CSE P 501 – Compilers

SSA

Hal Perkins

Autumn 2021

Agenda

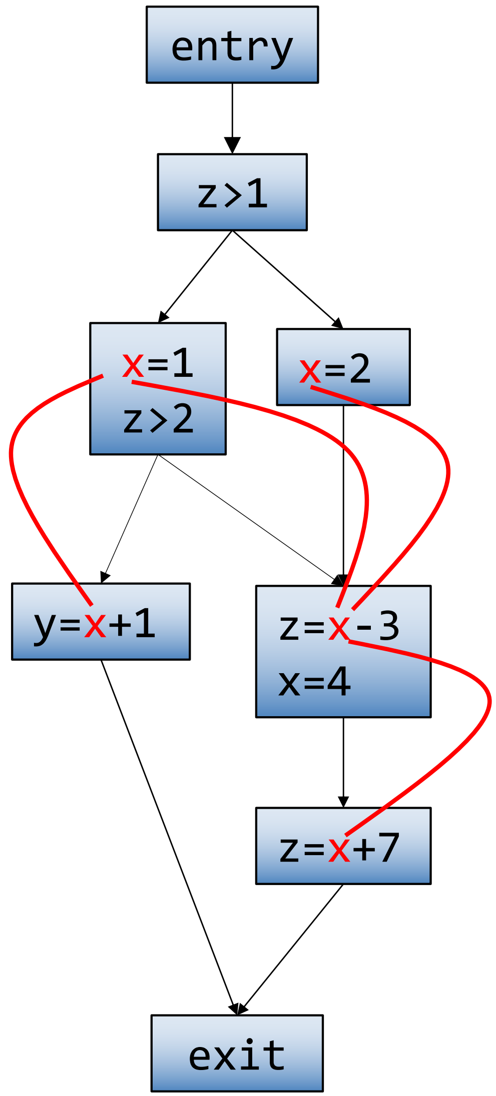
- Overview of SSA IR
  - Constructing SSA graphs
  - Sample of SSA-based optimizations
  - Converting back from SSA form

- Sources: Appel ch. 19, also an extended discussion in Cooper-Torczon sec. 9.3, Mike Ringenburg's CSE 401 slides (13wi)

Def-Use (DU) Chains

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

Def-Use (DU) Chains



In this example, two DU chains intersect

DU-Chain Drawbacks

- Expensive: if a typical variable has N uses and M definitions, the total cost *per-variable* is O(N * M), i.e., $O(n^2)$
  - Would be nice if cost were proportional to the size of the program
- Unrelated uses of the same variable are mixed together
  - Complicates analysis – variable looks live across all uses even if unrelated

SSA: Static Single Assignment

- IR where each variable has only one definition in the program text
  - This is a single *static* definition, but that definition can be in a loop, function, or other code that is executed dynamically many times
- Makes many analyses (and related optimizations) more efficient
- Separates values from memory storage locations
- Complementary to CFG/DFG – better for some things, but cannot do everything

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 definition point.

- 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

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

→

if (…)
  a_1 = x;
else
  a_2 = y;
b_1 = ??;

Merge Points

- The issue is how to handle merge points

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

```
if (…)
  a1 = x;
else
  a2 = y;
a3 =Φ(a1, a2);
b1 = a3;
```

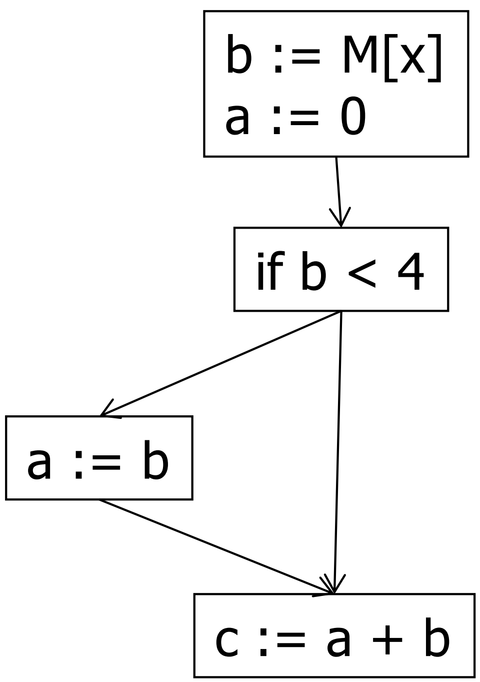
- Solution: introduce a Φ-function

  $a_3 := \Phi(a_1, a_2)$

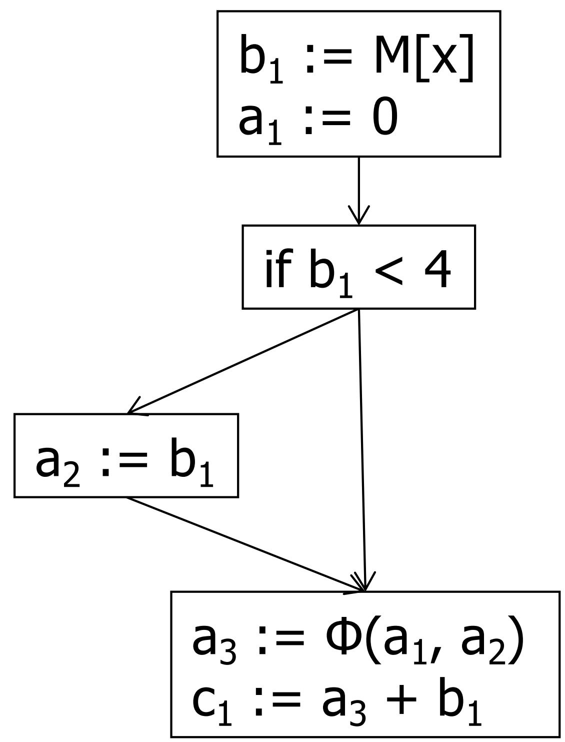
- Meaning: a_3 is assigned either a_1 or a_2 depending on which control path is used to reach the Φ-function

Another Example

Original



SSA

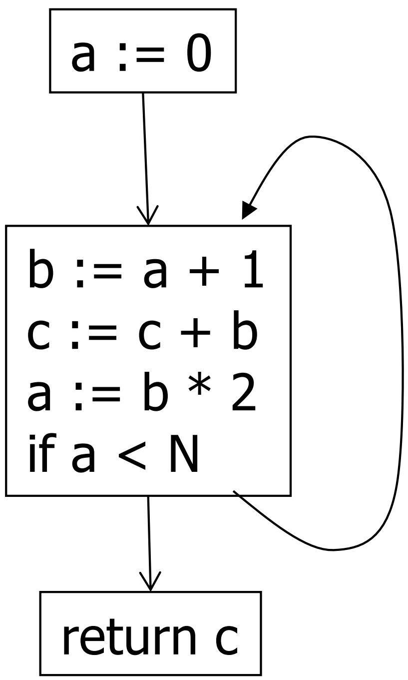


How Does Φ "Know" What to Pick?

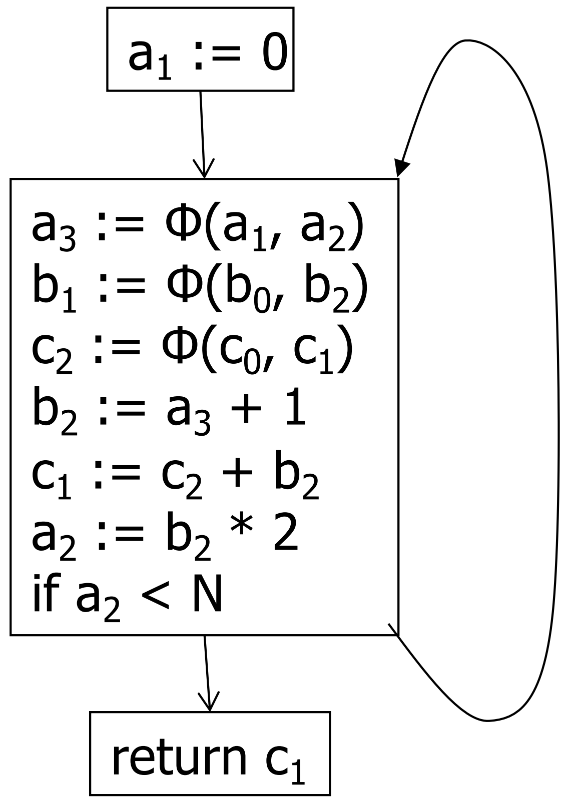
- It doesn't
- Φ-functions don't actually exist at runtime
  - When we're done using the SSA IR, we translate back out of SSA form, removing all Φ-functions
    - Basically by adding code to copy all SSA x_i values to the single, non-SSA, actual x
  - For analysis, all we typically need to know is the connection of uses to definitions – no need to "execute" anything

Example With a Loop

Original



SSA



Notes:

- Loop back edges are also merge points, so require Φ-functions
- a_0, b_0, c_0 are initial values of a, b, c on entry to initial block
- b_1 is dead – can delete later
- c is live on entry – either input parameter or uninitialized

What does SSA “buy” us?

