Data Flow Analysis and Computation of SSA
Definitions

A basic block is the longest sequence of three-address codes with the following properties.

- The control flows to the block only through the first three-address code\(^a\).
- The control flows out of the block only through the last three-address code\(^b\).

\(^a\)There is no label in the middle of the code.
\(^b\)No three-address code other than the last one can be branch or jump.
Example

1: L2: v1 = i
2: v2 = j
3: if v1>v2 goto L3
4: v1 = j
5: v2 = i
6: v1 = v1 - v2
7: j = v1
8: goto L4
9: L3: v1 = i
10: v2 = j
11: v1 = v1 - v2
12: i = v1
13: L4: v1 = i
14: v2 = j
15: if v1<>v2 goto L2
Basic Block - $b_1$

1: L2: $v_1 = i$
2: $v_2 = j$
3: if $v_1 > v_2$ goto L3
3a: goto (4)
Basic Block - $b_2$

4: \quad v1 = j
5: \quad v2 = i
6: \quad v1 = v1 - v2
7: \quad j = v1
8: \quad goto L4
Basic Block - $b_3$

9: L3: v1 = i
10: v2 = j
11: v1 = v1 - v2
12: i = v1
12a: goto L4
Basic Block - $b_4$

13: L4: v1 = i
14    v2 = j
15    if v1<>v2 goto L2
15a   exit
A control-flow graph (CFG) is a directed graph $G = (V, E)$, where the nodes are the basic blocks and the edges correspond to the flow of control from one basic block to another. The edge $e_{ij} = (v_i, v_j)$ corresponds to the flow of control from the basic block $v_i$ to the basic block $v_j$. 
Control-Flow Graph

Entry

\[ b1: \]
L2: \[ v1 = i \]
\[ v2 = j \]
if \( v1 > v2 \) goto L3

\[ b2: \]
v1 = j
v2 = i
v1 = v1 - v2
j = v1
goto L4

\[ b3: \]
L3: \[ v1 = i \]
\[ v2 = j \]
v1 = v1 - v2
i = v1

\[ b4: \]
L4: \[ v1 = i \]
\[ v2 = j \]
if \( v1 <> v2 \) goto L2

Exit

Code Gen Example

Goutam Biswas
Note

- We assume that a CFG has a unique entry node and also has a unique exit node.
- In a CFG corresponding to a function/procedure may have multiple exit and entry points. A compiler can introduce unique initial and final basic blocks and edges to (from) multiple entry (exit) points.
Note

- A compiler may use 3-address code for each basic block and a CFG for a whole procedure or a program showing flow of control among the basic blocks.

- Different global optimizations require analysis of dataflow through the CFG.
Definition

- In a CFG with the unique entry block $B_0$, a basic block $B_i$ dominates a basic block $B_j$ if every path from $B_0$ to $B_j$ has $B_i$ on it.

- Dominance is a binary relation on the set of nodes (basic blocks) of a CFG. The relation $B_i$ dominates $B_j$ is written as $B_i \gg B_j$.

- If there is an edge from $B_i$ to $B_j$ in the CFG, then $B_i$ is called a predecessor of $B_j$. 

Code Gen Example
Definition

- For all node $B_i$, $B_i \gg B_i$ (reflexive).
  Similarly for all nodes $B_i, B_j, B_k$, if $B_i \gg B_j$ and $B_j \gg B_k$, then $B_i \gg B_k$ (transitive).

- For every node $B_i$, the set $\text{Dom}(B_i)$ is the collection of all dominators of $B_i$.

- For every node $B_i$, the set $\text{Pred}(B_i)$ is the collection of all predecessors of $B_i$. 
Code Gen Example
### Examples

<table>
<thead>
<tr>
<th>Node</th>
</tr>
</thead>
<tbody>
<tr>
<td>$B_0$</td>
</tr>
<tr>
<td>$B_1$</td>
</tr>
<tr>
<td>$B_2$</td>
</tr>
<tr>
<td>$B_3$</td>
</tr>
<tr>
<td>$B_8$</td>
</tr>
<tr>
<td>$B_9$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Predecessors</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\emptyset$</td>
</tr>
<tr>
<td>$B_0$</td>
</tr>
<tr>
<td>$B_0$</td>
</tr>
<tr>
<td>$B_1$</td>
</tr>
<tr>
<td>$B_4, B_5$</td>
</tr>
<tr>
<td>$B_7, B_8$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Dominators</th>
</tr>
</thead>
<tbody>
<tr>
<td>$B_0$</td>
</tr>
<tr>
<td>$B_0, B_1$</td>
</tr>
<tr>
<td>$B_0, B_2$</td>
</tr>
<tr>
<td>$B_0, B_1, B_3$</td>
</tr>
<tr>
<td>$B_0, B_2, B_8$</td>
</tr>
<tr>
<td>$B_0, B_9$</td>
</tr>
</tbody>
</table>
The dominator set of any node $B_i$ can be computed using the following inductive definition.

$$\text{Dom}(B_i) = \begin{cases} 
\{B_0\}, & \text{if } i = 0 \\
\{B_i\} \cup \bigcap_{B_j \in \text{Pred}(B_i)} \text{Dom}(B_j), & \text{if } i > 0.
\end{cases}$$
Note

- This equation corresponds to a forward data-flow analysis. The dominator set of \( B_i \) depends on the dominator set of its predecessor nodes.

- Similarly, there are properties of nodes that can be defined using its successors. This gives rise to backward data-flow analysis.
Algorithm

1. $\text{Dom} (B_0) \leftarrow \{B_0\}$
2. for $i \leftarrow 1$ to $n - 1$
3.     $\text{Dom} (B_i) \leftarrow \{B_0, \cdots, B_{n-1}\}$
4.     $\text{chFlg} \leftarrow \text{true}$
5. while $\text{chFlg} = \text{true}$
6.     $\text{chFlg} \leftarrow \text{false}$
7.     for $i \leftarrow 1$ to $n - 1$
8.         $\text{temp} \leftarrow \{B_i\} \cup \bigcap_{B_j \in \text{Pred}(B_i)} \text{Dom}(B_j)$
9.     if $\text{temp} \neq \text{Dom}(B_i)$ then
10.    $\text{Dom}(B_i) \leftarrow \text{temp}; \text{chFlg} \leftarrow \text{true}$
This is an example of fixed-point computation. The iteration of the while-loop terminates when the $\text{Dom}(b)$ cannot be reduced further. It is the fixed-point of a monotone function defined by the equation.
Liveness and Safety

- The algorithm terminates as in step 8 the size of \( \text{Dom}(B_i) \) may decrease or remain unchanged. This cannot continue indefinitely.

- The correctness of the algorithm comes from the correctness of the definition of \( \text{Dom}(B_i) \).

- The speed of termination depends on the ordering of the nodes in the CFG.
Live Variable Analysis

- Using a variable before initialization is a logical error.
- If a variable $x$ is (i) assigned a value in a 3-address code $i$, (ii) is used as an operand in a 3-address code $j$, and (iii) is not redefined in a path $p$ in the CFG from $i$ to $j$, then $x$ is live at $i$ and all points on path $p$. 
Liveness and Next Use

- For each 3-address code $a \leftarrow b \odot c$, we want to know the liveness and next-uses of the variables.

