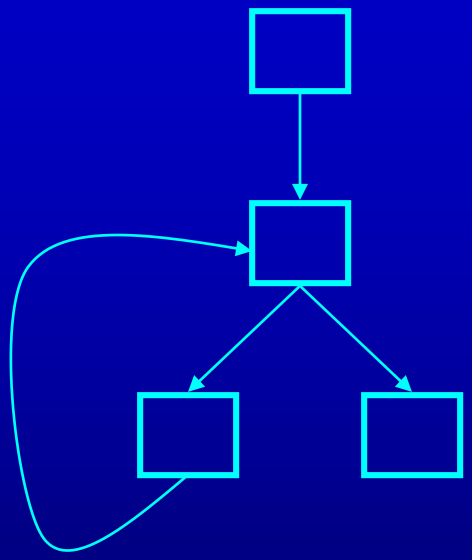
MIT 6.035
Loop Optimizations

Martin Rinard

Loop Optimizations

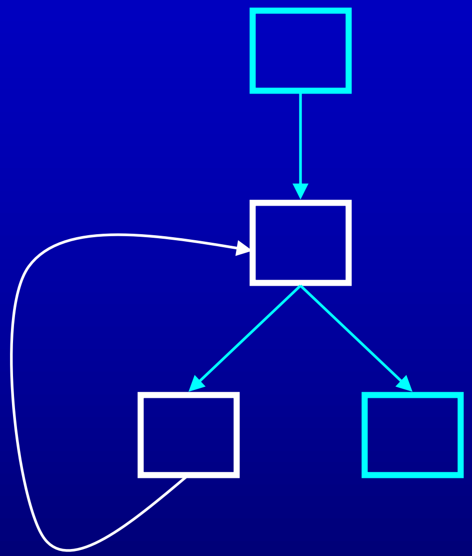
- Important because lots of computation occurs in loops
- We will study two optimizations
  - Loop-invariant code motion
  - Induction variable elimination

What is a Loop?



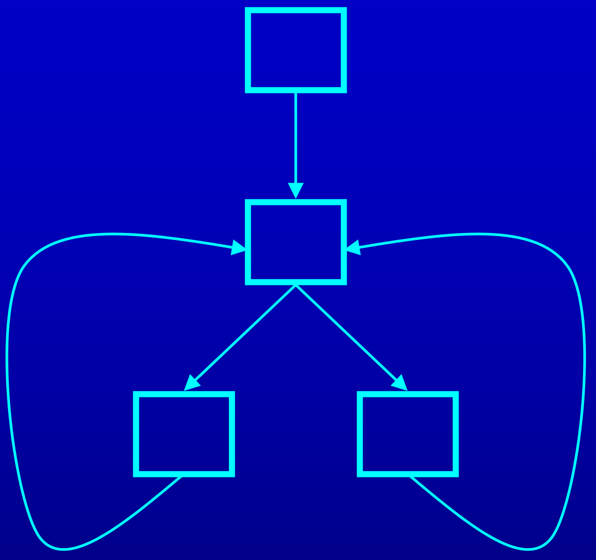
What is a Loop?

- Set of nodes
- Loop header
  - Single node
  - All iterations of loop go through header
- Back edge