- No need for DU or UD chains – implicit in SSA
- Compact representation
- SSA is “recent” (i.e., 80s)
- Prevalent in real compilers for { } languages

Converting To SSA Form

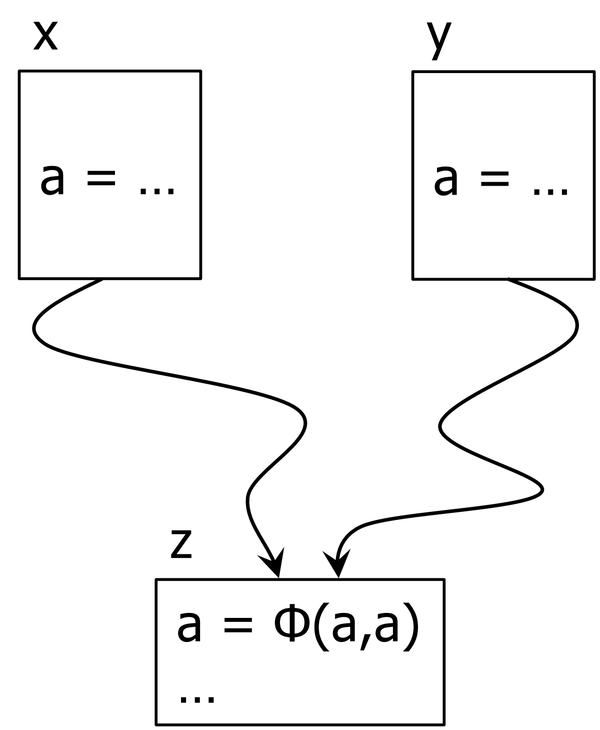
- Basic idea
  - First, add Φ-functions
  - Then, rename all definitions and uses of variables by adding subscripts

Inserting Φ-Functions

- Could simply add Φ-functions for every variable at every join point(!)
- Called "maximal SSA"
- But
  - Wastes *way* too much space and time
  - Not needed in many cases

Path-convergence criterion

- Insert a Φ-function for variable a at point 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 flow graph is considered to define every variable (even if “undefined”)
- 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
- How can we do this efficiently?
  Use a new concept: dominance frontiers

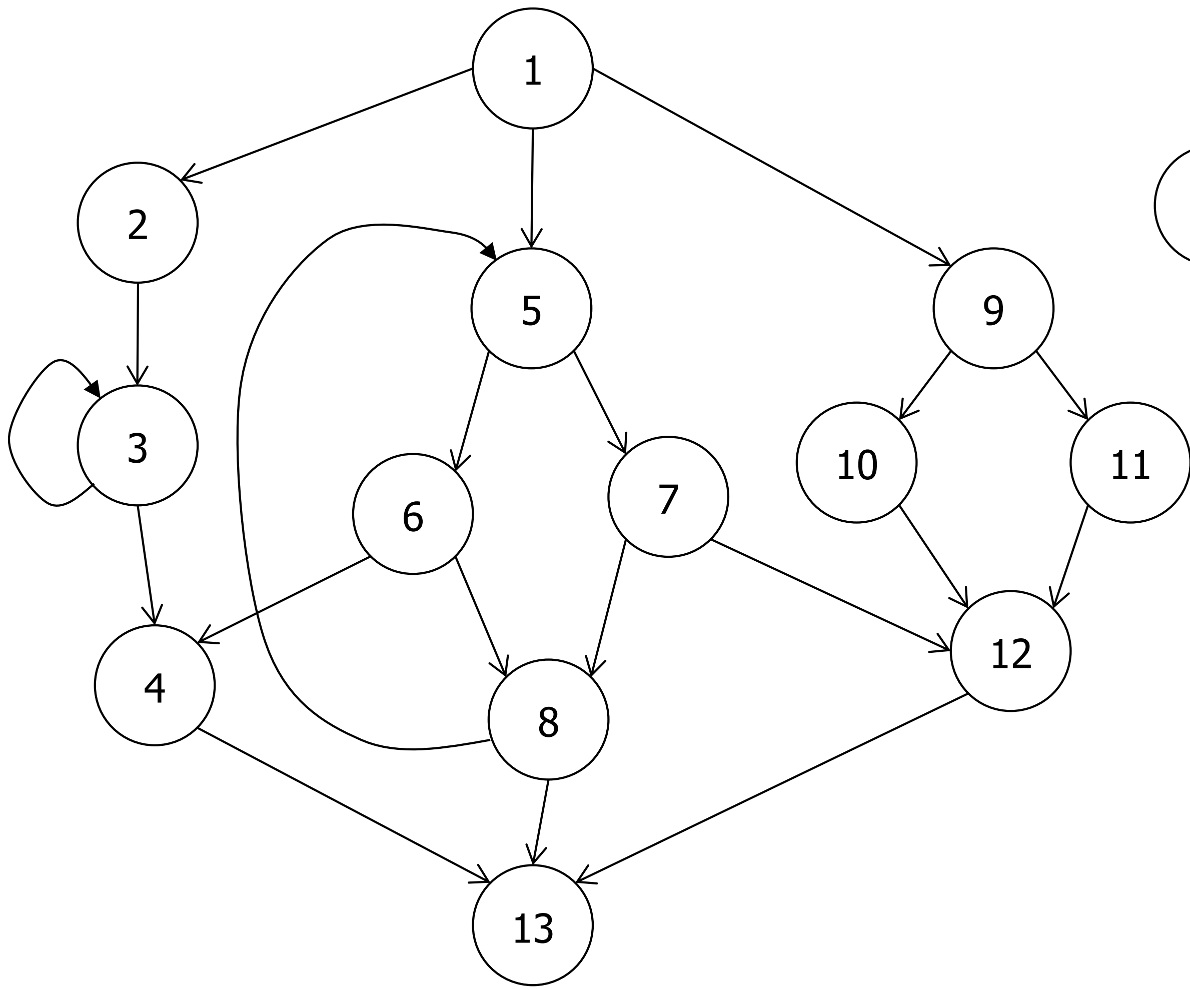
Dominators (review)

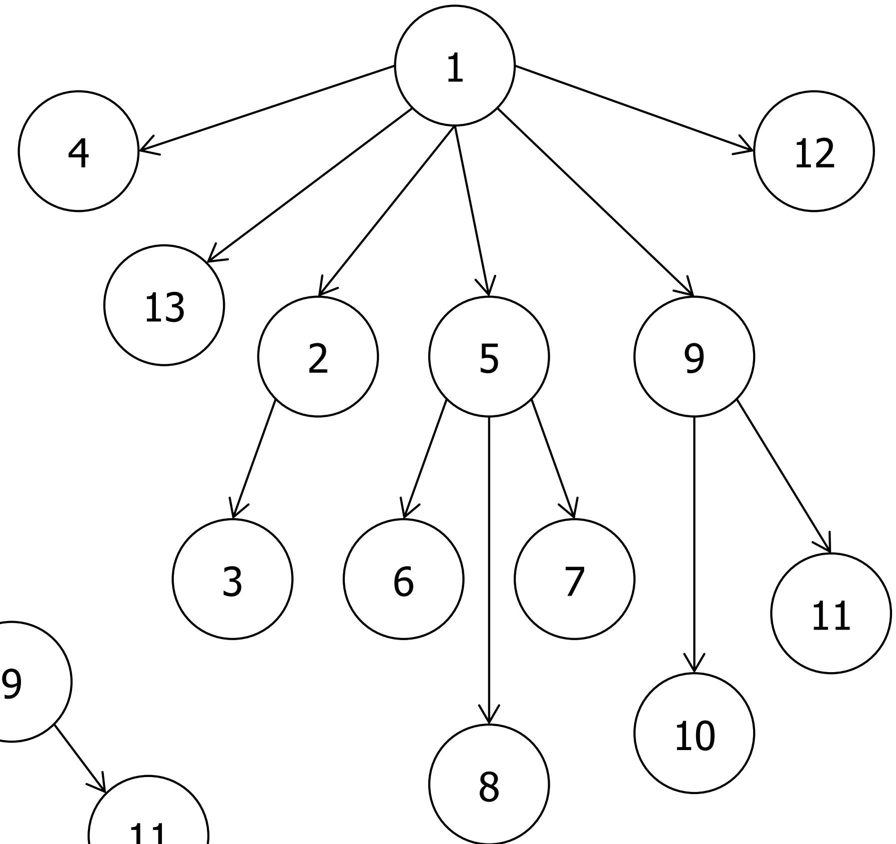
- Definition: a block x *dominates* a block y iff every path from the entry of the control-flow graph to y includes x
- So, by definition, x dominates x
- We can associate a Dom(inator) set with each CFG node x – set of all blocks *dominated by* x

  $| \text{Dom}(x) | \geq 1$

- Properties:
  - Transitive: if a dom b and b dom c, then a dom c
  - There are no cycles, thus can represent the dominator relationship as a tree
- WARNING: this is different from the DOM(x) relationship we used with loops – that was the set of all blocks that *dominate* x (sigh)

Example





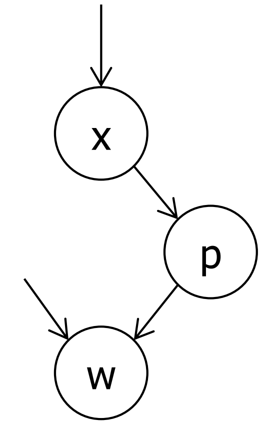
Dominators and SSA

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

Dominance Frontier (1)