- It is easy to compute them within a basic block. So the present goal is to determine the liveness and next-use for 3-address codes in a basic block.
Liveness and Next Use in a Basic Block

- **Input**: a sequence of 3-address codes of a basic block $B$.
- **All variables within $B$** are initialized as live on exit from $B$, but next-use is unknown - (live, −)
Liveness and Next Use in a Basic Block

- The algorithm starts at the last 3-address code of $B$ and works backward.
- It attaches liveness and next-use information to variables in each assignment statement.
- These information may be stored in the symbol-table.
1. Get liveness and next use information (from the symbol table) of $x, y, z$ and attach it to the instruction $i$.

2. Update symbol-table: $x$: (not Live, noNextUse) as it is redefined at $i$.

3. Update symbol-table: $y, z$: (live, $i$) - the next use of both $y$ and $z$ is 3-address code $i$. 

$i : x = y \oplus z$
Basic Block - $b_3$

9: L3: v1 = i
10: v2 = j
11: v1 = v1 - v2
12: i = v1
12a: goto L4
Basic Block - $b_3$

symTab: $i:(L, U), j:(L, U), v1:(L, U), v2:(L, U)$

12: $i = v1$  \# $i: (L, U), v1: (L, U)$

symTab: $i:(D, N), j:(L, U), v1:(L, 12), v2:(L, U)$

11: $v1 = v1 - v2$  \# $v1: (L, 12), v2: (L, U)$

symTab: $i:(D, N), j:(L, U), v1:(L, 11), v2:(L, 11)$
Basic Block - $b_3$

\[
\text{symTab: } i: (D, N), j: (L, U), v1: (L, 11), v2: (L, 11)
\]

10: \[v2 = j\] \# \(v2: (L, 11), j: (L, U)\)

\[
\text{symTab: } i: (D, N), j: (L, 10), v1: (L, 11), v2: (D, N)
\]

9: \(L3: v1 = i\) \# \(v1: (L, 11), i: (D, N)\)

\[
\text{symTab: } i: (L, 9), j: (L, 10), v1: (D, N), v2: (D, N)
\]
Note - \( b_3 \)

<table>
<thead>
<tr>
<th>Var</th>
<th>Beginning</th>
<th>End</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Live</td>
<td>Next Use</td>
</tr>
<tr>
<td>( i )</td>
<td>Yes</td>
<td>9</td>
</tr>
<tr>
<td>( j )</td>
<td>Yes</td>
<td>10</td>
</tr>
<tr>
<td>( v_1 )</td>
<td>No</td>
<td>—</td>
</tr>
<tr>
<td>( v_2 )</td>
<td>No</td>
<td>—</td>
</tr>
</tbody>
</table>
Live-Variable Analysis on a CFG

- Given a variable $x$ and a point $p$ in a CFG, it is important to know whether $x$ is live at $p$ i.e. whether the value of $x$ at $p$ will be used at some path in the CFG starting at $p$.

- Computation of this information can be formulated as a dataflow equation. But in this case the information is computed opposite to the direction of the control-flow - backward dataflow.
Formulation of Data Flow Equation

- Given a basic block $B$, $\text{LIn}(B)$ and $\text{LOut}(B)$ are the sets of all variables that are live at the entry and exit of the block $B$.
- Let $\text{uFst}(B)$ be the set of variables whose values are used in $B$ before any definition.
- Let $\text{def}(B)$ be the set of variables that are defined in $B$. 
Formulation of Data Flow Equation

- Variables live at the exit of $B$ is the union of the variables live at the entry of its successors blocks.
  \[ \text{LOut}(B) = \bigcup_{S \in \text{succ}(B)} \text{LIn}(S). \]
- Variables live at the entry of $S$ contains all the variables in $\text{uFst}(S)$. 

Code Gen Example
Formulation of Data Flow Equation

- Variables of $\text{def}(S)$ cannot be live at the entry of $S$ unless it is in $\text{uFst}(B)$ i.e. used before a definition. Again $\text{def}(S) \subseteq \text{LOut}(S)$.

- So $\text{LOut}(S) \setminus \text{def}(S)$ is also contained in $\text{LIn}(S)$. So we have
  
  \[ \text{LOut}(B) = \bigcup_{S \in \text{succ}(B)} (\text{uFst}(S) \cup (\text{LOut}(S) \setminus \text{def}(S))) \]
Computation

- It is necessary to compute \(\text{def}(S)\) and \(\text{uFst}(S)\) sets of each basic block.
- This will be done by examining the 3-address codes of a basic block from the beginning.
Computation of $uFst(B)$ and $def(B)$

Following steps are performed for each basic block $B$ with $1 \cdots k$ 3-adders codes of type assignment e.g. $x \leftarrow y \oplus z$.

1. $uFst(B), \; def(B) \leftarrow \emptyset$
2. for $i \leftarrow 1$ to $k$
3. \; \; if $y \not\in def(B)$, then
4. \; \; \; $uFst(B) \leftarrow uFst(B) \cup \{y\}$
5. \; \; if $z \not\in def(B)$, then
6. \; \; \; $uFst(B) \leftarrow uFst(B) \cup \{z\}$
7. \; $def(B) \leftarrow def(B) \cup \{x\}$
Computation of $uFst(B)$ and $def(B)$

- For a 3-address code like `if x < y goto L` there is no change in $def(B)$. But $uFst(B)$ will include $\{x, y\}$ unless they are already in $def(B)$.

- Other type of 3-address codes can be handled similarly.
Computation of $\text{LOut}(B)$

It is similar to the computation of $\text{Dom}(B)$.

1. $\text{LOut}(B_i)$ for $i = 0, \cdots, n - 1$ are initialized to $\emptyset$.

2. All $\text{LOut}(B_i)$’s are recomputed until a fixed point is reached.

3. If the CFG corresponds to a function, the formal parameters are in $\text{LIn}()$ of the entry block.
Basic Block - $b_1$

1: L2: $v_1 = i$
2: $v_2 = j$
3: if $v_1 > v_2$ goto L3

$\text{uFst}(b_1) = \{i, j\}$ and $\text{def}(b_1) = \{v_1, v_2\}$
Basic Block - $b_2$

4: $v_1 = j$

5: $v_2 = i$

6: $v_1 = v_1 - v_2$

7: $j = v_1$

8: goto L4

$uF_{st}(b_2) = \{i, j\}$ and $\text{def}(b_2) = \{v_1, v_2, j\}$
Basic Block - $b_3$

9: L3: $v_1 = i$

10: $v_2 = j$

11: $v_1 = v_1 - v_2$

12: $i = v_1$

$uFst(b_3) = \{i, j\}$ and $def(b_2) = \{v_1, v_2, i\}$
Basic Block - $b_4$

13: L4: v1 = i
14: v2 = j
15: if v1<>v2 goto L2

uFst($b_3$) = \{i, j\} and def($b_2$) = \{v_1, v_2\}
Control-Flow Graph
L2: \( v1 = i \)  
\( v2 = j \)  
if \( v1 > v2 \) goto L3

L4: \( v1 = i \)  
\( v2 = j \)  
if \( v1 \neq v2 \) goto L2

L3: \( v1 = i \)  
\( v2 = j \)  
\( v1 = v1 - v2 \)  
\( i = v1 \)

L4: \( v1 = i \)  
\( v2 = j \)  
if \( v1 \neq v2 \) goto L2

b1: Entry

b2: \( v1 = j \)  
\( v2 = i \)  
\( v1 = v1 - v2 \)  
\( j = v1 \)  
goto L4

b3: L3

b4: L4
### Computation of $LOut(b_i)$

<table>
<thead>
<tr>
<th>$b_i$</th>
<th>Iteration &amp; $LOut(b_i)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$b_i$</td>
<td>0</td>
</tr>
<tr>
<td>$b_1$</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>$b_2$</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>$b_3$</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>$b_4$</td>
<td>$\emptyset$</td>
</tr>
</tbody>
</table>
Note

- The variables \( i, j \) are live at the exit of all four blocks. They may be kept in a register.