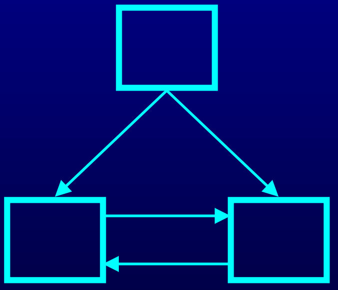
Anamalous Situations

- Two back edges, two loops, one header



- Compiler merges loops

- No loop header, no loop



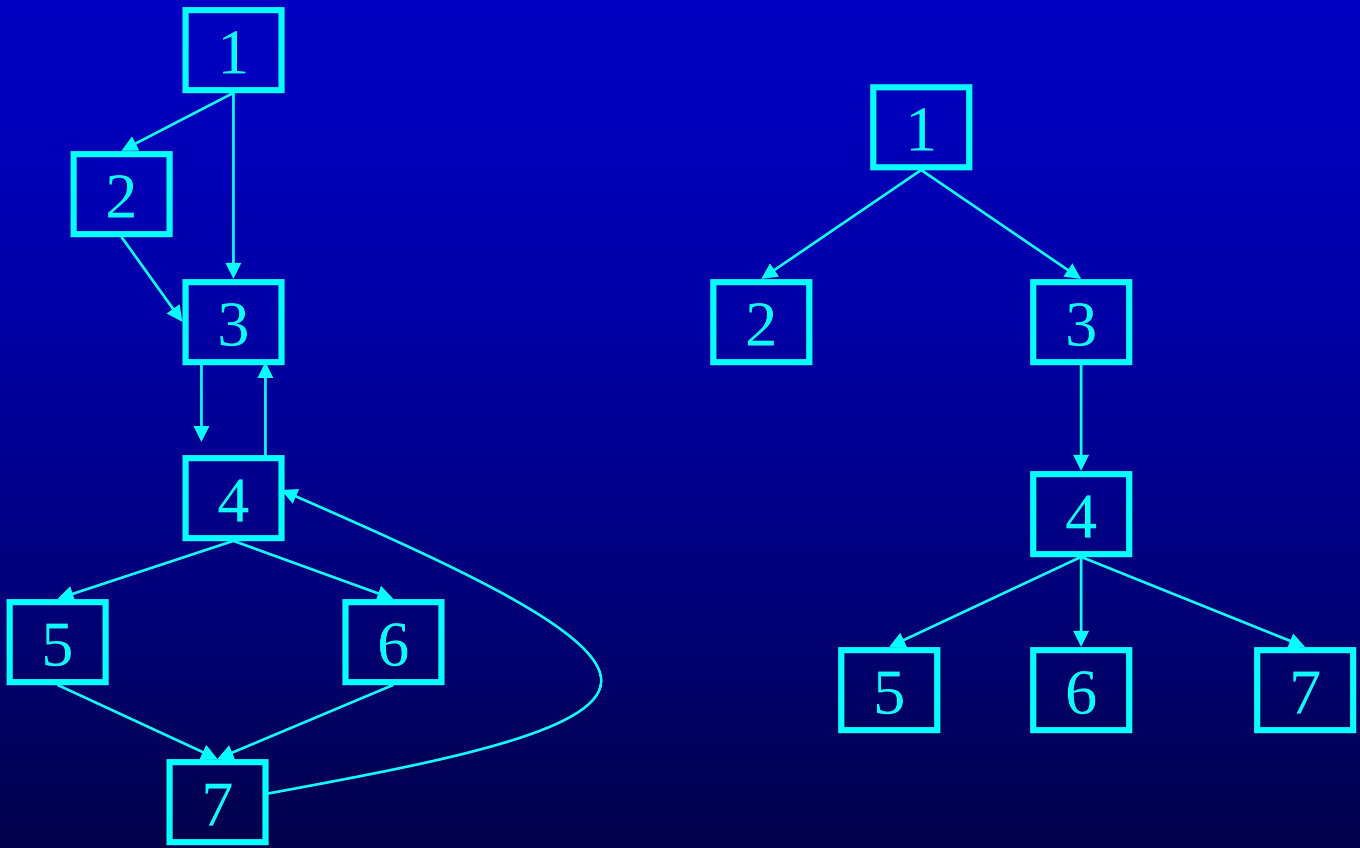
Defining Loops With Dominators

- Concept of dominator
  - Node n dominates a node m if all paths from start node to m go through n
  - "The road to the Super Bowl goes through New England" Conclusion? New England dominates the Super Bowl!
- If d_1 and d_2 both dominate n, then either
  - d_1 dominates d_2, or
  - d_2 dominates d_1 (but not both – look at path from start)
- Immediate dominator n – last dominator of n on any path from start node

Dominator Tree

- Nodes are nodes of control flow graph
- Edge from d to n if d immediate dominator of n
- This structure is a tree
- Rooted at start node

Example Dominator Tree



Dominator Conditions

- When does n dominate m?
  - When n dominates all predecessors of m
  - When n = m (convenient default)
- Suggests dataflow-like algorithm for computing dominators

