Code Optimization

Mrs. Sunita M Dol

(sunitaaher@gmail.com)

Assistant Professor, Dept of Computer Science and Engineering

Walchand Institute of Technology, Solapur
(www.witsolapur.org)

Code Optimization

- Introduction
- Principle Sources of optimization
- Optimization of basic blocks
- Loops in flow graphs
- Peephole Optimization
- Global Data flow analysis
- Code improving transformation

Introduction

- The code produced by the straight forward compiling algorithms can often be made to run faster or take less space, or both. This improvement is achieved by program transformations that are traditionally called optimizations. Compilers that apply code-improving transformations are called optimizing compilers.

- Optimizations are classified into two categories. They are
  o Machine independent optimizations:
  o Machine dependant optimizations:

Introduction

- **Machine independent optimizations:**
  - Machine independent optimizations are program transformations that improve the target code without taking into consideration any properties of the target machine.

- **Machine dependant optimizations:**
  - Machine dependant optimizations are based on register allocation and utilization of special machine-instruction sequences.

Introduction

- **The criteria for code improvement transformations:**
  - Simply stated, the best program transformations are those that yield the most benefit for the least effort.
  - The transformation must preserve the meaning of programs. That is, the optimization must not change the output produced by a program for a given input, or cause an error such as division by zero, that was not present in the original source program. At all times we take the "safe" approach of missing an opportunity to apply a transformation rather than risk changing what the program does.
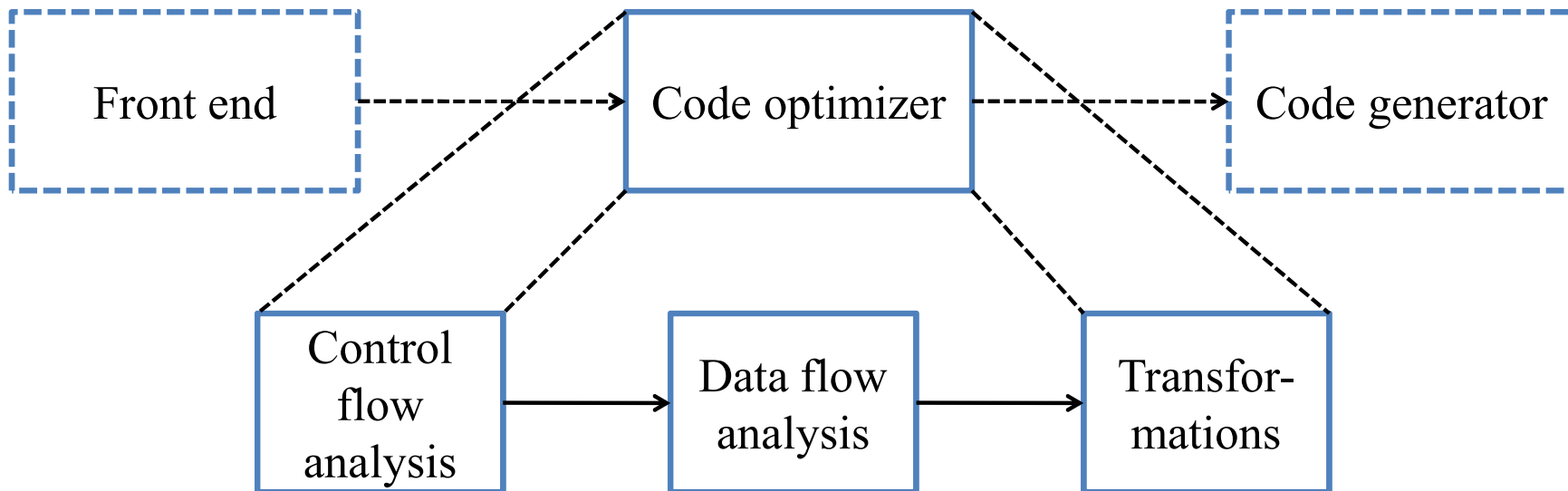
Introduction

- **The criteria for code improvement transformations:**
  o A transformation must, on the average, speed up programs by a measurable amount. We are also interested in reducing the size of the compiled code although the size of the code has less importance than it once had. Not every transformation succeeds in improving every program, occasionally an "optimization" may slow down a program slightly.
  o The transformation must be worth the effort. It does not make sense for a compiler writer to expend the intellectual effort to implement a code improving transformation and to have the compiler expend the additional time compiling source programs if this effort is not repaid when the target programs are executed. "Peephole" transformations of this kind are simple enough and beneficial enough to be included in any compiler.

Introduction

- **Organization for an Optimizing Compiler:**

Introduction

- **Organization for an Optimizing Compiler:**
    - Code improvement phase consist of control flow and data flow analysis followed by the application of transformations.
    - Control flow analysis (CFA) represents flow of control usually in form of graphs
    - Data flow analysis (DFA) is the process of ascerting and collecting information prior to program execution about the possible modification, preservation, and use of certain entities (such as values or attributes of variables) in a computer program.

Introduction

- **Organization for an Optimizing Compiler:**
  - o Organization for an Optimizing Compiler has the following advantages
    - ✓ The operation needed to implement high-level constructs are made explicit in the intermediate code so I is possible to optimize them.
    - ✓ The intermediate code can be independent of the target machine so that the optimizer does not have to change much if the code generator is replaced by one for a different machine.

  - o In code optimizer, programs are represented by flow graphs in which edges indicate the flow of control and nodes represents basic blocks.

The Principle Sources of Optimization

- A transformation of a program is called local if it can be performed by looking only at the statements in a basic block; otherwise, it is called global.
- Many transformations can be performed at both the local and global levels. Local transformations are usually performed first.

The Principle Sources of Optimization

- **Function-Preserving Transformations**
  - o **Common Sub expressions elimination**
  - o **Copy Propagation**
  - o **Dead-Code Eliminations**
  - o **Constant folding**
- **Loop Optimizations**
  - o **Code Motion**
  - o **Induction Variables**
  - o **Reduction In Strength**

Sunita M Dol

The Principle Sources of Optimization

- **Function-Preserving Transformations**
  - o There are a number of ways in which a compiler can improve a program without changing the function it computes.
  - o The transformations
    - ✓ **Common Sub expressions elimination**
    - ✓ **Copy Propagation**
    - ✓ **Dead-Code Eliminations**
    - ✓ **Constant folding**

    are common examples of such function-preserving transformations. The other transformations come up primarily when global optimizations are performed.
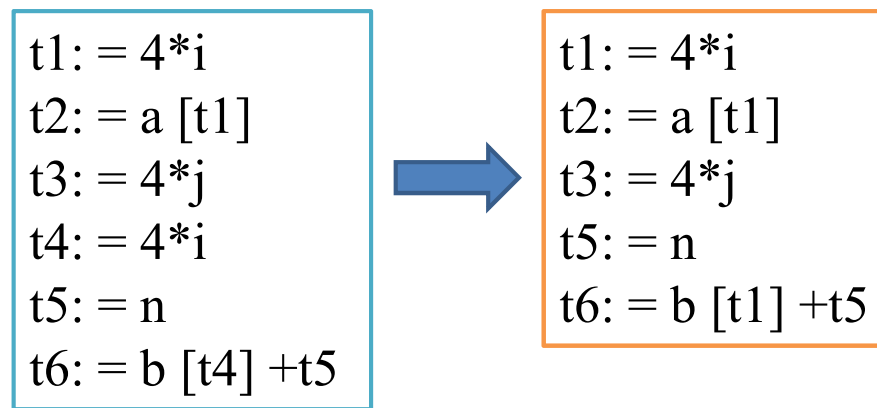
The Principle Sources of Optimization

- **Function-Preserving Transformations**
  - **Common Sub expressions elimination**
    - ✓ An occurrence of an expression E is called a common sub-expression if
      - ▪ E was previously computed, and
      - ▪ The values of variables in E have not changed since the previous computation.
    - ✓ We can avoid recomputing the expression if we can use the previously computed value.

The Principle Sources of Optimization

- **Function-Preserving Transformations**
 - **Common Sub expressions elimination**

    Example

    ```
    t1: = 4*i
    t2: = a [t1]
    t3: = 4*j
    t4: = 4*i
    t5: = n
    t6: = b [t4] +t5
    ```
    →
    ```
    t1: = 4*i
    t2: = a [t1]
    t3: = 4*j
    t5: = n
    t6: = b [t1] +t5
    ```

    The common sub expression t4: =4*i is eliminated as its computation is already in t1. And value of i is not been changed from definition to use.

The Principle Sources of Optimization

- **Function-Preserving Transformations**
  - **Copy Propagation**
    - ✓ Assignments of the form f : = g called copy statements, or copies for short. The idea behind the copy-propagation transformation is to use g for f, whenever possible after the copy statement f: = g.
    - ✓ Copy propagation means use of one variable instead of another.
    - ✓ Example

x := t3
a[t2]:=t5
a[t4]:= GOTO B2
a[t4]:=x

      →

x := t3
a[t2]:=t5
a[t4]:= GOTO B2
a[t4]:=t3
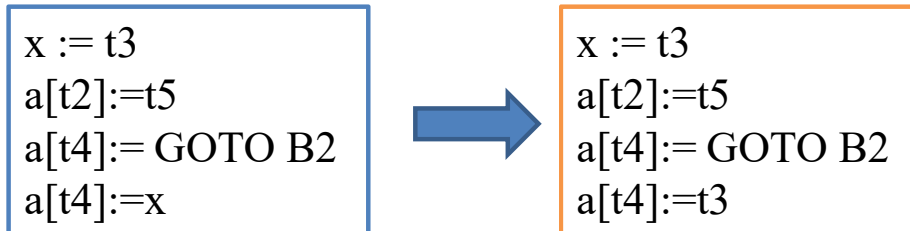
    - ✓ No obvious improvement but enables other transformation

The Principle Sources of Optimization

- **Function-Preserving Transformations**
 - **Dead-Code Eliminations**
    - ✓ A variable is live at a point in a program if its value can be used subsequently; otherwise, it is dead at that point.
    - ✓ A related idea is dead or useless code, statements that compute values that never get used.
    - ✓ While the programmer is unlikely to introduce any dead code intentionally, it may appear as the result of previous transformations.
    - ✓ An optimization can be done by eliminating dead code.

