS BY THEIR CORRESPONDING ENTRIES.
      IF(IPD.EQ.0)GO TO 84
      I=PUSHDN(IPD)
C*****IF WE USED IT PUT IN RIGHT PARENTHESIS(IF NEEDED) AND CONTINUE
C     RETURN PROCESS.
      IF(.NOT.USED(IPD))GO TO 85
86    IF(.NOT.PAR(IPD))GO TO 83
      IOUT=IOUT+1
      OUTPUT(IOUT)=RPR
      GO TO 83
C*****IF WE DID NOT USE IT PUT OPERATOR IN.
 85   CALL PUT(IOUT,TRIPLE(3,I))
C******IS TRIPLE (2,I) SINGLE VARIABLE OR REFFERENCE TO ANOTHER TRIPLE?
      IF(TRIPLE(2,I).GT.1000)GO TO 82
C*****IF WE HAVE A VARIABLE PUT IT IN.
      CALL PUT(IOUT,TRIPLE(2,I))
      GO TO 86
 84   IF(M.EQ.11)GO TO 47
      IOUT=IOUT+1
      OUTPUT(IOUT)=RPR
      IF(M.EQ.33)GO TO 58
```

```
      GO TO 53
 54   IF(M.NE.33)GO TO 55
      IF(TRIPLE(2,M1).EQ.0)GO TO 52
      GO TO 53
 55   K1=TRIPLE(2,M1)
      IF(RES(K1).LT.ZERO)GO TO 46
      IFOR=1
      GO TO 46
 56   IF(M.NE.11)GO TO 45
      IS=IS+1
      K=I2
      GO TO 61
 58   I4=I4+1
      IF(I4.NE.I1)GO TO 44
      WRITE(6,403)ILBL,INV
403   FORMAT(///' THE IMPROVEMENTS(?) MADE IN THE PROGRAM ARE',/3X,I2,2X
     1,'COMMON SUB-EXPRESSIONS ELIMINATED',/3X,I2,2X,'INSTRUCTIONS MOVED
     2')
      STOP
      END
C                             *****************
C                             * SUBROUTINE PUT *
C                             *****************
C
C*************************************************************************
C
C     USAGE
C       CALL PUT(IOUT,K)
C
C     DESCRIPTION OF PARAMETERS
C     IOUT - POINTER TO THE LAST CHARACTER IN ARRAY OUTPUT
C     K - POINTER TO THE OPRND OR OPRTR TABLES.
C
C     PURPOSE
```

```
C      WHEN A REAL*8 VARIABLE IS TO BE ADDED TO OUTPUT,SUBROUTINE 'PUT'
C      DCES THAT DISCARCING THE BLANK BYTES.
C
C***********************************************************************
C
       SUBRCUTINE PUT(IOUT,K)
       INTEGER*2 OUTPUT(68),TR,BLANK/' '/,K,IOUT
       REAL*8 OPRND( 80),OPRTR(24),CARR
       COMMON/A9/OPRTR/A6/OPRND,N3/A11/OUTPUT
       LOGICAL*1 G(8),RD(2),BL/' '/
       EQUIVALENCE(CARR,G),(TR,RD)
       RD(2)=BL
C*****IS IT AN OPERANC OR AN OPERATOR?
       IF(K-100)1,1,2
C*****OPERATOR.
 1     CARR=CPRTR(K)
       GO TO 3
C*****OPERAND.
 2     CARR=OPRND(K-100)
C*****LOAC ALL BYTES CF THE DOUBLE WORD BUT BLANKS,
 3     DO 4 I=1,8
       RD(1)=G(I)
       IF(TR.EQ.BLANK)GO TO 5
       IOUT=IOUT+1
 4     OUTPUT(IOUT)=TR
 5     RETURN
       END
C                       *********************
C                       * SUBROUTINE TRSLTE *
C                       *********************
C
C***********************************************************************
C
C      USAGE
```

```
C        CALL TRSLTE(IF,IB)
C
C     DESCRIPTION OF PARAMETERS
C     IF - POINTS TO THE FIRST CHARACTER OF A STRING
C     IB - POINTS TO THE LAST CHARACTER OF THE SAME STRING.
C
C     PURPOSE
C    1)TRANSLATION OF AN ARITHMETIC OR LOGICAL STATEMENT FROM STANDARD
C     MATHEMATICAL NOTATION TO S-LANGUAGE.
C    2)TRANSLATION FROM THE S-LANGUAGE TO EARLY OPERATOR REVERSE
C     POLISH NOTATION.
C    3)SHORTING OF ARGUMENTS IN COMMUTATIVE OPERATORS IN LEXICOGRAPHIC
C     ORDER.
C    4)GENERATION OF TRIPLES.
C
C***********************************************************************
C
      SUBROUTINE TRSLTE(IF,IB)
      INTEGER*2 DATA1(72),DATA2(68),BLANK/' '/,ALPHA
     1/'A'/,NINE/'9'/,STAR/'*'/,A9,LPAR/'('/,RPAR/')'/,R,
     2ISGN/'='/,AUXLRY(10),MINUS/'-'/,PERIOD/'.'/,BEG/'<'/,END/'>'
     3/,HRCH(24),POLISH(68),OPRSTK(68),S    ,P     ,O,TRIPLE(3,600)
     5,ITRPLE,SEQ( 900),ISEQ,FL( 80),IFL,ZERO/'0'/
     6,FIRST(10),LAST(10),SUBSTR,ARRAY(60)
      REAL*8 OPRTR(24),OPRND( 80),TEMPRY
      COMMON /A1/DATA1/A5/TRIPLE,ITRPLE,SEQ,ISEQ/A6/OPRND,N3/A7/FL,IFL
     1/A9/OPRTR/A10/HRCH
      LOGICAL*1 L1(144),L2(8),BL/' '/,PER/'.'/
      EQUIVALENCE(DATA1,L1),(TEMPRY,L2)
      S=1
      A9=0
      P=69
      O=69
C*******************************************************************
```

```
C*****INITIALIZATION
      N1=0
      J=0
      I=IF-1
      IND=0
      K=IB
6     IF(I.EQ.IB)GO TC 62
      I=I+1
C*****HAVE WE AN OPERATOR OR AN OPERAND?
      IF(CATA1(I).LT.0)GCTO 10
      IF(DATA1(I).NE.PERIOD)GO TO 200
      IF(CATA1(I+1).GE.ZERO)GO TO 1C
 200  J=J+1
C*****WHERE ARE WE COMING FROM?
      IF(A9.NE.2)GO TO 12
C*****LOAD CCNSTRUCTED VARIABLE TO THE OPRND TABLE.
      IF(N3.EQ.0)GC TC 74
      DO 72 II=1,N3
      IF(OPRND(II).NE.TEMPRY)GO TO 72
      DATA2(J)=II+100
      GO TO 73
72    CONTINUE
74    N3=N3+1
      OPRND(N3)=TEMPRY
      DATA2(J)=N3+100
73    J=J+1
      N1=C
C*****HAVE WE AN ARRAY ELEMENT?
      IF(CATA1(I).NE.LPAR)GO TO 12
      DATA2(J)=24
      L=I+1
      R=0
C*****IF YES FIND THE RIGHT PARENTHESIS.
      DO 13 K1=L,K
```

```
      IF(CATA1(K1).EQ.RPAR.AND.R.EQ.0)GO TO 14
      IF(CATA1(K1).EQ.LPAR)R=R+1
      IF(DATA1(K1).EQ.RPAR)R=R-1
13    CONTINUE
C*****PUT THE POSITION OF THIS RIGHT PARENTHESIS TO THE AUXILIARY STACK.
14    IND=IND+1
      AUXLRY(IND)=K1
61    A9=1
      GO TC 6
C*****HAVE WE AN EXPONENTIATION OPERATCR?
12    IF(CATA1(I).NE.STAR)GO TO 11
      K2=I+1
      IF(DATA1(K2).NE.STAR)GO TO 60
      DATA2(J)=23
      I=I+1
      GO TC 61
6C    DATA2(J)=20
      GO TO 61
C*****HAVE WE UNARY MINUS?
11    IF(DATA1(I).NE.MINUS)GO TO 18
      K3=I-1
      IF(CATA1(K3).NE.ISON.AND.CATA1(K3).NE.LPAR.AND.DATA1(K3).NE.STAR)GO TO 181
     2O TC 181
      DATA2(J)=22
      GO TO 61
181   DATA2(J)=18
      GO TO 61
C*****HAVE WE A RIGHT PARENTHESIS?IF YES SEARCH AUXILIARY TABLE.
18    IF(CATA1(I).NE.RPAR)GO TO 191
      IF(IND.EQ.0)GO TC 102
      IF(AUXLRY(IND).NE.I)GO TO 102
101   DATA2(J)=17
      IND=IND-1
      GO TO 61
```

```
1C2   DATA2(J)=4
      GO TO 61
C*****TEST FOR LOGICAL OPERATORS.
C*****INITIALIZATION CF TEMPRY.
191   DO 94 K8=1,8
94    L2(K8)=BL
      IF(CATA1(I).NE.PERIOD)GO TO 20
      L2(1)=PER
      J1=1
19    I=I+1
      J1=J1+1
      N4=2*I-1
      L2(J1)=L1(N4)
      IF(CATA1(I).NE.PERIOD)GO TO 19
      GO TO 21
20    N4=2*I-1
      L2(1)=L1(N4)
 21   DO 22 J2=1,24
      IF(TEMPRY.EQ.OPRTR(J2))GO TO 23
22    CONTINUE
23    DATA2(J)=J2
      GO TO 61
10    N1=N1+1
      N2=2*I-1
      IF(A9.EQ.2)GO TC 43
      DO 42 K8=1,8
42    L2(K8)=BL
43    L2(N1)=L1(N2)
      A9=2
      GO TO 6
C*****TRANSLATION TC REVERSE POLISH.
62    CONTINUE
81    K=0
C*****IS IT AN OPERAND OR AN OPERATOR?
```

```
      IF(DATA2(S).GT.1C0)GO TO 91
C*****IS IT A RIGHT PARENTHESIS OR  AN END MARK?
      IF(CATA2(S).NE.4.ANC.CATA2(S).NE.5)GO TO 31
C******IF WE FINISHED GET OUT.
      IF(DATA2(S).EQ.5.AND.O.EQ.68)GO TO 103
C*****DISCARD CURRENT VALUES OF SOURCE AND OPERATOR STACK.
41    S=S+1
      O=O+1
      GO TO 71
C*****IS IT A RIGHT SQUARE BRACKET?
31    IF(DATA2(S).NE.17)GC TO 51
      K=1
      GC TC 121
C*****IS IT A LEFT SQUARE BRACKET?
51      IF(DATA2(S).NE.24)GO TO 63
      O=O-1
      OPRSTK(C)=CATA2(S)
      S=S+1
      O=O-1
      OPRSTK(C)=3
      GO TO 81
63    O=O-1
      OPRSTK(C)=DATA2(S)
      S=S+1
      GO TO 81
91    P=P-1
      PCLISH(P)=CATA2(S)
      S=S+1
C*****TEST FOR PRIORITY.
71    IF(FRCH(OPRSTK(C)).LT.HRCH(DATA2(S)))GO TO 81
121   P=P-1
      POLISH(P)=OPRSTK(O)
      O=O+1
      IF(K.EQ.0)GC TO 71
```

```
      K=0
      GC TC 41
1C3   CONTINUE
C*****SHORTING OF RESULTED POLISH STRING.
C     WE CONSIDER ONE  OPERATOR AT A TIME FROM LEFT TO RIGHT.DURING THE
C     CCMPARISON OF TWC OPERANDS WE COMPAIRE THE FIRST VARIABLE OF EACH.
C*****TWO POINTERS POINT TO THE FIRST AND THE LAST DIGIT OF EVERY OPERAND.
      SUBSTR=0
      IP=69
C*****THE ORIGIN OF THE PCLISH STACK IS AT THE END OF AN ARRAY.
142   IP=IP-1
      IF(IP.EQ.P-1)GO TO 150
      K1=0
      IF(POLISH(IP).GT.100)GO TO 142
C*****IT IS AN OPERATOR.
      IF(PCLISH(IP).EQ.22)K1=1
      IF(POLISH(IP).EQ.8)K1=1
      IP=IP+1
      K=0
C*****IN THE FOLLOWING DC-LOOP WE FINC THE LIMITS OF THE TWO OPERANDS OF
C     CURRENT OPERATOR.
      DO 143 I=1,2
C*****IF THE CRIGINAL OPERATOR IS THE UNARY MINUS DO NOT CONTINUE TO THE SECOND
C     OPERAND.
      IF(K1.EQ.1.AND.I.EQ.2)GO TO 161
C*****IF THE FIRST CHARACTER IS AN OPERATOR DO NOT CREATE A NEW ENTRY TC 'FIRST'
C     AND 'LAST' BECAUSE  HAS ALREADY BEEN DONE.
      IF(POLISH(IP).LT.100)GO TO 144
C*****BUT IF THE FIRST CHARACTER IS AN OPERAND DC THAT.
      SUBSTR=SUBSTR+1
      FIRST(SUBSTR)=IP
      LAST(SUBSTR)=IP
C*****PREPARE IP FOR THE NEXT STEP.
      IP=IP+1
```

```
      GO TO 143
C*****IF WE ARE IN THE FIRST STEP PREPARE IP FOR THE NEXT STEP.
144   IF(I.EQ.2)GO TO 143
      IP=FIRST(SUBSTR)+1
143   CONTINUE
C*****BECAUSE OF THE WAY WE INFORMED ARRAYS 'FIRST' AND 'LAST' WE DO NOT
C     KNOW WHICH ENTRY(THE LAST OR THE ONE BEFORE THE LAST)CORRESPONDS
C     TO THE NEAREST TO THE OPERATOR OPERAND(OR BETTER RESULT).
C     WE LEARN THAT WITH THE NEXT TEST.
      IF(FIRST(SUBSTR-1).LT.FIRST(SUBSTR))K=1
C*****THE VALUE OF M IS THE OPERATOR UNDER CONSIDERATION.
      M=POLISH(LAST(SUBSTR-K)-1)
C*****THE VALUE OF J2 POINTS THE ENTRY OF THE NEAREST INSTRUCTION TO THE CURRENT
C     OPERATOR.
161   J2=SUBSTR-K
      I2=LAST(J2)
C*****IF THE CURRENT OPERATOR IS THE UNARY MINUS CHANGE THE VALUE OF LAST(SUBSTR
C     ) IN ORDER TO POINT THE END OF THIS INSTRUCTION AND REPEAT ALGORITHM.
      IF(K1.EQ.1)GO TO 160
C*****LET'S FIND OUT SOME INFORMATION CONCERNING THE OTHER OPERAND.
      J1=SUBSTR+K-1
      N1=FIRST(J1)
C*****IS IT A COMMUTATIVE OPERATOR?
      IF(M.EQ.19)GO TO 145
      IF(M.EQ.6)GO TO 145
      IF(M.EQ.7)GO TO 145
      IF(M.NE.20)GO TO 149
145   M2=POLISH(FIRST(J2))
      M1=POLISH(FIRST(J1))
C*****IF IT IS A COMMUTATIVE OPERATOR COMPAIRE THE FIRST VARIABLES OF  OUR
C     TWO OPERANDS.
      IF(OPRND(M1-100).GE.OPRND(M2-100))GO TO 149
C*****PERMUT THE TWO OPERANDS.
      I1=FIRST(J2)
```

