Static Single Assignment Form

Many compilers use a form of intermediate code called "Static Single Assignment" or SSA. This was developed by Ron Cytron and others at IBM in the 1980's. Cytron is now a professor at Washington U. in St. Louis.

SSA is used by gcc (at least gcc v.4 about 10 years ago) and by some versions of the Java VM.

With SSA we translate the code so that there are multiple versions of each variable. Each version can be assigned to only once. This means that when we see a variable we know it can never be changed, so we can rearrange the code without changing the value of any variable. This greatly facilitates many optimizations.

For example, consider the code

```
x = 0
x = 5
y = x+1
z = (x+1)*5
if (x < 10)
    print( "Hi, Mom!")
```

SSA would write this:

$x_1=0$

$x_2=5$

$y_1=x_2+1$

$z_1=(x_2+1)*5$

if ($x_2<10$)

    print("Hi, Mom!")

x_1=0
x_2=5
y_1=x_2+1
z_1=(x_2+1)*5
if (x_2<10)
    print("Hi, Mom!")

Now we can start analyzing this. The assignment to x_1 is pointless since x_1 never appears again. This can be eliminated. The two x_2+1 expressions must have the same value, since x_2 can never be assigned to.

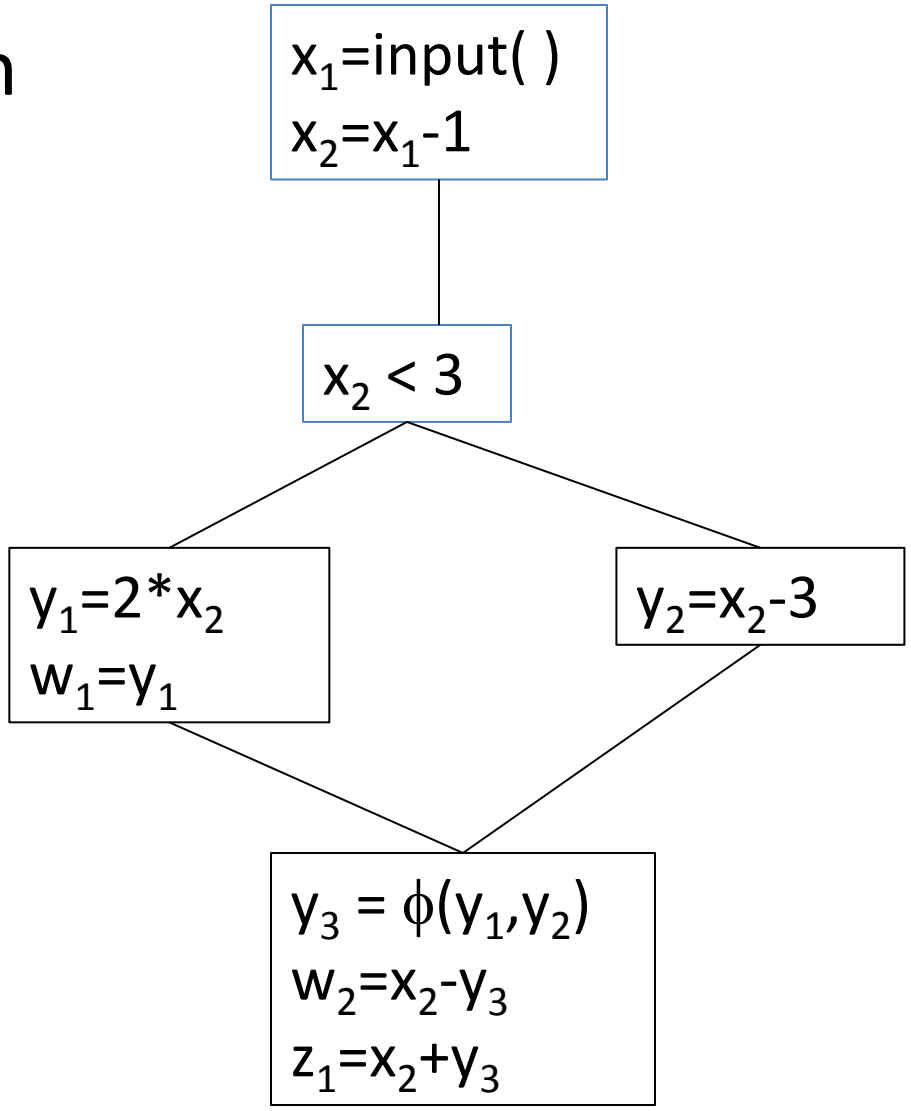
x_2=5
y_1=x_2+1
z_1=(x_2+1)*5
if (x_2<10)
    print("Hi, Mom!")