The Principle Sources of Optimization

- **Function-Preserving Transformations**
  - **Dead-Code Eliminations**
    - Example

x := t3
a[t2]:=t5
a[t4]:=x
GOTO B2

➡

~~x := t3~~
a[t2]:=t5
a[t4]:=t3
GOTO B2
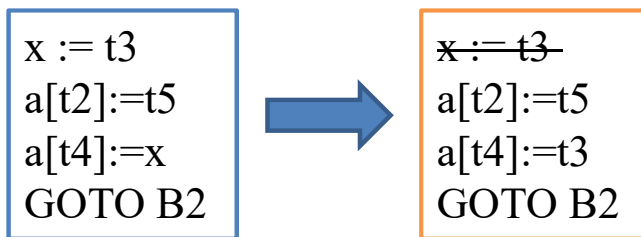
The Principle Sources of Optimization

- **Function-Preserving Transformations**
  - **Constant folding**
    - ✓ We can eliminate both the test and printing from the object code. More generally, deducing at compile time that the value of an expression is a constant and using the constant instead is known as constant folding.
    - ✓ One advantage of copy propagation is that it often turns the copy statement into dead code.
    - ✓ For example,
      a=3.14157/2 can be replaced by
      a=1.570 there by eliminating a division operation.

The Principle Sources of Optimization

- **Loop Optimizations**
  - We now give a brief introduction to a very important place for optimizations, namely loops, especially the inner loops where programs tend to spend the bulk of their time.
  - The running time of a program may be improved if we decrease the number of instructions in an inner loop, even if we increase the amount of code outside that loop.
  - Three techniques are important for loop optimization:
    - ✓ **Code Motion**
    - ✓ **Induction Variables**
    - ✓ **Reduction In Strength**

The Principle Sources of Optimization

- **Loop Optimizations**
  - **Code Motion**
    - ✓ An important modification that decreases the amount of code in a loop is code motion. This transformation takes an expression that yields the same result independent of the number of times a loop is executed and places the expression before the loop.
    - ✓ The notion "before the loop" assumes the existence of an entry for the loop. For example, evaluation of limit-2 is a loop-invariant computation in the following while-statement:
    - ✓ Code motion will result in the equivalent of

while (i <= limit-2)
/* statement does not change limit*/
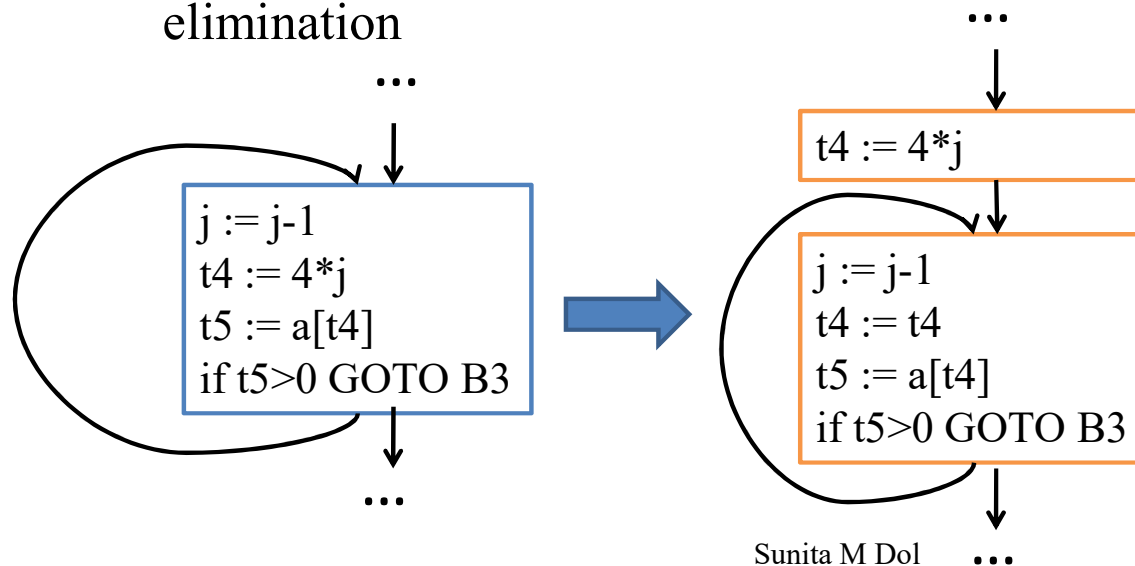
→

t= limit-2;
while (i<=t)
/* statement does not change limit or t */

The Principle Sources of Optimization

- **Loop Optimizations**
  - **Induction Variables**
    - ✓ Loops are usually processed inside out
    - ✓ When there are two or more induction variables in a loop, it may be possible to get rid of all but one, by the process of induction-variable elimination

The Principle Sources of Optimization

- **Loop Optimizations**

  o **Reduction In Strength**

    ✓ Reduction in strength replaces expensive operations by equivalent cheaper ones on the target machine. Certain machine instructions are considerably cheaper than others and can often be used as special cases of more expensive operators.

    ✓ For example, x^2 is invariably cheaper to implement as x*x than as a call to an exponentiation routine. Fixed-point multiplication or division by a power of two is cheaper to implement as a shift. Floating-point division by a constant can be implemented as multiplication by a constant, which may be cheaper.
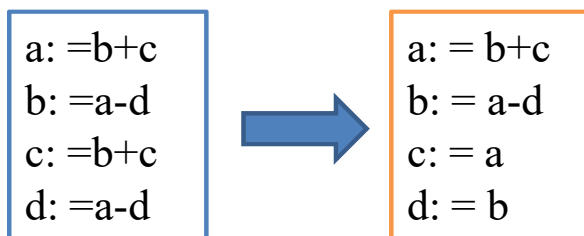
Optimization of Basic Blocks

- There are two types of basic block optimizations. They are :
  o Structure-Preserving Transformations
  o Algebraic Transformations

- **Structure-Preserving Transformations:**
  o The primary Structure-Preserving Transformation on basic blocks are:
    ✓ **Common sub-expression elimination**
    ✓ **Dead code elimination**
    ✓ **Renaming of temporary variables**
    ✓ **Interchange of two independent adjacent statements.**

Optimization of Basic Blocks

- **Structure-Preserving Transformations:**
  - **Common sub-expression elimination:**
    - ✓ Common sub expressions need not be computed over and over again. Instead they can be computed once and kept in store from where it's referenced when encountered again – of course providing the variable values in the expression still remain constant.

a: =b+c
b: =a-d
c: =b+c
d: =a-d

➡

a: = b+c
b: = a-d
c: = a
d: = b

Optimization of Basic Blocks

- **Structure-Preserving Transformations:**
  - o **Dead code elimination**
    - ✓ It's possible that a large amount of dead (useless) code may exist in the program. This might be especially caused when introducing variables and procedures as part of construction or error-correction of a program – once declared and defined, one forgets to remove them in case they serve no purpose. Eliminating these will definitely optimize the code.

  - o **Renaming of temporary variables**
    - ✓ A statement t:=b+c where t is a temporary name can be changed to u:=b+c where u is another temporary name, and change all uses of t to u.
    - ✓ In this we can transform a basic block to its equivalent block called normal-form block.

Optimization of Basic Blocks

- **Structure-Preserving Transformations:**

  o **Interchange of two independent adjacent statements**

  Two statements

      t1:=b+c

      t2:=x+y

  can be interchanged or reordered in its computation in the basic block when value of t1 does not affect the value of t2.

Optimization of Basic Blocks

- **Algebraic Transformations**
  o Algebraic identities represent another important class of optimizations on basic blocks. This includes simplifying expressions or replacing expensive operation by cheaper ones i.e. reduction in strength.
  o Another class of related optimizations is constant folding. Here we evaluate constant expressions at compile time and replace the constant expressions by their values. Thus the expression 2*3.14 would be replaced by 6.28.
  o The relational operators <=, >=, <, >, + and = sometimes generate unexpected common sub expressions.

Optimization of Basic Blocks

- **Algebraic Transformations**

  o Associative laws may also be applied to expose common sub expressions. For example, if the source code has the assignments

    a :=b+c

    e :=c+d+b

  the following intermediate code may be generated:

    a :=b+c

    t :=c+d

    e :=t+b

Optimization of Basic Blocks

- **Algebraic Transformations**
  - Example

    x:=x+0 can be removed

    x:=y**2 can be replaced by a cheaper statement x:=y*y
  - The compiler writer should examine the language carefully to determine what rearrangements of computations are permitted, since computer arithmetic does not always obey the algebraic identities of mathematics. Thus, a compiler may evaluate x*y-x*z as x*(y-z) but it may not evaluate a+(b-c) as (a+b)-c.