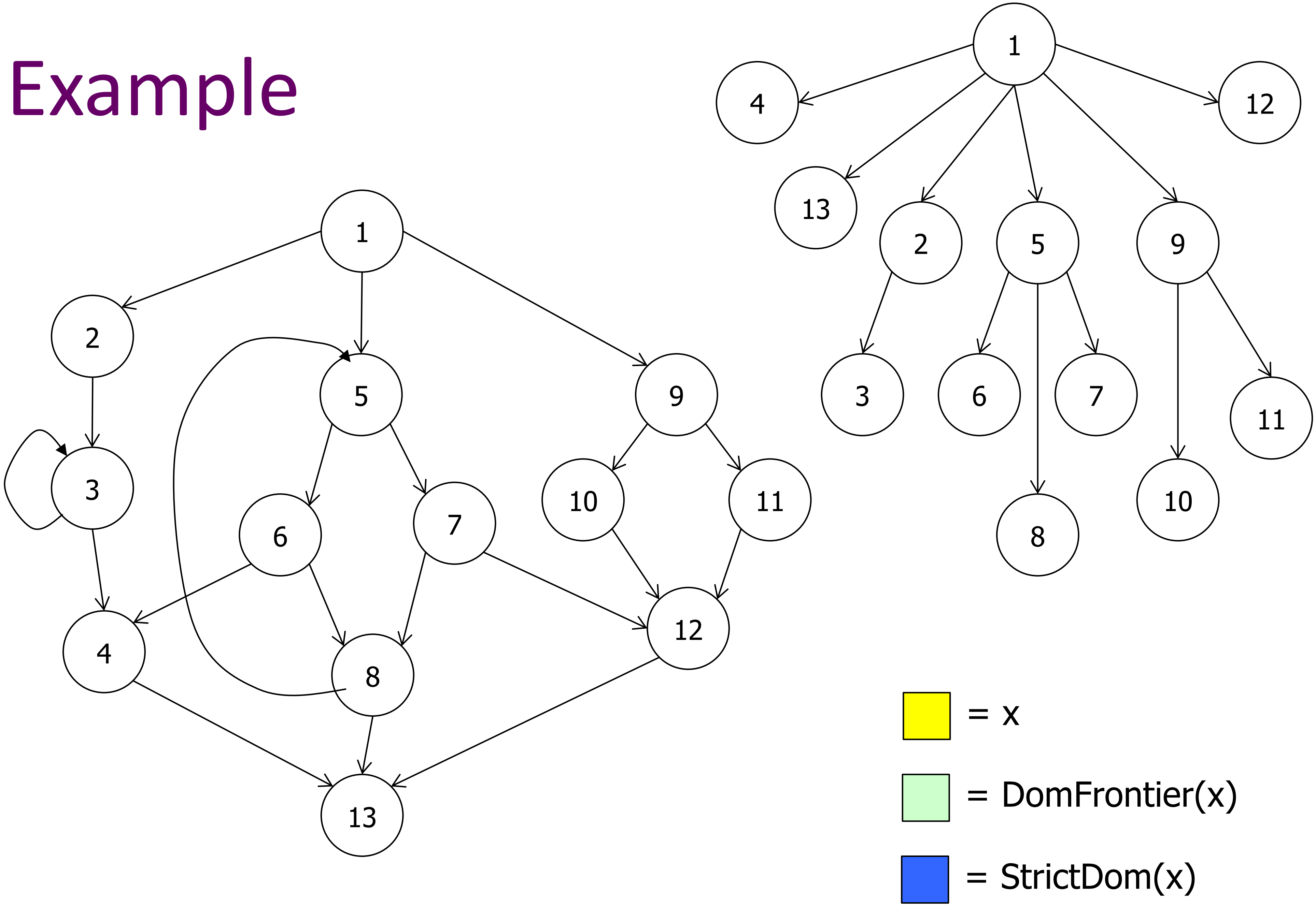
- To get a practical algorithm for placing Φ-functions, we need to avoid looking at all combinations of nodes leading from x to y
- Instead, use the dominator tree in the flow graph

Dominance Frontier (2)

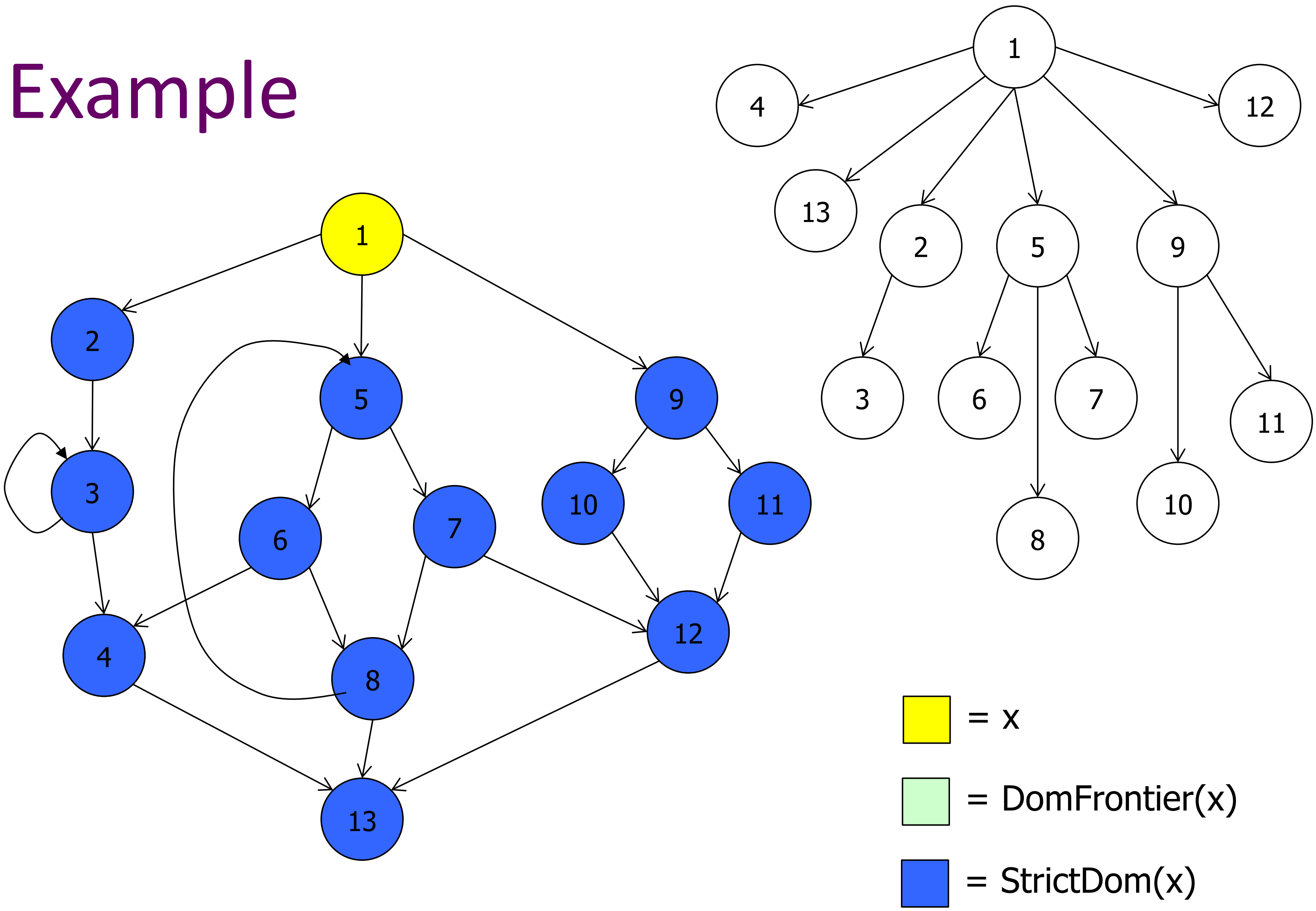


- Definitions
  - x *strictly dominates* y if x dominates y and $x \neq y$
  - The *dominance frontier* of a node x is the set of all nodes w such that x dominates a predecessor of w, but x does not strictly dominate w
    - This means that x can be in *it's own* dominance frontier! That can happen if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