```
      IND=1
      DO 146  J=I2,I1
      ARRAY(IND)=POLISH(J)
146   IND=IND+1
      ARRAY(IND)=0
      N2=LAST(J1)
      IND=I2
      DO 147 J=N2,N1
      POLISH(IND)=POLISH(J)
147   IND=IND+1
      J=0
148   J=J+1
      IF(ARRAY(J).EQ.0)GO TO 149
      POLISH(IND)=ARRAY(J)
      IND=IND+1
      GO TO 148
149   SUBSTR=SUBSTR-1
      FIRST(SUBSTR)=N1
160   LAST(SUBSTR)=I2-1
      IP=I2-1
      GO TO 142
150   CONTINUE
C*****TRANSLATION TO TRIPLES.
C*****WE SCAN THE POLISH STACK FROM THE BEGINNING AND WHEN AN OPERATOR IS FOUND
C      WE CREATE A NEW TRIPLE.
C*****IN THE SAME TIME WE CHECK IF THE CURRENT TRIPLE WAS ENCOUNTERED BEFORE IN
C      THE BLOCK.
C*****IF THIS IS TRUE WE DO NOT CREATE A NEW ENTRY TO THE TRIPLE TABLE.
C      ONLY THE SEQ TABLE IS INFORMED IN THIS CASE.
      IP=69
151   IP=IP-1
C*****ARE WE DONE?
      IF(IP.EQ.P-1)GO TO 154
      IF(POLISH(IP).EQ.0)GO TO 151
```

```
C*****OPERATOR OR OPERAND?
      IF(PCLISH(IP).GT.100)GO TO 151
C*****CREATE TRIPLE.
      TRIPLE(3,ITRPLE)=POLISH(IP)
      K=IP
      DO 152 I=1,2
      IF(I.EQ.1)GO TO 153
C*****IS IT A UNARY OPERATOR?
      IF(POLISH(IP).EQ.22)GO TO 155
      IF(POLISH(IP).EQ.8)GO TO 155
153   K=K+1
      IF(PCLISH(K).EQ.0)GO TO 153
      TRIPLE(3-I,ITRPLE)=POLISH(K)
152   POLISH(K)=0
155   M=IFL-1
      L=FL(M)+1
      IF(L.EQ.ISEQ)GO TO 156
C*****LET'S SEE IF THERE IS ANY OTHER SIMILAR TRIPLE WITH UNDEFINED
C*****OPERANDS IN THE PATH BETWEEN THESE TRIPLES.
      I=ISEQ
157   I=I-1
      DO 162 IA=1,3
      IF(TRIPLE(IA,SEQ(I)).NE.TRIPLE(IA,ITRPLE))GO TO 163
162   CONTINUE
      IF(TRIPLE(3,ITRPLE).NE.3)GO TO 158
      IF(K.EQ.68)GO TO 156
      IF(POLISH(68).NE.SEQ(I)+1000)GO TO 158
      IF(POLISH(P).NE.11)GO TO 158
163   IF(TRIPLE(3,SEQ(I)).NE.11.AND.TRIPLE(3,SEQ(I)).NE.25)GO TO 164
      M=TRIPLE(1,SEQ(I))
      IF(M.LT.1000)GO TO 165
      IF(TRIPLE(3,ITRPLE).NE.3)GO TO 164
      M=TRIPLE(1,M-1000)
      IF(M.EQ.TRIPLE(1,ITRPLE))GO TO 156
```

```
      GO TO 164
165   IF(M.EQ.TRIPLE(1,ITRPLE))GO TO 156
      IF(M.EQ.TRIPLE(2,ITRPLE))GO TO 156
164   IF(I.NE.L)GO TO 157
156   POLISH(K)=ITRPLE+1000
      SEQ(ISEQ)=ITRPLE
      ITRPLE=ITRPLE+1
      GO TO 159
158   POLISH(K)=SEQ(I)+1000
      SEQ(ISEQ)=SEQ(I)
      TRIPLE(1,ITRPLE)=0
159   ISEQ=ISEQ+1
      POLISH(IP)=0
      GO TO 151
 154  RETURN
      END
C                        ******************
C                        * SUBROUTINE HELP *
C                        ******************
C
C*****************************************************************************
C
C     USAGE
C      CALL HELP(I1,I2,I3,I4,I5,K)
C
C     DESCRIPTION OF PARAMETERS
C     I1 - THE FIRST ENTRY OF THE TRIPLE TO BE INSERTED INTO THE
C      TRIPLE TABLE.
C     I2 - THE SECOND ENTRY OF THE ABOVE TRIPLE.
C     I3 - THE THIRD ENTRY OF THE SAME TRIPLE.IF I3=0 THEN NO TRIPLE
C      WILL BE INSERTED.
C     I4 - POINTER TO THE BEGINNING OF A STRING TO BE LOADED IN THE WORK
C      TABLE.BUT IF I4=0 THEN NO STRING IS TO BE LOADED.
C     I5 - THE END OF THE ABOVE MENTIONED STRING.
```

```
C      K - IS A CODE NUMBER.IF K=0 THERE IS NC CPERATICN TO BE PERFORMED.
C       IF K=1 THE CURRENT BLOCK WILL BE CLOSED ,A NEW CNE WILL BE
C       INITIALIZED AND THE SECOND BLOCK IS CONSIDERED AS A SUCCESSOR
C       OF THE FIRST.
C       IF K=2 THE CURRENT BLOCK WILL BE CLOSED A NEW CNE WILL BE CREATED
C       BUT NC LINKAGE WILL BE PERFORMED.
C       IF K=3 THE CURRENT BLOCK WILL BE CLOSED,A NEW ONE WILL BE INI-
C       TIALIZED AND A LINKAGE WILL BE PERFORMED BETWEEN FIRST-SECOND AND
C       FIRST-THIRD.
C
C***********************************************************************
C
      SUBROUTINE HELP(I1,I2,I3,I4,I5,K)
      INTEGER*2 IRES,TRIPLE(3, 600),ITRPLE,SEQ( 900),ISEQ,ILBL,SUC( 6CC)
     1,ISUC,FL( 80),IFL,INPUT(72),BLANK/' '/,RES(1600),MRK/Z5240/
      COMMON/A1/INPUT/A3/IRES,RES/A5/TRIPLE,ITRPLE,SEQ,ISEQ/A7/FL,IFL/A8/ILBL,SU
     1/ILBL,SUC,ISUC
      IF(I4.EQ.0)GO TO 2
C*****LOAD LINE.
      DC 1 J=I4,I5
      RES(IRES)=INPUT(J)
 1    IRES=IRES+1
      RES(IRES)=MRK
      IRES=IRES+1
 2    IF(I3.EQ.0)GO TO 3
C*****LOAD TRIPLE.
      TRIPLE(3,ITRPLE)=I3
      TRIPLE(2,ITRPLE)=I2
      TRIPLE(1,ITRPLE)=I1
      SEQ(ISEQ)=ITRPLE
      ISEQ=ISEQ+1
      ITRPLE=ITRPLE+1
 3    IF(K.EQ.0)GO TO 6
C*****PERFORM FLOW OPERATIONS.
```

```
 4    ILBL=ILBL+1
      IF(K.EQ.1)GO TO 5
      SUC(ISUC)=IFL
      ISUC=ISUC+1
      IF(K.EQ.2)GO TO 5
      SUC(ISUC)=IFL+1
      ISUC=ISUC+1
 5    ISUC=ISUC+1
      SUC(ISUC)=IFL
      ISUC=ISUC+1
      FL(IFL)=ISEQ
      IFL=IFL+1
      ISEQ=ISEQ+1
 6    RETURN
      END
C                          ********************
C                          * SUBROUTINE FNDDLR *
C                          ********************
C
C*******************************************************************************
C
C     USAGE
C      CALL FNDDLR(I4,I5,I1,I2,I3)
C
C     DESCRIPTION OF  PARAMETERS
C     I4 - POINTS TO THE POSITION FROM WHICH SCANNING SHOULD START
C      AND RETURNS A NUMBER POINTING TO THE FIRST OCCURRENCE OF I1
C      FOLLOWED BY I2.
C     I5 - RETURNS A NUMBER POINTING TO THE FIRST OCCURRENCE OF I3.
C     I1 - IS A CHARACTER IN EBCDIC FORM.IF IT IS EQUAL TO ZERO,I1,I2
C      ARE IGNORED.
C     I2 - IS A CHARACTER IN EBCDIC FORM.IT MUST  FOLLOW CHARACTER I1
C      IN ORDER TO BE A MATCING.IF I2=0 THE SECOND CHARACTER IS CONSI-
C      DERED BY DEFAULT AS AN ALPHANUMERICAL CHARACTER.
```

```
C     I3 - IS A CHARACTER IN EBCDIC FORM.
C
C***********************************************************************
C
      SUBRCUTINE FNDDLR(I4,I5,I1,I2,I3)
      INTEGER*2 I1,I2,I3,INPUT(72)
      COMMON/A1/INPUT
      IF(I1.EQ.0)GO TC 4
      DO 1 I=I4,72
      IF(INPUT(I).NE.I1)GO TO 1
      IF(I2.EQ.0)GO TO 2
      IF(INPUT(I+1).NE.I2)GO TO 1
 3    I4=I
      GO TO 4
 2    IF(INPUT(I+1).GT.0)GO TO 1
      GO TO 3
 1    CONTINUE
 4    IF(I3.EQ.0)GO TO 7
      DC 5 J=I4,72
      IF(INPUT(J).EQ.I3)GO TO 6
 5    CONTINUE
 6    I5=J-1
 7    RETURN
      END
C                         *****************
C                         * FUNCTION NUMBER *
C                         *****************
C
C***********************************************************************
C
C     DESCRIPTION OF PARAMETERS
C     I - POINTS TO THE END OF A STRING OF NUMBERS
C
C     PURPCSE
```

```
C      THE SUBROUTINE FUNCTION NUMBER TRNSFORMS A SEQUENCE OF NUMBERS
C       FRCM EBCDIC TO ARITHMETIC FORM.
C      THE PROCESS STOPS WHEN A NON NUMERICAL CHARACTER IS FOUND.
C
C***********************************************************************
C
      INTEGER FUNCTION NUMBER*2(I)
      INTEGER*2 INPUT(72),NUMBER,OU/'C'/,ZERO/'0'/
      COMMON/A1/INPUT
      NUMBER=0
      K=0
1     I=I-1
      IF(I.EQ.0)GO TO 2
      IF(INPUT(I).GT.C)GC TO 2
      IF( INPUT(I).EQ.OU)GO TO 2
      NUM=(INPUT(I)-ZERO)/256
      NUMBER=NUMBER+NUM*10**K
      K=K+1
      GO TO 1
2     RETURN
      END
C                     ******************
C                     * SUBROUTINE DFED *
C                     ******************
C
C***********************************************************************
C
C      USAGE
C       CALL DFED(I1,IB)
C
C      DESCRIPTION OF PARAMETERS
C      I1 - IS THE ID NUMBER OF A STATEMENT
C      IB - POINTS TO THE POSITION OF THE BEGINNING OF THE STATEMENT.
C
```

```
C      PURPOSE
C      THE SUBROUTINE DFED SCANS STATEMENTS LIKE SUBROUTINE CALLS,
C       READ AND DO STATEMENTSIN ORDER TO FIND OUT WHICH VARIABLE(S)
C       IS DEFINED THERE.WHEN A DEFINED VARIABLE IS SPOTTED A NEW
C       TRIPLE IS INSERTED TO INDICATE THIS DEFINITION.
C
C*****************************************************************************
C
      SUBROUTINE DFED(I1,IB)
      INTEGER*2 INPUT(72),TRIPLE(3, 600),SEQ( 900),ITRPLE,ISEQ,
     1APA/'A'/,ISN/'='/,RPR/')'/,BLANK/' '/,ZERO/'0'/
      REAL*8 OPRND( 80),TMRY/' '/
      LOGICAL*1 BL/' '/,T(8)/8*' '/,L(144)
      COMMON/A1/INPUT/A5/TRIPLE,ITRPLE,SEQ,ISEQ/A6/OPRND,N3
      EQUIVALENCE (TMRY,T),(INPUT,L)
      I=IB+4
      IF(I1.EQ.8)I=IB+2
      IT=0
 1    I=I+1
C*****FINISHED?
      IF(INPUT(I).NE.BLANK)GO TO 19
C*****YES.BUT IS TMRY EMPTY?
      IF(IT.EQ.0)GO TO 16
C*****IS THE END OF THE LINE?
 19   IF(72-I)2,8,8
 2    IF(IT.EQ.0)GO TO 16
C*****FIND THE ID-POINTER OF THE VARIABLE.
 6    IF(N3.EQ.0)GO TO 4
      DO 3 J=1,N3
      IF(OPRND(J).EQ.TMRY)GO TO 5
 3    CONTINUE
 4    N3=N3+1
      OPRND(N3)=TMRY
      J=N3
```

```
C*****CREATE THE NEW TRIPLE.
 5    TRIPLE(1,ITRPLE)=100+J
      TRIPLE(3,ITRPLE)=25
      SEQ(ISEQ)=ITRPLE
      ISEQ=ISEQ+1
      ITRPLE=ITRPLE+1
 20   IT=C
      DO 7 J=1,8
 7    T(J)=BL
      GO TO 1
C*****IS CURRENT ENTRY OF INPUT A BREAKING CHARACTER?
 8    IF(INPUT(I).LT.0)GO TO 10
C*****IF IT IS THE EQUAL SIGN SKIP NEXT CHARACTERS UNTIL A RIGHT
C*****PARENTHESIS OR BLANK.
      IF(INPUT(I).NE.ISN)GO TO 9
      DO 11 I=I,72
      IF(INPUT(I).EQ.RPR)GO TO 9
      IF(INPUT(I).EQ.BLANK)GO TO 9
 11   CONTINUE
C*****IS TMRY EMPTY?
 9    IF(IT)1,1,6
 10   IF(IT.NE.0)GO TO 12
      IF(INPUT(I).GE.ZERO)GO TO 1
 12   IT=IT+1
      T(IT)=L(2*I-1)
      GO TO 1
 16   RETURN
      END
C                   ****************
C                   * SUBROUTINE ID *
C                   ****************
C
C**********************************************************************
C
```