Loops in Flow Graphs

- A graph representation of three-address statements, called a flow graph, is useful for understanding code-generation algorithms, even if the graph is not explicitly constructed by a code-generation algorithm.
- Nodes in the flow graph represent computations, and the edges represent the flow of control.
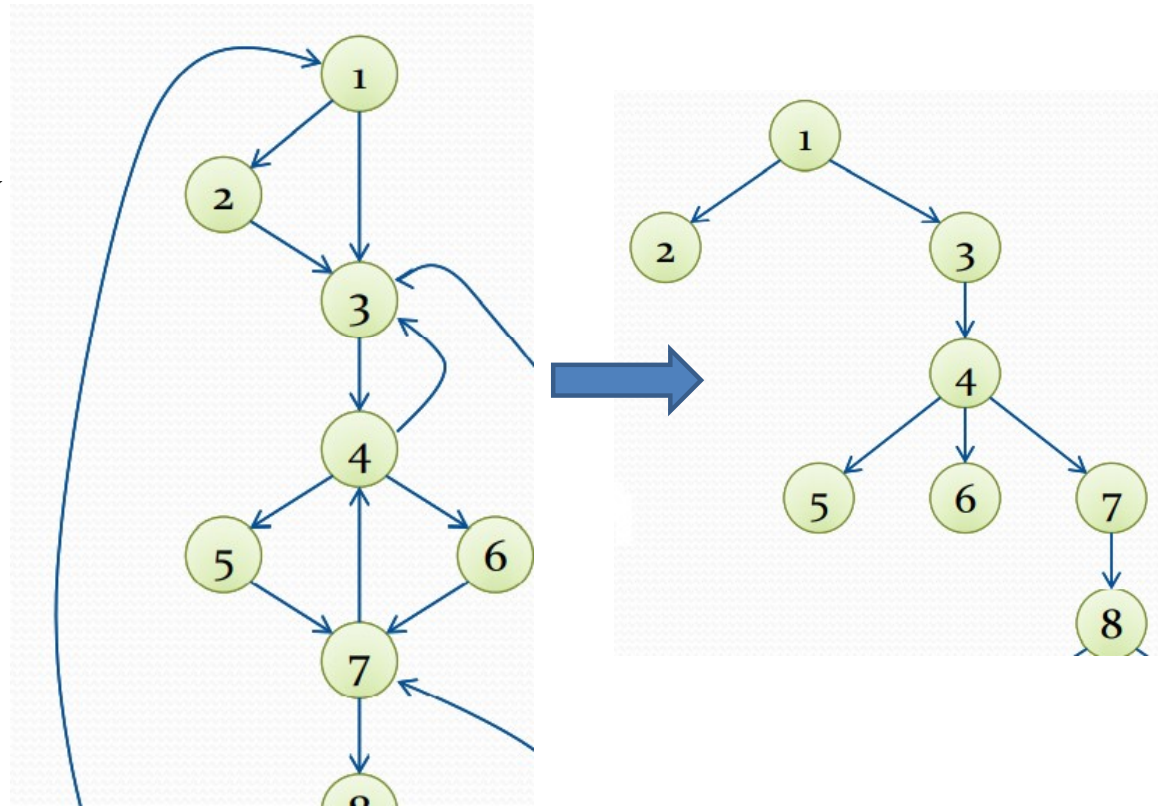
Loops in Flow Graphs

- **Dominators:**
  o d dom n
    - ✓ Node d of a CFG dominates node n if every path from the initial node of the CFG to n goes through d
    - ✓ The loop entry dominates all nodes in the loop
  o The immediate dominator m of a node n is the last dominator on the path from the initial node to n
    - ✓ If d ≠ n and d dom n then d dom m

Loops in Flow Graphs

- **Dominators:**
  - Example
  - In the flow graph below,
  - Initial node, node1 dominates every node.
  - node 2 dominates itself
  - node 3 dominates all but 1 and 2.
  - node 4 dominates all but 1,2 and 3.
  - node 5 and 6 dominates only themselves, since flow of control can skip around either by join through the other.
  - node 7 dominates 7,8 ,9 and 10.
  - node 8 dominates 8,9 and 10.
  - node 9 and 10 dominates only themselves.

Loops in Flow Graphs

- **Dominators:**
  - The way of presenting dominator information is in a tree, called the dominator tree in which the initial node is the root.
  - The parent of each other node is its immediate dominator.
  - Each node d dominates only its descendents in the tree.
  - The existence of dominator tree follows from a property of dominators; each node has a unique immediate dominator in that is the last dominator of n on any path from the initial node to n.
  - In terms of the dom relation, the immediate dominator m has the property is d=!n and d dom n, then d dom m.

Loops in Flow Graphs

- **Dominators:**
  - o Example

D(1)={1}
D(2)={1,2}
D(3)={1,3}
D(4)={1,3,4}
D(5)={1,3,4,5}
D(6)={1,3,4,6}
D(7)={1,3,4,7}
D(8)={1,3,4,7,8}
D(9)={1,3,4,7,8,9}
D(10)={1,3,4,7,8,10}

Sunita M Dol

34
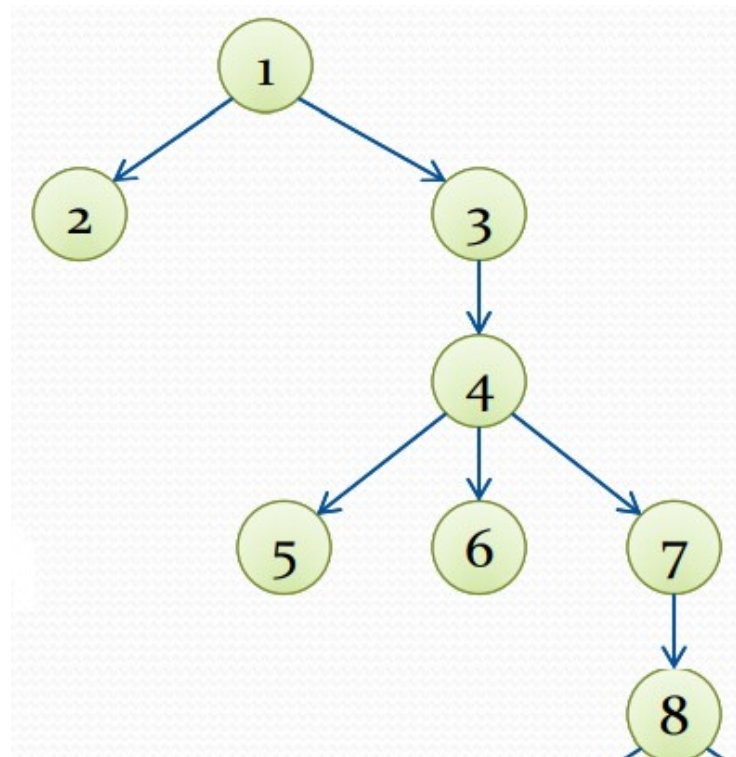
Loops in Flow Graphs

- **Natural Loop**
  - One application of dominator information is in determining the loops of a flow graph suitable for improvement.
  - The properties of loops are
    - ✓ A loop must have a single entry point, called the header. This entry point-dominates all nodes in the loop, or it would not be the sole entry to the loop.
    - ✓ There must be at least one way to iterate the loop(i.e.)at least one path back to the header.
  - One way to find all the loops in a flow graph is to search for edges in the flow graph whose heads dominate their tails. If a→b is an edge, b is the head and a is the tail. These types of edges are called as back edges.

Loops in Flow Graphs

- **Natural Loop**

  o Example

  In the graph,

  7 → 4 4 DOM 7
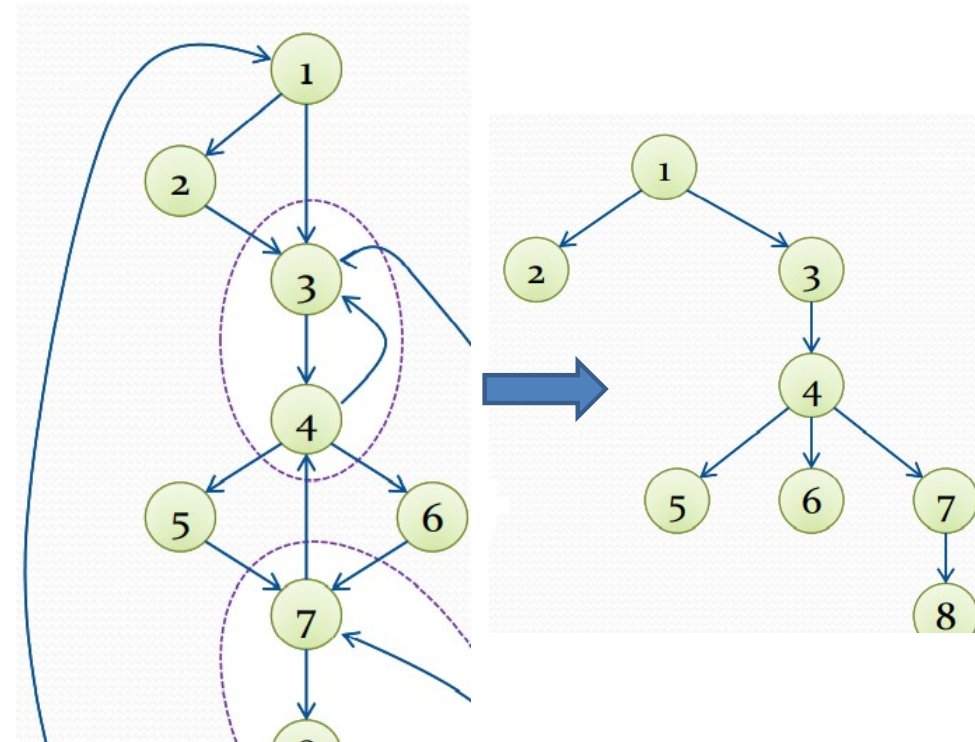
  10 →7 7 DOM 10

  4 → 3

  8 → 3

  9 →1

  The above edges will form loop in flow graph.

  o Given a back edge n → d, we define the natural loop of the edge to be d plus the set of nodes that can reach n without going through d. Node d is the header of the loop.