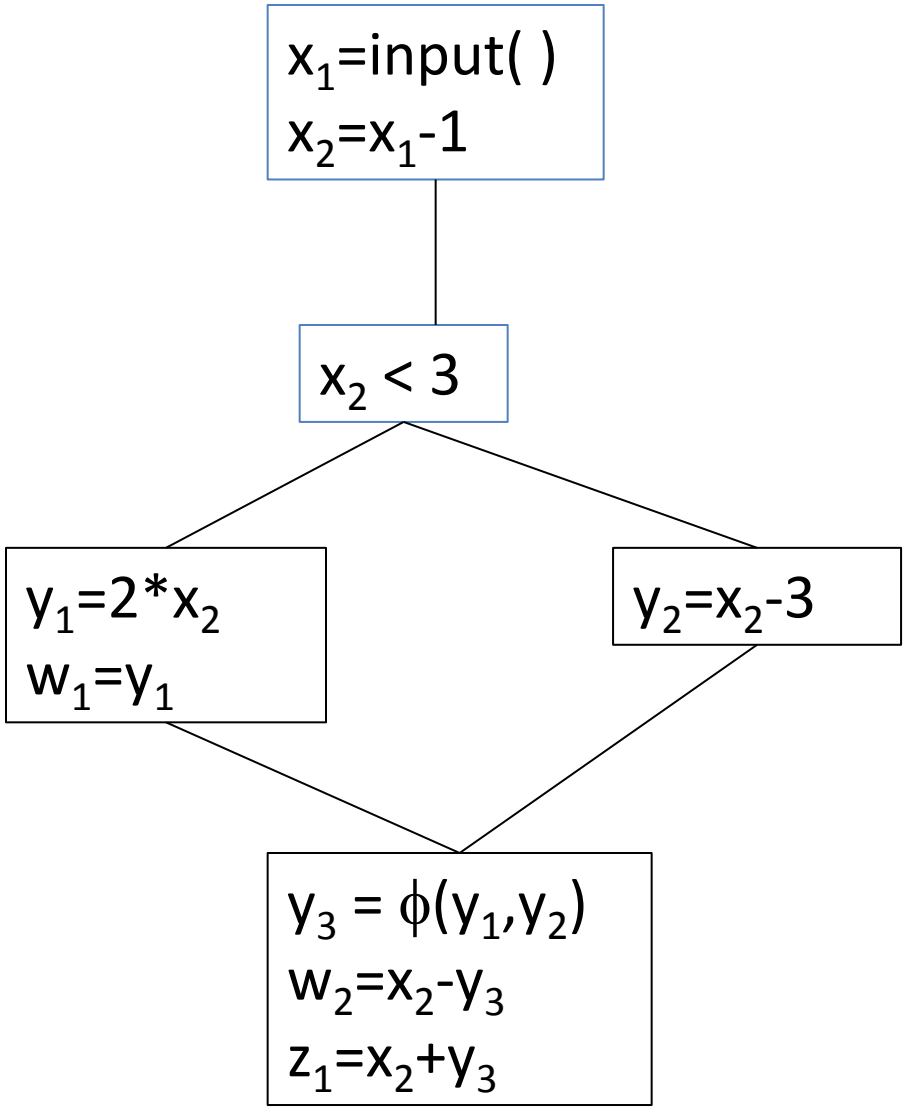
Even better, since x_2 is assigned a constant, we know *at compile time* the values of y_1 and z_1. This reduces the program to

x_2=5
y_1=6
z_1=30
print("Hi, Mom!")

