Data structures for data flow analysis and the elimination of common subexpressions

Janet M. Laubenstein
Lehigh University

Follow this and additional works at: http://preserve.lehigh.edu/etd

Recommended Citation
DATA STRUCTURES FOR DATA FLOW ANALYSIS

AND THE

ELIMINATION OF COMMON SUBEXPRESSIONS

by

Janet M. Laubenstein

A Thesis
Presented to the Graduate Committee
of Lehigh University
in Candidacy for the Degree of
Master of Science
in
Computer Science

Lehigh University
1993
This thesis is accepted and approved in partial fulfillment of the requirements for the degree of Master of Science in Computer Science.

May 2, 1993

(date)

Professor in Charge

CS Division Chairman

CSEE Department Chairman
TABLE OF CONTENTS

ABSTRACT 1

CHAPTER 1 -- INTERMEDIATE CODE GENERATION 3
   Source To Grammar Symbol Translation 3
   Emitting Quadruple Code 5
      Quadruple Code Structure 5
      Temporary Symbol Usage 8
   Representing Expressions 9
   Representing Constants 11
   Symbol Table Structure 13
   Representing Control Structures 14

CHAPTER 2 -- THE FLOW GRAPH 18
   Basic Block Determination 18
   Basic Block Structure 21
   Branching Statements 23
   Building the Flow Graph 25
   Predecessor and Successor Functions 26
   Predecessor Block Determination 28
   Building a Predecessor Block List 29

CHAPTER 3 -- GLOBAL DATA FLOW ANALYSIS 31
   Use-Definition of Symbols 31
   Reaching Definitions 32
   Definitions Generated within Blocks 33
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Definitions Killed by Blocks</td>
<td>35</td>
</tr>
<tr>
<td>Data Flow Equations for Reaching Definitions</td>
<td>35</td>
</tr>
<tr>
<td>Solving the Data Flow Equations</td>
<td>36</td>
</tr>
<tr>
<td>Topological Sorting for Equation Solution</td>
<td>37</td>
</tr>
<tr>
<td><strong>CHAPTER 4 -- COMMON EXPRESSIONS</strong></td>
<td>40</td>
</tr>
<tr>
<td>Common Expressions Spanning Blocks</td>
<td>40</td>
</tr>
<tr>
<td>Available Expressions</td>
<td>43</td>
</tr>
<tr>
<td>Generated and Killed Expressions</td>
<td>44</td>
</tr>
<tr>
<td>Determining Available Expressions</td>
<td>49</td>
</tr>
<tr>
<td>Eliminating Common Subexpressions</td>
<td>51</td>
</tr>
<tr>
<td><strong>CHAPTER 5 -- INTERPROCEDURAL DATA FLOW</strong></td>
<td>54</td>
</tr>
<tr>
<td>Recognizing Procedures</td>
<td>54</td>
</tr>
<tr>
<td>Procedure Calls</td>
<td>54</td>
</tr>
<tr>
<td>Definitions Generated by Procedures</td>
<td>55</td>
</tr>
<tr>
<td>Actual and Formal Parameters</td>
<td>56</td>
</tr>
<tr>
<td>The Calling Graph</td>
<td>57</td>
</tr>
<tr>
<td>Common Expression in Multiple Procedures</td>
<td>58</td>
</tr>
<tr>
<td>Additional Work</td>
<td>59</td>
</tr>
<tr>
<td><strong>BIBLIOGRAPHY</strong></td>
<td>61</td>
</tr>
<tr>
<td><strong>APPENDIX A -- The Edison Subset</strong></td>
<td>62</td>
</tr>
<tr>
<td><strong>APPENDIX B -- Quadruple Structure</strong></td>
<td>65</td>
</tr>
<tr>
<td><strong>APPENDIX C -- NXT_QUAD and NXT_BLOK Usage</strong></td>
<td>67</td>
</tr>
<tr>
<td><strong>VITA</strong></td>
<td>68</td>
</tr>
</tbody>
</table>
Data Structures for Data Flow Analysis
and the
Elimination of Common Subexpressions

Janet M. Laubenstein

ABSTRACT

This paper describes the implementation of data structures required to perform data flow analysis and the use of these structures by an optimizing algorithm that eliminates common subexpressions.

The analysis and optimization is performed using source programs written in a subset of the Edison programming language. Constructs for concurrent statements, module declarations and library procedures which are included in the Edison language designed by Per Brinch Hansen are not supported in the subset. Neither are structured data types except for arrays. An initial program translates the Edison source program into numeric grammar symbols.

Construction of a symbol table and a flow graph for the Edison source code is performed by a second program using the numeric grammar symbols. The flow graph forms the basis for analyzing the state of all variables during program execution. These states are
represented by sets of statements which change or use the contents of the variables. The sets are formed by solving a series of data flow equations which inspect the flow graph. The last phase of the second program identifies and eliminates expressions that are needlessly recomputed during program execution. The final output of this second program is an optimized version of the source program represented by an intermediate code in the form of quadruples.
CHAPTER 1 -- INTERMEDIATE CODE GENERATION

SOURCE TO GRAMMER SYMBOL TRANSLATION

The preliminary program performs the lexical scanning, parsing and scope analysis of an Edison source program. It uses a recursive descent parser to check that the program conforms to the valid syntax and scope rules for the Edison subset. The subset of Edison that is used does not include record or enumerated data structures, concurrent statement structures or module definitions. A complete list of all productions in the subset is contained in Appendix A.

The output from this initial program is the grammar symbols and unique integer values that are assigned to each declared identifier. These identifiers are used to define constants, variables, procedures, functions and user-defined data types, specifically arrays. During this phase, a scaled-down symbol table is maintained as an aid in resolving ambiguous identifier references. When identifiers for named items within the Edison source code are encountered, they are entered into the symbol table, if not already there, and they are assigned the index value of the symbol table entry as their symbol value.
The symbol table contains fields for the minimum and maximum level numbers to indicate the scope of each identifier. The level number is increased during the parsing process each time a procedure heading is found and is decreased at the end of each procedure declaration. In this way, references to objects by the same name but in different blocks will be able to be distinguished from one another and will have unique symbol table index values.

Errors in syntax or scope generate a special error symbol to be placed in the output file but do not halt the preliminary program execution. These errors do not immediately halt the initiation of the execution of the second program which performs the data flow analysis. It is not until the error symbol is read in phase 2 during generation of the quadruple code array that the second program terminates.
EMITTING QUADRUPLE CODE

The data flow analysis cannot proceed directly from the grammar symbols output by the preliminary program. Two additional structures must be built before any flow analysis can be attempted. These structures are the intermediate code consisting of a quadruple array and a control flow graph of the Edison program execution. A second program uses the grammar symbols output from the preliminary program and constructs the symbol table for the symbols defined in the given Edison source program. It also assigns temporary variables from unused locations in the symbol table as needed when generating the quadruple code that will be used during the optimization process. In addition, type checking needs to be completed. To minimize the time required for these activities, the semantic analysis and generation of the quadruple code have been combined in one pass of the file produced by the preliminary program.

Quadruple Code Structure

An intermediate coding scheme in the form of quadruple instructions establishes the framework for the data flow analysis. The quadruples are stored in
an array whose elements are records containing a quadruple opcode followed by three arguments. Not all opcodes require entries for all three arguments but this is the maximum needed for some quadruples. The structure of the quadruple code is modeled after the formats described by Aho and Ullman in *Principles of Compiler Design*. [Aho79] The arguments themselves are pointers to the appropriate variables, procedures or constants in the symbol table.

