Introduction to Machine-Independent Optimizations - 5

Control-Flow Analysis

Y.N. Srikant

Department of Computer Science and Automation
Indian Institute of Science
Bangalore 560 012

NPTEL Course on Principles of Compiler Design

Outline of the Lecture

- What is code optimization? (in part 1)
- Illustrations of code optimizations (in part 1)
- Examples of data-flow analysis (in parts 2,3, and 4)
- Fundamentals of control-flow analysis
- Algorithms for two machine-independent optimizations
- SSA form and optimizations

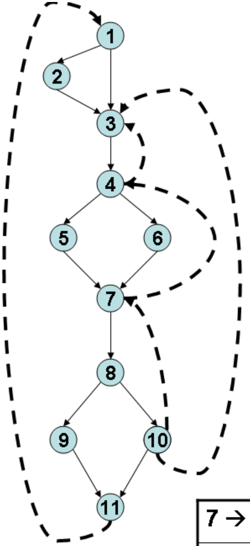
Dominators and Natural Loops

- Edges whose heads dominate their tails are called *back edges* ($a \rightarrow b: \ b = head, \ a = tail$)
- Given a back edge $n \rightarrow d$
  - The *natural loop* of the edge is d plus the set of nodes that can reach n without going through d
  - d is the header of the loop
    - A single entry point to the loop that dominates all nodes in the loop
    - At least one path back to the header exists (so that the loop can be iterated)

Algorithm for finding the Natural Loop of a Back Edge

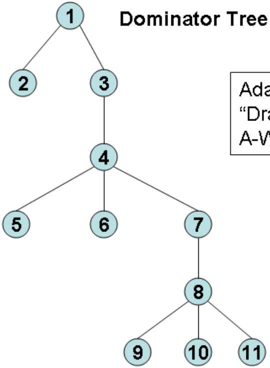
```
/* The back edge under consideration is n → d /*
{ stack = empty; loop = {d};
 /* This ensures that we do not look at predecessors of d */
  insert(n);
  while (stack is not empty) do {
    pop(m, stack);
    for each predecessor p of m do insert(p);
  }
}

  procedure insert(m) {
    if m ∉ loop then {
      loop = loop ∪ {m};
      push(m, stack);
    }
  }
```



Flow Graph

Dominator Tree

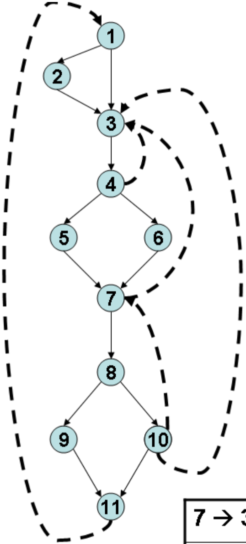


Adapted from the "Dragon Book", A-W, 1986

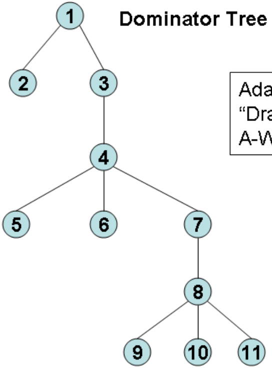
Back edges and their natural loops

$7 \rightarrow 4$	$10 \rightarrow 7$	$4 \rightarrow 3$	$10 \rightarrow 3$	$11 \rightarrow 1$
{4,5,6,7,8, 10}	{7,8,10}	{3,4,5,6,7, 8,10}	{3,4,5,6,7, 8,10}	{1,2,3,4,5, 6,7,8,9,10,11}