We have to be careful with branches. Consider

```
x=input()
x=x-1
if (x<3) {
        y=2*x
        w=y
}
else
        y=x-3
w=x-y
z=x+y
```





Notice the line

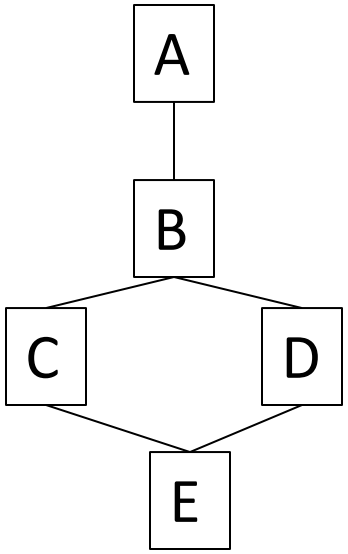
$y_3 = \phi(y_1, y_2)$

It is necessary to bring the two branches of the if-statement together. Initially ϕ was thought of as a crystal ball that would magically know which branch the execution took.

Eventually the researchers realized that no magic was needed. $\phi(y_1,y_2)$ can be thought of as a directive to the code generator to assign y_1 and y_2 to the same register or memory location. That way the execution can continue without knowing which branch of the *if* the execution took.

But where do we need ϕ-functions? And what do we do about loops? The answers rely on the notion of *dominance.*

- In a flow graph, we say node A *dominates* node B if every path from the start to B passes through A.
- We say A *immediately dominates* B if A dominates B and every dominator of B dominates A, For example, in this graph B immediately dominates E:



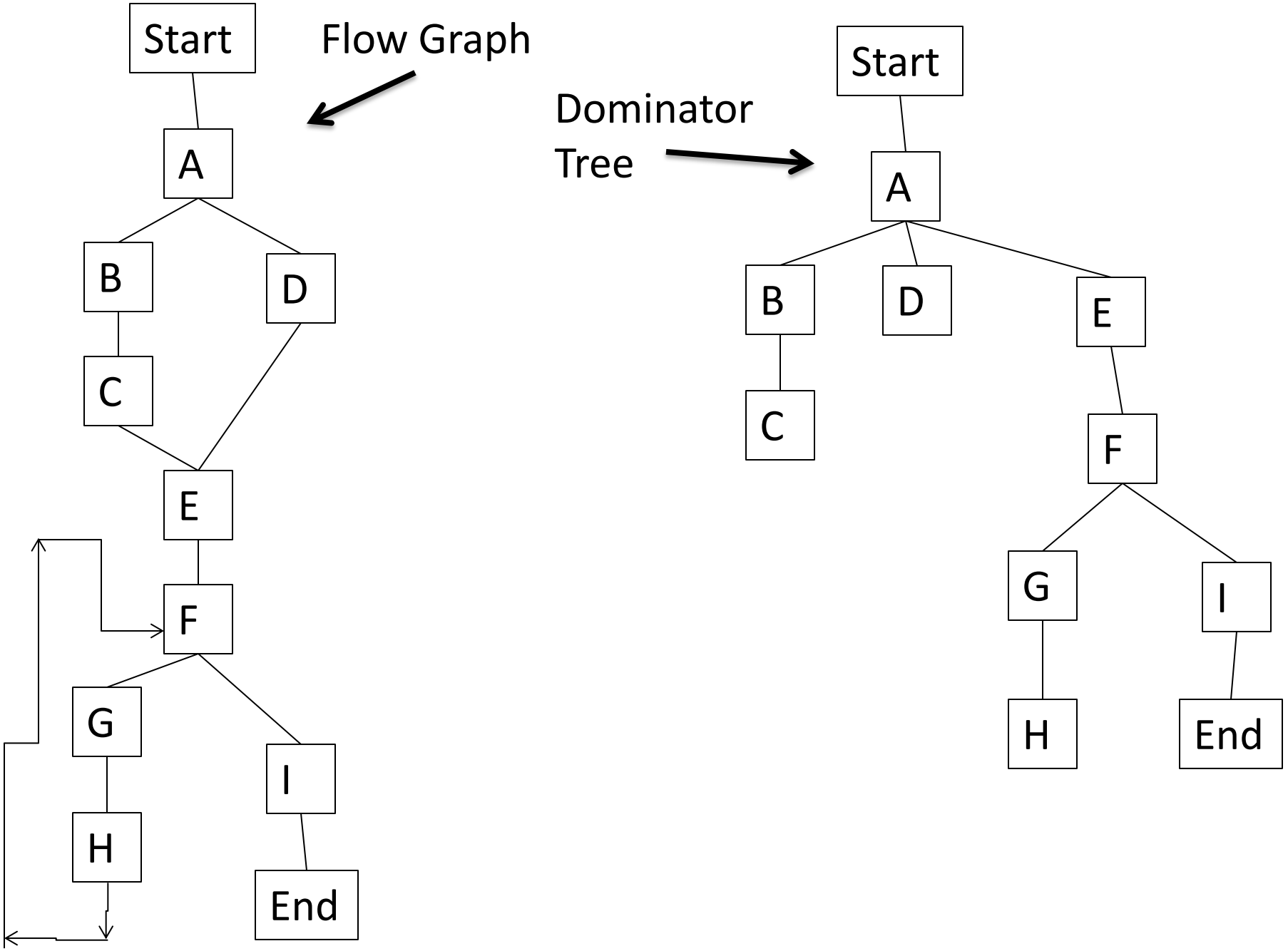
Note that if A dominates B and B dominates A, then A and B must be the same node. We can find a path from start to B passing through A. If A is not B the portion of this path from start to A doesn't pass through B, contradicting the requirement that B dominates A.