Quadruples that require three arguments are quadruples that assign values to variables after the evaluation of a binary operation or those that involve assignments to or use of array elements. Binary operations in the Edison subset are the standard arithmetic operations of addition, subtraction, multiplication, division, and modulo, the logical operations of AND, OR and NOT and the relational operations of equal, greater, less, notequal, notgreater and notless. These quadruples have the following form:

\[
\begin{align*}
A & := B \text{ op } C \\
A[I] & := B \\
A & := B[I]
\end{align*}
\]
Two-argument quadruples are produced for the unary operations of minus (negation) and NOT, for assignments from one variable to another and for conditional GOTO statements such as:

\[
\begin{align*}
A & := \text{op } B & \text{op } & B <\text{none}> A \\
A & := B & \text{assign } B <\text{none}> A \\
\text{if not } A & \text{ goto } L & \text{eqfalse } A <\text{none}> L
\end{align*}
\]

Quadruples generated for procedure or function calls take the following forms:

\[
\begin{align*}
\text{valparam } A <\text{none}> <\text{none}> \\
\text{varparam } A <\text{none}> <\text{none}>
\end{align*}
\]

which monitor actual parameters and enable linking or allocation of actual parameters immediately prior to procedure calls. The following quadruple is generated for the procedure call itself,

\[
\begin{align*}
\text{call } P, L & \text{ proccall } P <\text{none}> L
\end{align*}
\]

where \( P \) is a pointer to the symbol table entry of the procedure or function and \( L \) is the target quadruple.
which is the first executable statement in the subprogram. It should be noted that the symbol table entry for procedure P contains a pointer to a linked list of formal parameters and that linking formal and actual parameters appropriately can be accomplished using this chain.

The unconditional branch statement used for implementing the IF-THEN-ELSE structure as well as the WHILE loop is a one-argument quadruple. Its format is:

\[
goto <\text{none}> <\text{none}> L
\]

where \( L \) is the target quadruple.

Additional quadruple opcodes that have been incorporated are the return and endcode which are used to signify the end of the subprogram statements and end of program respectively. These opcodes require no arguments.

Temporary Symbol Usage

Several problems arise during the construction of the quadruple code. Expressions involving more than one operand must be broken down into the appropriate binary, or in some cases unary, operations. The
question then becomes where to store the results of these partial calculations. In addition to this, since the preliminary program reduced all string names to integer values, it is necessary to ensure that literal values for integers can be correctly differentiated from variable name symbols. And because recursive descent is likewise used in this second phase, we have the necessary back-patching of GOTO labels for IF and WHILE statement constructs.

Representing Expressions

To deal with the difficulty with compound expressions, a stack is maintained to hold all references to symbol names that are used in executable statements. When a variable name is encountered as the target of an assignment statement or as an operand in any expression, its corresponding symbol table index value is pushed onto the stack. Constant names when used as operands are treated similarly. In the case of a binary operation, for example \(A + B\), it is not known in which context this expression may be used. In fact, it could be part of any of the example statements below or one of many other possibilities.
1) \[ X := L[A + B] + C \]
2) \[ X := A + B \times C \]
3) \[ X := A + B \]

In example 1) it is not difficult to observe that variable \( X \) and \( L \) must be pushed onto the stack until we can parse the expression for the subscript. Subsequently, the quadruple sequence listed below would result.

\[
\begin{align*}
X := L[A + B] + C &\implies T_1 := A + B \\
T_2 := L[T_1] &\\
T_3 := T_2 + C &\\
X := T_3 &
\end{align*}
\]

Example 2) would produce the quadruples listed below.

\[
\begin{align*}
X := A + B \times C &\implies T_1 := B \times C \\
T_2 := A + T_1 &\\
X := T_2 &
\end{align*}
\]

The resulting quadruples for example 3) are:

\[
\begin{align*}
X := A + B &\implies T_1 := A + B \\
X := T_1 &
\end{align*}
\]
The introduction of the temporary variables, denoted by $T_n$, is necessary until we know the exact usage of the variables that appear in assignment statements. The second phase procedures allocate a temporary variable for each unary and binary operator that is encountered and this process does result in some unnecessary copy propagations as in 1) above where

$$T_3 := T_2 + C$$
$$X := T_3$$

could be replaced by the single assignment,

$$X := T_2 + C.$$ 

Algorithms to reduce copy propagations within blocks can be applied by constructing directed acyclic graphs (DAGs) for each unique block of the flow graph which is created during the next phase. The subject of local optimizations is beyond the scope of this paper's investigations and is not addressed further.

Representing Constants

Various methods could be employed that would allow us to recognize the use of a literal value versus a variable or constant name represented as an integer. One could maintain additional flag fields in each
quadruple record to indicate if an argument is a literal integer value. Since we are using two arguments, we would need a flag for each argument. The result field, of course, must be a variable and would not require a flag. The disadvantage is that this requires a larger record for each quadruple when many quadruples will not necessarily need to make use of these fields.

Not wanting to expand the code array any further, the method chosen for the second phase involves assigning each literal value a temporary entry in the symbol table. The symbol table already includes fields for distinguishing the entry as a constant type as well as its constant value. The program makes no attempt to search the symbol table for temporary literals of a specific value being present before allocating a new temporary constant. It was felt that the time it would take for this activity would not be worth the effort since there were plenty of free entries in the symbol table anyway. Most likely professional programmers would declare all necessary constants (except perhaps zero and one) in a constant declaration section within the Edison source program making a search of the table an unnecessary and time-consuming activity. It might be worthwhile to include two additional standard
constants of zero and one in the symbol table since these literal values are frequently used and would eliminate multiple occurrences of these literals to be assigned temporary locations. To accommodate these new constants, the preliminary program could be modified to output the corresponding grammar symbols essentially handling them in a manner similar to the standard constants of TRUE and FALSE.

Another possibility is to maintain the temporary constants in a separate table. This offers no real advantages over putting them in the main symbol table except if searching the table for specific literal entries is implemented. In addition, the problem of distinguishing between index values for variables and constants in the main symbol table and index values of temporaries in the temporary table arises again.

Symbol Table Structure

The symbol table has been implemented as an array of records. Standard identifiers provided with the Edison compiler use the first twenty entries in the array and symbols declared within the Edison source program are allocated space in ascending sequence starting at index value twenty. In order to be able to keep program-declared symbols and temporary symbols
separated in this table, the initial temporary is allocated from the highest index value and subsequent temporaries are assigned slots in descending sequence.

Representing Control Structures

Generating quadruples for the conditional and iterative statements presents a different difficulty. Consider the following Edison WHILE statement.

```
while X <= 10 do
  SUM := SUM + X;
  X := X + 1
end
```

The quadruple code required for the first line, while X <= 10 do is:

```
T_1 := X <= 10
IF NOT T_1 GOTO ???
```

The temporary variable T_1 is introduced because it is not known until the next symbol is retrieved if the expression X <= 10 is the complete Boolean expression or part of a larger expression. The destination for the conditional GOTO is not known until the quadruples
for the statements in the conditional list have been completed. Lastly, an unconditional GOTO will need to transfer control back to the beginning of the evaluation of the expression to implement the looping process. The complete quadruple sequence is

q1: \( T_1 := X \leq 10 \)
q2: IF NOT \( T_1 \) GOTO q8
q3: \( T_2 := \text{SUM} + X \)
q4: \( \text{SUM} := T_2 \)
q5: \( T_3 := X + 1 \)
q6: \( X := T_3 \)
q7: GOTO q1
q8: .....

After the generation of quadruple q7, we must back-patch the target of the conditional GOTO in q2.