- We can detect uninitialized variable through this analysis. If the set of live-variables at the exit of the dummy entry node of a CFG is non-empty, then all variables of the set are uninitialized.
Control-Flow Graph

Ent.

\[
\begin{align*}
u\text{FST} &= \{\} \\
def &= \{\}
\end{align*}
\]

b1

\[
\begin{align*}
&j = 5 \\
u\text{FST} &= \{\} \\
def &= \{j\}
\end{align*}
\]

b2

\[
\begin{align*}
&j = j + i \\
u\text{FST} &= \{i,j\} \\
def &= \{j\}
\end{align*}
\]

Ext.

\[
\begin{align*}
&u\text{FST} = \{} \\
def &= \{}\n\end{align*}
\]
### Computation of $\text{LOut}(b_i)$

<table>
<thead>
<tr>
<th>$b_i$</th>
<th>Iteration &amp; $\text{LOut}(b_i)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ent.</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>$b_1$</td>
<td>$\emptyset$, ${i, j}$</td>
</tr>
<tr>
<td>$b_2$</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>Ext.</td>
<td>$\emptyset$</td>
</tr>
</tbody>
</table>

Indicates that $i$ is uninitialized.
Static Single Assignment: SSA

- SSA is a naming method that encodes both flow of control and flow of data in a program.
- Each name is defined by an operation at a particular point in the code. So each use of a name has a unique definition.
- The flow of control is handled by using a selection function \( \phi \).
Static Single-Assignment: SSA

- SSA representation looks similar to three-address code with two main differences.
- Each name is defined only once, so it is called static single-assignment.
- If the same variable is defined on different control paths, they are renamed as distinct variables by adding subscript to the base name.
Static Single-Assignment: SSA

- When two or more flow-paths join, a \( \phi \)-function is used to select different names of the same variable on these paths.
- A \( \phi \)-function also defines a new name.
- The arguments of \( \phi \)-functions of a join-block are ordered according to some order of incoming flow-paths.
Three-Address & SSA Codes

\[
\begin{align*}
i &= 1 \\
f &= 1 \\
\text{L2: if } i > n & \text{ goto L1} \\
f &= f \times i \\
i &= i + 1 \\
goto \text{ L2}
\end{align*}
\]

\[
\begin{align*}
i0 &= 1 \\
f0 &= 1 \\
\text{if } i0 > n & \text{ goto L1} \\
i1 &= \phi(i0, i2) \\
f1 &= \phi(f0, f2) \\
f2 &= f1 \times i1 \\
i2 &= i1 + 1 \\
\text{if } i2 & \leq n \text{ goto L2} \\
i3 &= \phi(i0, i2) \\
f3 &= \phi(f0, f2)
\end{align*}
\]
Note

- The $\phi$-function is inserted at the beginning of a basic block where different values of a program variable can reach along different flow-paths.

- In our example $(i_0,f_0)$ are copied to $(i_1,f_1)$ when the control flows from the top. Otherwise, $(i_2,f_2)$ is copied to $(i_1,f_1)$. 
• All $\phi$-functions at the beginning of a basic block are assumed to be evaluated concurrently. So the ordering does not matter. But it may complicate the implementation of $\phi$-function.

• The single assignment property simplifies code optimization. As a definition is never killed, the value of a name is available on any path starting from the definition.
Note

- The $\phi$-function can take any number of arguments. The number of arguments depends on the number of control-flow paths entering a join-block.

- So it may have more than 3-addresses and there should be mechanism to accommodate it in 3-address data-structure.
Note

- Some of the arguments of a $\phi$-function may be undefined during its execution. In our example $i_2$ is undefined when the loop is entered first time.

- But it should not create trouble as $\phi$-function selects the argument corresponding to current control-flow path taken to enter the join-block (where the argument is defined).
Building SSA: A Simple Method

- For every variable name $x$ used in the code, a $\phi$-function, $x \leftarrow \phi(y, \cdots, y)$ is inserted at the beginning of each basic block with more than one predecessors.

- The number of arguments of a $\phi$-function is equal to the number of predecessors of the basic block.
The ordering of \( \phi \)-function inserted at the beginning of a basic block does not matter as they are executed concurrently i.e. they select the input parameters simultaneously and write them simultaneously.

But the process may insert many useless \( \phi \)-functions.
**$\phi$-functions**

**Entry**

**L2:**
- $i = \phi(i,i)$
- $j = \phi(j,j)$
- $u = \phi(u,u)$
- $v = \phi(v,v)$
- $u = i$
- $v = j$
- If $u > v$ goto L3

**L3:**
- $u = i$
- $v = j$
- $u = u - v$
- $i = u$

**L2:**
- $u = j$
- $v = i$
- $u = u - v$
- $j = u$
- Goto L4

**L4:**
- $i = \phi(i,i)$
- $j = \phi(j,j)$
- $u = \phi(u,u)$
- $v = \phi(v,v)$
- $u = i$
- $v = j$
- If $u \neq v$ goto L2

**Exit**
After Renaming: SSA

Entry

L2: \( i_2 = \phi (i_0, i_1) \)
    \( j_2 = \phi (j_0, j_1) \)
    \( u_2 = \phi (u_0, u_1) \)
    \( v_2 = \phi (v_0, v_1) \)
    \( u_3 = i_2 \)
    \( v_3 = j_2 \)
    if \( u_3 > v_3 \) goto L3

L3: \( u_6 = i_2 \)
    \( v_5 = j_2 \)
    \( u_7 = u_6 - v_5 \)
    \( i_3 = u_7 \)

L4: \( i_1 = \phi (i_2, i_3) \)
    \( j_1 = \phi (j_3, j_2) \)
    \( u_8 = \phi (u_5, u_7) \)
    \( v_6 = \phi (v_4, v_5) \)
    \( u_1 = i_1 \)
    \( v_1 = j_1 \)
    if \( u_1 <> v_1 \) goto L2

Exit

\( b_1: \)

\( b_2: \)

\( b_3: \)

\( b_4: \)
Both in $b_1$ and $b_3$, the $\phi$-functions for $u$ and $v$ are useless.
Reaching Definitions

• For renaming variables for SSA we need to perform a data-flow analysis called reaching definitions. Each φ-function is also a new definition.

• A definition \( d \) of a name \( v \) \((d : v \leftarrow \cdots)\) reaches a use \( i \) \((i : \cdots \leftarrow \cdots v \cdots)\) if there is a path from \( d \) to \( i \) on which \( v \) is not redefined.
Reaching Definitions

- **Reaches**\( (n) \) is the set of definitions reaches the beginning of the basic block \( n \).