So immediate dominators are unique (otherwise two immediate dominators would have to dominate each other).

Here is an algorithm for finding the immediate dominator of any node B: Consider any path from the start to B. Let $A_1 \ldots A_n$ be the dominators of B on this path, enumerated in the order they appear on this path. I claim that A_n must be an immediate dominator of B.

To see this, suppose A_n does not immediately dominate B. Then there must be a node A that dominates B but not A_n. This means there is a path from start to A_n not passing through A. Since A_n dominates B, this path can be extended to a path from start to B not passing through A, contradicting the assumption that A dominates B.

In other words, to find the immediate dominator of any node B, find a path from the start to B. The last dominator of B on this path is its immediate dominator.


Start
Flow Graph
A
B
D
C
E
F
G
I
H
End
Dominator
Tree
Start
A
B
D
E
C
F
G
I
H
End

The *dominator tree* of a flow graph has an edge from A to B only if A immediately dominates B. This forms a tree since if there were two different paths connecting node C to node D we would have to have multiple immediate dominators of one node.

Let *dom(A)* be the set of all nodes in a flowgraph that dominate node A.

Note that

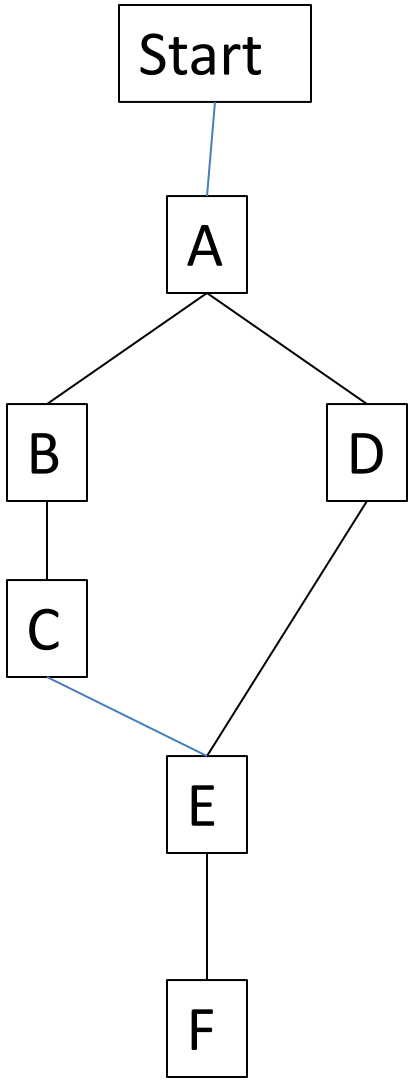
$$dom(A) = \{A\} \cup \bigcap_{B \to A \text{ is an edge of the flowgraph}} dom(B)$$

This just says that a dominator of A must dominate all of A's immediate predecessors.