Dominators, Back Edges, and Natural Loops



Flow Graph



Adapted from the "Dragon Book", A-W, 1986

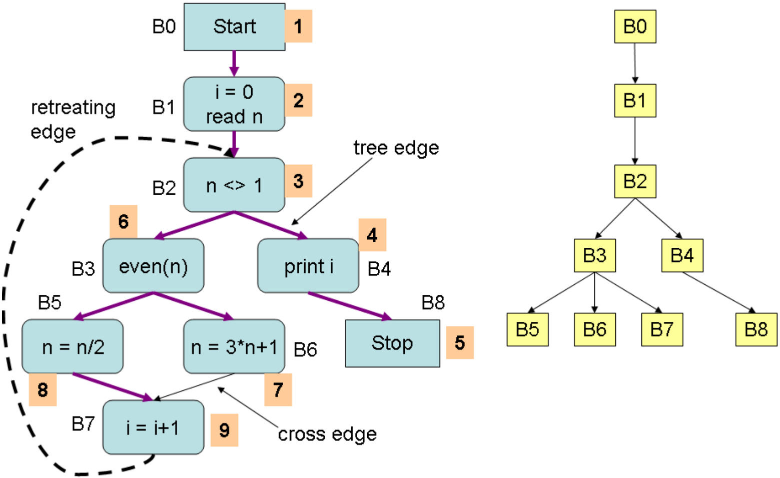
Back edges and their natural loops

7 → 3	10→7	4→3	10→3	11→1
{3,4,5,6,7,8, 10}	{7,8,10}	{3,4}	{3,4,5,6,7, 8,10}	{1,2,3,4,5, 6,7,8,9,10,11}

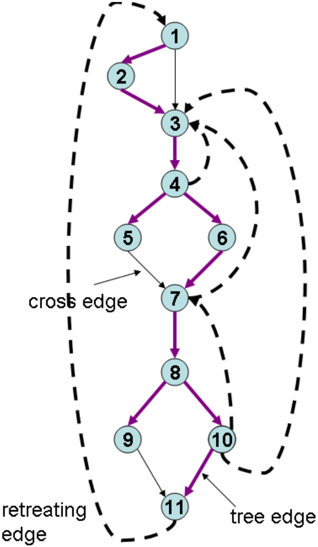
Depth-First Numbering of Nodes in a CFG

```
void dfs-num(int n) {
    mark node n "visited";
    for each node s adjacent to n do {
        if s is "unvisited" {
            add edge n →s to dfs tree T;
            dfs-num(s);
        }
     depth-first-num[n] = i ; i-- ;
}
// Main program
{  T = empty; mark all nodes of CFG as "unvisited";
   i = number of nodes of CFG;
   dfs-num(n0);// n0 is the entry node of the CFG
}
```

Depth-First Numbering Example 1

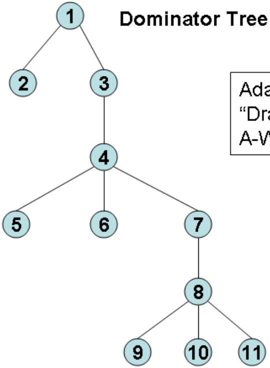


Depth-First Numbering Example 2



Flow Graph

Dominator Tree



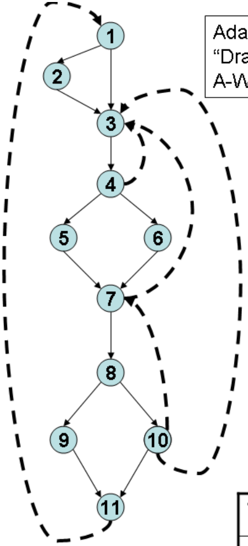
Adapted from the "Dragon Book", A-W, 1986

Nodes of the CFG show the DF-numbering

Inner Loops

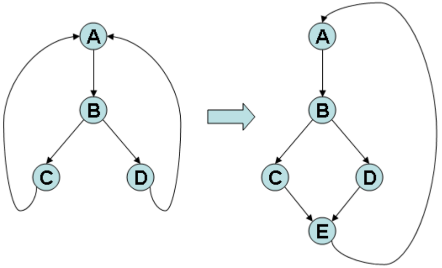
- Unless two loops have the same header, they are either disjoint or one is nested within the other
- Nesting is checked by testing whether the set of nodes of a loop A is a subset of the set of nodes of another loop B
- Similarly, two loops are disjoint if their sets of nodes are disjoint
- When two loops share a header, neither of these may hold (see next slide)
- In such a case the two loops are combined and transformed as in the next slide

