73-9133

GESCHKE, Charles Matthew, 1939-
GLOBAL PROGRAM OPTIMIZATIONS. [Pages 135-137, previously copyrighted material not microfilmed at request of author. Available for consultation at the Carnegie-Mellon University Library].

Carnegie-Mellon University, Ph.D., 1973
Computer Science

GLOBAL PROGRAM OPTIMIZATIONS

Charles Matthew Geschke

Department of Computer Science
Carnegie-Mellon University
Pittsburgh, Pennsylvania 15213
October, 1972

Submitted to Carnegie-Mellon University in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

This work was supported by the Advanced Research Projects Agency of the Office of the Secretary of defense (F44620-70-C-0107) and is monitored by the Air Force Office of Scientific Research. This document has been approved for public release and sale; its distribution is unlimited.

Carnegie-Mellon University

CARNEGIE INSTITUTE OF TECHNOLOGY

AND

MELLON INSTITUTE OF SCIENCE

THESIS

SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF Doctor of Philosophy

TITLE Global Program Optimizations

PRESENTED BY Charles M. Geschke

ACCEPTED BY THE DEPARTMENT OF Computer Science

Wm. A. Wulf — MAJOR PROFESSOR — Oct. 2, 1972 — DATE

J F Traub — DEPARTMENT HEAD — Oct 3, 1972 — DATE

APPROVED BY THE COLLEGE COUNCIL

[illegible] — DEAN — Oct 4, 1972 — DATE

PLEASE NOTE:

Some pages may have

indistinct print.

Filmed as received.

University Microfilms, A Xerox Education Company

ABSTRACT

The dissertation investigates the optimization of object code produced by compilers of higher level languages. Its primary goal is the isolation of a set of primitives which lead to a concise description and correspondingly concise implementation of program optimizations. In addition to being powerful enough to provide a concise representation, the primitives are also basic enough to apply to a wide range of languages and optimization techniques.

The concept of similarity functions is introduced. A set of new optimizations described in terms of the similarity notion is proposed. A translator is described which implements code motion, redundant expression elimination, and new similarity-induced optimizations using the primitives developed in the dissertation. Examples are presented demonstrating the effect of these optimizations.

ACKNOWLEDGEMENTS

My sincere thanks to:

Professor William Wulf, my thesis advisor, who provided constant encouragement, initiated many of the thesis ideas, and oversaw their development;

Professors Fuller, Grayson, Habermann, and Parnas, members of my thesis committee, who helped shape the final form of the dissertation;

my fellow graduate students, especially the Bliss group, Jerry Apperson, Rich Johnsson, Chuck Weinstock, and Dave Wile, who formed an initial, critical audience to many of these ideas;

Peter, Kathleen, and John, who often endured an absent father; and

Nancy, my wife, whose patient understanding and unfailing support sustained me throughout my graduate education and especially during these last months while the dissertation was being completed.

TABLE OF CONTENTS

CHAPTER I

INTRODUCTION

Since the advent of the first FORTRAN compilers, the loss in object code efficiency incurred by the use of higher-level languages has concerned both programmers and compiler designers alike. The proponent of a language intended for compilation, even though he may argue that the cost in lost efficiency is far outweighed by the power and elegance of his language, must generally supply a compiler which produces reasonably efficient code in order to attract a community of users. The new breed of "languages for implementation of systems" is measured against this criterion of efficiency in the extreme.

This thesis investigates the area of object code optimization in the presence of control flow. Its major goal is the isolation of a set of primitives which lead to a concise definition and a correspondingly concise implementation of program optimizations. In addition to being powerful enough to provide a concise representation, these primitives must also be basic enough to apply to a wide range of languages and optimization techniques.

The search for a set of primitives to describe a collection of varied optimizations is motivated initially by a desire to achieve a uniform representation of these optimization strategies. A uniform representation, in turn, leads to an implementation which can be easily structured into combinations of the set of primitives. As a result the same clarity and concision which is inherent in the primitives is reflected in the implementation. In order to demonstrate this correlation between the description and implementation of various optimizations, a later chapter will discuss the structure of an actual optimization pass within a real compiler which uses the primitives.

The identification of a collection of primitives produces another benefit. The ability to perform formal manipulations on these primitives aids in exposing new optimization strategies and helps identify the common characteristics of apparently unrelated techniques. This effect is, of course, more difficult to document. It has been our experience that even though the discovery of an optimization strategy may not develop solely from manipulating the primitives, the ability to grasp the essential characteristcs of an optimization is significantly enhanced by the availability of a set of objects which can be used to describe that strategy concisely.

A BRIEF HISTORY OF OPTIMIZATION

Our investigation has evolved through a set of selections among various alternatives and been motivated by several goals, some already described above and others yet to be stated. Any such evolution of ideas builds upon the work of our predecessors who have investigated the problem of object code optimization. We will not attempt to produce a complete catalog but instead will select those efforts which have guided our choices among alternatives either by contrast or in parallel.

In June, 1965 an article by J. Nievergelt[N65] provided a principle for this area of investigation that seems to remain valid today. He states a limiting constraint on the extent of optimization strategies corresponding to our own: a programmer can optimize his program by relying to a great extent on his knowledge of what that program is to do. Indeed his initial encoding of the solution was already a significant optimization of some less well-defined general problem solving technique. The optimizations we consider are restricted to those which depend on the form of the program only. The results of this thesis show that there continues to be a significant gain in object code efficiency resulting from this level of optimization. As the sophistication of programming languages progresses, it becomes the responsibility of the optimizing compiler to remove the burden of the more tedious details of low-level optimizations from the user. Indeed, as the class of operators and the complexity of data structures grow in power and breadth, the programmer becomes further removed from the target machine (as does the language, perhaps). At some point, then, he is no longer capable of dealing with (or better; he should no longer be as concerned with) the complexities of optimizing his constructs.

In August, 1965 C.W. Gear[G65] summarized and collected information on the state-of-the-art of machine independent optimizations and proposed a three pass compilation incorporating those strategies. That collection of optimizations remains the basis for most of today's investigations.

A significant amount of research into the area of optimization has centered around the work of F. Allen[A69,A70], J. Cocke[C70], and J. Schwartz[CS70]. Their influence is very evident in the optimizations of the FORTRAN-H compiler which are described by E. Lowry and C. Medlock[LM69]. The authors 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. These measurements indicate the real effectiveness of the collection of optimizations implemented in FORTRAN-H.

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. We have chosen instead to restrict the control flow semantics to a go-to-less form of control as exhibited in Bliss[B71,WRH71] and concentrate on primitives which relate to the data flow semantics of a program. These data flow primitives concentrate heavily on exposing the issue of re-ordering evaluations in a language independent manner. Since the suggestion to eliminate the goto by Dijkstra[D68], a debate has proceeded on the merits of the proposal[H72,W71,W72]. Our own experience in reading, writing, and compiling go-to-less programs (in the Bliss sense) supports the adoption of this programming style. Moreover, the assumption of this form of control flow has had a significant impact on our investigation of optimization since it enables us to enumerate a small set of control environments and restrict our attention to optimizations related to those control structures. Previous investigations into optimization techniques described in the more general control flow environment, in general, assume that the program can be converted to a representation which is essentially modeled by the control flow semantics of Bliss.

The preliminary notes written by Cocke and Schwartz[CS70] appear to be the single most comprehensive catalog of optimization techniques available. Throughout the thesis we will refer to the collection of optimizations described in that text as the set of "classical" optimization strategies. The text by Cocke and Schwartz provides us with another motivation for proposing a set of primitives. Most of the descriptions of optimization techniques and their implementations are presented in terms of algorithms which often cover several pages and which are closely related to intermediate representations of the program. A major point in introducing our primitives is to demonstrate an alternative method for describing and implementing optimizations which is considerably more concise, understandable, and independent of the intermediate representation.

Finally, anyone investigating the area of optimization must be aware of the interaction of this area with the study of the equivalence of programs and the detection of potential parallelism in a computation. The issue of equivalence of programs arises from recognizing that an optimization strategy is concerned with transforming a program P to a program P' which is input-output equivalent to P. The area of program equivalence is broad in scope but there has been some work done by A. Aho and J. Ullman[ASU70,AU70] from the viewpoint of an application to optimization. In general, however, their work has been restricted to straight-line programs.

Many optimization techniques involve the re-ordering of the evaluation of expressions in a program. Equivalently those expressions, whose order of evaluation can be interchanged, can in fact be executed in parallel with sufficient interlocks. Some very interesting work in representing the inherent parallelism in a program has been done by R. Shapiro and H. Saint[SS69] using Petri Nets. While the Petri Net model provides an elegant framework for their investigations, this thesis proposes primitives which are more easily implementable in the environment of a compiler.

In addition to the influence of the above work, another principle has directed our selection among several areas of program optimization. We intend to investigate only machine independent optimizations. Thus, for example, we will not discuss "peephole" otpimization. Typically optimizations of this class exploit the instruction set of a particular computer by combining a sequence of several operations into a single machine instruction. Also the thesis will not investigate the area of register allocation. Although this area still requires extensive investigation, the time space constraints on a dissertation have led us to concentrate on those machine independent optimizations which most directly evolve into the new optimizations presented later in the thesis.

THESIS OUTLINE

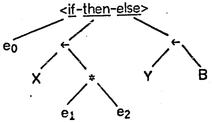
The thesis contains five chapters and two appendices. The remainder of the introduction summarizes our initial assumptions and gives a brief introduction to Bliss. Chapter II introduces the primitives and describes various optimizations techniques in terms of those primitives. Chapter III discusses a concept called similarity which is then used to describe an additional collection of new optimization techniques. Chapter IV presents a set of examples illustrating the various optimization strategies proposed in Chapters II and III. Chapter V contains a summary of our results and suggestions for future research.

INITIAL ASSUMPTIONS

It is inappropriate that a thesis in the area of optimization should tie itself to a single language or single target machine. On the other hand some assumptions are necessary to form the starting point of an investigation. The viewpoint adopted throughout the thesis holds that the optimization algorithms operate on a tree representation of the source program. The syntax analyzer produces a tree in which each control environment and each operator of the source program is represented by a unique node. In the case of an operator its subnodes are its operands whereas in the case of a control environment the subnodes are its subcomponents. For example the program text

<u>if</u> e_0 <u>then</u> $X \leftarrow e_1 * e_2$ <u>else</u> $Y \leftarrow B$

is represented as



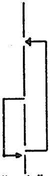
Terminal nodes are always literals or names. The following notational convention is observed for a node, e, such as the <u>if</u>-<u>then</u>-<u>else</u> expression above:

e[operator] = <<u>if</u>-<u>then</u>-<u>else</u>>
e[# of operands] = 3
e[operand_1] = e_0
e[operand_2] = $X \leftarrow e_1 * e_2$
e[operand_3] = $Y \leftarrow B$

The goal of the optimizer is to produce a transformation of this tree which is more optimal in accordance with whatever time/space guidelines the target machine (and perhaps) the user has imposed. The variability in target machines is factored out of the optimizations strategies by: (1) allowing input of the characteristics of the target machine to decision making procedures of the optimizer, and (2) requiring that the optimizer encode sufficient information for the code generators and temporary storage allocators of a particular machine to use in their decisions.

The validity of any reshaping of the program tree is dependent upon the semantics of both the control flow and data flow of the source language. The discussions involving references to control structures are couched in terms of flow diagrams such as the following.

"case/if-then-else" expression



"while-do" expression

While they have their obvious counterparts in the syntax of many languages, all development is independent of a particular syntax.

The major assumption about the control flow semantics, as was stated above, is that the language is go-to-less. The thesis does not consider the problem of detecting programs which fit this model nor the problem of transforming programs into this form. This area has been examined extensively by a number of people. As a result of this go-to-less assumption the tree emitted by the syntax analyzer gives a complete representation of the control flow semantics without further analysis.

In addition to treating the control flow semantics in a general fashion, we wish to factor out of the development the issue of side-effects which result from the semantics of the language's data flow. To this end, a primitive relation, essential predecessor, whose function is to remove the language dependent issue of side-effects, will be introduced. Given a particular language, the semantics of the applications of side-effects within that language define this relation.

The initial assumptions of the thesis are summarized:

(1) algorithms employing the optimization primitives assume a tree representation of the source program as input and produce a similar representation as output;

(2) target machine independence is achieved by parameterizing the optimization algorithms and requiring them to produce information for subsequent machine dependent optimizations in the output representation;

(3) the control flow semantics of the source language are assumed to be go-to-less; and

(4) language dependent data flow semantics are to be isolated by primitive ordering relations so that subsequent development becomes language independent.

A SHORT BLISS PRIMER

Throughout the thesis we will present examples to clarify and motivate concepts as they are introduced. Bliss (and occasionally Algol[AL60]) will be the languages used in these examples. We emphasize that Bliss is introduced for use as a syntactic representation of the control structures and its use does not reduce the language-independence of the optimizations. Bliss is sufficiently Algol-like in many aspects so that a brief introduction to the language should be sufficient for understanding the examples. More detailed information on Bliss is available elsewhere[B71,WRH71].

INTERPRETATION OF NAMES

A Bliss program operates with and on a number of storage "segments". A segment consists of a fixed and finite number of "words". A word may be "named"; the value of a name is called a "pointer" to the word. Identifiers are bound to names by declarations. Thus the value of an instance of an identifier, say x, is <u>not</u> the value of the word named by x, but rather a pointer to x. This interpretation requires a "contents of" operator for which the symbol "." has been chosen.

This context independent interpretation of identifiers as pointers is maintained consistently throughout the language. It is the operators of

Bliss which place an interpretation on the value of an expression. So, for example, the assignment operator "←" interprets its right hand operand as a value which is to be stored in the word pointed to by the value of the left hand operand. As a result the effect of the Algol assignment statement "A:=B+C" is identical to the Bliss assignment "A←.B+.C". This interpretation of names also allows the computation of pointers in Bliss so that the effect of the assignment "(A+3)←.(A+5)" is to store the value of the fifth location past A into the third location past A.

CONTROL STRUCTURES

Bliss is a block-structured, go-to-less, "expression language". That is, every executable construct, including those which manifest control, is an expression and computes a value. Expressions may be concatenated with semicolons to form expression sequences. An expression sequence is evaluated in strictly left-to-right order and its value is that of its last (rightmost) component expression. A pair of symbols, <u>begin</u> and <u>end</u>, or left and right parentheses, may be used to embrace such an expression sequence to form a simple expression. A block is a special case of the construction which contains declarations.

Other than expressions and functions, control mechanisms in Bliss fall into four classes: conditional, selection, looping, and leave. The conditional expression

<u>if</u> e_0 <u>then</u> e_1 <u>else</u> e_2

is defined to have the value e_1 just in the case that e_0 evaluates to <u>true</u> and e_2 otherwise. The abbreviated form "<u>if</u> e_0 <u>then</u> e_1"is considered to be "<u>if</u> e_0 <u>then</u> e_1 <u>else</u> Ø".

The conditional expression provides two-way branching. The <u>case</u> and <u>select</u> expressions provide n-way branching:

<u>case</u> e_0 <u>of</u> <u>set</u> e_1; e_2; ... ; e_n <u>tes</u>

<u>select</u> e_0 <u>of</u> <u>nset</u> e_1: e_2; ... ; e_{2n-1}: e_{2n} <u>tesn</u>

The <u>case</u> expression is executed as follows: (1) the expression e_0 is evaluated, (2) the value of e_0 is used as an index to choose one of the e_j's ($1 \le j \le n$). The value of e_0 is constrained to lie in the range $1 \le e_0 \le n$. The value of the <u>case</u> expression is e_i ($i=e_0$). The <u>select</u> expression is similar to the <u>case</u> expression except that e_0 is used in conjunction with the e_{2j-1}'s to choose among the e_{2j}'s. The execution of the <u>select</u> expression above is described by the following, equivalent Bliss expression.

(T←e_0; V←-1; <u>if</u> e_1 <u>eql</u> .T <u>then</u> V←e_2; ...

<u>if</u> e_{2n-1} <u>eql</u> .T <u>then</u> V←e_{2n}; .V)

Hence the value of the <u>select</u> expression is that of the last e_{2j} to be executed or -1 if none of them is executed.

The loop expressions imply repeated execution (possible zero times) of an expression until a specific condition is satisfied. There are several forms, some of which are:

<u>do</u> e_0 <u>while</u> e_1

<u>incr</u> <id> <u>from</u> e_0 <u>to</u> e_1 <u>by</u> e_2 <u>do</u> e_3

In the first form the expression e_0 is repeated so long as e_1 satisfies the Bliss definition of <u>true</u>. The second form is similar to the "<u>step</u> ... <u>until</u>" construct of Algol, except (1) the control variable, <id>, is local to the <u>incr</u> expression, and e_0, e_1, and e_2 are evaluated only once (before the evaluation of the loop body, e_3). Except for the possibility of a <u>leave</u> expression within e_3 (see below) the value of a loop expression is uniformly taken to be -1.

The control mechanisms described above are either similar to, or slight generalizations of constructs in many other languages. Of themselves they do not remove the inconveniences generated by removing the <u>goto</u>. Another mechanism is desirable -- the <u>leave</u> mechanism. A <u>leave</u> is a highly structured form of forward branch which is constrained to terminate coincidentally with some control environment in which the <u>leave</u> is nested. The general form is:

<u>leave</u> <label> <u>with</u> <expression>

where <label> must be attached to a control environment within which the <u>leave</u> expression is nested. A <u>leave</u> expression causes control to immediately exit from a specified control environment. The <expression> defines the value of the environment.

Finally, functions are defined and called in Bliss in a manner similar to that in Algol, except that there are no specifications and all parameters are implicitly call-by-value. The value of a function is the value of the expression forming its body.

CHAPTER II

OPTIMIZATION PRIMITIVES

This chapter develops a set of primitive relations, functions, and operators to be used in defining a class of feasible object code optimizations. There are several goals that direct this development.

First, the primitives are to form a basis for a set of concise descriptions of various optimizations. The compact notation of the system of primitives provides a basis for succinct descriptions of optimization strategies which in the past have often been described by lengthy algorithms.

Second, the primitives make possible a uniform representation of a large class of optimizations. The pyramid effect resulting from a buildup of primitives defined in terms of combinations of more basic primitives creates this uniformity. In addition this buildup produces a common basis for describing a wide range of optimizations.

Finally, the collection of primitives must allow an implementation of optimizations which is as concise as their descriptions. This final goal directs the selection among a number of different sets of primitives satisfying the preceding two criteria.

PRIMITIVE ORDERING RELATIONS

The problem of object code optimization can be viewed as the search for a transformation T which when applied to a program P produces an program P' that is more efficient. In general the optimization of a program effects a trade-off among a number of measures of program "efficiency". The most important include: size of the object code, execution time, and the amount of storage for data including user requested space and compiler generated temporary storage. The primitives presented in this thesis will concentrate on exposing the set of feasible optimizations in a program. Even though a particular aspect of a program could be optimized (i.e. feasible), it may not be desirable because it only moderately decreases one of the above measures while increasing the cost of another. It should also be pointed out that the notion of efficiency for an algorithm P cannot always be divorced from the data on which P executes. The optimization strategies to be considered and the primitives to be developed are in the class of data independent optimizations that are realizable at compile time. Data sensitive optimizations in general require the collection of run-time statistics which can be used subsequently in re-compilation of the program. As the various optimization strategies are described their effect on the measures listed above will be noted.

We approach the problem of describing feasible optimizations for a program P by considering the ordering relations inherent in a representation of P. There are several: the lexical order of the input text, the precedence-induced order of evaluation, both data-sensitive and data-insensitive order induced by control flow, a leftmost, depth-first tree order, and so forth. Two such orderings are of interest to the development.

The first is the order relation that results from considering a program as a mapping from its set of input variables to its set of output variables. Stated another way, this ordering, called the essential ordering and symbolized by "$\prec$", is the ordering on evaluation of expressions that results from the application of the data flow and control flow semantics of a language L to the set of expressions E in a program P. The optimizations to be considered will regard the essential order in a program as immutable.

The second ordering to be defined allows the selection of subsets of the total set of expressions in a program which at a given point are of interest to an optimization strategy. The following set of examples helps motivate the particular definition given for Bliss.

A representation of a program defines (at least partially) an evaluation order on its set of expressions. For example, the compound expression

begin e_1; e_2; ... ; e_n end

defines an ordering implying that evaluation of e_1 precedes evaluation of

e_2 and so on. However the ordering inherent in this particular representation may or may not correspond to the $\prec$-ordering. The $\prec$-ordering might allow a number of permutations of the components of this compound expression. Consider the expression

$$e_1 + e_2.$$

While the $\prec$-ordering requires that the evaluation of e_1 and e_2 precede the evaluation of the sum, some languages may not define the $\prec$-ordering between the evaluation of the operands e_1 and e_2.

The <u>initial</u> <u>ordering</u> on a program is symbolized by "$\lhd$". Intuitively the relation $e \lhd e'$ expresses the notion that in a straightforward evaluation (i.e. that performed by a classical one-pass, non-optimizing compiler) of this representation of the program the evaluation of e would necessarily have preceded the evaluation of e'. This ordering reflects the precedence relationships of the program as exemplified in the addition expression above. It also reflects the sequential nature of execution as in the case of the compound expression. It does not, on the other hand, necessarily reflect the subnode relationship between nodes. Again, it is to be emphasized that the purpose of this ordering relation is to enable us to select subsets of expressions over which particular optimization strategies will operate.

Definition

The initial ordering on the set of expressions E of a Bliss program is defined as follows:
Let e be a well-formed Bliss expression.

Define $S(e)=\{e' \in E: e' \lhd e$ and e' is a subexpression of $e\} \cup \{e\}$.

One of the following cases applies for e:
(1) e_1 <binop> e_2: $e_1 \lhd e$, $e_2 \lhd e$
(2) <unop> e_1: $e_1 \lhd e$
(3) begin e_1; ... ;e_n end: $e_i \lhd S(e_{i+1})$ $(1 \le i < n)$, $e_n \lhd e$
(4) case e_0 of set e_1; ... ; e_n tes: $e_0 \lhd e$, $e_0 \lhd S(e_i)$ $(1 \le i \le n)$
(5) if e_0 then e_1 else e_2: $e_0 \lhd S(e_1)$, $e_0 \lhd S(e_2)$, $e_0 \lhd e$
(6) select e_0 of nset e_1:e_2; ... ; e_{2n-1}:e_{2n} tesn:
$e_0 \lhd e$, $e_{2i-1} \lhd e$ $(1 \le i \le n)$, $e_0 \lhd S(e_{2i-1})$ $(1 \le i \le n)$,
$e_{2i-1} \lhd S(e_{2n})$ $(1 \le i \le n)$
(7) while e_1 do e_2: $e_1 \lhd S(e_2)$
(8) do e_1 while e_2: $e_1 \lhd S(e_2)$
(9) incr I from e_1 to e_2 by e_3 do e_4:
$e_1 \lhd e_2 \lhd e_3 \lhd e$, $e_1 \lhd S(e_2)$, $e_2 \lhd S(e_3)$, $e_3 \lhd S(e_4)$
(10) $e_0(e_1, \ldots, e_n)$: $e_i \lhd S(e_{i+1})$ $(0 \le i < n)$, $e_n \lhd e$
(11) leave <label> with e_1: $e_1 \lhd e$.

Then e initially precedes e' (notation: $e \lhd e'$) if and only if in the $\lhd$-transitive closure of E there is a subset $\{e_1, \ldots, e_k\}$ such that $e \lhd e_1 \lhd \ldots \lhd e_k \lhd e'$.

