# Compiler Design Notes

Ayush Raina

February 9, 2025

## Control Flow Analysis

### Why Control Flow Analysis?

Helps to understand structure of Control Flow Graph (CFG), To detect loops in the CFG, Dominator Information,Dominator Frontier Information required for Single Static Assignment (SSA) form. Interval Information used in Data Flow Analysis, Control Dependence Information used in Parallelization.

### Dominators

1. We write ($d$ dom $n$) if every path from initial node $n_0$ to node $n$ passes through node $d$. A node dominates itself.
2. Node $x$ strictly dominates node $y$ if $x$ dominates $y$ and $x \neq y$.
3. $x$ is immediate dominator of $y$ if $x$ is closest strict dominator of $y$.

### Algorithm to find Dominators

Let us denote the set of Dominators of node $n$ as $D(n) = OUT(n)$. These sets are very helpful to construct the dominator tree quickly.

- Start with $n_0$ such that $D(n_0) = \{n_0\}$.
- for each $n \in N - \{n_0\}$ do the following:
  - First set $D(n) = N$.
  - Take intersection of all the outsets of the predecessors of $n$ denoted by $IN(n) = \bigcap_{p \in pred(n)} OUT(p)$.
  - $D(n) = IN(n) \cup \{n\}$.
- Repeat until $D(n)$ stabilizes for all $n$.

### Back Edges and Natural Loops

**Back Edges:** Edges whose head dominates the tail are called back edges. To find the back edges, one easy way to look if opposing edges according to dominator tree is present in the graph.
**Natural Loops:** Given a back edge $(n, d)$, the natural loop of the edge is $d \cup$ all the nodes that reach $n$ without going through $d$. In this case $d$ is the header of the loop and dominates all the nodes in the loop.

### Algorithm for finding the natural loop of a back edge

Let the back edge be $(n, d)$. Then initialize a empty stack. Push $n$ into the stack. Initialize a vector with $d$ in it. Then until the stack is empty, pop the top element and push its predecessors into the stack. If the predecessor is not in the vector, add it to the vector. Continue until the stack is empty. The vector now contains the natural loop.

### Depth First Numbering of Nodes in a CFG

We start a $DFS(n)$ from initial node $n_0$ where $n$ is total number of nodes. We mark a node to $n$ when it is last visit to that node. Last visit means that all the children of that node are visited and now this node will not be visited again. Then we decrement $n$ and continue the process. This way we can number the nodes in a CFG.

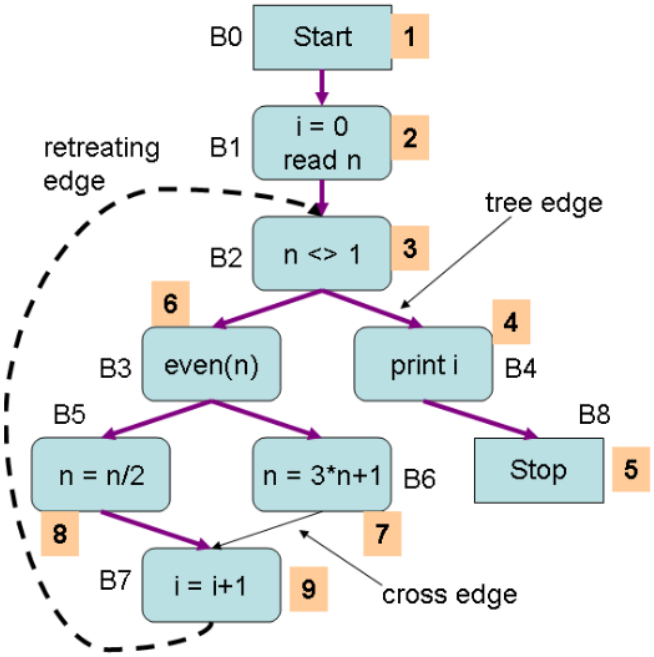


Figure 1: DFS Numbering of Nodes

## Reducibility

A flow graph $\mathcal{G}$ is reducible iff every back it can be partitioned into 2 disjoint sets of forward and back edges such that:

- Forward edges form a DAG in which every node is reachable from the initial node.
- All the retreating edges are back edges.
- In an irreducible flow graph, some retreating edges will not be back edges, hence graph of forward edges will contain a cycle.

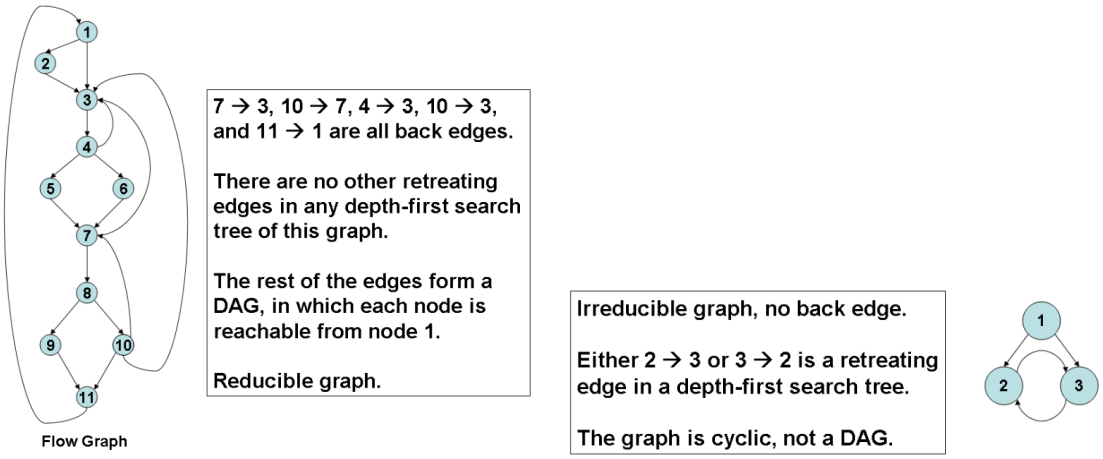


Figure 2: Reducible and Irreducible CFG

## Inner Loops

Unless two loops have same header $n \longrightarrow d$ (here d is header), they are either disjoint or one is nested inside the other. If natural loop set of one header is subset of the other then there is nesting. Similarly two loops are disjoint if their natural loop sets are disjoint and two loops are identical if their natural loop sets are identical.

If two loops share a header, neither of these may hold and in such a case loops are combined and transformed.