Inner Loops and Loops with the same header



Adapted from the "Dragon Book", A-W, 1986

C→A	D→A	E→A
{A,B,C}	{A,B,D}	{A,B,C, D,E}

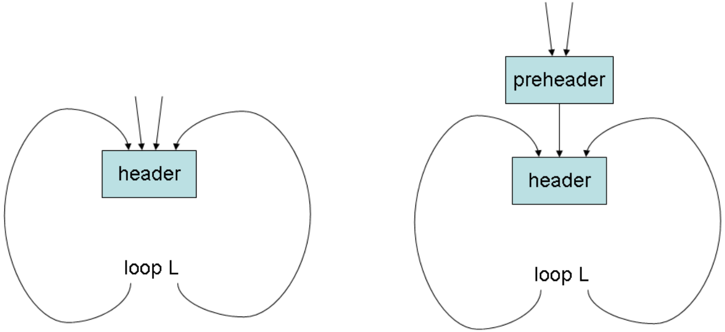


E is a dummy node

Back edges and their natural loops

7 → 3	10→7	4→3	10→3	11→1
{3,4,5,6, 7,8,10}	{7,8,10}	{3,4}	{3,4,5,6, 7,8,10}	{1,2,3,4,5, 6,7,8,9,10,11}

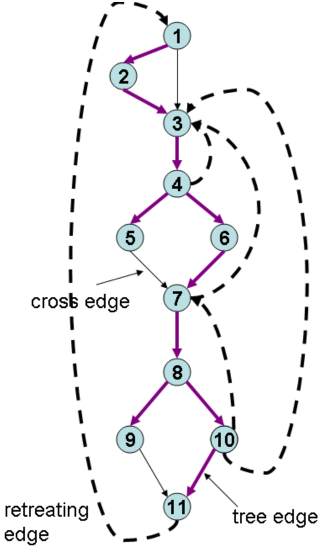
Preheader



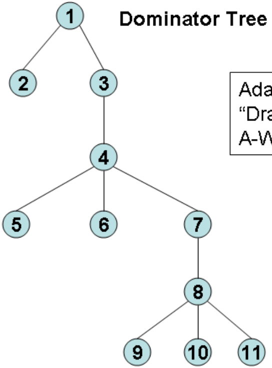
Depth of a Flow Graph and Convergence of DFA

- Given a depth-first spanning tree of a CFG, the largest number of retreating edges on any cycle-free path is the *depth* of the CFG
- The number of passes needed for convergence of the solution to a forward DFA problem is (1 + depth of CFG)
- One more pass is needed to determine *no change*, and hence the bound is actually (2 + depth of CFG)
- This bound can be actually met if we traverse the CFG using the *depth-first numbering* of the nodes
- For a backward DFA, the same bound holds, but we must consider the reverse of the depth-first numbering of nodes
- Any other order will still produce the correct solution, but the number of passes may be more

Depth of a CFG - Example 1



Flow Graph

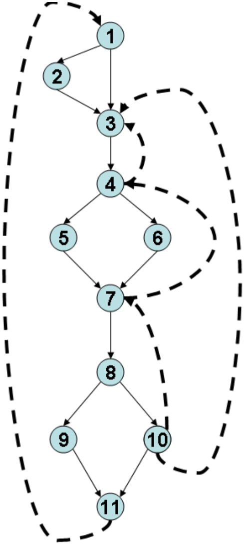


Adapted from the “Dragon Book”, A-W, 1986

Nodes of the CFG show the DF-numbering

Depth of the CFG = 2 (10-7-3)