Consider the following piece of program text:

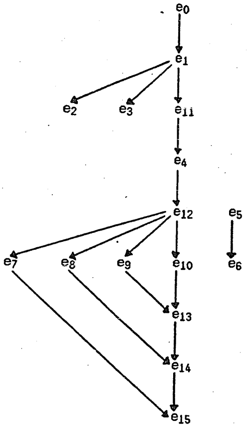
e_0; if e_1 then e_2 else e_3; e_4; do e_5 while e_6; $e_7 \leftarrow e_8 + e_9 * e_{10}$

where $e_0, \ldots, e_{10} \in E$. In addition define:

e_{11}: if e_1 then e_2 else e_3, e_{12}: do e_5 while e_6;

e_{13}: $e_9 * e_{10}$, e_{14}: $e_8 + e_{13}$, e_{15}: $e_7 \leftarrow e_{14}$.

The following lattice illustrates the total set of $\lhd$-relations that hold among the expressions $e_0, \ldots, e_{15}$. ($e_i \lhd e_j$ if there is a path downward from e_i to e_j).



As the set of primitives continues to emerge, we will point out more detailed motivation for some components of the $\lhd$-ordering definition.

DECOMPOSITION OF $\prec$-ORDERING

In the case of simple non-control expressions such as e_{15} the $\lhd$-ordering reflects the precedence-induced ordering of the binary operation. For example the expression e_{13} initially precedes e_{14}. The same relation, i.e. $e_{13} \prec e_{14}$, held in the essential ordering. The differences between the initial and essential orderings must be examined in more detail.

Most languages contain control environments whose components are potentially ⊲-order independent. Consider the following compound expression:

(A ← .B; C ← .D; E ← .F),

where A, ... ,F are distinct variables. Certainly the ⊲-order of execution of these three assignments can be altered. For example

(E ← .F; A ← .B; C ← .D)

produces the same effect. Even within the context of a simple expression such as

.A * .B + .C * .D

the commutivity of the "+" operator is reflected in the fact that the ⊲-order of the two multiplications is not defined. Nevertheless, the ⊲-order still reflects the requirement that both products be evaluated before the addition. Considering a third case,

.A * .B + F()

is an expression where the semantics of Bliss and Algol differ. In the case of Bliss neither the ⊲-order nor the ≺-order of evaluation of the product ".A * .B" and the function call "F()" is well-defined. The usual interpretation of the semantics of Algol, on the other hand, imposes a strict left-to-right evaluation in the presence of the potential side-effects resulting from the call on F. Our observations to this point on the $\lhd$-order and $\prec$-order can be summarized by noting that in general the $\lhd$-order is weaker than the $\prec$-order, i.e. $e \prec e'$ implies that $e \lhd e'$ whereas the converse does not necessarily hold. That is, in some instances, the fact that e has been placed "before" e' (in the $\lhd$ sense) by the programmer is essential and sometimes it is not.

The optimization strategies discussed below will alter the $\lhd$-ordering in a program. Since the validity of such an alteration is constrained by the $\prec$-ordering, a means must be provided for expressing the validity of transformations of a program. Given a pair of expressions e, e' where $e \lhd e'$, two aspects of the essential ordering can be identified that decide the validity of an optimizing transformation re-ordering e and e'. The first of these orderings reflects the relationship between an expression e and those of its subexpressions essential to its evaluation.

<u>Definition</u>

Let $e_1, e_2 \in E$. e_1 is a <u>necessary</u> <u>constituent</u> of e_2 (notation: $e_1 < e_2$) if and only if (iff)
(1) e_1 is a subexpression of e_2, and
(2) evaluation of e_2 requires prior evaluation of e_1.

At first sight conditions (1) and (2) above may appear redundant. Indeed, if the language is Algol, they are redundant. However, in an expression language like Bliss, the following example illustrates their non-redundancy.

<u>Example</u>

Let e_1: $.A*.B$, e_2: $e_1+.C$, e_3: $D \leftarrow e_1$, e_4: $(e_3;\ e_2)$. Then the following relations hold: $e_1 < e_2$, $e_1 < e_3$, $e_2 < e_4$, $e_3 \not< e_4$.†

Notice that the $<$-relation reflects a relationship only between values of expressions. In the example above the existence of e_4 in a program requires that e_3 be executed at some point. However $e_3 \not< e_4$ indicates that the value of e_4 can be computed without prior computation of the value of e_3. The second ordering related to the essential ordering deals with the issue of side effects.

<u>Definition</u>

Let $e_1, e_2 \in E$. The expression e_1 is an <u>essential</u> <u>predecessor</u> of e_2 (notation: $e_1 \ll e_2$) iff

(1) $e_1 \lhd e_2$
(2) the evaluation of the sequences $\{e_1,e_2\}$ and $\{e_2\}$ ($\{e_2,e_1\}$ and $\{e_1\}$) produce distinct values for e_2 (e_1).

<u>Example</u>

Let e_1: $A \leftarrow .A+1$, e_2: $C \leftarrow .B*.A+.D$, e_3: $E \leftarrow .A*(A \leftarrow .A+1)$, e_4: $D \leftarrow .B*.C$, e_5: $.B*.A$. The following relations hold in the context of the compound expression: $(e_3;\ e_2;\ e_4)$. $e_1 \ll e_2$, $e_1 < e_3$ and $e_1 \ll e_3$, $e_1 \not\ll e_4$, $e_5 < e_2$ and $e_5 \not\ll e_2$.

It is important to state precisely the relationship between the orderings $<$ and $\ll$ and the $\prec$-ordering. If these orderings are considered in

----

† Uniformly, a slash through a relational operator denotes the complementary relation.

their standard mathematical representations as subsets of ExE, then their relationship can be stated as: $\{\prec\} \subset \{<\} \cup \{\ll\}$. Hence it follows that if $e < e'$ or $e \ll e'$ then $e \prec e'$; or equivalently if e does not precede e' in the $\prec$-ordering then $e \not< e'$ and $e \not\ll e'$.

This section concludes by defining a relation on ExE which makes some of the subsequent discussions more convenient. Independent expressions are those whose $\prec$-ordering is not determined by the semantics of the language.

Definition

Let $e_1, e_2 \in E$. e_1 is independent of e_2 (notation: $e_1 \diamond e_2$) iff $e_1 \not< e_2$, $e_2 \not< e_1$, $e_1 \not\ll e_2$, $e_2 \not\ll e_1$.

The usefulness of these primitive relations will become apparent during the discussion of the classical optimization strategies involving code motions.

SIMILARITY FUNCTIONS--AN INTRODUCTION

Another primitive notion to be used in defining optimization strategies is a class of real-valued functions defined on the domain ExE. called similarity functions. Chapter III will contain a more detailed discussion.

First, we introduce an equivalence relation called congruence on ExE which is an extension of the equality relation on E. Intuitively, two expressions are congruent if there exists a one-to-one correspondence between them that preserves the tree structure and in which the corresonding nodes are identical operators or terminals. More precisely, the elements of E, considered as nodes in the tree representation, can be decomposed into non-terminal (N) and terminal nodes (T). Moreover T itself can be decomposed into names and literals. Recalling from our description of the tree representation of an expression in Chapter I that a node in E specifies its operator and operands, the notion of congruence is defined as follows.

Definition

Let e, e' $\in$ E. e is congruent to e' (notation: e $\cong$ e') iff either
(1) e, e' $\in$ N,
e[operator] = e'[operator],
e[# of operands] = e'[# of operands]=n, and
e[operand_i] $\cong$ e'[operand_i] ($1 \leq i \leq n$);
(2) e, e' $\in$ T,
e and e' are equal literals or identical names.†

The idea for the similarity function grows out of the recognition that common subexpressions, which among other characteristics are congruent, are a rich source of optimizable expressions in a program. This observation suggests the consideration of those expressions which are "almost" common subexpressions but fail only because they are not quite congruent. As an

----

† The statement that e and e' are identical names is stronger than character string equality. Here we mean that they in fact refer to the unique variables accessible by that identifier within the present environment.

example, let e: .A+.B*.C and e': .A+.B*(.D+.E). When viewed in the form of the tree representations of e and e':

```
e:      +               e':     +
       / \                     / \
     .A   *                  .A   *
         / \                     / \
       .B   .C                 .B   +
                                   / \
                                 .D   .E
```

a strong correspondance is noticeable in the overall super-structure of the expression trees. The intuitive notion, then, of a similarity function is that it is a measure of how "close" two expressions come to being congruent. This intuition leads to the following minimal requirements for a function to be considered a similarity function:

σ is a <u>similarity function</u> only if

(1) σ: $E \times E \to [\emptyset,\infty)$,

(2) $\sigma(e_1,e_2)=\emptyset$ iff $e_1 \cong e_2$, and

(3) $\sigma(e_1,e_2) = \sigma(e_2,e_1)$ for all $e_1,e_2 \in E$. (symmetric)

These three requirements should elicit the intuition that σ satisfies the requirements of a metric on E. That is indeed a reasonable intuition. What is not clear at this point is whether the additional metric requirement of the triangular inequality would add anything to the notion of a similarity function. It is clear, on the other hand, that the above restrictions are not sufficient to provide in themselves a very interesting class of functions. Further discussion of the characteristics of this class of functions is deferred to Chapter III.

Given a particular similarity function, σ, a parameter δ (very σ-dependent) can be selected in terms of which the following relation on ExE can be defined.

Definition

e is strongly similar to e' (notation: $e \simeq e'$) iff $\sigma(e,e') < \delta$. †

The primary reason for interjecting this brief introduction to the similarity function at this point has been to establish the interconnection between the concepts of congruence, strong similarity, and the notion of similarity function. Throughout the remainder of this chapter the similarity function will be used in the very restricted sense of congruence. Chapter III will concentrate on exposing the overall motivation and usefulness of the concept.

This section concludes by introducing one more relation on ExE. Later on in Chapter II there will be a discussion demonstrating how the set of "redundant computations" as defined in Cocke and Schwartz are exposed by our primitives. For the present the notion of common subexpression, which specifies a subset of the collection of all redundant expressions, is defined in terms of the primitives developed so far.

----

† Local context will be sufficient to distinguish the use of "<" as a symbol for "less than" and for "necessary constituent".

Definition

e and e' are common subexpressions (notation: e = e') iff
(1) $e \cong e'$,
(2) $e \lhd e'$ or $e' \lhd e$, and
(3) assuming $e \lhd e'$, $\forall e''$ such that $e \lhd e'' \lhd e'$, $e \not\leftarrow e''$.

The intuition to be conveyed by this definiton of a common subexpression is that if e = e', then (1) the values returned from the evaluation of e and e' are always identical and (2) the control flow of P is such that whenever e' is evaluated then e has been evaluated prior to it (or vice versa). The components of the definition mirror this intuition by saying that (1) e and e' are congruent, (2) the evaluation of e initially precedes e' (or vice versa) by definition of the $\lhd$-ordering, and (3) all the expressions that intervene between e and e' have the property that they do not produce side-effects that affect the value of e (equivalently: e') nor does e produce side effects on them. The latter condition says intuitively that the evaluation of e' is unnecessary since its value is available from the evaluation of e.

CODE MOTION OPTIMIZATIONS

The literature on object code optimization in the presence of control flow identifies a collection of optimization strategies called code motions. This set of optimizations falls into two subcategories: (1) moving evaluations of expressions to less frequently executed points in the program and (2) avoiding unnecessary re-evaluation of expressions whose component values have not changed. The definition of common subexpression in the preceding section is an example of category 2.

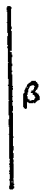
CODE MOTION IN LINEAR BLOCKS

The collection of code motion optimizations about to be described are all predicated on a recursive, inside-out approach for their detection. For example, in detecting code motions relative to an if-then-else control environment, the detection proceeds by first invoking the optimization on the "then" and "else" expressions. The optimization on each of these expressions will (1) detect the feasible optimizations within its own local environment, and (2) return information to be used in detecting optimizations relative to the if-then-else environment. This overall approach requires that a means be provided for stating precisely what information about the sub-components of a control expression is required in order to detect optimizations for the control expression itself. The notion of a <u>linear</u> <u>block</u> is introduced for this purpose. Roughly speaking, β corresponds to those subexpressions of e through which a linear (i.e. $\lhd$-order) flow of control passes.

<u>Definition</u>

Let $e \in E$ and $E' = \{e' \in E$: e' is a subexpression of e$\}$. The <u>linear</u> <u>block</u> β relative to e (notation: $\beta|e$) is the set $\beta|e = \{e' \in E': e' \lhd e\}$.

Since in the context of the use of $\beta|e$ the expression e is quite often obvious, "|e" is simply omitted in most cases. By convention, the linear block relative to e_i will be denoted by β_i. In flow diagrams, linear blocks are depicted as unbroken vertical lines (flow passing from top to bottom):



Example

Consider the expression:
 if e_0
  then (e_1; do e_2 while e_3; e_4)
  else (e_5; if e_6 then e_7 else e_8)
and define:
 e_9: do e_2 while e_3, e_{10}: if e_6 then e_7 else e_8,
 e_{11}: (e_1; e_9; e_4), and e_{12}: (e_5; e_{10}).
Then $\beta_{11} = \{e_1, e_9, e_4\}$ and $\beta_{12} = \{e_5, e_6, e_{10}\}$.

Consider a linear block β that contains the element e: A←.B*.C. We wish to develop a concise description of the potential movability of e backward (to the top) or forward (to the bottom) of the block. It may be feasible to move the evaluation of .B*.C backwards even though the entire expression e cannot be moved. For example:

(F(.A); A←.B*.C...

Assuming F does not produce side effects on B or C, we recognize that the evaluation of .B*.C can be moved backward to the head of the linear block whereas the store into A must follow the parameter evaluation for the call on F. In our terminology, the expression F(.A) is an essential predecessor of e but not of .B*.C. On the other hand the evaluation of .B*.C can never be moved forward to a point after the evaluation of e since .B*.C is a necessary constituent of e.

The following definition defines three sets which make the succeeding definition less cumbersome.

<u>Definition</u>

Let $e \in \beta$, β a linear block.

pro-dominator(β,e) = $\{e' \in \beta\colon e' \lhd e,\ e' \ll e \text{ or } e \ll e'\}$,
epi-dominator(β,e) = $\{e' \in \beta\colon e \lhd e',\ e \ll e' \text{ or } e' \ll e\}$,
post-dominator(β,e) = $\{e' \in \beta\colon e \lhd e',\ e' \not\diamond e\}$.

The pro-dominator set contains those elements of β which initially precede e such that they produce a side effect on e or e produces a side-effect on them. The epi-dominator set differs from the pro-dominator only in that its elements initially follow e. Intuitively the pro-dominator (epi-dominator) contains those elements of β which prevent the movement of e backward (forward) to the head (tail) of β because they produce a side-effect on e or vice versa. The post-dominator set consists of those elements of β which initially follow e and are not independent of e. Hence the post-dominator consists of those elements which prevent the movement of e forward either because of a side-effects relationship or because their evaluation requires the evaluation of e. It follows from the definitions of "<<" and "<>" that: epi-dominator(β,e) $\subseteq$ post-dominator(β,e).

Definition

Let β be a linear block.
prolog(β) = {e $\in$ β: pro-dominator(β,e) = $\emptyset$},
epilog(β) = {e $\in$ β: epi-dominator(β,e) = $\emptyset$},
postlog(β) = {e $\in$ β: post-dominator(β,e) = $\emptyset$}.

Note that it follows immediately that postlog(β) $\subseteq$ epilog(β).

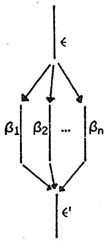
Example

Let e: (A←.B; if .A gtr .B then C←.B*.C; D←.C; B←.X*.Y; X←3). Then β = {.B, A←.B, .A, .B, .A gtr .B, .C, D←.C, .X, .Y, .X*.Y, B←.X*.Y, X←3}. Note that in our discussions of code motion and the related subsets of linear blocks, constants (names and literals) will not be listed since they do not enter into the feasibility of code motions. Now:
prolog(β) = {.B, A←.B, .B, .X, .Y, .X*.Y},
epilog(β) = {.C, D←.C, .Y, X←3}, and
postlog(β) = {D←.C, X←3}.
Observe that the second .B in prolog(β) is the right operand of .A gtr .B and that the .C in epilog(β) is the right side of the store D←.C.

These sets define those expressions that can be moved forward or backward relative to the head or tail of β. At this point the utility of these sets is not yet clear but their usefulness becomes apparent in the context of control environments. In particular the next two sections on optimization strategies for branching and loop control environments stress the expressive power of the primitives for generating concise descriptions of a variety of optimizations.

CODE MOTION IN FORKED CONTROL

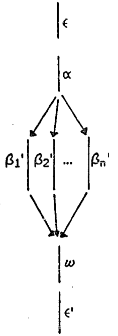
Consider a branching control construct of the form



where ϵ functions as a selector among the n branches. This form of control represents both if-then-else and case types of control environments. The following sections describe several optimization strategies relative to this form of control environment.

ALPHA-OMEGA CODE MOTIONS

The first form of feasible optimization exploits the code motions of the linear blocks β_1, ... ,β_n so that the following flow diagram results:



The linear blocks α and ω contain those expressions factored forward and backward from all of the branches, β_i.

Example

Let β_1: (A←.X*.Y; Y←3) and β_2: (B←.X*.Y; Y←3). Consider the expression: if ϵ then β_1 else β_2. A feasible optimization is to let α: T←.X*.Y, ω: Y←3, β_1': A←.T, β_2': B←.T. This yields the expression: (if (T←.X*.Y; ϵ) then A←.T else B←.T; Y←3).

A primary goal of the development of our optimization primitives is to provide a means of concisely describing the set of feasible members of sets such as α and ω. To that end an operator on the power set of E is introduced.

Definition

Let $E_1, \ldots, E_n$ be subsets of E. The formal intersection of the sets E_i is defined as

$$\wedge E_i = \{e \in E: \forall\ i, 1 \le i \le n, \exists\ e_i \in E_i \text{ such that } e \cong e_i\}.$$

While formal intersection is different from ordinary set intersection the analogy should be obvious: formal intersection differs from set intersection in that the equivalence relation of equality of elements is replaced by that of congruence.

Example

Let β_1 and β_2 be as defined in the preceding example. Then $\beta_1 \wedge \beta_2$ = {.X, .Y, .X*.Y, Y←3}. We reinforce the fact that the "$\wedge$" operator differs from ordinary set intersection by noting that $\beta_1 \cap \beta_2 = \emptyset$.

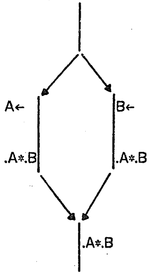
The notion of formal intersection provides us with a powerful tool to concisely define the sets α and ω.

> Given a forked control environment with branches $e_1, \ldots, e_n$, the domain of elements (α) available for pre-evaluation is described by: $\alpha \subseteq \wedge$ prolog(β_i). The domain of elements (ω) available for post-evaluation is described by: $\omega \subseteq \wedge$ postlog(β_i).

The optimizations described by the sets α and ω are examples of optimizations that save space, do not effect time, but may prolong the life-time of temporary storage locations.

POST-MERGE RE-EVALUATIONS

In addition to the goal of providing a collection of primitives that allow a concise definition of a variety of optimizations, these primitives should also be "complete" in the sense that they may be used to describe the class of "redundant" computations in a program. Consider the following example:

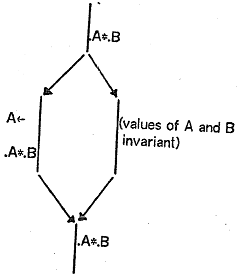


It is apparent that the product that follows the merge point need not be re-evaluated. The set of expressions available for this optimization is described by:

$\wedge$ epilog(β_i).

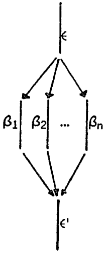
Recall that an element of the epilog set cannot in general have its evaluation moved to the end of the linear block. The value, however, of such expressions is not altered by the expressions that succeed them in the block.

In practice, a more general case can be considered. For example:



once again the evaluation of the product after the merge is not necessary. Since .A*.B does not appear in the right hand branch, it would not be an element of the formal intersect of the epilog of the branches. The extension is straightforward.

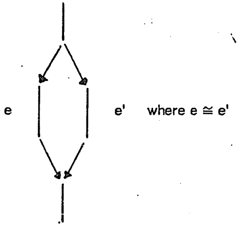
Consider



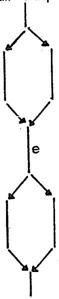
Given a forked control environment with selector expression ϵ and branches $e_1, \ldots, e_n$, define $e_i' = (\epsilon; e_i)$ and $\beta_i' = \beta_i'|e_i'$, $1 \le i \le n$. Then the set of expressions whose evaluation at the merge point would be redundant is the set: $\wedge$ epilog(β_i').

WASP-WAISTING

There is another class of optimizations one might consider in a branching control environment. Consider the example:



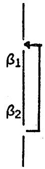
Assume the unspecified portions of the branches are such that the simultaneous motion of e and e' backward as well as forward is impossible. One can consider an optimization which because of the altered appearance of the flow diagram, we call "wasp-waisting".



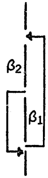
The jump to and return from the common evaluation point can be accomplished by subroutine call and return instructions or by re-testing the branch condition. Wasp-waisting turns out to be a particular case of a more general class of optimizations using the notion of similarity, which will be discussed in Chapter III.

CODE MOTION IN LOOPS

The looping constructs to be considered consist of a body β_1 and a predicate β_2 to be evaluated on each iteration. This section will consider two types. A "do-while" form has its test at the bottom of the loop.



A "while-do" form has its test at the top of the loop.



Other forms of loops such as counting types can be modeled by these forms.

LOOP INVARIANT EXPRESSIONS

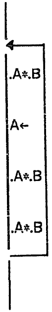
The first optimization strategy considered is the pre-evaluation of the "loop invariant expressions", i.e. those whose values do not change on any iteration of the loop. In terms of the primitives developed, the description of the set of loop invariant expressions is straightforward.

> Given a loop control environment, the set of loop invariant expressions is described by: prolog(β) $\cap$ epilog(β), where β is the linear block relative to the compound expression (β_1; β_2) in the "do-while" case and (β_2; β_1) in the "while-do" case.

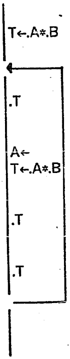
The description has intuitive appeal since it simply states that any expression whose evaluation is not affected by occurring either before or after the loop is not changed by execution of the loop.

CYCLIC RE-EVALUATIONS

The cyclic nature of loop control gives rise to a particular class of "redundant" computations. Consider the following example:



Clearly the expression .A*.B is not invariant throughout the loop. However if the expression .A*.B were pre-evaluated at entry to the loop and stored in a temporary T and if after each recomputation of A or B the expression .A*.B were again evaluated in T, there would be no need to re-evaluate .A*.B at the top of the loop on each iteration. The restructured computation is:



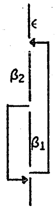
> Given a loop control envrionment where β is the linear block relative to the expression (β_1; β_2) ("do-while") or (β_2; β_1) ("while-do"), the set of expressions whose evaluations at the head of β are redundant to evaluations at the tail of β are described by the set: prolog(β) $\wedge$ epilog(β).

Comparison of this set with the set of loop invariant expressions described above reinforces the distinction between the notions of formal intersection and set intersection. In the case of a loop invariant expression e, the expression e itself appeared in both the prolog and epilog sets whereas an element of the formal intersect is simply an expression which has a formally identical image in both the prolog and epilog sets. Since the first instance of .A*.B can be moved backward, it appears in the prolog but the redefinition of A prevents its appearance in the epilog. However the second instance of .A*.B can be moved forward and so appears in the epilog.

POST LOOP RE-EVALUATIONS

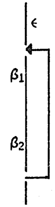
Finally, let us point out how loops particpate in the exposure of the set of redundant expressions to their surrounding environment.

In the case of a "while-do" construct



the set of expressions whose values are available on exit from the loop is the set epilog(ϵ; β_2).

For the case of a "do-while" construct



the set of available expressions on exit is epilog (ϵ; β_1; β_2).