- **Gen**\( (n) \) is the set of set of definitions generated in the basic block \( n \) but not killed within it. So they go out of \( n \).

- **Kill**\( (n) \) is the set of all definitions (globally) killed in the basic block \( n \).
Gen() and Kill()

Kill(b1) = \{d3,d5,d7,d9,d11, d4, d8,d12\}
Gen(b1) = \{d1,d2\}

Kill(b2) = \{d1,d2,d3,d7,d8,d9,d11,d12\}
Gen(b2) = \{d4,d5,d6\}

Kill(b3) = \{d1,d2,d3,d4,d5,d7,d11,d12\}
Gen(b3) = \{d8,d9,d10\}

L2: \begin{align*}
v1 &= i \\
v2 &= j \\
\text{if } v1 > v2 & \text{ goto } L3
\end{align*}

L3: \begin{align*}
v1 &= i \\
v2 &= j \\
v1 &= v1 - v2 \\
i &= v1
\end{align*}

L4: \begin{align*}
v1 &= i \\
v2 &= j \\
\text{if } v1 \neq v2 & \text{ goto } L2
\end{align*}

Kill(b4) = \{d1,d2,d3,d4,d5,d7,d8,d9\}
Gen(b4) = \{d11,d12\}
The reaching definition can be formulated as a forward data-flow problem.

\[
\text{reaches}(n) = \bigcup_{m \in \text{pred}(n)} \text{gen}(m) \cup (\text{reaches}(m) \setminus \text{kill}(m))
\]

We start with \(\text{reaches}(n) = \emptyset\) for all basic block \(n\).
Building SSA: A Simple Method

- The $\phi$-function ensures that only one definition reaches an use.
- Variable in each use are renamed according to the definition that reaches it.
- For each $\phi$-function, names are ordered according to the control-path of their arrival.
The algorithm correctly translates a 3-address code to SSA form. But there are many redundant ϕ-functions e.g. $x_i \leftarrow \phi(x_j, x_j)$ or the name defined may not be live (redefinition of $x$ before use).

- Redundant ϕ-functions increases algorithmic cost.
Note

- There is an algorithm based on the notion of dominator that will not generate useless $\phi$-functions.
- For every block $b_i$ it finds out the blocks that require $\phi$-functions for a definition $d$ in $b_i$. 
• If $b_i \in \text{Dom}(b_j)$ and $d : x \leftarrow \cdots$ is a definition in $b_i$, then $b_j$ does not need a $\phi$-function for $x$ unless it is redefined in some control path that reaches $b_j$.

• If there is a redefinition of $x$ in some intermediate block $b_k$ on a control path between $b_i$ and $b_j$, then the redefinition forces a $\phi$-function.
Forcing a $\phi$-function

A definition $d : x \leftarrow \cdots$ in a block $b_i$ forces a $\phi$-function for $x$ in a block $b_j$ where more than one control-paths meet (join point), in the following cases.

- $b_i \in \text{Dom}(b_k)$, where $b_k$ is a predecessor of $b_j$.
- $b_i = b_j$. We call the set $\text{Dom}(b_j) \setminus \{b_j\}$ as the set of strict dominators of $b_j$. 
In our example CFG, a definition in block $B_2$ forces a $\phi$-function in block $B_9$.

The collection of blocks where the block $b_i$ can force a $\phi$-function is called the dominance frontier of $b_i$, $DF(b_i)$. This is the

$$DF(B_2) = \{B_9\}.$$
Given a block $b_j$ in a CFG, the block $b_i \in \text{DOM}(b_j) \setminus \{b_j\}$, closest to $b_j$, is called the **immediate dominator** of $b_j$, $\text{IDom}(b_j)$.

For every block $b_i$ of a CFG we draw a directed-edge from $\text{IDom}(b_i) \rightarrow b_i$. This forms a tree with the **entry-node** of the CFG as the root. It is known as the **dominator tree** of the CFG.
Dominator Tree

Code Gen Example

Goutam Biswas
Note

- The **dominator tree** of a CFG has all blocks (nodes) of the CFG.
- For a node \( b_i \), \( \text{IDom}(b_j) \) is the parent of \( b_i \) in the tree.
- The **Dom**(\( b_i \)) is the collection of nodes (blocks) on the path from the root (entry node) to \( b_i \) e.g. \( \text{IDom}(B_{10}) = B_8 \), \( \text{Dom}(B_{10}) = \{B_0, B_2, B_8, B_{10}\} \).
Properties: Node in DF

If a node (block) \( b_j \) is in some DF, then

- Multiple control-paths meet at \( b_j \) i.e. \( b_j \) is a join-point.
- For each predecessor \( b_k \), the node \( b_j \in DF(b_k) \). As \( b_j \) is the first join-point after \( b_k \) and \( b_j \) is not dominated by \( b_k \).
- For each predecessor \( b_k \), the node \( b_j \in DF(b_l) \) for all \( b_l \in \text{Dom}(k) \setminus \text{Dom}(j) \).
1. For each block $b_j$ with multiple incoming control paths (join-point) do the following.

2. For each predecessor $b_k$ of $b_j$ do the following.

3. Walk up the dominator tree until some $b_i \in \text{Dom}(b_j)$ is found.

4. For each node $b_l$ in the path of $b_i$ to $b_k$, except $b_i$, put $b_j$ in $\text{DF}(b_l)$. 

**Computing Dominance Frontier**
1 for $i \leftarrow 0$ to $n - 1$ do
2 $\text{DF}(b_i) \leftarrow \emptyset$
3 for $j \leftarrow 0$ to $n - 1$ do
4 \hspace{1em} if join-point($b_j$) then
5 \hspace{2em} for each $b_k \in \text{pred}(b_j)$ do
6 \hspace{3em} $\text{temp} \leftarrow b_k$
7 \hspace{2em} while $\text{temp} \neq \text{IDom}(b_j)$ do
8 \hspace{3em} $\text{DF}(\text{temp}) \leftarrow \text{DF}(\text{temp}) \cup \{b_j\}$
9 \hspace{3em} $\text{temp} \leftarrow \text{IDom}(\text{temp})$
Example

- Blocks/nodes that are **join-points** (multiple control-paths meet):
  \[ B_1, B_2, B_7, B_8, B_9, B_{11}, B_{12}. \]
Dominator Tree

B0

B1

B3

B6

B11

B12

B2

B4

B8

B10

B5
**Dominance Frontier:**

<table>
<thead>
<tr>
<th>Blocks</th>
<th>Predecessors</th>
<th>DF</th>
</tr>
</thead>
<tbody>
<tr>
<td>$B_0$</td>
<td>$\emptyset$</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>$B_1$</td>
<td>$B_0, B_6$</td>
<td>$B_1, B_9, B_{11}$</td>
</tr>
<tr>
<td>$B_2$</td>
<td>$B_0, B_{10}$</td>
<td>$B_2, B_9, B_{11}$</td>
</tr>
<tr>
<td>$B_3$</td>
<td>$B_1$</td>
<td>$B_1, B_7, B_{11}$</td>
</tr>
<tr>
<td>$B_4$</td>
<td>$B_2$</td>
<td>$B_8$</td>
</tr>
<tr>
<td>$B_5$</td>
<td>$B_2$</td>
<td>$B_8$</td>
</tr>
<tr>
<td>$B_6$</td>
<td>$B_3$</td>
<td>$B_1, B_{11}$</td>
</tr>
</tbody>
</table>
### Example