Dominator Algorithm

$D(n_0) = \{n_0\}$

for n in N – $\{ n_0\}$ do $D(n) = N$

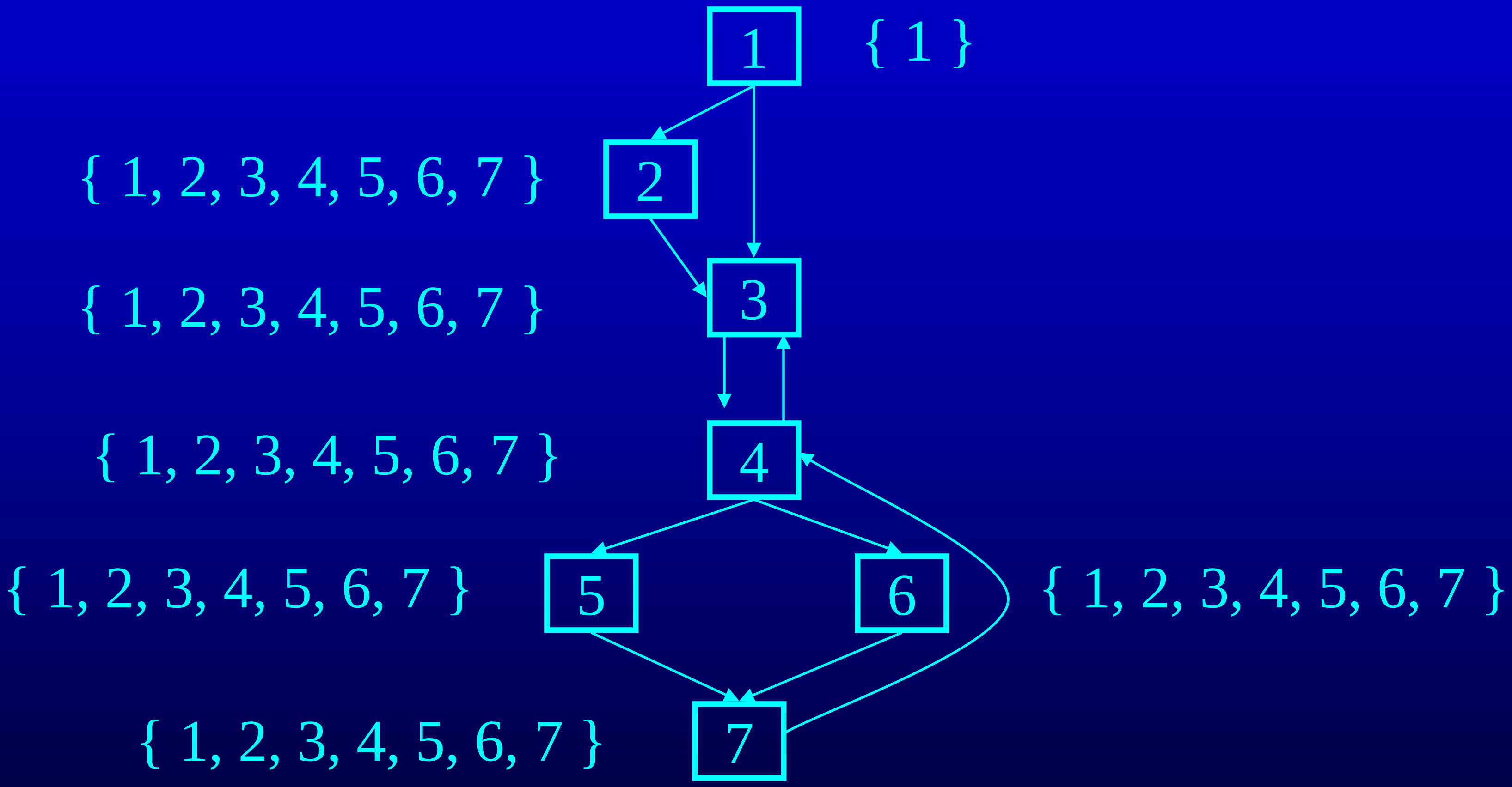
while D changes do

  for $n \in N - \{ n_0\}$ do

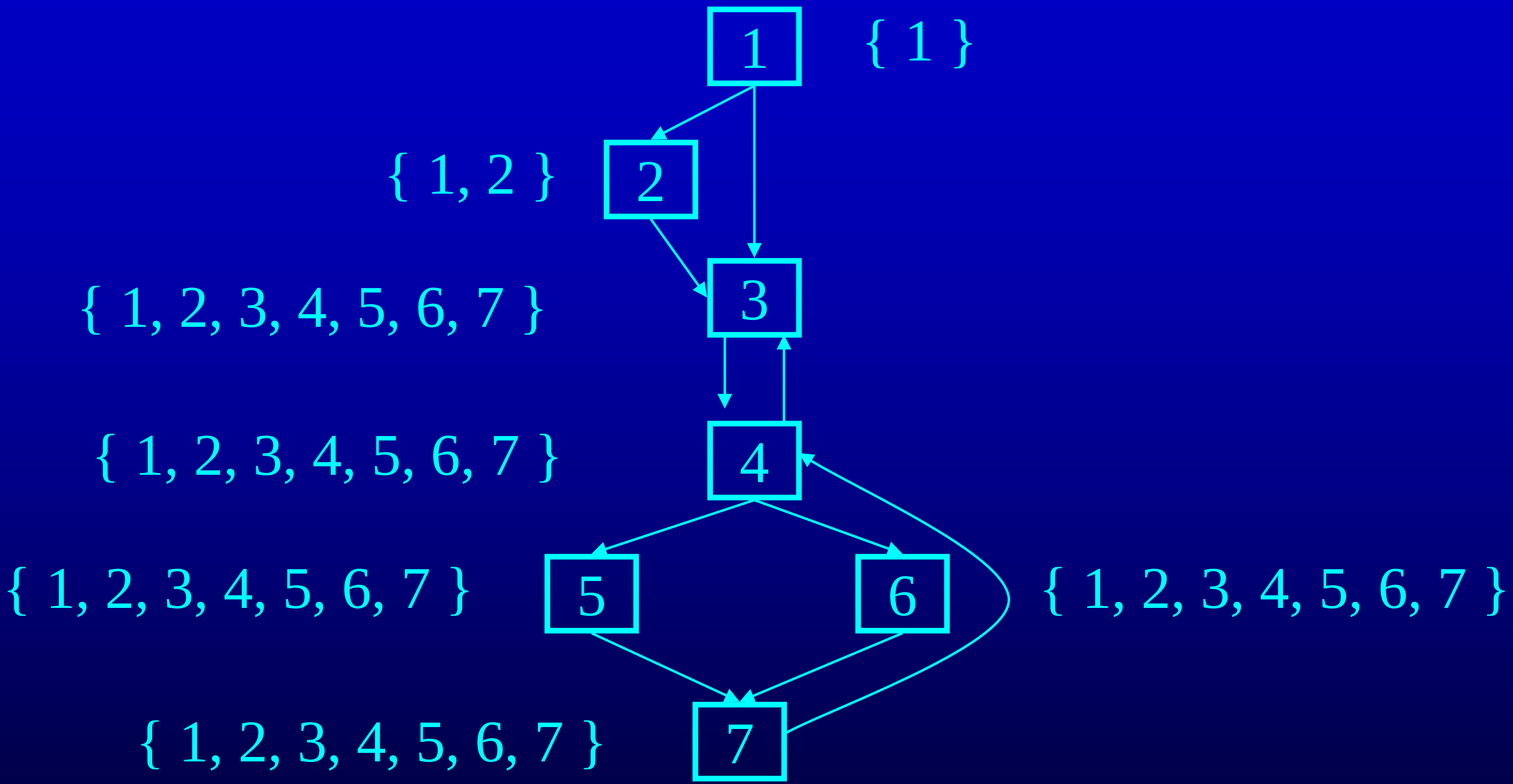
    $$D(n) = \{ n \} \cup \bigcap_{p \in pred(n)} D(p)$$

- Why does the algorithm complete?
- Why does the algorithm get right answer?
- Note complete recomputation of D on every iteration
- Could run a worklist algorithm