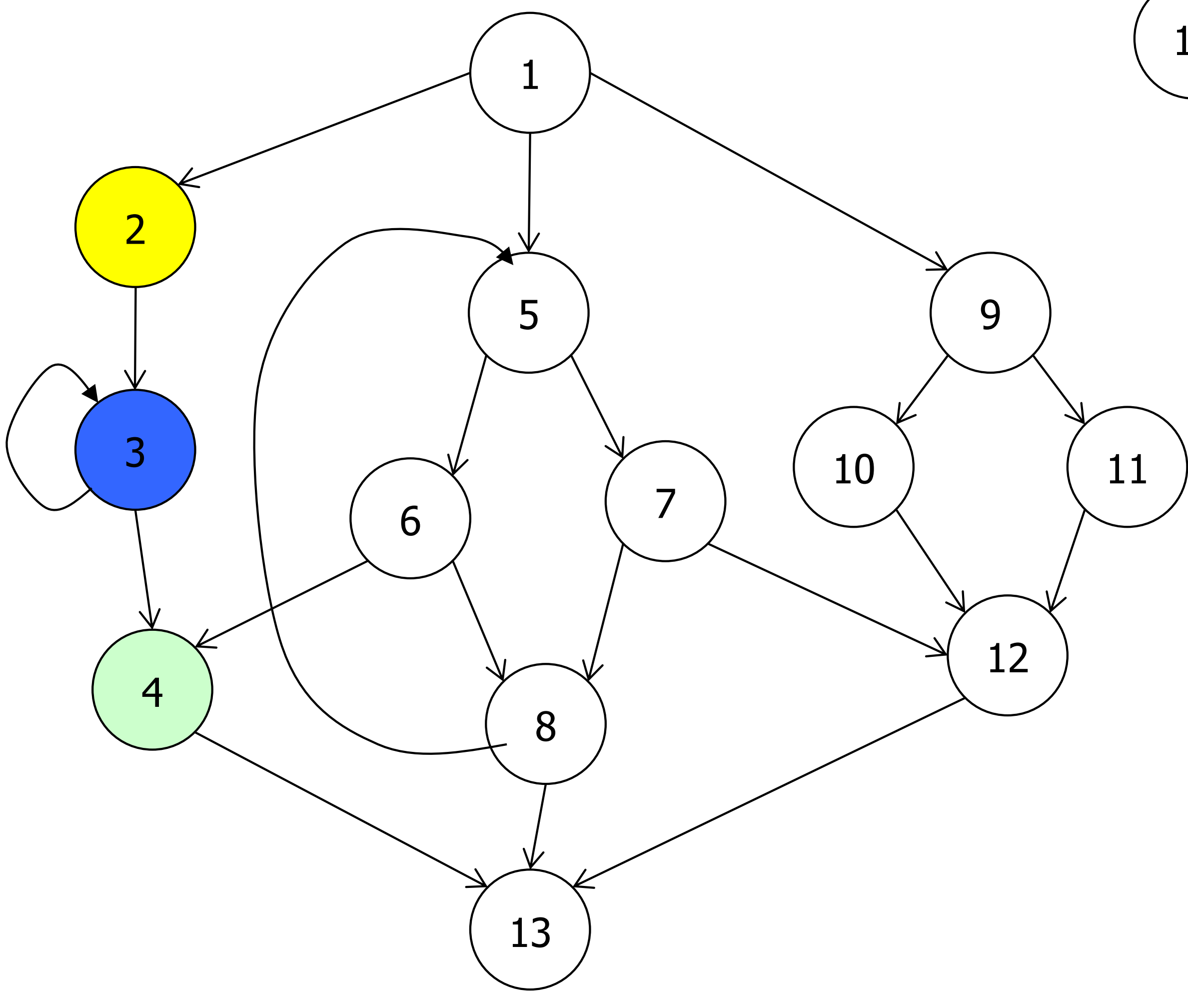
Example

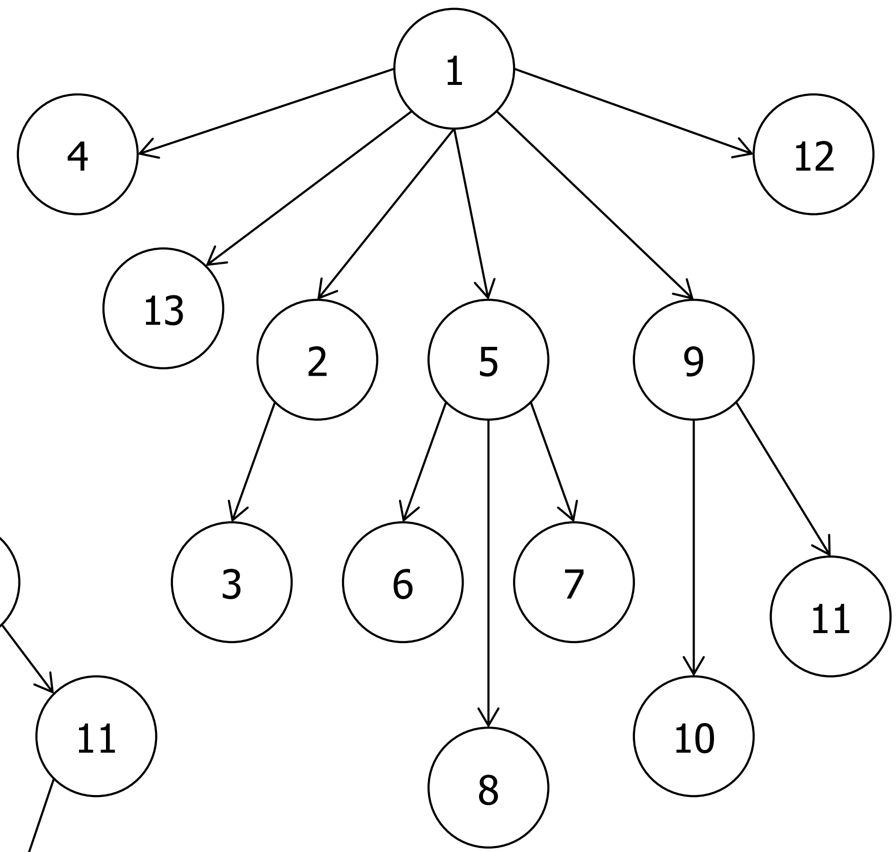


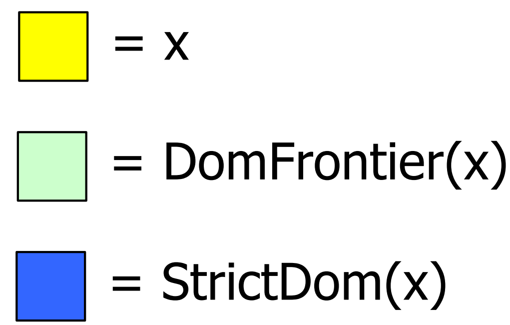
Example



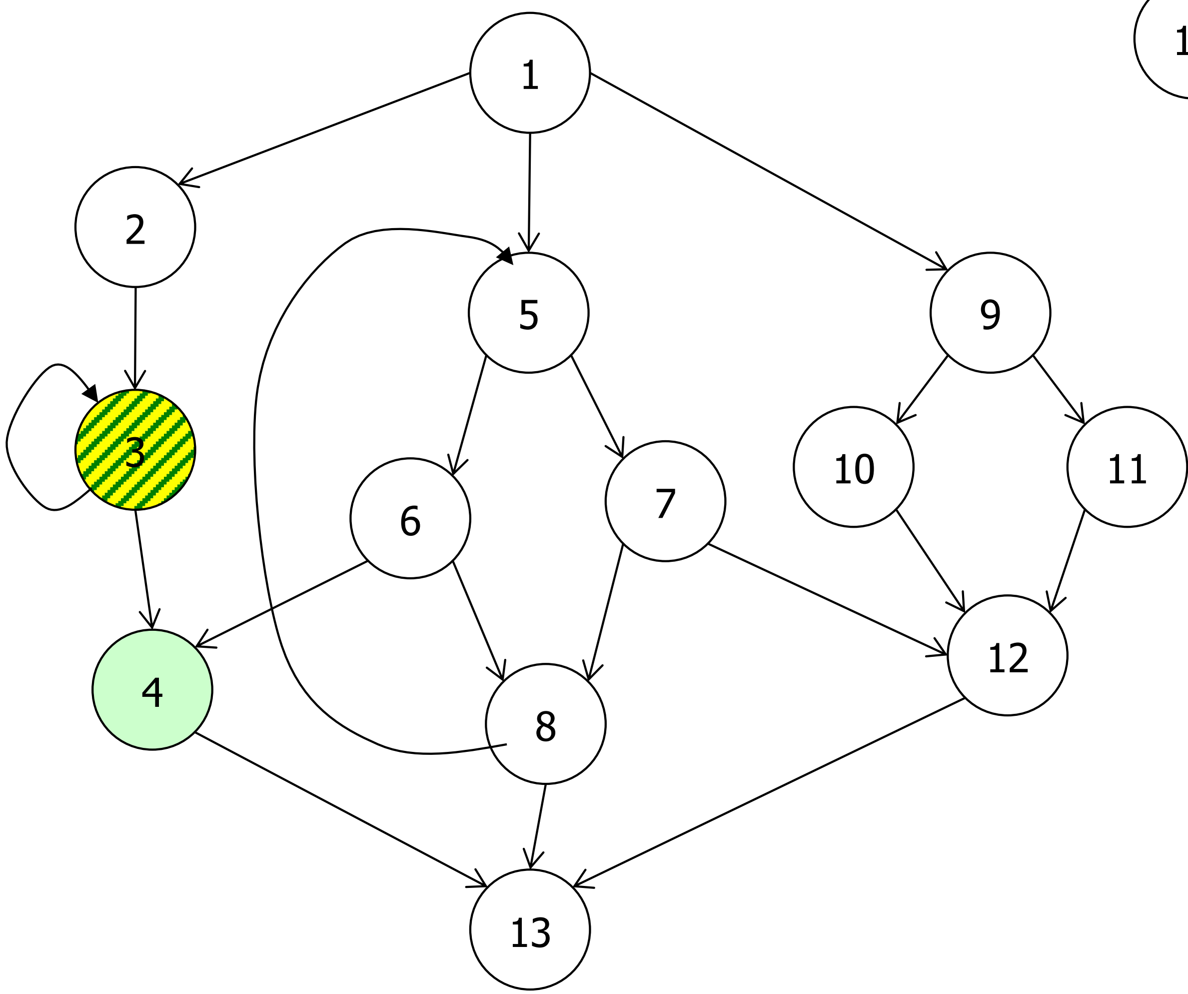
Example

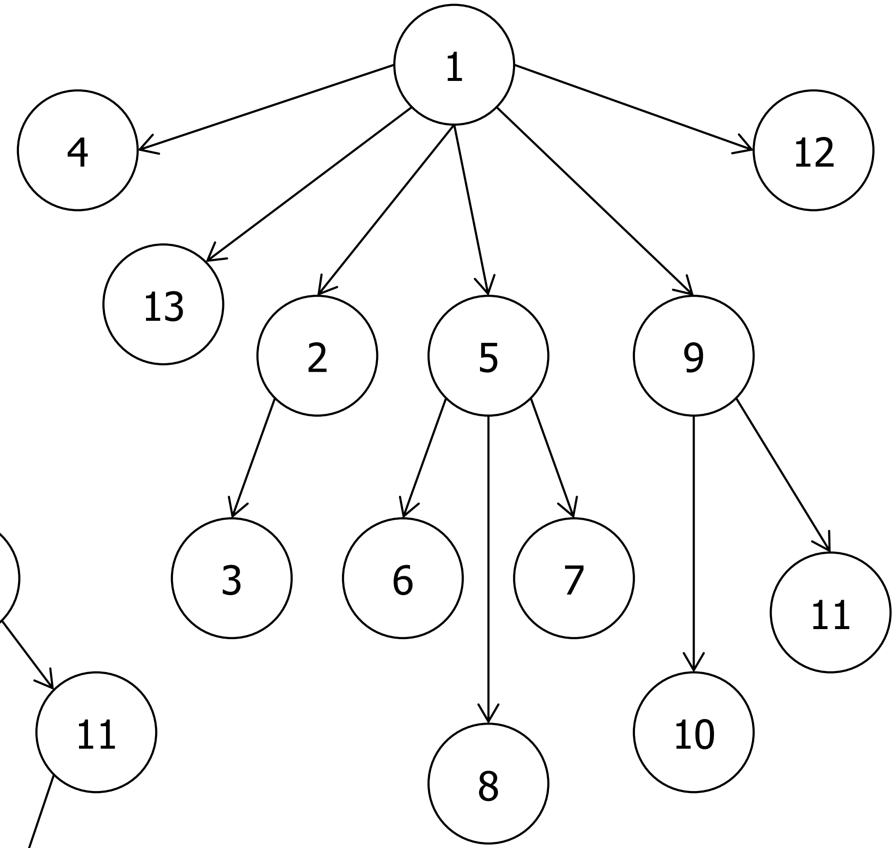

1
2
3
4
5
6
7
8
9
10
11
12
13


1
4
12
13
2
5
9
3
6
7
11
8
10


= x
= DomFrontier(x)
= StrictDom(x)