- **Dominance Frontier (cont.):**

<table>
<thead>
<tr>
<th>Blocks</th>
<th>Predecessors</th>
<th>DF</th>
</tr>
</thead>
<tbody>
<tr>
<td>$B_7$</td>
<td>$B_3, B_1$</td>
<td>$B_9$</td>
</tr>
<tr>
<td>$B_8$</td>
<td>$B_4, B_5$</td>
<td>$B_2, B_9, B_{11}$</td>
</tr>
<tr>
<td>$B_9$</td>
<td>$B_7, B_8$</td>
<td>$B_{12}$</td>
</tr>
<tr>
<td>$B_{10}$</td>
<td>$B_8$</td>
<td>$B_2, B_{11}$</td>
</tr>
<tr>
<td>$B_{11}$</td>
<td>$B_6, B_{10}$</td>
<td>$B_{12}$</td>
</tr>
<tr>
<td>$B_{12}$</td>
<td>$B_9, B_{11}$</td>
<td></td>
</tr>
</tbody>
</table>
Example

- The node $B_1$ has two predecessors $B_0, B_6$. As $B_0 = \text{IDom}(B_1)$, the while-loop is not entered. So it does not contribute any node in the dominance frontier. For the other predecessor $B_6$, the loop is entered with temp $\leftarrow B_6, B_3, B_1$, so $B_1$ goes in to $\text{DF}(B_6)$, $\text{DF}(B_3)$, $\text{DF}(B_1)$. 
Placement of $\phi$-function

• In the first algorithm (simple) a $\phi$-function was introduced for every variable $x$ at the beginning of each join-block -
  \[ x \leftarrow \phi(x, \cdots, x). \]

• In the first modification, for every $x$ defined in a block $b_i$, we introduce a $\phi$-function at the beginning of each $b_j \in DF(b_i)$. 
Placement of $\phi$-function

- If the life span of a variable is restricted to a block, no $\phi$-function is required for it. Only the names ‘global’ to a CFG may require $\phi$-functions.

- The union of $uFst(B)$, the set of variables whose values are used before any definition in block $B$, is the set of ‘global’ names.
Algorithm for Global Names

- Following algorithm computes global names, glovVar and
- \( \text{varBlkLst}(x) \) for the list of blocks where the variable \( x \) is defined.
Algorithm

1. globVar ← ∅
2. for all variable \( x \), varBlkLst(\( x \)) ← ∅
3. for each block \( B \)
   4. uFst(\( B \)) ← ∅
   5. for each 3-address code \( a ← b \odot c \) from beginning
      6. if \( b \notin \text{def}(B) \) then globVar ← globVar ∪ \{b\}
      7. if \( c \notin \text{def}(B) \) then globVar ← globVar ∪ \{c\}
      8. def(B) ← def(B) ∪ \{a\}
      9. varBlkLst(\( a \)) ← varBlkLst(\( a \)) ∪ \{B\}
Algorithm for Inserting $\phi$-function

- For each global name $x \in \text{globVar}$ there is a list of blocks ($\text{varBlkLst}(x)$) where $x$ is defined.
- For each $b_i \in \text{varBlkLst}(x)$ a $\phi$-function for $x$ is inserted in every block $b_j \in \text{DF}(b_i)$.
- But then every $\phi$-function in some block $b_j$ is also a definition of $x$ and gives rise to new $\phi$-functions in $b_k \in \text{DF}(b_j)$. 
Algorithm

1. for each $x \in \text{globVar}$
2. \hspace{1cm} $\text{curBlkLst} \leftarrow \text{varBlkLst}(x)$
3. \hspace{1cm} for each $b_i \in \text{curBlkLst}$
4. \hspace{2cm} for each $b_j \in \text{DF}(b_i)$
5. \hspace{3cm} if $b_j$ does not have $\phi(x, \cdots, x)$ insert one
6. \hspace{1cm} $\text{curBlkLst} \leftarrow \text{curBlkLst} \cup \{b_j\}$
It is necessary to put more ‘flesh’ in the basic blocks of the example CFG.

<table>
<thead>
<tr>
<th>Block</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>(B_0)</td>
<td>(a \leftarrow \cdots)</td>
</tr>
<tr>
<td></td>
<td>(b \leftarrow \cdots)</td>
</tr>
<tr>
<td>(B_1)</td>
<td>(b \leftarrow \cdots)</td>
</tr>
<tr>
<td></td>
<td>(c \leftarrow \cdots)</td>
</tr>
<tr>
<td>(B_2)</td>
<td>(a \leftarrow \cdots)</td>
</tr>
<tr>
<td></td>
<td>(d \leftarrow \cdots)</td>
</tr>
<tr>
<td>(B_3)</td>
<td>(d \leftarrow \cdots)</td>
</tr>
<tr>
<td></td>
<td>((a &gt; c) : 3, 7))</td>
</tr>
<tr>
<td>(B_4)</td>
<td>(x \leftarrow \cdots)</td>
</tr>
<tr>
<td></td>
<td>(x \leftarrow a - 1)</td>
</tr>
<tr>
<td>(B_5)</td>
<td>(x \leftarrow \cdots)</td>
</tr>
<tr>
<td></td>
<td>((d \leq b) : 4, 5)</td>
</tr>
<tr>
<td>(B_6)</td>
<td>(y \leftarrow \cdots)</td>
</tr>
<tr>
<td></td>
<td>(b \leftarrow \cdots)</td>
</tr>
<tr>
<td>(B_7)</td>
<td>(y \leftarrow \cdots)</td>
</tr>
<tr>
<td></td>
<td>((y = b) : 1, 11)</td>
</tr>
<tr>
<td>(B_8)</td>
<td>(b \leftarrow \cdots)</td>
</tr>
<tr>
<td></td>
<td>((y = b) : 9, 10)</td>
</tr>
</tbody>
</table>
### Example

<table>
<thead>
<tr>
<th>Block</th>
<th>Equation</th>
<th>Equation</th>
<th>Condition</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$B_9$</td>
<td>$z \leftarrow b + d$</td>
<td>$a \leftarrow z + d$</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>$c \leftarrow a + b$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$B_{10}$</td>
<td>$d \leftarrow d + 1$</td>
<td></td>
<td>$(d &lt; y)$</td>
<td>2, 11</td>
</tr>
<tr>
<td>$B_{11}$</td>
<td>$a \leftarrow b + y$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>$b \leftarrow d + x$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
• **Global names:** $a, b, c^a, d, x, y$ but $z$ is local to $B_9$.