Depth of a CFG - Example 2



Flow Graph

Adapted from the "Dragon Book", A-W, 1986

Depth of the CFG = 3 (10-7-4-3)

Algorithms for Machine-Independent Optimizations

Y.N. Srikant

Department of Computer Science and Automation
Indian Institute of Science
Bangalore 560 012

NPTEL Course on Principles of Compiler Design

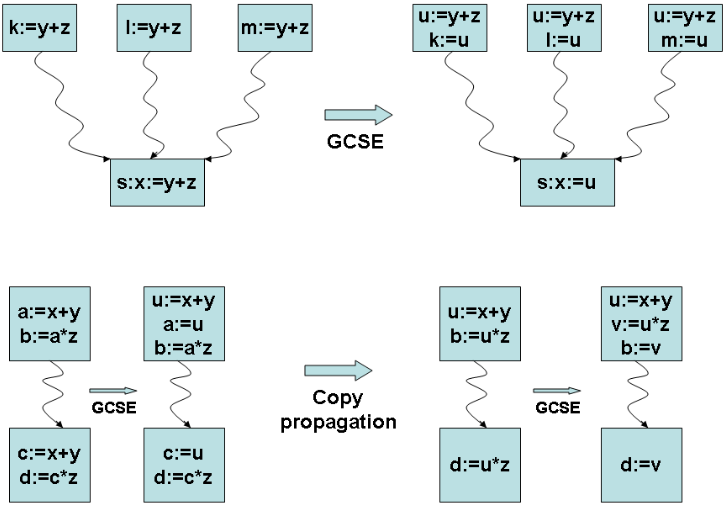
Outline of the Lecture

- Global common sub-expression elimination
- Copy propagation
- Simple constant propagation
- Loop invariant code motion

Elimination of Global Common Sub-expressions

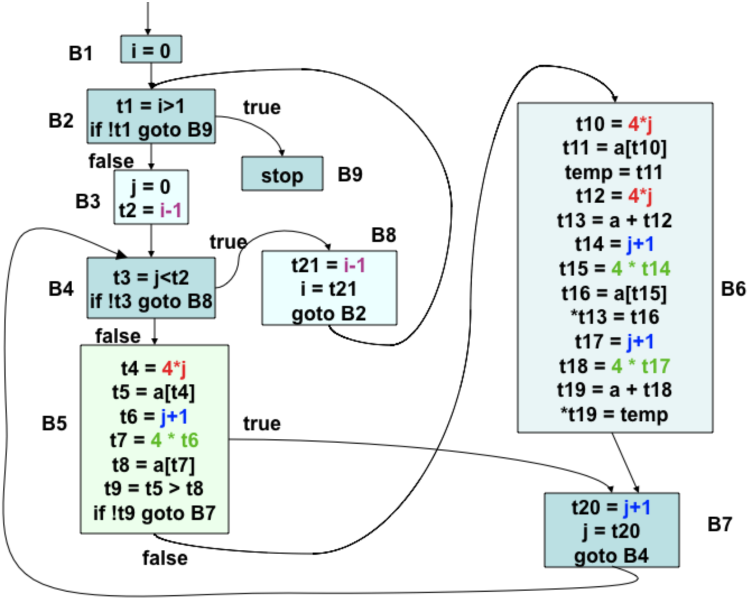
- Needs available expression information
- For every $s: \ x := y + z$, such that $y + z$ is available at the beginning of s' block, and neither y nor z is defined prior to s in that block, do the following
  1. Search backwards from s' block in the flow graph, and find first block in which $y + z$ is evaluated. We need not go *through* any block that evaluates $y + z$.
  2. Create a new variable u and replace each statement $w := y + z$ found in the above step by the code segment $\{u := y + z; w := u\}$, and replace s by $x := u$
  3. Repeat 1 and 2 above for every predecessor block of s' block
- Repeated application of GCSE may be needed to catch "deep" CSE