The Edison language subset allows any number of 'else' ConditionalStmts for the WHILE statement as well as the IF statement. To illustrate, we can examine the following IF statement.

\[
\text{if } B > 0 \text{ do}
\]
\[
\quad \text{if } A > 0 \text{ do}
\]
\[
\quad \text{C := 1}
\]

15
end
else B = 0 do
C := 0
else true do
C := -1
end

The corresponding quadruple sequence is

q1: \( T_1 := B > 0 \)
* q2: IF NOT \( T_1 \) GOTO q8
q3: \( T_2 := A > 0 \)
* q4: IF NOT \( T_2 \) GOTO q7
q5: \( C := 1 \)
* q6: GOTO q7
* q7: GOTO q11
q8: \( T_3 := B = 0 \)
* q9: IF NOT \( T_3 \) GOTO q12
q10: \( C := 0 \)
* q11: GOTO q15
* q12: IF NOT TRUE GOTO q16
q13: \( T_4 := -1 \)
q14: \( C := T_4 \)
* q15: GOTO q16
q16: . . .
At various points in the process of examining the IF statement, the target quadruples (marked by *) for the conditional and unconditional GOTOs are back-patched. Note in the example above that the quadruples q6 and q7 could certainly be replaced by one quadruple that consists of "GOTO q11". However at the instance when q6 is generated, it is not known if the next symbol is an else or an end and a GOTO must be generated to allow for an else eventuality. This type of chaining series of GOTO statements will be produced whenever nested conditional statements are used. The final GOTO of quadruple q15 is generated for the same reason. The elimination of these unnecessary GOTOs is discussed in the next section which describes the flow graph.
CHAPTER 2 -- THE FLOW GRAPH

Basic Block Determination

In order to perform global data flow analysis, a directed graph is used to represent the flow of control of the program. The vertices of the graph represent segments of code and the edges represent the flow of control between the code segments. Each segment of code that comprises a vertex has the property that if the first instruction in the block is executed then so must each remaining instruction in the block be executed. This means that any branching statement will signify the end of a block, referred to as a basic block, and statements that are the target of branch statements (including subprogram calls) will constitute a new vertex or basic block. It follows that each block will have only one entry point for the start of its execution. The particular type of graph that will be constructed is known as a flow graph which has a unique entry point $B_0$ from which all other nodes can be reached. Since execution of an Edison program always begins with the first statement of the main procedure, a unique entry point is guaranteed.

Once the quadruple code array is intact a flow graph is constructed by first partitioning the quadruples
into basic blocks. Each block will be a sequence of quadruples with one entry point for execution and where execution progresses in simple sequence until the end of the block is reached. There are no quadruples that contain opcodes for conditional or unconditional branching within a block except as the last statement. By the same token, a procedure or function call will indicate the end of a block since they cause a transfer of control to statements that are not in sequence. The quadruple code is not altered at all during this process.

Determination of the leaders begins with the first statement in each program which is always a leader. In addition, leaders consist of any statement that is the target of a GOTO (conditional or unconditional), any statement that is the first statement of a procedure or function (determined by the target in the call) or any statement that follows a conditional GOTO. Any statement that follows an unconditional GOTO must itself be the target of a conditional or unconditional GOTO or subprogram call or it will never be executed and hence could be removed from the program. Since the Edison language does not contain a GOTO statement, this type of programmer error cannot occur and this possibility is not inspected here. This does not mean
that all statements will be executed in a program since there are conditional branches along which the execution may never proceed. There may also be declared procedures that are never called. This phase of the program does not look for this condition as of this writing.

Using the previous example WHILE statement, its basic block structure would be

BLOCK 1 --> q0: . . .

BLOCK 4 --> q1: \( T_1 := X <= 10 \) (leader)

q2: IF NOT \( T_1 \) GOTO q8

BLOCK 2 --> q3: \( T_2 := SUM + X \) (leader)

q4: SUM := \( T_2 \)

q5: \( T_3 := X + 1 \)

q6: X := \( T_3 \)

q7: GOTO q1

BLOCK 3 --> q8: . . . (leader)

Several observations should be noted. The quadruple labelled q8 is a leader because it is the target of a GOTO in q2. A more subtle division of blocks occurs when q7 is inspected during the construction process. It is possible that prior to arriving at q7, the quadruple q0 belonged to the same block as q1 and q2.
But as soon as "q7: GOTO q1" is found, q1 becomes the leader for a new block and q0 which is not a branching statement becomes the last quadruple of the block and its next block field must be adjusted to Block 4. This in effect splits q1 and q2 into a block separate from q0 and the block numbers are no longer in sequence with the quadruple numbers. This is not a problem only an observation.

Basic Block Structure

We represent the flow graph by using a linked list of basic blocks where each node consists of a block number and a pointer to the quadruple that is the leader. The quadruple code is threaded with the list of quadruples that make up the block. A field that contains a pointer to the next quadruple is maintained for this purpose. The last quadruple in each block contains the block number where control of execution will be passed. This "next block number" is maintained in a separate field in the quadruple record.

This arrangement offers the most flexibility when code optimization may involve the elimination of a particular quadruple or when a quadruple may be moved from one block to another. Also, some optimizations, notably the elimination of common subexpressions may
involve the addition of quadruples within a given block.

Recall our example IF statement. Its resulting block designations are listed below.

BLOCK 1 --> q1: $T_1 := B > 0$ (leader)
q2: IF NOT $T_1$ GOTO q8

BLOCK 3 --> q3: $T_2 := A > 0$ (leader)
q4: IF NOT $T_2$ GOTO q7

BLOCK 5 --> q5: C := 1 (leader)
q6: GOTO q7

BLOCK 4 --> q7: GOTO q11 (leader)

BLOCK 2 --> q8: $T_3 := B = 0$ (leader)
q9: IF NOT $T_3$ GOTO q12

BLOCK 8 --> q10: C := 0 (leader)

BLOCK 6 --> q11: GOTO q15 (leader)

BLOCK 7 --> q12: IF NOT TRUE GOTO q16 (leader)

BLOCK 11 --> q13: $T_4 := -1$ (leader)
q14: C := $T_4$ (leader)

BLOCK 9 --> q15: GOTO q16 (leader)

BLOCK 10 --> q16: . . . (leader)

We previously discovered that quadruples q6 and q7 could be combined into one quadruple that reads GOTO q12. It is imperative that a quadruple that is a
leader is never eliminated since it must be the target of a GOTO and perhaps may be the target of multiple GOTOS as in the case of q7. So, in this case, q6 can be eliminated which would cause Block 3 to contain just one quadruple q5. This can be accomplished by updating q5's Next Block field to the value of Block 4 thus indicating the end of a block.

Branching Statements

For conditional GOTOS and returns from subprogram calls the "next block number" is not sufficient to enable construction of the flow graph so the following exceptions have been employed.

First, for conditional branches, the flow graph requires that we know to which block control should pass if the condition is TRUE as well as where to pass control if the condition is FALSE. Rather than maintain separate fields in the quadruple array and enlarge the record further the Next Quadruple field contains the block number to transfer control to if the condition is TRUE and Next Block field contains the block number to go to if the condition is FALSE. The second exception involves procedure or function calls which are, in essence, unconditional GOTOS but are handled differently because of the return address
problem. This is due to the last executable statement in any procedure or function being a RETURN opcode that contains no arguments. Since the return address will be retrieved from the run-time stack during execution, this information is not available during the construction of the flow graph. Also, procedures will need to be able to be recognized later as we perform data flow analysis in order to know which variables passed by reference may be altered by the execution of the subprogram. Multiple calls to the same procedure from different parts of the program preclude us from storing the block number to return to within the procedure itself, so the block number that should gain control upon return from the procedure call must be stored in the "next quadruple" field of the quadruple containing the procedure call. Since a procedure call is a form of branching statement and will always signal the end of a block, the quadruple for the procedure call must contain the first block number of the procedure body in the "Next Block" field. See Appendix C for a summary of the usage of the NXT_QUAD and NXT_BLOK fields in the quadruple records.
Building the Flow Graph

Partitioning the program into basic blocks proceeds according to the following algorithm.

For each quadruple in the quadruple code array

If it is the first quadruple (Quad # 0)

Create a Block with Leader 0

else

if quadruple's opcode = UNCONDITIONAL GOTO | CONDITIONAL GOTO | SUBPROGRAM CALL

Create a Block with the target quadruple as the leader (* BRANCH BLOCK *)

if quadruple's opcode = CONDITIONAL GOTO | SUBPROGRAM CALL

Create a Block with the next quadruple in the code array as the leader (* CONTINUE BLOCK *)

pointer to next quadruple= CONTINUE BLOCK #

pointer to next block= BRANCH BLOCK #

else

(* THIS IS A UNCONDITIONAL GOTO *)

pointer to next quadruple = nil

pointer to next block = BRANCH BLOCK #

else

if quadruple opcode = RETURN from PROCEDURE CALL

pointer to next quadruple = nil

pointer to next block = nil

else

(* NOT A BRANCHING STATEMENT *)

pointer to next quadruple = next quad #

pointer to next block = nil
Predecessor and Successor Functions

As stated earlier, the edges or arcs of the flow graph represent the flow of control between the basic blocks. Vertices that are connected by an arc are said to be adjacent. In the case of a directed graph an arc that connects two vertices, $V_1$ and $V_2$, as illustrated below,

$$V_1 * \longrightarrow * V_2$$

we say that $V_1$ is adjacent to $V_2$ and that $V_2$ is adjacent from $V_1$. These adjacency relationships allow us to define predecessor and successor operations on the flow graph which are necessary for its traversal. The successor operation, denoted by $\text{ScB}(X)$, is the set of all vertices adjacent from a given set of vertices $X$. The predecessor operation, denoted by $\text{PrB}(X)$, is the set of all vertices adjacent to a given set of vertices $X$.