Example



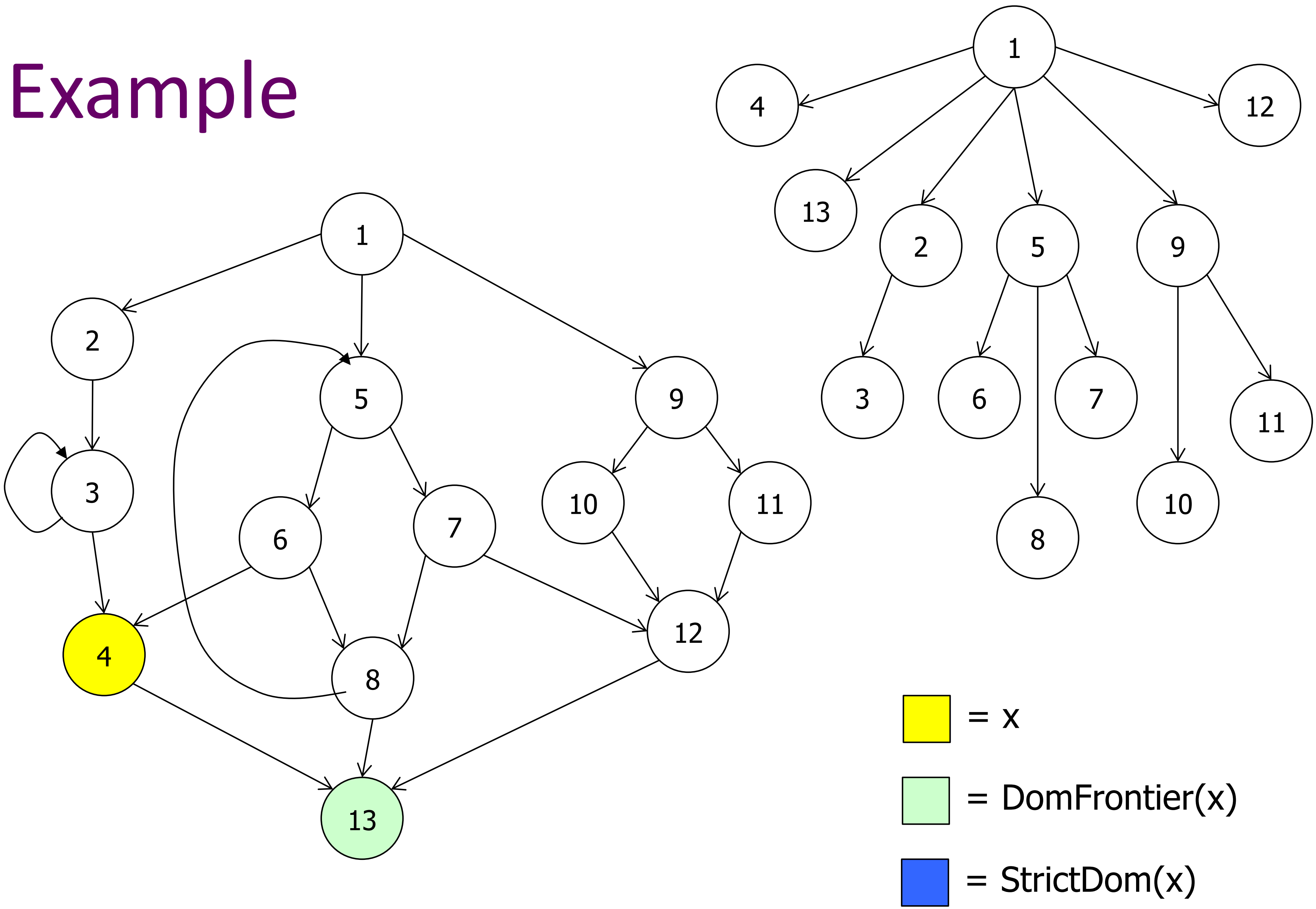


= x

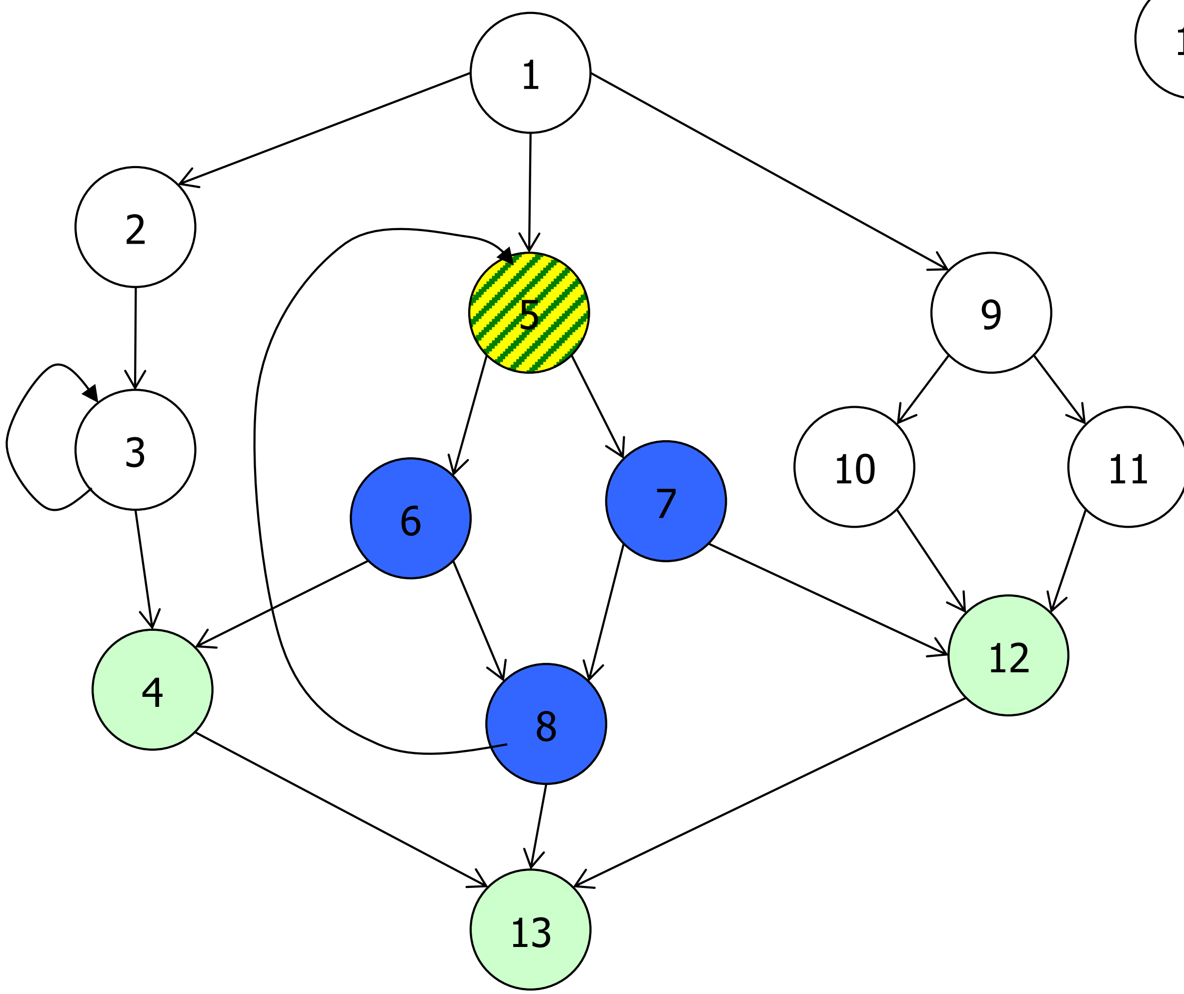
= DomFrontier(x)

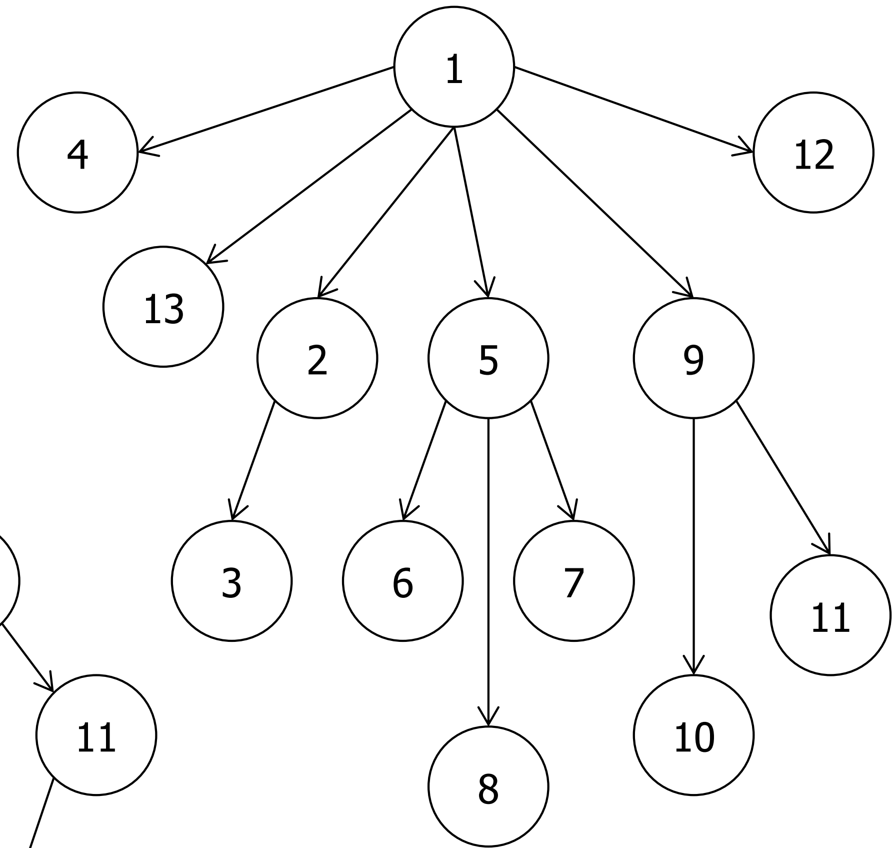
= StrictDom(x)

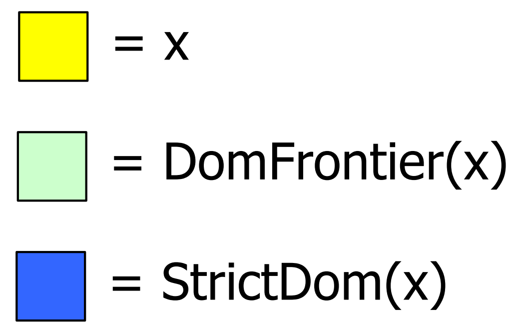
Example


1
2
3
4
5
6
7
8
9
10
11
12
13
1
4
12
13
2
5
9
3
6
7
11
8
10
= x
= DomFrontier(x)
= StrictDom(x)

