CSEP 590 – Programming Systems
University of Washington

Lecture 3: SSA, Register Allocation

Michael Ringenburg
Spring 2017

Course News

- Submit presentation topic proposals by April 14
  - If you would like to work with a partner, both of you will have to present, and I will expect a more in depth/longer presentation
  - We're up to 19 students – tricky to fit >18 into final 3 weeks. Let me know if you'd be willing to present May 9.
    - Otherwise may have to come early or stay late one class (we'll vote)
- Today:
  - Finish discussing optimization techniques:
    - A couple more dataflow examples
    - SSA Form
  - Register allocation via graph coloring
- After that, broaden our horizons a bit and look at other types of programming systems
  - Next week: Specialized programming systems for Big Data
  - Following week: Garbage collection

Dataflow, Continued

Example: Reaching Definitions

- A write (definition) of a variable *reaches* a read if the read might use the defined value.
- Formally: A definition *d* of some variable *v* *reaches* operation *i* if and only if *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:

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

Using Dataflow Information

- A few examples of possible transformations…

Classic Common-Subexpression Elimination

- In a statement s: t := x op y, if x op y is *available* at s (from last week) 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
  - As we saw in last week's 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 w
  - Rewrite n as

    n: w := x op y

    n': v := w
  - If multiple reaching definition points, rewrite all of them
  - Modify statement s to be

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

Constant Propagation

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

Copy Propagation

- Similar to constant propagation
- Setup:
  - Statement d: t := z
  - Statement n uses t
- If d reaches n and no other definition of t reaches n, and there is no definition of z on any path from d to n, then rewrite n to use z instead of t
  - 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 subexpressions

Dead Code Elimination

- If we have an instruction

    s: a := b op c

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

  – Provided it has no implicit side effects that are visible (output, exceptions, etc.)
  – E.g., 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
- And then: facts open opportunities for code improvement
- Next up: SSA (single static assignment) form – transform program to a new form where each variable has only a *single* definition.
  - Can make many optimizations/analyses more efficient

SSA Form

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;
```

→

```
if (…)
  a₁ = x;
else
  a₂ = y;
b₁ = ??;
```

Merge Points

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

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

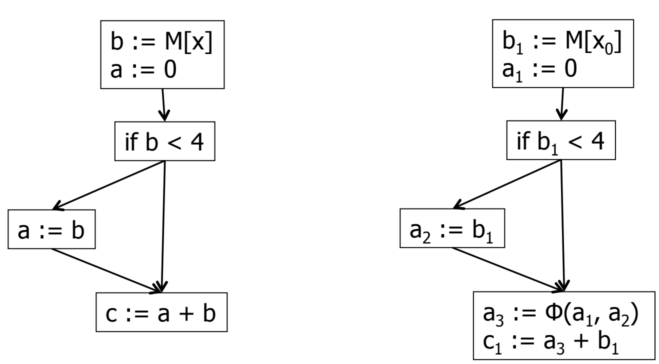
→