```
C      USAGE
C       CALL ID(I1,I2)
C
C      DESCRIPTION OF PARAMETERS
C      I1 - ID OF A STATEMENT
C      I2 - '' ''      ''
C
C      PURPOSE
C      THIS SUBROUTINE RECOGNIZES EVERY STATEMENT.
C      IN CASE OF LOGICAL IF THE VALUE OF I2 IS NEEDED TOO.IN ADDITION
C       IF WE HAVE A LOGICAL OR ARITHMETIC IF STATEMENT THE SIGN OF
C       THE I1 OR I2 INDICATES IF THE STATEMENTS ARE OPTIMIZABLE OR NOT.
C
C
C***********************************************************************
C
C
      SUBROUTINE ID(I1,I2)
      INTEGER*2 INPUT(72),DECL(20),IDECL,DE/'D'/,LA/'L'/,SC(7,2)/'E','E'
     1,'I','I','C','C','R','X','Q','M','N','M','P','L'/,NOTA/1/,LPR/'('/,RPR/')'
     2,RPR/')'/,APT/''''/,BLANK/' '/,ISN/'='/,CM/','/,APA/'A'/,NINE/'9'/,
     3,IPASS/0/,IRES,ZERO/'0'/,GI/'G'/,ISW1/0/,ADD/'+'/,RES(1600),MRK/Z5240/
     4240/
      COMMON/A1/INPUT/A3/IRES,RES/A4/DECL,IDECL
      I2=0
      ISW2=1
      I=6
      ISW=0
 3    I=I+1
C*****FINISHED?
      IF(I.EQ.73)GO TO 15
C*****IF THE TEST BELLOW IS POSITIVE WE FAILLED TO RECOGNIZE A ZERO LEVEL
C*****EQUAL SIGN OR COMMA.
      IF(INPUT(I).EQ.BLANK)GO TO 15
```

```
C*****HAVE WE A LEFT PARENTHESIS?
      IF(INPUT(I).NE.LPR)GO TO 4
C*****LET'S TRY TO GET OUT OF IT.
      IS=31
      GO TO 5
C*****HAVE WE AN APOSTROPHE?
 4    IF(INPUT(I).NE.APT)GO TO 10
      IS=-30
 5    I=I+1
      DO 6 I=I,72
      IF(INPUT(I).NE.APT)GO TO 7
      IS=-IS
      GO TO 9
 7    IF(IS.LT.0)GO TO 6
      IF(INPUT(I).NE.LPR)GO TO 8
      IS=IS+1
      GO TO 6
 8    IF(INPUT(I).NE.RPR)GO TO 6
      IS=IS-1
 9    IF(IS.EQ.30)GO TO 3
 6    CONTINUE
 10   IF(ISW.EQ.1)GO TO 11
C*****WE ARE OUT.IS IT AN EQUAL SIGN?
      IF(INPUT(I).NE.ISN)GO TO 3
      ISW=1
      GO TO 3
C*****IS IT A COMMA?
 11   IF(INPUT(I).NE.CM)GO TO 3
      I1=8
      GO TO 12
 15   IF(ISW.EQ.0)GO TO 19
C*****IT IS AN ASSIGNMENT STATEMENT.IS IT ALONE OR WITH A LOGICAL
C*****IF STATEMENT?
 16   I=I-1
```

```
      IF(I.EQ.7)GO TO 17
      L=I-1
      IF(INPUT(L).EQ.RPR.AND.INPUT(I).LT.0)GO TO 18
      GO TO 16
17    I1=10
      GC TC 12
 18   ISW2=2
      I1=1C
      I=8
      GC TC 51
 19   IF(IPASS.EQ.1)GC TC 24
C*****SPECIFICATION TEST.
      IF(INPUT(7).EQ.DE)GO TO 21
      IF(INPUT(7).EQ.LA)GO TO21
      J=2
      DO 20 I=1,7
      IF(I.GT.2*J)J=J+1
      IF(INPUT(7).NE.SC(I,1))GC TC 20
      IF(INPUT(6+J).EQ.SC(I,2))GO TO 21
 20   CONTINUE
      IPASS=1
      GO TO 24
 21   IDECL=IDECL+1
C*****LOAD INPUT LINE TO THE WORK TABLE,
      DECL(IDECL)=IRES
      DO 22 I=7,72
      RES(IRES)=INPUT(I)
      IRES=IRES+1
 22   CONTINUE
      RES(IRES-1)=MRK
 23   I1=C
      I2=C
      GO TO 12
 24   I=7
```

```
C*****IS IT AN IF STATEMENT?
 25   IF(INPUT(I).NE.SC(3,1))GO TO 30
 51   IS=1
C*****IS IT OPTIMIZABLE?
      DO 26 I=I,72
      IF(INPUT(I)-ADD)26,40,39
39    IF(INPUT(I).EQ.RPR)GO TO 41
40    IS=-1
      GO TO 26
41    K=INPUT(I+1)
C*****WITH THE HELP OF THE TWO TESTS BELLOW WE SEPARATE LOGICAL AND
C*****ARITHMETIC IFS.
      IF(K.LT.ZERO)GO TO 27
      IF(K.LE.NINE)GO TO 29
 26   CONTINUE
 27   IF(ISW2.EQ.2)GO TO 50
      I=I+1
C*****CALL THE ROUTINE FROM THE BEGINNING TO FIND THE ID OF THE SECOND STATEMENT
      ISW1=1
      GO TO 25
 28   ISW1=0
 50   I2=I1
      I1=3*IS
      ISW2=1
      GO TO 12
29    I1=IS
C*****FROM NOW ON WE RECOGNIZE ALL OTHER STATEMENTS FROM THEIR INITIAL
C*****CHARACTERS.
      GO TO 36
 30   IF(INPUT(I).NE.GI)GO TO 32
      IF(INPUT(I+4).EQ.LPR)GO TO 31
      I1=2
      GO TO 36
 31   I1=4
```

```
      GO TO 36
 32   IF(INPUT(I+2).NE.LA)GO TO 33
      I1=5
      GO TO 36
 33   IF(INPUT(I).NE.SC(7,1))GO TO 34
      I1=6
      GO TO 36
 34   IF(INPUT(I+3).NE.SC(6,2))GO TO 35
      I1=7
      GO TO 36
 35   I1=9
 36   IF(ISW1.EQ.1)GO TO 28
 12   RETURN
      END
C                       *******************
C                       * SUBROUTINE REDUNT *
C                       *******************
C
C*********************************************************************
C
C     USAGE
C      CALL REDUNT(IBLOCK,IC)
C
C     DESCRIPTION OF PARAMETERS
C     IBLOCK - THE NUMBER OF THE BLOCK TO BE EXAMINED
C     IC - IC=1 IF REDUNDANT EXPRESSION ELIMINATION IS REQUIRED.
C          IC=2 IF REMOVAL OF INVARIANT INSTRUCTIONS IS REQUIRED.
C
C     PURPOSE
C     THIS SUBROUTINE IS USED FOR REDUNDANT INSTRUCTION ELIMINATION AND
C     REMOVAL OF INVARIANT INSTRUCTIONS.
C
C*********************************************************************
C
```

```
      SUBROUTINE REDUNT(IBLOCK,IC)
      INTEGER*2 TRIPLE(3,600),ITRPLE,SEQ(900),ISEQ,ILBL,SUC(600),ISUC,FL(80),IFL
     1(80),IFL,LBL(80),VAR,MAX,UNRST(10),IUNRST,ERS(10),IERS,PSB(20),IPSB,
     2B,DISTCE(80,80),MLT(7)
      REAL*8 OPRND(80),NEWVAR,IWTA/'I'/,NI/'N'/,NIKO
      LOGICAL*1 TAF/'T'/,NUM(10)/'0','1','2','3','4','5','6','7','8','9'/,
     1/,NA(8)/8*' '/,IW/'I'/,RESULT(30,80),COPY(80,80)
      EQUIVALENCE(NEWVAR,NA)
      COMMON /A5/TRIPLE,ITRPLE,SEQ,ISEQ/A6/OPRND,N3/A7/FL,IFL/A8/ILBL,SUC,ISUC
     1C,ISUC/A12/PSB,IPSB/A13/DISTCE,RESULT,COPY
      NA(2)=NUM(1)
      NA(1)=TAF
      IERS=0
      IUNRST=0
C*****FIND THE FIRST INSTRUCTION OF CURRENT BLOCK.
      ICST=FL(IBLOCK)
C*****FIND THE LAST INSTRUCTION OF CURRENT BLOCK.
      I2=FL(IBLOCK+1)-1
      IR=ICST
21    IR=IR+1
      IF(SEQ(IR)+1.EQ.0)GO TO 21
      IF(IR.GT.I2)GO TO 34
C*****WHAT PROCEDURE WAS REQUESTED?
      IF(IC.EQ.2)GO TO 56
      M=TRIPLE(3,SEQ(IR))
C*****ANY DEFINITIONS?
      IF(M.NE.11.AND.M.NE.25)GO TO 56
      M=TRIPLE(1,SEQ(IR))
C*****IS IT A DEFINITION OF AN ARRAY?
      IF(M.LT.1000)GO TO 57
      M=TRIPLE(1,M-1000)
 57   M=M-100
C*****INFORM COPY.
      COPY(M,IBLOCK)=.TRUE.
```

```
      GO TO 21
 56   IK=C
C*****FINISHED?
      IF(TRIPLE(3,SEQ(IR)).GT.24)GO TO 21
C*****IS THE FIRST ENTRY OF THE CURRENT INSTRUCTION POINTING TO ANOTHER
C*****INSTRUCTION?IF YES,IGNORE THE WHCLE INSTRUCTION.
      IF(TRIPLE(1,SEQ(IR)).GT.1000)GO TO 21
      IP=0
C*****BUT IF THE SECOND ENTRY OF THE CURRENT INSTRUCTION IS POINTING TO ANOTHER
C*****INSTRUCTION LET'S SEE IF THE POINTED INSTRUCTION IS ASSEMBLING
C*****A MULTIPLE-DIMENSIONAL ARRAY.
      IF(TRIPLE(2,SEQ(IR)).LT.1000)GO TO 30
      IF(TRIPLE(3,SEQ(IR)).NE.3)GO TC 21
      IMLT=0
      M=TRIPLE(2,SEQ(IR))-1000
 38   IF(TRIPLE(3,M).NE.16)GO TO 21
      IF(TRIPLE(2,M).GT.1000)GO TO 21
      IF(IC.EQ.1)GO TC 10
      IMLT=IMLT+1
      MLT(IMLT)=M
      DO 11 I=1,2
      IF(TRIPLE(I,M).GT.1000)GO TO 11
      K=IFL-1
      DO 12 J=1,K
      IF(.NOT.RESULT(ISUC,J))GO TO 12
      IF(COPY(TRIPLE(I,M)-100,J))GO TC 21
 12   CONTINUE
 11   CONTINUE
 10   IF(TRIPLE(1,M).GT.1000)GO TO 39
      IP=1
      GO TO 30
 39   M=TRIPLE(1,M)-1000
      GO TO 38
C*****IS THERE ANY BREAKING CHARACTER?
```

```
 30    M=TRIPLE(3,SEQ(IR))
       IF(M.LT.6)GO TO 1
       IF(M.LT.17)GO TC 25
C*****IF IT WAS A MULTIPLE-DIMENSICNAL ARRAY SKIP THE NEXT DO-LOOP.
 1     IF(IP.EQ.1)GO TO 42
C*****IN THIS DC-LCOP WE STOP THE PROCESS IF THERE IS NO MATCHING
C*****WITH THE INFORMATICN HOLD IN THE UNRST STACK.OTHERWISE WE CLASSIFY
C*****THE NEW TRIPLE ACCORDING TO THE CASE AND WE GO CN.
       DO 23 I=1,2
       M=TRIPLE(I,SEQ(IR))
       IF(M.EQ.0)GO TO 23
       IF(M.LT.1000)GO TO 23
       IF(IUNRST.EQ.0)WRITE(6,32)
       IF(SEQ(UNRST(IUNRST)).NE.M-1000)GC TC 24
       ISTTUS=1
       GC TC 52
24     IF(IUNRST.EQ.1)GC TC 25
       IF(SEQ(UNRST(IUNRST-1)).NE.M-1CCO)GC TO 25
       ISTTUS=2
       GC TC 52
23     CONTINUE
42     ISTTUS=3
C*****WHAT CPTION WAS REQUESTED?
 52    GO TO (58,59),IC
C*****LET'S SEE IF THERE IS ANY COMMON TRIPLE.
 58    K=IR+1
       IS=C
       DO 22 M=K,I2
       IF(SEQ(M)+1.EQ.C)GO TO 22
       IF(SEQ(IR).NE.SEQ(M))GO TO 22
       IF(IS.NE.0)GC TC 26
C*****SAVE THE POSITION OF THIS TRIPLE.
       NCTA=M
26     SEQ(M)=-1
```

```
      IS=IS+1
22    CONTINUE
      ILBL=ILBL+IS
C*****IF NO COMMON TRIPLE FOUND GET OUT.
      IF(IS.EQ.0)GO TC 25
      IF(IK.EQ.0)GO TO 3
      IF(IK.EQ.IS)GO TO 3
      IK=0
      GO TO 51
3     IK=IS
      GO TO 60
C*****ARE THE OPERANDS OF THE CURRENT INSTRUCTION INVARIANT?
 59   DO 71 IA=1,2
      IF(TRIPLE(IA,SEQ(IR)).GT.1000)GO TO 71
      K=IFL-1
      DO 70 IB=1,K
      IF(.NOT.RESULT(ISUC,IB))GO TO 70
      IF(CCPY(TRIPLE(IA,SEQ(IR))-100,IB))GO TO 25
 70   CONTINUE
C*****THEY ARE INVARIANT.PUT TRIPLE IN THE PSEUDO-BLOCK.
 71   CONTINUE
      IF(IP.EQ.0)GO TC 4
      DO 5 IA=1,IMLT
      IB=IMLT-IA+1
      IPSB=IPSB+1
  5   PSB(IPSB)=MLT(IB)
 4    IPSB=IPSB+1
      PSB(IPSB)=SEQ(IR)
C*****PREPAIRE PARAMETERS FOR THE NEXT STEP.
 60   GO TO (53,54,51),ISTTUS
53    ERS(IERS)=NOTA
      UNRST(IUNRST)=IR
      IR=IR+1
      IF(SEQ(IR)+1.EQ.0)GO TO 25
```