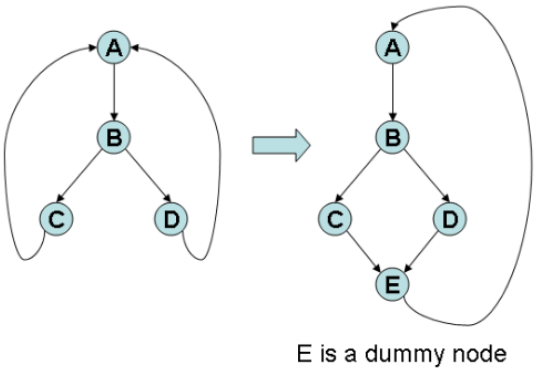


Figure 3: Transformation in case of shared header

## Pre Header

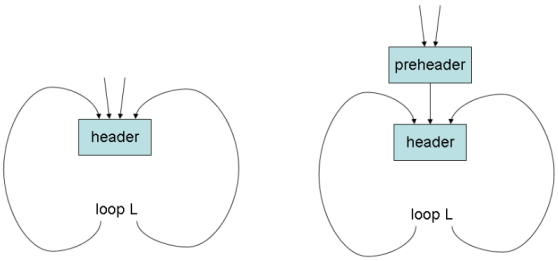


Figure 4: Pre Header

## Depth of a CFG and Convergence of DFA Problem

To be continued.

## Intervals

Intervals have a header node which dominates all the other nodes in that interval. Given a flow graph $\mathcal{G}$ with initial node $n_0$.The interval with header $n$ denoted by $I(n)$ is the set defined as:

1. $n \in I(n)$.
2. If $\exists\ m$ such that $Pred(m) \subset I(n)$ then $m \in I(n)$.
3. Nothing else is in $I(n)$.

Constructing $I(n)$:
1. Start with $I(n) = \{n\}$.
2. While there exists a node $m$ such that $Pred(m) \subset I(n)$, do $I(n) = I(n) \cup \{m\}$.

## Paritioning a flow graph into disjoint intervals

1. Mark all the nodes as unvisited.
2. Construct $I(n_0)$ and mark all the nodes in $I(n_0)$ as visited.
3. while there is an unvisited node $m$ with atleast one predecessor $p$ which is selected, construct $I(m)$ and mark all the nodes in $I(m)$ as visited and repeat the process.

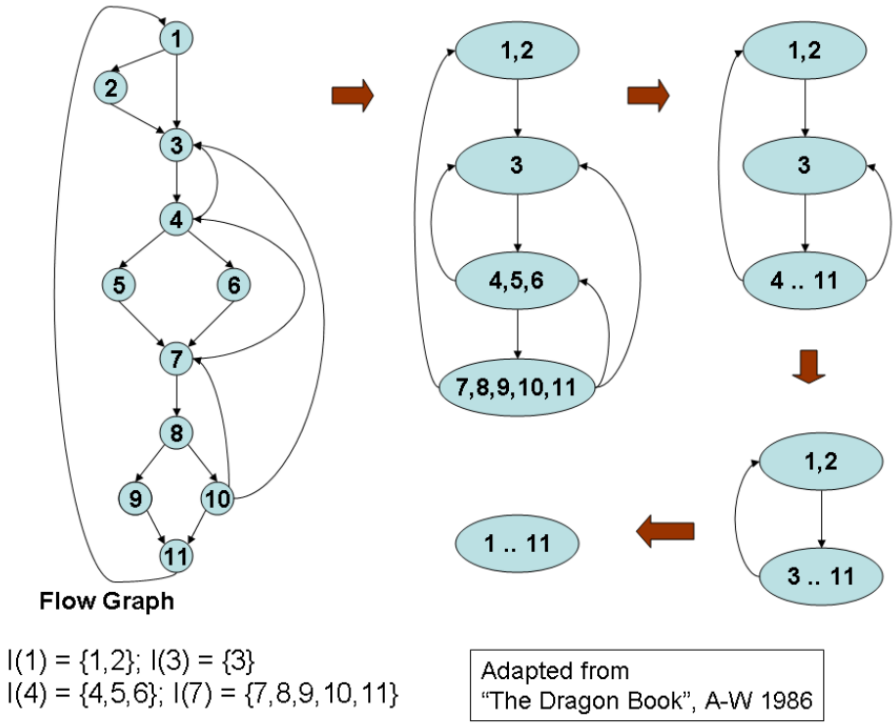


Figure 5: Intervals

In 1st step we got disjoint intervals. Now in the above figure we repeat same procedure further on the interval graph until we reach single node.

## Interval Graphs

1. Interval graph is denoted by $I(\mathcal{G})$. Intervals correspond to nodes and interval containing $n_0$ is the initial node of $I(\mathcal{G})$.
2. If there is an edge from a node in $I(m)$ to the header of interval $I(n)$ then there is an edge from $I(m)$ to $I(n)$ in $I(\mathcal{G})$.
3. We make intervals in interval graph until we reach a limit flow graph which cannot be reduced further.
4. A flow graph is reducible iff its limit flow graph is a single node.

## Node Splitting (Extra)

If we reach limit flow graph which is not a single node, we can still proceed further if we split one or more nodes.
1. If a node has $k$ predecessors then we may replace $n$ by $k$ nodes $n_1, n_2, \ldots, n_k$ such that $i^{th}$ predecessor of $n$ becomes predecessor of $n_i$ and all the successors of $n$ become successors of $n_i \forall i \in [1, k]$.
2. After splitting the nodes, we continue reduction process and do node splitting wherever required to reach a single node limit flow graph, however success is not guaranteed.