• **List of blocks where a variable is defined are:**

<table>
<thead>
<tr>
<th>Var</th>
<th>$a$</th>
<th>$b$</th>
<th>$c$</th>
<th>$d$</th>
<th>$x$</th>
<th>$y$</th>
<th>$z$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Blk</td>
<td>$B_0, B_2$</td>
<td>$B_0, B_1$</td>
<td>$B_1$</td>
<td>$B_2$</td>
<td>$B_3$</td>
<td>$B_6$</td>
<td>$B_9$</td>
</tr>
<tr>
<td></td>
<td>$B_7, B_9$</td>
<td>$B_6, B_7$</td>
<td>$B_9$</td>
<td>$B_3$</td>
<td>$B_4$</td>
<td>$B_8$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$B_{11}$</td>
<td>$B_8, B_{11}$</td>
<td>$B_{10}$</td>
<td>$B_5$</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

---

[a] It is pointed out by Ashray Sudhir that $c$ is not a global name.
Placements of $\phi$-function

- Consider the variable $a$. It is defined in $B_0, B_2, B_7, B_9, B_{11}$.
- These definitions induce $\phi$-functions for $a$ in the blocks

$$\text{DF}(B_0) \cup \text{DF}(B_2) \cup \text{DF}(B_7) \cup \text{DF}(B_9) \cup \text{DF}(B_{11})$$

$$= \emptyset \cup \{B_2, B_9, B_{11}\} \cup \{B_9\} \cup \{B_{12}\} \cup \{B_{12}\}$$

$$= \{B_2, B_9, B_{11}, B_{12}\}.$$  

As $B_{12}$ is exit block, we ignore it.
Placement of $\phi$-function

- As $\{B_2, B_9, B_{11}\} \subseteq \text{varBlkLst}(a)$, no new block is added in the $\text{curBlkLst}$.  

- After introducing $a \leftarrow \phi(a, a)$ blocks are

<table>
<thead>
<tr>
<th>Block</th>
<th>Instruction</th>
<th>Block</th>
<th>Instruction</th>
<th>Block</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>$B_2$</td>
<td>$a \leftarrow \phi(a, a)$</td>
<td>$B_9$</td>
<td>$z \leftarrow b + d$</td>
<td>$B_{11}$</td>
<td>$a \leftarrow b + y$</td>
</tr>
<tr>
<td></td>
<td>$a \leftarrow \cdots$</td>
<td></td>
<td>$a \leftarrow z + d$</td>
<td></td>
<td>$b \leftarrow d + x$</td>
</tr>
<tr>
<td></td>
<td>$d \leftarrow \cdots$</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>$(d \leq b): 4, 5$</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Code Gen Example
Note

- There are redundant $\phi$-functions in $B_9$ and $B_{11}$ as $a$ is not live in these two blocks.
- The live-variable analysis result may be used to detect that $a$ is not live at the beginning of $B_9$ and $B_{11}$. It can be done at step-5 of $\phi$-function insertion algorithm.
Placement of \( \phi \)-function

So blocks \( B_2, B_9, B_{11} \) are

<table>
<thead>
<tr>
<th>( B_2 )</th>
<th>( B_9 )</th>
<th>( B_{11} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( a \leftarrow \phi(a, a) )</td>
<td>( z \leftarrow b + d )</td>
<td>( a \leftarrow b + y )</td>
</tr>
<tr>
<td>( a \leftarrow \cdots )</td>
<td>( a \leftarrow z + d )</td>
<td>( b \leftarrow d + x )</td>
</tr>
<tr>
<td>( d \leftarrow \cdots )</td>
<td>( c \leftarrow a + b )</td>
<td></td>
</tr>
<tr>
<td>( (d \leq b) : 4, 5 )</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Placement of $\phi$-function

- Consider the variable $b$. It is defined in $B_0, B_1, B_6, B_7, B_8, B_{11}$.
- $\phi$-functions for $b$ are introduced in

$$DF(B_0) \cup DF(B_1) \cup DF(B_6) \cup DF(B_7) \cup DF(B_8) \cup DF(B_{11})$$

$$= \emptyset \cup \{B_1, B_9, B_{11}\} \cup \{B_1, B_{11}\} \cup \{B_9\} \cup \{B_2, B_9, B_{11}\} \cup \{B_{12}\}$$

$$= \{B_1, B_2, B_9, B_{11}, B_{12}\}.$$
Placement of $\phi$-function

- The block $B_2$ is added in the `curBlkLst`.
- $DF(B_2) = \{B_2, B_9, B_{11}\}$ are added in the list of blocks where $b \leftarrow \phi(b, b)$ are introduced.
- Final set of blocks are $\{B_1, B_2, B_9, B_{11}\}$.
Blocks $B_1, B_2, B_9, B_{11}$ after insertion of $b \leftarrow \phi(b, b)$ are

<table>
<thead>
<tr>
<th>Block</th>
<th>Code</th>
</tr>
</thead>
</table>
| $B_1$ : | $b \leftarrow \phi(b, b)$
| | $b \leftarrow \cdots$
| | $c \leftarrow \cdots$
| | $(a > c) : 3, 7$
| $B_{11}$ : | $b \leftarrow \phi(b, b)$
| | $a \leftarrow b + y$
| | $b \leftarrow d + x$
| $B_2$ : | $b \leftarrow \phi(b, b)$
| | $a \leftarrow \cdots$
| | $d \leftarrow \cdots$
| | $(d \geq b) : 4, 5$
| $B_9$ : | $b \leftarrow \phi(b, b)$
| | $z \leftarrow b + d$
| | $a \leftarrow z + d$
| | $c \leftarrow a + b$
Placement of $\phi$-function

- The variable $c$ introduces $\phi$-function in $B_1, B_9, B_{11}$, all are useless, but the given algorithm cannot detect that.

- The variable $d$ introduces $\phi$-function in $B_1, B_2, B_7, B_9, B_{11}$. But it is redundant in $B_1, B_7$ that our algorithm cannot detect.
Placement of $\phi$-function

- The variable $x$ introduces $\phi$-function in $B_1, B_2, B_7, B_8, B_9, B_{11}$. Again they are redundant in $B_1$ and $B_7$.

- The variable $y$ introduces $\phi$-function in $B_1, B_2, B_9, B_{11}$ out of which $B_1, B_2$ and $B_9$ are redundant.
Renaming of Variables

- Each **global** name is treated as a base of name in each new definition of it e.g. $x$.
- For each definition it is differentiated using **subscripts** e.g. $x_0, x_1, \ldots$.
- One algorithm for renaming traverse the **dominator tree** in pre-order. It works as follows.
Renaming of Variables

- Each variable defined by the \( \phi \)-function at the beginning of the block \((b)\) are renamed first. After that the 3-address codes of \( b \) are visited in order.

- If the 3-address code is \( x \leftarrow y \odot z \) and the current indices of \( x, y, z \) are \( i, j, k \) respectively, then after renaming it will be \( x_{i+1} \leftarrow y_j \odot z_k \) and the current indices are modified to \( i+1, j, k \).
Renaming of Variables

- After renaming the variables in the 3-address codes of the block $b$, appropriate arguments of the $\phi$-functions in the CFG-successors of $b$ are renamed.
- Then the renaming is recursively called on each children block of $b$ in the dominator tree.
Renaming of Variables

- On return from the recursive of renaming on block $b$, the state of name indices is restored to the value that existed before entering the block.

- It is necessary to maintain a stack of indices for all global names.
Data Structure

- For each global name we use a separate stack. This requires a second pass over the block \((b)\) at the end of the recursive call to restore the status.

- This reduces the total memory requirement for the stack.