```
if (…)
  a_1 = x;
else
  a_2 = y;
a_3 = Φ(a_1, a_2);
b_1 = a_3;
```

- 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

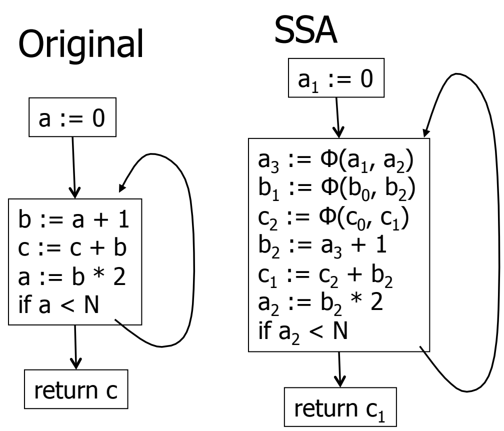
Original SSA



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

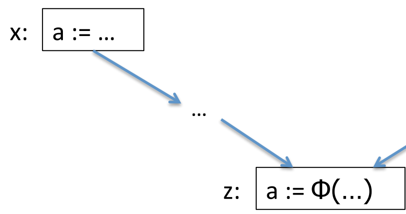
Converting To SSA Form

- Basic idea
  - First, add Φ-functions
  - Then, rename all definitions and uses of variables by adding subscripts
- Renaming is straightforward. Inserting Φ-functions is where things get a little tricky.

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



- We need a Φ-function for variable a at entry to block z whenever
  - 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