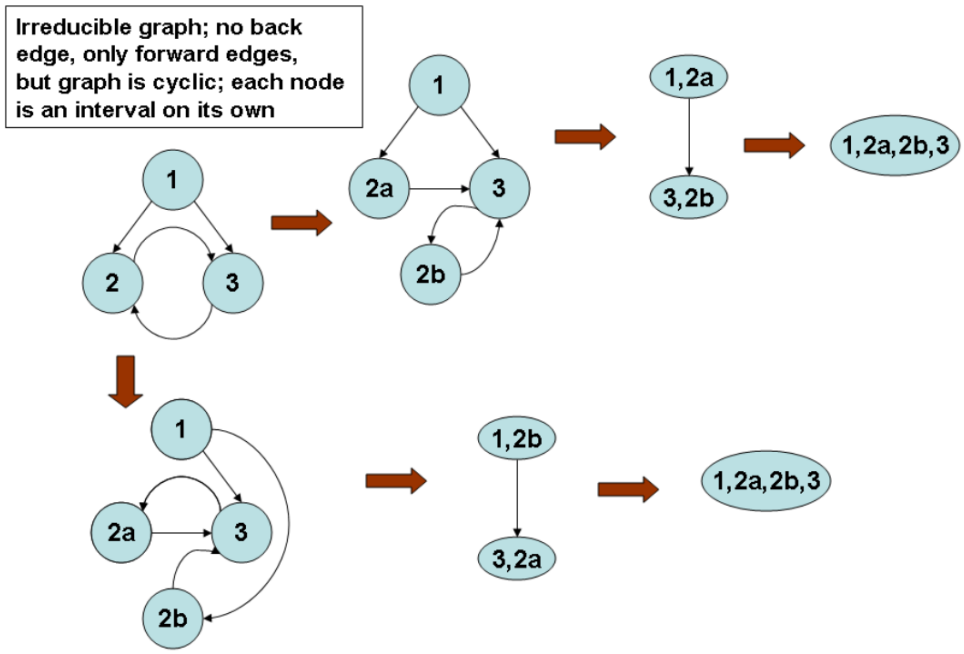


Figure 6: Node Splitting

## T1-T2 Transformations and Graph Reduction

**T1 Transformation:** If n is a node with a loop i.e $(n \to n)$ edge exists then delete that edge.
**T2 Transformation:** If there is a node $n$, not the initial node with a unique predecessor $m$ then combine $n$ into $m$.

By Applying T1,T2 transformations in any order we may reach limit flow graph and node splitting may be required wherever necessary in order to reach a single node limit flow graph.

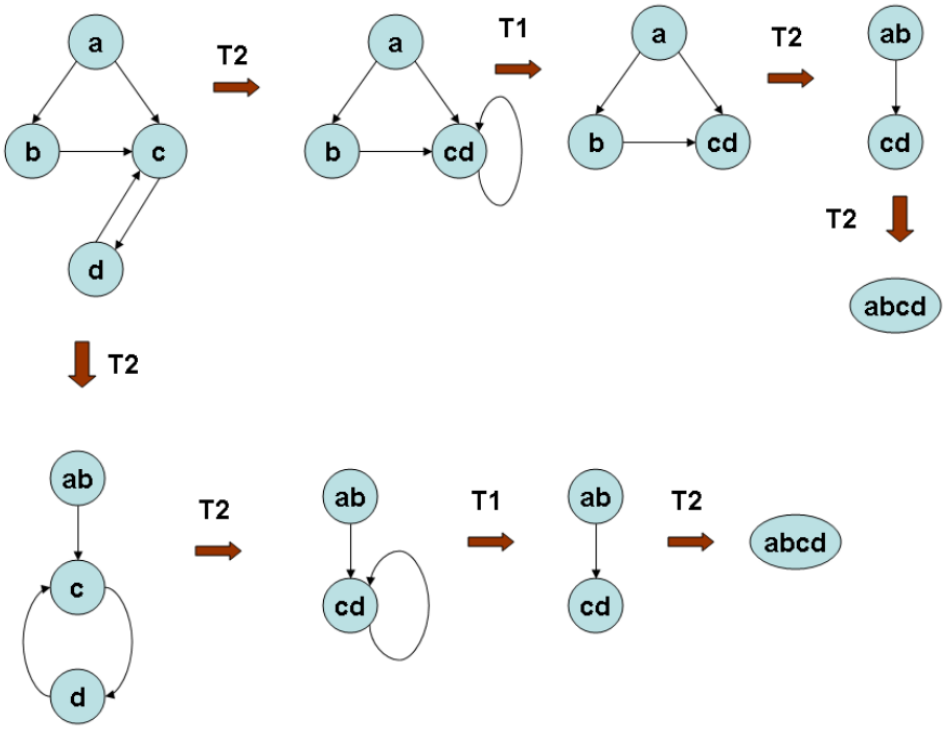


Figure 7: T1-T2 Transformations

One more example of a bigger CFG can be found in slides available on course webpage.

## Regions

A set of nodes $N$ that includes a header and it dominates all other nodes in $N$ is called region and all these edges between nodes in $N$ are in the region except (possibly) the edges that enter the header.

All intervals are regions but all regions are not intervals. A region may have multiple exits but an interval has only one exit. As we reduce a flow graph G using $T1 - T2$ transformations, at each step the node obtained is a region.

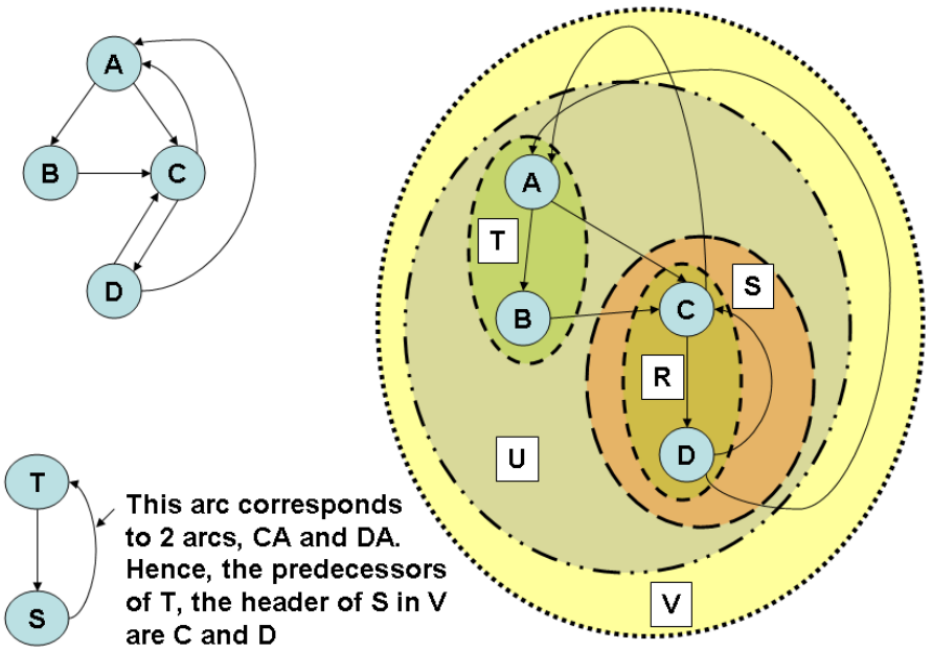


Figure 8: R,S,T,U,V are different regions marked

## End of Control Flow Analysis

# Data Flow Analysis

These are the techniques that derive information about flow of data along execution paths.
An execution path from path $p_1$ to $p_n$ is sequence of points such that for each $i \in [1, n-1]$ either $p_i$ is preceeding some statement $s$ and $p_{i+1}$ is immediately following the same statement $s$ or $p_i$ is end of some block and $p_{i+1}$ is start of successor block.
In general there can be infinite number of paths from a program and there is no bound on length of a path.