- We also may use a counter that holds the current indices of global names.
Algorithm

1. for each $x \in \text{globVar}$
2. \hspace{1em} $\text{cInd}[x] \leftarrow 0$
3. \hspace{1em} $\text{stack}[x] \leftarrow \emptyset$
4. \hspace{1em} $\text{rename}(B_0)$
sub(x)\[sub(x)\]

Returns the current subscript for the global name \(x\), pushes it in the stack of \(x\) and increments it.

\[
\begin{align*}
\text{sub}(x) \\
1 & \quad i \leftarrow cInd[x] \\
2 & \quad \text{push}(\text{stack}[x], i) \\
3 & \quad cInd \leftarrow cInd + 1 \\
4 & \quad \text{return } i
\end{align*}
\]
1. For each $\phi$-function $x \leftarrow \phi(\cdots)$ of block $b$ rewrite $x$ by $x_{\text{sub}(x)}$.

2. For each 3-address code in order where $y$ is an operand change it to $y_i$, where $i = \text{top}(\text{stack}[y])$.

3. If $x$ is a target of a 3-address code i.e. $x \leftarrow \cdots$, it is replaced by $x_{\text{sub}(x)}$. 
rename(b)

4. For each successor block of $b$ in the CFG, initialize the appropriate parameters of $\phi$-functions.

5. Recursively call rename() on the dominoator tree children of $b$.

6. For each definition of $x \leftarrow \cdots$ in block $b$, pop(stack[$x$]).
Example

In our example we have six global names - globVar = \{a, b, c, d, x, y\}. Initial cInd and stack[x] are

<table>
<thead>
<tr>
<th></th>
<th>cInd</th>
<th>stack[x]</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>0</td>
<td>⊥</td>
</tr>
<tr>
<td>b</td>
<td>0</td>
<td>⊥</td>
</tr>
<tr>
<td>c</td>
<td>0</td>
<td>⊥</td>
</tr>
<tr>
<td>d</td>
<td>0</td>
<td>⊥</td>
</tr>
<tr>
<td>x</td>
<td>0</td>
<td>⊥</td>
</tr>
<tr>
<td>y</td>
<td>0</td>
<td>⊥</td>
</tr>
</tbody>
</table>
rename($B_0$)

<table>
<thead>
<tr>
<th>$u$</th>
<th>cInd</th>
<th>stack[$u$]</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a$</td>
<td>1</td>
<td>$a_0$</td>
</tr>
<tr>
<td>$b$</td>
<td>1</td>
<td>$b_0$</td>
</tr>
<tr>
<td>$c$</td>
<td>0</td>
<td>$\bot$</td>
</tr>
<tr>
<td>$d$</td>
<td>0</td>
<td>$\bot$</td>
</tr>
<tr>
<td>$x$</td>
<td>0</td>
<td>$\bot$</td>
</tr>
<tr>
<td>$y$</td>
<td>0</td>
<td>$\bot$</td>
</tr>
</tbody>
</table>

Old $B_0$: $a \leftarrow \cdots$

New $B_0$: $a_0 \leftarrow \cdots$

$b \leftarrow \cdots$

$b_0 \leftarrow \cdots$

Modified
### CFG Successor of $B_0$

<table>
<thead>
<tr>
<th>$B_1$</th>
<th>$B_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>( b \leftarrow \phi(b, b_0) )</td>
<td>( a \leftarrow \phi(a_0, a) )</td>
</tr>
<tr>
<td>( c \leftarrow \phi(c, \bot) )</td>
<td>( b \leftarrow \phi(b_0, b) )</td>
</tr>
<tr>
<td>( d \leftarrow \phi(d, \bot) )</td>
<td>( d \leftarrow \phi(\bot, d) )</td>
</tr>
<tr>
<td>( x \leftarrow \phi(x, \bot) )</td>
<td>( x \leftarrow \phi(\bot, x) )</td>
</tr>
<tr>
<td>( y \leftarrow \phi(y, \bot) )</td>
<td>( y \leftarrow \phi(\bot, y) )</td>
</tr>
<tr>
<td>( b \leftarrow \ldots )</td>
<td>( a \leftarrow \ldots )</td>
</tr>
<tr>
<td>( c \leftarrow \ldots )</td>
<td>( d \leftarrow \ldots )</td>
</tr>
<tr>
<td>( (a &gt; c) : 3, 7 )</td>
<td>( (d \geq b) : 4, 5 )</td>
</tr>
</tbody>
</table>
rename($B_1$): DT Successor of $B_0$

$$B_1: \quad b_1 \leftarrow \phi(b, b_0)$$

$$c_0 \leftarrow \phi(c, \bot)$$

$$d_0 \leftarrow \phi(d, \bot)$$

$$x_0 \leftarrow \phi(x, \bot)$$

$$y_0 \leftarrow \phi(y, \bot)$$

$$b_2 \leftarrow \cdots$$

$$c_1 \leftarrow \cdots$$

$$(a_0 > c_1): 3, 7$$
### Modified cInd and stack[$u$]

<table>
<thead>
<tr>
<th>$u$</th>
<th>cInd</th>
<th>stack[$u$]</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a$</td>
<td>1</td>
<td>$a_0$</td>
</tr>
<tr>
<td>$b$</td>
<td>3</td>
<td>$b_0 b_1 b_2$</td>
</tr>
<tr>
<td>$c$</td>
<td>2</td>
<td>$c_0 c_1$</td>
</tr>
<tr>
<td>$d$</td>
<td>1</td>
<td>$d_0$</td>
</tr>
<tr>
<td>$x$</td>
<td>1</td>
<td>$x_0$</td>
</tr>
<tr>
<td>$y$</td>
<td>1</td>
<td>$y_0$</td>
</tr>
</tbody>
</table>
CFG Successor of $B_1$

<table>
<thead>
<tr>
<th>$B_3$</th>
<th>$B_7$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$d \leftarrow \cdots$</td>
<td>$d \leftarrow \phi(d, d_0)$</td>
</tr>
<tr>
<td>$x \leftarrow \cdots$</td>
<td>$x \leftarrow \phi(x, x_0)$</td>
</tr>
<tr>
<td></td>
<td>$a \leftarrow \cdots$</td>
</tr>
<tr>
<td></td>
<td>$b \leftarrow \cdots$</td>
</tr>
</tbody>
</table>
rename($B_3$): DT Successor of $B_1$

\[ B_3 : \quad d_1 \leftarrow \cdots \\
\quad x_1 \leftarrow \cdots \]
## Modified cInd and stack\([u]\)

<table>
<thead>
<tr>
<th>(u)</th>
<th>cInd</th>
<th>stack([u])</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a)</td>
<td>1</td>
<td>(a_0)</td>
</tr>
<tr>
<td>(b)</td>
<td>3</td>
<td>(b_0\ b_1\ b_2)</td>
</tr>
<tr>
<td>(c)</td>
<td>2</td>
<td>(c_0\ c_1)</td>
</tr>
<tr>
<td>(d)</td>
<td>2</td>
<td>(d_0\ d_1)</td>
</tr>
<tr>
<td>(x)</td>
<td>2</td>
<td>(x_0\ x_1)</td>
</tr>
<tr>
<td>(y)</td>
<td>1</td>
<td>(y_0)</td>
</tr>
</tbody>
</table>
**CFG Successor of $B_3$**