    - i.e., this is where the paths first merge

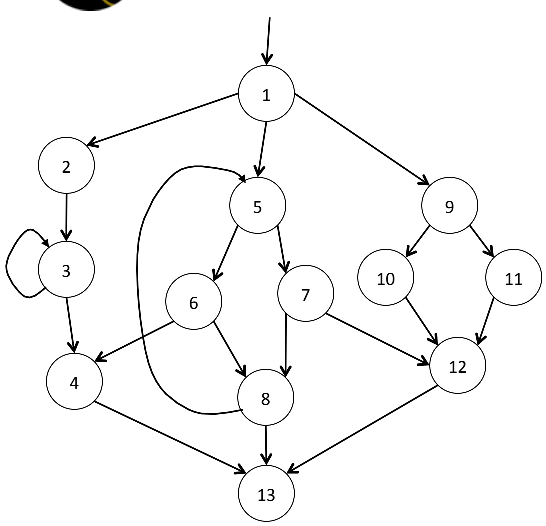
Some Details

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

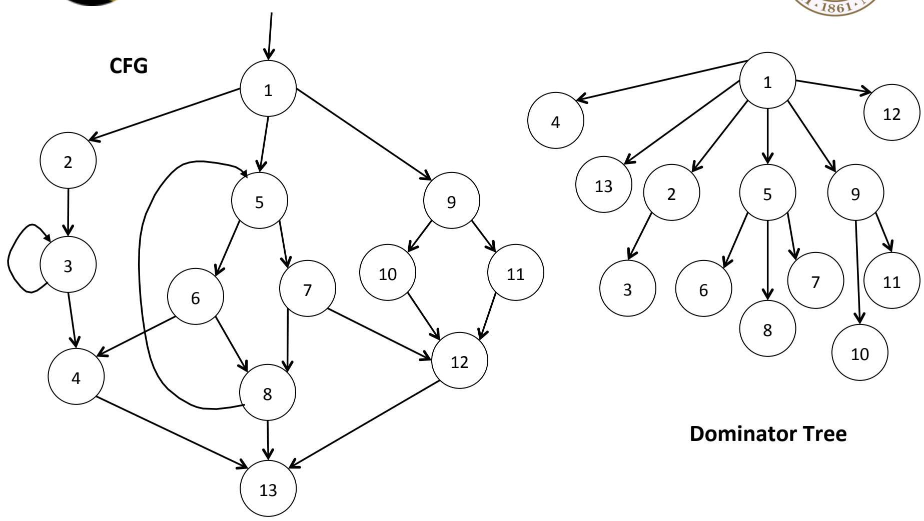
Dominators

- Definition
  - A block x *dominates* a block y if and only if 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
  - The set of all basic blocks that must execute before x
  - | Dom(x) | ≥ 1
- Properties:
  - Transitive: if a dom b and b dom c, then a dom c
  - No cycles, thus can view dominators as a tree

Example



Example



Dominators and SSA

- Important property of SSA: definitions must dominate uses
