# CSE 401 – Compilers

Lecture 23: Dataflow Analysis/SSA

Michael Ringenburg

Winter 2013



# Reminders

- Project Part 4 due on Friday, March 15.
- There will be a short project report due on Sunday, March 17 – at most **one** late day may be used for the report (if you have any left).
  - One-two pages
  - See posted assignment

# Today's Agenda

- Finish discussing Dataflow Analysis, with more examples
- Begin discussing Single Static Assignment (SSA) form.
  - An IR where every variable has exactly one *static* assignment (may be more dynamically, if assignment is in a loop).
  - Makes many analyses/optimizations more efficient.



# Example From End of Last Class: Live Variable Analysis

- A variable *v* is *live* at point *p* if and only if there is *any* path from *p* to a use of *v* along which *v* is not redefined
  - I.e., *v* might be used before it is redefined

# Liveness Analysis Sets

- We will propagate liveness *backwards* through the control flow graph.
- For each block b, define the following sets
  - use[b] = variables used in b before being defined
    - **Generates** liveness
  - def[b] = variables defined in b before being used
    - **Kills** liveness
  - in[b] = variables live on entry to b
  - out[b] = variables live on exit from b



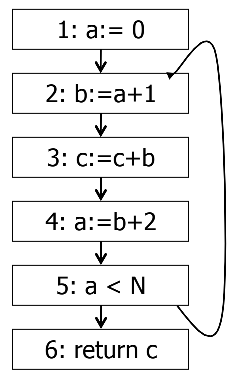
# Equations for Live Variables

- Given the preceding definitions and dataflow framework equations, we have

  $\text{in}[b] = \text{use}[b] \cup (\text{out}[b] - \text{def}[b])$

  $\text{out}[b] = \bigcup_{s \in \text{succ}[b]} \text{in}[s]$
- I.e., live at entry iff this blocks generates liveness (use[b]) or it was live at the exit and this block does not kill liveness (out[b] – def[b]).
- And live at exit iff live at entry to any successor.
- Algorithm
  - Set in[b] = out[b] = ∅
  - Compute use[b] and def[b] for every block (one time)
  - Repeatedly update in, out until no change, using worklist style algorithm we saw last time

# Calculation




$$in[b] = use[b] \cup (out[b] - def[b])$$
$$out[b] = \cup_{s \in succ[b]} in[s]$$



# Equations for Live Variables v2

- Many problems have more than one formulation. For example, Live Variables…
- Sets
  - USED(b) – variables used in b before being defined in b (**generates**)
  - NOTDEF(b) – variables not defined in b **(doesn't kill)**
  - LIVE(b) – variables live on *exit* from b
- Equation – live at exit if live at entry to any successor, and live at entry if generated or live at exit and not killed:

$$LIVE(b) = \cup_{s \in succ(b)} USED(s) \cup (LIVE(s) \cap NOTDEF(s))$$

# Example: Reaching Definitions

- A definition *d* of some variable *v* *reaches* operation *i* iff *i* reads the value of *v* and there is a path from *d* to *i* that does not define *v* (i.e., *i* might use value defined at *d*)
- Uses
  - Find all of the possible definition points for a variable in an expression



# Equations for Reaching Definitions

- Sets
  - DEFOUT(b) – set of definitions in b that reach the end of b (i.e., not subsequently redefined in b). **Generates**.
  - SURVIVED(b) – set of all definitions not obscured by a definition in b. **Doesn't kill**.
  - REACHES(b) – set of definitions that reach b
- Propagate forward through CFG
- Equation – definition reaches b if any predecessor of b generates it, or if it reaches any predecessor and that predecessor does not kill it:

$$\text{REACHES(b)} = \cup_{p \in preds(b)} \text{DEFOUT(p)} \cup (\text{REACHES(p)} \cap \text{SURVIVED(p)})$$

## Example: Very Busy Expressions

- An expression *e* is considered *very busy* at some point *p* if *e* is evaluated and used along every path that leaves *p*, and evaluating *e* at *p* would produce the same result as evaluating it at the original locations
- Uses
  - Code hoisting – move *e* to *p* (reduces code size; no effect on execution time unless moving out of a loop)



## Equations for Very Busy Expressions

- Propagate backwards
- Sets
  - USED(b) – expressions used in b before they are killed. **Generates busy-ness**.
  - KILLED(b) – expressions redefined in b before they are used. **Kills busy-ness**.
  - VERYBUSY(b) – expressions very busy on exit from b
- Equation – expression very busy at exit of b if it is very busy at *every* successor. Very busy at a successor if successor generates busy-ness or if it is busy at successor's exit and successor does not kill busy-ness:

$$\text{VERYBUSY(b)} = \cap_{s \in succ(b)} \text{USED(s)} \cup (\text{VERYBUSY(s)} - \text{KILLED(s)})$$

## Using Dataflow Information

- Dataflow analysis provides a nice framework for doing analysis.
- Optimizations require analysis and transoformations.
- Next, a few examples of possible transformations that rely on dataflow analysis



## Classic Common-Subexpression Elimination

- In a statement s: z := x op y, if x op y is *available* at s (our previous analysis) then it need not be recomputed
- Compute *reaching expressions* i.e., statements n: v := x op y such that the path from n to s does not compute x op y or define x or y
  - Where the expression is available from
  - As we saw in earlier example, available expressions may be available from different places in different paths (e.g., 5*n earlier).