Incidence is the term used to specify the relation between vertices and edges. Each edge of a graph is incident upon precisely two vertices that are its end points. The degree of a given vertex is the number of edges that are incident upon it and may vary from 0 to
V-1 where V is the number of vertices in the graph. For any vertex X in a directed graph, we refer to the in-degree of X as the number of predecessors or \(|\text{PrB}(X)|\) and the out-degree of X as the number of successors or \(|\text{ScB}(X)|\).

For our Edison program flow graph, the in-degree for block 0, the initial node, is always zero. The in-degree of other blocks will be 1 if they are not the target of branching statements and will be greater than 1 if they are the target of branching statements.

The out-degree of a given block can be ascertained to be zero if the block ends in a return from a procedure call including a return from the main procedure call which would signify the end of the execution of the program. The out-degree of a block will be 1 if the statement following the last statement in the block is the target of a branch statement or if the last statement in the block is a GOTO statement. If the last statement in the block is a conditional branch statement or procedure call, both of which require two target blocks or vertices then the out-degree will be 2. No blocks will have out-degrees that are greater than 2.
Predecessor Block Determination

In order to perform data flow analysis it is necessary to determine which blocks were executed immediately prior to each block. Depending on the branching of the program either through conditional or unconditional GOTOs or subprogram calls, any block may have multiple predecessor blocks. A predecessor block can be defined as one that passes control to a given block via a conditional or unconditional jump or a block that immediately precedes the given block in the order of the program and which does not end in a jump. After the construction of the flow graph is complete, the graph must be traversed for the purpose of building a predecessor block list for each vertex in the graph.

Some Edison programs that contain no procedure calls or loops will result in flow graphs that consist only of a simple path which is a sequence of nodes \( B_0 B_1 B_2 \ldots B_j \) where all the nodes are distinct and where each pair of nodes \( B_i B_{i+1}, i=1,\ldots,j-1 \) is connected by an edge. Some Edison programs however may result in a flow graph that contains a cycle which is a simple path \( B_0 B_1 B_2 \ldots B_j \) as described above except that \( B_0 \) and \( B_j \) are the same node. If cycles are present then any traversal will cause some vertices to be encountered more than once. Recursive procedures and while constructs are the cause
for circuits in the flow graph. It is also possible that some vertices may not be encountered at all. Any vertex that does not appear in the search tree from the initial node $B_0$ will never be executed and can be eliminated. The only situation that could cause this to happen would be a procedure that is declared but never called.

Building a Predecessor Block List

A Depth First Search (DFS) can proceed by visiting and marking each vertex $X$ starting at the initial node $B_0$. The elements of $\text{ScB}(X)$, its successors, are examined one at a time. The successors are found by inspecting the last statement or quadruple in the basic block that represents vertex $X$. Recall that the number of successors to $X$ is its out-degree which is known to be either 0, 1 or 2. If a vertex $Y$ in $\text{ScB}(X)$ is unmarked, DFS is immediately applied to it. Vertex $X$ is added to $\text{PrB}(Y)$ regardless of whether or not vertex $Y$ was previously visited. Since the graph is directed but may contain cycles the process must mark each node as visited to eliminate an endless circular traversal. The traversal ends for a given path when either it returns to a node that has already been marked as visited or it encounters a RETURN from PROCEDURE CALL.
opcode as the last statement in a block. It should be noted that every Edison quadruple code array generated by this program contains at least one RETURN opcode that is placed in the main execution block.
CHAPTER 3 -- GLOBAL DATA-FLOW ANALYSIS

Global data-flow analysis involves examination of the entire program using the flow graph and the solution set to the data-flow equations that are described below. In particular, analyses of variable use and assignments of values to the variables can identify unnecessary computations or unused variables and permit substitution of a constant value for a variable. Use-definition chaining can determine at what points in a program the value of a given variable A could have been defined.

Use-Definition of Symbols

The opcodes of the quadruples can be grouped into the following basic categories in order to determine its flow function:

1) arithmetic operations that use the first and second arguments as operands and store the result in the third argument

2) single argument assignments that store the first argument's value (either constant or variable) in the third argument

3) operations that do not alter the value of the third (or any other) argument.
The term use of an identifier refers to an occurrence of it as an operand, that is either the first or second argument in the quadruple. The definition of an identifier refers to an assignment to the identifier and can be found by examining the result field (third argument) of a quadruple whose opcode is a member of category 1) or 2) above. Since there can be only one definition of an identifier per quadruple, it is sufficient to note only the quadruple number when constructing the definition lists for each block.

Examination of the third argument of a particular quadruple is used to find the appropriate symbol table entry.

Reaching Definitions

We say a definition of a variable A reaches a point p if there is path in the flow graph from that definition of A to p, such that no other definitions of A appear on the path. [Aho79] To find these reaching definitions, two sets are assembled for each basic block in the program. First, a list of all definitions that reach the end of a block is built and is referred to as the generated definitions. The second set is comprised of the set of definitions outside of a given block B, that define identifiers that also have definitions in the
given block $B_i$. These outside definitions are necessarily replaced by the definitions within the block under examination and are referred to as the killed definitions. At the same time that the generated definitions are computed, each definition that reaches the end of a block is placed in a list in the appropriate symbol table entry for the identifier in question so that each variable in the program has a list of all quadruples that make assignments to it regardless of the block number in which the assignment occurs. This information can ultimately be used to form the ud-chain.

Definitions Generated within Blocks

When determining generated definitions, it is required that each symbol table entry contain a field (called defn henceforth) which contains the quadruple number of the most recent definition of an identifier in the block $B_k$ that is being inspected. This field is initialized to zero at the beginning of the inspection of each block. Whenever a definition for the given symbol $S_i$ is found in block $B_k$ the defn field is updated with the quadruple number of this new definition. For symbol $S_i$, the symbol table bit vector for all definitions of $S_i$ that reach the end of a block is updated during the
determination of all generated definitions which proceeds according to the following algorithm.

For each block B

Set all defn fields in symbol table to zero

Initialize dfGen[B] to /

For each quadruple q in block B

if q is a Definition** of i

if i has been previously defined in block B

then

In the symbol table entry for i

remove q from the definition list

Remove q from the generated definition list for block B

In the symbol table entry for i

Set the defn field to q

Add q to the definition list

Add q to the generated definition list for block B

** Definitions can be determined by the opcode of the quadruple. The identifier represented by i will always be the third argument of the quadruple.
Definitions Killed by Blocks

Determination of the definitions that are killed by definitions with any given block $B_k$ cannot be completed until after the generated definitions are known. These killed definitions are represented as a bit vector in the block table for the block $B_k$ and is established according to the following algorithm.

For each block $B$

Initialize $df\text{ Kill}[B]$ to $\emptyset$

For each definition $d$ of identifier $i$ in $df\text{ Gen}[B]$

Using the symbol table entry for $i$

$df\text{ Kill}[B]$ is the set of all definitions in the definition list of identifier $i$

NOT including definition $d$

Data Flow Equations for Reaching Definitions

The final step required in finding the reaching definitions is to build the set of all definitions for each block $B$ that reach the point just before the first statement of block $B$ and the set of definitions reaching the point just after the last statement of each block $B$. These sets are described by the following data-flow equations. [Aho79]
For all blocks B

(1) \( \text{dfOut}[B] = (\text{dfIn}[B] - \text{dfKill}[B]) \cup \text{dfGen}[B] \)

(2) \( \text{dfIn}[B] = \bigcup \text{dfOut}[P] \)

where \( P \) is a predecessor of \( B \)

This means that a given definition is a member of \( \text{dfOut}[B] \) provided that either a) it reaches the point just before \( B \) and is not killed by \( B \) or b) it is a definition that is generated by \( B \). For \( \text{dfIn}[B] \), any definition that reaches \( B \) must reach the end of one of \( B \)'s predecessors.