CODE MOTIONS AND LEAVE EXPRESSIONS

To this point our discussion has been limited to go-to-less control structures. In this section we consider the effect of introducing the Bliss "leave" mechanism for exiting control environments. In particular, are the set of primitives powerful enough to describe the code motion optimizations in the presence of leave expressions?

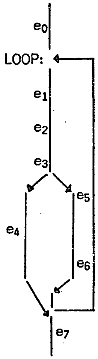
Consider the following example:

e_0;

LOOP: <u>while</u> e_1 <u>do</u>

(e_2; <u>if</u> e_3 <u>then</u> <u>leave</u> LOOP <u>with</u> e_4; e_5; e_6); e_7

The flow diagram for this expression is:



The class of code motion optimizations we have been discussing can be divided into three subclasses: (1) moving an evaluation backward, (2) moving an evaluation forward, or (3) eliminating an evaluation because it is available on all control paths leading to the present evaluation. The linear block relative to the leave expression participates in optimizations of class (1) in a manner analogous to the optimizations proposed for an expression of the form:

<u>if</u> e_3 <u>then</u> e_4 <u>else</u> (e_5; e_6).

As for optimizations of the classes (2) and (3), it is analogous to optimizations for the expression:

<u>if</u> <arbitrary predicate> <u>then</u> e_4

<u>else</u> <u>while</u> e_1 <u>do</u> (e_2; e_3; e_5; e_6).

In effect a leave expression is a forking construct whose optimizations involving backward motion of code behave as though the fork is local to the environment surrounding the leave and whose optimizations involving forward motion of code behave as though the fork terminates at the termination point of the labelled expression. As a particular example, the set of expressions whose evaluation is available on termination of the LOOP expression above is the set: $\text{epilog}(e_0;e_4) \wedge \text{epilog}(e_0;e_1)$.

STRENGTH REDUCTION

A classical optimization in the presence of iterative loop control is "strength reduction". Basically strength reduction exploits the inductive behavior of the control variable in a loop in the attempt to replace relatively expensive operations with less expensive ones by applying recursion relations to express the expensive operation in terms of the less expensive one.

The following example illustrates the technique. Assume a segment of storage named A has been structured so that access to the I-th element of A is defined by the Bliss expression: A+3*.I+5. The loop which follows will zero out every 3*K-th element of A starting at the (3*M+5)-th element and ending at the (3*N+5)-th element of A.

<u>incr</u> I <u>from</u> .M <u>to</u> .N <u>by</u> .K <u>do</u> (A+3*.I+5)←0

Note that on each iteration of the loop the relatively expensive multiplication 3*.I must be re-evaluated in the loop body.

Strength reduction on such a loop transforms the loop expression above to the following:

<u>incr</u> I <u>from</u> (A+3*.M+5) <u>to</u> (A+3*.N+5) <u>by</u> 3*.K <u>do</u> .I←0.

This latter loop has the same semantic effect as the former but now there are no multiplications taking place in the loop body.

Unrolling the first few iterations of the original loop will help motivate the discussion which follows.

```
(0) I←.M
(1) if .I gtr .N then <endloop>;
(2) (A+3*.I+5)←0;
(3) I←.I+.K;
(4) if .I gtr .N then <endloop>;
(5) (A+3*.I+5)←0;
(6) I←.I+.K;
        etc.
```

Notice that the accessing expressions in (2) and (5) are congruent. They are not common subexpressions, however, because of the intervening re-evaluation of I at (3). This unrolled representation of the loop example suggests an investigation into a more general form of the strength reduction notion.

STRENGTH REDUCTION--A GENERALIZATION

Consider the following question: given e, e' $\in$ E and e $\lhd$ e', can one characterize the cases in which there exist an expression Δe and a function F such that e' $\equiv$ F(e,Δe) and the computational cost of F(e,Δe) is less than the computational cost of e' evaluated in the usual manner? We have already seen two cases:

(1) Clearly the example of a strength reduction optimization in the preceding section fits this situation. In general it reduced

the cost of execution.

(2) The second case involves the redundant expression elimination discussed earlier in the chapter. The sequence (X←e; Y←e <op> e') will make use of the fact that it need not recompute the left hand operand of <op>. Such optimizations save both time and space.

Our discussion of strength reduction examines the possible extensions of the notion and the corresponding difficulties in exploiting those extensions. In the process of this development the primitives already developed are used and a few specialized notions are defined.

PRIMITIVES FOR STRENGTH REDUCTION

Returning to the context of the unrolled strength reduction example presented above, the necessity of stating more precisely the interaction between the re-evaluation of I and the corresponding change in the value of A+3*I+5 is evident. We begin by proposing a definition that describes the set of expressions involved in the evaluation of e, e' and all the expressions between them in the $\lhd$-ordering.

<u>Definition</u>

Let $e, e' \in E$, $e \lhd e'$ and $E' = \{e'' \in E: e \unlhd e'' \lhd e'\}$. The <u>interval</u> from e to e' (notation: int(e,e')) is defined as the set

$E' \cup \{e \in E: e'' \in E', e \text{ a subexpression of } e''\}$.

Example

Let $e:(e_1;\ e_2;\ e_3;\ e_4)$ where e_3: do e_5 while e_6. Then $int(e_2,e_4) = \{e_2, e_3, e_5, e_6, e_4\}$. Similarly $int(e_1,e_4) = \{e_1, e_2, e_3, e_4, e_5, e_6\}$. Contrast this with the linear block $\beta|e$. $\beta|e = \{e_1, e_2, e_3, e_4\}$ does not include e_5 and e_6 because of the definition of the $\triangleleft$-ordering.

The notion of linear block is defined relative to a single expression. As a result it is impossible to talk about the linear block relative to an interval. This difficulty is resolved by defining the minimal expression containing an interval, which will be called the cover of the interval. In some cases the cover will itself be an expression in the program. Consider the case, however, where the expressions e and e' appear in a compound expression (as did the two instances of A+3*.I+5 in the unrolled loop example above). For example let $e'':(e_1;\ e;\ e_2;\ e_3;\ e';\ e_4;\ e_5)$ be the minimal expression containing e and e'. The interval from e to e' is the set $\{e, e_2, e_3, e'\}$ and the cover should not contain expressions which will not enter into the consideration of what occurs as execution passes from e to e'. Hence the cover, in this case, will be defined as the compound expression $(e;\ e_2;\ e_3;\ e')$ which does not occur as an expression in the program.

Definition

Let $e, e' \in E$, $e \triangleleft e'$, and let e" be the minimal expression in E which contains the elements of int(e,e') as subexpressions. The cover of int(e,e') is defined as

$$cover(e,e') = \begin{cases} e'', & e'' \text{ not a compound expression} \\ c, & \text{otherwise.} \end{cases}$$

c is defined as follows. Let $e'':(e_1;\ \ldots\ ;\ e_i;\ \ldots\ ;\ e_j;\ \ldots\ ;\ e_n)$. Then c is the compound expression $(e_i;\ \ldots\ ;\ e_j)$ where:

$\forall\, x \in int(e,e')\ \exists\ k,\ i \le k \le j$ such that x is a subexpression of e_k,

and

$\forall\, e_k,\ i \le k \le j,\ \exists\ x \in int(e,e')$ such that x is a subexpression of e_k.

Refering to the preceding example, it follows from the definition of a cover that cover(e_1,e_3) = (e_1; e_2; e_3) and cover(e_5,e_6) = e_3.

The next concept is well understood but is defined for completeness.

<u>Definition</u>

Let $e_0, \dots, e_n \in$ E and let $I_1, \dots, I_n$ be variables. A <u>linear polynomial</u> e in the n variables $I_1, \dots, I_n$ is denoted by $e\langle I_1, \dots, I_n\rangle$ and is an expression of the form

$$e_0 + e_1 {*}.I_1 + \dots + e_n {*}.I_n.$$

STRENGTH REDUCTION WITHOUT LOOPS

Now the conditions that make a strength reduction optimization possible in an environment such as the specific unrolled example above can be described. Let e, e' $\in$ E, e $\lhd$ e', and e $\cong$ e'. Let e (and e') be linear polynomials in n variables: $e\langle I_1, \dots, I_n\rangle$ ($e'\langle I_1, \dots, I_n\rangle$). Let β be the linear block relative to cover(e,e'). Assume that for all k, $1 \le k \le n$, the only redefinitions of I_j (if any) in int(e,e') are of the form: $I_j \leftarrow .I_j + \Delta_j$ where $\Delta_j \in$ prolog(β) $\cap$ epilog(β). Also assume that the coefficient expressions $e_0, \dots, e_n$ are elements of prolog(β) $\cap$ epilog(β). Define: $\Delta e = e\langle I_1+\Delta_1, \dots, I_n+\Delta_n\rangle - e\langle I_1, \dots, I_n\rangle$. The following two observations can be made:

(1) Δe is a polynomial ($\Delta e<\Delta_1, \ldots ,\Delta_n>$) and moreover

if $e=e_0+e_1*I_1+ \ldots +e_n*I_n$ then $\Delta e=e_1*\Delta_1+ \ldots +e_n*\Delta_n$.

(2) value(e')=value(e)+value(Δe).

Example

... X←3*.I+4*.J+5; I←.T+2; J←.J+7; Y←3*.I+4*.J+5; ... Let e be the right side of the assignment to X and e' the right side of the assignment to Y. We see: Δe=(3*(.I+2) + 4*(.J+7) + 5) - (3*.I+4*.J+5) = 34. And so value(Y) = value(X) + 34.

Admittedly, the above example is biased by the fact that both the polynomial coefficients and Δ_I and Δ_J are all constants. If, for example, the re-definition of I were: I←.I+.K, then Δe = 3*.K+28. The product 3*.K is more palatable if we consider a sequence such as:

X_1←e; I←.I+.K; J←.J+7;

X_2←e; I←.I+.K; J←.J+7;

.
.
.

X_m←e; I←.I+.K; J←.J+7; ...

Now the evaluation of Δe occurs only once and the successive stores in the X_i's can be accomplished by the sequence:

Δe←3*.K+28;

X_1←.e;

X_2←.X_1+Δe;

X_3←.X_2+Δe;

.
.
.

X_m←.X_{m-1}+Δe;

Assume that the sequence of names $X_1, X_2, \ldots X_{m-1}, X_m$ is computable in the sense that a function f exists such that for all i, $2 \leq i \leq m$, $X_i = f(X_{i-1})$. Then the sequence of stores above strongly suggests an unrolled loop.

STRENGTH REDUCTION WITH LOOPS

The observations made in the preceding section can be restated within the context of a looping control expression.

> Given a loop of the form: <u>incr</u> I <u>from</u> e_0 <u>to</u> e_1 <u>by</u> e_2 <u>do</u> e_3, let e, a subexpression of e_3, be a polynomial in I for which we wish to perform a strength reduction optimization. Let e<I> = e'*.I+e" and let β be the linear block relative to e_3. Then the following conditions must hold for the strength reduction to be feasible:
> (1) e',e" $\in$ prolog(β) $\cap$ epilog(β), i.e. e',e" are loop invariants,
> (2) the only redefinition of I is the loop increment I←.I+e_2. (The semantics of Bliss require that e_2's value be evaluated prior to loop entry and perserved. Hence e_2 is loop invariant.)

The strength reduction optimization is realized by transforming the original loop to the following:

<u>incr</u> I <u>from</u> (e_0; I'←e<e_0>) <u>to</u> e_1 <u>by</u> (e_2; Δe←e'*e_2)

<u>do</u> (e_3'; I'←.I'+Δe);

where e_3' is obtained from e_3 by replacing e by .I'. If e were the only expression in e_3 that accessed the value of I then a more significant strength reduction of the form:

<u>incr</u> I <u>from</u> e<e_0> <u>to</u> e<e_1> <u>by</u> e'*e_2 <u>do</u> e_3'

can be performed where again e_3' is obtained by replacing e with .I in e_3. This loop has only one induction variable and the "to" test on e_1 has been replaced by $e\langle e_1\rangle$. The following section examines extensions of the strength reduction notion and the corresponding problems.

STRENGTH REDUCTION EXTENDED

In the preceding sections on strength reduction a set of requirements was imposed in order that a specific form of strength reduction would be feasible. Consider what occurs as we begin to relax some of those requirements. First of all, what effect does the removal of the linearity requirement on polynomial have? For example let e: 3*.I*.I - 4*.I + 7. Then Δe = (3*(.I+Δ_I)*(.I+Δ_I) - 4*(.I+Δ_I) + 7) - (3*.I*.I - 4*.I + 7) = 6*.I*Δ_I + 3*Δ_I*Δ_I - 4*Δ_I. The computation of Δe still involves a non-constant multiplicative term: 6*.I*Δ_I. Strength reduction on M=6*.I*Δ_I removes the necessity of performing this evaluation on each iteration of the loop. Then ΔM = 6*(.I+Δ_I)*Δ_I - 6*.I*Δ_I = 6*Δ_I*Δ_I. This allows a transformation of the loop

```
incr I from Ø to .N by 3 do F(3*.I*.I-4*.I+7);
```

to

```
I←7; T←3*.N*.N-4*.N+7; M←Ø;
while .I leq .T do
    begin
        F(.I);
        I←.I+(.M+15);
        M←.M+54;
    end;.
```

In general if the expression e upon which a strength reduction is being performed is an n-th degree polynomial, then n-1 additional variables, like M in the example above, must be introduced in order to maintain the partial accumulations.

Having removed the linearity requirement for polynomials, consider the possiblity of relaxing the polynomial requirement itself. The point of the reduction in strength optimization is to replace an expensive operation with a less expensive one. In the case of multiplication and addition, the feasibilty of such a replacement comes from the inductive relationship between the operands of the successive multiplications and the fact that a product can be accumulated by a sequence of additions. This overall relationship is reflected in the fact that given a n-th degree polynomial $e<I>$ then the polynomial $\Delta e = e<I+\Delta_I> - e<I>$ is always of degree n-1. There are two critical points here:

(1) Δe is a polynomial and so a closed form solution is available to the difference $e<I+\Delta_I> - e<I>$, and

(2) Δe is of degree n-1, which means that successive reductions will eventually reduce all multiplications to additions.

Hence the question remains: are there other strength reductions besides those between "*" and "+"? All the preceding development holds equally well if we replace "*" by exponentiation and "+" by "*". For example the loop:

```
incr I from 1 to .N by 1 do A[.I]←.X <exp> .I;
```

can be replaced by the following expression in which no exponentiation occurs:

```
(J←.X;
 incr I from 1 to .N by 1
    do (A[.I]←.J; J←.J*.J));.
```

The section on strength reduction began by asking the question: given e, e' ∈ E and e ◁ e', can one characterize the case where e' ≡ F(e,Δe) and the cost of computing F is less than the cost of computing e'? The attempt to isolate the essential characteristics of strength reduction with a view to extending the notion initially motivated that question. Subsequent discussion has pointed out two directions for extension: (1) strength reduction in non-looping environments and (2) strength reduction between non-polynomial expressions. The primitives were able to define the feasible strength reductions independent of the presence of the loop control environment. The challenge remains in case (1) to discover an algorithm for directing the search through the set E for pairs (e,e') on which a strength reduction can be performed. Loops have the property that they both localize the search and, in the case of incr loops, immediately identify an induction variable. As for case (2), the challenge is to discover a more general means of constructing closed-form representations of the recursion relation F. Polynomials have the property that the expression Δe is an easily identifiable polynomial itself.

REDUNDANT EXPRESSIONS

Several references have been made in the preceding sections to the concept of redundant expressions in a program. In the present section we demonstrate that our primitives expose the set of redundant expressions in a program consisting of the forked and looping control environments discussed above. The following definition is a direct quote from the text by J. Cocke and J. Schwartz.†

Definition

An operation A*B (i.e. an operation which combines two inputs A and B to give some sort of result, which we write as A*B) is redundant if there exists no track in the program graph, either beginning at the program entry block, or beginning at any assignment of a new value to one of the variables A or B, which reaches the given operation without passing through some preceding calculation of the result A*B.

The definition of common subexpression identified a collection of redundant expressions, i.e. if e = e', then e' is redundant (assuming $e \lhd e'$). The fact that $e \lhd e'$ implies that every control path that leads to an evaluation of e' has previously evaluated e. There is no intervening assigment to the components of e since by part(3) of the definition of common subexpression: $\forall$ e", $e \unlhd e'' \unlhd e'$, $e \not\lhd e''$.

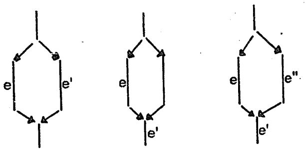
Assume that e' is a redundant expression and that e is the congruent expression that "creates" this redundancy. Furthermore, assume that this

----

† cf. [CS70], pp. 427-428.

redundancy was not exposed by the optimization techniques presented above. Now if $e = e'$, then e' would be redundant. Hence one of the three conditions for a common subexpression must not hold. The first condition, viz. $e \cong e'$, must be satisfied by e and e' since congruence is a property of redundant expressions. If the second condition ($e \lhd e'$) is assumed to hold, then the third condition of the c-s-e definition indicates the existence of an expression e" such that $e \not\lhd e''$. However, the existance of the expression e" again violates the definition of redundancy for e and e'. Thus we have only to consider the cases in which $e \not\lhd e'$.

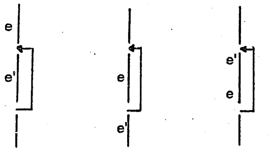
There are three cases to consider for forking control environments:



CASE 1 CASE 2 CASE 3

In case 1; the expression e' is not redundant since no control path leads from the left-hand branch to the right. Notice, however, that optimizations have been proposed above which attempt to combine the two evaluations. The α and ω optimizations expose the feasibility of simultaneously moving the evaluations of e and e' backward or forward. Wasp-waisting is a feasible optimization for those cases where forward or backward motion is impossible. In case 2, e' is not redundant since control potentially passes down the right-hand branch and so does not evaluate e. Finally in case 3, e' is potentially redundant since the expressions e and e" are evaluated on each control path. If no side-effect producing expressions occur between the evaluation points of e and e" and the evaluation point of e', then e' is redundant. This class of redundant expressions, as described in the section on post-merge re-evaluations, is detected by the formal intersection of the epilogs of the branches.

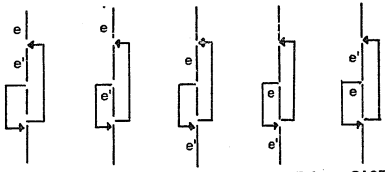
A "do-while" looping environment presents three cases to consider in which e $\not\triangleleft$ e':



CASE 1 CASE 2 CASE 3

For convenience, we define χ to be the set of loop-invariant expressions in a loop. Thus χ = prolog(β) $\cap$ epilog(β) where β is the linear block relative to the body and predicate of the loop. In case 1, the evaluation of e' is redundant only if e' $\in \chi$. The redundancy of e' in case 2 does not require that e $\in \chi$ but simply that e $\in$ epilog(β). Case 3 is exactly the situation discussed in the section on cyclic re-evaluations. In this case e' is redundant if e' $\in$ prolog(β) $\wedge$ epilog(β).

The "while-do" form of loop presents the following cases for consideration:



Again let χ be the set of loop-invariant expressions. Let β_1 and β_2 be the linear blocks relative to the <u>while</u> and <u>do</u> expressions respectively. Let $\beta = \beta|(\beta_1; \beta_2)$. In both cases 1 and 2, the expression e' is redundant only if e' $\in \chi$. In case 3, e' is redundant if e' $\in$ epilog(β_1). e' is not redundant in case 4 since there is no guarantee that the <u>do</u> expression will be executed. Finally, case 5 is again an example of a cyclic re-evaluation and so e' is redundant if e' $\in$ prolog(β) $\wedge$ epilog(β').

SUMMARY

A primary goal of the thesis is to propose a collection of primitives for describing object code optimizations which are powerful enough to provide concise descriptions of optimizations. The set of primitives presented in this chapter was motivated, defined, and used in describing the code motion, redundant expression elimination, and strength reduction optimizations discussed in Cocke and Schwartz. The collection of paragraphs delineated by vertical lines describe these optimizations. Their concision is self-evident.

The primitives also apply to a broad class of optimizations. In particular, it would be inappropriate that disjoint collections of primitives would be used in describing each class of optimizations. An examination of the set of descriptions shows that most of the primitives permeate through all the descriptions. The ordering relations ($\lhd$, $\prec$, $\ll$, $<$) and the subsets defined in terms of them (prolog, epilog, postlog) are used consistently throughout the chapter. As a result, although the optimizations themselves may on the surface appear to be unrelated, the primitives provide a homogeneous description of them. This homogeniety, in turn, leads to a compact, cleanly structured implementation.

Another objective of the thesis is that the primitives be language independent. This objective has been achieved by isolating the language dependent relationships in the "necessary constituent" ($<$) and "essential predecessor" ($\ll$) relations. The ability to isolate these language dependent relationships contributes significantly to the concision of the descriptions.

The primitives have been developed in a representation-independent manner. No inherent characteristics of the primitives are concerned with the data structure of the program's representation. Hence there is no implied implementation strategy underlying the primitives. Again, this contributes to their concision and clarity. This aspect of the primitives allows relative freedom in implementation strategies. In addition it has resulted in a set of primitives that can be manipulated purely on a formal level. Potentially, this can lead to results whose discovery would be hopelessly obscured by any specific representation.

Finally, previous investigations in the area of object code optimizations often describe optimizations in terms of lengthy algorithms which manipulate particular representations. Our primitives have succeeded in partitioning those algorithms into operators, relations, and the characteristic functions of particular sets of expressions. Hence, we are able to describe optimizations in terms of the primitives without regard to the representation of the program or the particular implementation details of the primitives. A good example of the effect of the homogeniety, concision, and representation-independence is the discussion of the completeness of redundant expression elimination in the preceding section.

CHAPTER III

SIMILARITY FUNCTIONS

In Chapter II a collection of primitives was developed to concisely describe previously known optimization techniques. This chapter examines a class of real-valued functions called similarity functions to be used in conjunction with the primitives of Chapter II in describing a set of new optimizations. These optimizations produce dramatic reductions in object code size in certain cases where the classical optimizations presented earlier have little effect. In particular an example presented in Chapter IV shows a 28 percent savings in a 1000-word program resulting from these techniques. This reduction is to be contrasted with the 6 percent savings that results when the same program is optimized using only the classical optimizations.

The presentation of similarity in this chapter is divided into three sections: (1) a discussion of the origins of the similarity concept, (2) the development of a particular similarity function, and (3) an examination of a collection of optimization techniques based on the concept.

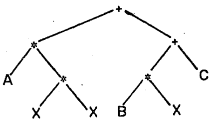
ORIGINS OF THE SIMILARITY CONCEPT

The optimization techniques described in Chapter II fell into two categories: (1) moving the evaluation of expressions either to reduce frequency of execution (e.g. pre-evaluation of loop-invariant expressions) or to eliminate parallel evaluations (e.g. alpha-omega code motions) and (2) avoiding the unnecessary re-evaluation of an expression (e.g. a common subexpression). The initial stimulus for the similarity concept arises from a consideration of the sets of expressions on which optimizations from category (2) operate. If e and e' are a pair of expressions such that the evaluation of e' is made unnecessary by the prior evaluation of e, then e and e' are congruent and can be translated into identical code sequences. The phrase "identical code sequence" is to be interpreted loosely as meaning an identical sequence of machine code operations ignoring the possibility of different temporary accumulators. The key intuition is that although congruent expressions are translated into identical code sequences, the converse does not follow. That is, identical code sequences can be produced for the evaluation (or partial evaluation) of expressions which are not congruent. For example, the code sequence

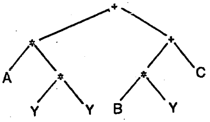
```
LOAD   T2,T1
MULT   T2,T2
MULT   T2,A
MULT   T1,B
ADD    T2,T1
ADD    T2,C
```

can be used to evaluate e: A*X*X+B*X+C or e': A*Y*Y+B*Y+C by loading T_1 with X or Y respectively. In terms of the tree representations of e and e':

e:



e':



the similarity of the code sequences produced for these two trees, and consequently the possiblity of using a single sequence such as that above, arises from the common superstructure of the two trees. The notion of a similarity function is introduced precisely in order to measure the degree of identity of the superstructures of two trees. The similarity notion provides a coherent mechanism for identifying expressions whose evaluations can be merged into identical code sequences.