# Classic CSE

- If x op y is defined at n and reaches s
  - Create new temporary t
  - Rewrite n: v := x op y as

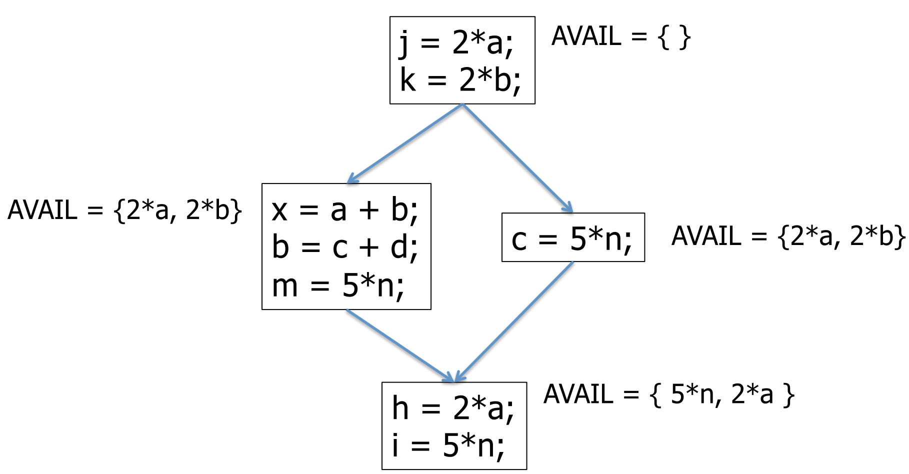
    n: t := x op y

    n': v := t
  - If multiple reaching definition points, rewrite all of them
  - Modify statement s: z := x op y to be

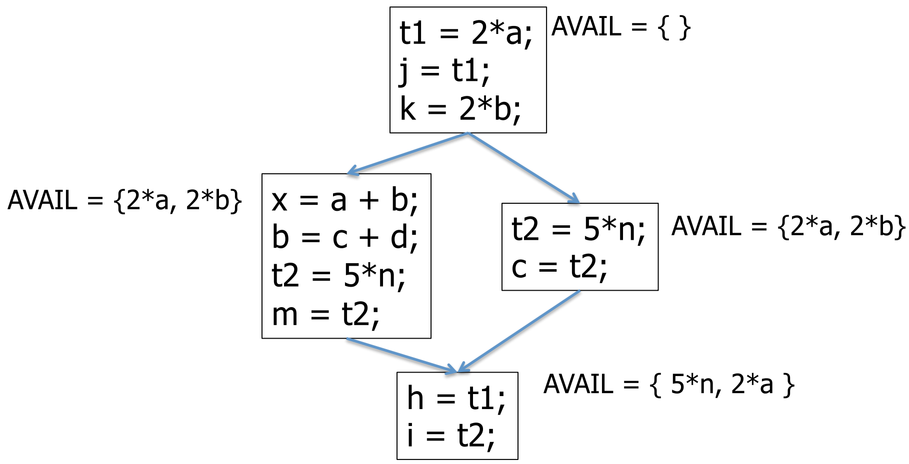
    s: z := t
  - (Rely on copy propagation to remove extra assignments if not really needed)



# Revisiting Earlier Example

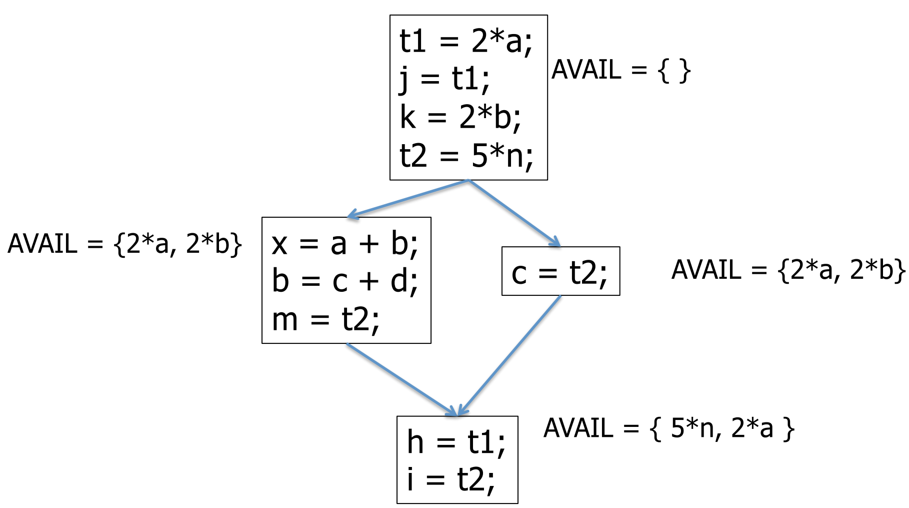


# Revisiting Earlier Example






# Then Apply Very Busy …



## Constant Propagation

- Suppose we have
  - Statement d: x := c, where c is constant
  - Statement n that uses x
- If d reaches n and no other definitions of x reach n, then rewrite n to use c instead of x
  - Or (less common), if all reaching definitions set x to *same* constant c.



## Copy Propagation

- Similar to constant propagation
- Setup:
  - Statement d: x := z
  - Statement n uses x
- If d reaches n and no other definition of x reaches n, and there is no definition of z on any path from d to n, then rewrite n to use z instead of x
  - We saw earlier how this can help remove dead assignments

## Copy Propagation Tradeoffs

- Downside is that this can increase the lifetime of variable z and increase need for registers or memory traffic
- But it can expose other optimizations, e.g.,

```
a := y + z
u := y
c := u + z  // Copy propagation makes this y + z
```

  – After copy propagation we can recognize the common subexpression



## Dead Code Elimination

- If we have an instruction

    s: a := b op c

  and a is not live after s, then s can be eliminated

  – Provided it has no implicit side effects that are visible (output, exceptions, etc.)
    - If b or c are a function call, they may have unknown side effects.