Solving the Data Flow Equations

An iterative method is used in this phase to solve these equations and starts by initializing \( \text{dfIn}[B] \) to the empty set and \( \text{dfOut}[B] \) to the definitions generated within \( B \) (i.e. \( \text{dfGen}[B] \)) for each block \( B \) in the flow graph. Then \( \text{dfIn}[B] \) is recomputed to be the union of all the definitions contained in \( \text{dfOut}[P] \) for each block \( P \) that is a predecessor to \( B \). Recall that the list of predecessor blocks has been computed and is contained in the record for the basic block \( B \). If this recomputed \( \text{dfIn}[B] \) is different from the \( \text{dfIn}[B] \) set before the recomputation then the successor block(s) must be examined since \( \text{dfOut}[B] \) may have changed as well. The
process continues inspecting each block in this manner until no further changes to dfIn[B] are encountered upon its recomputation for any block.

The rate at which the computations of the two sets dfIn[B] and dfOut[B] stabilize is dependent upon the size of the flow graph. The manner in which DFS partitions the vertices of a digraph is dependent upon the starting vertices. To obtain solutions to the data flow equations for each vertex it is necessary to conduct DFS from each vertex in turn. [Smi87] The flow graph contains no more than two successors as determined by its out-degree and the sample programs are not large hence the time required for this phase is not unmanageable.

**Topological Sorting for Equation Solution**

An alternate method of solving the data flow equations for dfIn and dfOut requires partitioning the graph into units larger than basic blocks. This minimizes the time spent generating the reaching definitions by enabling us to find the dominators of each node. Dominator information permits access to the nodes in the same order that program execution will proceed along the edges of the flow graph. Hence it is possible to gather information about predecessor blocks while traversing the graph instead of treating each block as a separate entity.
and storing any needed information about predecessor blocks within each block record.

The algorithm for this method involves partitioning the graph into intervals where each interval of a node \( B \) is the largest subgraph where \( B \) is the only entry node and all loops contain \( B \). The edges of this reduced graph can be grouped into two sets denoted as forward edges and back edges. The forward edges form a directed acyclic graph or DAG in which every node is reachable from an initial node \( B_0 \) and the back edges consist only of edges whose heads dominate their tails.

If a programming language such as the Edison subset which is used here, allows only structured flow of control statements, then multiple entry points into the middle of loops in the flow graph will never occur. This means that each loop will consist of a unique entry point and hence will ultimately produce a reducible flow graph. Testing a flow graph to determine if it is reducible can be done by inspecting each edge just once and this activity can replace the procedure that builds the predecessor block list in the flow graph.

DFS is also used in this alternate method to discover the appropriate topological ordering of the vertices for the dominance relation, that is, where the vertices are ordered such that there is a path from \( B_i \)
to $B_j$ and where $B_j$ appears after $B_i$ in the ordering. A
topological ordering is not possible if the graph has a
cycle since for two vertices $B_i$ and $B_j$ on a cycle $B_i$ would
precede $B_j$, and vice versa.

Pre-order topological sorting algorithms mark each
node in ascending sequence upon entry into the procedure
starting with the initial node. Post-order topological
sequence can be obtained by marking each node just before
exiting the procedure. Reverse post-order topological
sequence is obtained by marking each node just before
exiting but with values that start with the number of
nodes and decrease with each marking.

If a graph is reducible and if the block nodes are
accessed in reverse post order sequence via DFS
topological sorting then the dominators can be computed
in one direct pass without iteration. [Smi87] This would
allow the computation of the dfIn and dfOut sets by
accessing the nodes in reverse post order sequence and
accumulating the information about definitions for
dfIn($B_k$) as the graph is traversed. That is to say,
dfIn($B_k$) = dfOut($B_{k-1}$).
CHAPTER 4 -- COMMON EXPRESSIONS

It may occur in a program that two or more different blocks perform the same calculation using the same variables. If the evaluation of the expression in one block precedes the evaluation of the same expression in the other block then the last evaluation is said to be a common subexpression provided the operands have not been redefined since the first evaluation. A common subexpression can be eliminated by storing the value of the original expression in a temporary and replacing each subexpression with this temporary.

Common Expressions Spanning Blocks

Elimination of common subexpressions across blocks is complicated by several situations. If a computation, \( C := \text{op} \, A, A \, z \) is executed at points \( p_m \) and \( p_n \) in two different blocks \( B_i \) and \( B_j \) respectively and if there exists a path from \( B_i \) to \( B_j \) then the expression \( \text{op}A, A \, z \) is common to both blocks provided that \( A, \) and \( A \, z \) have not been defined along the path between the two blocks from point \( p_m \) to \( p_n \). However, if the in-degree of \( B_j \) is greater than 1, it is not sufficient to check only for definitions of \( A, \) and \( A \, z \) along the path from \( B_i \) to \( B_j \). Each path to \( B_j \) must be inspected for definitions of \( A, \)
and $A_2$. In fact, it may be possible for the program to reach $B_j$ without going through $B_i$. The existence of procedure calls along paths to $B_j$ and the presence of aliases for $A_1$ and $A_2$ within the blocks of called procedures may also result in the values of $A_1$ and $A_2$ being altered between statements $p_m$ and $p_n$.

As an example, consider the following Edison code segment.

1) $X := Y \times Z$;
2) $I := 0$;
3) $J := 1$;
4) WHILE $J \leq Y$ DO
5) $I := I + X$;
6) $J := J + 1$
7) END;
8) $K := Y \times Z$;
9) . . .

The expression $Y \times Z$ on line 8 is a common subexpression of the same computation performed on line 1. By observation, we can see that no assignments to the variables $Y$ and $Z$ have been made between lines 1 and 8. The resulting flow graph and quadruple code for this is listed below.
Block 1 -->  
q1: \( T_1 := Y \times Z \) (leader)*
q2: \( X := T_1 \)
q3: \( I := 0 \)
q4: \( J := 1 \)

Block 4 -->  
q5: \( T_2 := J \leq Y \) (leader)
q6: IF NOT \( T_2 \) GOTO q12

Block 3 -->  
q7: \( T_3 := I + X \) (leader)
q8: \( I := T_3 \)
q9: \( T_4 := J + 1 \)
q10: \( J := T_4 \)
q11: GOTO q5

Block 2 -->  
q12: \( T_5 := Y \times Z \) (leader)
q13: \( K := T_5 \)
q14: . . .

* assumes q1 is the first quadruple in code array

The separation of the code into basic blocks has caused the common expression to be placed in two different blocks with the first occurrence of the expression in Block 1 at q1 and the second occurrence in Block 2 at q12. Notice that the flow of control from block 1 does not pass directly to Block 2. Instead, there are two paths to Block 2 from Block 1, namely,
1) Block 1, Block 4, Block 2  (if the while statement body does not execute)

and

2) Block 1, Block 4, [Block 3, Block 4]^*, Block 2
   (if the loop does execute)

Hence, any algorithm that can detect the common subexpression at q12 must be able to verify that variables Y and Z have not been redefined in either Block 4 or Block 3.

Available Expressions

Solving a set of data flow equations that are similar in structure to the equations used in the determination of reaching definitions can be used to determine available expressions at any point in a program. The expression \( \text{opA}_1 \text{A}_2 \) is available at point \( p_n \) in block \( B_j \) if \textbf{every} path from the initial node \( B_0 \) to \( p_n \) in block \( B_j \) evaluates \( \text{opA}_1 \text{A}_2 \) and if all paths from this evaluation to point \( p_n \) contain no subsequent definitions of \( \text{A}_1 \) or \( \text{A}_2 \).
Generated and Killed Expressions

Two additional sets will need to be constructed. The first set is the set of expressions generated by each block $B_k$ where each member is an expression $\text{op}A_1A_2$ that is evaluated in $B_k$ and where $A_1$ and $A_2$ are not redefined in any statement that follows this evaluation in the block $B_k$. The second set is the set of expressions that are killed by a block $B_k$ where each member is an expression $\text{op}A_1A_2$ in which either $A_1$ or $A_2$ is redefined within the block $B_k$ and $\text{op}A_1A_2$ is not recomputed in any statement following the redefinition within the block $B_k$.