Additional intuition for the similarity concept is derived from a consideration of the requirements imposed on a pair of expressions e and e' by the definition of common subexpression. There are three: (1) e is congruent to e' ($e \cong e'$); (2) e initially precedes e' ($e \lhd e'$); and (3) none of the expressions intervening between e and e' have a side-effects relationship with e ($\forall$ e'', $e \unlhd e'' \unlhd e'$, $e \not< e''$ and $e'' \not< e$).

The class of optimizations considered in Chapter II uniformly imposed condition (1). That set of optimizations was described in terms of formal intersection or ordinary set intersection. Because congruence is an inherent characteristic of these two operators, those optimization strategies necessarily dealt with sets of expressions that were congruent. The same optimization techniques did, on the other hand, consider cases in which conditions (2) and (3) did not hold. In the collection of code motions related to forking environments, the sets α and ω contained expressions that did not satisfy condition (2) since in the initial ordering of the program those expressions were on parallel branches and hence did not initially precede one another. The optimizations involving strength reduction and cyclic re-evaluations relaxed condition (3) by allowing the existence of intervening side-effects related expressions. Naturally enough, since several optimization strategies involved relaxation of conditions (2) and (3), one is led to consider relaxing condition (1).

As a framework for the ensuing discussion, we will present examples of optimization techniques involving a relaxation of condition (1) which were not exposed by the primitives developed to this point. For example:

```
if e0
     then (e1; ... ; ek; A←.B+.C*.D)
     else (ek+1; ... ; en; A←.B+.C*.E).
```

Clearly both assignments to A can be evaluated by the code sequence

```
MULT   T,C
ADD    T,B
STORE  T,A
```

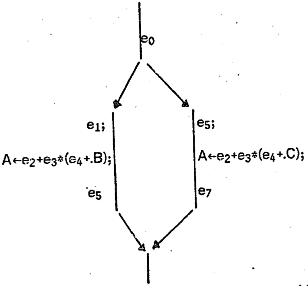
where on the <u>then</u> and <u>else</u> branches T has been loaded with D and E respectively. Hence, an optimization strategy for this expression consists of replacing the expression with:

(if e_0
then (e_1; ... ; e_k; T←.D)
else (e_{k+1}; ... ; e_n; T←.E); A←.B+.C*.T).

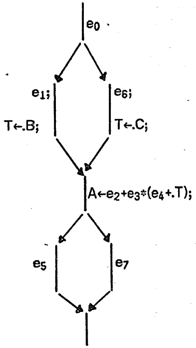
The ω set for forked control environments described in Chapter II did not expose this optimization since the assignment expressions are not congruent.

The preceding optimization technique depends on the fact that, although the expressions are not quite identical, they are very close to being identical. Hence one of the properties of a similarity function must be the quantification of this notion of "closeness". In addition, the measurement must be sufficiently fine so that it can distinguish degrees of "closeness" rather than compute a simple boolean indicating "close" or "not close". These observations point out a major distinction between the similarity concept and the primitives developed in Chapter II. Those primitives served to expose the feasible optimizations in a program. For the most part, the optimizations were not only feasible but also desirable since, in general, they reduced both object code size and execution time at the potential expense of prolonging the life-time of temporary memory locations. However, many of the optimization techniques described in this chapter will effect more significant trade-offs between code size, execution time, and temporary storage. As a result, similarity functions are to provide the necessary data in terms of which the desirability of initiating a particular optimization can be measured. The feasibility of the optimizations exposed by similarity will be described in terms of the primitives developed in Chapter II.

This aspect of a similarity function is illustrated by the following example:



Assume that the previous optimization strategy which moved part of the evaluation to a point after the merge is precluded by the fact that e_5 and e_7 produce side-effects on constituents of the assignment to A and so block the forward motion of the assignments within their respective linear blocks. A possible optimization of this example consists of replacing the control expression above with



The jump to and return from the common evaluation point of the expression $A \leftarrow e_2 + e_3 * (e_4 + .T)$ incurs an overhead cost associated with the execution of the additional control. This overhead did not occur in the preceding example since the common code sequence was entered after the merge. The decision to invoke this optimization must be made in terms of the trade-off between (1) the amount of code saved by the common code sequence, and (2) the space and time overhead incurred by the introduction of additional code for control. The measurements involving code size are computable at compile time. Measurements involving execution time can at best only be estimated at compile time with assumptions relating to factors such as depth of loop nesting and equi-probable selection of parallel branches in forked control environments. If the similarity function is to be a useful tool for describing optimization strategies such as those listed above, then it must provide the information required to evaluate these trade-offs.

The two preceding examples involved optimization techniques that altered the control flow of the original program. The second example models the standard programming construct of a call to a single parameter procedure. This latter observation opens up a whole spectrum of applications for the similarity concept. The fact that the two assignments to A are on parallel branches of a forked construct is not essential to the feasibility of the optimization strategy. The effectiveness of invoking the optimization technique is measured in terms of the trade-off between the cost of the parameter mechanisms and calling sequence overhead and the cost of the paramaterized code sequences which the "almost identical" expressions can share.

EXAMPLES OF SOME SIMILARITY FUNCTIONS

Chapter II presented a minimal set of requirements to be satisfied by a similarity function. They reflect the notion that members of this family of functions are to "measure" the degree of identity of the superstructure of pairs of expressions.

σ is a similarity function only if

(1) σ: ExE $\rightarrow [0,\infty)$,

(2) $\sigma(e_1,e_2)=\emptyset$ iff $e_1 \cong e_2$, and

(3) $\sigma(e_1,e_2) = \sigma(e_2,e_1)$ for all $e_1,e_2 \in E$. (symmetric)

These requirements alone, however, do not suffice to convey the notion of a measurement of "almost-congruence". For example the function:

$$C(e,e') = \begin{cases} \emptyset & \text{if } e \cong e' \\ 1 & \text{otherwise} \end{cases}$$

satisfies requirements (1)-(3) but conveys precisely the same information as the "≅" relation. A similarity function must provide a more selective measurement.

A first approximation to a similarity function is provided by the function F defined below. F does a coordinated tree walk returning from each corresponding pair of nodes which are not congruent with a value of 1. The following algorithm† gives a precise description of F.

----

† These algorithms are presented in pseudo-Bliss. Their translation to "true" Bliss would require specification of data formats to a level of detail exceeding our present needs.

```
routine F(e,e')=

        ! N is the set of non-terminals of E.
        ! T is the set of terminals. L is the set
        ! of literals and I the set of names.

    begin local S;
        if e ∈ N xor e' ∈ N then return 1;

        if e ∈ N then
            begin
                if e[operator] ≠ e'[operator]
                    then return 1;

                if e[# of operands] ≠ e'[# of operands]
                    then return 1;

                S←0;
                incr I from 1 to e[# of operands] do
                    S←.S+F(e[operand_i], e'[operand_i]);
                return .S;
            end;

        if e ∈ L xor e' ∈ L then return 1;

        if e ∈ L
            then return literalvalue(e) ≠ literalvaiue(e');

        return name(e) ≠ name(e')

    end;
```

<u>Example</u>

Let e_0: .E+.B*.D, e_1: .A+.B*.C, e_2: .A+.B*(.C+.E), e_3: .A+.B*(.D+.E).
Then $F(e_0,e_1)=2$, $F(e_1,e_2) = F(e_1,e_3) = F(e_2,e_3) = 1$.

In effect, F provides a count of the number of dissimilar nodes in a pair of expressions. It does not, however, provide a very selective measure since it does not distinguish between the pairs (e_1,e_2) and (e_2,e_3). The expressions e_2 and e_3 are more alike in some intuitive sense because their dissimilarities occur at a lower level in the tree. A function which could distinguish between such pairs would be preferable since differences at a greater depth correspond to longer identical code sequences.

The following function G incorporates the weighting factor of tree depth by a slight modification of F. The difference between F and G occurs at the point of the recursive call where the reciprocal of the number of operands is inserted as a multiplicative factor. The procedure for G is:

```
routine G(e,e')=

    begin local S;
        if e ∈ N xor e' ∈ N then return 1;

        if e ∈ N then
            begin
                if e[operator] ≠ e'[operator]
                    then return 1;

                if e[# of operands] ≠ e'[# of operands]
                    then return 1;

                S←0;
                incr I from 1 to e[# of operands] do
                    S←.S+(1/e[# of operands])*
                        G(e[operand_i], e'[operand_i]);
                return .S;
            end;

        if e ∈ L xor e' ∈ L then return 1;

        if e ∈ L
            then return literalvalue(e) ≠ literalvalue(e');

        return name(e) ≠ name(e')

    end;
```

The function, G, produces a finer measure on pairs of expressions than F. For example, let e_1, e_2 and e_3 be as defined in the preceding example. $G(e_1,e_2)=0.25$ and $G(e_2,e_3)=0.125$ whereas $F(e_1,e_2)=F(e_2,e_3)=1.0$. However, if we define e: (.A+1)*(.A+2)*(.A+3), e': (.B+1)*(.B+2)*(.B+3), and e": (.A+1)*(.B+2)*(.C+3), then G(e,e') = G(e',e") = G(e,e") = 0.5. Hence G does not reflect the fact that the pair e,e' can be implemented as a single-parameter subroutine whereas a subroutine implementation of the pair e,e" requires three parameters.

The final similarity function presented here is the one that has been implemented in the optimization pass which produces the examples in Chapter IV. SIGMA initializes the variables NPARMS to zero and COSTAV to the estimated object code size of the expression e. Whenever the recursive subroutine S encounters a pair of dissimilar subexpressions of e and e' it calls the subroutine TRYPARMS. The subroutine TRYPARMS determines if a new parameter must be created. If so, it increments NPARMS by one and decreases COSTSAV by the estimated object code size of the parameter subexpression of e. When control returns to SIGMA from S, the variable NPARMS contains the number of parameters necessary to evaluate the pair e, e' by a common code sequence and COSTSAV contains an estimate of the size of the object code sequence sharable by the expressions.

```
routine SIGMA(e,e')=

        ! The subroutine S does a coordinated tree walk on the
        ! expressions e and e' setting the variables NPARMS to
        ! the number of parameters and COSTSAV to the amount
        ! of code saved by the shared code sequences. The
        ! subroutine TRYPARMS (not defined here) increments
        ! NPARMS and decrements COSTSAV by e[cost] if a new
        ! parameter must be created. e[cost] is the amount
        ! of code necessary to evaluate the entire expression e.
        ! e[count] is the number of formally identical
        ! instances of this expression.

    begin own NPARMS, COSTSAV, M;
```

```
routine S(e,e')=

    begin
        if e ∈ N xor e' ∈ N
            then return TRYPARMS(e,e');

        if e ∈ N then
            begin
                if e[operator] ≠ e'[operator]
                    then return TRYPARMS(e,e');

                if e[# of operands] ≠ e'[# of operands]
                    then return TRYPARMS(e,e');

                if e ≅ e' then return;

                incr I from 1 to e[# of operands] do
                    S(e[operand_i],e'[operand_i]);
                return
            end;

        if e ∈ L and e' ∈ L
            then return
                if literalvalue(e) ≠ literalvalue(e')
                    then TRYPARMS(e,e');

        if e ∈ I and e' ∈ I
            then return
                if name(e) ≠ name(e')
                    then TRYPARMS(e,e');

        TRYPARMS(e,e')
    end;

if e ≅ e' then return Ø;
NPARMS←Ø; COSTSAV←e[cost];
S(e,e');
M←e[count] + e'[count];
(.M*.NPARMS+.M+1)/((.M-1)*.COSTSAV)
end;
```

The final expression in the body of SIGMA requires some explanation. The numerator is the estimated cost in code size of the overhead required to set up parameters (.M*.NPARMS), call (+.M), and return (+1) from a similarity-created subroutine. The denominator is the amount of code saved by replacing M-1 of the expressions with calls to a common sequence of code. Hence if SIGMA(e,e')<1, then code size will be reduced by implementing e and e' as calls on a common subroutine.

The application of the similarity function SIGMA (more precisely its subroutine S) partitions an expression into a body and a collection of parameter expressions. In subsequent discussions, body(e) refers to the expression resulting from the removal of the parameter nodes in e, and parms(e) refers to the set of sub-expressions identified by S as parameters of e.

Example

Define: e_1: (.A+1)*(.A+2)*(.A+3), e_2: (.B+1)*(.B+2)*(.B+3),
e_3: (.A+1)*(.B+2)*(.C+3), e_4: .A+.B*(.C+.E),
e_5: .A+.B*(.D+.E), e_6: .A+.B, e_7: .A+.C

The following table shows the values returned from F, G, and SIGMA.

	F	G	SIGMA
e_1,e_2	3.Ø	Ø.5	Ø.625 = (2*1+2+1)/8
e_1,e_3	3.Ø	Ø.5	1.125 = (2*3+2+1)/8
e_4,e_5	1.Ø	Ø.125	1.5 = (2*1+2+1)/4
e_6,e_7	1.Ø	Ø.5	2.5 = (2*1+2+1)/2

It must be emphasized that we have presented an example of a particular similarity function that has produced extremely interesting results in our optimization pass. There are a variety of such functions each sharing common basic characteristics with SIGMA. Indeed this particular similarity function ignores the execution time overhead resulting from introducing subroutine linkages and so identifies those optimizations that minimize object code size as "desirable" without regard to their effect on execution time.

Throughout the remainder of this chapter, the existence of a similarity function, σ, whose essential characteristics are mirrored by SIGMA and its subroutine S will be assumed. The following sections will present a collection of optimization techniques defined in terms of similarity and the primitives defined in Chapter II.

CONVERTING EXPRESSIONS TO SUBROUTINES

A programmer selects macros and procedures to define in his program on the basis of logically coherent units of computation. Macros (expanded in line) save time by avoiding execution time linkage and parameter passing mechanisms at the expense of increasing object code size. Closed procedures, on the other hand, reduce object code size at the expense of run-time overhead. The decision to choose a macro over a procedure or vice versa is typically made on the basis of some rough and usually intuitive estimate of the ratio of the object code size to the frequency of occurence.

An optimization strategy described in terms of similarity eliminates this decision for the programmer by expanding the simple (i.e. non-recursive) procedures in line. The decision to close some of these procedures or portions of them is made on the basis of information collected by a similarity function. This process, that identifies expressions to be implemented as closed subroutines, operates only on the form of the program. As a result it can identify computationally coherent sequences which do not possess a logical coherence that would lead to their identification as a macro or procedure by the programmer. The examples in Chapter IV demonstrate that these situations occur in real programs!

Similarity can be used to identify those expressions which occur sufficiently often that their implementation as subroutines will reduce object code size. The similarity function SIGMA returns a value which is the ratio of the overhead to amount of code saved by creating a subroutine out of a pair of expressions. If that ratio is less than 1, then a savings in code size results.

As we mentioned above, the value returned from SIGMA(e,e') indicates whether a subroutine creation is desirable, however it does not imply that such a creation is also feasible. Consider the example of an expression e that is to be implemented as a subroutine with a single parameter. Furthermore assume that for one of the calls on e the actual parameter expression contains .X as an operand. Finally assume that the subroutine implementation adopts a call-by-value convention for parameters. Thus, the value of X will be accessed during the parameter evaluation prior to evaluation of the expression e. If the expression e alters the value of X prior to the original evaluation point of the parameter expression, then the data flow semantics for e have been violated. Furthermore since the parameter expression can appear within a loop contained in e, it is not sufficient that no re-evaluation of X precede the parameter expression. This set of observations can be summarized as follows:

> A subroutine creation from the expression e and e' is feasible if $p \in$ prolog $(\beta) \cap$ epilog(β) $\forall$ $p \in$ parms(e), $p' \in$ prolog$(\beta') \cap$ epilog(β') $\forall$ $p' \in$ parms(e'), where $\beta = \beta|$cover(e) and $\beta' = \beta'|$cover(e').

The criterion that SIGMA(e,e')<1 is sufficient to guarantee that the subroutine implementation of e and e' will reduce code size. It is quite reasonable to define a controlling heuristic that weighs the amount of code saved against the storage required for parameters (especially if they are passed in registers) and some expected value of increased execution time. This observation argues for a function DELTA which is SIGMA dependent and encodes the heuristics to be applied in deciding the desirability of implementing a set of expressions as a subroutine. Hence the decision to evoke these optimizations will be made by a predicate of the form: σ(e,e') < δ(e,e'). Logically the function δ is defined in terms of the expressions e and e'. However, in an implementation of δ, one expects the subroutine DELTA to share information collected by SIGMA. In particular DELTA should have access to the own variables NPARMS and COSTSAV. A straightforward extension of the notion of strong similarity makes the dependence of δ on the expressions e and e' explicit: e $\simeq$ e' iff σ(e,e') < δ(e,e'). The examples in Chapter IV demonstrate the results that occur when δ is set to a constant value of 1.0. We will refer to this optimization technique which creates subroutines from sets of strongly similar expressions as the strong similarity subroutine optimization (S^3 optimization). Throughout the remainder of the chapter, we will fix the interpretation of δ to be 1.0 and as a result $e \simeq e'$ iff $\sigma(e,e') < \delta \equiv 1.0$.

PARTIAL POST-EVALUATION IN FORKS

The S^3 optimization will generally use subroutine call and return instructions in its implementation. The next few sections point out cases that simplify the linkage mechanism.

Reconsider an example presented earlier in the chapter

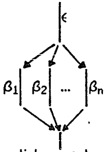
```
if e0
    then (e1; ... ; ek; A←.B+.C*.D)
    else (ek+1; ... ; en; A←.B+.C*.E).
```

The two assignments to A are strongly similar making it feasible to apply an S^3 optimization to them. However, the optimization:

```
(if e0
    then  (e1; ... ; ek; T←.D)
    else (ek+1; ... ; en; T←.E); A←.B+.C*.T).
```

avoids a subroutine mechanism. The following general description applies to optimizations of this form:

Given an n-way branching environment



a partial post-evaluation of the strongly similar expressions $e_1 \in \beta_1$, ... , $e_n \in \beta_n$ is feasible if $body(e_1) \in postlog(\beta_1)$, ... , $body(e_n) \in postlog(\beta_n)$ and $p \in prolog(\beta|cover(e_i)) \cap epilog(\beta|cover(e_i))$ $\forall$ $p \in parms(e_i)$, $1 \le i \le n$.

This optimization is accomplished without adding additional linkage mechanism and so saves space without increasing execution time.

WASP-WAISTING -- REVISITED

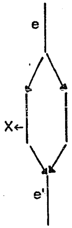
In Chapter II a brief reference was made to an optimization we called "wasp-waisting". A representative example is

<u>if</u> e_0
    <u>then</u> $(e_1; ...; e_i; ... ;e_j)$
    <u>else</u> $(e_{j+1}; ... ; e_k; ... ; e_n)$

where $e_i \cong e_k$. The optimization strategy for this example consisted of replacing the expressions e_i and e_k with calls to a common subroutine. The S^3 optimization technique extends this strategy to cases in which the expressions e_i and e_k are not formally identical but only strongly similar. The feasibility requirements for a wasp-waisting optimization are identical to the requirements for an S^3 optimization. The difference between the two optimizations lies in the possibility of implementing the subroutine call by a simple branch instruction and the return by retesting the selector.

GENERATING CONDITIONAL SUBROUTINES

Consider the following example



where $e \simeq e'$ and e' is an expression involving X. Compile time data flow analysis clearly indicates that e and e' are not redundant expressions because of the potential assignment to X. At run time, on the other hand, whenever control passes down the right hand branch, the post-merge evaluation of e' is unnecessary.

An optimization strategy consists of replacing e and e' by calls on a subroutine which conditionally executes depending on a boolean value. The form of the subroutine is

<u>if</u> <boolean> <u>then</u> (<subr-body>; <boolean>←false).

The boolean is set to true initially and reset to true at the point of the store into X or anywhere on the left branch of the fork. This optimization technique saves space as does any S^3 optimization. It also saves time presuming that the time to evaluate the subroutine body exceeds the time involved in setting and testing the boolean.

GENERATING LOOPS

The optimizations described by similarity to this point have involved the introduction of additional branches and subroutine calls. This section will investigate two optimization strategies described using similarity which introduce loops into a program.

Consider a compound expression e: (e_1; e_2; ... ; e_n) in which $e_i \simeq e_j$, $1 \le i,j \le n$ and (for simplicity) assume the set parms(e_i) is a singleton $\{p_i\}$, $1 \le i \le n$.

Case 1:

Independent of the relationship between the corresponding parameters of the expressions, this compound expression can be implemented by a control environment which models the Algol <u>for</u> statement

<u>for</u> I := p_1, p_2, ... , p_n <u>do</u> e"<I>

where e"<I> is body(e_i) in which parm(e_i) = I.

Case 2:

> If in addition the parameter expressions are such that $p_{i-1}-p_i = \Delta p$, $1 \le i < n$ and $\Delta p \in$ prolog(β|e) $\cap$ epilog(β|e) then the compound expression e can be implemented by
>
> <u>incr</u> I <u>from</u> p_1 <u>to</u> p_n <u>by</u> Δp <u>do</u> e"<I>
>
> where e"<I> is as described in case 1.

The restriction to single parameter subroutines can be removed by updating a set of control variables, one for each parameter, on each iteration of the loop. Both examples reduce object code size and replace the subroutine linkage mechanism of the S^3 optimizations with loop control. In addition case 2 reduces both time and space costs by incrementally computing successive parameters.

The pair of optimization strategies relates quite closely to our discussion of strength reduction in Chapter II. In particular we sought a technique for discovering a relation F such that given a pair of expressions e and e', F(e,Δe)$\equiv$e'. Both cases 1 and 2 provide solutions. The relation F is precisely the loop body expression e" and the parameter Δe is the loop variable I. The fact that e' $\simeq$ e guarantees that the size of the object code to compute F(e,Δe) is less that that required to evaluate e' in the ususal manner. Case 2 also demonstrates the discovery of an inductive relationship among the expressions e_1, e_2, ..., e_n without assumptions on the form of the expressions. In particular no restriction to polynomial expressions is required.

SIMILARITY AND ITERATIVE TECHNIQUES

The next optimization described in terms of similarity arises often in algorithms concerned with various forms of iterative analysis. A simple example motivates the usefulness of the optimization strategy.

The following algorithm accumulates in S the trapezoidal approximation to the definite integral of F over the interval $[X_0, X_n]$:†

```
incr I from .X0+ΔX to .Xn by ΔX do
     S←.S + ((F(.I-ΔX) + F(.I))/2) * ΔX.
```

The important item to note here is that on the k-th iteration of the loop the value of F(.I-ΔX) is precisely the same as the value of F(.I) on the (k-1)-st iteration. Recognizing this relationship between F(.I-ΔX) and F(.I), an optimization strategy that requires only one evaluation of F per iteration is given by:

```
if .X0+ΔX leq .Xn then OLDF←F(.X0);
incr I from .X0+ΔX to .Xn by ΔX do
    begin
        NEWF←F(.I);
        S←.S+((.OLDF+.NEWF)/2)*ΔX;
        OLDF←.NEWF
    end.
```

A description of the expressions in a loop body for which this optimization is feasible is given by:

----

† The loop models the "calculus-text" description of the trapezoidal rule, even though a numerical analyst would not program it in this form.