# Dataflow…

- General framework for discovering facts about programs
  - Although not the only possible story
- Can fit many common compiler analyses into this framework
- These facts open opportunities for code improvement



# Next Topic: SSA Form

- SSA (Single Static Assignment) is a very common IR used by optimizing compilers
  - Makes many analyses (and thus optimizations) more efficient.
  - Key property: Each variable has exactly one *static* definition. May have multiple dynamic definitions, e.g., a loop.
- Our next topic: An overview of the SSA IR
  - Constructing SSA graphs
  - SSA-based optimizations
  - Converting back from SSA form

## Motivation: Def(ine)-Use Chains

- Common dataflow analysis problem: Find all sites where a variable is used, or find the possible definition sites of a variable used in an expression
- Traditional solution: def-use (DU) chains – additional data structure on top of the IR
  - Link each statement defining a variable to all statements that use it
  - Link each use of a variable to its possible definitions



## DU-Chain Drawbacks

- Expensive: if a typical variable has N uses and M definitions, total cost is O(N * M * numVariables)
  - Would be nice if cost were proportional to the size of the program
- Unrelated uses of the same variable are mixed together
  - Complicates analysis

# SSA: Static Single Assignment

- IR where each variable has only one definition in the program text
  - This is a single *static* definition, but it may be in a loop that is executed dynamically many times



# SSA in Basic Blocks

Idea: For each original variable x, create a new variable $x_n$ at the $n^{th}$ definition of the original x. Subsequent uses of x use $x_n$, until the next def.

- Original
  - a := x + y
  - b := a – 1
  - a := y + b
  - b := x * 4
  - a := a + b
- SSA
  - $a_1$ := x + y
  - $b_1$ := $a_1$ – 1
  - $a_2$ := y + $b_1$
  - $b_2$ := x * 4
  - $a_3$ := $a_2$ + $b_2$

# Merge Points

- The issue is how to handle merge points in the CFG.

```
if (…)
  a = x;
else
  a = y;
b = a;
```

→

$$\begin{array}{l} \texttt{if (…)} \\ \quad a_1 = x; \\ \texttt{else} \\ \quad a_2 = y; \\ b_1 = ??; \end{array}$$



# Merge Points

- The issue is how to handle merge points in the CFG.

```
if (…)
  a = x;
else
  a = y;
b = a;
```

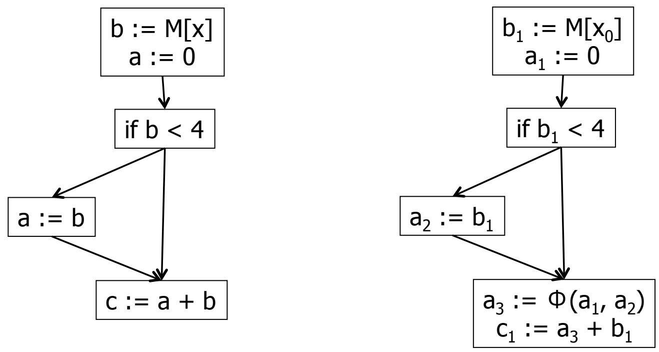
→

$$\begin{array}{l} \texttt{if (…)} \\ \quad a_1 = x; \\ \texttt{else} \\ \quad a_2 = y; \\ a_3 = \Phi(a_1, a_2); \\ b_1 = a_3; \end{array}$$

- Solution: introduce a Φ-function $a_3 := \Phi(a_1, a_2)$
- Meaning: $a_3$ is assigned either $a_1$ or $a_2$ depending on which control path is used to reach the Φ-function

# Another Example

Original | SSA

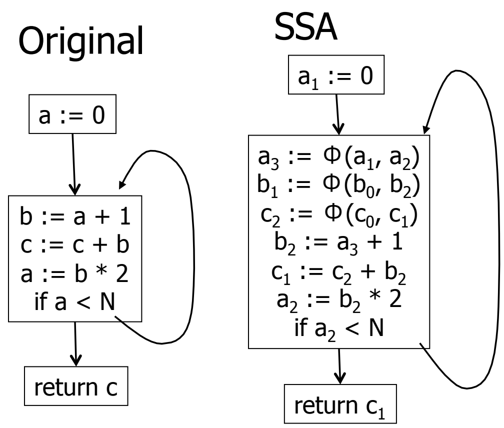





# How Does Φ "Know" What to Pick?

- Φ-functions seem a bit "magical" – how do they know what value to pick??
- They don't actually need to, because they don't exist at run-time …
  - When we're done using the SSA IR, we translate back out of SSA form, removing all Φ-functions.
  - For analysis, all we typically need to know is the connection of uses to definitions – no need to "execute" anything.