```
      IF(IR.GT.I2)GO TO 25
      IF(IC.EQ.2)GO TO 30
      M=TRIPLE(3,SEQ(IR))
      IF(M.NE.11.AND.M.NE.25)GO TO 30
      M=TRIPLE(1,SEQ(IR))
      IF(M.LT.1000)GO TO 62
      M=TRIPLE(1,M-1000)
 62   M=M-100
      COPY(M,IBLOCK)=.TRUE.
      GO TO 30
 54   IERS=IERS-1
      IUNRST=IUNRST-1
      GO TO 53
 51   IP=0
      IERS=IERS+1
      IUNRST=IUNRST+1
      GO TO 53
C*****SCANNING DISCONTINUED.IS ANYTHING IN THE UNRST STACK?
 25   IF(IUNRST.EQ.0)GO TO 21
      K=IUNRST
      K1=IERS
C*****CREATE A NEW VARIABLE.
 27   L=L+1
      N3=N3+1
      I=L/10
      NA(3)=NUM(I+1)
      J=L-I*10
      NA(4)=NUM(J+1)
      OPRND(N3)=NEWVAR
      TRIPLE(1,ITRPLE)=N3+100
      TRIPLE(2,ITRPLE)=SEQ(UNRST(K))+1000
      TRIPLE(3,ITRPLE)=11
      GO TO (66,63),IC
 66   M=ERS(K1)
```

```
C*****INFCRM COPY FOR THE NEW VARIABLE.
      COPY(N3,IBLOCK)=.TRUE.
C*****CREATE SPACE FOR NEW TRIPLE.
36    M=M-1
      SEQ(M+1)=SEQ(M)
      IF(M.NE.UNRST(K)+1)GO TO 36
      SEQ(UNRST(K)+1)=ITRPLE
       K2=UNRST(K)+2
      GO TO 64
 63   IPSB=IPSB+1
      COPY(N3,DISTCE(ISUC,1))=.TRUE.
      PSB(IPSB)=ITRPLE
      K2=UNRST(K)+1
 64   ITRPLE=ITRPLE+1
C*****ANY REFFERENCES TO THE ELIMINATED TRIPLE?
C*****PUT THE NEW VARIABLE INSTEAD.
      DO 28 I=K2,I2
      IF(SEQ(I)+1.EQ.0)GO TO 28
      IF(TRIPLE(1,SEQ(I)).EQ.SEQ(UNRST(K))+1000)TRIPLE(1,SEQ(I))=N3+100
      IF(TRIPLE(2,SEQ(I)).EQ.SEQ(UNRST(K))+1000)TRIPLE(2,SEQ(I))=N3+100
28    CONTINUE
29    K=K-1
      IF(IC.EQ.2)GO TO 65
      K1=K1-1
      DO 35 I=1,K1
      IF(SEQ(ERS(I))+1.EQ.0)GO TO 35
      ERS(I)=ERS(I)+1
35    CONTINUE
 65   IF(K.NE.0)GO TO 27
49    IUNRST=0
      IERS=0
      GO TO 21
32    FORMAT(' ','ERROR')
34    RETURN
```

```
          END
 STOP      0
                                                                    11.51.21

******************    ST.ANDREWS UNIVERSITY COMPUTING LABORATORY - 44MFT/SPOOLER END-OF-JOB RECORD    **************
****************** JOB NAME : NAP        DATE : 73247       START TIME : 11.49.25   ELAPSED : 00.01.57 **************
******************    CPU TIME : 00.00.49     WAIT TIME : CC.CO.CC      SYSTEM OVERHEAD : 00.00.22     **************
******************    PAGES PRINTED :    60     CARDS PUNCHED :     C     CARDS READ :  1888           **************
******************    A GRAND TOTAL OF         4121  INPUT/OUTPUT REQUESTS WERE HANDLED FOR THE JOB.   **************
******************    THE AVAILABLE CORE WAS :  80 K     OF WHICH AT LEAST   16 K WAS USED             **************
******************                THE APPROXIMATE COST OF THIS JOB WAS  £  1.13                       **************
******************              THIS JOB WAS EXECUTED IN THE FOREGROUND PARTITION                     **************
 THIS LINE PRINTED AT 11.55.59  ;
```

CHAPTER 5

THESIS RESULTS

This final chapter discusses a few examples of problem programs as run on the developed optimizer and summarizes the results of this thesis.

5.1 Examples

As it was mentioned in previous chapters, the optimizer accepts a program in FORTRAN and generates, after optimization, code in the same language. This fact implies that candidate instructions for elimination or removal should be exposed explicitly in the source program in order to be detected by the optimizing algorithm. The optimizer is not supposed to be used independently but rather as an additional pass on the output of some high level language translators which because of their nature produce inefficient code.

Most of the examples given in this section are, therefore, not realistic in terms of size or code quality. The primary intention here is to show that the developed algorithm "works" correctly with all possible data combinations, no matter if some of these combinations are not very likely to appear in real programs.

Example 1

The first example given involves the finding of the roots of a quadratic equation. In this small program there are three redundant subexpressions detected by the optimizer, namely- B, SQRT(B**2-4*A*C) and 2*A. Obviously the elimination of the first subexpression as redundant does not lead necessarily to an improvement since most computers have a fast "negate" command in their instruction set. In addition, in a good optimizing compiler the multiplication in T003 would be replaced by an addition and the exponentiation in T001 would be replaced by a multiplication (eliminating a possible subroutine call).

This example is one in which the optimization would probably not be worth the effort, because the original program does not require much computer time. If, however, the calculation of quadratic roots must be performed many times, the total time saved will make the effort well worth it.

Example 2

The source program in this example calculates the volumes of 50 cylinders and cones and the areas of 50 cicles, whose radius, height, varies from 0 to 10, 1.0 to 16.0, in 0.2, 0.3, increments respectively. This example demonstrates the usefulness of the sorting algorithm in the detection of redundant subexpressions and illustrates the action taken when a subexpression of a redundant expression is found redundant in a different place of the source program. More specifically, the expression:

HEIGHT*PI*RADIUS**2

is redundant in VCYL and VCONE but only a subexpression of it is redundant in VCICL. As it can be seen in the output, the above expression is broken down into two parts in order to provide the correct results in every position where redundancies were encountered.

Example 3

This example illustrates the actions taken by the optimizer in the two following cases.

(a) An expression does not change value between multiple occurrences of it; and

(b) the value of the expression is changed between two occurrences of that expression. In this example, the expression A/B is redundant in V and U but not in V1 because of the definition: A= V-U.

Example 4

The same principle is exemplified here as the one mentioned in the previous example, but the defined item is now an array element, C(I,I+1), and not a simple variable.

Example 5

The function of the source program in this example is the calculation of the product:

C=(A-B)x(A+B)

where A,B, and C are three square matrices. As we can see in the assignment statement labelled with the number one, the reference of the array element C(I,J) is not detected as redundant although no defining instruction appears in the Sequence Table between the two occurrences of the C(I,J).

Example 6

This example shows how code is moved from frequently executed areas to less frequently executed areas. The source program in this example consists of two versions of the same program. The loops in the first version are programmed with DO statements and in the second version with logical IFs, but as it can be easily confirmed from the output, this difference did not affect the final results.

Example 7

The source program in this example is a classical illustration of nested loops. It finds all the three-digit numbers that are equal to the sum of the cubes of their digits. The logical expression of the IF was expanded into triples because it contains more than one arithmetic operators. This example shows how code is moved as far as possible out from the frequently executed areas of a program.

Timing considerations of the original program and its optimized version are also given in order to offer a further measure of the improvements made on the source program.

Example 8

In this example the array element C(I) was first moved out of two different blocks of the region 3-4-5. Then the second occurrence of C(I) was found redundant in block 2 and it was, therefore, eliminated.

Example 9

The source "program" in this last example is simply a set of statements in an almost random order. This example illustrates the functions of the optimizer in a somewhat general case. Note that the expressions:

B(N,L,R)-7

and (B(N,L,R)-7)**3

ended up in the first block where they were examined again for redundant subexpressions.

EXAMPLE 1

INPUT PROGRAM

```
      READ(5,1)A,B,C
1     FORMAT(3F10.5)
      R1=(-B+SQRT(B**2-4*A*C))/(2*A)
      R2=(-B-SQRT(B**2-4*A*C))/(2*A)
      WRITE(6,2)R1,R2
2         FORMAT(' R1=',F10.5,'R2=',F10.5)
       STOP
```

OUTPUT PROGRAM

```
C BLOCK  1
      READ(5,1)A,B,C
1     FORMAT(3F10.5)
      T002=-B
      T001=SQRT(B**2-A*4*C)
      T003=A*2
      R1=(T002+T001)/T003
      R2=(T002-T001)/T003
      WRITE(6,2)R1,R2
2     FORMAT(' R1=',F10.5,'R2=',F10.5)
      STOP
```

```
THE IMPROVEMENTS(?) MADE IN THE PROGRAM ARE
   7  COMMON SUB-EXPRESSIONS ELIMINATED
   0  INSTRUCTIONS MOVED
```

EXAMPLE 2

2

INPUT PROGRAM

```
      REAL PI/3.14159/,RADIUS/0.0/,HEIGHT/1.0/
      DO 1 I=1,50
      RADIUS=RADIUS+0.2
      HEIGHT=HEIGHT+0.3
      VCYL=RADIUS**2*PI*HEIGHT
      VCICL=PI*RADIUS**2
      VCONE=PI*RADIUS**2*HEIGHT/3.
1     WRITE(6,2)VCYL,VCICL,VCONE
2     FORMAT(' VCYL=',F10.5,' VCICL=',F10.5,' VCONE=',F10.5)
      STOP
```

OUTPUT PROGRAM

```
      REALPI/3.14159/,RADIUS/0.0/,HEIGHT/1.0/
C BLOCK  1
      DO1I=1,50
      RADIUS=RADIUS+0.2
      HEIGHT=HEIGHT+0.3
      T0C2=PI*RADIUS**2
      T0C1=HEIGHT*T0C2
      VCYL=T001
      VCICL=T002
      VCONE=T001/3.
C BLOCK  2
    1 WRITE(6,2)VCYL,VCICL,VCONE
C BLOCK  3
 2    FORMAT(' VCYL=',F10.5,' VCICL=',F10.5,' VCONE=',F10.5)
      STOP
```

```
THE IMPROVEMENTS(?) MADE IN THE PROGRAM ARE
   5  COMMON SUB-EXPRESSIONS ELIMINATED
   0  INSTRUCTIONS MOVED
```

EXAMPLE 3

3

INPUT PROGRAM

```
      READ(5,1)A,B
    1 FORMAT(2F10.5)
      V=(A/B+1)/(A/B-1)
      U=(A/B)**2
      A=V-U
      V1=(A/B+2)**3
      STOP
```

OUTPUT PROGRAM

```
C BLOCK  1
      READ(5,1)A,B
    1 FORMAT(2F10.5)
      T001=A/B
      V=(T001+1)/(T001-1)
      U=T001**2
      A=V-U
      V1=(A/B+2)**3
      STOP
```

```
THE IMPROVEMENTS(?) MADE IN THE PROGRAM ARE
    2  COMMON SUB-EXPRESSIONS ELIMINATED
    0  INSTRUCTIONS MOVED
```

EXAMPLE 4

INPUT PROGRAM

```
      REAL C(30,30),Y/10.0/
      READ(5,1)C
1     FORMAT(30I2)
      DO 2 I=1,29
      X=C(I,I+1)+Y
      X1=(C(I,I+1)+Y)*3.
      C(I,I+1)=X+X1
      X2=(C(I,I+1)+Y)**6
2     CONTINUE
      STOP
```

FLOW ANALYSIS

```
 BLOCK     SUCCESSORS
       1          2,
       2          3,
       3          2,  4,
       4 IS AN EXIT NODE
```

```
THE UNIQUE CLOSED PATHS OF LENGTH LESS THAN OR EQUAL  TO L ARE
 FOR L=  1
   NONE
 FOR L=  2
       2-  3-
 FOR L=  3
   NONE
 FOR L=  4
   NONE
```

```
 THE OPTIMIZABLE STRONGLY-CONNECTED REGIONS ARE
       2-  3-
```

```
THE IMMEDIATE PREDECESSORS OF THE ABOVE REGIONS ARE
  1
```

OUTPUT PROGRAM

```
      REALC(30,30),Y/10.0/
C BLOCK  1
      READ(5,1)C
 1    FORMAT(30I2)
C BLOCK  2
      DO 2I=1,29
      T001=I+1
      T002=C(I,T001)+Y
      X=T002
      X1=T002*3.
      C(I,T001)=X+X1
      X2=(C(I,T001)+Y)**6
C BLOCK  3
    2 CONTINUE
C BLOCK  4
      STOP
```

```
THE IMPROVEMENTS(?) MADE IN THE PROGRAM ARE
   5  COMMON SUB-EXPRESSIONS ELIMINATED
   0  INSTRUCTIONS MOVED
```

EXAMPLE 5

INPUT PROGRAM

```
      REAL A(50,50)/2500*1/,B(50,50)/2500*2/,C(50,50)
      DO 1 I=1,50
      DO 1 J=1,50
      C(I,J)=0.0
      DO 1 K=1,50
 1    C(I,J)=C(I,J)+(A(I,K)-B(I,K))*(A(I,K)+B(I,K))
      WRITE(6,2)((C(I,J),J=1,50),I=1,50)
 2    FORMAT(' ',50I2)
      STOP
```

FLOW ANALYSIS

```
BLOCK     SUCCESSORS
     1         2,
     2         3,
     3         4,
     4         1,  2,  3,  5,
     5 IS AN EXIT NODE
```

```
THE UNIQUE CLOSED PATHS OF LENGTH LESS THAN OR EQUAL  TO L ARE
 FOR L=  1  .
   NONE
 FOR L=  2
      3-  4-
 FOR L=  3
      2-  3-  4-
 FOR L=  4
      1-  2-  3-  4-
 FOR L=  5
   NONE
```

```
THE OPTIMIZABLE STRONGLY-CONNECTED REGIONS ARE
     3-  4-
     2-  3-  4-
```

```
THE IMMEDIATE PREDECESSORS OF THE ABOVE REGIONS ARE
  2
  1
```

CUTPUT PROGRAM

```
      REALA(50,50)/2500*1/,B(50,50)/2500*2/,C(50,50)
C BLOCK  1
      DO1I=1,50
C BLOCK  2
      DO1J=1,50
      C(I,J)=0.0
C BLOCK  3
      DO1K=1,50
C BLOCK  4
      T001=A(I,K)
      T002=B(I,K)
    1 C(I,J)=(T001-T002)*(T001+T002)+C(I,J)
C BLOCK  5
      WRITE(6,2)((C(I,J),J=1,50),I=1,50)
 2    FORMAT(' ',50I2)
      STOP
```