Given a loop <u>incr</u> I <u>from</u> e_0 <u>to</u> e_1 <u>by</u> e_2 <u>do</u> e_3 where f and g are subexpressions of e_3, $f \simeq g$, let parms(f)={p} and parms(g)={p+e_2}. Then it is feasible to eliminate the evaluation of f on each iteration of the loop (and replace it by the old value of g) if body(f) $\in$ prolog(β_3) $\cap$ epilog(β_3) and body (g) $\in$ prolog(β_3) $\cap$ epilog(β_3).

The restriction that body(f) and body(g) be loop invariant is equivalent to stating that on any two iterations of the loop the evaluations of f(c) and g(c) produce the same value for a fixed parameter c.

SUMMARY

In the preceding sections the similarity function has been used to decribe a variety of apparently unrelated optimization strategies. This fact reflects its usefulness as a unifying primitive which can be employed in describing a wide range of concepts. Indeed this property may be sufficient justification in itself for proposing the similarity notion.

However, the ensuing chapter presents a strong case that similarity has very practical application in an optimizing compiler. The reductions in code size that result from application of the S^3 optimization alone are remarkable. In addition the S^3 optimizations demonstrate interesting results in identifying computationally coherent expressions from analysis of a program's form. Sometimes these computationally coherent expressions correspond to those which the programmer considered logically coherent and sometimes not.

CHAPTER IV

EXAMPLES

Chapters II and III propose a number of optimizations. This chapter discusses the relative significance of some of these optimizations in some specific cases. A program is described which implements both the optimizations described in Chapter II and the S^3 optimization (using the particular similarity function, SIGMA, described in Chapter III). The chapter is subdivided into two parts: (1) a description of the program and the form of its output and (2) a discussion of a set of examples which show the effect of the optimizations.

The purpose of the program is the evaluation of the effectiveness of the S^3 optimization as compared with the classical optimizations of Chapter II. Therefore, it was not to our purpose to construct a complete compiler. However, since the Bliss-10 compiler already accepts Bliss syntax and produces PDP-10 machine code, we have chosen the PDP-10 as the target machine and will demonstrate shortly that the estimates produced by our program correspond to the actual number of machine language instructions produced by Bliss-10. To enable the comparison between the classical optimizations and S^3, the program is constructed so that programs may be compiled with various subsets of the optimizations enabled.

KATE

The program KATE† is a translator from Bliss to a three-address code. KATE may be thought of as four modules each of which works on a representation of the program and global information prepared by other modules. The first module, LEXSYN, performs lexical and syntactic analysis on the source text, builds a symbol table and produces a tree representation of the program as described in Chapter I. The second module, FLOW, implements the optimizations described in Chapter II except for strength reduction. Ommision of strength reduction does not effect the comparison between the classical optimizations and S^3 since it has relatively little effect on object code size. The third module, S^3, implements the S^3 optimization described in Chapter III. The fourth module, CODE, produces a three address code and an estimate of the number of PDP-10 instructions that would result if the three address code were translated into real machine code. The following diagram illustrates the possible paths which KATE can follow in translating source text to three address code.

---

† For those who feel that acronyms require interpretations, we suggest Algorithmic Translating Engine. The K, of course, is silent.

```
           ┌→ LEXSYN ─────────────→ ┐
source →   ├→ LEXSYN ──→ FLOW ───→  ├→ CODE → 3-address code
           └→ LEXSYN ──→ FLOW ──→ S3 →
```

Although the FLOW module can be thought of as a separate pass over the representation produced by LEXSYN, in fact, it processes the tree from the inside out while the tree is being built by LEXSYN. The FLOW module is invoked by LEXSYN at the completion of each linear block to build the prolog, epilog, and postlog sets. As syntactic analysis is completed for each control environment, flow is called to invoke the various optimization strategies. In particular a node representing a forked control expression points to the α and ω sets for the expression and each node representing a looping control expressions points to the χ (loop invariant expressions) and ρ (cyclic re-evaluations) set for that expression. S^3, on the other hand, makes a completely separate pass since it must have information on <u>all</u> occurences of strongly similar expressions to make its decisions. LEXSYN, FLOW, and S^3 implement the primitives developed in Chapters II and III. However, CODE requires more detailed explanation of its output to facilitate understanding of the examples.

CODE

The CODE module translates the tree representation of the program into a three address code. The machine code operations which are used in CODE were selected to facilitate an accurate estimate of the number of PDP-1Ø[P7Ø] machine language instructions that would result from the three address code. Again, the PDP-1Ø was chosen because the Bliss-1Ø compiler enabled us to verify the accuracy of the estimates made by CODE.

The three address code is formatted as:

operator operand$_1$, operand$_2$, operand$_3$.

Each operator has a fixed number (Ø,1,2,3) of operands. The operand of an instruction can be:

(1) a name -- e.g. X
(2) the value pointed to by a name -- e.g. .X
(3) a level of indirection on (2) -- e.g. ..X
(4) a constant
(5) a label.

The following table lists the machine code operations and describes their semantics. In general e_1 and e_2 are the operands of the opcode and e_3 is the result returned to the parent node of the subnode which produced the result.

opcode	operands	semantics	
ADD	e_1,e_2,e_3	$e_3 \leftarrow e_1+e_2$	
SUB	e_1,e_2,e_3	$e_3 \leftarrow e_1-e_2$	
MUL	e_1,e_2,e_3	$e_3 \leftarrow e_1 * e_2$	
LTSH	e_1,e_2,e_3	$e_3 \leftarrow e_1 \uparrow e_2$	
RTSH	e_1,e_2,e_3	$e_3 \leftarrow e_1 \uparrow (-e_2)$	
DIV	e_1,e_2,e_3	$e_3 \leftarrow e_1 / e_2$	
MOD	e_1,e_2,e_3	$e_3 \leftarrow e_1$ <u>mod</u> e_2	
GTR	e_1,e_2,e_3	$e_3 \leftarrow e_1 > e_2$	
LEQ	e_1,e_2,e_3	$e_3 \leftarrow e_1 \leq e_2$	
LSS	e_1,e_2,e_3	$e_3 \leftarrow e_1 < e_2$	
GEQ	e_1,e_2,e_3	$e_3 \leftarrow e_1 \geq e_2$	
EQL	e_1,e_2,e_3	$e_3 \leftarrow e_1 = e_2$	
NEQ	e_1,e_2,e_3	$e_3 \leftarrow e_1 \neq e_2$	
AND	e_1,e_2,e_3	$e_3 \leftarrow e_1$ <u>and</u> e_2	
ANDCR	e_1,e_2,e_3	$e_3 \leftarrow e_1$ <u>and</u> <u>not</u> e_2	
ANDCL	e_1,e_2,e_3	$e_3 \leftarrow$ <u>not</u> e_1 <u>and</u> e_2	
ANDCB	e_1,e_2,e_3	$e_3 \leftarrow$ <u>not</u> e_1 <u>and</u> <u>not</u> e_2	
OR	e_1,e_2,e_3	$e_3 \leftarrow e_1$ <u>or</u> e_2	
ORCR	e_1,e_2,e_3	$e_3 \leftarrow e_1$ <u>or</u> <u>not</u> e_2	
ORCL	e_1,e_2,e_3	$e_3 \leftarrow$ <u>not</u> e_1 <u>or</u> e_2	
ORCB	e_1,e_2,e_3	$e_3 \leftarrow$ <u>not</u> e_1 <u>or</u> <u>not</u> e_2	
LD	e_1,e_2,e_3	$e_1 \leftarrow e_2$ (e_3 is value of exp.)	
LDN	e_1,e_2,e_3	$e_1 \leftarrow -e_2$ (")	
LDC	e_1,e_2,e_3	$e_1 \leftarrow$ <u>not</u> e_2 (")	
XCT	e_1,e_2	execute inst. at e_1+e_2	
PARM	e_1	set up parameter e_1	assumes stack
DPARM	e_1	deallocate e_1 parameters	discipline
CALL	e_1,e_2	save PC+1; PC $\leftarrow e_1$; value in e_2	
RTRN		PC $\leftarrow$ saved value	
BR	e_1	PC $\leftarrow e_1$	
BRT	e_1,e_2	<u>if</u> e_1 <u>then</u> PC $\leftarrow e_1$	
BRF	e_1,e_2	<u>if</u> <u>not</u> e_1 <u>then</u> PC $\leftarrow e_2$	
INC	e_1	$e_1 \leftarrow e_1+1$	
DEC	e_1	$e_1 \leftarrow e_1-1$	
SSCAL	e_1,e_2,e_3	save e_2; PC $\leftarrow e_1$; value in e_3	

An example of output from KATE demonstrates the use of these operations.

```
begin own I,V[1Ø],X,Y,Z,A,B,C,D,F;
  X←-.Y-.Z; V[2*.I]←-.Y-.Z;
  Z←.A*.B+.C*.D; F(.Z)
end eludom
```

```
ADD      .Y,.Z,-.T1        2*
LDN      X,.T1,-.T1        1*
MUL      2,.I,.T2          2*
ADD      V,.T2,.T2         Ø*
LDN      .T2,.T1,-.T1      1*
MUL      .A,.B,.T1         2*
MUL      .C,.D,.T2         2*
ADD      .T1,.T2,.T1       1*
LD       Z,.T1,.T1         1*
PARM     .Z                1*
CALL     F,.T0             1*
DPARM    1                 1*

         TOTAL COST= 15
```

The following points should be noted:

(1) The code generators are table driven. They attempt to do peephole optimization on a very local level. For example, note that the expression -.X-.Y was converted to -(.X+.Y).

(2) CODE allocates temporary storage (T_1 and T_2) in a straightforward manner. If a temporary location is allocated to a redundant expression, it remains reserved for the value of that expression until the last occurence of the use of that value. In the example above the product 2*.I was formed in T_2 since the last use of the value in T_1 followed the index computation.

(3) Note that there is only one instruction for transferring the value stored in one memory location to another. In particular a LD instruction can correspond to (a) loading a temporary -- LD T_0,.A, (b) storing a temporary into a user-defined memory location -- LD A,.T_0, or (c)

transferring the contents of one memory location to another -- LD A,.B.

(4) The column of numbers to the right contains estimates of the number of PDP-10 instructions required by each operation. An actual PDP-10 code sequence for this example is:

```
MOVE     T1,Y
ADD      T1,Z
MOVNM    T1,X
MOVE     T2,I
IMULI    T2,2      (or: ASH    T2,1)
MOVNM    T1,V(T2)
MOVE     T1,A
IMUL     T1,B
MOVE     T2,C
IMUL     T2,D
ADD      T1,T2
MOVEM    T1,Z
PUSH     $S,Z
PUSHJ    $S,F      ;$S points to the stack
SUB      $S,[1000001]
```

In particular notice that KATE estimates 0 as the cost of the operation $V+T_2$ since the addition can be accomplished by indexing.

(5) The indentation exhibiting the columns of asterisks indicates the nesting of linear blocks and the number at the base of a column is the cumulative total of the code size for that linear block. This facilitates comparison of the number of instructions in critical regions such as inner loops.

VALIDATION OF KATE'S ESTIMATES

The estimates of object code size are generated on an instruction by instruction basis. Corresponding to each machine code operation produced by KATE there is a 12x12 table. An index into the table is computed by analyzing each operand into one of twelve states:

Ø, 1, -1, L, N, .N, ..N, T, .T, ..T, .T', ..T'.

L is a literal (absolute value greater than 1), N is a user defined storage location, T is a compiler defined temporary (whose contents may be destroyed by the execution of the instruction), and T' is a temporary whose contents must be preserved.

In order to demonstrate that the numbers produced by KATE are in fact reasonable when applied to sequences of code, a comparison was made between the estimates produced by KATE and actual PDP-1Ø machine code produced by Bliss-1Ø. Both compilers were run with all optimization turned off. This was done since even though the two compilers apply different sets of optimizations, they both produce straightforward, simple machine code with all optimizations turned off. We have selected two examples (to be examined in more detail for other purposes later in the chapter) to exemplify the results. The first example is a large sub-program taken from the Bliss-1Ø compiler itself. Bliss-1Ø produces 983 PDP-1Ø instructions. The estimate produced by KATE is 979 instructions. The difference is less than Ø.5%.

A second example, an implementation of the quadratic formula, is small enough to be reproduced in its entirety. The source text is the following:

```
begin
  macro
    POSROOT(A,B,C)=(-B)/(2*A)+SQRT(DISC(A,B,C))/(2*A)$,
    NEGROOT(A,B,C)=(-B)/(2*A)-SQRT(DISC(A,B,C))/(2*A)$,
    DISC(A,B,C)=B*B-4*A*C$;
  external SQRT;
  global ERROR,R1,R2;
  routine ROOT(X,Y,Z)=
    begin
      if DISC(.X,.Y,.Z) lss Ø then ERROR←1 else
      if DISC(.X,.Y,.Z) eql Ø then
        (R1←-.Y/(2*.X); R2←-.Y/(2*.X))
      else (R1←POSROOT(.X,.Y,.Z);R2←NEGROOT(.X,.Y,.Z));
    end;
end eludom
```

The output on the left column of the next page is produced by KATE; the output on the right is produced by Bliss-1Ø. In the Bliss-1Ø output: A ≡ -4($F), B ≡ -3($F), and C ≡ -2($F).

KATE

```
ROOT:
     MUL     .Y..Y..T$1            2  *
     MUL     4..X..T$2             2  *
     MUL     .T$2..2..T$2          1  *
     SUB     .T$1..T$2..T$1        1  *
     LSS     .T$1.0..T$1           0  *
     BRF     .T$1.L$1              1  *
     LD      ERROR.1.1             2     *
     BR      L$2                   1     *
                                         3
L$1:
     MUL     .Y..Y..T$1            2     *
     MUL     4..X..T$2             2     *
     MUL     .T$2..2..T$2          1     *
     SUB     .T$1..T$2..T$1        1     *
     EQL     .T$1.0..T$1           0     *
     BRF     .T$1.L$3              1     *
     MUL     2..X..T$1             2        *
     DIV     .Y..T$1.-.T$1         2        *
     LDN     R1..T$1.-.T$1         1        *
     MUL     2..X..T$1             2        *
     DIV     .Y..T$1.-.T$1         2        *
     LDN     R2..T$1.-.T$1         1        *
     BR      L$4                   1        *
                                            11
L$3:
     MUL     2..X..T$1             2        *
     DIV     .Y..T$1.-.T$1         2        *
     MUL     .Y..Y..T$2            2        *
     MUL     4..X..T$3             2        *
     MUL     .T$3..2..T$3          1        *
     SUB     .T$2..T$3..T$2        1        *
     PARM    .T$2                  1        *
     CALL    SQRT..T$0             1        *
     DPARM   1                     1        *
     MUL     2..X..T$2             2        *
     DIV     .T$0..T$2..T$0        1        *
     SUB     .T$0..T$1..T$0        1        *
     LD      R1..T$0..T$0          1        *
     MUL     2..X..T$1             2        *
     DIV     .Y..T$1.-.T$1         2        *
     MUL     .Y..Y..T$2            2        *
     MUL     4..X..T$3             2        *
     MUL     .T$3..2..T$3          1        *
     SUB     .T$2..T$3..T$2        1        *
     PARM    .T$2                  1        *
     CALL    SQRT..T$0             1        *
     DPARM   1                     1        *
     MUL     2..X..T$2             2        *
     DIV     .T$0..T$2..T$0        1        *
     ADD     .T$1..T$0.-.T$1       1        *
     LDN     R2..T$1.-.T$1         1        *
                                            36
L$4:
                                         54
L$2:
     LD      T$0.0..T$0            1  *
     RTRN                          1  *
                                      66

          TOTAL COST= 66
```

Bliss-10

```
         JSP      12..ENT.0
         MOVE     04.-3($F)
         IMUL     04.-3($F)
         MOVE     05.-4($F)
         ASH      05.2
         IMUL     05.-2($F)
         SUB      04.5
         JUMPGE   04.L2020
         MOVEI    $V.1
         MOVEM    $V.ERROR
         JRST     $S.L1536
L2020:   MOVE     07.-3($F)
         IMUL     07.-3($F)
         MOVE     10.-4($F)
         ASH      10.2
         IMUL     10.-2($F)
         SUB      07.10
         JUMPN    07.L2406
         MOVE     12.-4($F)
         ASH      12.1
         MOVN     05.-3($F)
         IDIV     05.12
         MOVEM    05.R1
         MOVE     $V.-4($F)
         ASH      $V.1
         MOVN     04.-3($F)
         IDIV     04.3
         MOVEM    04.R2
         JRST     $S.L1536
L2406:   MOVE     $V.-3($F)
         IMUL     $V.-3($F)
         MOVE     12.-4($F)
         ASH      12.2
         IMUL     12.-2($F)
         SUB      $V.12
         PUSH     $S.3
         PUSHJ    $S.SQRT
         SUB      $S.[000001..000001]
         MOVE     05.-4($F)
         ASH      05.1
         MOVN     06.-3($F)
         IDIV     06.5
         MOVE     04.-4($F)
         ASH      04.1
         IDIV     $V.4
         ADD      $V.6
         MOVEM    $V.R1
         MOVE     $V.-3($F)
         IMUL     $V.-3($F)
         MOVE     10.-4($F)
         ASH      10.2
         IMUL     10.-2($F)
         SUB      $V.10
         PUSH     $S.3
         PUSHJ    $S.SQRT
         SUB      $S.[000001..000001]
         MOVE     11.-4($F)
         ASH      11.1
         MOVN     12.-3($F)
         IDIV     12.11
         MOVE     05.-4($F)
         ASH      05.1
         IDIV     $V.5
         SUB      $V.12
         MOVNM    $V.R2
L1536:   SETZ     $V.0
         JRST     $S..EXT.0

MODULE LENGTH =67+0
```

The additional instruction (JSP 12,.ENT.Ø) in Bliss-1Ø executes routine entry code. These examples demonstrate that the estimates of object code size produced by KATE are indeed reliable predictions of the actual number of PDP-1Ø machine language instructions that would be generated from the three-address code.

The remainder of Chapter IV discusses three examples which contrast the effect of the classical optimizations and the S^3 optimization introduced in Chapter III. The examples demonstrate the potential of the S^3 optimization for producing significant reductions in object code size. KATE was run in three modes on the examples: (1) NOOPT: no optimizations, (2) ALLBUTSIM: S^3 by-passed, (3) ALLOPT: S^3 included.

QUADRATIC FORMULA

The first example involves three implementations of a program to evaluate the quadratic formula. The main routine, ROOT, is identical in all three implementations. The difference occurs in the evaluation of the square root.

R1:

SQRT is a subroutine implementation of the sequence $\{(x_n^2+A)/(2x_n)\}$ which converges to square root of A. (Newton's method).

```
BEGIN
  MACRO
    POSROOT(A.B.C)=(-B)/(2*A)+SQRT(DISC(A.B.C))/(2*A)$.
    NEGROOT(A.B.C)=(-B)/(2*A)-SQRT(DISC(A.B.C))/(2*A)$.
    DISC(A.B.C)=B*B-4*A*C$;
  FORWARD SQRT;
  GLOBAL ERROR.R1.R2;
  ROUTINE ROOT(X.Y.Z)=
    BEGIN
      IF DISC(.X..Y..Z) LSS 0 THEN ERROR←1 ELSE
      IF DISC(.X..Y..Z) EQL 0 THEN
        (R1←-.Y/(2*.X); R2←-.Y/(2*.X))
      ELSE (R1←POSROOT(.X..Y..Z);R2←NEGROOT(.X..Y..Z));
    END;
  ROUTINE SQRT(X)=
    BEGIN
      LOCAL XI.XJ; GLOBAL EPSILON; MACRO INFINITY=#777777$;
      XI←.X; XJ←INFINITY;
      WHILE (.XJ-.XI) GTP .EPSILON
        DO (XI←.XJ; XJ←(.XI*.XI+.X)/(2*.XI));
      .XJ
    END;
END ELUDOM
```

R2:

SQRT is the expression resulting from expanding the sequence in R1 to the fourth term.

```
BEGIN
  MACRO
    POSROOT(A.B.C)=(-B)/(2*A)+SQRT(DISC(A.B.C))/(2*A)$.
    NEGROOT(A.B.C)=(-B)/(2*A)-SQRT(DISC(A.B.C))/(2*A)$.
    DISC(A.B.C)=B*B-4*A*C$.
    SQ(X)=((X)*(X))$.
    SQRT(X)=((SQ((X)+4)+4*(X))/(2*(4*((X)+4)))+
                 (((4*((X)+4))*(X))/(2*(SQ((X)+4)+4*(X))))$;
  GLOBAL ERROR.R1.R2;
  ROUTINE ROOT(X.Y.Z)=
    BEGIN
      IF DISC(.X..Y..Z) LSS 0 THEN ERROR←1 ELSE
      IF DISC(.X..Y..Z) EQL 0 THEN
        (R1←-.Y/(2*.X); R2←-.Y/(2*.X))
      ELSE (R1←POSROOT(.X..Y..Z);R2←NEGROOT(.X..Y..Z));
    END;
END ELUDOM
```

R3:

SQRT is a macro identical to the subroutine in R1.

```
BEGIN
  MACRO
    POSROOT(A.B.C)=(-B)/(2*A)+SQRT(DISC(A.B.C))/(2*A)$.
    NEGROOT(A.B.C)=(-B)/(2*A)-SQRT(DISC(A.B.C))/(2*A)$.
    DISC(A.B.C)=B*B-4*A*C$.
    SQRT(X)=(XI←X; XJ←INFINITY;
             WHILE (.XJ-.XI) GTP .EPSILON
                DO (XI←.XJ; XJ←(.XI*.XI+(X))/(2*.XI));
             .XJ)$.
    INFINITY=#77777$;
  GLOBAL ERROR.R1.R2.EPSILON;
  ROUTINE ROOT(X.Y.Z)=
    BEGIN LOCAL XI.XJ;
      IF DISC(.X..Y..Z) LSS 0 THEN ERROR←1 ELSE
      IF DISC(.X..Y..Z) EQL 0 THEN
        (R1←-.Y/(2*.X); R2←-.Y/(2*.X))
      ELSE (R1←POSROOT(.X..Y..Z);R2←NEGROOT(.X..Y..Z));
    END;
END ELUDOM
```

The results of running KATE on R1, R2, and R3 are summarized in the following table:

	R1	R2	R3
NOOPT	86	196	108
ALLBUTSIM	52	42	62
ALLOPT	52	42	47

The output produced by KATE in the ALLOPT mode for each example is reproduced on the next three pages.

