Def-Use chains are expensive

```c
... switch (i) {
    case 0: x=3;
    case 1: x=1;
    case 2: x=6;
    case 3: x=7;
    default: x = 11;
}

switch (j) {
    case 0: y=x+7;
    case 1: y=x+4;
    case 2: y=x-2;
    case 3: y=x+1;
    default: y=x+9;
}
...```

Previously... Def-Use Chains

```c
for (i=0; i++<10) {
    ...
}
for (i=j; i++<20) {
    ...
}
```

Def-Use chains are expensive

```c
In general, N defs
M uses
⇒ O(NM) space and time
A solution is to limit each var
  to ONE def site```

```
... switch (i) {
    case 0: x=3; break;
    case 1: x=1; break;
    case 2: x=6; break;
    case 3: x=7; break;
    default: x = 11;
}

switch (j) {
    case 0: y=x+7;
    case 1: y=x+4;
    case 2: y=x-2;
    case 3: y=x+1;
    default: y=x+9;
}
...```
Def-Use chains are expensive

```java
... switch (i) {
    case 0: x=3; break;
    case 1: x=1; break;
    case 2: x=6; break;
    case 3: x=7; break;
    default: x = 11;
}
// x1 is one of the above x's
switch (j) {
    case 0: y=x1+7; break;
    case 1: y=x1+4; break;
    case 2: y=x1-2; break;
    case 3: y=x1+1; break;
    default: y=x1+9; break;
}
...}
```

A solution is to limit each var to ONE def site

SSA

- Static single assignment is an IR where every variable is assigned a value *at most once*

Advantages of SSA

- Makes du-chains explicit
  - every definition knows its uses
  - every use knows its *single* definition
- Makes dataflow optimizations
  - easier
  - faster
- For most optimizations reduces space/time requirements

Simple SSA Optimizations

- Dead Code Elimination
  - a definition with no uses (and no ????)

- Constant Propagation
  - a use with a constant definition
SSA History

- Developed by Wegman, Zadeck, Alpern, and Rosen in 1988
  - and improved by Cytron, Ferrante, Wegman, and Zadeck in 1989

- New to gcc 4.0, used in ORC, used in both IBM and Sun Java JIT compilers

Converting to SSA

- Easy for a basic block:
  - assign to a fresh variable at each stmt.
  - Each use uses the most recently defined var.

  \[
  \begin{align*}
  a &\leftarrow x + y \\
  b &\leftarrow a + x \\
  a &\leftarrow b + 2 \\
  c &\leftarrow y + 1 \\
  a &\leftarrow c + a \\
  \end{align*}
  \]

Converting to SSA

- Easy for a basic block:
  - assign to a fresh variable at each stmt.
  - Each use uses the most recently defined var.

  \[
  \begin{align*}
  a &\leftarrow x + y \\
  b &\leftarrow a + x \\
  a &\leftarrow b + 2 \\
  c &\leftarrow y + 1 \\
  a &\leftarrow c + a \\
  \end{align*}
  \]

What about at joins in the CFG?

Merging at Joins

- Use a notional fiction: A $\Phi$ function
Merging at Joins

\[
\begin{align*}
c_1 &\leftarrow 12 \\
\text{if (i)} &
\end{align*}
\]

\[
\begin{align*}
a_1 &\leftarrow x + y \\
b_1 &\leftarrow a_1 + x \\
a_2 &\leftarrow b + 2 \\
c_2 &\leftarrow y + 1 \\
a_3 &\leftarrow \Phi(a_1, a_2) \\
c_3 &\leftarrow \Phi(c_1, c_2) \\
b_2 &\leftarrow \Phi(b_1, \ ?) \\
a_4 &\leftarrow c_3 + a_3
\end{align*}
\]

The \( \Phi \) function

- \( \Phi \) merges multiple definitions along multiple control paths into a single definition
- At a BB with \( p \) predecessors, there are \( p \) arguments to the \( \Phi \) function
  \[
  x_{\text{new}} \leftarrow \Phi(x_1, x_2, x_3, \ldots, x_p)
  \]
- How does \( \phi \) choose which \( x_i \) to use?
  - We don’t really care!
  - If we care, use moves on each incoming edge

“Implementing” \( \Phi \)

\[
\begin{align*}
c_1 &\leftarrow 12 \\
\text{if (i)} &
\end{align*}
\]

\[
\begin{align*}
a_1 &\leftarrow x + y \\
b_1 &\leftarrow a_1 + x \\
a_3 &\leftarrow a_1 \\
c_1 &\leftarrow c_1 \\
a_3 &\leftarrow \Phi(a_1, a_2) \\
c_3 &\leftarrow \Phi(c_1, c_2) \\