```
THE IMPROVEMENTS(?) MADE IN THE PROGRAM ARE
   2  COMMON SUB-EXPRESSIONS ELIMINATED
   0  INSTRUCTIONS MOVED
```

EXAMPLE 6

INPUT PROGRAM

```
      INTEGER*2 A(50,50,50)
      X=5.
      Y=10.0
      DO 1 I=1,50
      DO 1 J=1,50
      DO 1 K=1,50
1     A(I,J,K)=(I**2+J**3+K**4)/(X+Y)
      X=50.0
      Y=30.0
      I=1
2     J=1
3     K=1
4     A(I,J,K)=(I**2+J**3+K**4)/(X+Y)
      K = K + 1
      IF(K.LE.50)GO TO 4
      J=J+1
      IF(J.LE.50)GO TO 3
      I=I+1
      IF(I.LE.50)GO TO 2
      STOP
```

FLOW ANALYSIS

```
BLOCK     SUCCESSORS
      1        2,
      2        3,
      3        4,
      4        5,
      5        2,  3,  4,  6,
      6        7,
      7        8,
      8        9,
      9        10, 11,
     10        9,
     11        12, 13,
     12        8,
     13        14, 15,
     14        7,
     15 IS AN EXIT NODE


THE UNIQUE CLOSED PATHS OF LENGTH LESS THAN OR EQUAL  TO L ARE
 FOR L=  1
   NONE
 FOR L=  2
        4-  5-
        9- 10-
 FOR L=  3
        3-  4-  5-
 FOR L=  4
        2-  3-  4-  5-
        8-  9- 11- 12-
 FOR L=  5
   NONE
 FOR L=  6
        7-  8-  9- 11- 13- 14-
        7-  8-  9- 10- 11- 12- 13- 14-
        8-  9- 10- 11- 12-
 FOR L=  7
   NONE
 FOR L=  8
        7-  8-  9- 10- 11- 13- 14-
 FOR L=  9
   NONE
 FOR L= 10
   NONE
 FOR L= 11
   NONE
 FOR L= 12
   NONE
 FOR L= 13
   NONE
 FOR L= 14
   NONE
 FOR L= 15
   NONE
```

THE OPTIMIZABLE STRONGLY-CONNECTED REGIONS ARE

```
      4-  5-
      9- 10-
      3-  4-  5-
      2-  3-  4-  5-
      8-  9- 10- 11- 12-
      7-  8-  9- 10- 11- 12- 13- 14-
```

THE IMMEDIATE PREDECESSORS OF THE ABOVE REGIONS ARE

```
 3
 8
 2
 1
 7
 6
```

OUTPUT PROGRAM

```
      INTEGER*2A(50,50,50)
C BLOCK  1
      X=5.
      Y=10.0
      T007=X+Y
C BLOCK  2
      DO1I=1,50
      T005=I**2
C BLOCK  3
      DO1J=1,50
      T001=T005+J**3
C BLOCK  4
      DO1K=1,50
C BLOCK  5
    1 A(I,J,K)=(TC01+K**4)/TC07
C BLOCK  6
      X=50.0
      Y=30.0
      I=1
      T010=X+Y
C BLOCK  7
    2 J=1
      T008=I**2
C BLOCK  8
    3 K=1
      T003=T008+J**3
C BLOCK  9
    4 A(I,J,K)=(TC03+K**4)/TC10
      K=K+1
C BLOCK 10
      IF(K.LE.50)GOTO4
C BLOCK 11
      J=J+1
C BLOCK 12
      IF(J.LE.50)GOTO3
C BLOCK 13
      I=I+1
C BLOCK 14
      IF(I.LE.50)COTO2
C BLOCK 15
```

```
      STOP


 THE IMPROVEMENTS(?) MADE IN THE PROGRAM ARE
    0  COMMON SUB-EXPRESSIONS ELIMINATED
   24  INSTRUCTIONS MOVED
```

EXAMPLE 7

INPUT PROGRAM

```
      INTEGER*2 A,B,C
      DO 1 A=1,9
      DO 1 B=0,9
      DO 1 C=0,9
      IF(100*A+10*B+C.EQ.A**3+B**3+C**3)WRITE(6,2)A,B,C
1     CONTINUE
2     FORMAT(' A=',I4,'B=',I4,'C=',I4)
      STOP
```

FLOW ANALYSIS

```
BLOCK     SUCCESSORS
      1          2,
      2          3,
      3          4,   5,
      4          5,
      5          1,   2,   3,   6,
      6 IS AN EXIT NODE


THE UNIQUE CLOSEC PATHS OF LENGTH LESS THAN CR EQUAL  TC L ARE
 FCR L=  1
   NCNE
 FCR L=  2
        3-  5-
 FOR L=  3
        2-  3-  5-
        2-  3-  4-  5-
        3-  4-  5-
 FOR L=  4
        1-  2-  3-  5-
        1-  2-  3-  4-  5-
 FCR L=  5
   NCNE
 FOR L=  6
   NONE


 THE OPTIMIZABLE STRONGLY-CCNNECTED REGICNS ARE
        3-  4-  5-
        2-  3-  4-  5-


THE IMMECIATE PREDECESSORS OF THE ABOVE REGICNS ARE
   2
   1
```

OUTPUT PROGRAM

```
      INTEGER*2A,B,C
C BLOCK  1
      DO1A=1,9
      T003=A*100
      T004=A**3
C BLOCK  2
      DO1B=0,9
      T001=T003+B*10
      T002=T004+B**3
C BLOCK  3
      DO1C=0,9
C BLOCK  4
      IF(T001+C.EQ.T002+C**3)WRITE(6,2)A,B,C
C BLOCK  5
    1 CONTINUE
C BLOCK  6
 2    FORMAT(' A=',I4,'B=',I4,'C=',I4)
      STOP
```

```
THE IMPROVEMENTS(?) MADE IN THE PROGRAM ARE
   0  COMMON SUB-EXPRESSIONS ELIMINATED
  12  INSTRUCTIONS MOVED
```

```
FORTRAN IV G LEVEL 44PS V 1.0          GMAIN44          DATE = 73218  TI

0001            INTEGER*2 A,B,C
0002            CALL CLCPTM(I1)
0003            DO 1 A=1,9
0004            DO 1 B=0,9
0005            DO 1 C=0,9
0006            IF(100*A+10*B+C.EQ.A**3+B**3+C**3)X=1
0007       1    CONTINUE
0008       2    FORMAT(' A=',I4,'B=',I4,'C=',I4)
0009            CALL CLCPTM(I2)
0010            I3=I2-I1
0011            WRITE(6,5)I1,I3,I2
0012       5    FORMAT(' CMPL'I8/' EXEC'I8/' TCTL'I8)
0013            STOP
0014            END
```

```
73/218          TRANSFER ADDR.          HICCRE          ESD

                028488                  028767          CSE

                                                        CSE

LOADER HIGHEST SEVERITY WAS  0  --  EXECUTION

CMPL   19584
EXEC    2304
TOTL   21888
  STOP      0
/&
```

```
FORTRAN IV G LEVEL 44PS V 1.0          GMAIN44          DATE = 73218   TIME

              INTEGER*2 A,B,C
              CALL CLCPTM(I1)
              DO 1 A=1,9
              T003=A*100
              T004=A**3
              DO 1 B=0,9
              T001=T003+B*10
              T002=T004+B**3
              DO 1 C=0,9
              IF(T001+C.EQ.T002+C**3)X=1
        1     CONTINUE
        2     FORMAT(' A=',I4,'B=',I4,'C=',I4)
              CALL CLCPTM(I2)
              I3=I2-I1
              WRITE(6,5)I1,I3,I2
        5     FORMAT(' CMPL'I8/' EXEC'I8/' TOTL'I8)
              STOP
              END
```

```
73/218          TRANSFER ADDR.          HICORE          ESD TY

                0284B8                  0287FF          CSECT
                                                        CSECT

LOADER HIGHEST SEVERITY WAS  0  --  EXECUTION

CMPL   19968
EXEC    1920
TOTL   21888
  STOP       0
/&
```

EXAMPLE 8

INPUT PROGRAM

```
      REAL A(40,40),C(40)
      READ(5,1)A,C
1     FORMAT(40I2)
      DO 2 I=1,40
      DO 2 J=1,40
      IF(A(I,J).EQ.C(I))A(I,J)=C(I)+A(I,I)
2     CONTINUE
      WRITE(6,1)A,C
      STOP
```

FLOW ANALYSIS

```
BLOCK      SUCCESSORS
       1          2,
       2          3,
       3          4,   5,
       4          5,
       5          2,   3,   6,
       6 IS AN EXIT NODE


THE UNIQUE CLOSED PATHS OF LENGTH LESS THAN OR EQUAL  TO L ARE
 FOR L=  1
   NONE
 FOR L=  2
        3-  5-
 FOR L=  3
        2-  3-  5-
        2-  3-  4-  5-
        3-  4-  5-
 FOR L=  4
   NONE
 FOR L=  5
   NONE
 FOR L=  6
   NONE


  THE OPTIMIZABLE STRONGLY-CONNECTED REGIONS ARE
        3-  4-  5-
        2-  3-  4-  5-


THE IMMEDIATE PREDECESSORS OF THE ABOVE REGIONS ARE
  2
  1
```

CUTPUT PROGRAM

```
      REAL A(40,4C),C(40)
C BLCCK  1
      READ(5,1)A,C
 1    FCRMAT(40I2)
C BLCCK  2
      DO2I=1,40
      T003=C(I)
C BLCCK  3
      DC2J=1,40
C BLCCK  4
      IF(A(I,J).EQ.T003)A(I,J)=A(I,I)+T003
C BLCCK  5
    2 CCNTINUE
C BLCCK  6
      WRITE(6,1)A,C
      STCP
```

THE IMPRCVEMENTS(?) MADE IN THE PROGRAM ARE
   1 COMMON SUB-EXPRESSIONS ELIMINATED
   4 INSTRUCTIONS MOVED

EXAMPLE 9

INPUT PROGRAM

```
      N=100
      DO 1 I=2,N
      IF(A(I+5).LT.9)GO TO 1
      IF(A(K+N).LT.I)GO TO 1
      DO 2 J=1,N
      A(J)=B(N,L,R)-7
      A(J+1)=B(N,I,R)-7
      IF(A(K+N).GT.J)GO TO 3
2     S=(B(N,L,R)-7)**3
      IS=7
1     CONTINUE
3     I=1
4     J=1
5     A(I,J)=(I**2+J**2)**1.5+(L+I)/(J**2+I**2)**1.5
      J=J+1
      IF(J.LE.5000)GO TO 5
      I=I+1
      IF(I.LE.5000)GO TO 4
      STOP
```

FLOW ANALYSIS

```
BLOCK     SUCCESSORS
     1         2,
     2         3,   4,
     3        10,
     4         5,   6,
     5        10,
     6         7,   8,
     7        11,
     8         6,   9,
     9        10,
    10         2,  11,
    11        12,
    12        13,
    13        14,  15,
    14        13,
    15        16,  17,
    16        12,
    17 IS AN EXIT NODE


THE UNIQUE CLOSED PATHS OF LENGTH LESS THAN OR EQUAL  TO L ARE
 FOR L=  1
   NONE
 FOR L=  2
      6-  8-
     13- 14-
 FOR L=  3
      2-  3- 10-
 FOR L=  4
      2-  4-  5- 10-
     12- 13- 15- 16-
 FOR L=  5
   NONE
 FOR L=  6
      2-  4-  6-  8-  9- 10-
     12- 13- 14- 15- 16-
 FOR L=  7
      2-  3-  4-  5- 10-
 FOR L=  8
```

```
        2-  4-  5-  6-  8-  9- 10-
FOR L=  9
        2-  3-  4-  6-  8-  9- 10-
FOR L= 10
        2-  3-  4-  5-  6-  8-  9- 10-
FOR L= 11
  NONE
FOR L= 12
  NONE
FOR L= 13
  NONE
FOR L= 14
  NONE
FOR L= 15
  NONE
FOR L= 16
  NONE
FOR L= 17
  NONE


 THE OPTIMIZABLE STRONGLY-CONNECTED REGIONS ARE
      13- 14-
      12- 13- 14- 15- 16-
       2-  3-  4-  5-  6-  8-  9- 10-


THE IMMEDIATE PREDECESSORS OF THE ABOVE REGIONS ARE
 12
 11
  1
```

OUTPUT PROGRAM

```
C BLOCK  1
      N=1CC
      T008=K+N
      TOC9=B(N,L,R)-7
      T007=T009**3
C BLOCK  2
      DO1I=2,N
C BLOCK  3
      IF(A(I+5).LT.9)GOTO1
C BLOCK  4
C BLOCK  5
      IF(A(TC08).LT.I)GOTO1
C BLOCK  6
      DO2J=1,N
      A(J)=T009
      A(J+1)=B(N,I,R)-7
C BLOCK  7
      IF(A(T008).GT.J)GOTO3
C BLOCK  8
    2 S=T007
C BLOCK  9
      IS=7
C BLOCK 10
    1 CONTINUE
C BLOCK 11
    3 I=1
C BLOCK 12
    4 J=1
      TOC2=I**2
      TOC3=I+L
C BLOCK 13
    5 TOC1=(T002+J**2)**1.5
      A(I,J)=T001+TCC3/T001
      J=J+1
C BLOCK 14
      IF(J.LE.50CC)GOTO5
C BLOCK 15
      I=I+1
C BLOCK 16
      IF(I.LE.50C0)GOTO4
C BLOCK 17
      STCP
```

```
THE IMPROVEMENTS(?) MADE IN THE PROGRAM ARE
   7  COMMON SUB-EXPRESSIONS ELIMINATED
  19  INSTRUCTIONS MOVED
 STCP       0
/&
```

5.2 Conclusions

In this thesis a variety of techniques that can be used in the compiling process in order to produce better quality code have been described. All these methods are usually incorporated into an optimizing compiler. They cannot always be used profitably outside of the compiler. One reason is that all these techniques described previously (even the machine dependent ones) are in some way or another related to each other. For example the effectiveness of transformations such as redundant subexpression elimination depends very much on the register assignment algorithm used, since the elimination of computations tends to separate the definition points for a quantity from the points at which it is used. Therefore, if the register assignment algorithm is not carefully designed, the contents of the registers may have to be stored before their last use and fetched again later. But (with certain machine architectures) the cost of storing and loading the eliminated quantity may be greater than the saving obtained from not recomputing it. Consequently, the output from the developed experimental optimizer should be compiled by a compiler with a fairly sophisticated register assignment algorithm. But, to the best of my knowledge, compilers of this level, perform the optimizations we do by themselves. However, in locations where good optimizing compilers are not available, this program could generally save a reasonable amount of time even if it is used with a low level FORTRAN compiler.

It should be evident from the above remarks that a compiler that aims at producing quality object code must pay careful attention to the design of its phases, taking into account the dependencies between these phases, as well as the peculiarities of the object machine. There is obviously a considerable benefit in terms of time and space to be gained by the implementation of the optimizing transformations discussed in this thesis. But, it is also quite clear, that the application of these transformations on a program do not always produce a completely optimum program. What is needed, therefore, is the development of new methods performing more extensive optimization. An equally efficient and much more realistic alternative solution is to feed some information about program execution back to the user, so that he may do his own optimization in the few places where it really matters. Such optimization may involve redesign of the user's algorithm. This information may be given in terms of a run time histogram depicting the frequency or the cost of each statement executed. A program developed for this purpose is given at the end of this thesis with an example of a program as run through it.