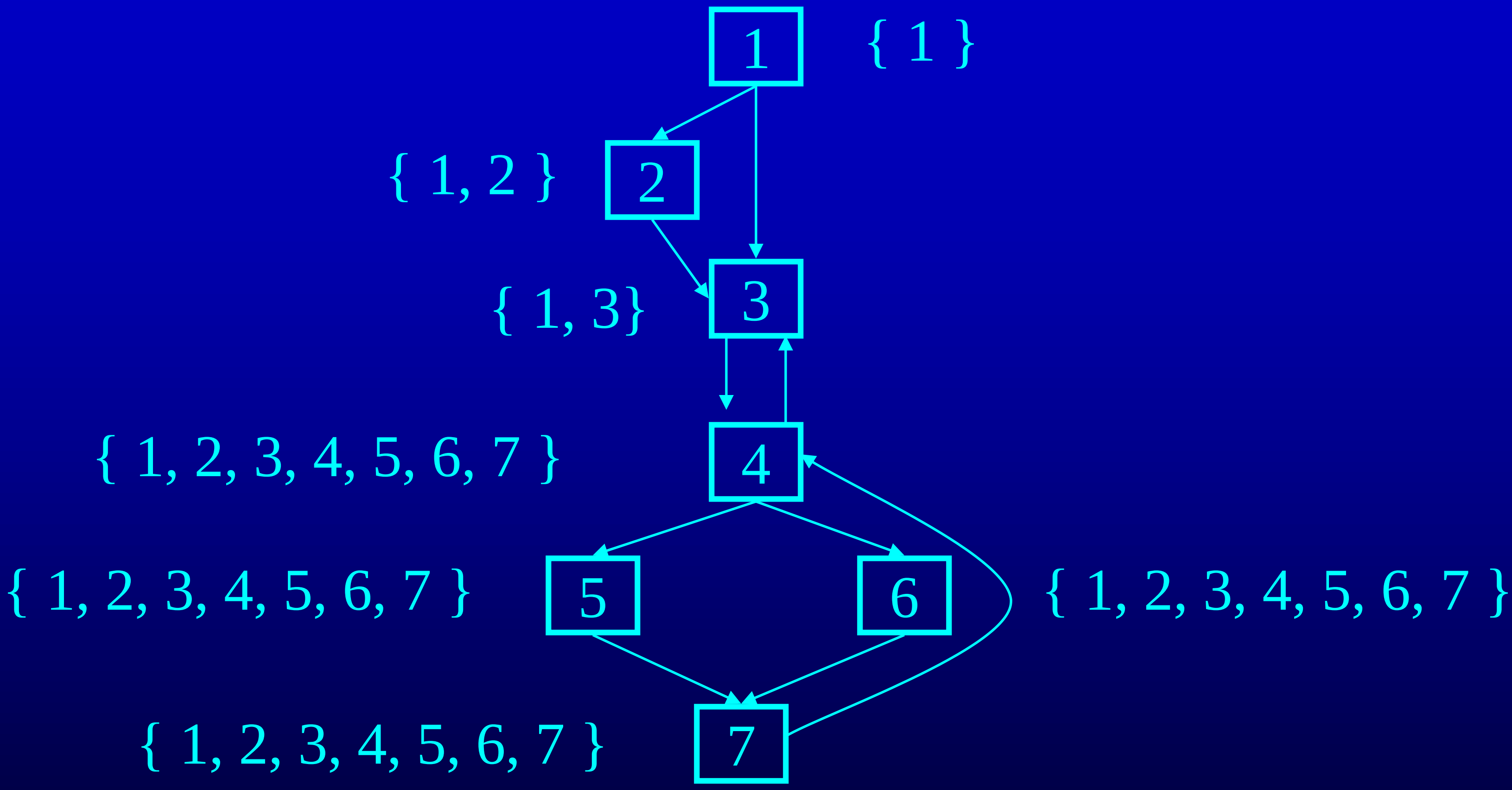
Dominator Computation



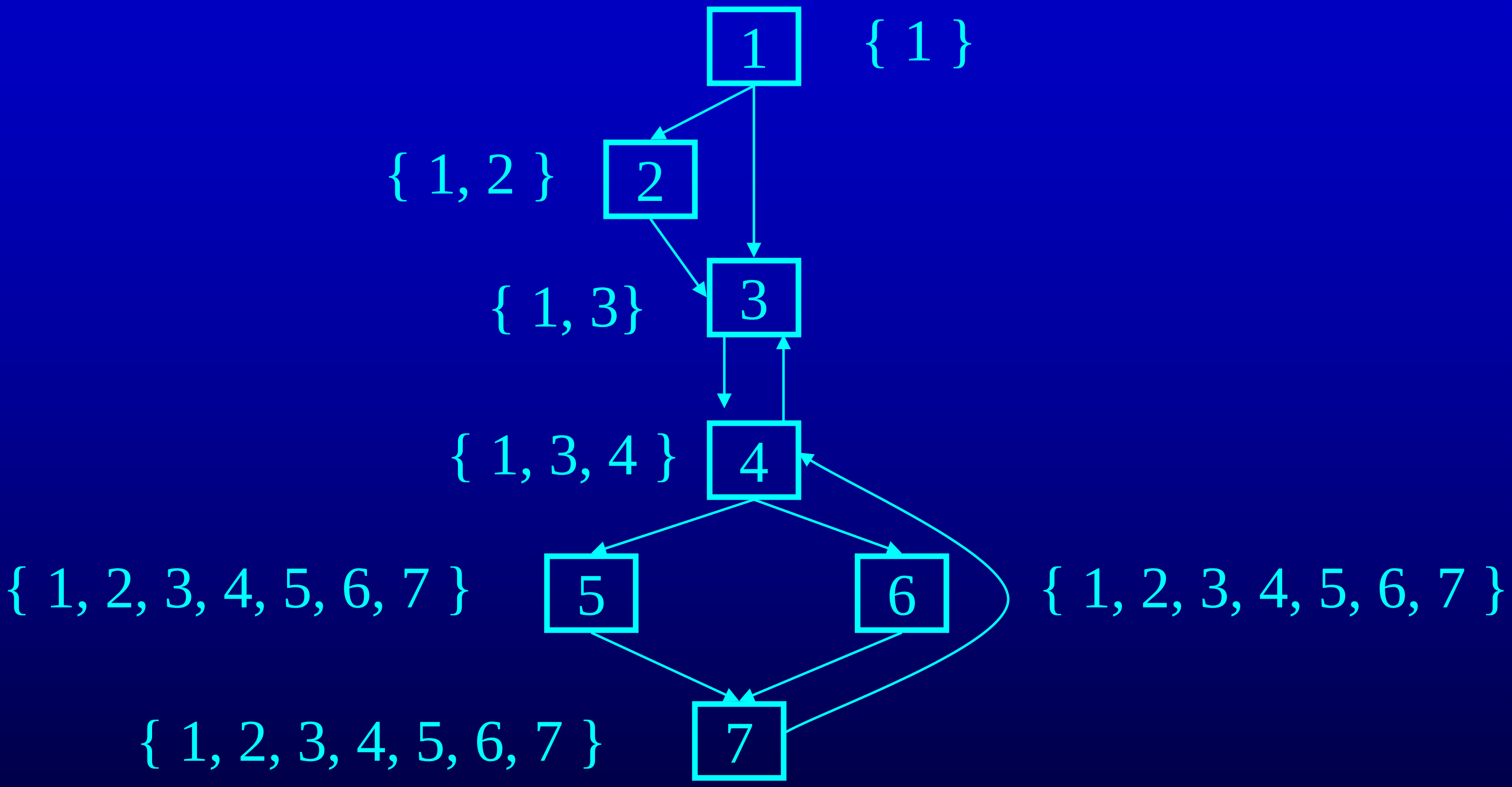
Dominator Computation



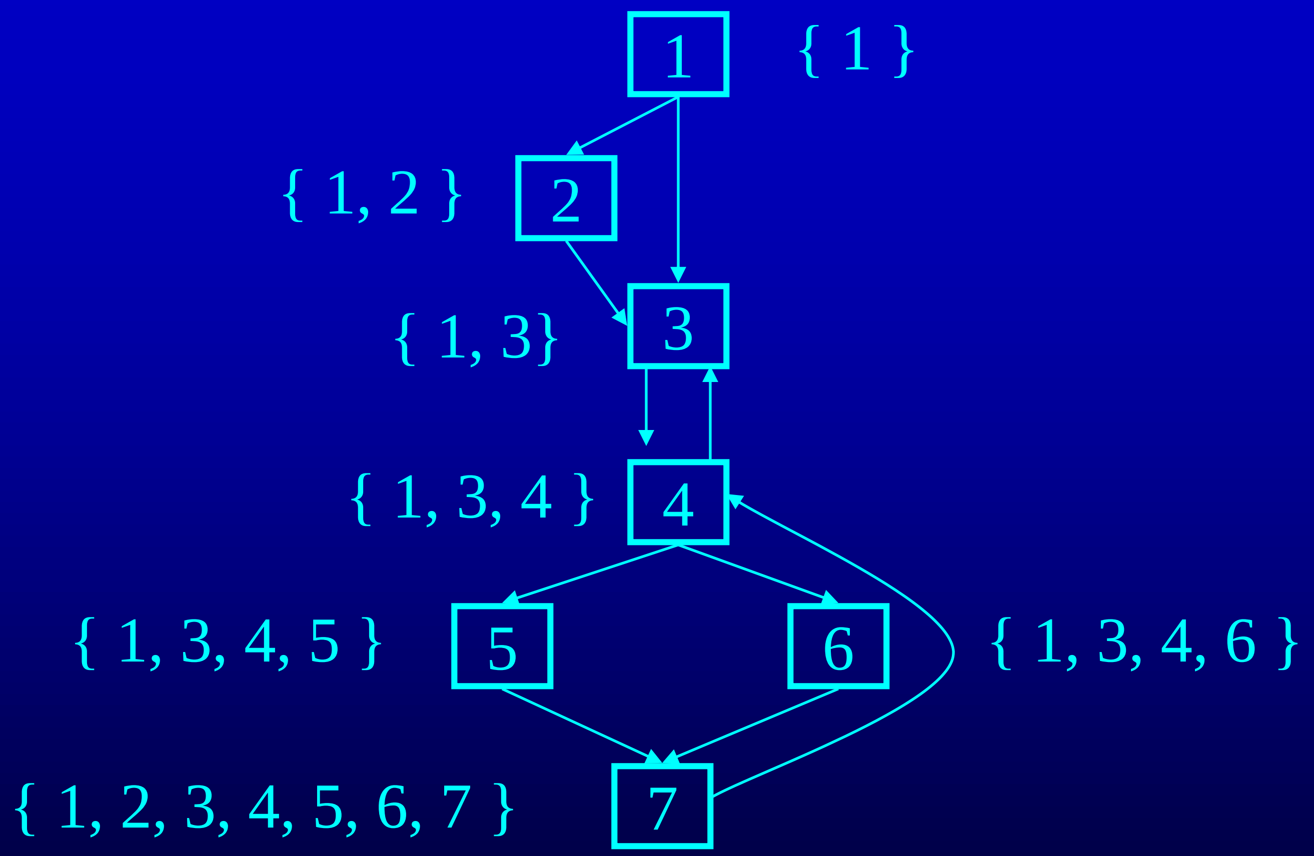
Dominator Computation



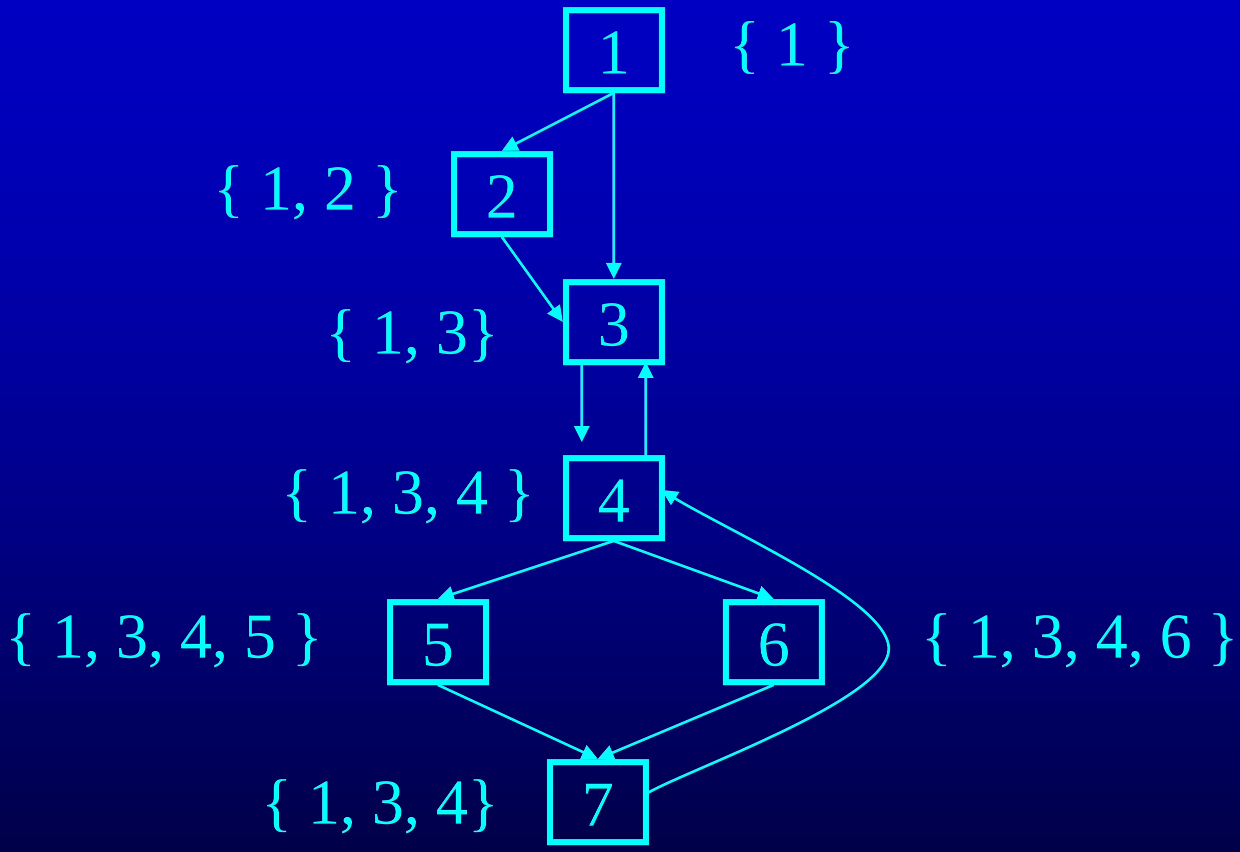
Dominator Computation



Dominator Computation



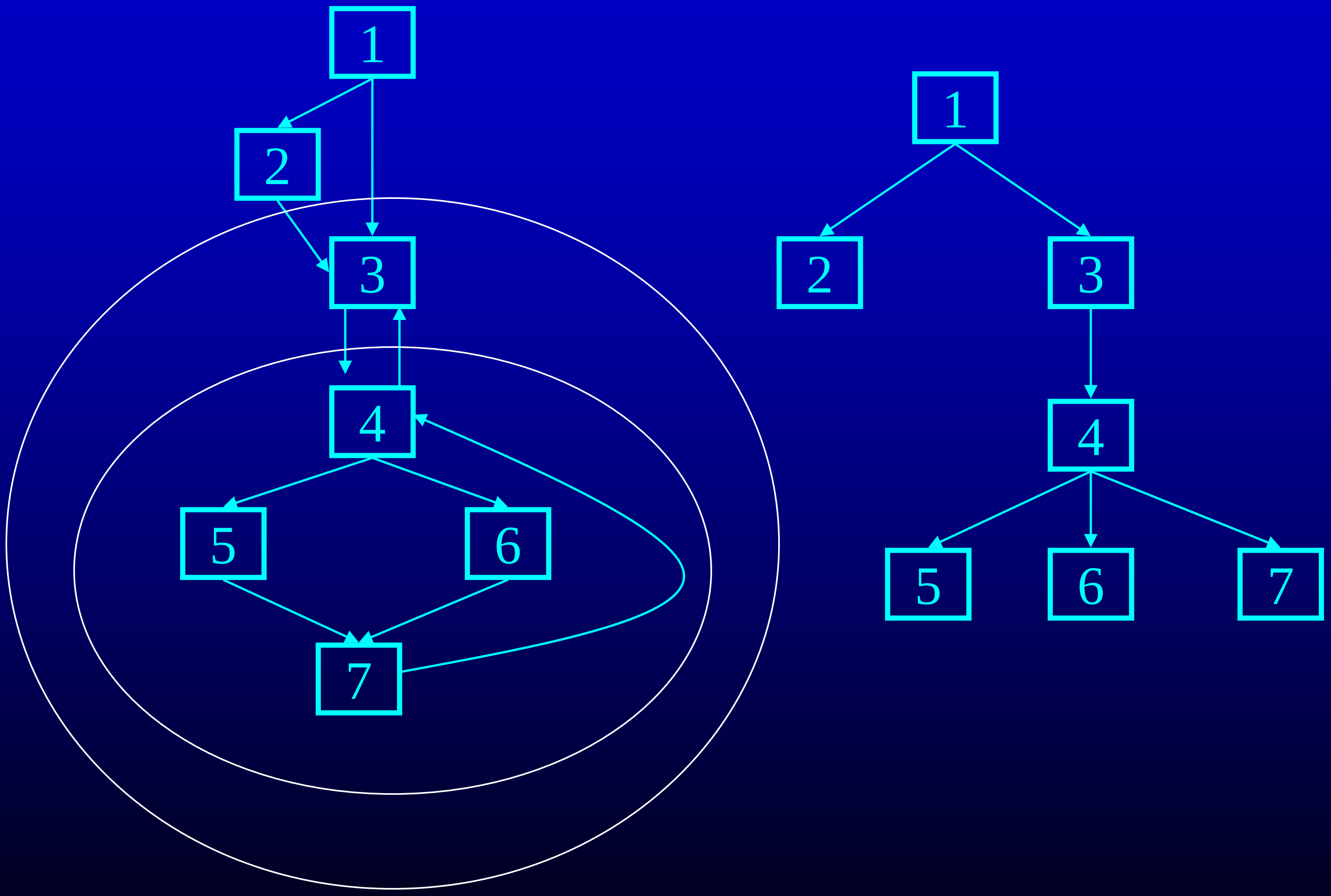