Loops in Flow Graphs

- **Natural Loop**

  Algorithm: Constructing the natural loop of a back edge.

  Input: A flow graph G and a back edge n→d.

  Output: The set loop consisting of all nodes in the natural loop n→d.

  Method: Beginning with node n, we consider each node m*d that we know is in loop, to make sure that m's predecessors are also placed in loop. Each node in loop, except for d, is placed once on stack, so its predecessors will be examined. Note that because d is put in the loop initially, we never examine its predecessors, and thus find only those nodes that reach n without going through d.

Loops in Flow Graphs

- **Natural Loop**

  Algorithm: Constructing the natural loop of a back edge.

  Procedure insert(m);
  if m is not in loop **then begin**
         loop := loop U {m};
         push m onto stack
  end;
  stack : = empty;
  loop : = {d};
  insert(n);
  while stack is not empty **do begin**
         pop m, the first element of stack, off stack;
         for each predecessor p of m **do** insert(p)
 **end**

Loops in Flow Graphs

- **Inner Loop**
    o If we use the natural loops as "the loops", then we have the useful property that unless two loops have the same header, they are either disjointed or one is entirely contained in the other. Thus, neglecting loops with the same header for the moment, we have a natural notion of inner loop: one that contains no other loop.
    o When two natural loops have the same header, but neither is nested within the other, they are combined and treated as a single loop.

Loops in Flow Graphs

- **Pre-headers**
  - Several transformations require us to move statements "before the header". Therefore begin treatment of a loop L by creating a new block, called the preheater.
  - The pre-header has only the header as successor, and all edges which formerly entered the header of L from outside L instead enter the pre-header.
  - Edges from inside loop L to the header are not changed.
  - Initially the pre-header is empty, but transformations on L may place statements in it.

header

loop L

pre-header

header

loop L
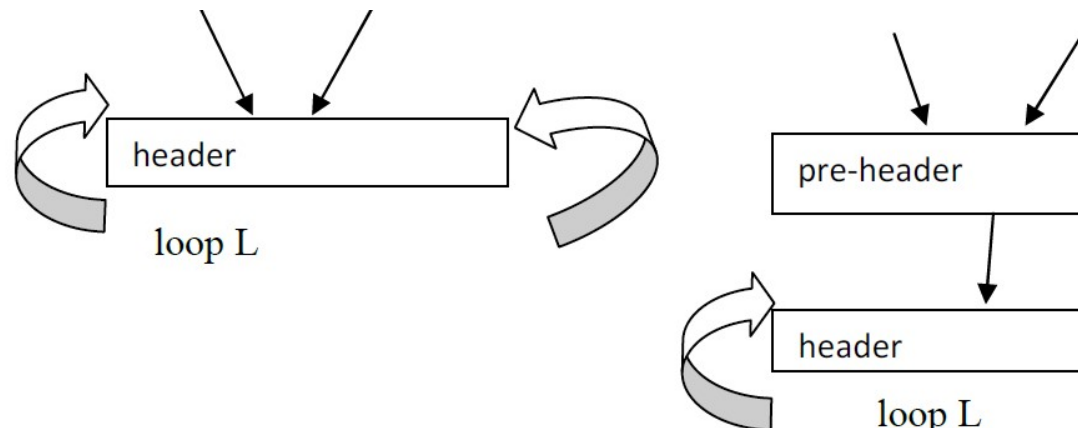
Loops in Flow Graphs

- **Reducible Flow Graph**

  o Reducible flow graphs are special flow graphs, for which several code optimization transformations are especially easy to perform, loops are unambiguously defined, dominators can be easily calculated, data flow analysis problems can also be solved efficiently.

  o Exclusive use of structured flow-of-control statements such as if-then-else, while-do, continue, and break statements produces programs whose flow graphs are always reducible.

  o The most important properties of reducible flow graphs are that there are no jumps into the middle of loops from outside; the only entry to a loop is through its header.

Loops in Flow Graphs

- **Reducible Flow Graph**

    o A flow graph G is reducible if and only if we can partition the edges into two disjoint groups, forward edges and back edges, with the following properties.

    ✓The forward edges from an acyclic graph in which every node can be reached from initial node of G.

    ✓The back edges consist only of edges where heads dominate theirs tails.

Loops in Flow Graphs

- **Reducible Flow Graph**
  - Example: The above flow graph is reducible.
    - ✓ If we know the relation DOM for a flow graph, we can find and remove all the back edges.
    - ✓ The remaining edges are forward edges.
    - ✓ If the forward edges form an acyclic graph, then we can say the flow graph reducible.
    - ✓ In the above example remove the five back edges 4→3, 7→4, 8→3, 9→1 and 10→7 whose heads dominate their tails, the remaining graph is acyclic.
    - ✓ The key property of reducible flow graphs for loop analysis is that in such flow graphs every set of nodes that we would informally regard as a loop must contain a back edge.
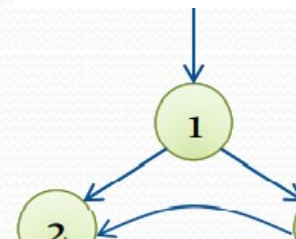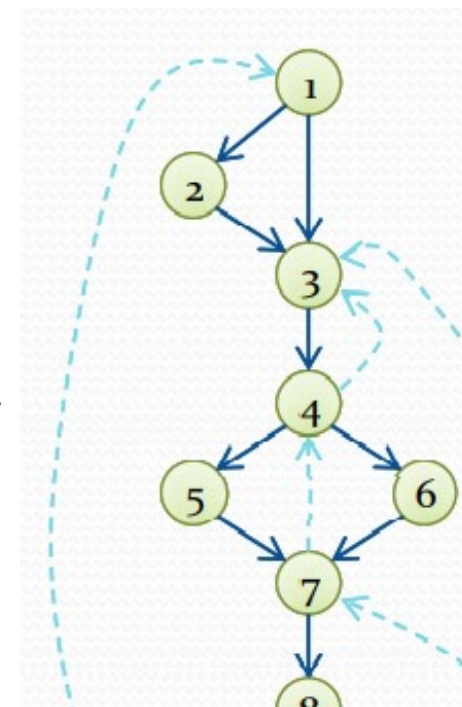
Example of a non-reducible flow graph

Peephole Optimization

- A statement-by-statement code-generations strategy often produce target code that contains redundant instructions and suboptimal constructs .The quality of such target code can be improved by applying "optimizing" transformations to the target program.

- A simple but effective technique for improving the target code is peephole optimization, a method for trying to improving the performance of the target program by examining a short sequence of target instructions (called the peephole) and replacing these instructions by a shorter or faster sequence, whenever possible.

Peephole Optimization

- The peephole is a small, moving window on the target program. The code in the peephole need not contiguous, although some implementations do require this. It is characteristic of peephole optimization that each improvement may spawn opportunities for additional improvements.

- Following are examples of program transformations that are characteristic of peephole optimizations:
  - Redundant-instructions elimination
  - Unreachable Code
  - Flow-of-control optimizations
  - Algebraic simplifications
  - Use of machine idioms

Peephole Optimization

- **Redundant-instructions elimination**

    If we see the instructions sequence

    (1) MOV R0,a

    (2) MOV a,R0

we can delete instructions (2) because whenever (2) is executed. (1) will ensure that the value of a is already in register R0.If (2) had a label we could not be sure that (1) was always executed immediately before (2) and so we could not remove (2).

Peephole Optimization

- **Unreachable Code**
  - Another opportunity for peephole optimizations is the removal of unreachable instructions. An unlabeled instruction immediately following an unconditional jump may be removed. This operation can be repeated to eliminate a sequence of instructions.
  - For example, for debugging purposes, a large program may have within it certain segments that are executed only if a variable debug is 1.
  - In C, the source code might look like:

    ```
    #define debug 0
    ….
    If ( debug ) {
    Print debugging information
    }
    ```

Peephole Optimization

- **Unreachable Code**
  - In the intermediate representations the if-statement may be translated as:

    > If debug =1 goto L2
    > goto L2
    > L1: print debugging information
    > L2: ………………………….(a)

  - One obvious peephole optimization is to eliminate jumps over jumps .Thus no matter what the value of debug; (a) can be replaced by:

    > If debug $\neq$1 goto L2
    > Print debugging information
    > L2: …………………………….(b)

Peephole Optimization

- **Unreachable Code**
 - o As the argument of the statement of (b) evaluates to a constant true it can be replaced by

    > If debug $\neq 0$ goto L2
    > Print debugging information
    > L2: ……………………………….(c)

  - o As the argument of the first statement of (c) evaluates to a constant true, it can be replaced by goto L2. Then all the statement that print debugging aids are manifestly unreachable and can be eliminated one at a time.