a_4 &\leftarrow c_3 + a_3
\end{align*}
\]

Trivial SSA

- Each assignment generates a fresh variable
- At each join point insert \( \Phi \) functions for all live variables

\[
\begin{align*}
x &\leftarrow 1 \\
y &\leftarrow 2 \\
x_1 &\leftarrow x_1 \\
y_2 &\leftarrow 2 \\
x_2 &\leftarrow \Phi(x_1, x_2) \\
y_2 &\leftarrow \Phi(y_1, y_2)
\end{align*}
\]

Way too many \( \Phi \) functions inserted.
**Minimal SSA**

- Each assignment generates a fresh variable.
- At each join point insert $\Phi$ functions for all variables with multiple outstanding defs.

\[
x \leftarrow 1
\]
\[
y \leftarrow x
\]
\[
y \leftarrow 2
\]
\[
z \leftarrow y + x
\]

\[
x_1 \leftarrow 1
\]
\[
y_1 \leftarrow x_1
\]
\[
y_2 \leftarrow 2
\]
\[
y_3 \leftarrow \Phi(y_1, y_2)
\]
\[
z_1 \leftarrow y_3 + x_1
\]

**Another Example**

\[
a \leftarrow 0
\]
\[
b \leftarrow a + 1
\]
\[
c \leftarrow c + b
\]
\[
a \leftarrow b \times 2
\]
\[
a < N
\]
\[
\text{return } c
\]

**Another Example**

\[
a \leftarrow 0
\]
\[
b \leftarrow a + 1
\]
\[
c \leftarrow c + b
\]
\[
a \leftarrow b \times 2
\]
\[
a < N
\]
\[
\text{return } c
\]

Notice use of $c_1$, entry block has implicit def of all variables.

**When do we insert $\Phi$?**

If there is a def of $a$ in block 5, which nodes need a $\Phi()$?

Alternatively, which nodes don’t need a $\Phi()$?

[CFG Diagram]
When do we insert $\Phi$?

- We insert a $\Phi$ function for variable $a$ in block $Z$ iff:
  - There exist blocks $X$ and $Y$, $X \neq Y$, such that $a$ is defined in $X$ and $Y$
  - $a$ is defined in more than one block
  - There exists a non-empty path from $X$ to $Z$, $P_{XZ}$, and a non-empty path from $Y$ to $Z$, $P_{YZ}$ s.t.
    - $P_{XZ} \cap P_{YZ} = \{ Z \}$
    - paths $P_{XZ}$ and $P_{YZ}$ have no nodes in common except for $Z$
  - $Z \notin P_{XZ} \cup P_{YZ}$ where $P_{XZ} = P_{X0} \rightarrow Z$ and $P_{YZ} = P_{Y0} \rightarrow Z$
    - if $Z$ is contained elsewhere in $P_{XZ}$ before the end, it is only found at the end of $P_{YZ}$ and vice versa

This is the path-convergence criterion.

Dominance Property of SSA

- In SSA definitions dominate uses.
  - If $x_i$ is used in $x \leftarrow \Phi(..., x_i, ...)$, then $BB(x_i)$ dominates $i$th pred of $BB(\Phi)$
  - If $x$ is used in $y \leftarrow ... x ...$, then $BB(x)$ dominates $BB(y)$
- We can use dominance information to get an efficient algorithm for converting to SSA.

Definitions

- $a sdom b$
  - If $a$ and $b$ are different blocks and $a dom b$, we say that $a$ strictly dominates $b$
- $a idom b$
  - If $a sdom b$, and there is no $c$ such that $a sdom c$ and $c sdom b$, we say that $a$ is the immediate dominator of $b$
Properties of Dom

Dominance is a partial order on the blocks of the flow graph, i.e.,
1. Reflexivity: \( a \ dom a \) for all \( a \)
2. Anti-symmetry: \( a \ dom b \) and \( b \ dom a \) implies \( a = b \)
3. Transitivity: \( a \ dom b \) and \( b \ dom c \) implies \( a \ dom c \)

NOTE: there may be blocks \( a \) and \( b \) such that neither \( a \ dom b \) or \( b \ dom a \) holds.

The dominators of each node \( n \) are linearly ordered by the \( \text{dom} \) relation. The dominators of \( n \) appear in this linear order on any path from the initial node to \( n \).

Computing dominators

We want to compute \( D[n] \), the set of blocks that dominate \( n \)

Initialize each \( D[n] \) (except \( D[\text{entry}] \)) to be the set of all blocks, and then iterate until no \( D[n] \) changes:

\[
D[\text{entry}] = \{\text{entry}\}
\]

\[
D[n] = \{n\} \cup \left( \bigcap_{p \in \text{pred}(n)} D[p] \right), \quad \text{for } n \neq \text{entry}
\]

Example