So here is an algorithm for finding the dominators of every node in a flowgraph:

Initially for every node A we make dom(A) be the set of all nodes. Create a worklist containing just the start node. Then

```
while worklist is not empty:
        Remove a node Y from the worklist
        Let New be {Y} + intersection of
                dom(X) for every edge X->Y
        If New != dom(Y):
                dom(Y)=New
                add the successors of Y to the
                        worklist
```



Initially all nodes have dom(X)={Start, A,B,C,D,E,F} and Worklist={Start}

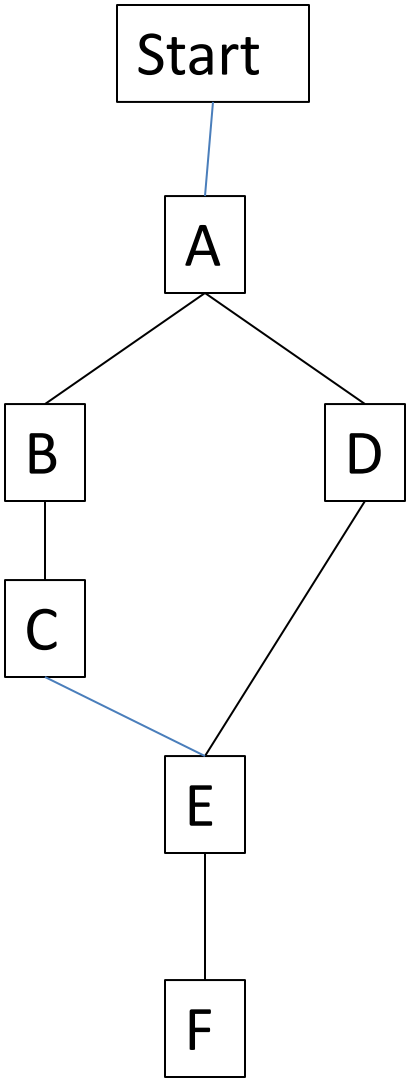
I. Take Start from Worklist. There are no incoming edges to Start, so dom(Start)={Start}. Add A to Worklist.

II. Remove A from Worklist. dom(A)={A}+{Start} = {A,Start}. Add B and D to Worklist.

III. Now Worklist = {B,D}. Suppose we take D from the Worklist. dom(D)={D}+dom(A) ={D,A,Start}. Add E to the Worklist.

IV. Now Worklist = {B,E}. Suppose we take E from the Worklist. dom(E)={E}+[dom(C) intersect dom(D)]={E,D,A,Start}. Add F to the Worklist.

V. Worklist is now {B,F} and E doesn't yet have its final dom(E) set.



At this point Worklist={B,F}, dom(Start)={Start}, dom(A)={A,Start}. dom(D)={D,A,Start}, dom(E)={E,D,A,Start} and the other nodes have everything in their dom sets.