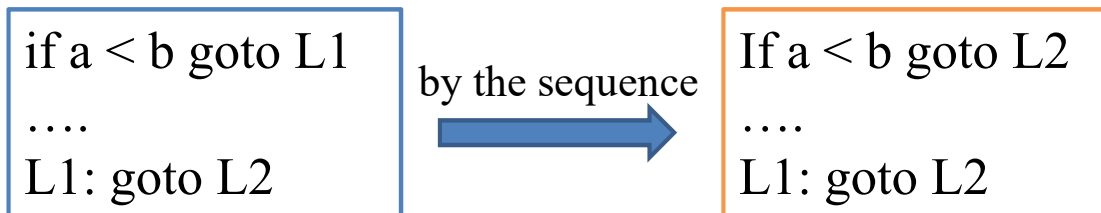# Peephole Optimization

- **Flow-of-control optimizations**
  - o The unnecessary jumps can be eliminated in either the intermediate code or the target code by the following types of peephole optimizations. We can replace the jump sequence

```
goto L1
….
L1: gotoL2
```
by the sequence
```
goto L2
….
L1: goto L2
```
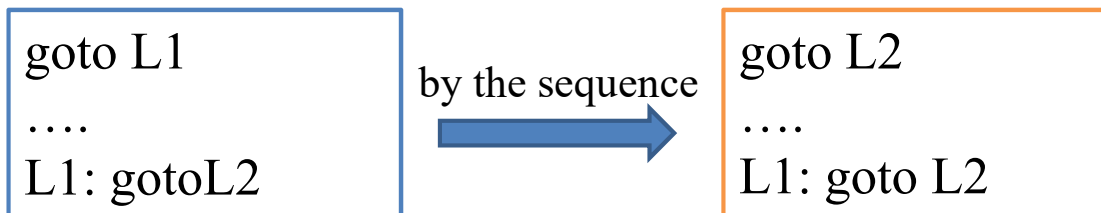
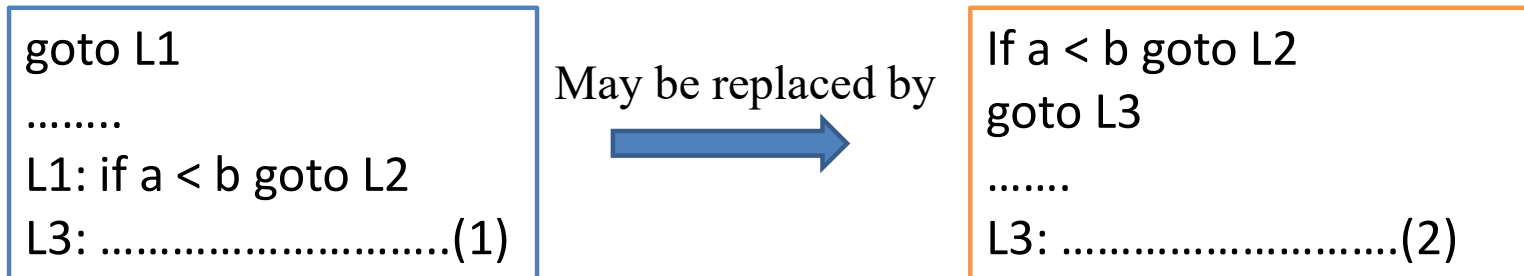# Peephole Optimization

- **Flow-of-control optimizations**
  - o If there are now no jumps to L1, then it may be possible to eliminate the statement L1:goto L2 provided it is preceded by an unconditional jump .Similarly, the sequence

if a < b goto L1	by the sequence	If a < b goto L2
….		….
L1: goto L2		L1: goto L2

Peephole Optimization

- **Flow-of-control optimizations**
  - Finally, suppose there is only one jump to L1 and L1 is preceded by an unconditional goto. Then the sequence

goto L1 …….. L1: if a < b goto L2 L3: ………………………..(1)	May be replaced by →	If a < b goto L2 goto L3 ……. L3: ………………………(2)

  - While the number of instructions in (1) and (2) is the same, we sometimes skip the unconditional jump in (2), but never in (1).Thus (2) is superior to (1) in execution time

Peephole Optimization

- **Algebraic Simplification:**
    - There is no end to the amount of algebraic simplification that can be attempted through peephole optimization. Only a few algebraic identities occur frequently enough that it is worth considering implementing them .For example, statements such as

        x := x+0

        Or

        x := x * 1

    - Are often produced by straightforward intermediate code-generation algorithms, and they can be eliminated easily through peephole optimization.

Peephole Optimization

- **Reduction in Strength**
    - Reduction in strength replaces expensive operations by equivalent cheaper ones on the target machine. Certain machine instructions are considerably cheaper than others and can often be used as special cases of more expensive operators.
    - For example, x^2 is invariably cheaper to implement as x*x than as a call to an exponentiation routine. Fixed-point multiplication or division by a power of two is cheaper to implement as a shift. Floating-point division by a constant can be implemented as multiplication by a constant, which may be cheaper.

$$X^2 \rightarrow X*X$$

Peephole Optimization

- **Use of Machine Idioms**
  - o The target machine may have hardware instructions to implement certain specific operations efficiently. For example, some machines have auto-increment and auto-decrement addressing modes. These add or subtract one from an operand before or after using its value.
  - o The use of these modes greatly improves the quality of code when pushing or popping a stack, as in parameter passing. These modes can also be used in code for statements like i :=i+1.

    i:=i+1 → i++

    i:=i-1 → i- -

Global Data-Flow Analysis

- In order to do code optimization and a good job of code generation , compiler needs to collect information about the program as a whole and to distribute this information to each block in the flow graph.

- A compiler could take advantage of "reaching definitions" , such as knowing where a variable like debug was last defined before reaching a given block, in order to perform transformations are just a few examples of data-flow information that an optimizing compiler collects by a process known as data-flow analysis.

Global Data-Flow Analysis

- Data-flow information can be collected by setting up and solving systems of equations of the form :

    out [S] = gen [S] U (in [S] – kill [S])

    This equation can be read as " the information at the end of a statement is either generated within the statement , or enters at the beginning and is not killed as control flows through the statement."

Global Data-Flow Analysis

- The details of how data-flow equations are set and solved depend on three factors.

  o The notions of generating and killing depend on the desired information, i.e., on the data flow analysis problem to be solved. Moreover, for some problems, instead of proceeding along with flow of control and defining out[s] in terms of in[s], we need to proceed backwards and define in[s] in terms of out[s].

  o Since data flows along control paths, data-flow analysis is affected by the constructs in a program. In fact, when we write out[s] we implicitly assume that there is unique end point where control leaves the statement; in general, equations are set up at the level of basic blocks rather than statements, because blocks do have unique end points.

  o There are subtleties that go along with such statements as procedure calls, assignments through pointer variables, and even assignments to array variables.
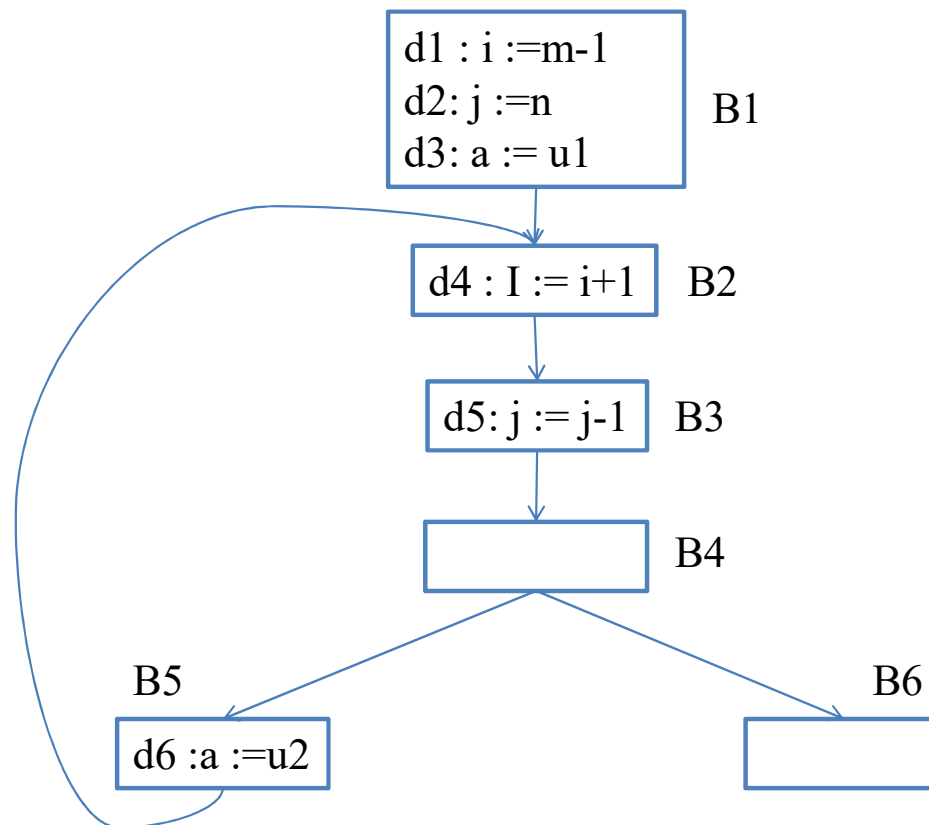
Global Data-Flow Analysis

- **Points and Paths:**
  o Within a basic block, we talk of the point between two adjacent statements, as well as the point before the first statement and after the last. Thus, block B1 has four points: one before any of the assignments and one after each of the three assignments.