GCSE Conceptual Example

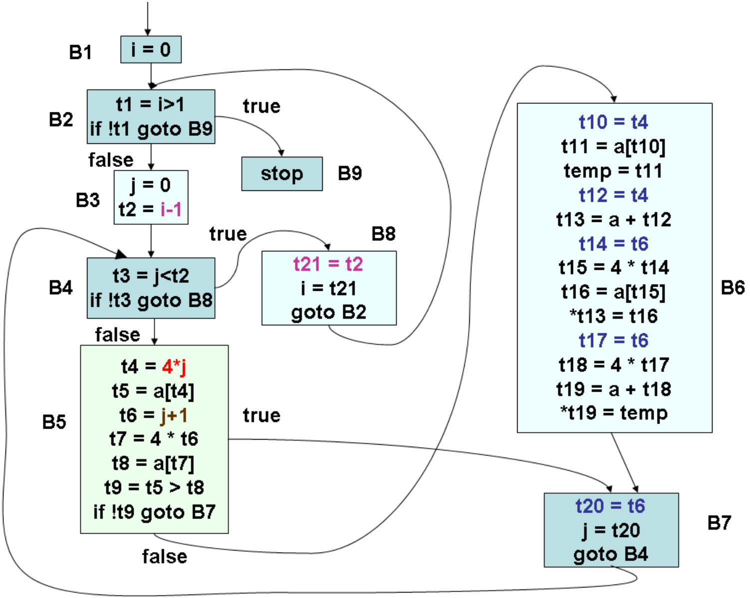


Demonstrating the need for repeated application of GCSE

GCSE on Running Example - 1



GCSE on Running Example - 2



Copy Propagation

- Eliminate copy statements of the form $s: \; x := y$, by substituting y for x in all uses of x reached by this copy
- Conditions to be checked
  1. u-d chain of use u of x must consist of s only. Then, s is the only definition of x reaching u
  2. On every path from s to u, including paths that go through u several times (but do not go through s a second time), there are no assignments to y. This ensures that the copy is valid
- The second condition above is checked by using information obtained by a new data-flow analysis problem
  - *c_gen*[B] is the set of all copy statements, $s : x := y$ in B, such that there are no subsequent assignments to either x or y within B, after s
  - *c_kill*[B] is the set of all copy statements, $s : x := y$, s not in B, such that either x or y is assigned a value in B
  - Let U be the universal set of all copy statements in the program

Copy Propagation - The Data-flow Equations

- $c\_in[B]$ is the set of all copy statements, $x := y$ reaching the beginning of B along every path such that there are no assignments to either x or y following the last occurrence of $x := y$ on the path
- $c\_out[B]$ is the set of all copy statements, $x := y$ reaching the end of B along every path such that there are no assignments to either x or y following the last occurrence of $x := y$ on the path

$$
\begin{aligned}
c\_in[B] &= \bigcap_{P \text{ is a predecessor of } B} c\_out[P],\ B \text{ not initial} \\
c\_out[B] &= c\_gen[B] \bigcup (c\_in[B] - c\_kill[B]) \\
c\_in[B1] &= \phi,\ \text{where } B1 \text{ is the initial block} \\
c\_out[B] &= U - c\_kill[B],\ \text{for all } B \neq B1\ (\text{initialization only})
\end{aligned}
$$

Algorithm for Copy Propagation

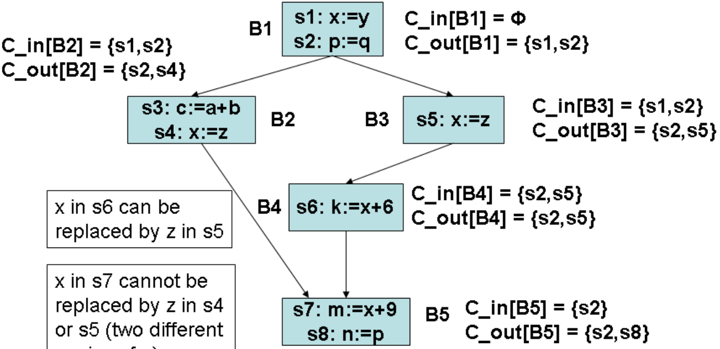
For each copy, $s : x := y$, do the following