Generated expressions denoted by $\text{exGen}[B_k]$ can be represented by a bit vector and maintained in the appropriate block record in the flow graph. These expressions can be found by inspecting the opcode of each quadruple in every block $B_k$ against a set of quadruple opcodes that are known to make assignments to declared or temporary variables. Each time a distinct expression is found it is added to $\text{exGen}[B_k]$. If the same expression is encountered more than once in a block then only the last occurrence of the expression in the block is considered the generated expression and the previous occurrence(s) of the same expression must be removed from $\text{exGen}[B_k]$. Hennessy and Tjiang use a separate field in the quadruple array and assign an expression number to each unique
expression. [Hen92] An expression is not considered to be generated by $B_k$ unless it is verified that no subsequent assignments to the operands of the expression are made in block $B_k$.

This information can be retrieved provided that the set of generated definitions of block $B_k$, $dfGen[B_k]$, has been computed. Then it is possible to determine that a given operand does not have an assignment that has a quadruple number greater than the quadruple number of the expression under consideration. Using this method it is necessary to inspect each quadruple in every block yet another time having previously visited each quadruple during the construction of the flow graph and then again in building the predecessor set for each block.

This repetition can be avoided however if whenever a definition of symbol $x$ is encountered in a block, the operands of generated expressions found thus far are inspected in order to eliminate any expressions that use $x$ as an operand in the given block. The intersection of the set of quadruples that use symbol $x$ in the program (maintained in the symbol table) and the generated expressions of $B_k$ found thus far will yield the required set. That is, for any symbol $x$ at point $p$ that is defined in a block $B_k$, 

45
\[ \text{exGen}_p[B_k] = \text{exGen}_{p-1}[B_k] - (\text{Uses}[x] \cap \text{exGen}_{p-1}[B_k]) \]

where \( x \) is defined in \( B_k \) at point \( p \)

followed by adding the new expression of \( p \) as shown below.

\[ \text{exGen}_p[B_k] = \{ p \} \cup \text{exGen}_p[B_k] \]

To illustrate, Block 1 from the previous example is reviewed. The quadruple code is listed below.

Block 1 \( \rightarrow \quad q1: \ T_1 := Y \* Z \quad \) (leader)
\[ q2: \ X := T_1 \]
\[ q3: \ I := 0 \]
\[ q4: \ J := 1 \]

The set of generated expressions for Block 1 is initially empty. After inspecting \( q1 \), the following information is acquired.

\[ \text{exGen}_{q1}[B_1] = \{ q1 \} \]
\[ \text{Uses}[A] = \emptyset \]
\[ \text{Uses}[B] = \emptyset \]
\[ \text{Uses}[X] = \emptyset \]
\[ \text{Uses}[Y] = \{ q1 \} \]
\[ \text{Uses}[Z] = \{ q1 \} \]
\[ \text{Uses}[T_1] = \{ q2 \} \]
After examination of q2 it is known,

\[ \text{Uses}[X] = \emptyset \]

\[ \text{exGen}_{q2}[B_1] = \text{exGen}_{q1}[B_1] - (\text{Uses}[X] \cap \text{exGen}_{q1}[B_1]) \]
\[ = \{q1\} - (\emptyset \cap \{q1\}) \]
\[ = \{q1\} \]

Since the opcode for q2 is not an instruction that generates an expression, it is not added to \( \text{exGen}_{q2}[B_k] \).

On the other hand, suppose the block does redefine one or both of the variables as happens below at q2.

Block 1 --> q1: \( T_1 := Y \times Z \)  
               q2: \( Y := T_1 \)  
               q3: I := 0  
               q4: J := 1

After examining q1, the same information as shown in the previous example is acquired. Now after q2 however, the generated expression for q1 should be removed since q2 has redefined one of its operands. Thus

\[ \text{Uses}[Y] = \{q1\} \]

\[ \text{exGen}_{q2}[B_1] = \text{exGen}_{q1}[B_1] - (\text{Uses}[Y] \cap \text{exGen}_{q1}[B_1]) \]
\[ = \{q1\} - (\{q1\} \cap \{q1\}) \]
\[ = \emptyset \]
Again, q2 is not added to \( \text{exGen}_{q2}[B_i] \) because it does not generate an expression.

In order to build the set of expressions that are killed by a block \( B_k \), \( \text{exKill}[B_k] \) is defined to be the set of expressions in \( U_E \) that are killed in \( B_k \) where \( U_E \) is the "universal" set of all expressions appearing on the right of one or more statements of the program. The universal set can be constructed by adding each expression that is encountered during construction of the initial flow graph to this set. Recalling that expressions are killed whenever their operands are redefined later in the block the computation for \( \text{exKill}[B_k] \) can be accomplished at the same time as \( \text{exGen}[B_k] \) is computed by taking the union of all of the statements that use symbols that are defined in block \( B_k \) and then removing any generated expressions. That is,

\[
\text{exKill}[B_k] = U_E \cap (U \text{ Uses}[x]) - \text{exGen}[B_k]
\]

where \( x \) is defined in \( B_k \).

For the previous WHILE example, the computation for \( U_E \) yields the set, \( \{ q1, q5, q7, q9, q12 \} \). In order to compute \( \text{exKill}[B_i] \), the set of definitions that are generated in \( B_i \), namely \( \text{dfGen}[B_i] \), will also be needed. When the last quadruple in the given block has been examined and \( \text{exGen}[B_i] \) is known, \( \text{dfGen}[B_i] \) is used to
look at the result field to find the symbol names of all defined variables. This is accomplished by examining the result field of all quadruples in this set. For example, if dfGen[B₁] consists of \( \{ q₁, q₂, q₃, q₄ \} \), it is discovered that \( q₁ \) is the instruction that defines \( T₁ \) and hence the set of all quadruples that use \( T₁ \) will be contained in its symbol table entry field denoted by \( \text{Uses}[T₁] \). The union of all the statements that use symbols that are defined in Block 1 can now be found with the dfGen[B₁] and \( \text{Uses}[x] \) sets. Therefore, the killed expressions for Block 1 can now be computed as follows:

\[
\text{exKil}(B₁) = \bigcup \big( \big( \bigcup \text{Uses}[x] \big) - \text{exGen}(B₁) \big) \\
= \bigcup \big( \{ q₂, q₅, q₇, q₉ \} - \{ q₁ \} \big) \\
= \{ q₅, q₇, q₉ \}
\]

where \( x \) is defined in Block 1 and

\[
\bigcup \text{Uses}[x] = \text{Uses}[T₁] \cup \text{Uses}[X] \cup \text{Uses}[I] \cup \text{Uses}[J] \\
= \{ q₂ \} \cup \{ q₇ \} \cup \{ q₇ \} \cup \{ q₅, q₉ \} \\
= \{ q₂, q₅, q₇, q₉ \}
\]

Determining Available Expressions

Once the generated and killed expressions for each block have been determined, the set of available expressions, those that are available just before the beginning of block \( B_k \), denoted by \( \text{exIn}[B_k] \) and the set of
available expressions following the end of block \( B_k \), denoted by \( \text{exOut}[B_k] \) can be found by again visiting the nodes of the flow graph using iteration or the reverse post-order traversal to solve the following data flow equations. [Aho79]

\[
\begin{align*}
\text{exOut}[B_k] &= (\text{exIn}[B_k] - \text{exKill}[B_k]) \cup \text{exGen}[B_k] \\
\text{exIn}[B_k] &= \bigcap \text{exOut}[P] \text{ for } n \text{ not initial} \\
&\quad \text{where } p \text{ is a predecessor of } n \\
\text{exIn}[B_0] &= \emptyset \text{ where } B_0 \text{ is the initial node}
\end{align*}
\]

The initial values for \( \text{exIn}[B_0] \) will be the empty set and \( \text{exOut}[B_0] \) will be the set of generated expressions in the initial block named \( \text{exGen}[B_0] \) which has just been computed.