Global Data-Flow Analysis

- **Points and Paths:**

d1 : i :=m-1
d2: j :=n
d3: a := u1
B1

d4 : I := i+1   B2

d5: j := j-1   B3

B4

B5

d6 :a :=u2

B6

Global Data-Flow Analysis

- **Points and Paths:**
  - o Now let us take a global view and consider all the points in all the blocks. A path from p1 to pn is a sequence of points p1, p2,….,pn such that for each i between 1 and n-1, either
    - ✓ Pi is the point immediately preceding a statement and pi+1 is the point immediately following that statement in the same block, or
    - ✓ Pi is the end of some block and pi+1 is the beginning of a successor block.

Global Data-Flow Analysis

- **Reaching definitions:**
  - A definition of variable x is a statement that assigns, or may assign, a value to x. The most common forms of definition are assignments to x and statements that read a value from an i/o device and store it in x.
  - These statements certainly define a value for x, and they are referred to as unambiguous definitions of x. There are certain kinds of statements that may define a value for x; they are called ambiguous definitions. The most usual forms of ambiguous definitions of x are:
    - ✓ A call of a procedure with x as a parameter or a procedure that can access x because x is in the scope of the procedure.
    - ✓ An assignment through a pointer that could refer to x. For example, the assignment $*q: = y$ is a definition of x if it is possible that q points to x. we must assume that an assignment through a pointer is a definition of every variable.

Global Data-Flow Analysis

- **Reaching definitions:**

    o   We say a definition d reaches a point p if there is a path from the point immediately following d to p, such that d is not "killed" along that path. Thus a point can be reached by an unambiguous definition and an ambiguous definition of the same variable appearing later along one path.

Global Data-Flow Analysis

- **Data-flow analysis of structured programs**
  - Flow graphs for control flow constructs such as do-while statements have a useful property: there is a single beginning point at which control enters and a single end point that control leaves from when execution of the statement is over. We exploit this property when we talk of the definitions reaching the beginning and the end of statements with the following syntax.
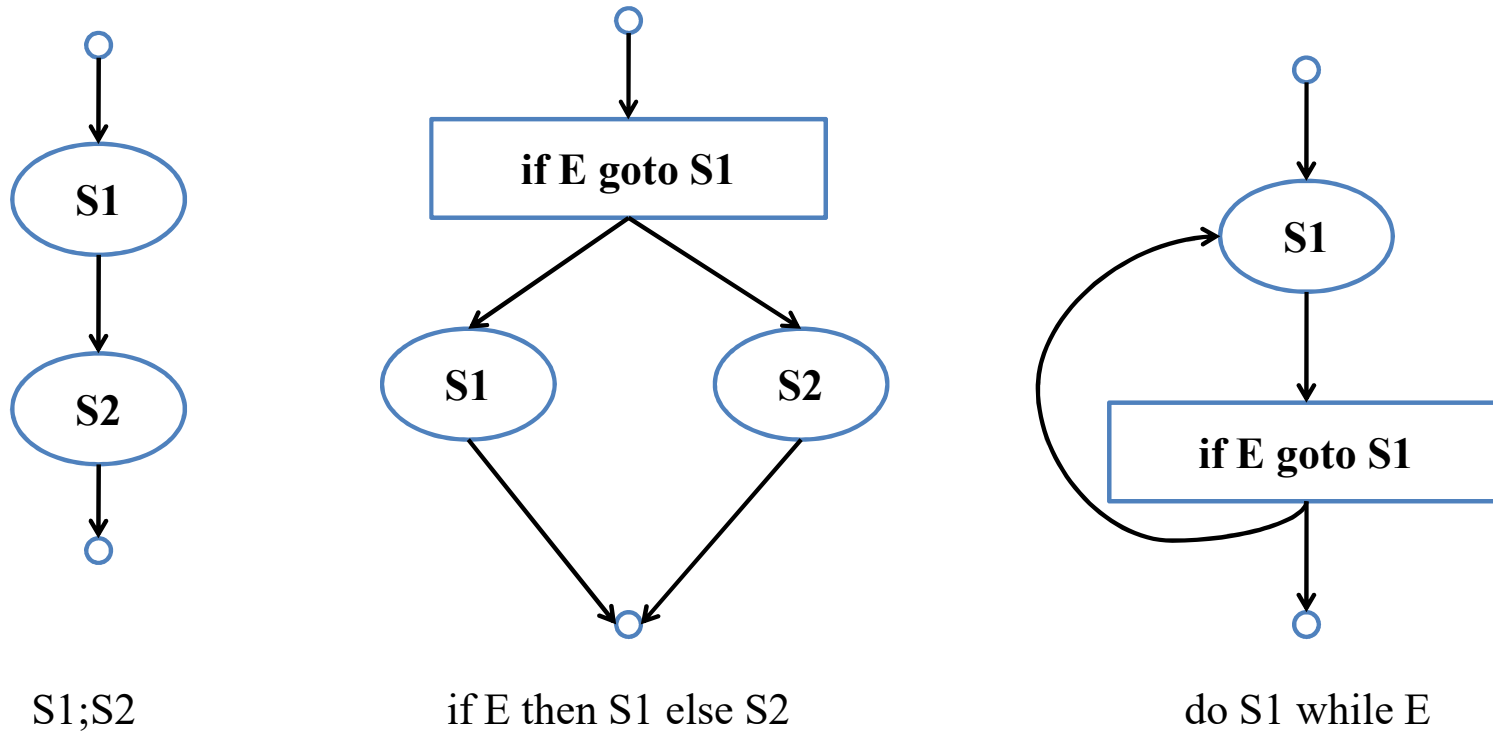
    S id: = E| S; S | if E then S else S | do S while E

    E id + id| id

  - Expressions in this language are similar to those in the intermediate code, but the flow graphs for statements have restricted forms.

Global Data-Flow Analysis

- **Data-flow analysis of structured programs**

S1;S2   if E then S1 else S2   do S1 while E

Global Data-Flow Analysis

- **Data-flow analysis of structured programs**
  - o We define a portion of a flow graph called a region to be a set of nodes N that includes a header, which dominates all other nodes in the region. All edges between nodes in N are in the region, except for some that enter the header.

  - o The portion of flow graph corresponding to a statement S is a region that obeys the further restriction that control can flow to just one outside block when it leaves the region.
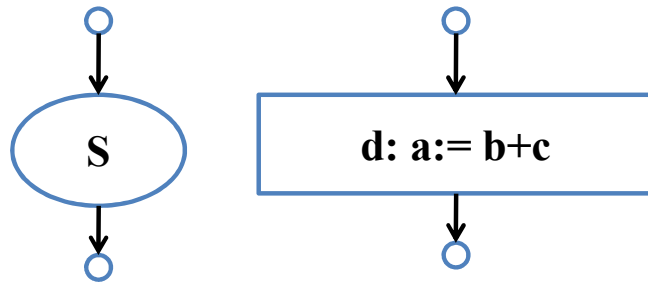
Global Data-Flow Analysis

- **Data-flow analysis of structured programs**

    o We say that the beginning points of the dummy blocks at the entry and exit of a statement's region are the beginning and end points, respectively, of the statement. The equations are inductive, or syntax-directed, definition of the sets in[S], out[S], gen[S], and kill[S] for all statements S.

    o gen[S] is the set of definitions "generated" by S while kill[S] is the set of definitions that never reach the end of S.

Global Data-Flow Analysis

- **Data-flow analysis of structured programs**

  o Consider the following data-flow equations for reaching definitions :

  $$gen\ [S] = \{\ d\ \}$$
  $$kill\ [S] = Da - \{\ d\ \}$$
  $$out\ [S] = gen\ [S]\ U\ (\ in[S] - kill[S]\ )$$

  o Observe the rules for a single assignment of variable a. Surely that assignment is a definition of a, say d. Thus

  $$Gen[S]=\{d\}$$

  o On the other hand, d "kills" all other definitions of a, so we write
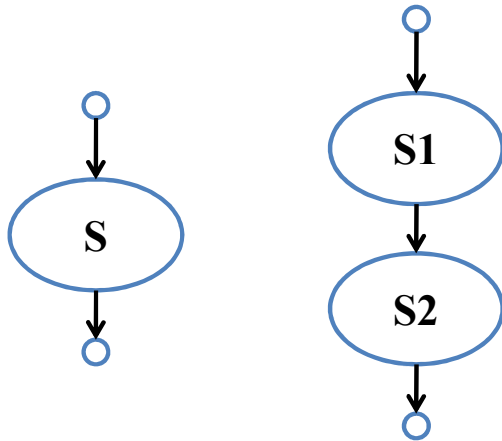
  $$Kill[S] = Da - \{d\}$$

  where, Da is the set of all definitions in the program for   variable a.