```
 0001               INTEGER*2 INPUT(72),DO(10),IDO/0/,STNUM(5)/5*' '/,COUNT/0/,
                   1STMAP(400),IDNUM/0/,SC(7,2)/'E','E','I','I','C','C','R','X','Q',
                   2'M','N','M','P','L'/,BLANK/' '/,APT/''''/,CM/','/,ZERO/'0'/,LPR/'('/,RPR/'
                   6'/,RPR/')'/,ISN/'='/,DE/'D'/,LA/'L'/,NINE/'9'/,GI/'G'/
 0002               INTEGER*2 IPASS/0/,FMT(6)/'F','O','R','M','A','T'/,OU/'U'/,TAF/'T'
                   1/,GT(9)/'G','O','T','O','9','9','9','9','9'/,BI/'B'/
 0003               LOGICAL*1 FLAG/.FALSE./,MN/.FALSE./,BL/.FALSE./
 0004         200   FORMAT(6X,'INTEGER*2 CONT(400)          ,COUNT,STMAP(400),STAR/''*''/
                   1'13X/6X'REALSWTCH'57X/6X'COMMON/A1/CONT,COUNT,STMAP'40X)
 CC05         2     READ(5,1,END=80)INPUT
 0006         1     FORMAT(72A1)
 0007               IF(INPUT(1).NE.SC(5,1))GO TO 61
 CC08               WRITE(6,60)INPUT
 0009               GO TO 2
 0010         61    IF(INPUT(6).EQ.BLANK)IDNUM=IDNUM+1
 CC11               WRITE(6,60)INPUT,IDNUM
 0012         6C    FORMAT(' ',72A1,2X,I5)
 0013               IF(INPUT(6).EQ.BLANK)GO TO 2C1
 C014               GO TO 16
 0015         2C1   DO 210 I=1,6
 0016               IF(INPUT(I+6).NE.FMT(I))GO TO 211
 0017         210   CONTINUE
 C018               GO TO 16
 0019         211   DO 62 I=1,5
 0020               IF(INPUT(I).EQ.BLANK)GO TO 62
 CC21               FLAG=.TRUE.
 0022               STNUM(I)=INPUT(I)
 0023               INPUT(I)=BLANK
 0024         62    CONTINUE
 CC25               IF(.NOT.FLAG)GO TO 67
 0026               FLAG=.FALSE.
 0027               COUNT=COUNT+1
 0028               WRITE(3,64)STNUM,COUNT,COUNT
 0029         64    FORMAT(5A1,1X,'CONT(',I4,')=CONT(',I4,')+1'43X)
 0030               IF(IDO.NE.0)GO TO 205
 0031               DO 206 K=1,5
 0032         2C6   STNUM(K)=BLANK
 0033         205   N=0
 0034               DO 65 I=1,5
```

```
0035            IF(STNUM(I).EQ.BLANK)GO TO 65
0036            N=N*10+(STNUM(I)-ZERO)/256
CC37            STNUM(I)=BLANK
0038       65   CONTINUE
0039            IF(DO(IDO).NE.N)GO TO 67
0040            STMAP(ICNUM)=COUNT
0041            N1=N+40CCO
0042            WRITE(3,66)N1,(INPUT(I),I=7,72)
0043       66   FORMAT(I5,1X,66A1)
CC44            IDO=IDO-1
CC45            GO TO 40
0046       67   STMAP(IDNUM)=COUNT
CC47            IB=7
0048            DO 68 I=7,72
0049            IF(FLAG)GO TO 69
0050            IF(INPUT(I).EQ.BLANK)GO TO 68
CC51       69   IF(INPUT(I).EQ.APT)FLAG=.NOT.FLAG
0052            IF(I.EQ.IB)GO TO 70
0053            INPUT(IB)=INPUT(I)
CC54            INPUT(I)=BLANK
0055       7C   IB=IB+1
0056       68   CONTINUE
CC57            FLAG=.FALSE.
0058            I=6
0059            ISW=0
0060        3   I=I+1
          C*****FINISHED?
0061            IF(I.EQ.73)GO TO 15
          C*****IF THE TEST BELLOW IS POSITIVE WE FAILLED TO RECOGNIZE A ZERC LEVEL
          C*****EQUAL SIGN OR COMMA.
0062            IF(INPUT(I).EQ.BLANK)GC TC 15
          C*****HAVE WE A LEFT PARENTHESIS?
0063            IF(INPUT(I).NE.LPR)GO TO 4
          C*****LET'S TRY TO GET CUT OF IT.
0064            IS=31
0065            GC TC 5
          C*****HAVE WE AN APOSTROPHE?
0066        4   IF(INPUT(I).NE.APT)GC TC 10
0067            IS=-30
```

```
    5   I=I+1
        DO 6 I=I,72
        IF(INPUT(I).NE.APT)GO TO 7
        IS=-IS
        GO TO 9
    7   IF(IS.LT.0)GO TO 6
        IF(INPUT(I).NE.LPR)GO TO 8
        IS=IS+1
        GO TC 6
    8   IF(INPUT(I).NE.RPR)GO TO 6
        IS=IS-1
    9   IF(IS.EQ.30)GO TO 3
    6   CONTINUE
   10   IF(ISW.EQ.1)GO TO 11
C*****WE ARE OUT.IS IT AN EQUAL SIGN?
        IF(INPUT(I).NE.ISN)GO TO 3
        ISW=1
        GO TO 3
C*****IS IT A COMMA?
   11   IF(INPUT(I).NE.CM)GO TO 3
        N=0
        DO 72 J=9,13
        IF(INPUT(J).GT.0)GO TO 73
        IF(INPUT(J).LT.ZERO)GO TO 73
   72   N=N*10+(INPUT(J)-ZERO)/256
        J=14
   73   N1=N+40000
        IF(IPASS.EQ.1)GO TO 203
        WRITE(3,200)
        IPASS=1
        IF(MN)GO TO 132
        WRITE(3,131)
  131   FORMAT('        DO 99905 I=1,400'50X/'99905 CONT(I)=0'57X)
  132   COUNT=COUNT+1
        WRITE(3,64)STNUM,COUNT,COUNT
  203   WRITE(3,74)(INPUT(K),K=1,8),N1,(INPUT(L),L=J,72)
   74   FORMAT(8A1,I5,70A1)
        IF(IDO.EQ.0)GO TO 22
        IF(DO(IDO).EQ.N)GO TO 40
```

```
 22   IDO=IDO+1
      DO(IDO)=N
 40   COUNT=COUNT+1
      WRITE(3,64)STNUM,COUNT,COUNT
      GO TO 2
 15   IF(ISW.EQ.0)GO TO 19
      IF(IPASS.EQ.1)GO TO 16
      WRITE(3,200)
      IF(MN)GO TO 138
      WRITE(3,131)
138   IPASS=1
      COUNT=COUNT+1
      WRITE(3,64)STNUM,COUNT,COUNT
      STMAP(IDNUM)=COUNT
 16   WRITE(3, 1)INPUT
      GO TO 2
 19   IF(IPASS.NE.0)GO TO 24
C*****SPECIFICATION TEST.
      IF(INPUT(7).NE.BI)GO TO 135
      BL=.TRUE.
      GO TO 16
135   IF(INPUT(7).EQ.DE)GO TO 101
      IF(INPUT(7).NE.LA)GO TO 100
101   STMAP(IDNUM)=0
      GO TO 16
100   IF(INPUT(8).EQ.OU)GO TO 101
110   J=2
      DO 20 I=1,7
      IF(I.GT.2*J)J=J+1
      IF(INPUT(7).NE.SC(I,1))GO TO 20
      IF(INPUT(6+J).EQ.SC(I,2))GO TO 101
 20   CONTINUE
      IPASS=1
      WRITE(3,200)
      IF(MN)GO TO 133
      WRITE(3,131)
133   COUNT=COUNT+1
      WRITE(3,64)STNUM,COUNT,COUNT
      STMAP(IDNUM)=COUNT
```

```
0143         24   I=7
0144              IF(INPUT(I).NE.SC(3,1))GO TO 104
0145              DO 26 I=I,72
0146              IF(INPUT(I).NE.RPR)GO TO 26
0147              K=INPUT(I+1)
0148              IF(K.LT.ZERO)GO TO 27
0149              IF(K.LE.NINE)GO TO 16
0150         26   CONTINUE
0151         27   I=I+1
0152              IF(INPUT(I+2).NE.TAF)GO TO 104
0153         105  WRITE(3, 1)INPUT
0154              GO TO 40
0155         104  IF(INPUT(I+1).NE.TAF)GO TO 107
0156              I=I-1
0157              L=63-I
0158              WRITE(3,60)(INPUT(K),K=1,I),GT,(BLANK,K=1,L)
0159              IF(I.EQ.6)GO TO 2
0160              GO TO 40
0161         107  IF(INPUT(I+1).NE.SC(4,2))GO TO 16
0162              IPASS=0
0163              IF(.NOT.BL)GO TO 136
0164              BL=.FALSE.
0165              GO TO 16
0166         136  IF(MN)GO TO 101
0167              MN=.TRUE.
0168              STMAP(IDNUM)=0
0169              WRITE(3,86)
0170         86   FORMAT('99999 READ(4,99904)COUNT,IDNUM,STMAP'36X/'99904 FORMAT(I4,
                 1I4/20(I4))'46X)
0171              WRITE(3,137)
0172         137  FORMAT(6X'WRITE(6,99907)'52X/'99907 FORMAT(''1ST.NUM.'','' FRQCY'')'
                 1)'39X)
0173              WRITE(3,108)
0174         108  FORMAT('      MAX=0'61X/6X'DO 99900 I=1,COUNT'48X/6X'IF(CONT(I).GT.
                 1.MAX)MAX=CONT(I)'37X/'99900 CONTINUE'58X/6X'SWTCH=100./MAX'52X/6X'DO
                 2DO 99901 I=1,IDNUM'48X/6X'IF(STMAP(I).NE.0)GO TO 99903'38X/6X'L=0'63X/6X'W
                 363X/'99906 WRITE(6,99902)I,L'49X/6X'GO TO 99901'55X/'99903 L=CONT(STMAP
                 3STMAP(I))'50X/6X'IF(L.EQ.0)GO TO 99906'45X
                 4         /6X'L1=L*SWTCH'56X/6X'WRITE(6,99902)I,L,(STAR,K=1,L1)'35X/'99
```

```
              5/'99901 CONTINUE'58X/'99902 FORMAT(1X,I5,2X,I5,2X,100A1)'38X/6X'STOP'62X)
              60P'62X)
 0175           GO TO 16
 0176        80 WRITE(4,85)COUNT,IDNUM,STMAP
 0177        85 FORMAT(I4,I4/20(I4))
 0178           STOP
 0179           END
```

```
  TOTAL MEMORY REQUIREMENTS C01724 BYTES

COMPILER HIGHEST SEVERITY CODE WAS  0
// EXEC CLOADER                                                  11.49.19
```

```
73/247              TRANSFER ADDR.         HICORE            ESD TYPE    LABEL       LOADED      REL-FACTOR

                    00C4B8                 00DBDB            CSECT       MAIN44&     00C4B8      00C4B8
                                                             * ENTRY     MAIN44      00C4B8

LOADER HIGHEST SEVERITY WAS  0  --  EXECUTION                                        LOAD TIME   0 MIN   1 SE

      IMPLICIT INTEGER(A-Z)                                                  1
      DIMENSION A1(8),B1(8),M(8),N(8),BOARD(12,12),U(8),V(8),ARR(8)          2
C
C    PROGRAM TO SIMULATE A KNIGHTS TOUR OF A CHESSBOARD
C    BOARD IS A 12 BY12 ARRAY,WITH THE CHESSBOARD AS AN 8 BY 8ARRAY OF ZE
C    WITHIN THIS.ONCE A SQUARE HAS BEEN VISITED IT IS GIVEN A NON-ZERO VA
C    IN ARRAY.BORDER SQUARES(OFFTHE BOARD) ARE SET ORIGINALLY TO MINUS ON
C    JUMPS TO THESE ARE FORBIDDEN
C    SQUARE VISITED IS GIVEN A NUMBER WHICH SIGNIFIES ON WHICH MOVE IT WA
C    VISITED,G THUS HAS A MAXIMUM VALUE OF 64 IF ALL SQUARES ARE VISITED
C    WHEN NO PERMISSABLE MOVES PROGRAM ENDS
C
      G=1                                                                    3
C
C    READ IN STARTING VALUE I,J
C
      READ(5,10)I,J,BOARD                                                    4
   10 FORMAT(2I3/(12I3))                                                     5
   15 BOARD(I,J)=G                                                           6
      CALL POSPOS(M,N,I,J)                                                   7
      CALL PERPOS(M,N,A1,B1,BOARD)                                           8
      IF(A1(1).EQ.0) GOTO 20                                                 9
      CALL NOALT(U,V,A1,B1,ARR,BOARD)                                       10
      CALL CHOOSE(A1,B1,ARR,I,J)                                            11
      G=G+1                                                                 12
C
C    CYCLE REPEATS WITH NEW VALUES OF I,J
C
      GOTO 15                                                               13
   20 WRITE(6,25)BOARD                                                      14
   25 FORMAT(12X,12I7//)                                                    15
      STOP                                                                  16
      END                                                                   17
      SUBROUTINE CHOOSE(A1,B1,ARR,I,J)                                      18
      IMPLICIT INTEGER(A-Z)                                                 19
      DIMENSIONA1(8),B1(8),ARR(8),MRAND(1)                                  20
```