# Example With Loop




- Loop back edges also represent merge points, and thus require Φ functions.
- Notes:
  - $a_0$, $b_0$, $c_0$ are initial values of a, b, c on block entry
  - $b_1$ is dead – can delete later



# Converting To SSA Form

- Basic idea
  – First, add Φ-functions
  – Then, rename all definitions and uses of variables by adding subscripts

## Inserting Φ-Functions

- Could simply add Φ-functions for every variable at every join point
- But
  - Wastes way too much space and time
  - Not needed



## When to Insert a Φ-Function

- Insert a Φ-function for variable a at block z when
  - There are blocks x and y, both containing definitions of a, and $x \neq y$
  - There are nonempty paths from x to z and from y to z
  - These paths have no common nodes other than z

# Details

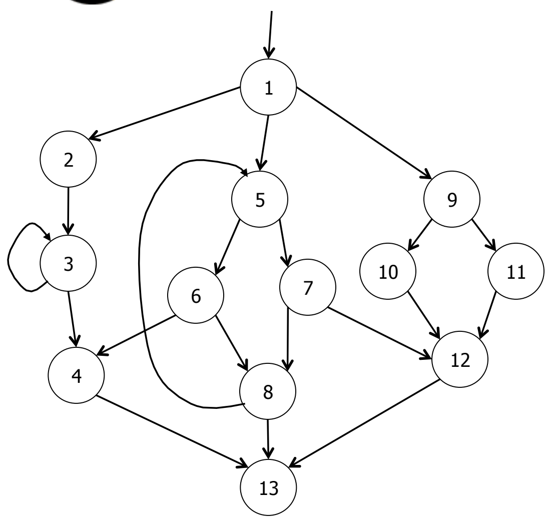
- The start node of the control flow graph is considered to define every variable
- Each Φ-function itself defines a variable, which may create the need for a new Φ-function.
  - So we need to keep adding Φ-functions until things converge (no more changes).
- How do we do this efficiently?
  - Using a new concept: dominance frontiers



# Dominators

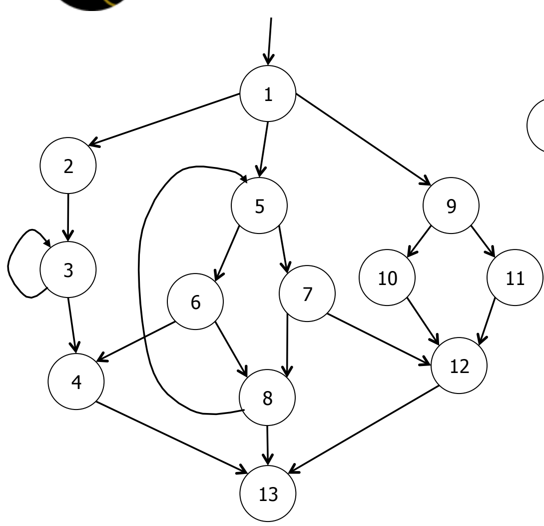
- Definition
  - A block x *dominates* a block y iff every path from the entry of the control-flow graph to y includes x
- By definition, x dominates x
- We can associate a Dom(inator) set with each CFG node
  - | Dom(x) | ≥ 1
- Properties:
  - Transitive: if a dom b and b dom c, then a dom c
  - No cycles, thus can view dominators a tree

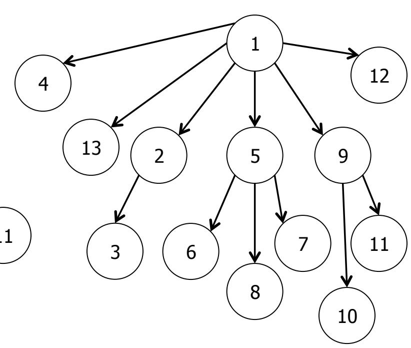
# Example






# Example



## Dominators and SSA

- One property of SSA is that definitions dominate uses; more specifically:
  - If x := Φ(…,$x_i$,…) in block n, then the definition of $x_i$ dominates the $i^{th}$ predecessor of n
  - If x is used in a non-Φ statement in block n, then the definition of x dominates block n



## Dominance Frontier (1)

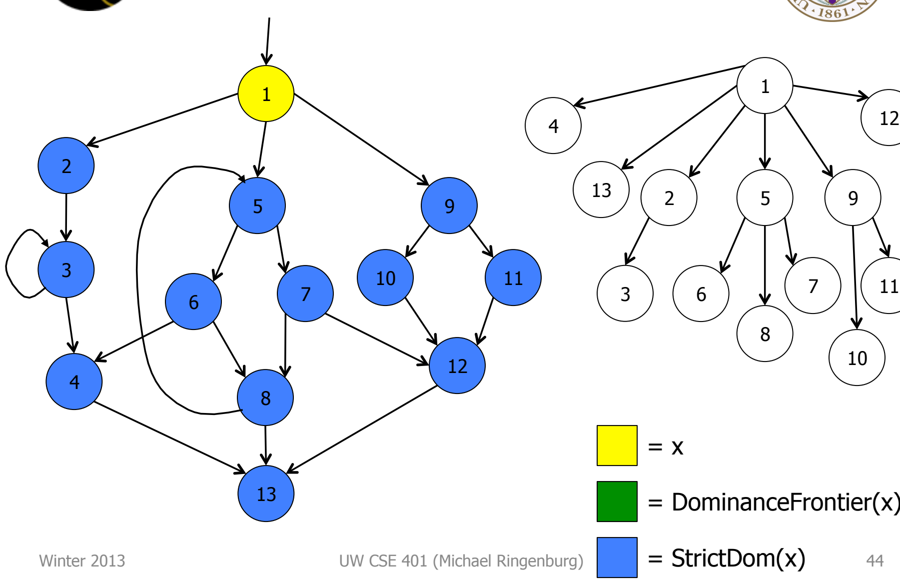
- To get a practical algorithm for placing Φ-functions, we need to avoid looking at all combinations of nodes leading from x to y
- Instead, use the dominator tree in the flow graph