<table>
<thead>
<tr>
<th>$B_6$</th>
<th>$B_7$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$y \leftarrow \cdots$</td>
<td>$d \leftarrow \phi(d_1, d_0)$</td>
</tr>
<tr>
<td>$b \leftarrow \cdots$</td>
<td>$x \leftarrow \phi(x_1, x_0)$</td>
</tr>
<tr>
<td></td>
<td>$a \leftarrow \cdots$</td>
</tr>
<tr>
<td></td>
<td>$b \leftarrow \cdots$</td>
</tr>
</tbody>
</table>
rename($B_6$): DT Successor of $B_3$

$B_6: \ y_1 \leftarrow \cdots$

$\ b_3 \leftarrow \cdots$
## Modified cInd and stack[\(u\)]

<table>
<thead>
<tr>
<th>(u)</th>
<th>cInd</th>
<th>stack[(u)]</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a)</td>
<td>1</td>
<td>(a_0)</td>
</tr>
<tr>
<td>(b)</td>
<td>4</td>
<td>(b_0\ b_1\ b_2\ b_3)</td>
</tr>
<tr>
<td>(c)</td>
<td>2</td>
<td>(c_0\ c_1)</td>
</tr>
<tr>
<td>(d)</td>
<td>2</td>
<td>(d_0\ d_1)</td>
</tr>
<tr>
<td>(x)</td>
<td>2</td>
<td>(x_0\ x_1)</td>
</tr>
<tr>
<td>(y)</td>
<td>2</td>
<td>(y_0\ y_1)</td>
</tr>
</tbody>
</table>
CFG Successor of $B_6$

<table>
<thead>
<tr>
<th>$B_1$</th>
<th>$B_{11}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$b_1 \leftarrow \phi(b_3, b_0)$</td>
<td>$a \leftarrow \phi(a_0, a)$</td>
</tr>
<tr>
<td>$c_0 \leftarrow \phi(c_1, \bot)$</td>
<td>$b \leftarrow \phi(b_3, b)$</td>
</tr>
<tr>
<td>$d_0 \leftarrow \phi(d_1, \bot)$</td>
<td>$c \leftarrow \phi(c_1, c)$</td>
</tr>
<tr>
<td>$x_0 \leftarrow \phi(x_1, \bot)$</td>
<td>$d \leftarrow \phi(d_1, d)$</td>
</tr>
<tr>
<td>$y_0 \leftarrow \phi(y_1, \bot)$</td>
<td>$x \leftarrow \phi(x_1, x)$</td>
</tr>
<tr>
<td>$b_2 \leftarrow \cdots$</td>
<td>$y \leftarrow \phi(y_1, y)$</td>
</tr>
<tr>
<td>$c_1 \leftarrow \cdots$</td>
<td>$a \leftarrow b + y$</td>
</tr>
<tr>
<td>$(a_0 &gt; c_1) : 3, 7$</td>
<td>$b \leftarrow d + x$</td>
</tr>
</tbody>
</table>
Return from rename($B_6$)

- There is no children of $B_6$ in the dominator tree. So the stack entries corresponding to $B_6$ are popped.
- Again $B_6$ is the only child of $B_3$ in the dominator tree.
- So the stack entries corresponding to $B_3$ are also popped.
### stack\([u]\) after pop-\(B_6\)

<table>
<thead>
<tr>
<th>(u)</th>
<th>cInd</th>
<th>stack(u)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a)</td>
<td>1</td>
<td>(a_0)</td>
</tr>
<tr>
<td>(b)</td>
<td>4</td>
<td>(b_0\ b_1\ b_2)</td>
</tr>
<tr>
<td>(c)</td>
<td>2</td>
<td>(c_0\ c_1)</td>
</tr>
<tr>
<td>(d)</td>
<td>2</td>
<td>(d_0\ d_1)</td>
</tr>
<tr>
<td>(x)</td>
<td>2</td>
<td>(x_0\ x_1)</td>
</tr>
<tr>
<td>(y)</td>
<td>2</td>
<td>(y_0)</td>
</tr>
</tbody>
</table>

\(b_3\) and \(y_1\) removed.
<table>
<thead>
<tr>
<th>( u )</th>
<th>cInd</th>
<th>stack[( u )]</th>
</tr>
</thead>
<tbody>
<tr>
<td>( a )</td>
<td>1</td>
<td>( a_0 )</td>
</tr>
<tr>
<td>( b )</td>
<td>4</td>
<td>( b_0 \ b_1 \ b_2 )</td>
</tr>
<tr>
<td>( c )</td>
<td>2</td>
<td>( c_0 \ c_1 )</td>
</tr>
<tr>
<td>( d )</td>
<td>2</td>
<td>( d_0 )</td>
</tr>
<tr>
<td>( x )</td>
<td>2</td>
<td>( x_0 )</td>
</tr>
<tr>
<td>( y )</td>
<td>2</td>
<td>( y_0 )</td>
</tr>
</tbody>
</table>

\( d_1 \) and \( x_1 \) removed.
Next step is to rename $B_7$. But \ldots

It is necessary to translate SSA form to 3-address code.
Removing $\phi$-function

- Let in a block $b$ there is $x_i \leftarrow \phi(x_j, x_k)$.

- The value is $x_j$ when the control-path to $b$ is from the block $b_u$; and it is $x_k$ when the control-path to $b$ is from the block $b_v$.

- $x_i \leftarrow \phi(x_j, x_k)$ in block $b$ is replaced by inserting $x_i \leftarrow x_j$ in $b_u$ and $x_i \leftarrow x_k$ in $b_v$ (at the end).
Example: $B_1$ and Two Predecessors

<table>
<thead>
<tr>
<th>$B_1$</th>
<th>$B_6$</th>
<th>$B_0$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$b_1 \leftarrow \phi(b_3, b_0)$</td>
<td>$y_1 \leftarrow \cdots$</td>
<td>$a_0 \leftarrow \cdots$</td>
</tr>
<tr>
<td>$c_0 \leftarrow \phi(c_1, \perp)$</td>
<td>$b_3 \leftarrow \cdots$</td>
<td>$b_0 \leftarrow \cdots$</td>
</tr>
<tr>
<td>$d_0 \leftarrow \phi(d_1, \perp)$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$x_0 \leftarrow \phi(x_1, \perp)$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$y_0 \leftarrow \phi(y_1, \perp)$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$b_2 \leftarrow \cdots$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$c_1 \leftarrow \cdots$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$(a_0 &gt; c_1): 3, 7$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Example: Replacing $\phi$ from $B_1$

<table>
<thead>
<tr>
<th>$B_1$</th>
<th>$B_6$</th>
<th>$B_0$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$b_2 \leftarrow \cdots$</td>
<td>$y_1 \leftarrow \cdots$</td>
<td>$a_0 \leftarrow \cdots$</td>
</tr>
<tr>
<td>$c_1 \leftarrow \cdots$</td>
<td>$b_3 \leftarrow \cdots$</td>
<td>$b_0 \leftarrow \cdots$</td>
</tr>
<tr>
<td>$(a_0 &gt; c_1) : 3, 7$</td>
<td>$b_1 \leftarrow b_3$</td>
<td>$b_1 \leftarrow b_0$</td>
</tr>
<tr>
<td></td>
<td>$c_0 \leftarrow c_1$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$d_0 \leftarrow d_1$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$x_0 \leftarrow x_1$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$y_0 \leftarrow y_1$</td>
<td></td>
</tr>
</tbody>
</table>
References