  - In other words, the single assignment must occur prior to any uses of the variable. (Although that single assignment may just be the start node assignment of "Undefined".)
- 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

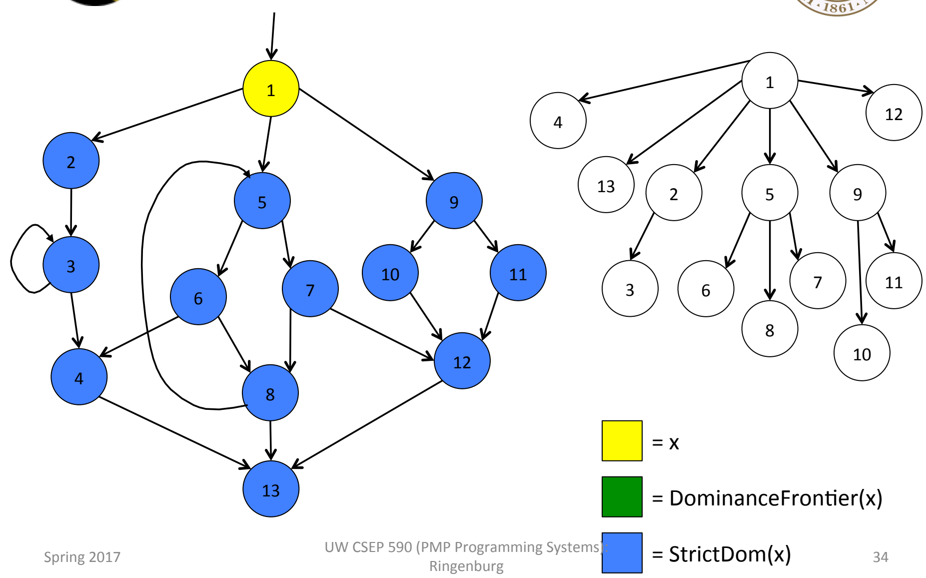
Dominance Frontier (1)

- 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.
  - Place merges *just beyond the end of the definitions' dominance*.
    - The first point where they may receive a value from an alternate definition.
  - This follows directly from the previous properties:
    - Φ-function means predecessors are dominated by defs
    - Non Φ usage means dominated by def
  - This is referred to as the *dominance frontier*.

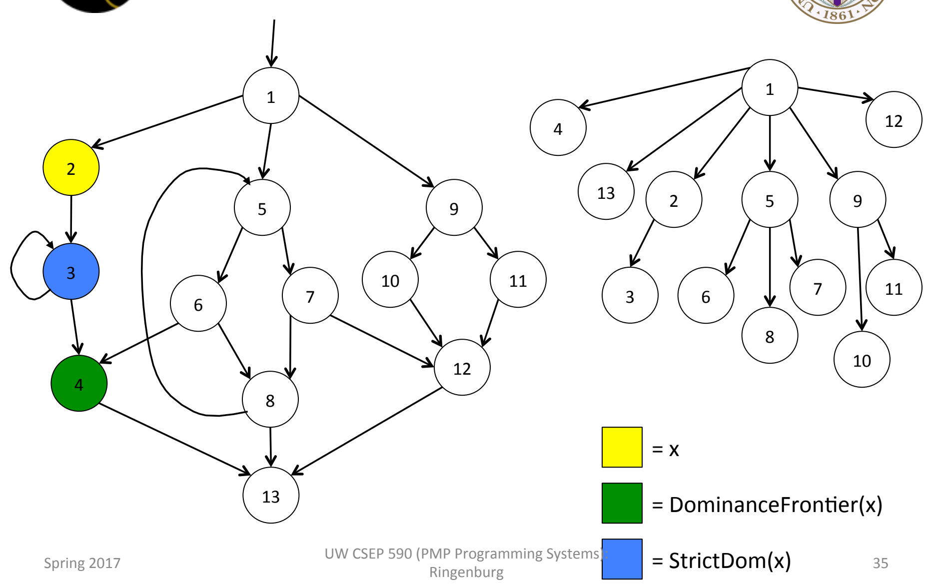