1. Using the $du - chain$, determine those uses of x that are reached by s
2. For each use u of x found in (1) above, check that
   (i) u-d chain of u consists of s only
      - This implies that s is the only definition of x that reaches this block

   (ii) s is in $c\_in[B]$, where B is the block to which u belongs.
      - This ensures that no definitions of x or y appear on this path from s to B

   (iii) no definitions x or y occur within B prior to u found in (1) above
3. If s meets the conditions above, then remove s and replace all uses of x found in (1) above by y

Copy Propagation Example 1



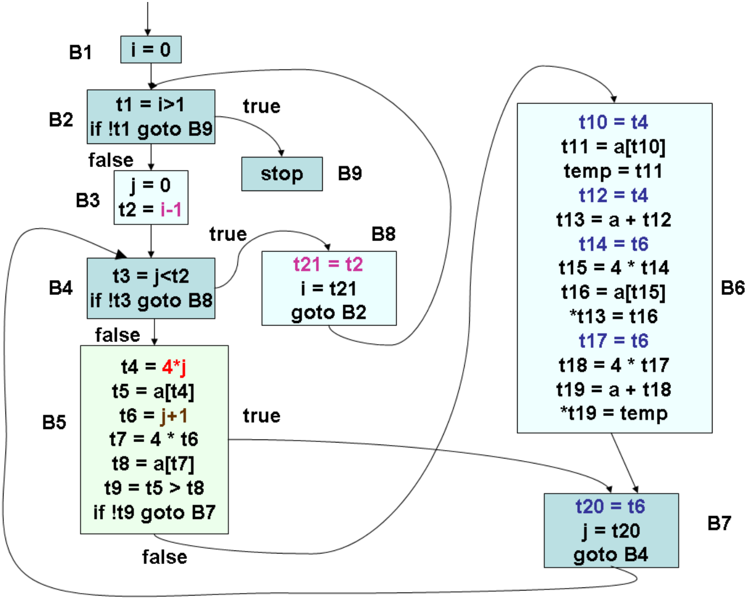
x in s6 can be replaced by z in s5

x in s7 cannot be replaced by z in s4 or s5 (two different copies of z)

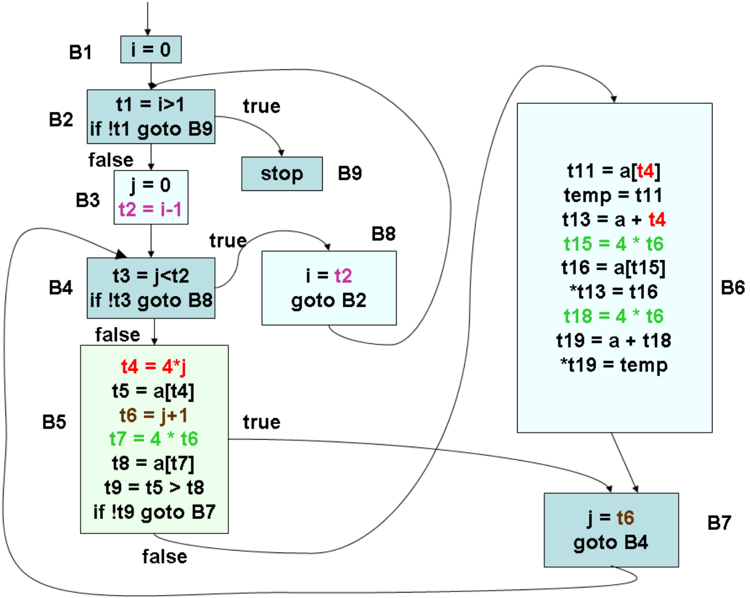
p in s8 can be replaced by q in s2 (s2 reaches B5 thro' both the paths)