-- R1 --

```
BEGIN
  MACRO
    POSROOT(A,B,C)=(-B)/(2*A)+SQRT(DISC(A,B,C))/(2*A)$,
    NEGROOT(A,B,C)=(-B)/(2*A)-SQRT(DISC(A,B,C))/(2*A)$,
    DISC(A,B,C)=B*B-4*A*C$;
  FORWARD SQRT;
  GLOBAL ERROR,R1,R2;
  ROUTINE ROOT(X,Y,Z)=
    BEGIN
      IF DISC(.X,.Y,.Z) LSS 0 THEN ERROR←1 ELSE
      IF DISC(.X,.Y,.Z) EQL 0 THEN
        (R1←-.Y/(2*.X); R2←-.Y/(2*.X))
      ELSE (R1←POSROOT(.X,.Y,.Z);R2←NEGROOT(.X,.Y,.Z));
    END;


ROOT:
    MUL     .Y,.Y,.T$1              2  *
    MUL     4,.X,.T$2               2  *
    MUL     .T$2,.Z,.T$2            1  *
    SUB     .T$1,.T$2,.T$1          1  *
    LSS     .T$1,0,.T$2             0  *
    BRF     .T$2,L$1                1  *
    LD      ERROR,1,1               2    *
    BR      L$2                     1    *
                                         3
L$1:
    MUL     2,.X,.T$2               2    *
    DIV     .Y,.T$2,-.T$3           2    *
    EQL     .T$1,0,.T$4             0    *
    BRF     .T$4,L$3                1    *
    LDN     R1,.T$3,-.T$3           1      *
    LDN     R2,.T$3,-.T$3           1      *
    BR      L$4                     1      *
                                           3
L$3:
    PARM    .T$1                    1      *
    CALL    SQRT,.T$0               1      *
    DPARM   1                       1      *
    DIV     .T$0,.T$2,.T$0          1      *
    SUB     .T$0,.T$3,.T$0          1      *
    LD      R1,.T$0,.T$0            1      *
    PARM    .T$1                    1      *
    CALL    SQRT,.T$0               1      *
    DPARM   1                       1      *
    DIV     .T$0,.T$2,.T$0          1      *
    ADD     .T$3,.T$0,-.T$3         1      *
    LDN     R2,.T$3,-.T$3           1      *
                                           12
L$4:
                                         20
L$2:
    LD      T$0,0,.T$0              1  *
    RTRN                            1  *
                                       32

  ROUTINE SQRT(X)=
    BEGIN
      LOCAL XI,XJ; GLOBAL EPSILON; MACRO INFINITY=#777777$;
      XI←.X; XJ←INFINITY;
      WHILE (.XJ-.XI) GTR .EPSILON
       DO (XI←.XJ; XJ←(.XI*.XI+.X)/(2*.XI));
      .XJ
    END;


SQRT:
    LD      XI,.X,.X                2  *
    LD      XJ,777777,777777        2  *
L$6:
    SUB     .XJ,.XI,.T$1            2    *
    GTR     .T$1,.EPSILON,.T$1      1    *
    BRF     .T$1,L$7                1    *
    LD      XI,.XJ,.XJ              2    *
    MUL     .XI,.XI,.T$1            2    *
    ADD     .T$1,.X,.T$1            1    *
    MUL     2,.XI,.T$2              2    *
    DIV     .T$1,.T$2,.T$1          1    *
    LD      XJ,.T$1,.T$1            1    *
    BR      L$6                     1    *
                                         14
L$7:
    LD      T$0,.XJ,.T$0            1  *
    RTRN                            1  *
                                       20

END ELUDOM

      TOTAL COST= 52
```

-- R2 --

```
BEGIN
  MACRO
    POSPOOT(A.B.C)=(-B)/(2*A)+SQRT(DISC(A.B.C))/(2*A)$,
    NEGROOT(A.B.C)=(-B)/(2*A)-SQRT(DISC(A.B.C))/(2*A)$,
    DISC(A.B.C)=B*B-4*A*C$,
    SQ(X)=((X)*(X))$,
    SQRT(X)=((SQ((X)+4)+4*(X))/(2*(4*((X)+4)))+
                (((4*((X)+4))*(X))/(2*(SQ((X)+4)+4*(X)))))$;
  GLOBAL ERROR.R1.R2;
  ROUTINE POOT(X.Y,Z)=
    BEGIN
      IF DISC(.X,.Y,.Z) LSS 0 THEN ERROR←1 ELSE
      IF DISC(.X,.Y,.Z) EQL 0 THEN
       (R1←-.Y/(2*.X); R2←-.Y/(2*.X))
      ELSE (R1←POSPOOT(.X,.Y,.Z);R2←NEGPOOT(.X,.Y,.Z));
    END;


ROOT:
    MUL     .Y,.Y,.T$1              2  *
    MUL     4,.X,.T$2               2  *
    MUL     .T$2,.2,.T$2            1  *
    SUB     .T$1,.T$2,.T$1          1  *
    LSS     .T$1.0,.T$2             0  *
    BRF     .T$2.L$1                1  *
    LD      ERROR.1.1               2    *
    BR      L$2                     1    *
                                         3
L$1:
    MUL     2,.X,.T$2               2    *
    DIV     .Y,.T$2,-.T$3           2    *
    EQL     .T$1.0,.T$4             0    *
    BRF     .T$4.L$3                1    *
    LDN     R1,.T$3,-.T$3           1      *
    LDN     R2,.T$3,-.T$3           1      *
    BR      L$4                     1      *
                                           3
L$3:
    ADD     .T$1.4,.T$4             2      *
    MUL     .T$4,.T$4,.T$5          2      *
    MUL     4,.T$1,.T$6             2      *
    ADD     .T$5,.T$6,.T$5          1      *
    MUL     4,.T$4,.T$4             1      *
    MUL     2,.T$4,.T$6             2      *
    DIV     .T$5,.T$6,.T$6          2      *
    MUL     .T$4,.T$1,.T$4          1      *
    MUL     2,.T$5,.T$5             1      *
    DIV     .T$4,.T$5,.T$4          1      *
    ADD     .T$6,.T$4,.T$6          1      *
    DIV     .T$6,.T$2,.T$6          1      *
    SUB     .T$6,.T$3,.T$2          2      *
    LD      R1,.T$2,.T$2            1      *
    ADD     .T$3,.T$6,-.T$3         1      *
    LDN     R2,.T$3,-.T$3           1      *
                                           22
L$4:
                                         30
L$2:
    LD      T$0,0,.T$0              1  *
    RTRN                            1  *
                                    42

 END ELUDOM

      TOTAL COST= 42
```

-- R3 --

```
BEGIN
  MACRO
    POSROOT(A.B.C)=(-B)/(2*A)+SQRT(DISC(A.B.C))/(2*A)$.
    NEGROOT(A.B.C)=(-B)/(2*A)-SQRT(DISC(A.B.C))/(2*A)$.
    DISC(A.B.C)=B*B-4*A*C$.
    SQRT(X)=(XI←X; XJ←INFINITY;
            WHILE (.XJ-.XI) GTR .EPSILON
               DO (XI←.XJ; XJ←(.XI*.XI+(X))/(2*.XI));
            .XJ)$.
    INFINITY=#77777$;
  GLOBAL ERROR.R1.R2.EPSILON;
  ROUTINE ROOT(X.Y.Z)=
    BEGIN LOCAL XI.XJ;
      IF DISC(.X,.Y,.Z) LSS 0 THEN ERROR←1 ELSE
      IF DISC(.X..Y..Z) EQL 0 THEN
       (R1←-.Y/(2*.X); R2←-.Y/(2*.X))
      ELSE (R1←POSROOT(.X,.Y,.Z);R2←NEGROOT(.X,.Y,.Z));
    END;


ROOT:
     MUL     .Y,.Y,.T$1                2 *
     MUL     4..X,.T$2                 2 *
     MUL     .T$2..2,.T$2              1 *
     SUB     .T$1,.T$2,.T$1            1 *
     LSS     .T$1,0..T$2               0 *
     BRF     .T$2.L$3                  1 *
     LD      ERROR.1.1                 2   *
     BR      L$4                       1   *
                                           3
L$3:
     MUL     2,.X,.T$2                 2   *
     DIV     .Y,.T$2.-.T$3             2   *
     EQL     .T$1.0,.T$4               0   *
     BRF     .T$4.L$5                  1   *
     LDN     R1..T$3.-.T$3             1     *
     LDN     R2..T$3.-.T$3             1     *
     BR      L$6                       1     *
                                             3
L$5:
     SSCAL   S$1.E$1                   1     *
S$1:
     LD      XI,.T$1..T$1              1     *
     LD      XJ.77777.77777            2     *
L$7:
     SUB     .XJ,.XI..T$4              2       *
     GTR     .T$4..EPSILON,.T$4        1       *
     BRF     .T$4.L$10                 1       *
     LD      XI,.XJ,.XJ                2       *
     MUL     .XI..XI..T$4              2       *
     ADD     .T$4,.T$1,.T$4            1       *
     MUL     2..XI..T$5                2       *
     DIV     .T$4..T$5..T$4            1       *
     LD      XJ..T$4,.T$4              1       *
     BR      L$7                       1       *
                                               14
L$10:
     DIV     .XJ,.T$2,.T$4             2     *
     RTRN                              1     *
E$1:
     SUB     .T$4..T$3..T$5            2     *
     LD      R1,.T$5,.T$5              1     *
     SSCAL   S$1,.+1,.T$4              1     *
     ADD     .T$3..T$4,-.T$3           1     *
     LDN     R2,.T$3.-.T$3             1     *
                                            27
L$6:
                                          35
L$4:
     LD      T$0.0..T$0                1 *
     RTRN                              1 *
                                         47
END ELUDOM

       TOTAL COST= 47
```

Notice that the S^3 optimization had no effect on either R1 or R2. In the case of R3, on the other hand, a 25% improvement was realized by applying S^3 optimization. The most interesting comparison, however, is between R1 and R3.

Both programs R1 and R3 represent the same logical structure to the programmer. The decision to declare SQRT as a macro or a routine does not effect that structure. Typically one expects the choice between the two is made in terms of some superficial estimate of the resulting time/space trade-off. The S^3 optimization makes that same decision but more precisely. Indeed the S^3 optimization did more than simply decide to open or close the SQRT computation in R3. The 10% reduction realized in R3 as compared with R1 results from:

(1) not requiring parameters for S_1 since DISC(.X,.Y,.Z) is available in T_1 and 2*.X is available in T_2, and

(2) creating a strong similarity subroutine (S_1) for SQRT(DISC(.X,.Y,.Z))/(2*.X). Notice that this expression has no "logical identity" (as subroutine or macro) in the algorithm but S^3, analyzing only the form of the program, identified it as a computational unit.

Item (2) is the critical point. The results in this example and the examples which follow demonstrate that computationally coherent expressions (candidates for S^3 optimization) do not necessarily correspond to the logically coherent expressions identified by the programmer as a macro or subroutine. Most discussion on optimization strategies which consider opening or closing subroutines has centered on examing those expressions which the programmer has identified as logically coherent. Similarity operates independently of the programmer's selection.

GADD-SUB

The second example comes from the Bliss-10 compiler. The routine GADD-SUB (abbreviated: GAS) generates code for add and subtract operations. The source and output from KATE's compilation of GAS in ALLOPT mode is reproduced in appendix A.

This version of GAS differs from the original version in the Bliss-10 compiler in that several of the macro declarations here were routines in the original. In particular, LITV, REGAK, TVRP, and RLITP were routines in the original version. The results of compiling GAS with NOOPT, ALLBUTSIM, and ALLOPT modes and of compiling the original with ALLBUTSIM are summarized as follows:

GAS

NOOPT	979			
ALLBUTSIM	914	6.6%		
ALLOPT	697		28.7%	18.2%
ORIGINAL	855			

Again the difference in code size that results when S^3 decides which expressions to close is striking. The table that follows is keyed to pages in appendix A and serves as a guide to locating the S^3 optimizations in the output.

SSNAME	SEMANTICS	CALLS	COST	PAGE
S_1	GNEG(.Y)	2	3	122
S_2	LITV(π)	3	7	122
S_3	RLITP(π)	4	5	122
S_5	RLEX(.X)	4	2	123
S_6	GANL(π_1,NAMELEX(.x),π_2)	2	8	123
S_7	GLTR(.X)	5	3	124
S_{10}	GASCOMMUTE	10	12	124
S_{12}	(X←GLTR(.X);REGAK(.X))↑23	2	5	125
S_{15}	REGAK(.X)	3	12	125
S_{26}	LITV(SLEX(π)) neq 0	4	11	127
S_{30}	TVRP(π)	3	15	127
S_{32}	SIGN(.X)	5	2	128
S_{33}	GNEG(GAS(.ABSX,.ABSY,.ADDPOSSIBLE))	2	8	131
S_{35}	(if .ADDPOSSIBLE then ADD else SUB)↑27 or S_{12}	2	9	131

There are several observations to make about the results of S^3. In the original source for GAS the routine REGAK was a single parmeter subroutine. The S^3 optimization created a zero parameter subroutine S_{15} since all calls within GAS to REGAK passed the same parameter .X. S_6 is a case where S^3 recognized that two calls on GANL passed the same second parameter and so created a new two parameter subroutine. S_{12} and S_{35} are examples of formally identical expressions which were not assigned a logical name (via macro or routine declaration) in the original source.

It is interesting to observe that the subroutines of the original text were re-recognized as subprograms by S^3. One might ask why a good programmer would not have identified himself all the choices made by S^3. In the case of GASCOMMUTE, it would seem natural for the programmer to have made that identification. However, it is extremely unlikely that the same programmer would have identified S_6, S_{12}, S_{33}, and S_{35} as code sequences to be closed although closing them did reduce code size by slightly less than 4%. More importantly, this example demonstrates that he need not be forced to make the choice between open and closed subprogram. An S^3 optimization can be used to perform this analysis.

CPOLY

The final example is selected from the algorithms section of the Communications of the ACM[JT72]. CPOLY is a Fortran program to find all the zeros of a complex polynomial. Being a translation of an Algol procedure, it conformed easily to Bliss control syntax. In addition, the translation to Fortran had precluded recursive calls among the various subroutines. The source for CPOLY is reproduced in Appendix B.

CPOLY was transcribed into two Bliss versions with the body of SUBROUTINE CPOLY as the main body of the Bliss program. In one version the remaining subroutines were declared as macros. KATE compiled this program in NOOPT, ALLBUTSIM, and ALLOPT modes. KATE also compiled a second version in which the original subroutines remained as routines. The results are summarized below:

CPOLY

NOOPT	2557			
ALLBUTSIM	2460	3.8%		
ALLOPT	952		62.7%	14.3%
ORIGINAL	1106			

The results are quite similar to those obtained in the GAS example. The large variation (62.7% vs. 3.8%) between invoking the S^3 optimization and not invoking it results from the size of the subroutines involved and the frequency of the calls on them. The savings of the ALLOPT compilation over the ORIGINAL results primarily from two characteristics of the program:

(1) Several of the subroutines, viz. SCALE, CAUCHY, NOSHFT, and FXSHFT were called only once. S^3 simply compiled them in line.

(2) Many of the procedures are passed parameters which are identical at all call sites. S^3 reduced calling overhead by removing those parameters

S^3 OPTIMIZATION AND EXECUTION TIME

The preceding discussion on the effects of the S^3 optimization has concentrated on reductions in code size. Since S^3 reduced program size by introducing subroutines, it is natural to assume that program execution time has been increased. In this section we will report on some preliminary analysis which demonstrates that such an assumption is not valid. We chose CPOLY for our analysis over GADD-SUB since the latter program is simply a large decision tree and has no loop expressions.

Our main difficulty in analyzing the effect of S^3 on execution time is selecting a reasonable method for performing a static evaluation. For example, consider the problem of estimating the execution time of a branching control expression. There are three obvious alternatives: (1) select a particular branch, (2) average the execution times of the branches (assuming equi-probable selection of a branch), or (3) compute a weighted average of the branches (assigning a probability of selection to each branch). The added constraint that we intended to collect the data by hand compelled us to choose the first alternative and to limit our investigation to the inner loop of CPOLY which we identified as the loop in VRSHFT called from FXSHFT.

Two control paths through VRSHFT and the subroutine called by VRSHFT were selected. The first path was chosen by selecting those branches which entail the largest number of instructions. That is the longest (deepest) path through VRSHFT and its calls. The number of instructions executed in the orginal version was 3630*NN + 10360 and for the S^3-produced version 3810*NN + 6090. The parameter NN is the degree of the input polynomial plus one and is constrained to be ≤50. Thus in the worst case (NN=50) the

S^3 version requires 2.5% more execution time than the original. As NN decreases the performance of the S^3 version improves. If NN=10, then the S^3 version requires 5.5% <u>less</u> execution time than the original.

The second control path which we selected was shorter (i.e. fewer instructions per iteration): The original version executed 860*NN + 3030 instructions whereas the S^3-produced version executed 860*NN + 1840. The NN-terms in the equations are identical since no S^3-created (and not specified by the programmer) subroutines were executed in the NN-dependent loops. If NN=50, then S^3 <u>reduced</u> execution time by 2.5%; and if N=10, then S^3 <u>reduced</u> execution time by 10%.

The effect of S^3 optimization on the execution time of a program clearly requires more study than that given by this preliminary analysis. The purpose of presenting the results of this initial investigation is to dispel the assumption that S^3 optimization necessarily increases the execution time of a program. Indeed that had been our assumption before we studied the effects of S^3 on CPOLY more closely.

SUMMARY

Having produced a set of numbers measuring the effects of the program KATE on a few examples, it is important to place this information in the proper perspective. Chapter II introduced a collection of primitives used to describe the class of classical optimization techniques. The effectiveness of those optimizations is not an issue to this thesis. The success of the Fortran-H experiment which embodies those optimizations has already verified their utility. The merit of Chapter II lies in the concise statement of these optimization strategies and a correspondingly simple implementation of them.

The similarity notion, on the other hand, is a new concept. Chapter III described a number of optimizations in terms of similarity. We selected one of those, the S^3 optimization, and implemented it in KATE. S^3 was selected because it dealt with an area of object code optimization not touched by Chapter II -- the opening and closing of subprograms. Cocke and Schwartz discuss this area in some detail. However they concentrate on working with subprograms already identified by the programmer rather than on discovering the subprograms independent of the programmer. In addition, they only consider <u>opening</u> subprograms and reducing the amount of linkage code. The results that KATE produced are not to be interpreted as conclusive evidence that. S^3 optimization will produce a 10% to 15% reduction in program size across the board. The results <u>do</u> say that S^3, which is concisely and coherently describable in terms of the similarity notion, has potential for producing significant reductions in object code size.

Finally, if one examines any of the above examples, he can find places where KATE could have done better or where, if the original program were restructured, S^3 would not have produced the same favorable results. We do not propose a contest between programmer and compiler to discover some "minimal" program. We see the S^3 optimization in the following light. Let the programmer design the logical structure of his program and identify his computational sequences on the basis of their logical coherence. An S^3 optimization can decide for him between implementing those sequences as closed or open subprograms.

CHAPTER V

CONCLUSION

This final chapter is divided into two sections. The first section summarizes the results of our investigation. The second part suggests future directions in which this study can progress.

SUMMARY OF THESIS RESULTS

Chapter II motivated, defined, and used a collection of concepts for describing code motion, redundant expression elimination, and strength reduction optimizations. The concision of those decriptions demonstrates that the goal of discovering a set of primitives sufficiently powerful to enable concise descriptions of a class of optimizations has been achieved. Furthermore, the descriptions are independent of the intermediate representation of the program. Language independence has been accomplished by isolating language-dependent characteristics in the ordering relations $(\lhd, \prec, \ll, <)$. Finally, although the optimizations themselves may on the surface appear to be unrelated, the primitives provide a homogeneous description which, in turn, leads to a compact, cleanly structured implementation.

CONCLUSION

SUMMARY OF THESIS RESULTS

A new concept, similarity, was introduced in Chapter III. A colllection of new optimizations was defined in terms of the similarity notion. One of these new optimizations, S^3, was examined in greater detail. The discussion in Chapter III (and the analysis in Chapter IV) demonstrates that S^3 opens a significant new area of investigation into program optimizations. Previous research in optimization has done very little in the area of optimizations involving subprograms. No work, known to us, has investigated the possibility of using a compiler to determine the computational units to be implemented as closed subroutines.

FUTURE RESEARCH

In the process of doing this research a number of areas of possibile future investigation have emerged. Some of them are short-range and reasonably well-defined while others are long-range and less specific.

The program KATE implemented the primitives of Chapter II and the similarity function, SIGMA, defined in Chapter III. The evolution of the primitives and the construction of KATE proceeded in parallel during our investigation. Each process provided information for the development of the other. However the major emphasis lay in the development of the primitives. Now that the primitives have evolved to their present state, it would be worthwhile to reconstruct KATE and observe the effect on the resulting program. Since optimizing compilers are noted for being expensive in terms of both time and space, one might conentrate on examining alternate implementations of the primitives which reduce this overhead.

Chapter III developed a particular similarity function, SIGMA. That function was evolved with the S^3 optimization technique in mind. It is not clear that SIGMA is the appropriate similarity function for all the optimizations defined in Chapter III. An obvious area of investigation lies in discovering other useful similarity functions. Particularly, one might examine similarity functions which are sensitive to execution time overhead and the use of temporary storage. The set of optimizations described in terms of the similarity concept in Chapter III are new. In addition to developing new similarity functions, there is certainly the potential for discovering more optimizations defined in terms of similarity.

Another area of investigation is related to the notion of strength reduction. In Chapter II we began the section on strength reduction by posing the problem of discovering a relation F, such that $F(e,\Delta e) = e'$ and the cost of evaluating $F(e,\Delta e)$ is less than the cost of evaluating e'. The statement of this problem is motivated by the observation that strength reduction seems too specialized. The restriction to polynomials and looping environments is reasonably restrictive. The thesis described the feasibility of strength reduction optimizations in non-looping environments. The generalization to non-polynomial expressions, on the other hand, remains an open question. The problem consists of discovering a set of non-polynomial expression pairs (e,e') for which there exists a closed-form relation F satisfying the equation $F(e,\Delta e) = e'$,

Finally and, to our mind most importantly, a spectrum of questions opened by the S^3 optimization technique remains to be studied. S^3 was developed in the context of an investigation into object code optimization. Indeed one area of study is an investigation into modifications to the heuristics implemented in SIGMA and reconsideration of the overall structure of the S^3 module in KATE. There are, however, other directions to be pursued.

At the end of Chapter IV we presented a brief summary of a preliminary investigation into the effect of S^3 optimization on the execution time of a program. That investigation suggests two area for future study. First, there is the problem of performing a static analysis on the execution time of a program. Can one determine a meaningful data-independent measure of execution time? Can a program be analyzed to determine the kind of information that must be known about the input data in order to perfrom a valid analysis? Presumably a programmer makes some assumptions about the data input to a program in order to decide among alternative algorithms. Perhaps those assumptions can be incorporated into a static analysis of execution time. Second, the function SIGMA was designed to minimize object code size. How does one design a similarity function that is more sensitive to execution time? There are obvious parameters like loop depth and calling overhead. It seems clear, however, that heuristics encoded in a execution-time-sensitive similarity function require the same kind of information used in a static evaluation of execution time. Hence these two areas appear to be closely related.

In analyzing the form of a program, KATE discovers a set of computationally coherent expressions. Our initial investigation into this area, discussed in Chapter IV, demonstrated deviations from the selections made by the programmer. It is interesting to consider what one might learn about the structure of programs by analyzing the results of allowing an S^3 pass to select subprograms. Will S^3 consistently outperform the programmer in terms of reducing program size? Do the (potentially) different subprograms selected by S^3 provide significant feedback on the programmer's choice of logically coherent subprograms?

Some current research by S.L. Gerhart[GE72] involves the verification of APL programs. One aspect of this work is concerned with investigating the effect of the powerful APL operators on the verification process. For example, one observes that an algortihm represented by a nested-loop expression in Algol can perhaps be represented by a single operator in APL. As a result, a verification of the APL program should proceed with less difficulty than the verification of the corresponding Algol program since the effect of the involved Algol control expressions has been captured in a single operator. Intuitively this models a mathematician's approach to generating a large, involved proof. He typically identifies a set of sub-goals (lemmas -- macros -- subroutines). Having verified the sub-goals, he proceeds to combine these into a verification of the original theorem. It seems promising, then, to investigate the usefulness of similarity for discovering sub-goals and thereby reduce the complexity of the verification process.