V. Take B from the Worklist.
dom(B)={B}+dom(A)={B,A,Start}. Add C to the Worklist

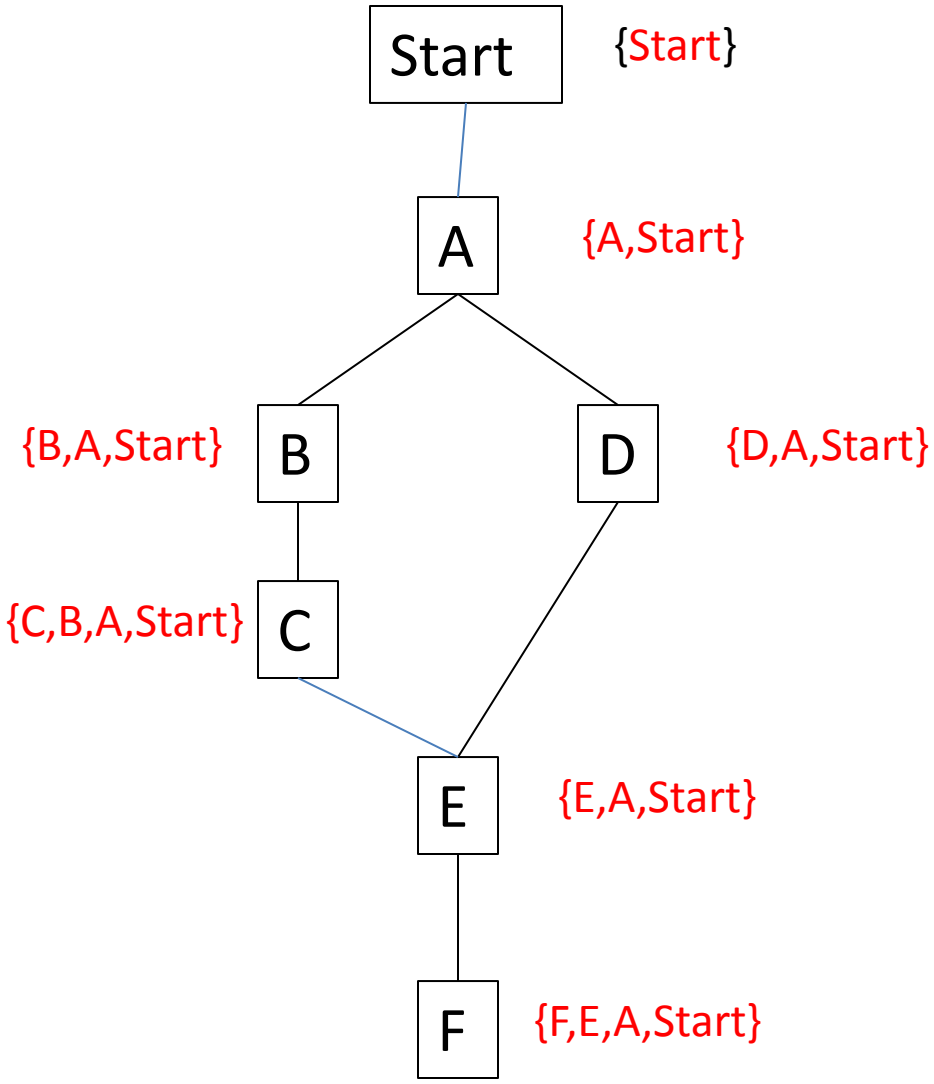
VI. Worklist={C,F}. Take C from the worklist.
dom(C)={C}+dom(B)={C,B,A,Start}. Add E to the worklist again.

VII. Worklist={E,F}. If we take F out we get
dom(F)={F}+dom(E)={F,E,D,A,Start}. F has no successors so there is nothing to add.

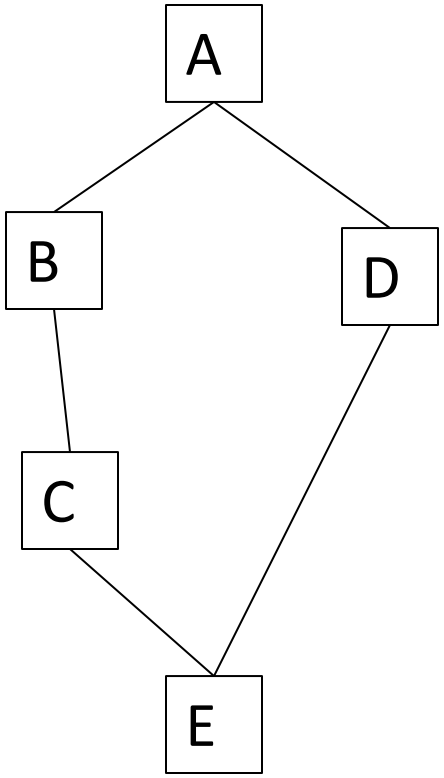
VIII. Worklist={E}. Take E out.
dom(E)={E}+[dom(C) intersect dom(D)]={E,A,Start}. Add F to the Worklist.