While each of the algorithms for constructing the flow graph and solving the data flow equations have been described separately some of them can be merged in order to reduce the number of times that all quadruples must be inspected or the number of times the flow graph is traversed. For example, during construction of the flow graph each quadruple is inspected to determine in which block it belongs. As this takes place, it is a simple addition to update the symbol table with the quadruple number of each symbol that is used as an operand in any
statements. Recall that this information is needed before we can determine which expressions are generated in each block.

Also, when building PrB[B_k], the set of predecessors to block B_k, since each block is represented as a linked list of quadruples, each quadruple must again be accessed in order to find the generated definitions and generated expressions. The sets for the killed definitions and killed expressions can be computed directly following this and before accessing the next block.

Eliminating Common Subexpressions

Elimination of common subexpressions involves identification of expressions which have been explicitly evaluated in separate blocks but whose operands have not changed between the evaluations. If, at the first occurrence of the common expression, the value is assigned to a temporary variable then subsequent occurrences of the same expression can be replaced by the temporary variable. The updated flow graph can be obtained by examining each expression \( x \in \text{exIn}[B_k] \). If there exists a statement within block B_k whose opcodes and operands match \( x \) then find all members \( d \in \text{dfIn}[B_k] \) where the operands and opcodes also match \( x \). After allocating a new temporary T, replace each definition \( d \)
which reaches $B_k$ and is of the form $C := \text{opA}_1A_2$ with two statements

\[
T := \text{opA}_1A_2 \\
C := T.
\]

The actual implementation of this optimization is much simpler than described. Recall that during the phase that emits the quadruple code each expression is separated by an operator and its result is allocated to a new temporary. Using the quadruple code from our previous example, notice that $q_1$ and $q_{12}$ evaluate the same expression and assign the values to temporaries $T_1$ and $T_3$ respectively.

Block 1 --> 
\begin{align*}
q1: & \quad T_1 := Y \ast Z \quad \text{(leader)} \\
q2: & \quad X := T_1 \\
q3: & \quad I := 0 \\
q4: & \quad J := 1
\end{align*}

Block 4 --> 
\begin{align*}
q5: & \quad T_2 := J \leq Y \quad \text{(leader)} \\
q6: & \quad \text{IF NOT } T_2 \text{ GOTO } q_{12}
\end{align*}

Block 3 --> 
\begin{align*}
q7: & \quad T_3 := I + X \quad \text{(leader)} \\
q8: & \quad I := T_3 \\
q9: & \quad T_4 := J + 1 \\
q10: & \quad J := T_4
\end{align*}
q11: GOTO q5

Block 2 --> q12: T s := Y * Z

q13: K := T_5

q14: . . .

With the first step in the elimination process completed, all that remains is to replace the common expression with T_1. The quadruple at q12 would now be

q12: T_5 := T_1

If the nodes of the flow graph are accessed in reverse post order sequence and replacements are performed during the traversal, then the common subexpressions will likewise be eliminated in the successor nodes. When this replacement for each definition of the appropriate form is complete, it is assured that when a common subexpression is reached, its value will now be residing in temporary T_i regardless of the path taken by the program during execution. Hence, it can be replaced by the temporary variable T_i. When judging the efficiency of introducing such seemingly useless copy statements such as T_5 := T_1, as mentioned earlier, another algorithm to eliminate unnecessary copy statements can be applied to the flow graph after this process.
CHAPTER 5 -- INTERPROCEDURAL DATA FLOW

The algorithms that have been implemented thus far have dealt exclusively with examples that contain only one main procedure where all variables are global. The source code is available in Packard Laboratory Room 235 at Lehigh University. Extensions to these algorithms could allow for procedure calls with parameters.

Recognizing Procedures

A list of procedures can be maintained in the flow graph by marking each block which is a procedure's entry point. The exit point can be recognized during traversal of the basic blocks of the procedure by the return opcode. In addition, the level number for each quadruple is saved in each entry of the code array during its construction to indicate the level of nesting of the procedure declarations themselves.

Procedure Calls

One method for interprocedural analysis is straightforward. Procedure calls can be modelled with a branch to get to the procedure, a labeled branch to return and assignments to model parameter passing. [Cha90] The implementation chosen in this program is similar to the Interprocedural Control Flow
Graph (ICFG) described by Landi and Ryder. [Lan92] In this method, the call nodes are connected to the entry nodes of procedures they invoke and exit nodes are connected to return nodes corresponding to these calls.

The instance of procedure calls within a block in the flow graph gives rise to the question of which variables might be changed by the procedures. One conservative approach is to kill all definitions when a procedure call is encountered, but this is unnecessarily drastic since only two situations exist that could cause a variable to be redefined. One is a definition of a global variable within a procedure. The other is a definition of a variable within the procedure which has an alias relationship with another variable outside the procedure. This alias relationship is established by means of a parameter passed by reference.

Definitions Generated by Procedures

The initial step in ascertaining aliases for variables entails building a set dfProc[P], which is the set of formal parameters and globals having explicit definitions within procedure P. This does not include definitions that result from a procedure call from within P.
Actual and Formal Parameters

Once the procedure definition set \( \text{dfProc}[P] \) is known, each correspondence between formal and actual parameters can be determined using the \texttt{varparam} quadruples before each call. It is not necessary to be concerned about value parameters as they are strictly local to the procedure when called and cannot be referenced by any blocks with lower level numbers. The set of variables that might be changed consists of:

1) any global variable or formal parameter which is defined within a given procedure \( P \) or
2) any global or formal parameter of \( P \) that is an actual parameter of a procedure call from procedure \( P \) or
3) any variable that is global to a procedure that is called from \( P \).

For the call by reference parameter, it is possible that it is not redefined in any call in which it is aliased. However the possibility of nested procedure calls dictate that the flow of control for each series of procedure calls be examined thoroughly in order to build a complete list of actual and formal parameter correspondences.
The Calling Graph

In determining the set of variables changed or possibly changed by a procedure a calling graph can be used. In this directed graph, each node is a procedure and an edge from A to B exists if A calls B. A collection of procedures that are not mutually exclusive will have an acyclic calling graph. [Aho79] The order in which the nodes of this graph should be visited in the determination of changed variables is reverse depth-first ordering since this will permit the changed variables for a procedure’s successors to be determined first. This information about called procedures is needed since a called procedure may change globals and formal parameters of the procedure from which it is called. For any procedure that does not call any other procedure, the set of changed variables consists of the set of global and formal parameters that exist within the blocks of the procedure itself.

So that implementation of the aliasing algorithm can proceed, each formal parameter must be given a unique entry in the symbol table during the syntax and scope analysis. This was already performed during the preliminary program of phase 1.
Common Expression in Multiple Procedures

Given a procedure $P$ we say that the expression $\text{opA}_1A_2$ is generated by a call to $P$ if and only if

1) $\text{opA}_1A_2$ occurs on every path from the initial node of procedure $P$ and,

2) $A_1$ and $A_2$ are not defined on any path after the expression has been evaluated. [Aho79] In the case where an expression $\text{opB}_1B_2$ is encountered, we cannot consider it to be a generated expression of $\text{opA}_1A_2$ unless we can verify that in every call of $P$, $B_1$ is an alias of $A_1$ and $B_2$ is an alias of $A_2$. If procedure $P$ is called only once then this determination can be made easily. In the case of multiple non-recursive calls to $P$ the problem is more complex. Recursive procedures have not been considered here.

Determining expressions that are killed by subsequent definitions of the operands requires that the following situations be recognized as killing the expression $\text{opA}_1A_2$ by a subsequent call to procedure $P$.

1) Either $A_1$ or $A_2$ is a member of the set of variables that are changed by $P$ and $A_1$ or $A_2$ are global to $P$ or,

2) either $A_1$ or $A_2$ are actual parameters of $P$ and the corresponding formal parameter is variable. An
alias is established provided that the alias named is also a member of the set of variables changed by a call to P, namely dfProc[P].