# Dominance Frontier (2)

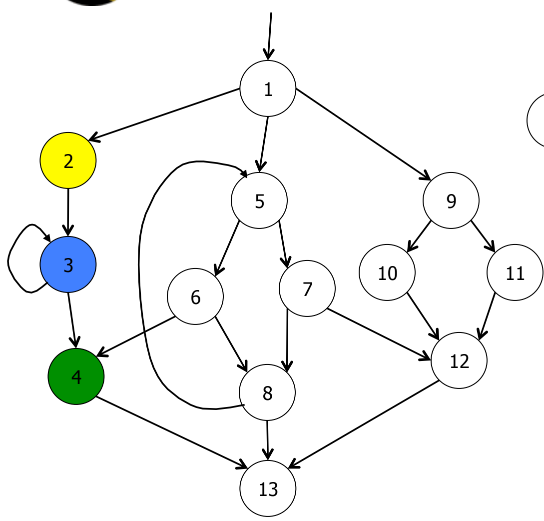
- Definitions
  - x *strictly dominates* y if x dominates y and x ≠ y
  - The *dominance frontier* of a node x is the set of all nodes w such that x dominates a predecessor of w, but x does not strictly dominate w
    - Interestingly, this means that x can be in *it's own dominance frontier!* This can happen if you have a back edge to x (x is the head of a loop).
- Essentially, the dominance frontier is the border between dominated and undominated nodes

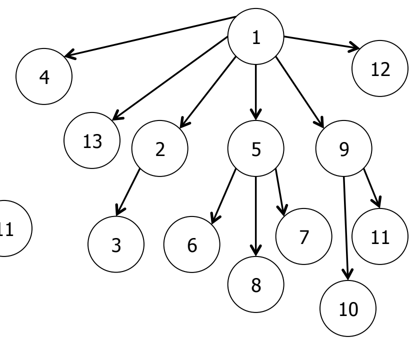


# Example



# Example



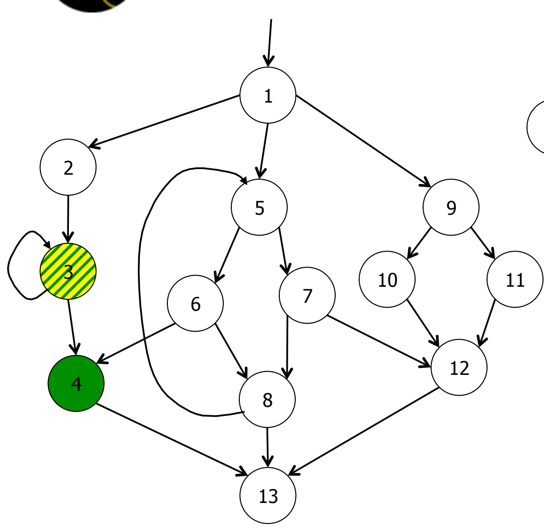

= x

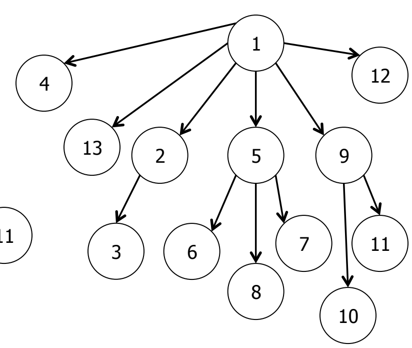
= DominanceFrontier(x)

= StrictDom(x)



# Example



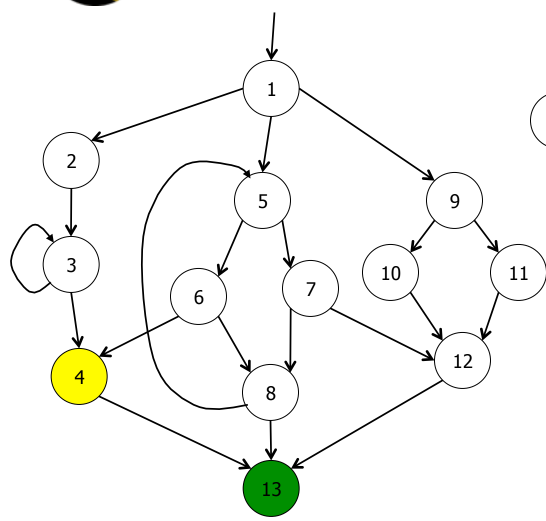



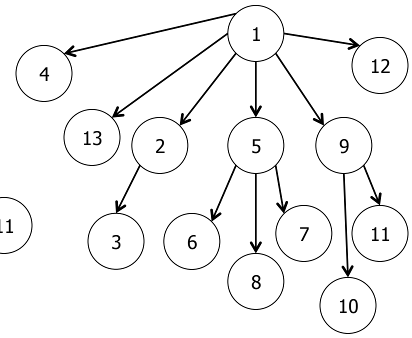

= x

= DominanceFrontier(x)

= StrictDom(x)