These last two suggestions are not directly related to the area of object code optimization, but are natural outgrowths from observing the effect of S^3. They offer a wide range of interest for future study.

```
BEGIN

STRUCTUPE VECTOR[I]=(.VECTOR+.I);

MACRO
        LEFTF=0$,          RIGHTM=#177777$.          NEGF=1$,
        POSNSIZEF=2$,      STEF=3$,                  RTEF=4$,
        ADDI=#271$,        SUBI=#275$.               ARTEF=0$,
        VEF=0$.            LTEF=1$.                  STACKVAR=1↑10+1↑5$,
        DTF=5$.            ADDM=#273$,               SUBM=#276$.
        ADD=#270$,         SUB=#274$.                LSM=1↑7$.
        NEGM=1↑15$.        DOTM=1↑13$,               RTEM=#377$,
        LSSTEM=#377↑18$.   STEM=#377$,               RTESTEM=#177777$,
        ZERO=#4000$.       ZERO36=36$.               POSNSIZEM=#177777$;

MACRO
        SIGN(X)=((X) AND NEGM)$,
        LITP(X)=(((X) AND NOT STEM) EQL 0)$.
        NAMP(X)=(((X) AND NOT STEM) EQL (LSM OR ZERO36))$,
        GABS(X)=((X) AND NOT NEGM)$.
        ZERONAMP(X)=(((X) AND NOT STEM) EQL LSM)$,
        STACKVARP(X)=STACKVAR↑(-.ST[X])$,
        LITV(X)=(IF (X)[VEF] THEN .LT[(X)[LTEF]] ELSE (X)[LTEF])$,
        REGAK(X)=(IF NOT (.RT[(X)[RTEF]] AND (X)[DTF]) THEN .RT[(X)[RTEF]]
                  ELSE GMA(X))$.
        CODE(F,A,M)=(INST←(F)↑27 OR (A)↑23 OR (M))$,
        TVRP(X)=(IF REGP(X) THEN
                   ITRP(X) OR (.RT[(X)[RTEF]] EQL .OPTTOREGADDR)
                 ELSE 0)$;
GLOBAL OPTTOREGADDR,RT,INST,LT,ST;

EXTERNAL %ROUTINES%
        PCIVR,GNEG,LITLEXEME,GANL,MPTRTYP,GLTR,GMA,TVMP,DCRP,
        GLAR,GLTM,REGSEARCH,SHOULDEXCH,REGAR,MEMORYA,GMLTR,REGP,
        READY,ITRP;

ROUTINE GAS(X,Y,F)=
  !GENERATE CODE FOR X&Y WHERE & IS CASE F OF SET +;- TES.
  !     THIS IS UNDOUBTEDLY THE BEST (WORST?) CASE FOR SHOWING THE
  ! COMPLEXITY OF THE DELAYING MECHANISM.  IT WOULD BE FAIR TO SAY
  ! THAT THIS ROUTINE IS BIASED TOWARDS OPTIMIZING STRUCTURE ACCESSING.
  ! I.E. ADDITION BY INDEXING.  FOR EXAMPLE, WHEN PASSED THE OPERANDS
  ! FOR .A + 1, GAS LOADS .A INTO A REGISTER (SAY R) AND RETURNS A LEXEME
  ! OF THE FORM (.R+1) (I.E. RTEF=R AND LSSTEF=1).  THE IDEA HERE IS THAT
  ! IF THE EXPRESSION .A + 1 HAS APPEARED IN THE CONTEXT "(.A+1)<0,36>←EXP"
  ! THEN THE ADDITION WOULD BE ACCOMPLISHED BY INDEXING IN THE INSTRUCTION:
  ! "MOVEM EXP,1(R)."
  !     THE SET OF SPECIAL CASES IS COMMENTED ON THE RIGHT SIDE OF
  ! THE CODE.  E.G. !(@R+N)+L IS TO BE INTERPRETED TO MEAN:
  !             X= LEXEME REP. REG + NAME
  !             Y= LITERAL L
  !             F= +.
  !     FOLLOWING THE SET OF SPECIAL CASES THE ROUTINE ATTEMPTS  TO
  ! HANDLE THE EIGHT CASES THAT ARISE FROM F AND THE POSSIBILITY OF
  ! UNARY MINUS ON X OR Y OR BOTH.
  !     (1) X+Y           (2) X-Y
  !     (3) X+-Y          (4) X--Y
  !     (5) -X+Y          (6) -X--Y
  !     (7) -X+-Y         (8) -X-Y

  BEGIN
    MACRO GASCOMMUTE=(GAS(IF .F THEN GNEG(.Y) ELSE .Y,.X AND NOT NEGM,.X[NEGF]))$;
    MACRO RLITP(X)=((X AND NOT RTESTEM) EQL 0 AND (X AND RTEM) NEQ 0)$;
    MACRO RLEX(X)=(X AND RTEM)$;
    MACRO NAMELEX(X)=((X AND LSSTEM) OR ZERO36)$;
    MACRO SLEX(X)=(X AND (LSSTEM OR POSNSIZEM))$;
    MACRO RNAMP(X)=
              IF X[POSNSIZEF] EQL 0 THEN
              IF (X AND RTEM) NEQ 0 THEN
                NAMP((X AND NOT RTEM) OR ZERO36)$;
  LOCAL
      YVALUE, ! VALUE OF LITERAL Y
      ABSY,   ! GABS(.Y)
      ABSX,   ! GABS(.X)
      XREG,YREG,R,
```

```
   ADDPOSSIBLE:     ! .F EQL SIGN(.Y)

MACRO
   TEMPX=R[0]$,    ! X IS A TEMP REG
   TEMPY=R[1]$;  ! Y IS A TEMP REG

MAP VECTOR X,Y;

PCIVR(.X,.Y);
ABSY+GABS(.Y); ABSX+GABS(.X);
IF LITP(.Y) THEN
                                                          !X-L
  IF .F THEN GAS(.X,GNEG(.Y),0) ELSE
                                                          !X+0
  IF .Y EQL ZERO THEN .X ELSE
  IF LITP(.X) THEN
                                                          !L+L
    LITLEXEME(LITV(.X)+LITV(.Y)) ELSE
  IF RLITP(.ABSX) THEN
                                                          !(@R+L)+L
   GAS(SLEX(.X),.Y,.X[NEGF]) OR (.X AND (NEGM OR RTEM)) ELSE
  IF NAMP(.X) THEN
                                                          !N+L
    GANL(0,.X,.Y) ELSE
• IF RNAMP(.X) THEN
                                                          !(@R+N)+L
    GANL(PLEX(.X),NAMELEX(.X),.Y) ELSE
                                                          !X+L
  IF (IF ZERONAMP(.X) THEN
       BEGIN
         YVALUE+LITV(.Y);
         (.YVALUE AND RIGHTM) EQL 0
           AND NOT STACKVARP(.X[STEF])
       END
     ELSE 0) THEN
                                                          !X<0,0>+L
   MPTRTYP(.YVALUE[LEFTF],.X) ELSE
  GLTR(.X) OR .Y
  ELSE
IF LITP(.X) THEN
                                                          !L+Y
   GASCOMMUTE ELSE
IF ZERONAMP(.Y) THEN
                                                          !X&Y<0,0>
   (CODE(CASE .F OF SET ADDI;SUBI TES, !X+GLTR(.X);REGAK(.X)), GMA(.Y OR DOTM));
   .X) ELSE
IF ZERONAMP(.X) THEN GASCOMMUTE ELSE
BEGIN
                                                          !X<0,0>&Y
ADDPOSSIBLE+.F EQL SIGN(.Y);
IF NAMP(.ABSY) AND .ADDPOSSIBLE THEN
  IF REGP(.X) THEN
                                                          !@R+N
    .X OR (.ABSY AND LSSTEM) ELSE
  IF RLITP(.X) THEN
                                                          !(@R+L)+N
    GANL(RLEX(.X),.ABSY,SLEX(.X)) ELSE
                                                          !X+N
  GLTR(.X) OR (.ABSY AND LSSTEM)
  ELSE
IF NAMP(.ABSX) THEN
                                                          !N&Y
   GASCOMMUTE ELSE
IF RNAMP(.ABSX) THEN
  IF  (IF PLITP(.ABSY) THEN LITV(SLEX(.Y)) NEQ 0) AND .ADDPOSSIBLE THEN
                                                          !(@R+N)+(@R'+L)
     BEGIN
     IF TVRP(PLEX(.X)) THEN
       (XREG+PLEX(.X);YREG+PLEX(.Y))
     ELSE (XREG+RLEX(.Y);YREG+PLEX(.X));
     GAS(GANL(.XREG,NAMELEX(.X),SLEX(.Y)),.YREG,0) OR (.X AND NEGM)
     END
    ELSE
                                                          !(@R+N)&Y
  GAS(GAS(PLEX(.X),.Y,.F),(.X AND NOT RTEM) OR ZERO36,0)
```

```
    ELSE
  IF PNAMP(.ABSY) THEN
                                                                          !X&(@R+N)
    GASCOMMUTE ELSE
  IF (IF RLITP(.ABSX) THEN LITV(SLEX(.ABSX)) NEQ 0) THEN
                                                                          !(@R+L)&Y
    BEGIN MACRO X1=ABSX$:
     X1←GAS(.X AND NOT LSSTEM,.Y,.F);
     IF .X[NEGF] AND .X1[NEGF] THEN
       GNEG(GAS(SLEX(.X),GABS(.X1),0)) ELSE
     GAS(IF .X[NEGF] THEN GNEG(SLEX(.X)) ELSE SLEX(.X),GABS(.X1),.X1[NEGF])
    END ELSE
  IF (IF RLITP(.ABSY) THEN LITV(SLEX(.ABSY)) NEQ 0) THEN
                                                                          !X&(@R+L)
    GASCOMMUTE ELSE
  IF TVMP(.Y) AND DCRP(.X) THEN
                                                                          !D&M
    (CODE(IF .ADDPOSSIBLE THEN ADDM ELSE SUBM,
          (X←GLAR(.X);REGAK(.X)),GMA(Y←GLTM(.ABSY))); .Y) ELSE
  IF TVMP(.X) THEN
                                                                          !M&Y
     GASCOMMUTE ELSE
  BEGIN
  REGSEARCH(X,Y);
  ABSX←GABS(.X); ABSY←GABS(.Y);
  IF (TEMPX←TVRP(.ABSX)) AND (TEMPY←TVRP(.ABSY)) THEN
      IF SHOULDEXCH(.X,.Y) THEN
       GASCOMMUTE ELSE
      IF SIGN(.X) THEN
        IF .ADDPOSSIBLE AND .RT[.X[RTEF]][ARTEF] NEQ .VREG THEN
                                                                          !5,6
          GASCOMMUTE ELSE
                                                                          !7,8
        GNEG(GAS(.ABSX,.ABSY,.ADDPOSSIBLE))
       ELSE
      (CODE(IF .ADDPOSSIBLE THEN ADD ELSE SUB,
            (X←GLTR(.X);REGAK(.X)),REGAR(GLTR(.ABSY)));
                                                                          !1-4
     .X)
    ELSE
  IF .TEMPX THEN
    IF SIGN(.X) THEN
                                                                          !5-8
      GNEG(GAS(.ABSX,.ABSY,.ADDPOSSIBLE)) ELSE
    (CODE(IF .ADDPOSSIBLE THEN ADD ELSE SUB,
          (X←GLTR(.X);REGAK(.X)),
                                                                          !1-4
          MEMORYA(.Y));
    .X)
     ELSE
  IF .TEMPY THEN
     GASCOMMUTE ELSE
  IF SIGN(.X) THEN
    IF .ADDPOSSIBLE THEN
                                                                          !5-6
       GASCOMMUTE ELSE
                                                                          !7-8
    BEGIN
    X←GOLTR(.X);
    IF SIGN(.X) THEN
      GNEG(GAS(GABS(.X),.ABSY,0)) ELSE
    GAS(.X,.ABSY,1)
    END ELSE
                                                                          !1-4
  IF READY(.X) THEN
    IF .ADDPOSSIBLE THEN GAS(GLTR(.ABSY),.X,0) ELSE
    IF READY(.ABSY) THEN GAS(GLTR(.X),.ABSY,1) ELSE
    GNEG(GAS(GLTR(.ABSY),.X,1))
    ELSE
  GAS(GLTR(.X),.ABSY,.F OR SIGN(.Y))
  END
  END

   END;
END ELUDOM
```

```
GAS:
     PARM    .X                           1  *
     PARM    .Y                           1  *
     CALL    PCIVR..T$0                   1  *
     DPARM   2                            1  *
     AND     .Y,-100001..T$1              2  *
     LD      ABSY..T$1..T$1               1  *
     AND     .X,-100001..T$1              2  *
     LD      ABSX,.T$1..T$1               1  *
     AND     .Y,-400..T$3                 2  *
     EQL     .T$3.0..T$4                  0  *
     BRF     .T$4.L$1                     1  *
     BRF     .F.L$3                       3    *
     PARM    .X                           1      *
     SSCAL   S$1,E$1                      1      *
S$1:
     PARM    .Y                           1      *
     CALL    GNEG..T$0                    1      *
     DPARM   1                            1      *
     RTRN                                 1      *
E$1:
     PARM    .T$0                         1      *
     PARM    0                            1      *
     CALL    GAS..T$0                     1      *
     DPARM   3                            1      *
     LD      T$4,.T$0,.T$4                1      *
     BR      L$4                          1      *
                                                 12
L$3:
     EQL     .Y,4000..T$6                 2      *
     BRF     .T$6.L$5                     1      *
     LD      T$5..X..T$5                  1        *
     BR      L$6                          1        *
                                                   2
L$5:
     AND     .X,-400..T$7                 2        *
     EQL     .T$7.0..T$10                 0        *
     BRF     .T$10.L$7                    1        *
     LD      T$10..X..T$10                1          *
     SSCAL   S$2,E$2                      1          *
S$2:
     ADD     .T$10.1..T$12                2          *
     ADD     .T$10.0..T$13                0          *
     BRF     .T$13.L$11                   1          *
     ADD     LT..T$12..T$13               1            *
     LD      T$11...T$13..T$11            1            *
     BR      L$12                         1            *
                                                       3
L$11:
     LD      T$11..T$12..T$11             1            *
                                                       1
L$12:
     RTRN                                 1          *
E$2:
     LD      T$10..Y..T$10                1          *
     SSCAL   S$2..+1..T$11                1          *
     ADD     .T$11,.T$11..T$12            2          *
     PARM    .T$12                        1          *
     CALL    LITLEXEME..T$0               1          *
     DPARM   1                            1          *
     LD      T$6..T$0..T$6                1          *
     BR      L$10                         1          *
                                                     19
L$7:
     LD      T$13..ABSX..T$13             1          *
     SSCAL   S$3.E$3                      1          *
S$3:
     AND     .T$13,-200000,.T$14          2          *
     EQL     .T$14.0..T$14                0          *
     AND     .T$13.377..T$15              2          *
     NEQ     .T$15.0..T$15                0          *
     AND     .T$14..T$15..T$14            1          *
```

```
    RTRN                              1         *
E$3:
    BPF     .T$14.L$13                2         *
    AND     .X.177777..T$15           2           *
    PARM    .T$15                     1           *
    PARM    .Y                        1           *
    ADD     X.1..T$15                 0           *
    PARM    ..T$15                    1           *
    CALL    GAS..T$0                  1           *
    DPARM   3                         1           *
    AND     .X.100377..T$15           2           *
    OR      .T$0..T$15..T$0           1           *
    LD      T$12..T$0..T$12           1           *
    BR      L$14                      1           *
                                                  12
L$13:
    EQL     .T$7.244..T$16            1           *
    BRF     .T$16.L$15                1           *
    PARM    0                         1             *
    PARM    .X                        1             *
    PARM    .Y                        1             *
    CALL    GANL..T$0                 1             *
    DPARM   3                         1             *
    LD      T$15..T$0..T$15           1             *
    BR      L$16                      1             *
                                                    7
L$15:
    ADD     X.2..T$20                 0             *
    EQL     ..T$20.0..T$20            1             *
    BRF     .T$20.L$21                1             *
    SSCAL   S$5.E$5                   1               *
S$5:
    AND     .X.377..T$21              2               *
    RTRN                              1               *
E$5:
    NEQ     .T$21.0..T$22             0               *
    BRF     .T$22.L$23                1               *
    OR      .T$7.44..T$22             2                 *
    AND     .T$22.-400..T$22          1                 *
    EQL     .T$22.244..T$22           1                 *
    LD      T$20..T$22..T$20          1                 *
    BR      L$24                      1                 *
                                                        6
L$23:
    LD      T$20.0..T$20              1                 *
                                                        1
L$24:
    LD      T$17..T$20..T$17          1               *
    BR      L$22                      1               *
                                                      14
L$21:
    LD      T$17.0..T$17              1               *
                                                      1
L$22:
    BRF     .T$17.L$17                1             *
    SSCAL   S$5..+1..T$21             1               *
    LD      T$22..T$21..T$22          1               *
    LD      T$23..Y..T$23             1               *
    SSCAL   S$6.E$6                   1               *
S$6:
    PARM    .T$22                     1               *
    AND     .X.177400..T$24           2               *
    OR      .T$24.44..T$24            1               *
    PARM    .T$24                     1               *
    PARM    .T$23                     1               *
    CALL    GANL..T$0                 1               *
    DPARM   3                         1               *
    RTRN                              1               *
E$6:
    LD      T$15..T$0..T$16           1               *
    BR      L$20                      1               *
                                                      15
L$17:
    EQL     .T$7.200..T$7             1               *
    BRF     .T$7.L$27                 1               *
```

```
    LD       T$10..Y..T$10               1               *
    SSCAL    S$2..+1..T$11               1               *
    LD       YVALUE..T$11..T$11          1               *
    AND      .YVALUE.177777..T$11        2               *
    EQL      .T$11.0..T$11               0               *
    ADD      X.3..T$7                    0               *
    ADD      ST...T$7..T$7               1               *
    RTSH     40...T$7..T$7               3               *
    ADD      2000..T$7..T$7              1               *
    ANDCR    .T$11..T$7..T$11            1               *
    LD       T$26..T$11..T$26            1               *
    BR       L$30                        1               *
                                                         13
L$27:
    LD       T$26.0..T$26                1               *
                                                         1
L$30:
    BRF      .T$26.L$25                  1              *
    ADD      YVALUE.0..T$26              0               *
    PARM     ..T$26                      1               *
    PARM     .X                          1               *
    CALL     MPTRTYP..T$0                1               *
    DPARM    2                           1               *
    LD       T$25..T$0..T$25             1               *
    BR       L$26                        1               *
                                                         6
L$25:
    SSCAL    S$7.E$7                     1               *
S$7:
    PARM     .X                          1               *
    CALL     GLTR..T$0                   1               *
    DPARM    1                           1               *
    RTRN                                 1               *
E$7:
    OR       .T$0..Y..T$0                1               *
    LD       T$25..T$0..T$25             1               *
                                                         7
L$26:
    LD       T$16..T$25..T$16            1              *
                                                        31
L$20:
    LD       T$15..T$16..T$15            1             *
                                                       65
L$16:
    LD       T$12..T$15..T$12            1            *
                                                      75
L$14:
    LD       T$6..T$12..T$6              1           *
                                                     98
L$10:
    LD       T$5..T$6..T$5               1          *
                                                    121
L$6:
    LD       T$4..T$5..T$4               1         *
                                                   127
L$4:
    LD       T$2..T$4..T$2               1        *
    BR       L$2                         1        *
                                                  144
L$1:
    AND      .X.-400..T$5                2        *
    EQL      .T$5.0..T$6                 0        *
    BRF      .T$6.L$31                   1        *
    SSCAL    S$10.E$10                   1          *
S$10:
    BRF      .F.L$33                     3          *
    SSCAL    S$1..+1..T$0                1            *
    LD       T$6..T$0..T$6               1            *
    BR       L$34                        1            *
                                                      3
L$33:
    LD       T$6..Y..T$6                 1            *
                                                      1
L$34:
    PARM     .T$6                        1          *
```

```
    PARM    .T$1                    1    *
    ADD     X,1..T$6                0    *
    PARM    ..T$6                   1    *
    CALL    GAS..T$0                1    *
    DPARM   3                       1    *
    RTRN                            1    *
E$10:
    LD      T$4..T$0,.T$4           1    *
    BR      L$32                    1    *
                                         16
L$31:
    EQL     .T$3.200,.T$3           1    *
    BRF     .T$3.L$35               1    *
    XCT     .F,L$37                 2      *
L$37:
    BR      L$40                    1      *
    BR      L$41                    1      *
L$40:
    LD      T$3,271,.T$3            1        *
    BR      L$42                    1        *
                                             2
L$41:
    LD      T$3.275,.T$3            1        *
                                             1
L$42:
    LTSH    .T$3.33..T$3            1      *
    SSCAL   S$12.E$12               1      *
S$12:
    SSCAL   S$7,.+1,.T$0            1      *
    LD      X,.T$0..T$0             1      *
    SSCAL   S$15.E$15               1      *
S$15:
    ADD     .X,4,.T$15              1      *
    ADD     RT,.T$15,.T$15          0      *
    ADD     .X.5..T$16              2      *
    AND     ..T$15,.T$16,.T$16      1      *
    BRT     .T$16,L$43              1      *
    LD      T$12,..T$15,.T$12       1        *
    BR      L$44                    1        *
                                             2
L$43:
    PARM    .X                      1        *
    CALL    GMA,.T$0                1        *
    DPARM   1                       1        *
    LD      T$12,.T$0,.T$12         1        *
                                             4
L$44:
    RTRN                            1      *
E$15:
    LTSH    .T$12,27,.T$16          2      *
    RTRN                            1      *
E$12:
    OR      .T$3,.T$16,.T$3         1      *
    OR      .Y.20000,.T$25          2      *
    PARM    .T$25                   1      *
    CALL    GMA,.T$0                1      *
    DPARM   1                       1      *
    OR      .T$3,.T$0,.T$3          1      *
    LD      INST,.T$3,.T$3          1      *
    LD      T$6..X,.T$6             1      *
    BR      L$36                    1      *
                                           37
L$35:
    EQL     .T$5,200,.T$25          1      *
    BRF     .T$25.L$45              1      *
    SSCAL   S$10,.+1,.T$0           1        *
    LD      T$3..T$0,.T$3           1        *
    BR      L$46                    1        *
                                             3
L$45:
    AND     .Y.100000,.T$25         2        *
    EQL     .F..T$25,.T$25          1        *
    LD      ANDPOSSIBL,.T$25,.T$25  1        *
    AND     .ABSY,-400,.T$26        2        *
    EQL     .T$26,244,.T$11         1        *
```

```
        AND     .T$11..ADDPOSSIBL..T$11     1       *
        BRF     .T$11.L$47                  1       *
        PARM    .X                          1         *
        CALL    REGP..T$0                   1         *
        DPARM   1                           1         *
        BRF     .T$0.L$51                   2         *
        AND     .ABSY.177400..T$7           2           *
        OR      .X..T$7..T$7                1           *
        LD      T$11..T$7..T$11             1           *
        BR      L$52                        1           *
                                                        5
L$51:
        AND     .X.-200000..T$27            2           *
        EQL     .T$27.0..T$27               0           *
        SSCAL   S$5..+1..T$21               1           *
        NEQ     .T$21.0..T$30               0           *
        AND     .T$27..T$30..T$27           1           *
        BRF     .T$27.L$53                  1           *
        PARM    .T$21                       1             *
        PARM    .ABSY                       1             *
        AND     .X.177777..T$27             2             *
        PARM    .T$27                       1             *
        CALL    GANL..T$0                   1             *
        DPARM   3                           1             *
        LD      T$7..T$0..T$7               1             *
        BR      L$54                        1             *
                                                          9
L$53:
        SSCAL   S$7..+1..T$0                1             *
        AND     .ABSY.177400..T$27          2             *
        OR      .T$0..T$27..T$0             1             *
        LD      T$7..T$0..T$7               1             *
                                                          5
L$54:
        LD      T$11..T$7..T$11             1           *
                                                        20
L$52:
        LD      T$25..T$11..T$25            1         *
        BR      L$50                        1         *
                                                      32
L$47:
        AND     .ABSX.-400..T$7             2         *
        EQL     .T$7.244..T$27              1         *
        BRF     .T$27.L$55                  1         *
        SSCAL   S$10..+1..T$0               1           *
        LD      T$11..T$0..T$11             1           *
        BR      L$56                        1           *
                                                        3
L$55:
        ADD     ABSX.2..T$31                0           *
        EQL     ..T$31.0..T$31              1           *
        BRF     .T$31.L$61                  1           *
        AND     .T$13.377..T$32             2             *
        NEQ     .T$32.0..T$32               0             *
        BRF     .T$32.L$63                  2             *
        OR      .T$7.44..T$7                1               *
        AND     .T$7.-400..T$7              1               *
        EQL     .T$7.244..T$7               1               *
        LD      T$31..T$7..T$31             1               *
        BR      L$64                        1               *
                                                            5
L$63:
        LD      T$31.0..T$31                1               *
                                                            1
L$64:
        LD      T$30..T$31..T$30            1             *
        BR      L$62                        1             *
                                                          12
L$61:
        LD      T$30.0..T$30                1             *
                                                          1
L$62:
        BRF     .T$30.L$57                  1           *
        SSCAL   S$5..+1..T$21               1             *
        LD      T$13..ABSY..T$13            1             *
```

```
    SSCAL   S$3..+1..T$14                1            *
    BRF     .T$14.L$67                   2            *
    LD      T$7..Y..T$7                  1              *
    SSCAL   S$26.E$26                    1              *
S$26:
    AND     .T$7.177777..T$34            2              *
    ADD     .T$34.1..T$35                2              *
    AND     .T$7.177777..T$34            2              *
    ADD     .T$34.0..T$36                0              *
    BRF     .T$36.L$71                   1              *
    ADD     LT..T$25..T$36               1                *
    LD      T$33,..T$36..T$33            1                *
    BR      L$72                         1                *
                                                          3

L$71:
    LD      T$33..T$35..T$33             1                *
                                                          1

L$72:
    NEQ     .T$33.0..T$35                0              *
    RTRN                                 1              *
E$26:
    LD      T$31..T$35..T$31             1              *
    BR      L$70                         1              *
                                                        16

L$67:
    LD      T$31.0..T$31                 1              *
                                                        1

L$70:
    AND     .T$31..ADDPOSSIBL..T$31      1            *
    BRF     .T$31.L$65                   1            *
    AND     .Y.377..T$31                 2              *
    LD      T$37..T$21..T$37             1              *
    SSCAL   S$30.E$30                    1              *
S$30:
    PARM    .T$37                        1              *
    CALL    REGP..T$0                    1              *
    DPARM   1                            1              *
    BRF     .T$0.L$75                    2              *
    PARM    .T$37                        1                *
    CALL    ??P..T$0                     1                *
    DPARM   1                            1                *
    ADD     .T$37.4..T$41                1                *
    ADD     RT..T$41..T$41               0                *
    EQL     ..T$41..OPTTOREGAD..T$41     2                *
    OR      .T$0..T$41..T$0              1                *
    LD      T$40..T$0..T$40              1                *
    BR      L$76                         1                *
                                                          9

L$75:
    LD      T$40.0..T$40                 1                *
                                                          1

L$76:
    RTRN                                 1              *
E$30:
    BRF     .T$40.L$73                   2              *
    LD      XREG..T$21..T$21             1                *
    LD      YREG..T$31..T$31             1                *
    BR      L$74                         1                *
                                                          3

L$73:
    LD      XREG..T$31..T$31             1                *
    LD      YREG..T$21..T$21             1                *
                                                          2

L$74:
    LD      T$22..XREG..T$22             1              *
    AND     .Y.177777..T$31              2              *
    LD      T$23..T$31..T$23             1              *
    SSCAL   S$6..+1..T$0                 1              *
    PARM    .T$0                         1              *
    PARM    .YREG                        1              *
    PARM    0                            1              *
    CALL    GAS..T$0                     1              *
    DPARM   3                            1              *
    SSCAL   S$32.E$32                    1              *
S$32:
```

```
     AND      .X,100000,.T$31          2          *
     RTRN                              1          *
E$32:
     OR       .T$0,.T$31,.T$0          1          *
     LD       T$30,.T$0,.T$30          1          *
     BR       L$66                     1          *
                                                  44
L$65:
     PARM     .T$21                    1          *
     PARM     .Y                       1          *
     PARM     .F                       1          *
     CALL     GAS,.T$0                 1          *
     DPARM    3                        1          *
     PARM     .T$0                     1          *
     OR       .T$5,44,.T$5             1          *
     PARM     .T$5                     1          *
     PARM     0                        1          *
     CALL     GAS,.T$0                 1          *
     DPARM    3                        1          *
     LD       T$30,.T$0,.T$30          1          *
                                                  12
L$66:
     LD       T$27,.T$30,.T$27         1         *
     BR       L$60                     1         *
                                                 84
L$57:
     ADD      ABSY,2,.T$41             0         *
     EQL      ..T$41,0,.T$41           1         *
     BRF      .T$41,L$101              1         *
     AND      .ABSY,377,.T$42          2          *
     NEQ      .T$42,0,.T$42            0          *
     BRF      .T$42,L$103              2          *
     OR       .T$26,44,.T$26           1           *
     AND      .T$26,-400,.T$26         1           *
     EQL      .T$26,244,.T$26          1           *
     LD       T$41,.T$26,.T$41         1           *
     BR       L$104                    1           *
                                                   5
L$103:
     LD       T$41,0,.T$41             1           *
                                                   1
L$104:
     LD       T$5,.T$41,.T$5           1          *
     BR       L$102                    1          *
                                                  12
L$101:
     LD       T$5,0,.T$5               1          *
                                                  1
L$102:
     BRF      .T$5,L$77                1         *
     SSCAL    S$10,.+1,.T$0            1          *
     LD       T$30,.T$0,.T$30          1          *
     BR       L$100                    1          *
                                                  3
L$77:
     LD       T$13,.ABSX,.T$13         1          *
     SSCAL    S$3,.+1,.T$14            1          *
     BRF      .T$14,L$107              2          *
     LD       T$7,.ABSX,.T$7           1           *
     SSCAL    S$26,.+1,.T$35           1           *
     LD       T$41,.T$35,.T$41         1           *
     BR       L$116                    1           *
                                                   4
L$107:
     LD       T$41,0,.T$41             1           *
                                                   1
L$110:
     BRF      .T$41,L$105              1          *
     AND      .X,-177401,.T$41         2           *
     PARM     .T$41                    1           *
     PARM     .Y                       1           *
     PARM     .F                       1           *
     CALL     GAS,.T$0                 1           *
     DPARM    3                        1           *
     LD       ABSX,.T$0,.T$0           1           *
```

```
        AND      .X,177777,.T$35            2           *
        AND      .ABSX,-100001..T$26        2           *
        ADD      X,1,.T$42                  0           *
        ADD      ABSX,1,.T$43               0           *
        AND      ..T$42...T$43..T$42        2           *
        BRF      .T$42,L$111                1           *
        PARM     .T$35                      1             *
        PARM     .T$26                      1             *
        PARM     0                          1             *
        CALL     GAS,.T$0                   1             *
        DPARM    3                          1             *
        PARM     .T$0                       1             *
        CALL     GNEG,.T$0                  1             *
        DPARM    1                          1             *
        LD       T$41,.T$0,.T$41            1             *
        BR       L$112                      1             *
                                                          10
L$111:
        BRF      ..T$42,L$113               3             *
        PARM     .T$35                      1               *
        CALL     GNEG,.T$0                  1               *
        DPARM    1                          1               *
        LD       T$42,.T$0,.T$42            1               *
        BR       L$114                      1               *
                                                            5
L$113:
        LD       T$42,.T$35,.T$42           1               *
                                                            1
L$114:
        PARM     .T$42                      1             *
        PARM     .T$26                      1             *
        PARM     ..T$43                     1             *
        CALL     GAS,.T$0                   1             *
        DPARM    3                          1             *
        LD       T$41,.T$0,.T$41            1             *
                                                          15
L$112:
        LD       T$5,.T$41,.T$5             1           *
        BR       L$106                      1           *
                                                        42
L$105:
        LD       T$13,.ABSY,.T$13           1           *
        SSCAL    S$3,.+1,.T$14              1           *
        BRF      .T$14,L$117                1           *
        SSCAL    S$26,.+1,.T$35             1             *
        LD       T$26,.T$35,.T$26           1             *
        BR       L$120                      1             *
                                                          3
L$117:
        LD       T$26,0,.T$26               1             *
                                                          1
L$120:
        BRF      .T$26,L$115                1           *
        SSCAL    S$10,.+1,.T$0              1             *
        LD       T$41,.T$0,.T$41            1             *
        BR       L$116                      1             *
                                                          3
L$115:
        PARM     .Y                         1             *
        CALL     TVMP,.T$0                  1             *
        DPARM    1                          1             *
        PARM     .X                         1             *
        CALL     DCRP,.T$0                  1             *
        DPARM    1                          1             *
        AND      .T$0,.T$0,.T$0             1             *
        BRF      .T$0,L$121                 2             *
        BRF      .ADDPOSSIBL,L$123          3               *
        LD       T$33,273,.T$33             1                 *
        BR       L$124                      1                 *
                                                              2
L$123:
        LD       T$33,276,.T$33             1                 *
                                                              1
L$124:
        LTSH     .T$33,33,.T$33             1               *
```

```
        PARM    .X                          1           *
        CALL    GLAR,.T$0                   1           *
        DPARM   1                           1           *
        LD      X,.T$0,.T$0                 1           *
        SSCAL   S$15,.+1,.T$12              1           *
        LTSH    .T$12,27,.T$12              1           *
        OR      .T$33,.T$12,.T$33           1           *
        PARM    .ABSY                       1           *
        CALL    GLTM,.T$0                   1           *
        DPARM   1                           1           *
        LD      Y,.T$0,.T$0                 1           *
        PARM    .T$0                        1           *
        CALL    GMA,.T$0                    1           *
        DPARM   1                           1           *
        OR      .T$33,.T$0,.T$33            1           *
        LD      INST,.T$33,.T$33            1           *
        LD      T$26,.Y,.T$26               1           *
        BR      L$122                       1           *
                                                        25
L$121:
        PARM    .X                          1           *
        CALL    TVMP,.T$0                   1           *
        DPARM   1                           1           *
        BRF     .T$0,L$125                  2           *
        SSCAL   S$10,.+1,.T$0               1             *
        LD      T$33,.T$0,.T$33             1             *
        BR      L$126                       1             *
                                                          3
L$125:
        PARM    X                           1             *
        PARM    Y                           1             *
        CALL    REGSEARCH,.T$0              1             *
        DPARM   2                           1             *
        AND     .X,-100001,.T$12            2             *
        LD      ABSX,.T$12,.T$12            1             *
        AND     .Y,-100001,.T$14            2             *
        LD      ABSY,.T$14,.T$14            1             *
        ADD     R,0,.T$35                   0             *
        LD      T$37,.ABSX,.T$37            1             *
        SSCAL   S$30,.+1,.T$40              1             *
        LD      .T$35,.T$40,.T$40           1             *
        ADD     R,1,.T$42                   0             *
        LD      T$37,.ABSY,.T$37            1             *
        SSCAL   S$30,.+1,.T$40              1             *
        LD      .T$42,.T$40,.T$40           1             *
        AND     .T$40,.T$40,.T$40           1             *
        BRF     .T$40,L$127                 1             *
        PARM    .X                          1               *
        PARM    .Y                          1               *
        CALL    SHOULDEXCH,.T$0             1               *
        DPARM   2                           1               *
        BRF     .T$0,L$131                  2               *
        SSCAL   S$10,.+1,.T$0               1                 *
        LD      T$40,.T$0,.T$40             1                 *
        BR      L$132                       1                 *
                                                              3
L$131:
        SSCAL   S$32,.+1,.T$31              1                 *
        BRF     .T$31,L$133                 2                 *
        ADD     X,4,.T$45                   0                   *
        ADD     RT,..T$45,.T$45             1                   *
        ADD     .T$45,0,.T$46               0                   *
        NEQ     ..T$46,.VREG,.T$46          1                   *
        AND     .ADDPOSSIBL,.T$46,.T$46     1                   *
        BRF     .T$46,L$135                 1                   *
        SSCAL   S$10,.+1,.T$0               1                     *
        LD      T$44,.T$0,.T$44             1                     *
        BR      L$136                       1                     *
                                                                  3
L$135:
        SSCAL   S$33,E$33                   1                     *
S$33:
        PARM    .ABSX                       1                     *
        PARM    .ABSY                       1                     *
        PARM    .ADDPOSSIBL                 1                     *
```

```
        CALL    GAS..T$0                 1                          *
        DPARM   3                        1                          *
        PARM    .T$0                     1                          *
        CALL    GNEG..T$0                1                          *
        DPARM   1                        1                          *
        RTRN                             1                          *
   E$33:                                                            *
        LD      T$44,.T$0,.T$44          1                          *
                                                                    11
   L$136:
        LD      T$43,.T$44,.T$43         1                        *
        BR      L$134                    1                        *
                                                                  20
   L$133:
        SSCAL   S$35,E$35                1                        *
   S$35:
        BRF     .ADDPOSSIBL,L$137        3                        *
        LD      T$44,270,.T$44           1                          *
        BR      L$140                    1                          *
                                                                    2
   L$137:
        LD      T$44,274,.T$44           1                          *
                                                                    1
   L$140:
        LTSH    .T$44,33,.T$44           1                        *
        SSCAL   S$12..+1,.T$16           1                        *
        OR      .T$44,.T$16,.T$44        1                        *
        RTRN                             1                        *
   E$35:
        PARM    .ABSY                    1                        *
        CALL    GLTR..T$0                1                        *
        DPARM   1                        1                        *
        PARM    .T$0                     1                        *
        CALL    REGAR..T$0               1                        *
        DPARM   1                        1                        *
        OR      .T$44..T$0..T$0          1                        *
        LD      INST..T$0..T$0           1                        *
        LD      T$43..X,.T$43            1                        *
                                                                  20
   L$134:
        LD      T$40,.T$43,.T$40         1                      *
                                                                44
   L$132:
        LD      T$14,.T$40,.T$14         1                    *
        BR      L$130                    1                    *
                                                              55
   L$127:
        BRF     ..T$35,L$141             3                    *
        SSCAL   S$32..+1,.T$31           1                      *
        BRF     .T$31,L$143              2                      *
        SSCAL   S$33..+1,.T$0            1                        *
        LD      T$35..T$0,.T$35          1                        *
        BR      L$144                    1                        *
                                                                  3
   L$143:
        SSCAL   S$35..+1,.T$44           1                        *
        PARM    .Y                       1                        *
        CALL    MEMOPYA..T$0             1                        *
        DPARM   1                        1                        *
        OR      .T$44..T$0..T$44         1                        *
        LD      INST,.T$44..T$44         1                        *
        LD      T$35..X,.T$35            1                        *
                                                                  7
   L$144:
        LD      T$40,.T$35,.T$40         1                      *
        BR      L$142                    1                      *
                                                                15
   L$141:
        BRF     ..T$42,L$145             3                      *
        SSCAL   S$10..+1,.T$0            1                        *
        LD      T$35..T$0,.T$35          1                        *
        BR      L$146                    1                        *
                                                                  3
   L$145:
        SSCAL   S$32..+1,.T$31           1                        *
```

```
     BRF     .T$31.L$147              2
     BRF     .ADDPOSSIBL.L$151        3
     SSCAL   S$10,.+1..T$0            1
     LD      T$44,.T$0..T$44          1
     BR      L$152                    1
                                                  3
L$151:
     PARM    .X                       1
     CALL    GOLTR..T$0               1
     DPARM   1                        1
     LD      X,.T$0..T$0              1
     SSCAL   S$32..+1..T$31           1
     BRF     .T$31.L$153              1
     AND     .X,-100001..T$31         2
     PARM    .T$31                    1
     PARM    .ABSY                    1
     PARM    0                        1
     CALL    GAS..T$0                 1
     DPARM   3                        1
     PARM    .T$0                     1
     CALL    GNEG..T$0                1
     DPARM   1                        1
     LD      T$43,.T$0,.T$43          1
     BR      L$154                    1
                                                  12
L$153:
     PARM    .X                       1
     PARM    .ABSY                    1
     PARM    1                        1
     CALL    GAS..T$0                 1
     DPARM   3                        1
     LD      T$43,.T$0..T$43          1
                                                  6
L$154:
     LD      T$44,.T$43,.T$44         1
                                                  25
L$152:
     LD      T$42..T$44..T$42         1
     BR      L$150                    1
                                                  33
L$147:
     PARM    .X                       1
     CALL    READY..T$0               1
     DPARM   1                        1
     BRF     .T$0.L$155               2
     BRF     .ADDPOSSIBL.L$157        3
     PARM    .ABSY                    1
     CALL    GLTR..T$0                1
     DPARM   1                        1
     PARM    .T$0                     1
     PARM    .X                       1
     PARM    0                        1
     CALL    GAS..T$0                 1
     DPARM   3                        1
     LD      T$43,.T$0..T$43          1
     BR      L$160                    1
                                                  10
L$157:
     PARM    .ABSY                    1
     CALL    READY..T$0               1
     DPARM   1                        1
     BRF     .T$0.L$161               2
     SSCAL   S$7..+1..T$0             1
     PARM    .T$0                     1
     PARM    .ABSY                    1
     PARM    1                        1
     CALL    GAS..T$0                 1
     DPARM   3                        1
     LD      T$31..T$0..T$31          1
     BR      L$152                    1
                                                  8
L$161:
     PARM    .ABSY                    1
     CALL    GLTR..T$0                1
     DPARM   1                        1
```

```
        PARM    .T$0                        1                          *
        PARM    .X                          1                          *
        PARM    1                           1                          *
        CALL    GAS..T$0                    1                          *
        DPARM   3                           1                          *
        PARM    .T$0                        1                          *
        CALL    GNEG..T$0                   1                          *
        DPARM   1                           1                          *
        LD      T$31..T$0..T$31             1                          *
                                                                       12
L$162:
        LD      T$43..T$31..T$43            1                        *
                                                                     26
L$160:
        LD      T$44..T$43..T$44            1                       *
        BR      L$156                       1                       *
                                                                    41
L$155:
        SSCAL   S$7..+1..T$0                1                       *
        PARM    .T$0                        1                       *
        PARM    .ABSY                       1                       *
        AND     .Y.100000..T$43             2                       *
        OR      .F..T$43..T$43              1                       *
        PARM    .T$43                       1                       *
        CALL    GAS..T$0                    1                       *
        DPARM   3                           1                       *
        LD      T$44..T$0..T$44             1                       *
                                                                    10
L$156:
        LD      T$42..T$44..T$42            1                     *
                                                                  57
L$150:
        LD      T$35..T$42..T$35            1                    *
                                                                 94
L$146:
        LD      T$40..T$35..T$40            1                  *
                                                               101
L$142:
        LD      T$14..T$40..T$14            1                 *
                                                              120
L$130:
        LD      T$33..T$14..T$33            1                *
                                                             194
L$126:
        LD      T$26..T$33..T$26            1               *
                                                            203
L$122:
        LD      T$41..T$26..T$41            1              *
                                                           238
L$116:
        LD      T$5..T$41..T$5              1             *
                                                          250
L$106:
        LD      T$30..T$5..T$30             1            *
                                                         303
L$100:
        LD      T$27..T$30..T$27            1           *
                                                        323
L$60:
        LD      T$11..T$27..T$11            1          *
                                                       422
L$56:
        LD      T$25..T$11..T$25            1         *
                                                      430
L$50:
        LD      T$3..T$25..T$3              1        *
                                                     472
L$46:
        LD      T$6..T$3..T$6               1       *
                                                    478
L$36:
        LD      T$4..T$6..T$4               1      *
                                                   518
L$32:
        LD      T$2..T$4..T$2               1     *
```

```
                                            538
LS2:
    LD       T$0..T$2..T$0                1  *
    RTRN                                  1  *
                                            697

        TOTAL COST= 697
```