## Uses of DFA

Common uses are program debugging and program optimizations.
**Program debugging:** may include what are the definitions of variables reaching a particular point ? These are reaching definitions.
**Program Optimization:** It includes optimizations like common subexpression elimination.

## Data Flow Analysis Schema

Let us first see some definitions:
1. **Data flow value:** for a program point represents an abstraction of all possible program states that can be observed from that particular point in the program. The set of all possible data flow values is called Domain of the application under consideration.
2. **IN(s) and OUT(s)**: represents the data flow values at beginning and end of each statement $s$.
3. **Data Flow Problem:** is to find a solution to set of constraints on $IN(s)$ and $OUT(s)$ for all statements $s$. These constraints are of two types: constraints based on semantics of statements such as transfer functions and constraints based on control flow such as if-else conditions.
4. **Safe Estimates:** A decision or estimate is safe if it never leads to a change in what is being computed in the program (after the change).

A DFA schema consists of:
1. Control Flow Graph (CFG)
2. Direction of Data Flow (Forward or Backward)
3. Set of Data Flow Values (Domain)
4. Confluence Operator
5. Transfer Functions for each block

# Reaching Definitions

Purpose of reaching definitions is to perform optimizations by computing loop invariant. 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 on that path. We say a definition of a variable is killed if between 2 points along the path, there is re-assignment to the variable.

The data flow equations(constraints) are as follows: Initialize $IN(B) = \phi$ for all blocks $B$. If some definitions reach $B_1$ (entry block) then $IN(B_1)$ is initialized with those definitions. Then we have:
1. $IN(B) = \bigcup_{p \in pred(B)} OUT(p)$
2. $OUT(B) = GEN(B) \cup (IN(B) - KILL(B))$

This is a forward $DFA$ problem since $OUT(B)$ is computed in terms of $IN(B)$ and Confluence operator is $\cup$. $GEN(B)$ is the set of definitions which are visible immediately after the block $B$ and $KILL(B)$ is the union of definitions in all basic blocks which are killed by individual definitions in $B$. Here is an example of reaching definitions:

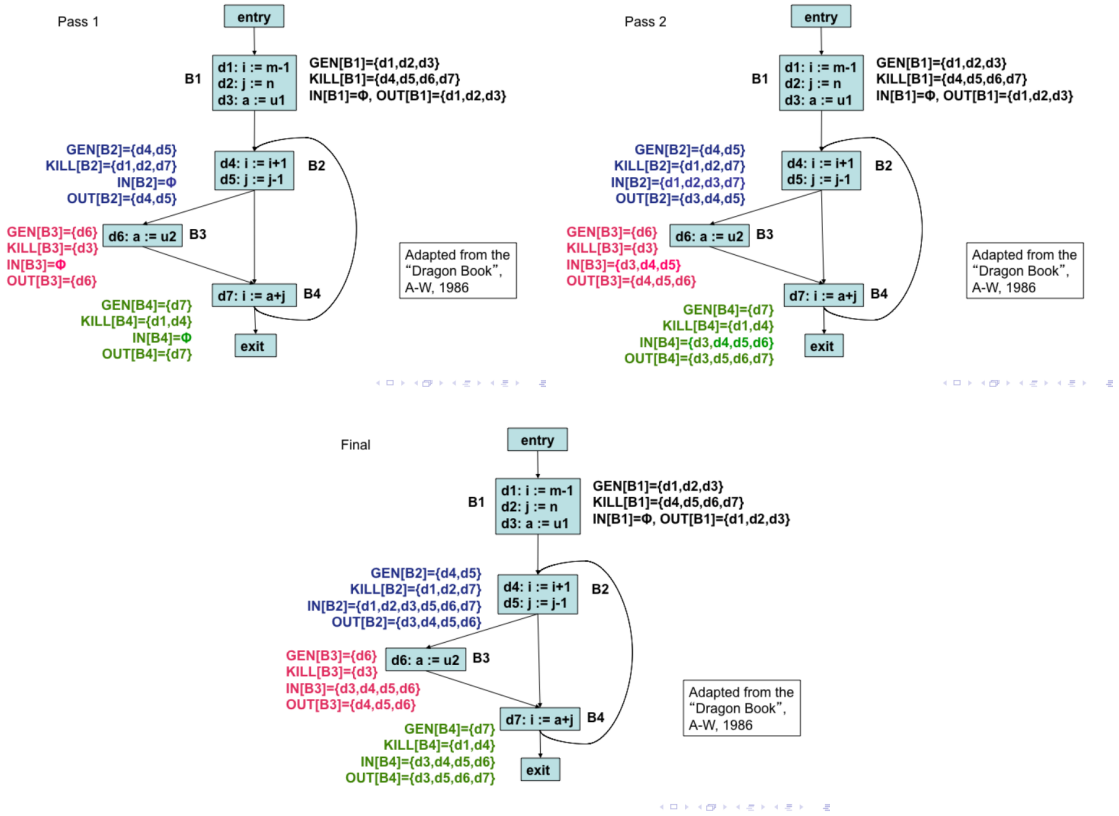


Figure 9: Reaching Definitions

In Pass 1, we initialize $IN(B) = \phi$ for all blocks and compute $OUT(B)$ using $GEN$, $KILL$ and $IN$ values. Now we repeat the process until $IN$ and $OUT$ values stabilize which in this case takes 3 iterations.

## Iterative Algorithm for computing Reaching Definitions

1. for each block $B$ initialize $IN(B) = \phi$ and $OUT(B) = GEN(B)$.
2. while there is a block $B$ such that $OUT(B)$ changes, do the following for each block $B$:

- $IN(B) = \bigcup_{p \in pred(B)} OUT(p)$
- $OUT(B) = GEN(B) \cup (IN(B) - KILL(B))$

GEN, KILL, IN, OUT are all represented as bit vectors with one bit for each definition in the flow graph which is shown in following figure.