Adapted from "The Dragon Book" A-W 1986

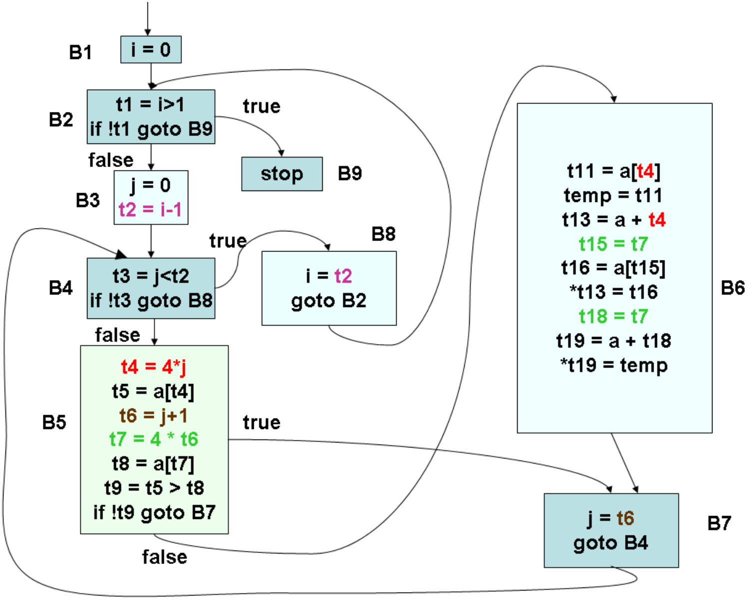
Copy Propagation on Running Example 1.1



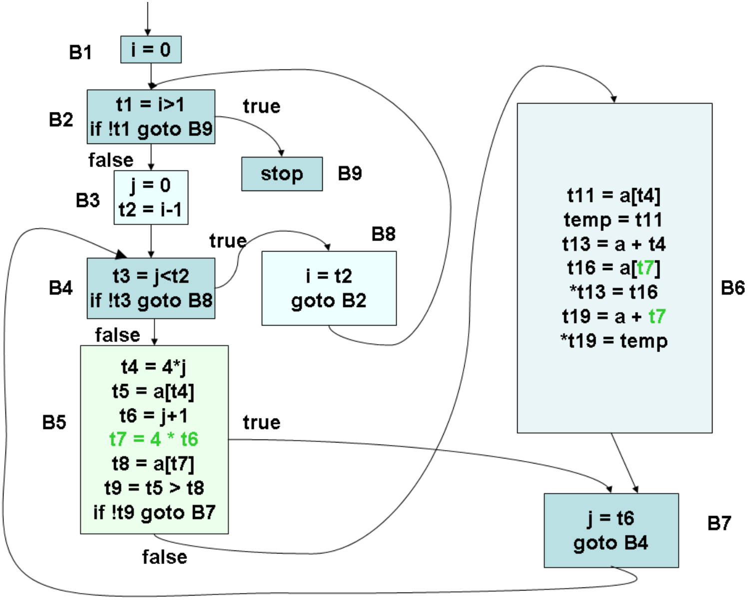
Copy Propagation on Running Example 1.2



GCSE and Copy Propagation on Running Example 1.1



GCSE and Copy Propagation on Running Example 1.2



Simple Constant Propagation

```
{   Stmtpile = {S|S is a statement in the program}
    while Stmtpile is not empty {
       S = remove(Stmtpile);
       if S is of the form x = c for some constant c
          for all statements T in the du-chain of x do
               if usage of x in T is reachable only by S
                 { substitute c for x in T; simplify T
                   Stmtpile = Stmtpile ∪ {T}
                 }
}
```

Note: If all usages of x are replaced by c, then $x = c$ becomes dead code and a separate dead code elimination pass will remove it.

Simple Constant Propagation Example