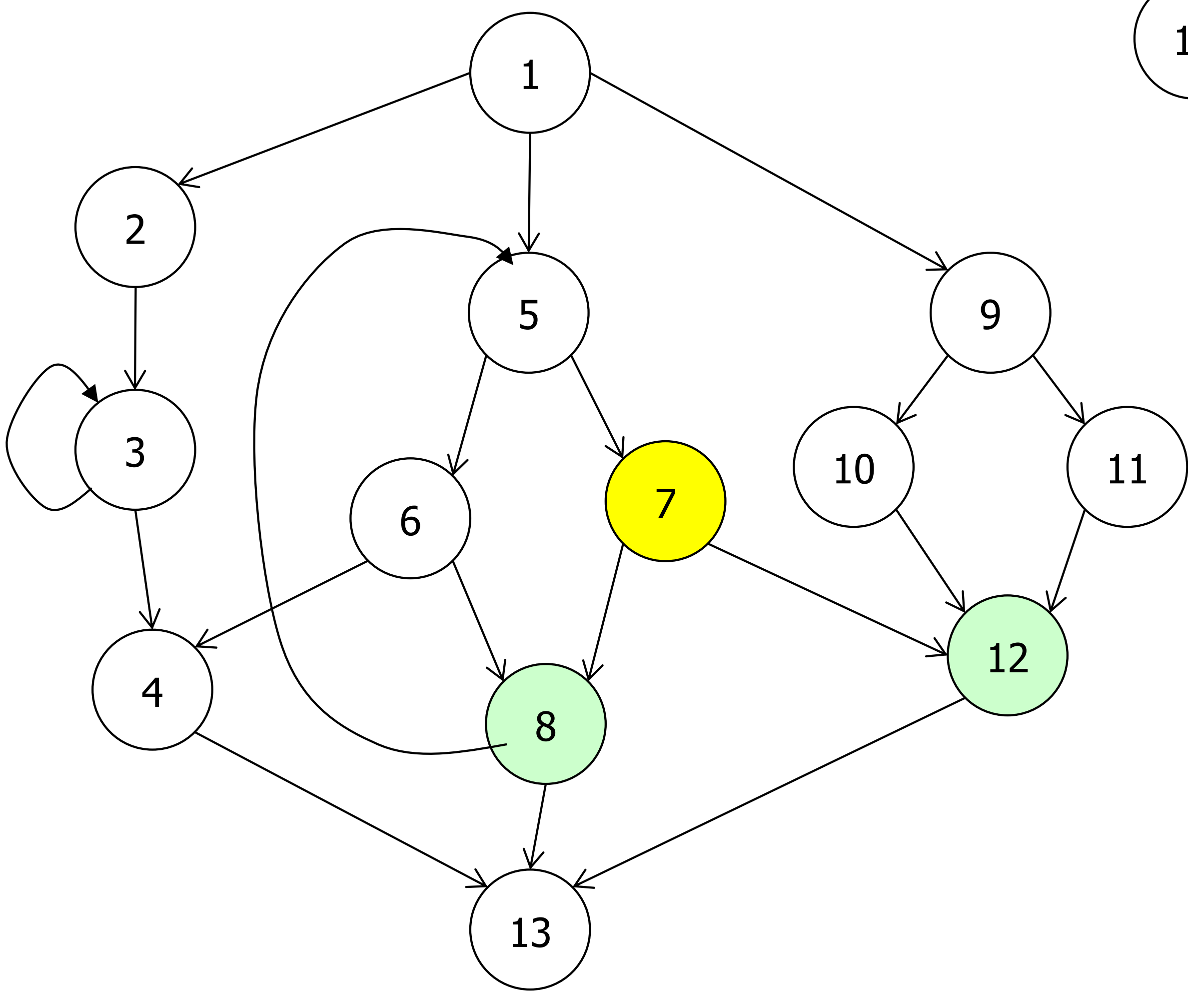
Example

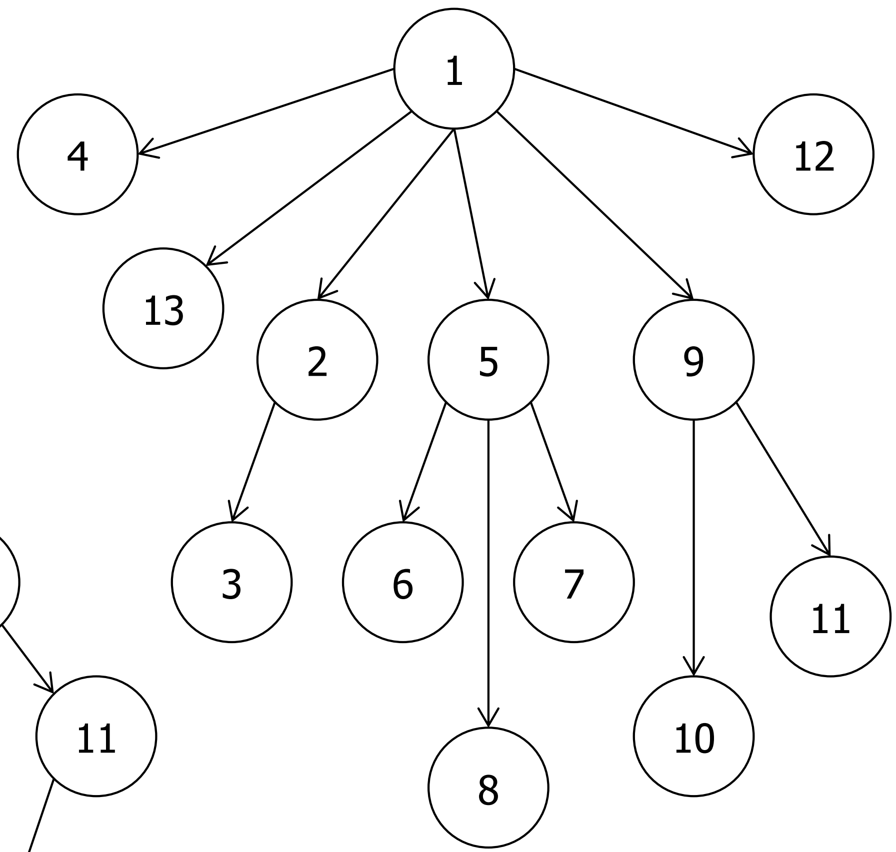

1
2
3
4
5
6
7
8
9
10
11
12
13

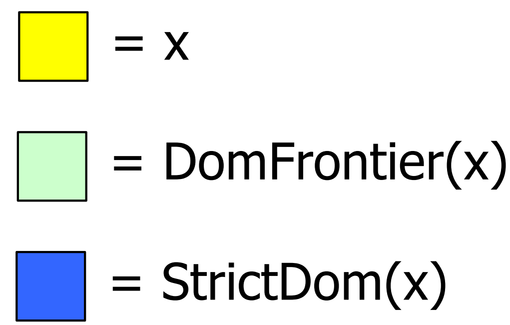

1
4
12
13
2
5
9
3
6
7
11
10
8


= x
= DomFrontier(x)
= StrictDom(x)