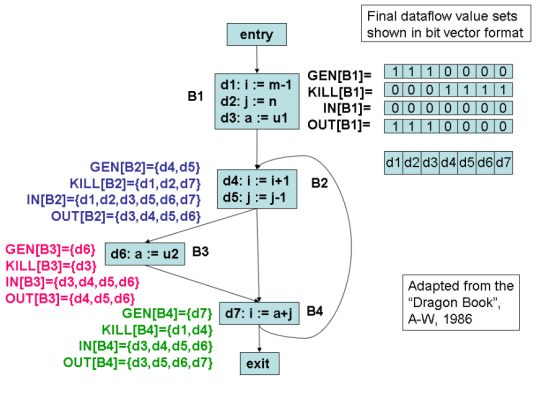


Figure 10: Bit Vector Representation

## Use-Definition Chains (u-d chains)

Convenient way to store reaching definitions once they are computed. A u-d chain is a **list of a use of a variable** and all the definitions that reach that use.

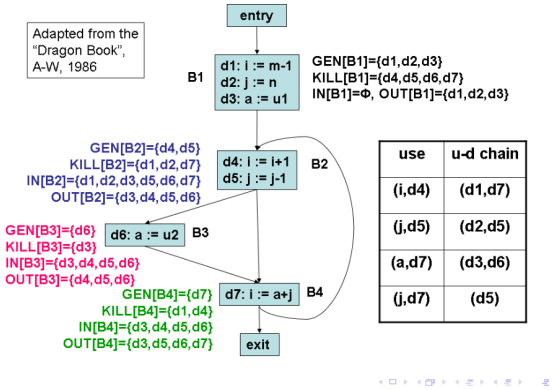


Figure 11: Use-Definition Chains

## Construction of ud chains

We need to consider 3 cases while constructing use-def chains which are as follows:

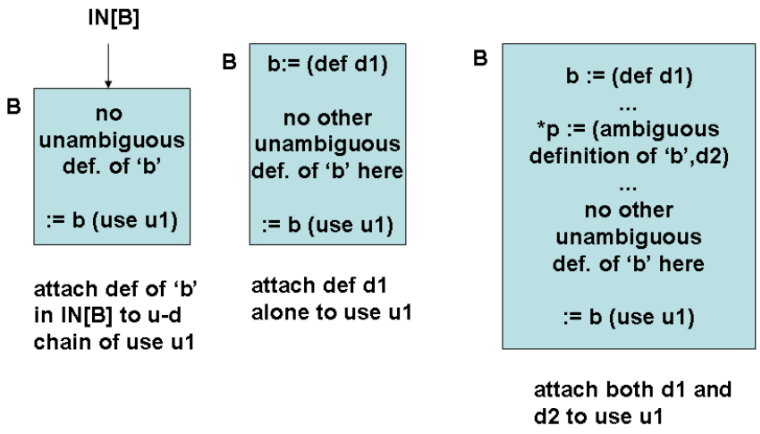


Figure 12: Construction of ud chains

# Available Expression Computation

This computation is necessary to handle common subexpression elimination optimization. This is again a forward flow problem with $\cap$ as Confluence operator.
An Expression $E_x$ is said to be available at a point $p$ if every path (may contain cycle) from initial node which reaches $p$ evaluates $E_x$ somewhere along the path before reaching $p$. There can be multiple evaluations of $E_x$ but after last such evaluation there are no assignments to any of the variables in $E_x$.

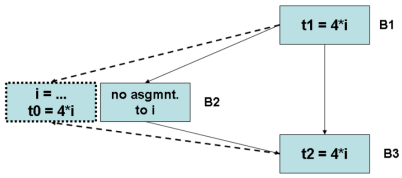


Figure 13: Available Expressions

In the figure above we can see that if we take path $B_2$ right then first evaluation to $4 * i$ is in $B_1$ and it is available at $B_2$. If we take path $B_2$ left then first evaluation to $4 * i$ then there is reassigment to $i$ and $4 * i$ is again computed,

hence it is available at $B_2$ in this case as well. In this case last evaluation of the expression is in $B_2$ and after the last evaluation there was no further re assignment to $i$.

A block kills $E_x$ if it assigns to any of the variables in $E_x$. For example consider $E(x, y) = x + y$. A block kills $E(x, y)$ if it assigns to $x$ or $y$. A block generates $E_x$ if it computes $E_x$ and does not kill it.

## Data Flow Equations for Available Expressions

The data flow equations are as follows:
1. $IN(B) = \bigcap_{p \in pred(B)} OUT(p)$ where $B$ is not the entry block.
2. $OUT(B) = \text{e\_gen}(B) \cup (IN(B) - \text{e\_kill}(B))$
3. $IN(B_1) = \phi$ where $B_1$ is the entry block.
4. $IN(B) = U$, for $B \neq B_1$ (Initialization)

$IN(B_1)$ is always $\phi$, U is the set of all expressions

## Computing e_gen and e_kill

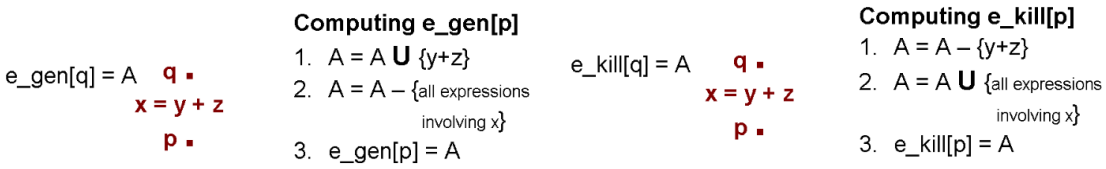


Figure 14: Computing e_gen and e_kill

In the left figure we did $A = A - \{\text{all expressions involving x}\}$ because of re assignment to $x$. In the right figure we did $A = A - \{y + z\}$ because in this case $A$ is set of killed expressions and $\{y + z\}$ is generated and due to re assignment to $x$, all expressions involving $x$ are added to $A$. Also for the statements of the form $x = a$, the step 1 of both algorithms shown does not apply.

## Iterative Algorithm for Computing Available Expressions

Algorithm 1:
1. for each block $B \neq B_1$ do $OUT(B) = U - e\_kill(B)$
2. while there is a block $B$ such that $OUT(B)$ changes, do the following for each block $B$:

- $IN(B) = \bigcap_{p \in pred(B)} OUT(p)$
- $OUT(B) = \text{e\_gen}(B) \cup (IN(B) - \text{e\_kill}(B))$

Algorithm2:
1. for each block $B \neq B_1$ do $IN(B) = U$
2. while there is a block $B$ such that $OUT(B)$ changes, do the following for each block $B$:

- $OUT(B) = \text{e\_gen}(B) \cup (IN(B) - \text{e\_kill}(B))$
- $IN(B) = \bigcap_{p \in pred(B)} OUT(p)$

In Algorithm 2, we have initialized $IN(B) = U$ first, and the only change in while loop is that $OUT(B)$ will be computed first and then $IN(B)$ will be computed.