# Example



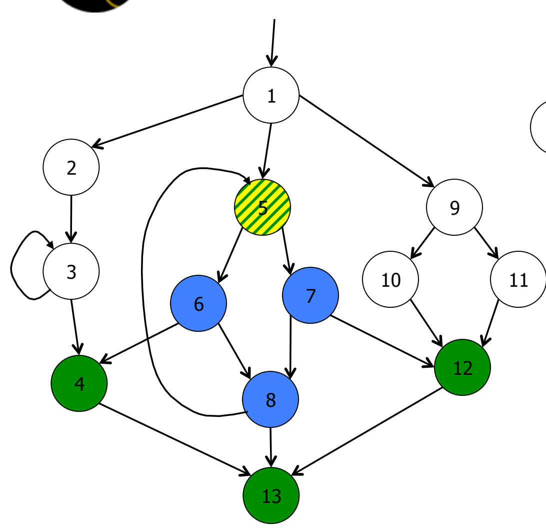

= x

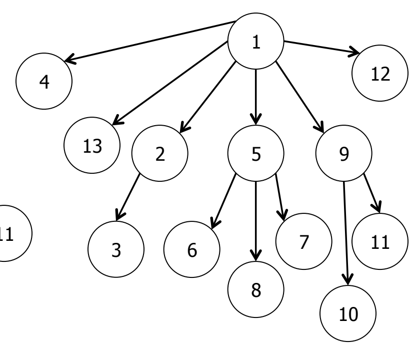
= DominanceFrontier(x)

= StrictDom(x)



# Example



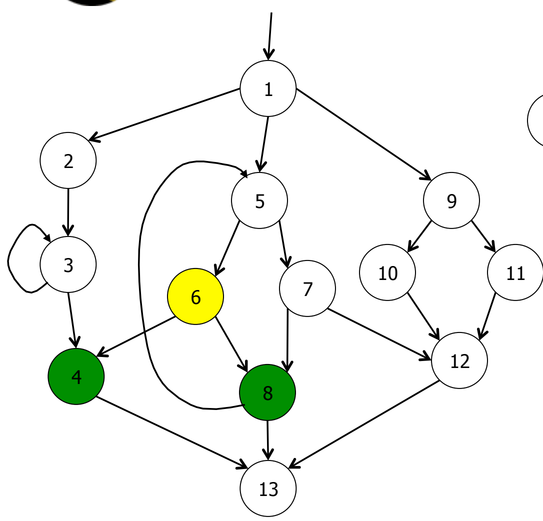



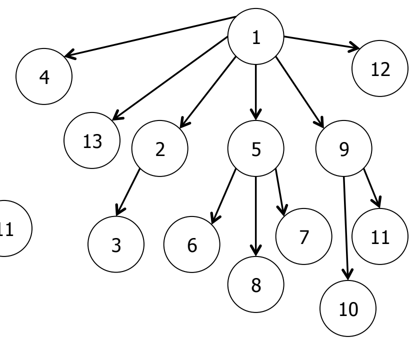

= x

= DominanceFrontier(x)

= StrictDom(x)

# Example



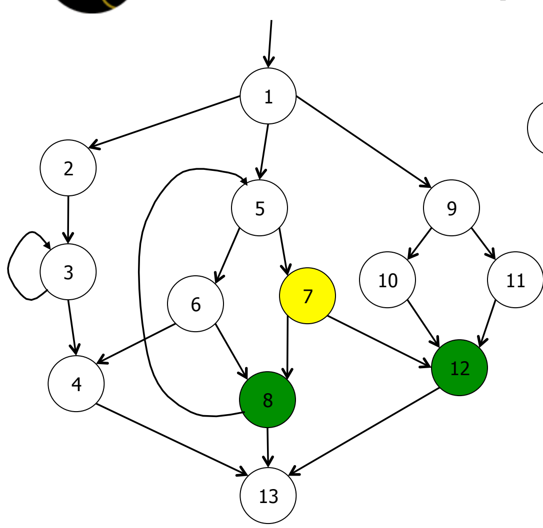

= x

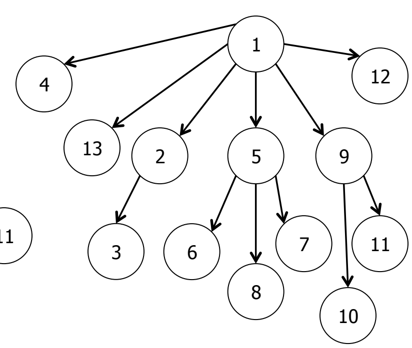
= DominanceFrontier(x)

= StrictDom(x)



# Example



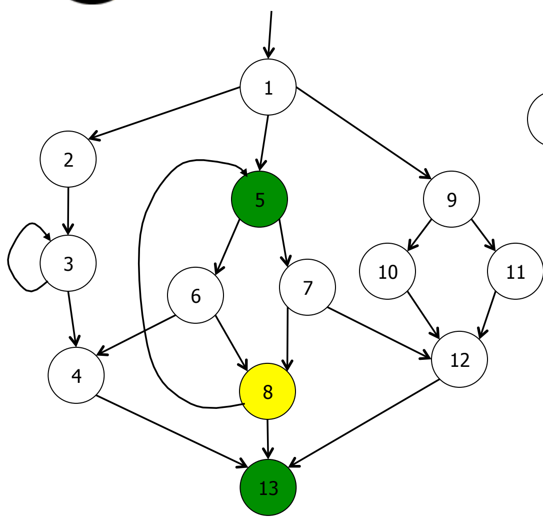



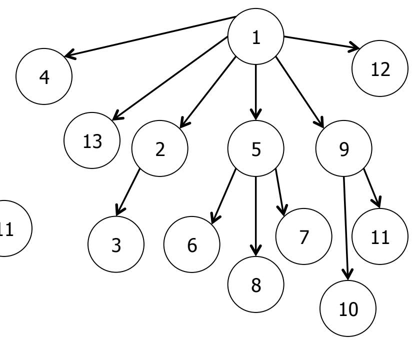

= x

= DominanceFrontier(x)

= StrictDom(x)