IX. Finally, dom(F)={F}+dom(E)={F,E,A,Start} and we are done.

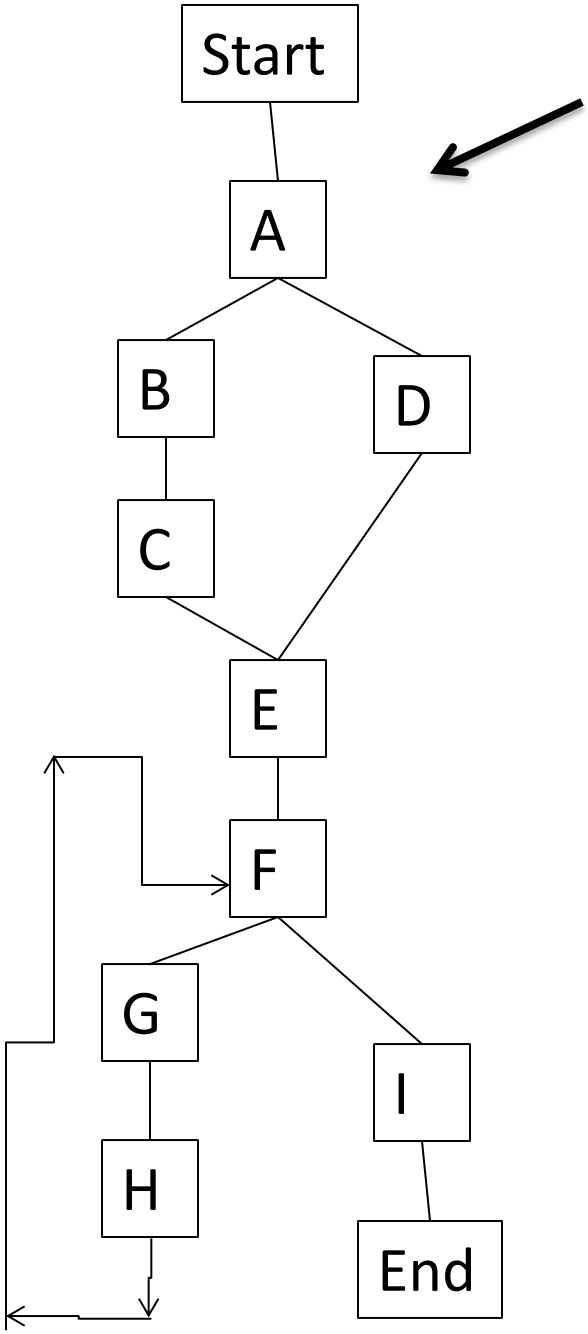
Here are the dominator sets:



DF(A), the *dominance frontier* of node A, is {B such that A does not dominate B, but either A is an immediate predecessor of B or A dominates an immediate predecessor of B}