Dominance Frontier (2)

- 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

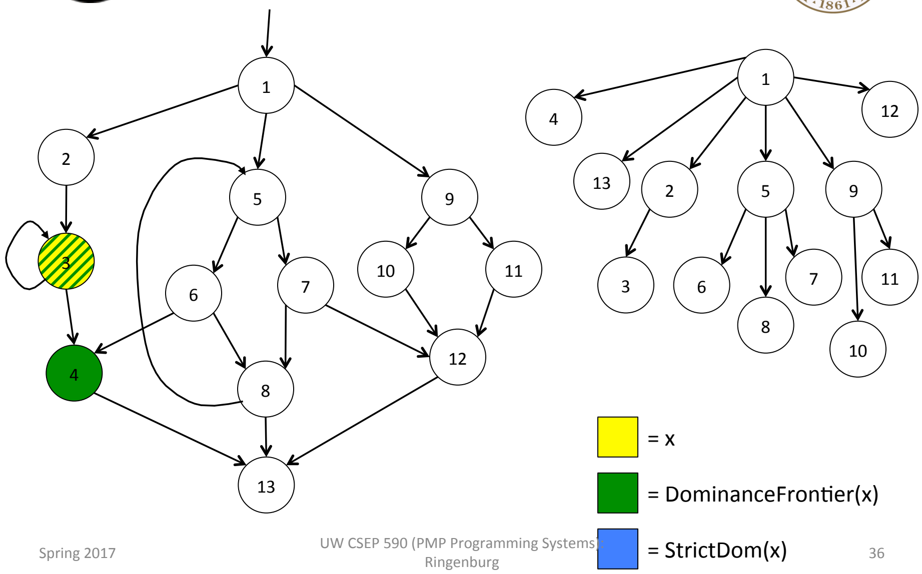
Example



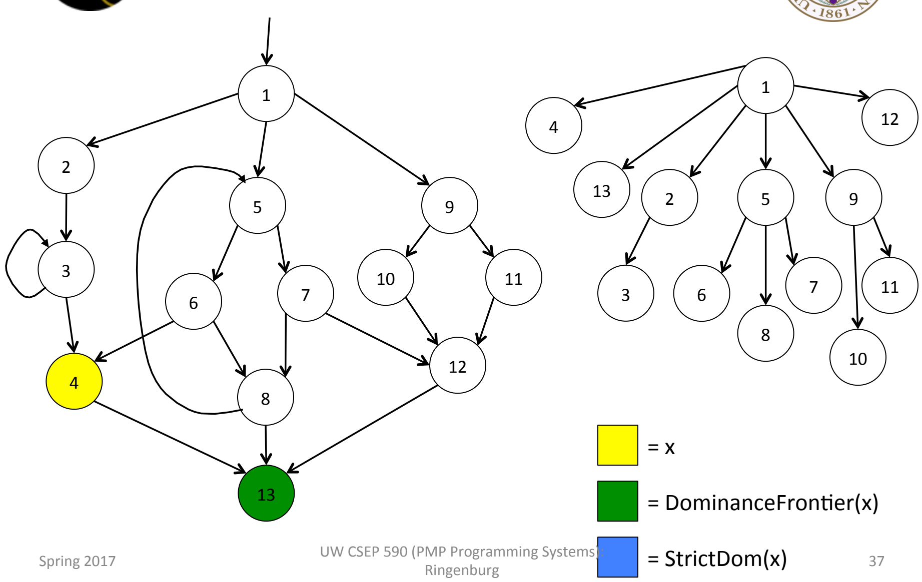
Example



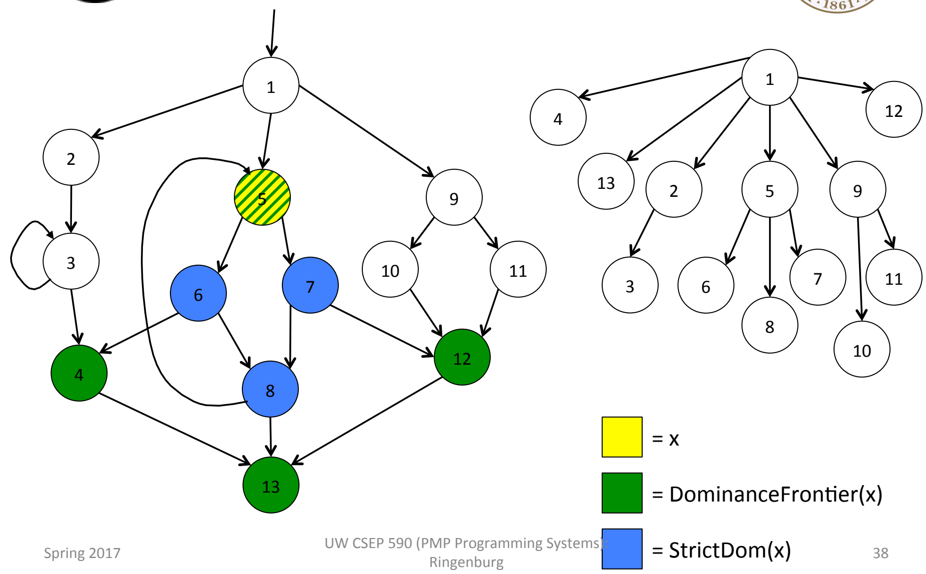
Example



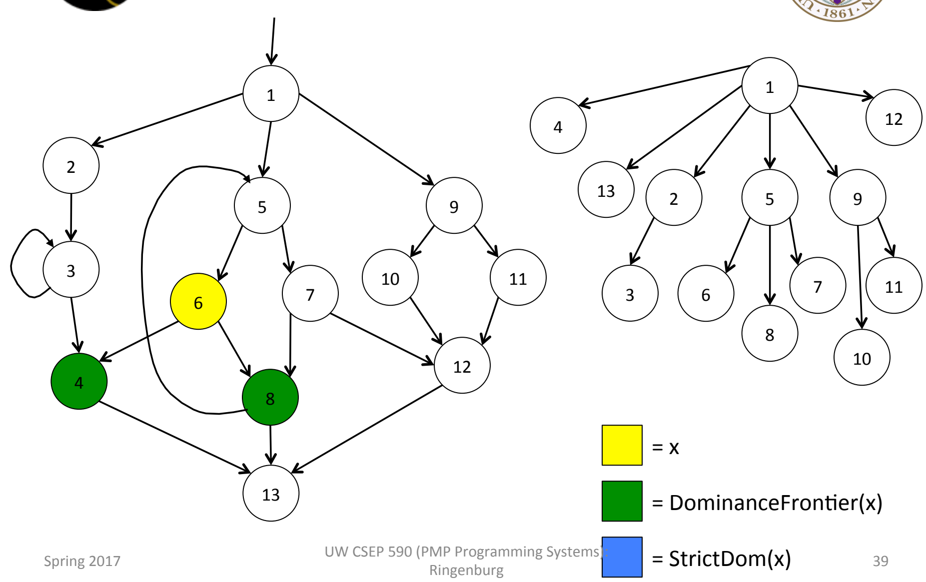
Example



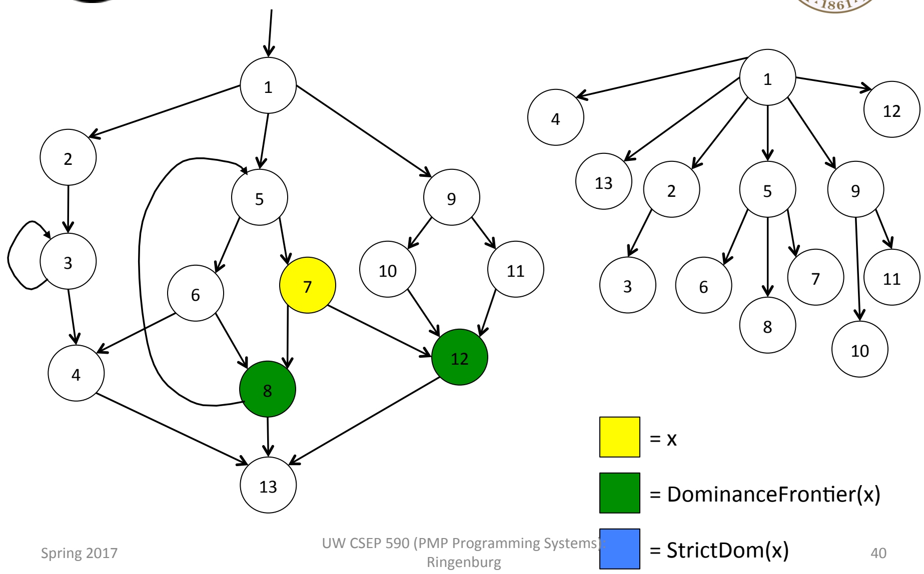
Example



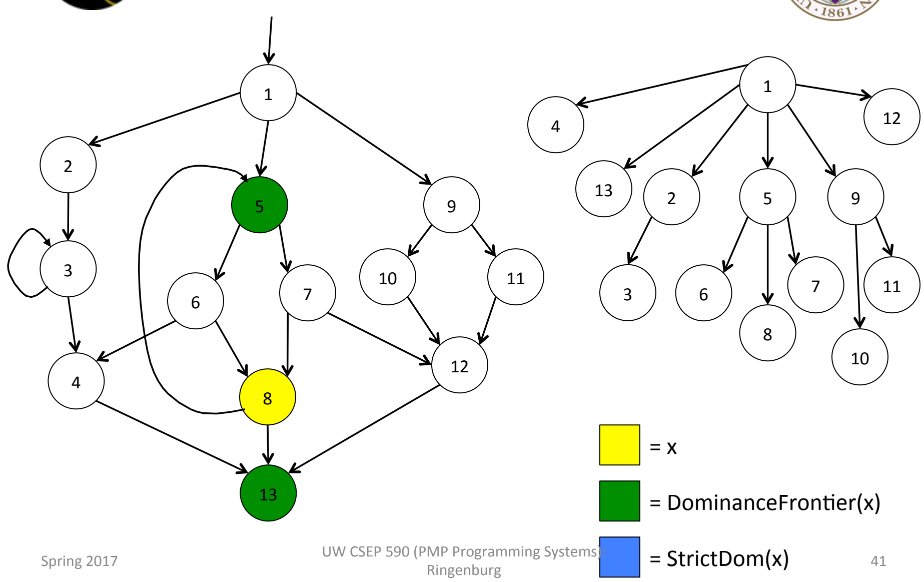
Example



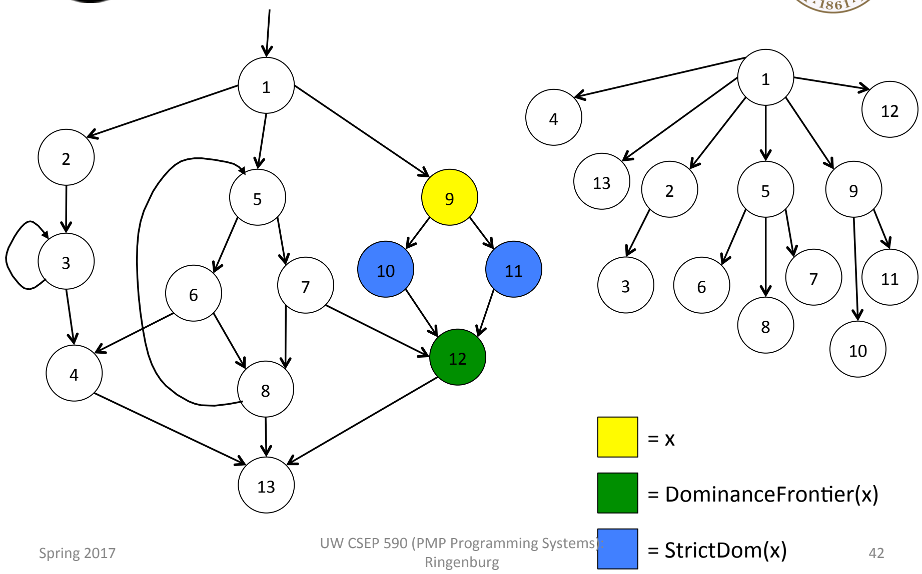
Example



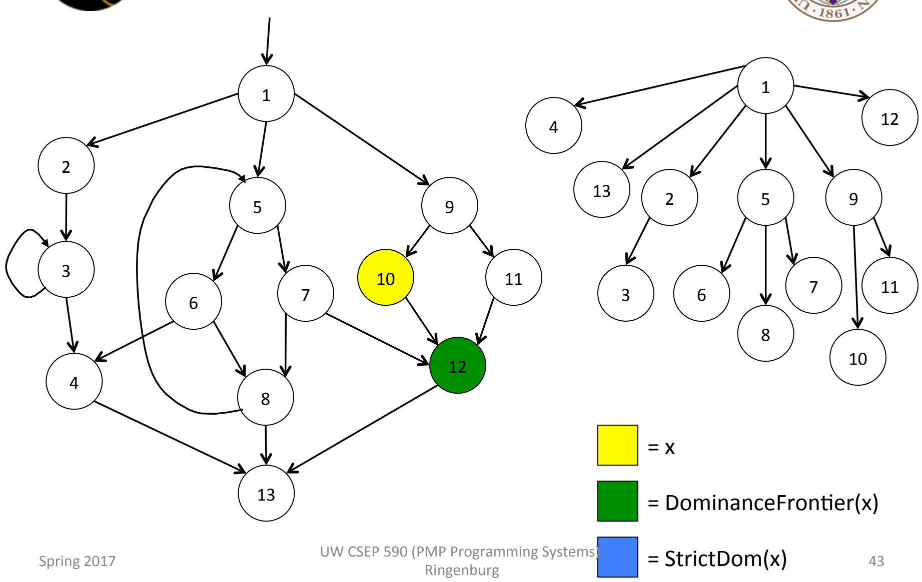
Example



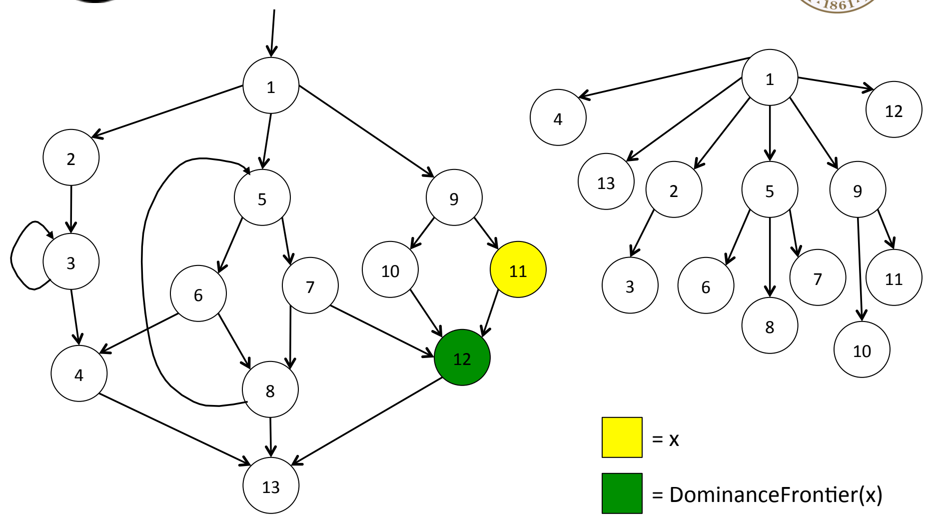
Example



Example



Example

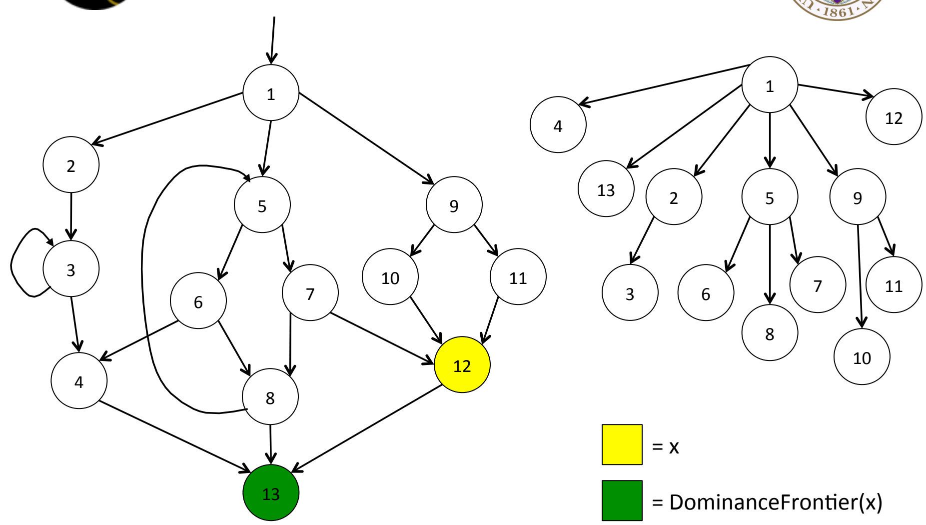


= x

= DominanceFrontier(x)

= StrictDom(x)

Example