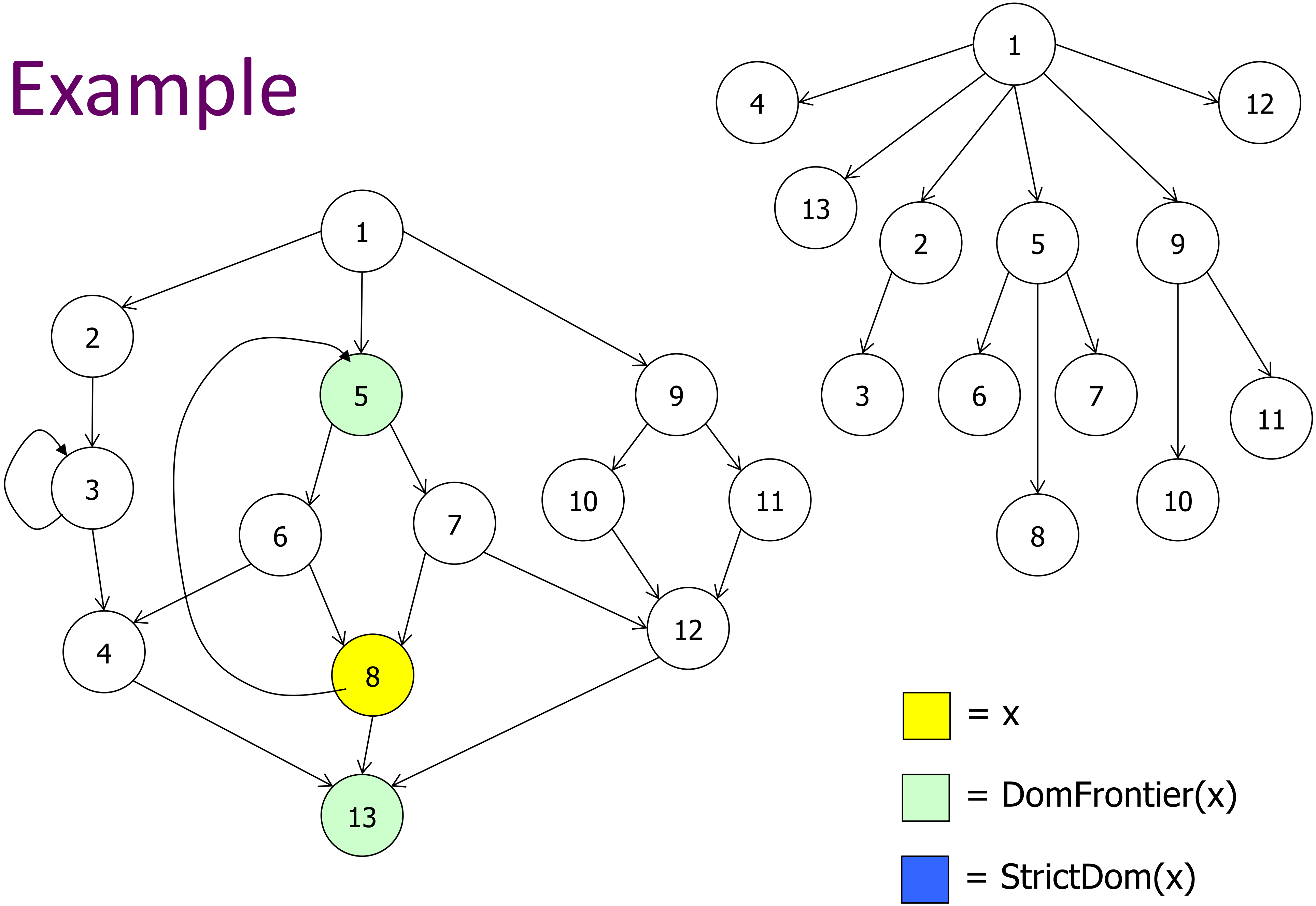
Example


1
2
5
9
3
6
7
10
11
4
8
12
13

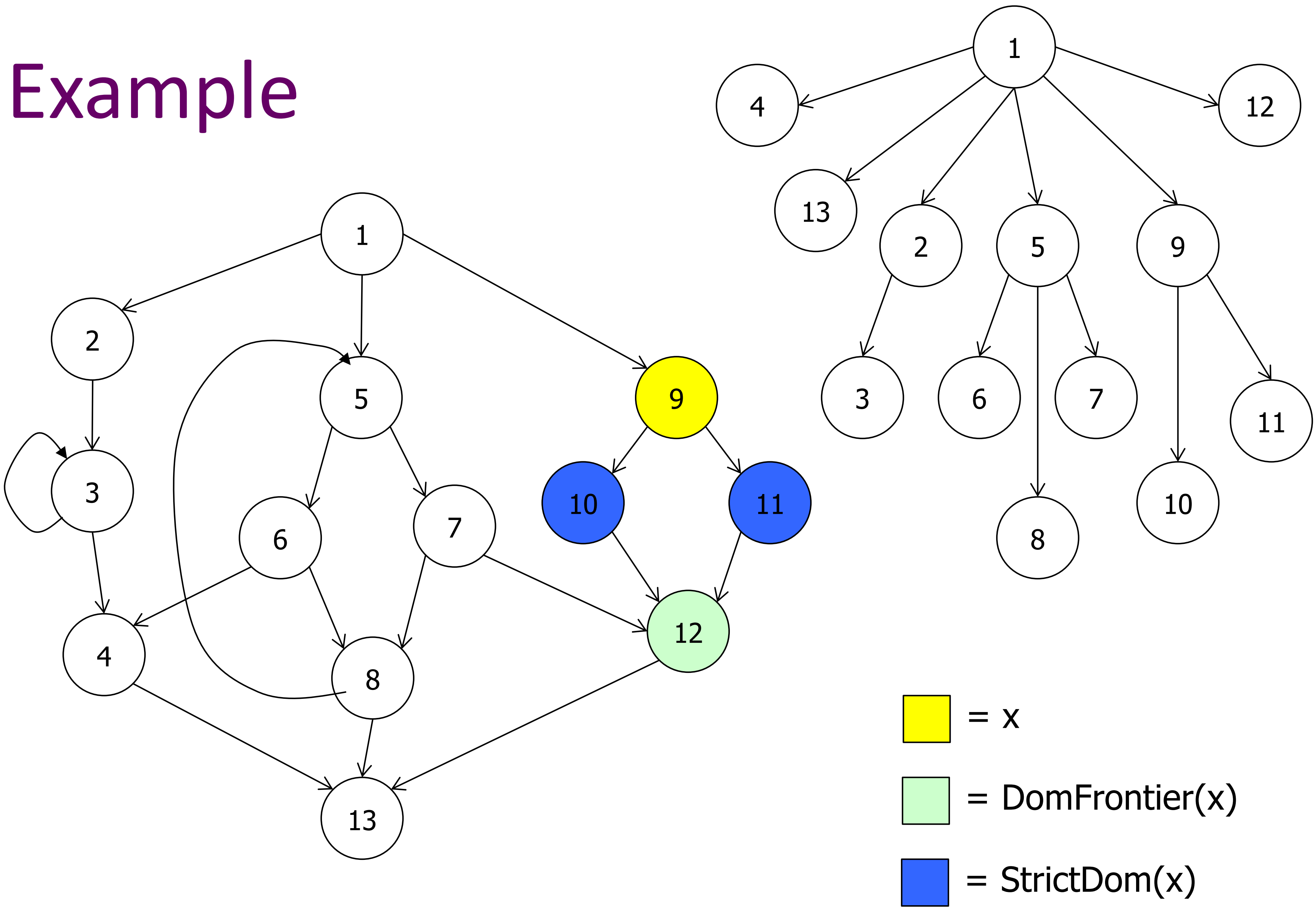

1
4
12
13
2
5
9
3
6
7
11
10
8


= x
= DomFrontier(x)
= StrictDom(x)