## Why not $IN(B) = \phi$ ?

This initialization can be restrictive. We will get a valid solution, but it is possible that it may not be the largest possible solution. We require largest possible solution so that we can eliminate as many common subexpressions as possible.

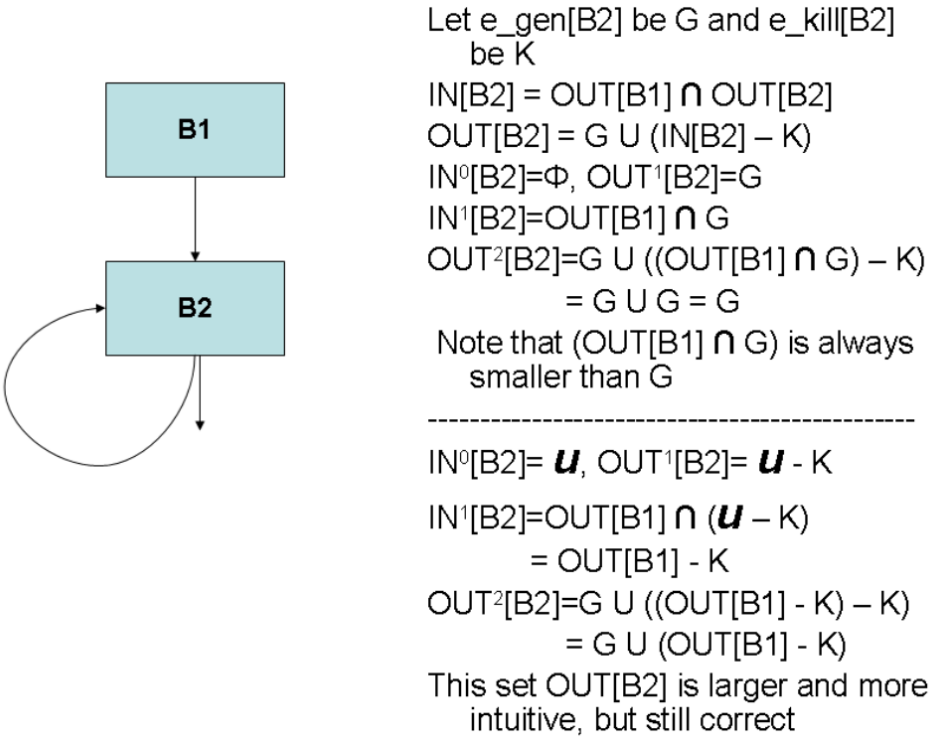


Figure 15: Why not $IN(B) = \phi$ ?

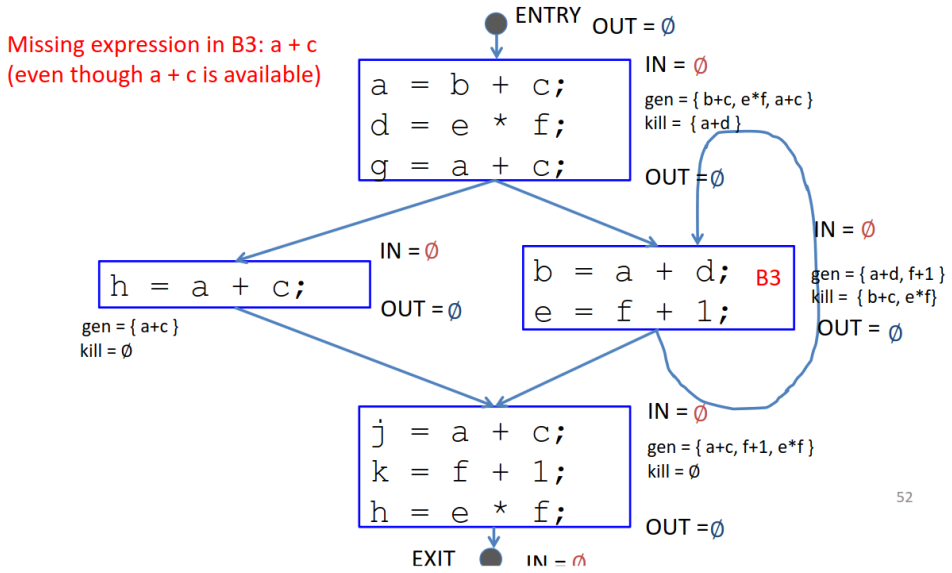


Figure 16: Available Expressions

In above figure an example is given in which there is a available expression but it is not computed because of initialization with $\phi$.

## Live Variable Analysis

We say a variable $x$ is live at a point $p$, if the value of $x$ at $p$ could be used at some path in the flow graph starting at $p$, otherwise $x$ is dead at $p$. In this case the domain of the data flow values is sets of variables. This is backward data flow problem, with confluence operator as $\cup$.

### Notations

$IN(B)$ is the set of variables live at the beginning of block $B$ and $OUT(B)$ is the set of variables live at the end of block $B$. DEF(B) is the set of variables which are assigned in B prior to being used where as $USE(B)$ is the set of variables

whose values may be used prior to definition of the variable.

## Data flow equations

Inittialize $IN(B) = \phi$ for all blocks $B$. Then we have:
1. $OUT(B) = \bigcup_{s \in succ(B)} IN(s)$
2. $IN(B) = USE(B) \cup (OUT(B) - DEF(B))$

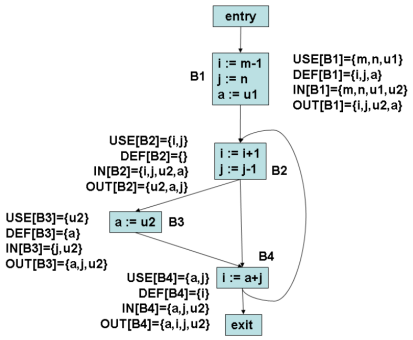


Figure 17: Final Live Variable Analysis after stabilization

Above figure shows the live variable analysis for a simple flow graph. Consider Block $B_2$, we can see that $i, j$ are used before assignment, hence they are in $USE(B_2)$. Similarly in $B_4$ - $a, j$ both are used before assignment and $i$ is assigned before use, hence $a, j \in USE(B_4)$ and $i \in DEF(B_4)$.

## Definition Use Chains

In this for each defintion we wish to attach the statement numbers of the uses of that defintion. This information is represented as sets of $(x, s)$ pairs where $x$ is the variable used in the statement $s$. In live variable analysis, we need information on whether a variable is used later but in $(x, s)$ computation we also need the statement number of the uses.