DF{A}={}
DF(B)={E}
DF(C)={E}
DF(D)={E}
DF(E)={}



DF(Start)={}
DF(A)={}
DF(B)={E}
DF(C)={E}
DF(D)={E}
DF(E)={}
DF(F)={}
DF(G)={F}
DF(H)={F}
DF(I)={}
DF(End)={}

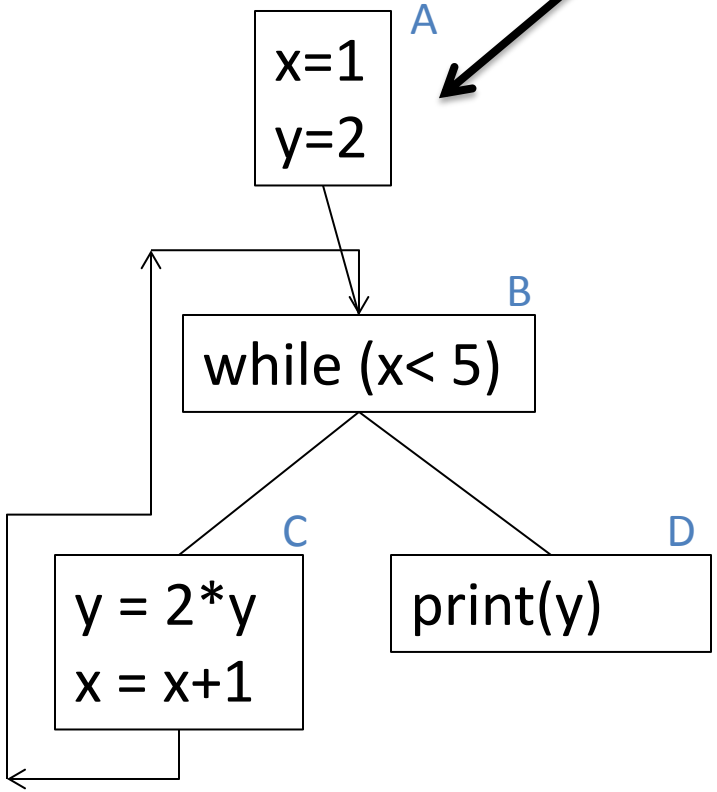
Now back to SSA. Suppose X is a node in a SSA flow graph and some variable p is defined in X. The definition of p is valid in any node that X dominates. Now suppose node Z is in the dominance frontier of X. The definition of p is valid in a predecessor of Z so it reaches node Z, but since X does not dominate Z some alternative definition of p could also reach node Z. We need a ϕ-function in Z to resolve the conflicting definitions of p.

In other words, variables defined in node X need ϕ-functions in the dominance frontier of X.

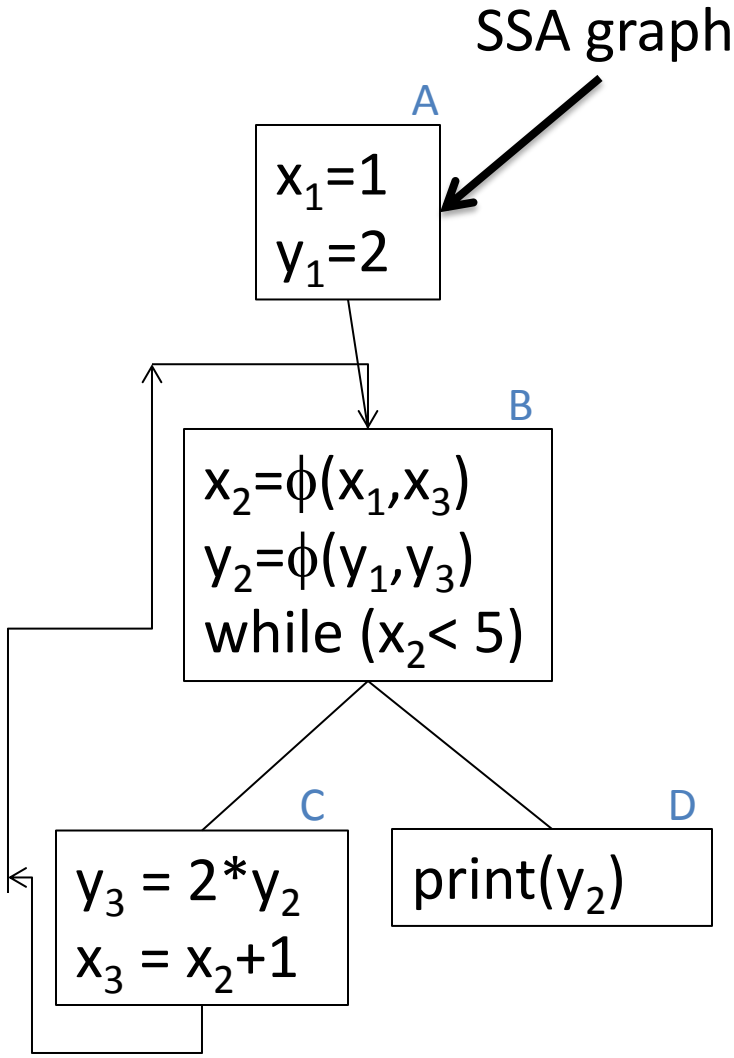
Example

```
x = 1
y = 2
while (x < 5) {
        y = 2*y
        x = x+1
}
print(y)
```

flow graph



SSA graph



We need the ϕ-functions in node B because that is the dominance frontier of node C.