Initialization

<table>
<thead>
<tr>
<th>block</th>
<th>( D[n] )</th>
</tr>
</thead>
<tbody>
<tr>
<td>entry</td>
<td>{entry}</td>
</tr>
<tr>
<td>0</td>
<td>{entry,0,1,2,3,4,5,exit}</td>
</tr>
<tr>
<td>1</td>
<td>{entry,0,1,2,3,4,5,exit}</td>
</tr>
<tr>
<td>2</td>
<td>{entry,0,1,2,3,4,5,exit}</td>
</tr>
<tr>
<td>3</td>
<td>{entry,0,1,2,3,4,5,exit}</td>
</tr>
<tr>
<td>4</td>
<td>{entry,0,1,2,3,4,5,exit}</td>
</tr>
<tr>
<td>5</td>
<td>{entry,0,1,2,3,4,5,exit}</td>
</tr>
<tr>
<td>exit</td>
<td>{entry,0,1,2,3,4,5,exit}</td>
</tr>
</tbody>
</table>

First Pass

<table>
<thead>
<tr>
<th>block</th>
<th>( D[n] )</th>
<th>( D'[n] )</th>
</tr>
</thead>
<tbody>
<tr>
<td>entry</td>
<td>{entry}</td>
<td>{entry}</td>
</tr>
<tr>
<td>0</td>
<td>{entry,0,1,2,3,4,5,exit}</td>
<td>{0,entry}</td>
</tr>
<tr>
<td>1</td>
<td>{entry,0,1,2,3,4,5,exit}</td>
<td>{1,0,entry}</td>
</tr>
<tr>
<td>2</td>
<td>{entry,0,1,2,3,4,5,exit}</td>
<td>{2,1,0,entry}</td>
</tr>
<tr>
<td>3</td>
<td>{entry,0,1,2,3,4,5,exit}</td>
<td>{3,1,0,entry}</td>
</tr>
<tr>
<td>4</td>
<td>{entry,0,1,2,3,4,5,exit}</td>
<td>{4,0,entry}</td>
</tr>
<tr>
<td>5</td>
<td>{entry,0,1,2,3,4,5,exit}</td>
<td>{5,4,0,entry}</td>
</tr>
<tr>
<td>exit</td>
<td>{entry,0,1,2,3,4,5,exit}</td>
<td>{exit,3,1,0,entry}</td>
</tr>
</tbody>
</table>

Update rule: \( D[n] = \{n\} \cup \left( \bigcap_{p \in \text{pred}(n)} D[p] \right) \)
Complexity

Iterative algorithm (assume bit vector sets)
- cost of set intersection: cost of single iteration:
- number of intersections in single iteration: number of iterations:
- total complexity:

Complexity: No. of Iterations

The data flow equations for dominator computation form a rapid* framework
- if nodes are visited in reverse post order the number of iterations is bounded by the loop-connectedness of CFG
  - number of back edges that can occur on any acyclic path through G
  - in a reducible graph, this is exactly the loop nesting level (typically a small value)
  - if the graph is irreducible this is worst case $O(n)$

**Aside: Reducible flow graphs**

**Definition:** 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 two properties.

1. The forward edges form an acyclic graph in which every node can be reached from the initial node of $G$.
2. The back edges consist only of edges whose heads dominate their tails.

Why isn't this reducible?

This flow graph has no back edges. Thus, it would be reducible if the entire graph were acyclic, which is not the case.

**Complexity**

**Iterative algorithm**

$O(n^2 e)$ worse case  
$O(<\text{loop nest depth}>ne)$ with reducible CFG

More efficient algorithm due to Lengauer and Tarjan  
$O(e\cdot \alpha(e,n))$ inverse Ackermann  
-- more complex, but up to 900x faster than bitset iterative algorithm  
-- used in gcc  
-- this algorithm has been improved to get better asymptotic behavior*

Improved (very clever) iterative algorithm*  
$O(n+e)$ per an iteration  
-- relatively simple to implement  
-- on real programs up to 2.5x faster than Lengauer and Tarjan


**Alternative definition**

**Definition:** A flow graph $G$ is **reducible** if we can repeatedly collapse (reduce) together blocks $(x,y)$ where $x$ is the only predecessor of $y$ (ignoring self loops) until we are left with a single node.

**Computing IDOM**

Let $sD[n]$ be the set of blocks that strictly dominate $n$, then  
$sD[n] = D[n] - \{n\}$

To compute $iD[n]$, the set of blocks (size $\leq 1$) that immediately dominate $n$  
Set  
$iD[n] = sD[n]$

Repeat until no $iD[n]$ changes:  
$iD[n] = iD[n] - \bigcup_{d \in iD[n]} sD[d]$
**Example**

<table>
<thead>
<tr>
<th>block</th>
<th>Initialization</th>