# Example



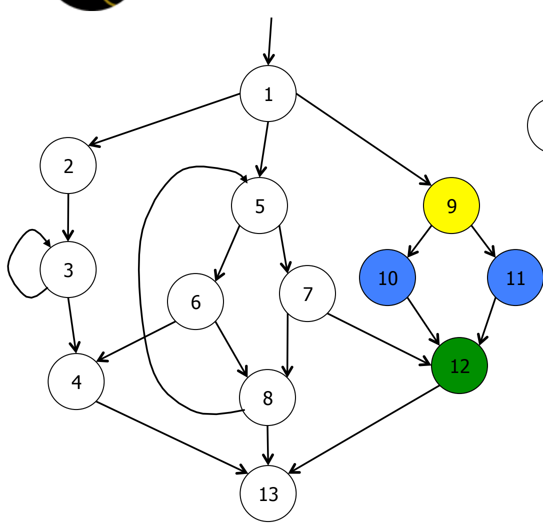

= x

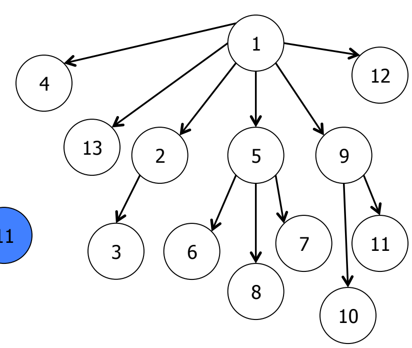
= DominanceFrontier(x)

= StrictDom(x)



# Example



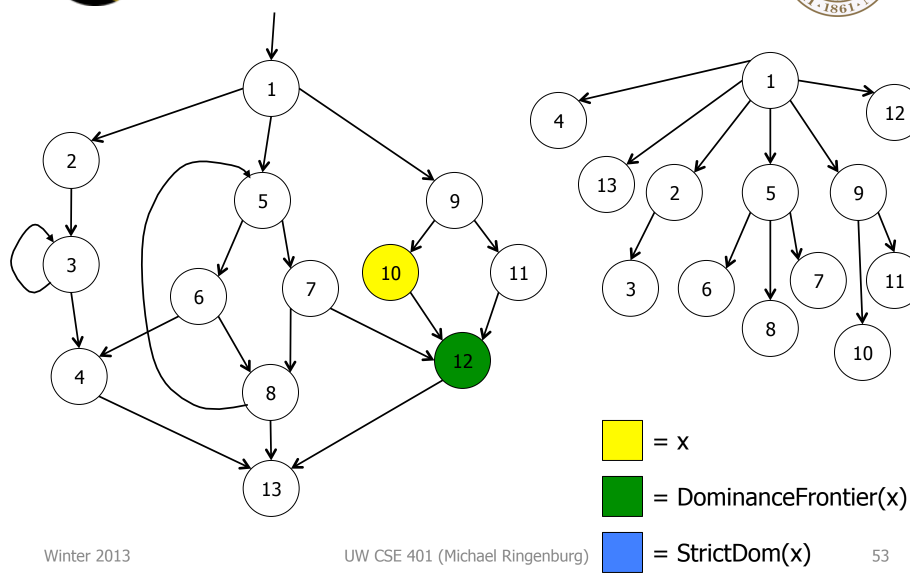




= x

= DominanceFrontier(x)

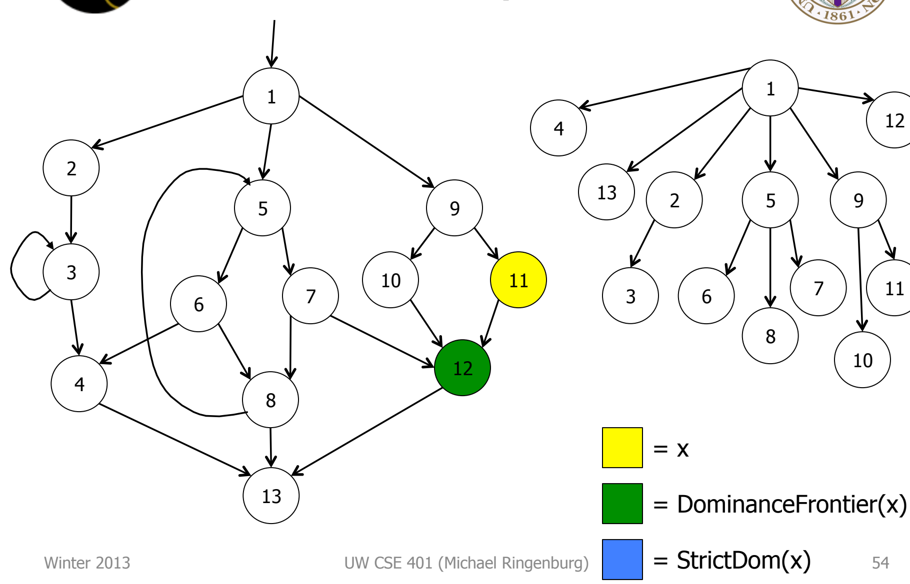
= StrictDom(x)