Dominator Computation



Defining Loops

- Unique entry point – header
- At least one path back to header
- Find edges whose heads dominate tails
  - These edges are back edges of loops
  - Given back edge n→d
  - Loop consists of n, d plus all nodes that can reach n without going through d
    (all nodes "between" d and n)
  - d is loop header

Two Loops In Example



Loop Construction Algorithm

insert(m)

    if m $\notin$ loop then

        loop = loop $\cup$ {m}

        push m onto stack

loop(d,n)

    loop = { d }; stack = $\emptyset$; insert(n);

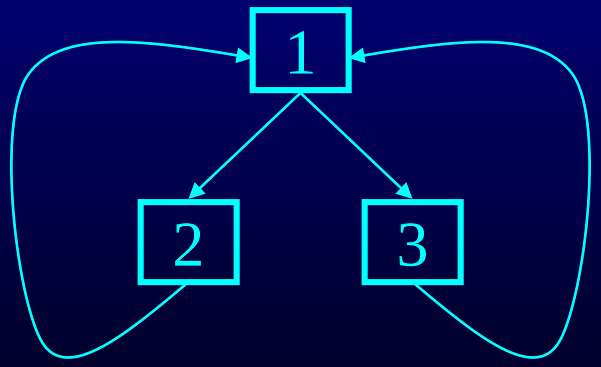
    while stack not empty do

        m = pop stack;