<th>First Pass</th>
</tr>
</thead>
<tbody>
<tr>
<td>entry</td>
<td>iD[n] = D[n]</td>
<td>iD[n]</td>
</tr>
<tr>
<td>0</td>
<td>(entry)</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>(0,entry)</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>(1,0,entry)</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>(0,entry)</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>(0,entry)</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>(4,0,entry)</td>
<td>0</td>
</tr>
<tr>
<td>exit</td>
<td>(3,0,entry)</td>
<td></td>
</tr>
</tbody>
</table>

Update rule: 
\[ iD[n] = iD[n] - \bigcup_{d \in D[n]} (sD[d]) \]

---

**Dominator Tree**

In the dominator tree the initial node is the entry block, and the parent of each other node is its immediate dominator.

**Calculating the Dominance Frontier**

- Let \( \text{dominates}[n] \) be the set of all blocks which block \( n \) dominates
  - subtree of dominator tree with \( n \) as the root
- The dominance frontier of \( n \), \( DF[n] \) is

\[
DF[n] = \left( \bigcup_{s \in \text{dominate}[n]} \text{succs}(s) \right) - (\text{dominates}[n] - \{n\})
\]
First calculate dominates[n] from the dominator tree.

<table>
<thead>
<tr>
<th>block</th>
<th>dominates[n]</th>
<th>succ(dominates[n])</th>
</tr>
</thead>
<tbody>
<tr>
<td>entry</td>
<td>(entry,0,1,2,3,4,5,exit)</td>
<td>(0,1,2,3,4,5,exit)</td>
</tr>
<tr>
<td>0</td>
<td>(0,1,2,3,4,5,exit)</td>
<td>(0,1,2,3,4,5,exit)</td>
</tr>
<tr>
<td>1</td>
<td>(1,2)</td>
<td>(1,2)</td>
</tr>
<tr>
<td>2</td>
<td>(2)</td>
<td>(2)</td>
</tr>
<tr>
<td>3</td>
<td>(3,exit)</td>
<td>(3,exit)</td>
</tr>
<tr>
<td>4</td>
<td>(4,5)</td>
<td>(4,5)</td>
</tr>
<tr>
<td>5</td>
<td>(5)</td>
<td>(5)</td>
</tr>
<tr>
<td>exit</td>
<td>(exit)</td>
<td>()</td>
</tr>
</tbody>
</table>

Then compute the successor set of dominates[n].

<table>
<thead>
<tr>
<th>block</th>
<th>dominates[n]</th>
<th>succ(dominates[n])</th>
</tr>
</thead>
<tbody>
<tr>
<td>entry</td>
<td>(entry,0,1,2,3,4,5,exit)</td>
<td>(0,1,2,3,4,5,exit)</td>
</tr>
<tr>
<td>0</td>
<td>(0,1,2,3,4,5,exit)</td>
<td>(0,1,2,3,4,5,exit)</td>
</tr>
<tr>
<td>1</td>
<td>(1,2)</td>
<td>(1,2)</td>
</tr>
<tr>
<td>2</td>
<td>(2)</td>
<td>(2)</td>
</tr>
<tr>
<td>3</td>
<td>(3,exit)</td>
<td>(3,exit)</td>
</tr>
<tr>
<td>4</td>
<td>(4,5)</td>
<td>(4,5)</td>
</tr>
<tr>
<td>5</td>
<td>(5)</td>
<td>(5)</td>
</tr>
<tr>
<td>exit</td>
<td>(exit)</td>
<td>()</td>
</tr>
</tbody>
</table>

Finally, remove all the blocks from the successor set that are strictly dominated by n to get DF[n].

<table>
<thead>
<tr>
<th>block</th>
<th>DF[n]</th>
</tr>
</thead>
<tbody>
<tr>
<td>entry</td>
<td>()</td>
</tr>
<tr>
<td>0</td>
<td>()</td>
</tr>
<tr>
<td>1</td>
<td>(3)</td>
</tr>
<tr>
<td>2</td>
<td>(3)</td>
</tr>
<tr>
<td>3</td>
<td>()</td>
</tr>
<tr>
<td>4</td>
<td>(3,4)</td>
</tr>
<tr>
<td>5</td>
<td>(3,4)</td>
</tr>
<tr>
<td>exit</td>
<td>()</td>
</tr>
</tbody>
</table>
Recall: SSA

If there is a def of a in block 5, which nodes need a $\Phi()$?

CFG

D-Tree

Using DF to compute SSA

- Place all $\Phi()$
  - use dominance frontier
  - the arguments to $\Phi$ initially unnamed
- Rename all variables
  - a unique def for each use

Using DF to Place $\Phi()$

Gather all the defsites of every variable
Then, for every variable
  foreach defsite
    foreach node in DF(defsite)
      if we haven’t put $\Phi()$ in node put one in
      if this node didn’t define the variable before: add this node to the defsites

This essentially computes the Iterated Dominance Frontier on the fly, inserting the minimal number of phi functions neccessary
**Using DF to Place $\Phi()$**