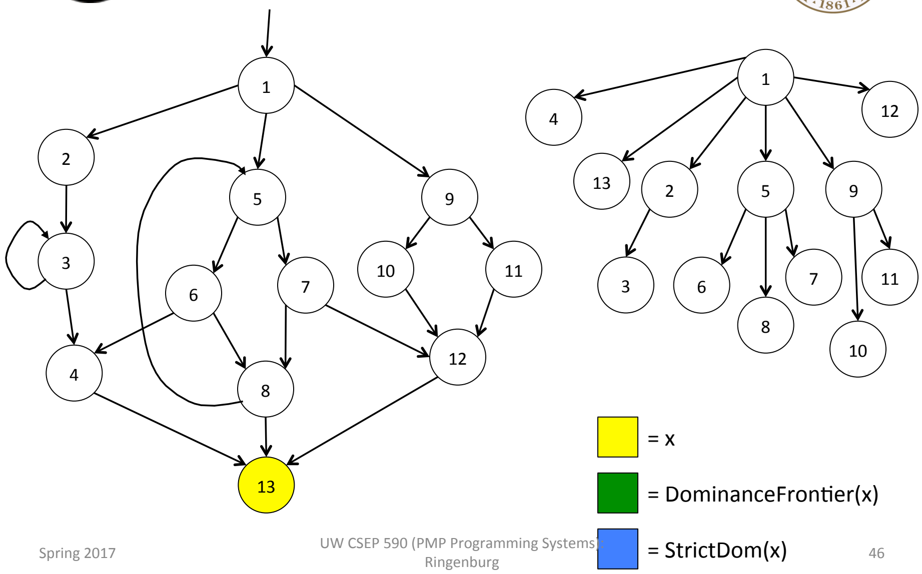


= x

= DominanceFrontier(x)

= StrictDom(x)

Example



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 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

- The basic steps are:
  1. Compute the dominance frontiers for each node in the control flow graph
  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 if and only if its list of uses is not empty(!)
  - Without SSA, possibly many stores to each variable. Have to disambiguate which might be used. With SSA each store defines a new variable, so this becomes trivial …
- 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 (SSCP)