Please Note:

Pages 135-137, Appendix B: "Algorithm 419, Zeros of a Complex Polynomial [C2]," copyright 1972 by Association for Computing Machinery, Inc., not microfilmed at request of author. Available for consultation at the Carnegie-Mellon University Library.

University Microfilms.

GLOSSARY OF DEFINITIONS

SYMBOL	DEFINED TERM	PAGE
$\prec$	essential ordering	18
$\triangleleft$	initial ordering	20
$<$	necessary constituent	23
$\ll$	essential predecessor	24
$\diamond$	independent	25
$\cong$	congruence	26
$\simeq$	strongly similar	28
$=$	common subexpression	29
β	linear block	31
	prolog	33
	epilog	33
	postlog	33
$\wedge$	formal intersection	36
	interval	49
	cover	50
	redundant expression	57

BIBLIOGRAPHY

[ASU70] Aho, A.V., Sethi, R. and Ullman, J.D. A formal approach to code optimization, Proc. ACM SIGPLAN Symposium on Compiler Optimization, SIGPLAN Notices, Vol. 5, No. 7 (July 1970), 86-100.

[AU70] Aho, A.V. and Ullman, J.D. Transformations on straight line programs. Proc. Second Annual ACM Symposium on Theory of Computing, (May 1970), 136-148.

[AL60] Naur, P. (Ed.) Revised report on the algorithmic language ALGOL 60, Comm. ACM 6, 1 (Jan. 1963), 1-17.

[A69] Allen, F.E. Program optimization. In Annual Review in Automatic Programming, Vol. 5, Pergamon, New York, 1969.

[A70] Allen, F.E. Control flow analysis. Proc. ACM SIGPLAN Symposium on Compiler Optimization, SIGPLAN Notices, Vol. 5, No. 7 (July 1970), 1-19.

[B71] Bliss reference manual. Computer Science Department Report, Carnegie-Mellon University, Pittsburgh, Pennsylvania, Oct. 1971.

[BE69] Busam, V.A. and Englund, D.E. Optimization of expressions in Fortran. Comm. ACM 12, 12 (Dec. 1969), 666-674.

[C70] Cocke, J. Global common subexpression elimination. Proc. ACM SIGPLAN Symposium on Compiler Optimization, SIGPLAN Notices, Vol .5, No. 7 (July 1970), 20-24.

[CS70] Cocke, J. and Schwartz, J.T. Programming Languages and their Compilers, Preliminary Notes, Courant Institute of Mathematical Sciences, New York University, New York, April 1970.

[D68] Dijkstra, E.W. "Goto statement considered harmful", Letter to the editor, Comm. ACM 11,3 (Mar. 1968).

[G65] Gear, C.W. High speed complation of efficient object code. Comm. ACM 8, 8 (Aug. 1965), 483-488.

[GE72] Gerhart, S.L. Verification of APL programs, Ph. D. dissertation, Carnegie-Mellon University, in progress.

[H72] Hopkins, M.E. A case for the goto, Proc. ACM Annual Conference, Aug. 1972, 787-790.

[JT72] Jenkins, M.A. and Traub, J.F. Zeros of a complex polynomial, Comm. ACM 15, 2 (Feb. 1972), 97-99.

[LM69] Lowry, E. and Medlock, C.W. Object code optimization. Comm. ACM 12, 1 (Jan. 1969), 13-22.

[N65] Nievergelt, J. On the automatic simplification of computer programs. Comm. ACM 8, 6 (June 1965), 366-370.

[P70] PDP10 reference manual. Digital Equipment Corporation, Maynard, Massachusetts, 1970.

[SS69] Shapiro, R.M. and Saint, H. The representation of algorithms. Rome Air Developement Center Technical Report, RADC-TR-69-313, Vol. 2, Sept. 1969.

[W71] Wulf, W.A. Programming without the goto, IFIP, 1971.

[W72] Wulf, W.A. A case against the goto, Proc. ACM Annual Conference, Aug. 1972, 791-796.

[WRH71] Wulf, W.A., Russell, D.B. and Habermann, A.N. Bliss: a language for systems programming. Comm. ACM 14, 12 (Dec. 1971), 780-790.