## Data Flow Analysis for $(x, s)$ pairs

Domain is $(x, s)$ pairs and this is again a backward data flow problem with confluence operator as $\cup$.

Here are some notations:
1. $USE(B)$ is the set of $(x, s)$ pairs $s$ is a statement in $B$ which uses $x$ before any reassignment to $x$ in $B$.
2. $DEF(B)$ is the set of $(x, s)$ pairs $s$ where is a statement number which uses $x$, $s$ is not in $B$ but $B$ contains a defintion of $x$ .
3. $IN(B)$ is the set of $(x, s)$ pairs such that statement $s$ uses $x$ and there is no re assignment to $x$ from beginning of $B$ to $s$.
4. $OUT(B)$ is the set of $(x, s)$ pairs such that statement $s$ uses $x$ and there is no re assignment to $x$ from ending of $B$ to $s$.

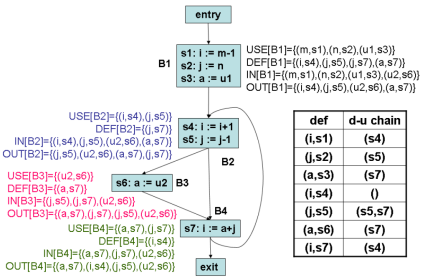


Figure 18: Definition Use Chains

Data Flow equations remain same as live variable analysis but the domain is $(x, s)$ pairs. Once $IN(B)$ and $OUT(B)$ are computed for $(x, s)$ pairs, we can construct definition use chains. as follows:

### Construction of Definition Use Chains

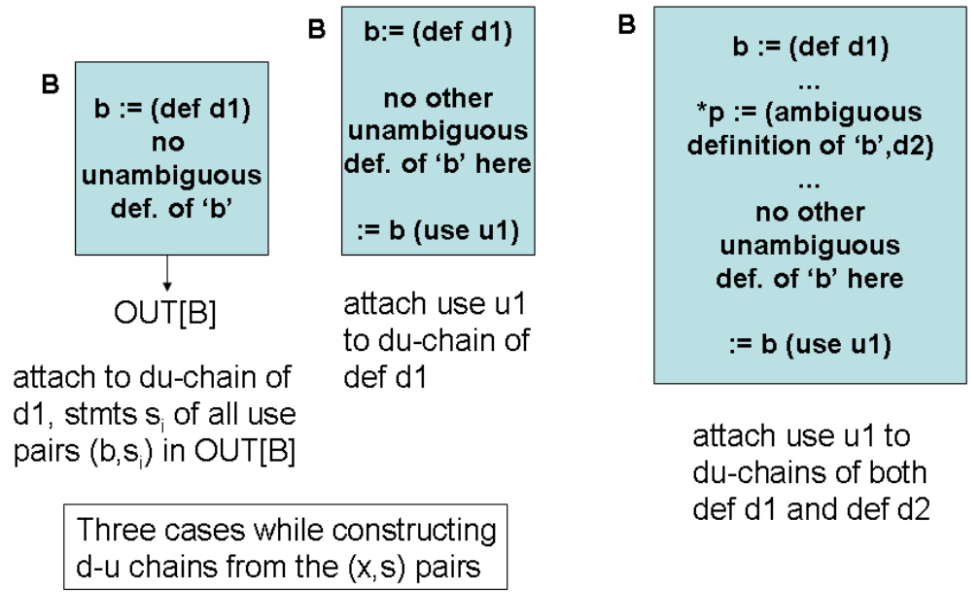


Figure 19: Construction of Definition Use Chains

## Anticipated Expressions or Very Busy Expressions

An expression $(B\ op\ C)$ is very busy or anticipated at a point $p$, if along every path from $p$, we come to a computation of $(B\ op\ C)$ before any computation of $B$ or $C$. This is useful in code hoisting which does not reduce time but reduces space.

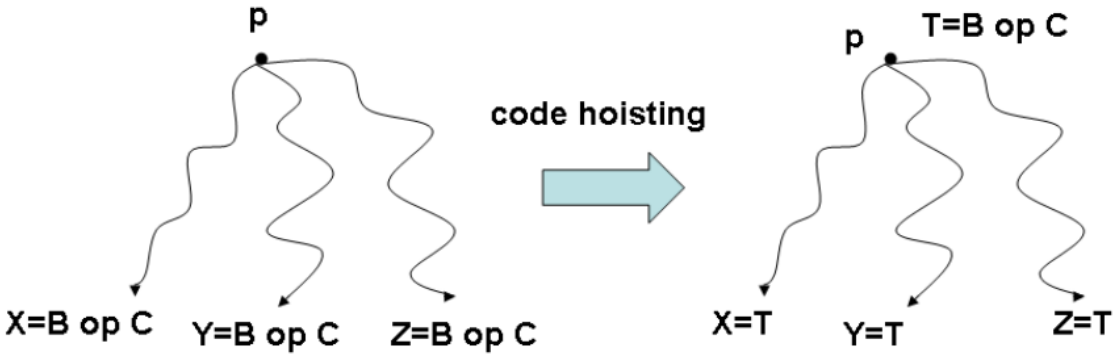


Figure 20: Code Hoising

In this case Domain is sets of expressions and this is also a backward data flow analysis problem with confluence operator as $\cap$.

Notations:
1. $V\_USE(n)$ is the set of expressions $(B\ op\ C)$ computed in $n$ with no prior assignment to $B$ or $C$ in $n$.
2. $V\_DEF(n)$ is the set of expressions $(B\ op\ C)$ in $U$ (universal set of expressions) for which either $B$ or $C$ is defined in $n$ prior to the computation of $(B\ op\ C)$.