- 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 (or "Undefined" via start node, if you like), we can replace this with v := c
- Can also incorporate copy propagation, constant folding, and others in the same worklist algorithm

Sparse 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

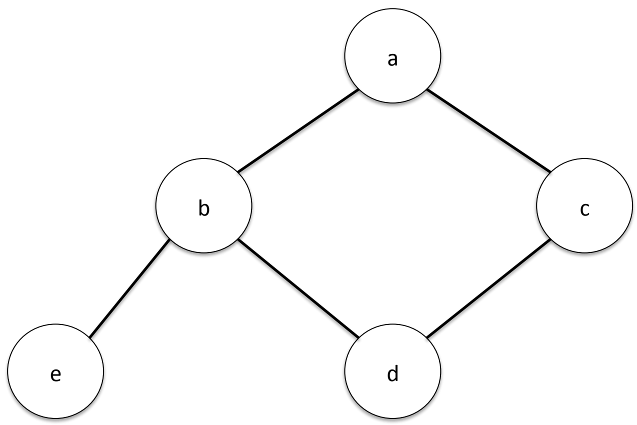
SSA

- There are many details needed to fully and efficiently implement SSA, but these are the main ideas
  - Most modern compiler texts give details:
    - One of my favorites: *Engineering a Compiler*, Cooper & Torczon, 2nd edition
- SSA is used in most modern optimizing compilers & has been retrofitted into many older ones (e.g., gcc)

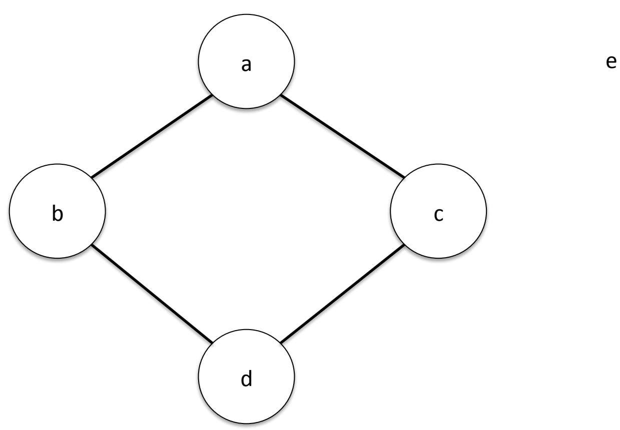
Register Allocation (Briggs-Chaitin)