# Example

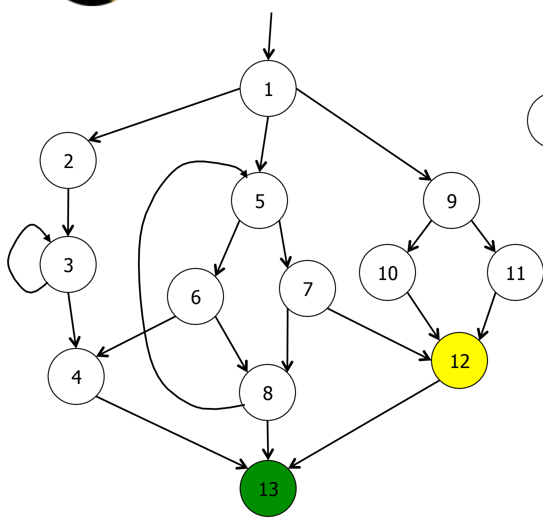


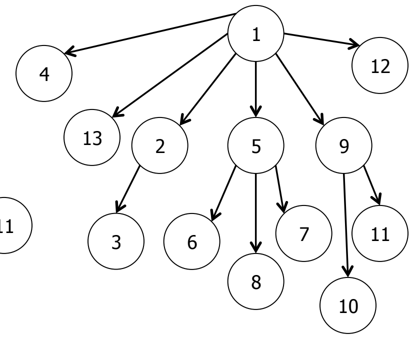



# Example



# Example



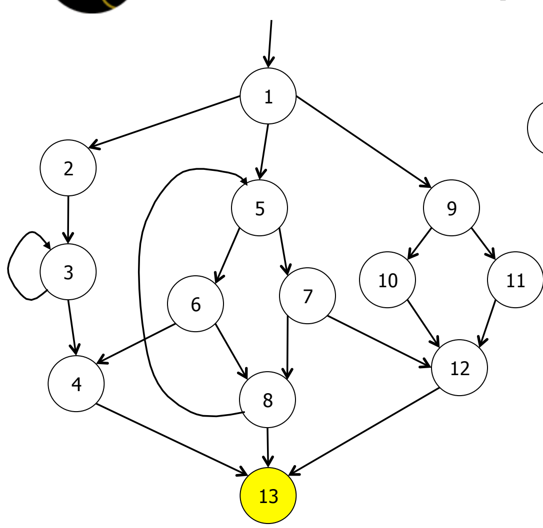

= x

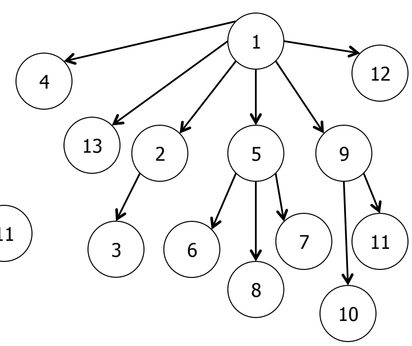
= DominanceFrontier(x)

= StrictDom(x)



# Example




= x

= DominanceFrontier(x)

= StrictDom(x)

## Placing Φ-Functions

- If a node *x* contains the definition of variable *a*, then every node in the dominance frontier of *x* needs a Φ-function for *a*
  - Idea: Everything dominated by x will see x's definition. Dominance frontier represents first nodes we could have reached via an alternate path, which *will* have an alternate reaching definition (recall that we say the entry defines everything).
    - Why does this work for loops? Hint: Strict dominance …
  - Since the Φ-function itself is a definition, this needs to be iterated until it reaches a fixed-point
- Theorem: this algorithm places exactly the same set of Φ-functions as the path criterion given previously.



## Placing Φ-Functions: Details

- We won't give the full constructions here (see your text). The basic steps are:
  1. Compute the dominance frontiers for each node in the flowgraph
  2. Insert just enough Φ-functions to satisfy the criterion. Use a worklist algorithm to avoid reexamining nodes unnecessarily
  3. Walk the dominator tree and rename the different definitions of variable a to be $a_1$, $a_2$, $a_3$, …

## SSA Optimizations

- Advantage of SSA: Makes many optimizations and analyses simpler and more efficient.
  – We'll show a couple examples.
- But first, what do we know? (i.e., what information is kept in the SSA graph?)



## SSA Data Structures

- Statement: links to containing block, next and previous statements, variables defined, variables used.
- Variable: link to its (single) definition statement and (possibly multiple) use sites
- Block: List of contained statements, ordered list of predecessors, successor(s)

# Dead-Code Elimination

- A variable is live iff its list of uses is not empty(!)
- Algorithm to delete dead code:

  while there is some variable v with no uses

    if the statement that defines v has no
      other side effects, then delete it

  - Need to remove this statement from the list of uses for its *operand variables* – which may cause those variables to become dead



# Sparse Simple Constant Propagation

- If c is a constant in v := c, any use of v can be replaced by c
  - Then update every use of v to use constant c
- If the $c_i$'s in v := $\Phi(c_1, c_2, \ldots, c_n)$ are all the same constant c, we can replace this with v := c
- Can also incorporate copy propagation, constant folding, and others in the same worklist algorithm

# Simple Constant Propagation

W := list of all statements in SSA program
while W is not empty
  remove some statement S from W
  if S is v:=Φ(c, c, …, c), replace S with v:=c
  if S is v:=c
    delete S from the program
    for each statement T that uses v
      substitute c for v in T
      add T to W



# Converting Back from SSA

- Unfortunately, real machines do not include a Φ instruction
- So after analysis, optimization, and transformation, need to convert back to a "Φ-less" form for execution

# Translating Φ-functions

- The meaning of $x := \Phi(x_1, x_2, \ldots, x_n)$ is "set $x := x_1$ if arriving on edge 1, set $x := x_2$ if arriving on edge 2, etc."
- So, for each i, insert $x := x_i$ at the end of predecessor block i
- Rely on copy propagation and coalescing in register allocation to eliminate redundant moves



# SSA

- There are many details needed to fully and efficiently implement SSA, but these are the main ideas
- SSA is used in most modern optimizing compilers & has been retrofitted into many older ones (gcc is a well-known example)