### Data Flow Equations

1. $IN(n) = U$ for all $n$ (Initialization)
2. $OUT(n) = \bigcap_{s \in succ(n)} IN(s)$
3. $IN(n) = V\_USE(n) \cup (OUT(n) - V\_DEF(n))$

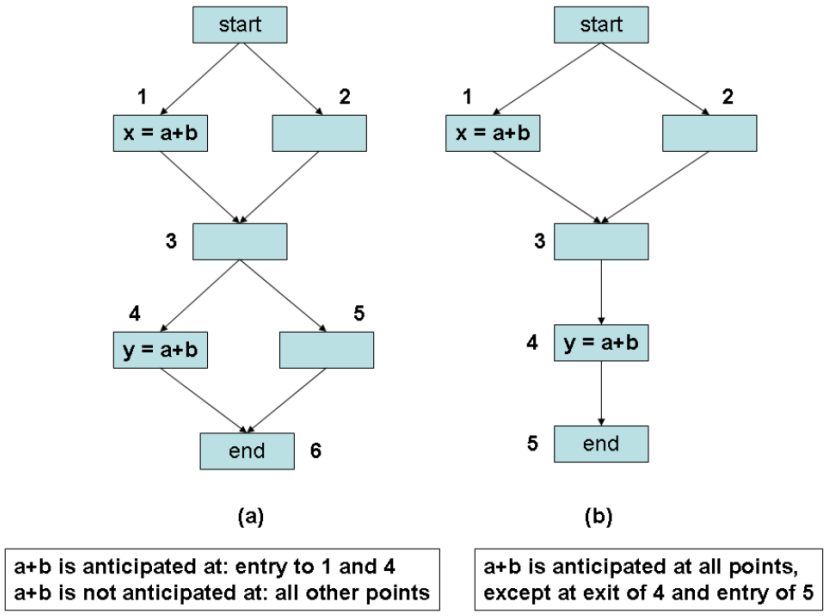


Figure 21: Anticipated Expressions

In the left figure from $n_1$ if we take path along $n_4$ then expression is computed in $n_4$, but if we consider path along $n_5$ then we will consider the computation of expression which is already done in $n_1$ because in definition it was mentioned that all paths from $p$.

The Reaching Definitions Problem

- Domain of data-flow values: sets of definitions
- Direction: Forwards
- Confluence operator: $\cup$
- Initialization: $IN[B] = \phi$
- Equations:

$$IN[B] = \bigcup_{P \text{ is a predecessor of } B} OUT[P]$$
$$OUT[B] = GEN[B] \bigcup (IN[B] - KILL[B])$$

The Available Expressions Problem

- Domain of data-flow values: sets of expressions
- Direction: Forwards
- Confluence operator: $\cap$
- Initialization: $IN[B] = U$
- Equations:

$$IN[B] = \bigcap_{P \text{ is a predecessor of } B} OUT[P]$$
$$OUT[B] = e\_gen[B] \bigcup (IN[B] - e\_kill[B])$$
$$IN[B1] = \phi$$

The Live Variable Analysis Problem

- Domain of data-flow values: sets of variables
- Direction: backwards
- Confluence operator: $\cup$
- Initialization: $IN[B] = \phi$
- Equations:

$$OUT[B] = \bigcup_{S \text{ is a successor of } B} IN[S]$$
$$IN[B] = USE[B] \bigcup (OUT[B] - DEF[B])$$

The Anticipated Expressions (Very Busy Expressions) Problem

- Domain of data-flow values: sets of expressions
- Direction: backwards
- Confluence operator: $\cap$
- Initialization: $IN[B] = U$
- Equations:

$$OUT[B] = \bigcap_{S \text{ is a successor of } B} IN[S]$$
$$IN[B] = V\_USE[B] \bigcup (OUT[B] - V\_DEF[B])$$

Figure 22: Summary of Data Flow Analysis

# Machine Independent Optimizations

## Elimination of global common subexpressions

To do this, we require available expression data flow information which we have already discussed.
Algorithm:
1. For every statement $s : x = y + z$ such that $(y + z)$ is already available at beginning of the block containing $s$, and neither $y$ nor $z$ is assigned in that block prior to $s$, do the following:

- Search backwards from block containing $s$ to find the first block where $(y + z)$ is computed.
- Create a new variable $u$ and replace each statement $w = y + z$ in these blocks by $u = y + z, w = u$.
- Set $x = u$ in statement $s$.
- Repeat steps 1-2 for every predecessor block of block containing $s$.

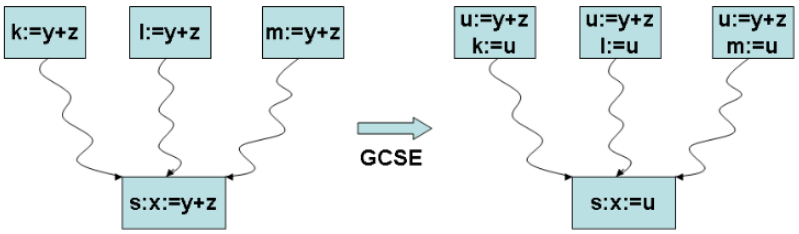


Figure 23: Global Common Subexpression Elimination

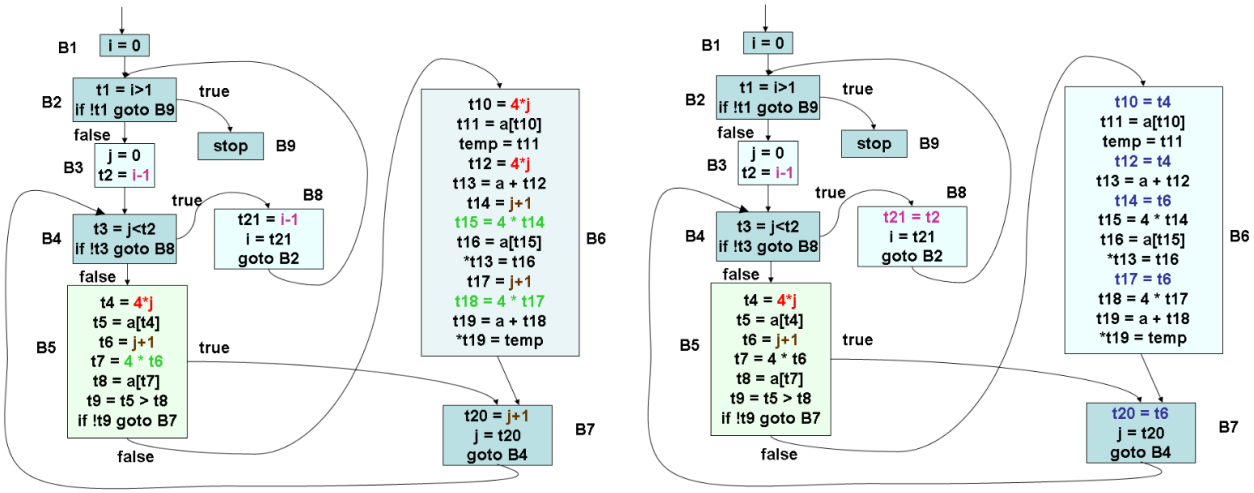


Figure 24: Global Common Subexpression Elimination

In this case we had only one predecessor block where the expression was computed, hence we did not created a new variable and instead used the same variable to eliminate the common subexpression.