Switch to slides courtesy of Preston Briggs

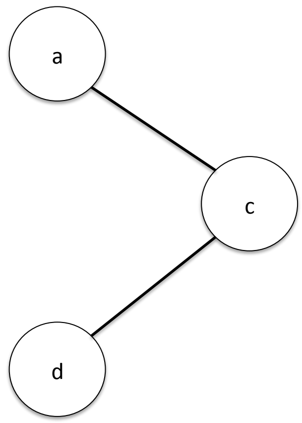
Diamond Graph (2 color)



Diamond Graph (2 color)

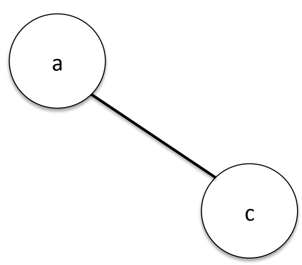


Diamond Graph (2 color)



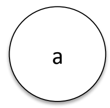
e
b*

Diamond Graph (2 color)



e
b*
d

Diamond Graph (2 color)

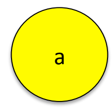


e
b*
d
c

Diamond Graph (2 color)

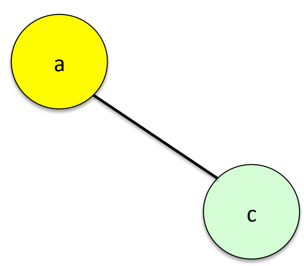
e
b*
d
c
a

Diamond Graph (2 color)



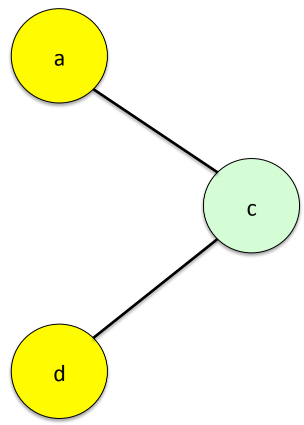
e
b*
d
c

Diamond Graph (2 color)



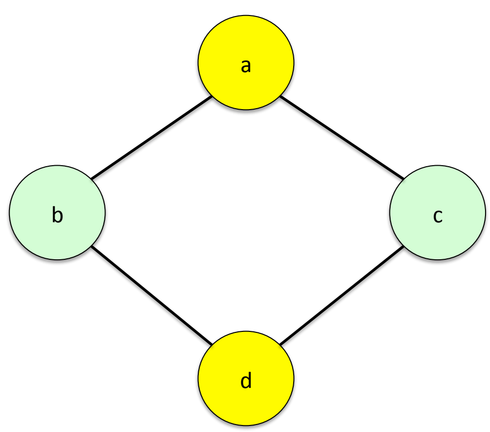
e
b*
d

Diamond Graph (2 color)



e
b*

Diamond Graph (2 color)



e

Diamond Graph (2 color)