```
C     IN ASCENDING CRDER.
C
        DO 20 S=1,8
        DO 10 T=1,8
        IF(ARR(T).LT.ARR(S)) GCTO 10
C
C     SWOP VALUES IN ARRAY SO THAT SMALLER PRECEDES THE LARGER
C
        C=ARR(T)
        ARR(T)=ARR(S)
        ARR(S)=C
C
C     ALSO SWOP CORRESPONDING VALUES IN A1 ANC B1.
C
        D=A1(T)
        A1(T)=A1(S)
        A1(S)=D
        E=B1(T)
        B1(T)=B1(S)
        B1(S)=E
   10 CONTINUE
   20 CONTINUE
        K=1
C
C     THE FIRST VALUE OF ARR IS NOW THE MINIMUM VALUE.K DENOTES THE MOVE N
C     EVENTUALLY CHOSEN IE MOVE CHOSEN IS A1(K),B1(K).IF MORE THAN ONE EQU
C     MINIMUM VALUE A RANDOM CHOICE IS MADE.
C
        DO 30 P=2,8
        Q=P-1
        IF(ARR(P).GT.ARR(Q)) GCTO 40
        CALL RANDO(4873,397,MRAND,102,7375,1,2)
C
C     BINARY RANDOM CHOICE MADE IE DEPENDING ON WHETHER RANDOM NUMBER IS O
C     ZERO A PARTICULAR MOVE IS CHOSEN.THIS IS REPEATED TILL MINIMA ARE EX
C
        IF(MRAND(1).EQ.0) GOTO 30
        K=P
   30 CONTINUE
   40 I=A1(K)
        J=B1(K)
C
C     THE MOVE IS MADE TO SQUARE I,J
C
        RETURN
        END
        SUBROUTINE RANDO(A,N,MRAND,C,M,Q,K)
```

```
C
C    SUBRCUTINE TC GENERATE A SERIES OF Q RANDOM NUMBERS BETWEEN 0 AND K.
C    THE PERIOD OF THE SERIES SHOULD BE OF THE CRDER OF M.
C     REFERENCE: SEMINUMERICAL ALGORITHMS, BY KNUTH PUB. ADD/WES, PAGE 9
C
      INTEGER A,C,M,N,Q
      DIMENSION MRAND(Q)
      DO 10 I=1,Q
      NN=A*N+C
      N=MOD(NN,M)
C
C    N= RANDOM NUMBER PRODUCED BY KNUTH'S METHOD-TO ENSURE A LARGE
C     PERIOD THIS NUMBER IS FAIRLY LARGE .FOR A SMALLER NUMBER,CONSIDER
C     THE REMAINDER OF N/MAXRAND WHERE MAXRAND IS THE LARGEST VALUE OF
C     RANCOM NUMBER REQUIRED
C
   1C MRAND(I)=MCD(N,K)
      RETURN
      ENC
      SUBRCUTINE NCALT(U,V,A1,B1,ARR,BOARD)
      IMPLICIT INTEGER(A-Z)
      DIMENSION U(8),V(8),A1(8),B1(8),ARR(8),PER(8,8),QER(8,8),A2(8),
     1B2(8),BOARC(12,12)
      DC 20 I=1,8
      F=A1(I)
      G=B1(I)
      IF (F.EQ.0) GCTC 70
      CALL POSPOS(U,V,F,G)
      DO 20 A=1,8
      PER(I,A)=U(A)
   2C QER(I,A)=V(A)
   70 Q=I
      DO 80 J=Q,8
      DC 80 K=1,8
      PER(J,K)=0
   80 QER(J,K)=0
C
C  NCW CALCULATED NC CF POSSIBLE POSITICNS FOR ONE  OF THE PERMISSABLE MO
C   VALUES STORED IN 2-D ARRAY
C
      DC 50 B=1,8
      DO 30 C=1,8
      U(C)=PER(B,C)
   30 V(C)=QER(B,C)
      IF(U(1).EQ.0) GOTO 55
C
C   POSITIONS MOVED BACK INTO 1-D ARRAY
```

```
      CALL PERPOS(U,V,A2,B2,BOARD)
      Z=0
C
C   Z IS A COUNTER.COUNT NO CF PERMISSABLE MCVES FOR EACH ORIGINAL PERMI
C   MOVE.STORE IN ARRAY ARR(8).FOR ORIGINAL MOVES NOT PERMITTED
C   MAKE VALUE CF ARR = 9
C
      DO 40 D=1,8
      IF (A2(D).EQ.0) GOTO 50
   40 Z=Z+1
   50 ARR(B)=Z
      RETURN
   55 DO 60 I=B,8
   60 ARR(I)=9
      RETURN
      END
      SUBROUTINE PERPOS(M,N,A1,B1,BOARD)
C
C   SUBROUTINE TC CALCULATE THE PERMISSIBLE MOVES FOR A KNIGHT KNOWING
C   THE POSSIBLE MOVES.IN THIS PROGRAM IF A SQUARE HAS ALREADY BEEN
C   VISITED IT IS NC LONGER PERMISSABLE.KNIGHT MAY NOT JUMP OFF BOARD.
C   ALL BORDER SQUARES AND VISITED SQUARES ARE NON-ZERO THUS NOT PERMISS
C
      IMPLICIT INTEGER(A-Z)
      DIMENSION M(8),N(8),A1(8),B1(8),BOARD(12,12)
      Y=C
      Z=0
      DC 10 I=1,8
      K=M(I)
      L=N(I)
      IF(BOARD(K,L).NE.0) GOTO 10
      Z=Z+1
      A1(Z)=K
      B1(Z)=L
   10 CONTINUE
      IF(Z.EQ.8) GOTO 30
      G=Z+1
C
C   PERMISSABLE MCVES STORED IN 2 ARRAYS.A1 HOLDS X-COORDS,B1 HOLDS Y-CO
C   IF LESS THAN 8 PERMISSABLE MOVES SURPLUS VALUES IN A1 ANDB1 ARE SET
C
      DO 20 H=G,8
C
C   FOR NON-PERMISSABLE MCVES ARRAY VALUE SET TO ZERO
C
      A1(H)=0
   20 B1(H)=0
```

```
 30 RETURN                                                                106
    END                                                                   107
    SUBROUTINE    POSPOS(M,N,I,J)                                         108
C
C   SUBROUTINE TO CALCULATE THE POSSIBLE MOVES OF A KNIGHT ON A CHESSBOA
C   IT'S POSITION ON THE BOARD IS REPRESENTED BY 2 COORDINATES.
C   I=X-COORD,J=Y-COORD.POSITIONS CALCULATED BY INCREASING/DECREASING
C   I/J BY 2/1
C
    DIMENSION M(8), N(8)                                                  109
C
C   POSSIBLE MOVES STORED IN 2 ARRAYS.M HOLDS X-COORDS,N HOLDS Y-COORDS
C
    M(1)=I+2                                                              110
    N(1)=J+1                                                              111
    M(2)=I+2                                                              112
    N(2)=J-1                                                              113
    M(3)=I-2                                                              114
    N(3)=J+1                                                              115
    M(4)=I-2                                                              116
    N(4)=J-1                                                              117
    M(5)=I+1                                                              118
    N(5)=J+2                                                              119
    M(6)=I+1                                                              120
    N(6)=J-2                                                              121
    M(7)=I-1                                                              122
    N(7)=J+2                                                              123
    M(8)=I-1                                                              124
    N(8)=J-2                                                              125
    RETURN                                                                126
    END                                                                   127
 STOP       0
/&                                                                            11.50.00

      *****************************       COMPUTING LABORATORY, UNIVERSITY OF ST. ANDREWS.       ******************
      * JOB START TIME = 11.49.03 *          44MFT END OF JOB ACCOUNTING INFORMATION            * JOB END TIME = 11
      ************************************************************************************************************
      *          |       |              TIME DISTRIBUTION              |    CARDS    |       |       |      |    MAXIMUM
      *   JOB    | DATE  |_____________________________________________|_____________| LINES | PAGES |  I/O |___________
      *   NAME   |       |   CPU    |   WAIT   | OVERHEAD | ELAPSED    |  IN  |  OUT |       |       |      | USED | AVA
      *-----------------------------------------------------------------------------------------------------------
      * NAP      247/73  00.00.14  00.00.00   00.00.09   CC.CO.58    422     0      436      13     1838   37 K     1
      ************************************************************************************************************
             THIS JOB WAS RUN IN THE BACKGROUND PARTITION AT A COST OF £ 00.36 , APPROXIMATELY.

* THIS LINE PRINTED AT 11.50.55  ;
```

```
* THIS LINE PRINTED AT 11.51.14  ;

                                          NAPNAPNAPNAPNAPNAPNAPNAPNAPNAPNAPNAPNAPNAPNAPNAP
                                          APNAPNAPNAPNAPNAPNAPNAPNAPNAPNAPNAPNAPNAPNAPNAPN
                                          PN  NAPNAPNA  APNAP          APNAP           PNA
                                          NA   PNAPNAP  PNAP            NAPN            AP
                                          AP    APNAPN  NAPN  NAPNAPNA  APNA  APNAPNAP  PN
                                          PN  N  NAPNA  APNA  APNAPNAP  PNAP  PNAPNAPN  NA
                                          NA  AP  PNAP  PNAP  PNAPNAPN  NAPN  NAPNAPNA  AP
                                          AP  PNA  APN  NAPN            APNA            PN
                                          PN  NAPN  NA  APNA            PNAP           PNA
                                          NA  APNAP  P  PNAP  PNAPNAPN  NAPN  NAPNAPNAPNAP
                                          AP  PNAPNA    NAPN  NAPNAPNA  APNA  APNAPNAPNAPN
                                          PN  NAPNAPN   APNA  APNAPNAP  PNAP  PNAPNAPNAPNA
                                          NA  APNAPNAP  PNAP  PNAPNAPN  NAPN  NAPNAPNAPNAP
                                          AP  PNAPNAPNA NAPN  NAPNAPNA  APNA  APNAPNAPNAPN
                                          PNAPNAPNAPNAPNAPNAPNAPNAPNAPNAPNAPNAPNAPNAPNAPNA
                                          NAPNAPNAPNAPNAPNAPNAPNAPNAPNAPNAPNAPNAPNAPNAPNAP

//NAP JOB ,T5 NICHOLAS                                                            11.50.02
IA00I  04/09/73                                                                   11.50.02
//SYSC04 ACCESS MUNI,1316='SA45V1'                                                11.50.02
//SYSIPT ACCESS KAULA                                                             11.50.02
//  EXEC FORTRAN                                                                  11.50.03
```

```
      IMPLICITINTEGER(A-Z)
      DIMENSIONA1(8),B1(8),M(8),N(8),BOARD(12,12),U(8),V(8),ARR(8)
      INTEGER*2 CONT(400)         ,COUNT,STMAP(4C0),STAR/'*'/
      REALSWTCH
      COMMON/A1/CONT,COUNT,STMAP
      DO 99905 I=1,400
99905 CONT(I)=0
      CONT(    1)=CONT(    1)+1
      G=1
      READ(5,10)I,J,BOARD
   10 FORMAT(2I3/(12I3))
   15 CONT(    2)=CONT(    2)+1
      BOARD(I,J)=G
      CALLPOSPOS(M,N,I,J)
      CALLPERPOS(M,N,A1,B1,BOARD)
      IF(A1(1).EQ.0)GOTO20
      CONT(    3)=CONT(    3)+1
      CALLNOALT(U,V,A1,B1,ARR,BOARD)
      CALLCHOOSE(A1,B1,ARR,I,J)
      G=G+1
      GOTO15
   20 CONT(    4)=CONT(    4)+1
      WRITE(6,25)BOARD
   25 FORMAT(12X,12I7//)
       GOTO99999
99999 READ(4,99904)CCUNT,IDNUM,STMAP
99904 FORMAT(I4,I4/20(I4))
      WRITE(6,99907)
99907 FORMAT('1ST.NUM.',' FRQCY')
      MAX=0
      DO 99900 I=1,COUNT
      IF(CONT(I).GT.MAX)MAX=CONT(I)
99900 CONTINUE
      SWTCH=1CC./MAX
      DO 99901 I=1,IDNUM
      IF(STMAP(I).NE.0)GO TO 99903
      L=0
99906 WRITE(6,99902)I,L
      GO TO 99901
```

```
99903 L=CONT(STMAP(I))
      IF(L.EQ.0)GO TO 99906
      L1=L*SWTCH
      WRITE(6,99902)I,L,(STAR,K=1,L1)
99901 CONTINUE
99902 FORMAT(1X,I5,2X,I5,2X,100A1)
      STOP
      END
```

TOTAL MEMORY REQUIREMENTS 0008C8 BYTES

```
            SUBROUTINECHOOSE(A1,B1,ARR,I,J)
            IMPLICITINTEGER(A-Z)
            DIMENSIONA1(8),B1(8),ARR(8),MRAND(1)
            INTEGER*2 CONT(400)       ,COUNT,STMAP(400),STAR/'*'/
            REALSWTCH
            COMMON/A1/CONT,COUNT,STMAP
            CONT(    5)=CONT(    5)+1
            DO40020S=1,8
            CONT(    6)=CONT(    6)+1
            DO40010T=1,8
            CONT(    7)=CONT(    7)+1
            IF(ARR(T).LT.ARR(S))GOTO10
            CONT(    8)=CONT(    8)+1
            C=ARR(T)
            ARR(T)=ARR(S)
            ARR(S)=C
            D=A1(T)
            A1(T)=A1(S)
            A1(S)=D
            E=B1(T)
            B1(T)=B1(S)
            B1(S)=E
         10 CONT(    9)=CONT(    9)+1
      40010 CONTINUE
            CONT(   10)=CONT(   10)+1
         20 CONT(   11)=CONT(   11)+1
      40020 CONTINUE
            CONT(   12)=CONT(   12)+1
            K=1
            DO40030P=2,8
            CONT(   13)=CONT(   13)+1
            Q=P-1
            IF(ARR(P).GT.ARR(Q))GOTO40
            CONT(   14)=CONT(   14)+1
            CALLRANDO(4873,397,MRAND,102,7375,1,2)
            IF(MRAND(1).EQ.0)GOTO30
            CONT(   15)=CONT(   15)+1
            K=P
         30 CONT(   16)=CONT(   16)+1
```

```
40030 CONTINUE
      CONT(   17)=CONT(   17)+1
   40 CONT(   18)=CONT(   18)+1
      I=A1(K)
      J=B1(K)
      RETURN
      END
```

TOTAL MEMORY REQUIREMENTS 000478 BYTES