It should be noted that the existence of a formal parameter passed by reference does not imply that any redefinition of it occurs within the procedure. Also, the calling graph can be used to locate additional aliases of $A_1$ and $A_2$ in instances where P itself calls one or more procedures in which either alias $A_1$ or $A_2$ may be actual parameters in that call.

ADDITIONAL WORK

Implementation of the interprocedural data flow analysis has not been attempted except as noted. The preliminary fields for forming the calling graph are currently available in the structure for the global flow graph and have been updated accordingly during the flow graph construction. Additional work is needed in the construction of the set of definitions generated by each procedure and the identification and representation of alias relationships. Actual construction of the calling graph is necessary before it can be determined which expressions are generated and killed by each procedure. Once the above
structures have been formed, the identification and elimination of expressions common to multiple procedures will be possible.
BIBLIOGRAPHY

[Aho79]

[Bar86]

[Cha90]

[Han82]

[Hen92]

[Lan92]

APPENDIX A - THE EDISON SUBSET

Program: [Init-Declar]* Complete-Proc

Init-Declar: Const-Declar-List | Type-Declar

Complete-Proc: Proc-Head Proc-Body

Proc-Head: 'proc' Proc-Name [ '()' Param-List ']' [ '::' Type-Name]

Proc-Body: [Declaration]* 'begin' Stmt-List 'end'

Declaration: Const-Declar-List | Type-Declar | Var-Declar-List | Proc-Declar

Stmt-List: Stmt [ ';' Stmt ]*

Stmt: 'skip' | Assign-Stmt | Proc-Call | If-Stmt | While-Stmt

Const-Declar-List: 'const' Const-Declar [ ';' Const-Declar]*

Const-Declar: Constant-Name '=' Const-Symbol

Type-Declar: Array-Type

Array-Type: 'array' Type-Name [ Range-Sym ] '()' Element-Type ')

Range-Sym: Const-Symbol '::' Const-Symbol

Element-Type: Type-Name

Constant-Name: Name

Const-Symbol: Numeral | Char-Sym | Name

Char-Sym: Graphic | Control-Sym

Graphic: "graphic character"

Control-Sym: 'char' '(' Numeral ')'}

Proc-Name: Name
Name: Letter [ Letter | Digit | '_' ]*

Numeral: Digit [ Digit ]*

Param-List: Param-Group [ ';' Param-Group ]*

Param-Group: [ 'var' ] Var-Group

Var-Group: Var-Name [ ',' Var-Name ]* ':' Type-Name

Var-Name: Name

Type-Name: Name

Var-Declar-List: 'var' Var-List

Var-List: Var-Group [ ';' Var-Group ]*

Proc-Declar: Complet-Proc | Pre-Proc | Post-Proc

Pre-Proc: 'pre' Proc-Head

Post-Proc: 'post' Complet-Proc

Assign-Stmt: Var-Sym ':=' Expression

Proc-Call: Proc-Name [ '(' Arg-List ')' ]

If-Stmt: 'if' Condit-Stmt-List 'end'

While-Stmt: 'while' Condit-Stmt-List 'end'

Condit-Stmt-List: Condit-Stmt [ 'else' Condit-Stmt ]*

Condit-Stmt: Expression 'do' Stmt-List

Arg-List: Argument [ ',' Argument ]*

Argument: Expression | Var-Sym

Var-Sym: Var-Name | Function-Var | Var-Sym-Select

Var-Name: Name

Function-Var: 'val' Proc-Name

Var-Sym-Select: Index-Select

Index-Select: [ Expression ]
Expression: Simple-Expr [ Relat-Op Simple-Expr ]

Relat-Op: '=' | '>' | '<' | '<>' | '<=' | '>='

Simple-Expr: Signed-Term [ Add-Op Term ]*

Add-Op: '-' | 'or' | '+'

Signed-Term: [ Sign-Op ] Term

Sign-Op: '-' | '+'

Term: Factor [ Mult-Op Factor ]*

Mult-Op: 'and' | '*' | 'div' | 'mod'

Factor: Const-Factor | Var-Factor | Constructor
| Function-Call | '('<' Expression ')')'
| 'not' Factor | Factor Type-Transfer

Const-Factor: Const-Symbol

Var-Factor: Var-Sym

Constructor: Array-Constructor

Array-Constructor: Type-Name '('<' Elem-Expr-List ')')'

Elem-Expr-List: Expression [ ',' Expression ]*

Function-Call: Proc-Call

Type-Transfer: ':' Type-Name
APPENDIX B - QUADRUPLE STRUCTURE

Quadruple Instruction Format

qop: opcode

arg1, arg2: symbol table values or quadruple numbers depending on opcode

result: symbol table value or quadruple number depending on opcode

qlevel: number indicating nesting of procedure declarations

nxt_quad: quadruple number of next quadruple in flow graph

nxt_blok: block number of next block in flow graph

Opcode Classification

Class 1: arithmetic operations that use arg1 and arg2 as operands and store value of expression in result

format: op arg1 arg2 result

where op = add, and, divide, indxassn, indxoper, modulo, multiply, or, subtract

arg1, arg2 and result contain symbol table values

Class 2: assignment of the value of arg1 to result

format: op arg1 result

where op = assign, minus, not

arg1 and result contain symbol table values
Class 3: statements that end blocks and do not alter the result field

format 1: \[ \text{op arg1 result} \]
where \( \text{op} = \text{eqfalse, proccall} \)

format 2: \[ \text{op result} \]
where \( \text{op} = \text{goto} \)

format 3: \[ \text{op} \]
where \( \text{op} = \text{endofde, return} \)

arg1 and result contain quadruple numbers

Class 4: statements used to establish relationship between actual and formal parameters

format: \[ \text{op arg1} \]
where \( \text{op} = \text{valparam, varparam} \)

arg1 contains a symbol table value
### APPENDIX C - NXT_QUAD and NXT-BLOK USAGE

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Class</th>
<th>Opcode</th>
<th>Nxt_quad</th>
<th>Nxt_blok</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 or 2</td>
<td>--</td>
<td>ptr to next quadruple*</td>
<td>nil*</td>
<td></td>
</tr>
<tr>
<td>1 or 2</td>
<td>--</td>
<td>nil**</td>
<td>ptr to next block**</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>eqfalse</td>
<td>ptr to TRUE block</td>
<td>ptr to FALSE block</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>goto</td>
<td>nil</td>
<td>ptr to target block†</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>proccall</td>
<td>ptr to RETURN block</td>
<td>ptr to CALL block</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>endcode</td>
<td>return</td>
<td>nil</td>
<td>nil</td>
</tr>
</tbody>
</table>

* quadruple does not end the block

** quadruple ends the block

† quadruple number for the block leader is stored in the result field
The author, Janet M. Laubenstein (Noonan), was born on January 26, 1954 in Peterborough, Ontario, Canada. She is the youngest daughter of A. Arlene Noonan (Montgomery) and John J. Noonan.

Janet's father was employed by General Electric as a mechanical engineer and job transfers precipitated family moves to Toronto, Ontario in 1965 and Leominster, Massachusetts in 1970. Upon graduation from Leominster High School in 1971, Janet attended Fitchburg State College in Fitchburg, Massachusetts. In 1974 she transferred to the University of Waterloo in Waterloo, Ontario where she majored in mathematics with a concentration in computer science and graduated in 1976.

Following graduation, Janet accepted a position as a programmer with Victaulic Company of Canada in Rexdale, Ontario. In 1978, she was promoted to Supervisor of Data Processing and in 1980 was transferred to Victaulic Company of America in Easton, Pennsylvania. She held the position of MIS Coordinator until 1983 when she resigned upon the birth of her first child.
Janet began teaching at Northampton Community College in 1984 and is currently Assistant Professor of Computer Information Science. She resides in Bethlehem with her husband, James P. Laubenstein (M.B.A. Lehigh 1971) and their two sons, Matt and Bryce.
END
OF
TITLE