```plaintext
given node n {
    foreach variable v defined in n {
        defsites[v] = defsites[v] ∪ \{n\}
    }
}

foreach variable v {
    W = defsites[v]
    while W not empty {
        remove n from W
        foreach y in DF[n]
            if y \notin PHI[v] {
                insert "$v \leftarrow \Phi(v,v,\ldots)" at top of y
                PHI[v] = PHI[v] ∪ \{y\}
                if v not originally defined in y
                    W = W ∪ \{y\}
            }
    }
}
```

**Renaming Variables**

- Walk the D-tree, renaming variables as you go
- Replace uses with more recent renamed def
  - For straight-line code this is easy
  - If there are branches and joins?

**Renaming Variables**

- Walk the D-tree, renaming variables as you go
- Replace uses with more recent renamed def
  - For straight-line code this is easy
  - If there are branches and joins use the closest def such that the def is above the use in the D-tree
- Easy implementation:
  - for each var: rename (v)
  - replace uses with top of stack at def; push onto stack
  - examine successors for $\Phi$ functions
    - mark $\Phi$ arguments appropriately
    - call rename(v) on all children in D-tree for each def in this block pop from stack

**Compute D-tree**

```
1: i ← 1
2: j ← 1
3: k ← 0
4: k < 100?
5: j < 20?
6: return j
7: j ← k
8: k ← k + 1
9: k ← k + 2
```

D-tree
Compute Dominance Frontier

```
1 i ← 1
2 j ← 1
3 k ← 0
4 k < 100?
5 j ← 20?
6 j ← k + 1
7 k ← k + 2
```

Insert $\Phi()$

```
1 i ← 1
2 j ← 1
3 k ← 0
4 k < 100?
5 j ← 20?
6 j ← k + 1
7 k ← k + 2
```

DFs defined[n] defsites[v]
1 {} i (1)
2 {} j (1,5,6)
3 {} k (1,5,6)
4 {} j,k
5 {7} j,k
6 {7} j,k
7 {7} j,k

var: i: $W={1}$

Insert $\Phi()$

```
1 i ← 1
2 j ← 1
3 k ← 0
4 k < 100?
5 j ← 20?
6 j ← k + 1
7 k ← k + 2
```

DFs defined[n] defsites[v]
1 {} i (1)
2 {} j (1,5,6)
3 {} k (1,5,6)
4 {} j,k
5 {7} j,k
6 {7} j,k
7 {7} j,k

var: j: $W={1,5,6}$

DFs

```
1
2
3
4
5
6
7
```

DF{1}
1. \( i \leftarrow 1 \)
2. \( j \leftarrow 1 \)
3. \( k \leftarrow 0 \)

4. \( k < 100? \)
5. \( j < 20? \)
6. \( j \leftarrow \Phi(j, j) \)
7. \( j \leftarrow \Phi(j, j) \)

\[ \text{DFs} \quad \text{defined[n]} \quad \text{defsites[v]} \]
1. \( (i, j, k) \) \( i \) \( (1) \)
2. \( 2 \) \( 0 \) \( j \) \( (1, 5, 6) \)
3. \( 2 \) \( 0 \) \( k \) \( (1, 5, 6) \)
4. \( 4 \) \( 0 \) \( j \) \( (1, 5, 6) \)
5. \( 5 \) \( j, k \) \( (1, 5, 6) \)
6. \( 6 \) \( j, k \) \( (1, 5, 6) \)
7. \( 2 \) \( 0 \) \( j \) \( (1, 5, 6) \)

\[ \text{var} \quad j: \quad W = \{6\} \]

DF(5)
SSA Recap

- SSA is an intermediate representation
- There is a single definition for every use
- Invariant maintained using fake $\Phi$ functions
  – an argument for every predecessor at a join
- Conversion to SSA can be very fast
- SSA form makes optimizing easier
  – often linear time
  – no need to analysis