Global Data-Flow Analysis

- **Data-flow analysis of structured programs**
  - Consider the following data-flow equations for reaching definitions :

gen[S]=gen[S2] U (gen[S1]-kill[S2])
Kill[S] = kill[S2] U (kill[S1] – gen[S2])
in [S1] = in [S]
in [S2] = out [S1]
out [S] = out [S2]

Global Data-Flow Analysis

- **Data-flow analysis of structured programs**
    - o Under what circumstances is definition d generated by S=S1; S2? First of all, if it is generated by S2, then it is surely generated by S. if d is generated by S1, it will reach the end of S provided it is not killed by S2. Thus, we write

        gen[S]=gen[S2] U (gen[S1]-kill[S2])

    - o Similar reasoning applies to the killing of a definition, so we have

        Kill[S] = kill[S2] U (kill[S1] – gen[S2])

gen[S]=gen[S2] U (gen[S1
Kill[S] = kill[S2] U (kill[S
in [S1] = in [S]
in [S2] = out [S1]
out [S] = out [S2]

Global Data-Flow Analysis

- **Conservative estimation of data-flow information:**
  - There is a subtle miscalculation in the rules for gen and kill. We have made the assumption that the conditional expression E in the if and do statements are "uninterpreted"; that is, there exists inputs to the program that make their branches go either way.
  - We assume that any graph-theoretic path in the flow graph is also an execution path, i.e., a path that is executed when the program is run with least one possible input.

Global Data-Flow Analysis

- **Conservative estimation of data-flow information:**
  - When we compare the computed gen with the "true" gen we discover that the true gen is always a subset of the computed gen. on the other hand, the true kill is always a superset of the computed kill.
  - These containments hold even after we consider the other rules. It is natural to wonder whether these differences between the true and computed gen and kill sets present a serious obstacle to data-flow analysis. The answer lies in the use intended for these data.

Global Data-Flow Analysis

- **Conservative estimation of data-flow information:**
  - o Overestimating the set of definitions reaching a point does not seem serious; it merely stops us from doing an optimization that we could legitimately do. On the other hand, underestimating the set of definitions is a fatal error; it could lead us into making a change in the program that changes what the program computes. For the case of reaching definitions, then, we call a set of definitions safe or conservative if the estimate is a superset of the true set of reaching definitions. We call the estimate unsafe, if it is not necessarily a superset of the truth.

Global Data-Flow Analysis

- **Conservative estimation of data-flow information:**

  o   Returning now to the implications of safety on the estimation of gen and kill for reaching definitions, note that our discrepancies, supersets for gen and subsets for kill are both in the safe direction. Intuitively, increasing gen adds to the set of definitions that can reach a point, and cannot prevent a definition from reaching a place that it truly reached. Decreasing kill can only increase the set of definitions reaching any given point.

Global Data-Flow Analysis

- **Computation of in and out:**

  - Many data-flow problems can be solved by synthesized translations similar to those used to compute gen and kill. It can be used, for example, to determine loop-invariant computations.

  - However, there are other kinds of data-flow information, such as the reaching-definitions problem. It turns out that in is an inherited attribute, and out is a synthesized attribute depending on in. we intend that in[S] be the set of definitions reaching the beginning of S, taking into account the flow of control throughout the entire program, including statements outside of S or within which S is nested.

Global Data-Flow Analysis

- **Computation of in and out:**
  - The set out[S] is defined similarly for the end of s. it is important to note the distinction between out[S] and gen[S]. The latter is the set of definitions that reach the end of S without following paths outside S.
  - Assuming we know in[S] we compute out by equation, that is

    Out[S] = gen[S] U (in[S] - kill[S])
  - Considering cascade of two statements S1; S2, as in the second case. We start by observing in[S1]=in[S]. Then, we recursively compute out[S1], which gives us in[S2], since a definition reaches the beginning of S2 if and only if it reaches the end of S1. Now we can compute out[S2], and this set is equal to out[S].

Global Data-Flow Analysis

- **Computation of in and out:**
  - Considering if-statement we have conservatively assumed that control can follow either branch, a definition reaches the beginning of S1 or S2 exactly when it reaches the beginning of S.

    $$In[S1] = in[S2] = in[S]$$

  - If a definition reaches the end of S if and only if it reaches the end of one or both sub statements; i.e,

    $$Out[S]=out[S1] \cup out[S2]$$

Global Data-Flow Analysis

- **Representation of sets:**
  - Sets of definitions, such as gen[S] and kill[S], can be represented compactly using bit vectors. We assign a number to each definition of interest in the flow graph. Then bit vector representing a set of definitions will have 1 in position I if and only if the definition numbered I is in the set.
  - The number of definition statement can be taken as the index of statement in an array holding pointers to statements. However, not all definitions may be of interest during global data-flow analysis. Therefore the number of definitions of interest will typically be recorded in a separate table.

Global Data-Flow Analysis

- **Representation of sets:**
  o A bit vector representation for sets also allows set operations to be implemented efficiently. The union and intersection of two sets can be implemented by logical or and logical and, respectively, basic operations in most systems-oriented programming languages. The difference A-B of sets A and B can be implemented by taking the complement of B and then using logical and to compute A

Global Data-Flow Analysis

- **Local reaching definitions:**
  - Space for data-flow information can be traded for time, by saving information only at certain points and, as needed, recomputing information at intervening points. Basic blocks are usually treated as a unit during global flow analysis, with attention restricted to only those points that are the beginnings of blocks.
  - Since there are usually many more points than blocks, restricting our effort to blocks is a significant savings. When needed, the reaching definitions for all points in a block can be calculated from the reaching definitions for the beginning of a block.

Global Data-Flow Analysis

- **Use-definition chains:**
  - It is often convenient to store the reaching definition information as" use-definition chains" or "ud-chains", which are lists, for each use of a variable, of all the definitions that reaches that use. If a use of variable a in block B is preceded by no unambiguous definition of a, then ud-chain for that use of a is the set of definitions in in[B] that are definitions of a. In addition, if there are ambiguous definitions of a ,then all of these for which no unambiguous definition of a lies between it and the use of a are on the ud-chain for this use of a.

Global Data-Flow Analysis

- **Evaluation order:**
  - o The techniques for conserving space during attribute evaluation, also apply to the computation of data-flow information using specifications. Specifically, the only constraint on the evaluation order for the gen, kill, in and out sets for statements is that imposed by dependencies between these sets. Having chosen an evaluation order, we are free to release the space for a set after all uses of it have occurred.
  - o Earlier circular dependencies between attributes were not allowed, but we have seen that data-flow equations may have circular dependencies.

Global Data-Flow Analysis

- **General control flow:**
  - Data-flow analysis must take all control paths into account. If the control paths are evident from the syntax, then data-flow equations can be set up and solved in a syntax directed manner.
  - When programs can contain goto statements or even the more disciplined break and continue statements, the approach we have taken must be modified to take the actual control paths into account.
  - Several approaches may be taken. The iterative method works arbitrary flow graphs. Since the flow graphs obtained in the presence of break and continue statements are reducible, such constraints can be handled systematically using the interval-based methods
  - However, the syntax-directed approach need not be abandoned when break and continue statements are allowed.

Code Improving Transformations

- Algorithms for performing the code improving transformations rely on data-flow information.

- Here we consider
  - Common sub-expression elimination,
  - Copy propagation and
  - Transformations for moving loop invariant computations out of loops and for eliminating induction variables.

- Global transformations are not substitute for local transformations; both must be performed.

Code Improving Transformations

- **Elimination of global common sub expressions:**
  - The available expressions data-flow problem discussed in the last section allows us to determine if an expression at point p in a flow graph is a common sub-expression.
  - The following algorithm formalizes the intuitive ideas presented for eliminating common subexpressions.

Code Improving Transformations

- **Elimination of global common sub expressions:**

    Algorithm: Global common sub expression elimination.

    Input: A flow graph with available expression information.

    Output: A revised flow graph.

    Method: For every statement s of the form x := y+z6 such that y+z is available at the beginning of block and neither y nor r z is defined prior to statement s in that block, do the following.

Code Improving Transformations

- **Elimination of global common sub expressions:**

  Algorithm: Global common sub expression elimination.

  Method:

  - ✓ To discover the evaluations of y+z that reach s's block, we follow flow graph edges, searching backward from s's block. However, we do not go through any block that evaluates y+z. The last evaluation of y+z in each block encountered is an evaluation of y+z that reaches s.

  - ✓ Create new variable u.

  - ✓ Replace each statement w: =y+z found in (1) by

      u : = y + z

      w : = u

  - ✓ Replace statement s by x:=u.

Code Improving Transformations

- **Elimination of global common sub expressions:**
  - ✓ Some remarks about this algorithm are in order.
    - o The search in step(1) of the algorithm for the evaluations of y+z that reach statement scan also be formulated as a data-flow analysis problem. However, it does not make sense to solve it for all expressions y+z and all statements or blocks because too much irrelevant information is gathered.
    - o Not all changes made by algorithm are improvements. We might wish to limit the number of different evaluations reaching s found in step (1), probably to one.

Code Improving Transformations

- **Elimination of global common sub expressions:**
  - ✓ Some remarks about this algorithm are in order.
    - o Algorithm will miss the fact that a*z and c*z must have the same value in

a :=x+y		b :=a*z
c :=x+y	vs	d :=c*z

    - o Because this simple approach to common sub expressions considers only the literal expressions themselves, rather than the values computed by expressions.

Code Improving Transformations

- **Copy Propagation**

  o Various algorithms introduce copy statements such as x :=copies may also be generated directly by the intermediate code generator, although most of these involve temporaries local to one block and can be removed by the dag construction. We may substitute y for x in all these places, provided the following conditions are met every such use u of x.

  o Statement s must be the only definition of x reaching u.

  o On every path from s to including paths that go through u several times, there are no assignments to y.