```
      SUBROUTINERANDO(A,N,MRAND,C,M,Q,K)
      INTEGERA,C,M,N,Q
      DIMENSIONMRAND(Q)
      INTEGER*2 CONT(400)        ,CCUNT,STMAP(400),STAR/'*'/
      REALSWTCH
      COMMON/A1/CONT,COUNT,STMAP
      CONT(  19)=CONT(  19)+1
      DO40010I=1,Q
      CONT(  20)=CONT(  20)+1
      NN=A*N+C
      N=MOD(NN,M)
   10 CONT(  21)=CONT(  21)+1
40010 MRAND(I)=MOD(N,K)
      CONT(  22)=CONT(  22)+1
      RETURN
      END
```

TOTAL MEMORY REQUIREMENTS 0002F4 BYTES

```
0001            SUBROUTINENOALT(U,V,A1,B1,ARR,BOARD)
0002            IMPLICITINTEGER(A-Z)
0003            DIMENSICNU(8),V(8),A1(8),B1(8),ARR(8),PER(8,8),QER(8,8),A2(8),
               1B2(8),BOARD(12,12)
0004            INTEGER*2 CONT(400)          ,COUNT,STMAP(4CC),STAR/'*'/
CC05            REALSWTCH
0006            COMMON/A1/CONT,COUNT,STMAP
0007            CONT(   23)=CONT(   23)+1
CC08            DC40020I=1,8
0009            CONT(   24)=CONT(   24)+1
0010            F=A1(I)
0011            G=B1(I)
CC12            IF(F.EQ.0)GCTC70
0013            CONT(   25)=CONT(   25)+1
0014            CALLPCSPCS(U,V,F,G)
CC15            DO40020A=1,8
0016            CONT(   26)=CONT(   26)+1
0017            PER(I,A)=U(A)
0018         20 CONT(   27)=CONT(   27)+1
CC19      4CC20 QER(I,A)=V(A)
0020            CONT(   28)=CONT(   28)+1
0021         70 CONT(   29)=CONT(   29)+1
CC22            Q=I
0023            DO40080J=Q,8
0024            CONT(   30)=CONT(   30)+1
0025            DC40080K=1,8
C026            CONT(   31)=CONT(   31)+1
0027            PER(J,K)=0
0028         80 CONT(   32)=CONT(   32)+1
CC29      4C080 QER(J,K)=0
0030            CONT(   33)=CONT(   33)+1
0031            DC40050B=1,8
CC32            CONT(   34)=CONT(   34)+1
0033            DO40030C=1,8
0034            CONT(   35)=CONT(   35)+1
CC35            U(C)=PER(B,C)
CC36         30 CONT(   36)=CONT(   36)+1
0037      40030 V(C)=QER(B,C)
0038            CONT(   37)=CONT(   37)+1
```

```
      IF(U(1).EQ.0)GOTO55
      CONT(  38)=CONT(  38)+1
      CALLPERPCS(U,V,A2,B2,BOARD)
      Z=0
      DO40040D=1,8
      CONT(  39)=CONT(  39)+1
      IF(A2(D).EQ.0)GOTO50
      CONT(  40)=CONT(  40)+1
   40 CONT(  41)=CONT(  41)+1
4CC40 Z=Z+1
      CONT(  42)=CONT(  42)+1
   50 CONT(  43)=CONT(  43)+1
4CC50 ARR(B)=Z
      CONT(  44)=CONT(  44)+1
      RETURN
   55 CONT(  45)=CONT(  45)+1
      DO40060I=B,8
      CONT(  46)=CONT(  46)+1
   60 CONT(  47)=CONT(  47)+1
4CC60 ARR(I)=9
      CONT(  48)=CONT(  48)+1
      RETURN
      END
```

TOTAL MEMORY REQUIREMENTS 000898 BYTES

```
      SUBROUTINEPERPCS(M,N,A1,B1,BOARD)
      IMPLICITINTEGER(A-Z)
      DIMENSIONM(8),N(8),A1(8),B1(8),BOARD(12,12)
      INTEGER*2 CONT(400)       ,COUNT,STMAP(400),STAR/'*'/
      REALSWTCH
      COMMON/A1/CONT,COUNT,STMAP
      CONT(  49)=CONT(  49)+1
      Y=0
      Z=0
      DO40010I=1,8
      CONT(  50)=CONT(  50)+1
      K=M(I)
      L=N(I)
      IF(BOARD(K,L).NE.0)GOTO10
      CONT(  51)=CONT(  51)+1
      Z=Z+1
      A1(Z)=K
      B1(Z)=L
   10 CONT(  52)=CONT(  52)+1
40010 CONTINUE
      CONT(  53)=CONT(  53)+1
      IF(Z.EQ.8)GOTO30
      CONT(  54)=CONT(  54)+1
      G=Z+1
      DO40020H=G,8
      CONT(  55)=CONT(  55)+1
      A1(H)=0
   20 CONT(  56)=CONT(  56)+1
40020 B1(H)=0
      CONT(  57)=CONT(  57)+1
   30 CONT(  58)=CONT(  58)+1
      RETURN
      END
```

TOTAL MEMORY REQUIREMENTS 000398 BYTES

```
      SUBROUTINEPCSPCS(M,N,I,J)
      DIMENSIONM(8),N(8)
      INTEGER*2 CONT(400)         ,COUNT,STMAP(400),STAR/'*'/
      REALSWTCH
      COMMON/A1/CONT,COUNT,STMAP
      CONT(   59)=CONT(   59)+1
      M(1)=I+2
      N(1)=J+1
      M(2)=I+2
      N(2)=J-1
      M(3)=I-2
      N(3)=J+1
      M(4)=I-2
      N(4)=J-1
      M(5)=I+1
      N(5)=J+2
      M(6)=I+1
      N(6)=J-2
      M(7)=I-1
      N(7)=J+2
      M(8)=I-1
      N(8)=J-2
      RETURN
      END
```

```
  TOTAL MEMORY REQUIREMENTS 000298 BYTES

COMPILER HIGHEST SEVERITY CODE WAS  0
// RESET SYSIPT                                                      11.50.54
// EXEC CLOADER                                                      11.50.54
```

```
73/247          TRANSFER ADDR.        HICORE          ESD TYPE    LABEL       LOADED    REL-FACTOR

COMMON                                                COMMON      A1          00E3B8    000642

                00C4B8                00E9FF          CSECT       MAIN44&     00C4B8    00C4B8
                                                      * ENTRY     MAIN44      00C4B8

                                                      CSECT       CHOOSE&     00CD80    00CD80
                                                        ENTRY     CHOOSE      00CD80

                                                      CSECT       RANDO&      00D1F8    00D1F8
                                                        ENTRY     RANDO       00D1F8

                                                      CSECT       NOALT&      00D4F0    00D4F0
                                                        ENTRY     NOALT       00D4F0

                                                      CSECT       PERPOS&     00DD88    00DD88
                                                        ENTRY     PERPOS      00DD88

                                                      CSECT       POSPOS&     00E120    00E120
                                                        ENTRY     POSPOS      00E120

LOADER HIGHEST SEVERITY WAS  0  --  EXECUTION                           LOAD TIME    0 MIN   3 SE
```

-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
-1	-1	41	20	39	8	43	18	63	10	-1	-1
-1	-1	38	7	42	19	62	9	30	17	-1	-1
-1	-1	21	40	37	44	31	60	11	64	-1	-1
-1	-1	6	45	22	61	50	55	16	29	-1	-1
-1	-1	23	36	49	56	59	32	51	12	-1	-1

-1	-1	46	5	58	33	54	13	28	15	-1	-1
-1	-1	35	24	3	48	57	26	1	52	-1	-1
-1	-1	4	47	34	25	2	53	14	27	-1	-1
-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1

ST.NUM.	FRQCY
1	0
2	0
3	1
4	1
5	0
6	64
7	64
8	64
9	64
10	63
11	63
12	63
13	63
14	1
15	0
16	1
17	0
18	0
19	0
2C	0
21	1
22	504
23	4032
24	3029
25	3029
26	3029
27	3029
28	3029
29	3029
30	3029
31	3029
32	3029
33	4032
34	504
35	63
36	63
37	82
38	82
39	19
40	19
41	9
42	19
43	63
44	63
45	63

```
46       0
47       0
48       0
49       0
50      63  *
51      19  *
52      19  *
53      19  *
54      19  *
55       0
56       0
57       0
58       0
59      19  *
60     231  *****
61     231  *****
62     231  *****
63     168  ****
64     168  ****
65    1344  ********************************
66    1344  ********************************
67      63  *
68      63  *
69     336  ********
70    2688  ****************************************************************
71    2688  ****************************************************************
72      63  *
73     231  *****
74    1848  ********************************************
75    1848  ********************************************
76     231  *****
77     168  ****
78     168  ****
79     168  ****
80     814  ********************
81     646  ****************
82     168  ****
83       0
84      63  *
85     336  ********
86      63  *
87       0
88       0
89       0
90       0
91     232  *****
92     232  *****
```

```
  95   1856  *********************************************
  96   1856  *********************************************
  97    814  ********************
  98    814  ********************
  99    814  ********************
 100   1856  *********************************************
 101    232  *****
 102    232  *****
 103    232  *****
 104   1042  *************************
 105   1042  *************************
 106    232  *****
 107      0
 108      0
 109      0
 110    232  *****
 111    232  *****
 112    232  *****
 113    232  *****
 114    232  *****
 115    232  *****
 116    232  *****
 117    232  *****
 118    232  *****
 119    232  *****
 120    232  *****
 121    232  *****
 122    232  *****
 123    232  *****
 124    232  *****
 125    232  *****
 126    232  *****
 127      0
 STOP      0
/&                                                   11.51.13
```

```
****************************      COMPUTING LABORATORY, UNIVERSITY OF ST. ANDREWS.        ******************
* JOB START TIME = 11.50.02 *          44MFT END OF JCB ACCCUNTING INFORMATION             * JOB END TIME = 11
**********************************************************************************************************
*          |       |              TIME CISTRIBUTION                  |    CARDS     |         |         |         |     MAXIMUM
*   JOB    | DATE  |_____________________________________________|____________| LINES | PAGES |  I/O    |_____________
*   NAME   |       |    CPU    |    WAIT   | OVERHEAD |  ELAPSED |  IN  |  CUT  |         |         |         | USED | AVA
*---------------------------------------------------------------------------------------------------------
* NAP       247/73  CC.00.10  00.00.00    00.00.13    00.01.12    23     0        444       21       1443   37 K       1
**********************************************************************************************************
        THIS JCB WAS RUN IN THE BACKGROUND PARTITION AT A COST OF £ CC.35 , APPRCXIMATELY.
* THIS LINE PRINTED AT 11.52.07  ;
```

BIBLIOGRAPHY

(1) Aho, A.V. & Ullman, J.D., "Transformations on Straight-Line Programs", Conference Record Second Annual ACM Symposium on Theory of Computing, 1970.

(2) Aho, A.V., Sethi, R. & Ullman, J.D., "A formal Approach to Code Optimization", Sigplan Proceedings, Vol. 5, No. 7, 1970.

(3) Allen, F.E., "Program Optimization", Annual Review in Automatic Programming", Vol. 5, Pergamon, New York, 1969.

(4) Berge, C., "The Theory of Graphs", Methuen & Co., Ltd., London, 1964.

(5) Breuer, M.A., "Generation of Optimal Code for expressions via Factorization", CACM, June, 1969.

(6) Busam, V. A. & England, D. E., "Optimization of Expressions in FORTRAN", CACM, 1969.

(7) Cocke, J., "Global Common Subexpression Elimination", Sigplan Proceedings, Vol. 5, No. 7, 1970.

(8) Cocke, J., & Miller, R., "Some Analysis Techniques for Optimizing Computer Programs", Proceedings of the 2nd Internation Conference of Systems Sciences, Hawii, 1969.

(9) Cocke, J., & Schwartz, J. T., "Programming Languages and their Compilers", Preliminary Notes, Courant Institute of Mathematical Sciences, New York University, N.Y., April, 1970.

(10) Cowan, D. D., & Graham, J. W., "Design Characteristics of the WATFOR Compiler", Sigplan Proceedings, Vol. 5, No. 7, 1970.

(11) Day, A. C., "FORTRAN Techniques", Cambridge, University Press, September, 1971.

(12) Ershov, A. P., "ALPHA--An Automatic Programming System of High Efficiency", JACM, Vol. 13, No. 1., 1966.

(13) Fateman, R. J., "Optimal Code for Serial and Parallel Computation", CACM, Vol. 12, No. 12, December, 1969.

(14) Floyd, R. W., "An Algorithm for coding Efficient Arithmetic Operations", CACM, January, 1961.

(15) Gear, C. W., "High Speed Compilation of Efficient Object Code", CACM, Vol. 8, 1965.

(16) Gries, D., "Compiler Construction for Digital Computers", Chapter 18, John Wiley, New York, 1971.

(17) Hopgood, F.R.A., "Compiling Techniques", Macdonald Computer Monographs 8, 1969.

(18) Huxtable, D. H. R., "On Writing an Optimizing Translator for ALGOL 60", Introduction to System Programming, P. Wegner (ed.), Academic Press, 1964.

(19) Katzan, H. (Jr.), "Advanced Programming", Van Nostrand, January, 1970.

(20) Kennedy, K., "An Algorithm for Common Subexpression Elimination and Code Motion", SETL Newsletter # 28, Courant Institute of Mathematical Sciences, New York, May, 1971.

(21) Kennedy, K., "A Global Flow Analysis Algorithm", International Journal of Computer Mathematics, Vol. III, No. 1, Gordon and Breach.

(22) Kennedy, K., "Algorithm for Live-Dead Analysis including Node-Splitting for Irreducible Program Graphs", SETL Newsletter #38, Courant Institute of Mathematical Sciences, New York, 1971.

(23) Kennedy, K. & Owens, P., "An Algorithm for Use-Definition Chaining", SETL Newsletter #37, Courant Institute of Mathematical Sciences, New York, 1971.

(24) Knuth, D. E., "The Art of Computer Programming", Volume 1. Addison-Wesley, New York, 1968.

(25) Kreitzberg, C. & Shneiderman, B., "The elements of FORTRAN style", Aarcourt B. J. Inc., 1972.

(26) Lowry, E. S. & Medlock, C. W., "Object Code Optimization", CACM, Vol. 12, 1969.

(27) Marimont, R. B., "A new method of Checking the consistency of Precedence matrices", JACM, Vol. 6, 1959.

(28) McKeeman, W. M., "Peephole Optimization", CACM, Vol. 8, 1965.

(29) Nieuergelt, J., "On the Automatic Simplification of Computer Programs", CACM, Vol. 6, June, 1965.

(30) Prosser, R. T., "Applications of Boolean Matrices to the Analysis of Flow Diagrams", Proceedings of the Eastern Joint Computer Conference, Spartan Books, December, 1959.

(31) Warshall, S., "A Theorem on Boolean Matrices", JACM, Vol. 9, No. 1, January, 1962.