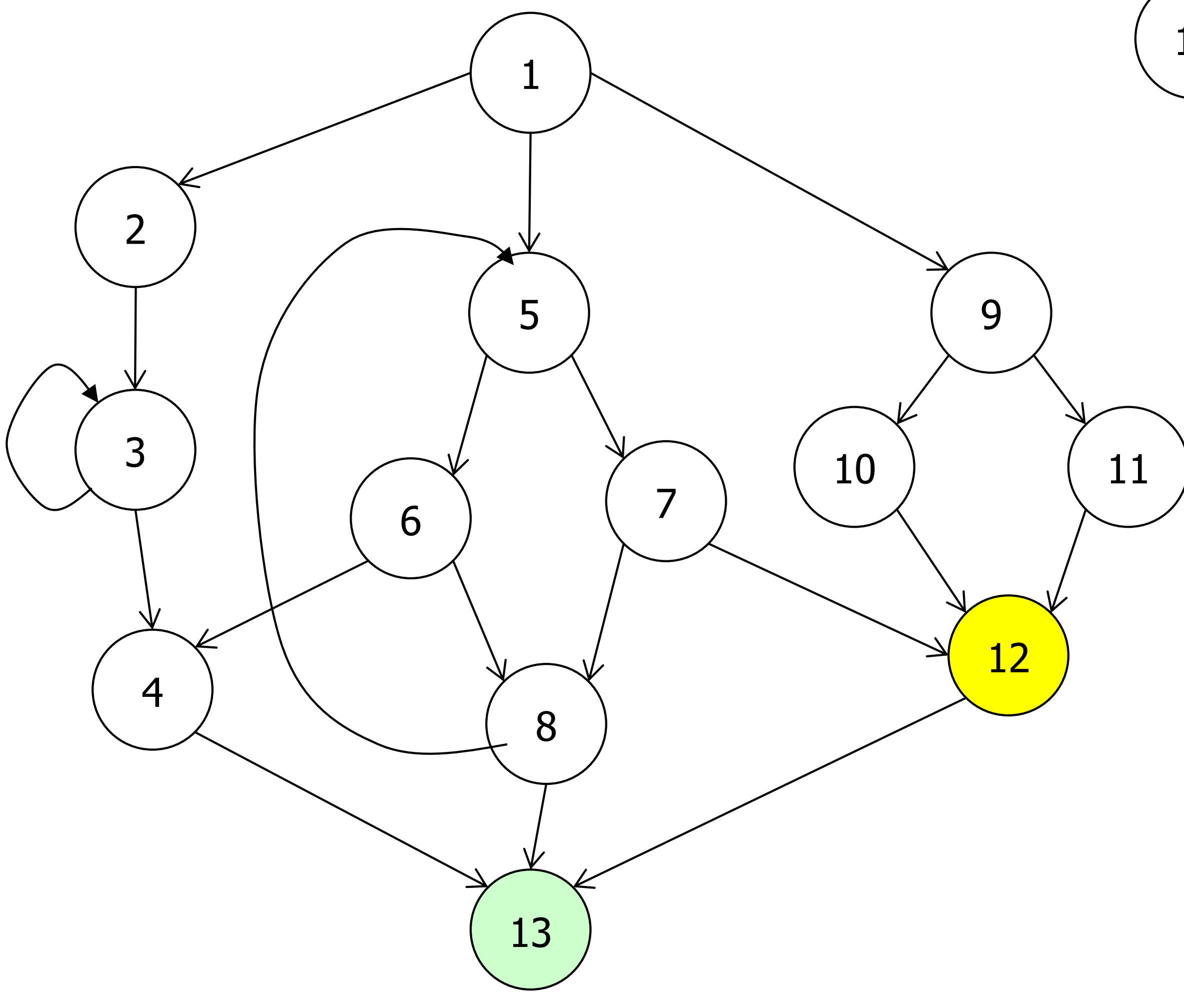
Example



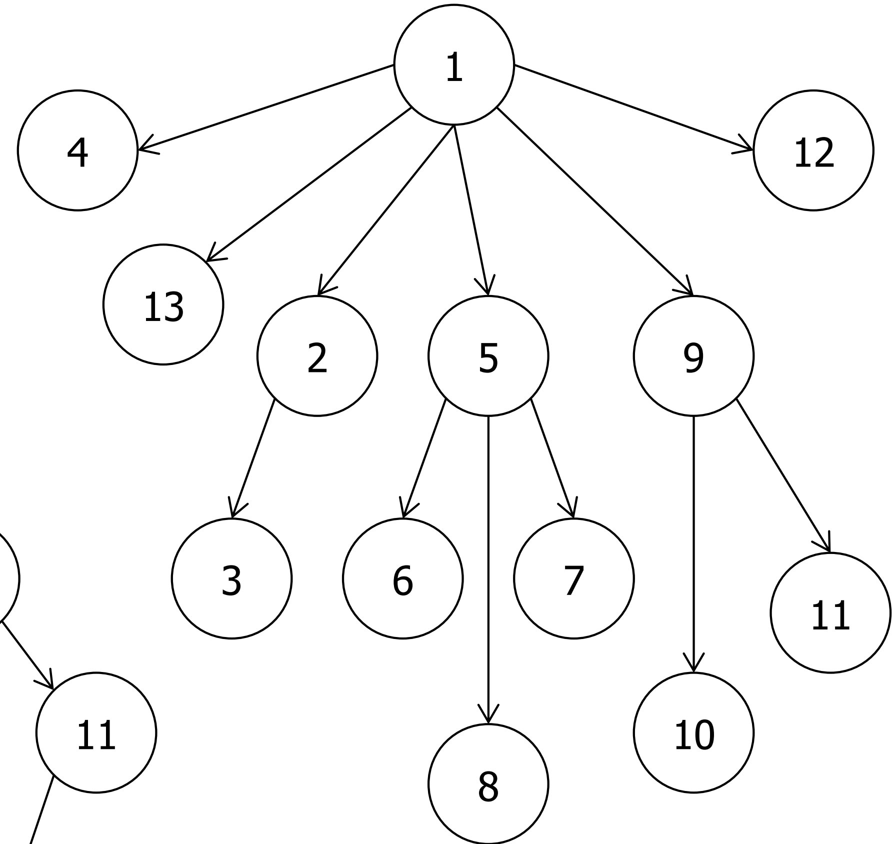
Example



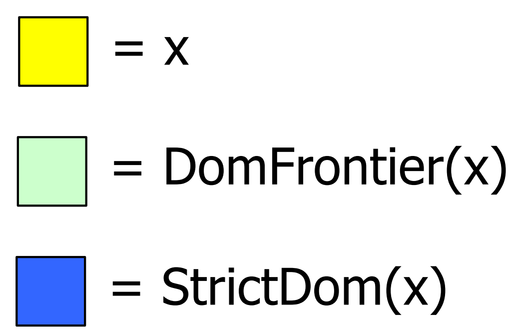
Example



1
2
3
4
5
6
7
8
9
10
11
12
13

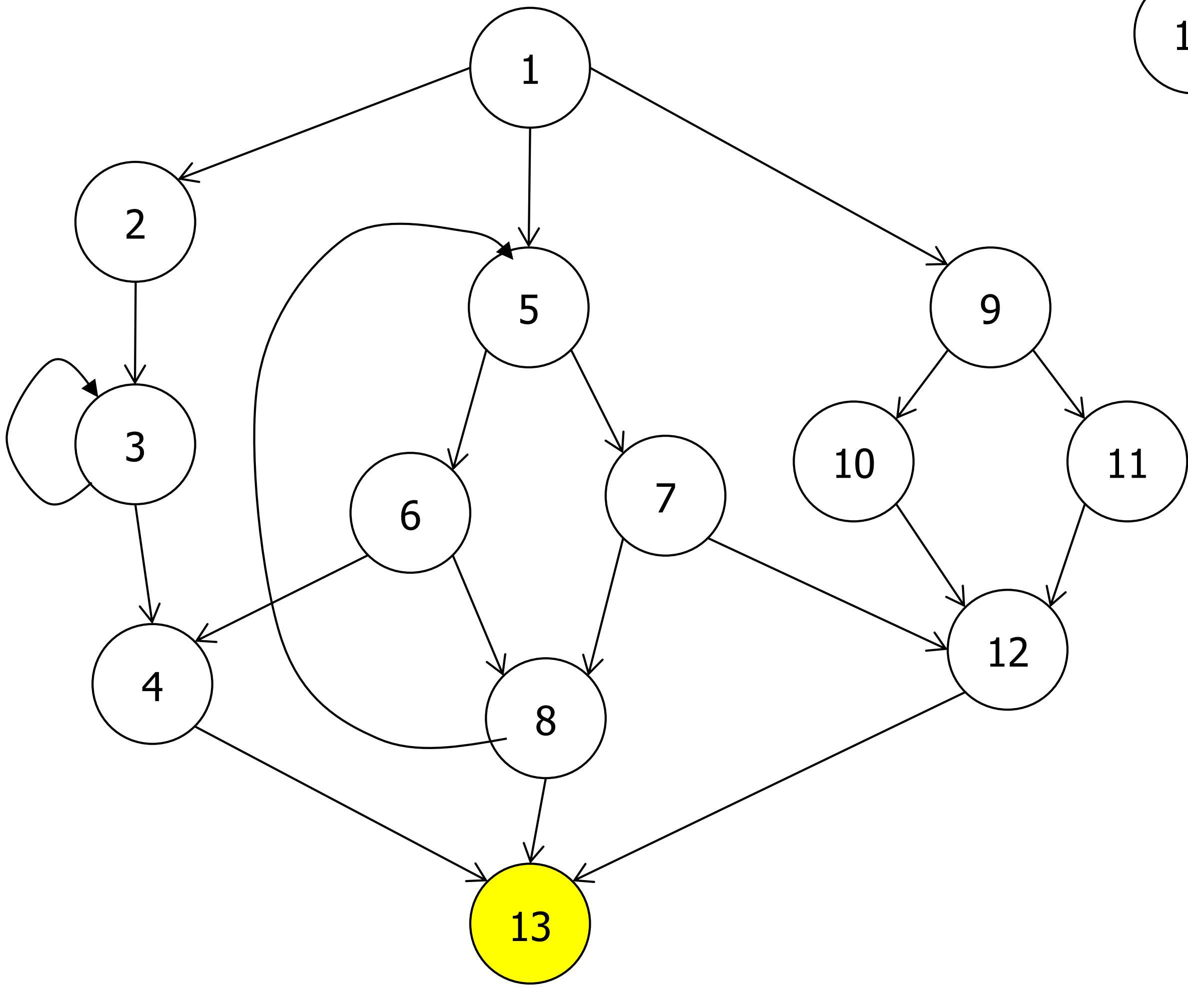


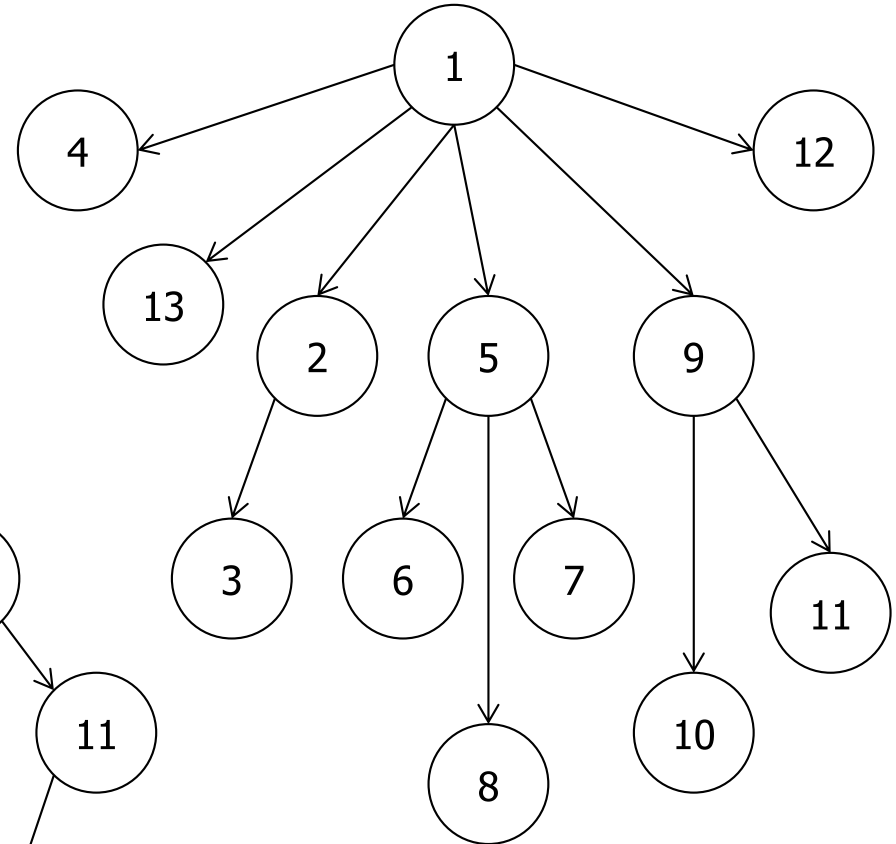
1
4
12
13
2
5
9
3
6
7
11
8
10



= x
= DomFrontier(x)
= StrictDom(x)

Example



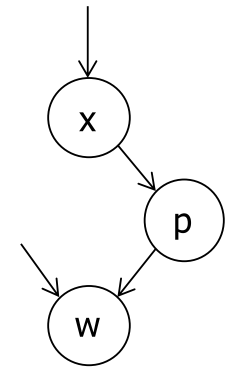


= x

= DomFrontier(x)

= StrictDom(x)

Dominance Frontier Criterion for 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 of a. The dominance frontier represents the first nodes we could have reached via an alternative path, which *will* have an alternate reaching definition of *a* (recall the convention that the entry node defines all variables with version 0 - a_0)
    - Why is this right for loops? Hint: strict dominance…
  - Since the Φ-function itself is a definition, this placement rule 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

- See the book for the full construction, but 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 each variable a to be a_1, a_2, a_3, …

Efficient Dominator Tree Computation

- Goal: SSA makes optimizing compilers faster since we can find definitions/uses without expensive bit-vector algorithms
- So, need to be able to compute SSA form quickly
- Computation of SSA from dominator trees are efficient, but…

Lengauer-Tarjan Algorithm

- Iterative set-based algorithm for finding dominator trees is slow in worst case
- Lengauer-Tarjan is near linear time
  - Uses depth-first spanning tree from start node of control flow graph
  - See books for details

SSA Optimizations

- Why go to the trouble of translating to SSA?
- Because SSA makes many optimizations and analyses simpler and more efficient
  - We'll give a couple of examples
- But first, what do we know? (i.e., what information is kept in the SSA graph?)

SSA Data Structures

For each …

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

Dead-Code Elimination

- A variable is live ⇔ its list of uses is not empty(!)
  - That's it! Nothing further to compute
- 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

Simple Constant Propagation

- If c is a constant in v := c, any use of v can be replaced by c
  - So 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
- 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
  - (Also sometimes needed for different kinds of analysis or transformation. A production optimizer might convert the IR into and out of SSA form multiple times)

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 copy instructions

SSA Wrapup

- More details needed to fully and efficiently implement SSA, but these are the main ideas
  - See recent compiler books (but not the Dragon book!)
- Allows efficient implementation of many optimizations
- SSA is used in most modern optimizing compilers (llvm is based on it) and has been retrofitted into many older ones (gcc is a well-known example)
- Not a silver bullet – some optimizations still need non-SSA forms, but very effective for many