Code Improving Transformations

- **Copy Propagation**

  o Condition (1) can be checked using ud-changing information. We shall set up a new dataflow analysis problem in which in[B] is the set of copies s: x:=y such that every path from initial node to the beginning of B contains the statement s, and subsequent to the last occurrence of s, there are no assignments to y.

Code Improving Transformations

- **Copy Propagation**

  Algorithm: Copy propagation.

  Input: a flow graph G, with ud-chains giving the definitions reaching block B, and with c_in[B] representing the solution to equations that is the set of copies x:=y that reach block B along every path, with no assignment to x or y following the last occurrence of x:=y on the path. We also need ud-chains giving the uses of each definition.

  Output: A revised flow graph.

  Method: For each copy s : x:=y do the following:

Code Improving Transformations

- **Copy Propagation**

  Algorithm: Copy propagation.

  - ✓ Determine those uses of x that are reached by this definition of namely, s: x:=y.

  - ✓ Determine whether for every use of x found in (1) , s is in c_in[B], where B is the block of this particular use, and moreover, no definitions of x or y occur prior to this use of x within B. Recall that if s is in c_in[B]then s is the only definition of x that reaches B.

  - ✓ If s meets the conditions of (2), then remove s and replace all uses of x found in (1) by y.

Code Improving Transformations

- **Detection of loop-invariant computations:**
  - Ud-chains can be used to detect those computations in a loop that are loop-invariant, that is, whose value does not change as long as control stays within the loop. Loop is a region consisting of set of blocks with a header that dominates all the other blocks, so the only way to enter the loop is through the header.
  - If an assignment x := y+z is at a position in the loop where all possible definitions of y and z are outside the loop, then y+z is loop-invariant because its value will be the same each time x:=y+z is encountered. Having recognized that value of x will not change, consider v := x+w, where w could only have been defined outside the loop, then x+w is also loop-invariant.

Code Improving Transformations

- **Detection of loop-invariant computations:**

  Algorithm: Detection of loop-invariant computations.

  Input: A loop L consisting of a set of basic blocks, each block containing sequence of three-address statements. We assume ud-chains are available for the individual statements.

  Output: the set of three-address statements that compute the same value each time executed, from the time control enters the loop L until control next leaves L.

  Method: we shall give a rather informal specification of the algorithm, trusting that the principles will be clear.

Code Improving Transformations

- **Detection of loop-invariant computations:**

  Algorithm: Detection of loop-invariant computations.

  ✓ Mark "invariant" those statements whose operands are all either constant or have all their reaching definitions outside L.

  ✓ Repeat step (3) until at some repetition no new statements are marked "invariant".

  ✓ Mark "invariant" all those statements not previously so marked all of whose operands either are constant, have all their reaching definitions outside L, or have exactly one reaching definition, and that definition is a statement in L marked invariant.

Code Improving Transformations

- **Performing code motion:**
  - o Having found the invariant statements within a loop, we can apply to some of them an optimization known as code motion, in which the statements are moved to pre-header of the loop.
  - o The following three conditions ensure that code motion does not change what the program computes. Consider s: x: =y+z.

Code Improving Transformations

- **Performing code motion:**
    o The block containing s dominates all exit nodes of the loop, where an exit of a loop is a node with a successor not in the loop.
        ✓ There is no other statement in the loop that assigns to x. Again, if x is a temporary
        ✓ assigned only once, this condition is surely satisfied and need not be changed.
        ✓ No use of x in the loop is reached by any definition of x other than s. This condition too will be satisfied, normally, if x is temporary.

Code Improving Transformations

- **Performing code motion:**

  Algorithm : Code motion

  Input: A loop L with ud-chaining information and dominator information.

  Output: A revised version of the loop with a pre-header and some statements moved to the pre-header.

  Method:

  - ✓ Use loop-invariant computation algorithm to find loop-invariant statements.
  - ✓ For each statement s defining x found in step(1), check:
    - i. That it is in a block that dominates all exits of L,
    - ii. That x is not defined elsewhere in L, and
    - iii. That all uses in L of x can only be reached by the definition of x in statement s.

Code Improving Transformations

- **Performing code motion:**

  Input: A loop L with ud-chaining information and dominator information.

  Output: A revised version of the loop with a pre-header and some statements moved to the pre-header.

  Method:
  - ✓ Move, in the order found by loop-invariant algorithm, each statement s found in (1) and meeting conditions (2i), (2ii), (2iii) , to a newly created pre-header, provided any operands of s that are defined in loop L have previously had their definition statements moved to the pre-header.

Code Improving Transformations

- **Performing code motion:**
  - To understand why no change to what the program computes can occur, condition (2i) and (2ii) of this algorithm assure that the value of x computed at s must be the value of x after any exit block of L. When we move s to a pre-header, s will still be the definition of x that reaches the end of any exit block of L. Condition (2iii) assures that any uses of x within L did, and will continue to, use the value of x computed by s.

Code Improving Transformations

- **Alternative code motion strategies:**
  o The condition (1) can be relaxed if we are willing to take the risk that we may actually increase the running time of the program a bit; of course, we never change what the program computes. The relaxed version of code motion condition (1) is that we may move a statement s assigning x only if:

  > 1'. The block containing s either dominates all exists of the loop, or x is not used outside the loop. For example, if x is a temporary variable, we can be sure that the value will be used only in its own block.

Code Improving Transformations

- **Alternative code motion strategies:**
  - o If code motion algorithm is modified to use condition (1'), occasionally the running time will increase, but we can expect to do reasonably well on the average. The modified algorithm may move to pre-header certain computations that may not be executed in the loop. Not only does this risk slowing down the program significantly, it may also cause an error in certain circumstances.
  - o Even if none of the conditions of (2i), (2ii), (2iii) of code motion algorithm are met by an assignment x: =y+z, we can still take the computation y+z outside a loop. Create a new temporary t, and set t: =y+z in the pre-header. Then replace x: =y+z by x: =t in the loop. In many cases, we can propagate out the copy statement x: = t.

Code Improving Transformations

- **Maintaining data-flow information after code motion:**
  - o The transformations of code motion algorithm do not change ud-chaining information, since by condition (2i), (2ii), and (2iii), all uses of the variable assigned by a moved statement s that were reached by s are still reached by s from its new position.
  - o Definitions of variables used by s are either outside L, in which case they reach the preheader, or they are inside L, in which case by step (3) they were moved to pre-header ahead of s.

Code Improving Transformations

- **Maintaining data-flow information after code motion:**
  - o If the ud-chains are represented by lists of pointers to pointers to statements, we can maintain ud-chains when we move statement s by simply changing the pointer to s when we move it. That is, we create for each statement s pointer ps, which always points to s.
  - o We put the pointer on each ud-chain containing s. Then, no matter where we move s, we have only to change ps , regardless of how many ud-chains s is on.
  - o The dominator information is changed slightly by code motion. The pre-header is now the immediate dominator of the header, and the immediate dominator of the pre-header is the node that formerly was the immediate dominator of the header. That is, the pre-header is inserted into the dominator tree as the parent of the header.

Code Improving Transformations

- **Elimination of induction variable:**
  - A variable x is called an induction variable of a loop L if every time the variable x changes values, it is incremented or decremented by some constant. Often, an induction variable is incremented by the same constant each time around the loop, as in a loop headed by for i := 1 to 10.
  - However, our methods deal with variables that are incremented or decremented zero, one, two, or more times as we go around a loop. The number of changes to an induction variable may even differ at different iterations.

Code Improving Transformations

- **Elimination of induction variable:**
  - A common situation is one in which an induction variable, say i, indexes an array, and some other induction variable, say t, whose value is a linear function of i, is the actual offset used to access the array. Often, the only use made of i is in the test for loop termination. We can then get rid of i by replacing its test by one on t.
  - We shall look for basic induction variables, which are those variables i whose only assignments within loop L are of the form i := i+c or i-c, where c is a constant.

Code Improving Transformations

- **Elimination of induction variable:**

  Algorithm: Elimination of induction variables.

  Method:

  ✓ Consider each basic induction variable i whose only uses are to compute other induction variables in its family and in conditional branches. Take some j in i's family, preferably one such that c and d in its triple are as simple as possible and modify each test that i appears in to use j instead. We assume in the following tat c is positive.

Code Improving Transformations

- **Elimination of induction variable:**

  Algorithm: Elimination of induction variables.

  Method:

  ✓ A test of the form 'if i relop x goto B', where x is not an induction variable, is replaced by

    r := c*x /* r := x if c is 1. */

    r := r+d /* omit if d is 0 */

    if j relop r goto B

  where, r is a new temporary. The case 'if x relop i goto B' is handled analogously. If there are two induction variables i1 and i2 in the test if i1 relop i2 goto B, then we check if both i1 and i2 can be replaced. The easy case is when we have j1 with triple and j2 with triple, and c1=c2 and d1=d2. Then, i1 relop i2 is equivalent to j1 relop j2.

Code Improving Transformations

- **Elimination of induction variable:**

  Algorithm: Elimination of induction variables.

  Method:

  - ✓ Now, consider each induction variable j for which a statement j: =s was introduced. First check that there can be no assignment to s between the introduced statement j :=s and any use of j. In the usual situation, j is used in the block in which it is defined, simplifying this check; otherwise, reaching definitions information, plus some graph analysis is needed to implement the

    check. Then replace all uses of j by uses of s and delete statement j: =s.

References

- Alfred V. Aho, Ravi Sethi and Jeffrey D. Ullmann "Compilers- Principles, Techniques and Tools", Pearson Education.

Thank You !!!