        for all p $\in$ pred(m) do insert(p)

Nested Loops

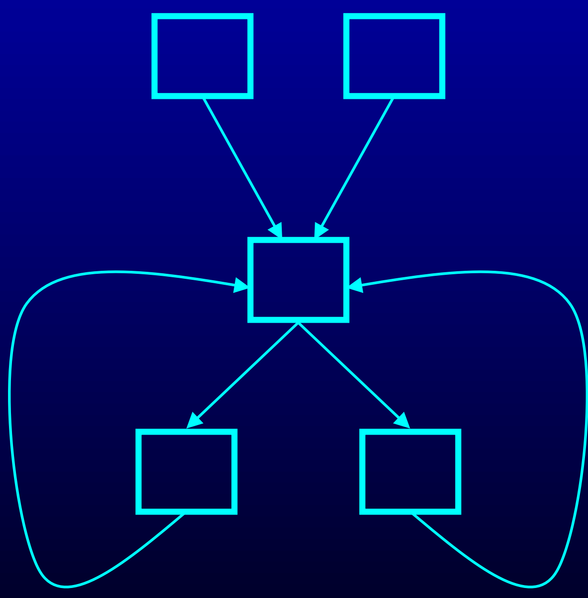
- If two loops do not have same header then
  - Either one loop (inner loop) contained in other (outer loop)
  - Or two loops are disjoint
- If two loops have same header, typically unioned and treated as one loop

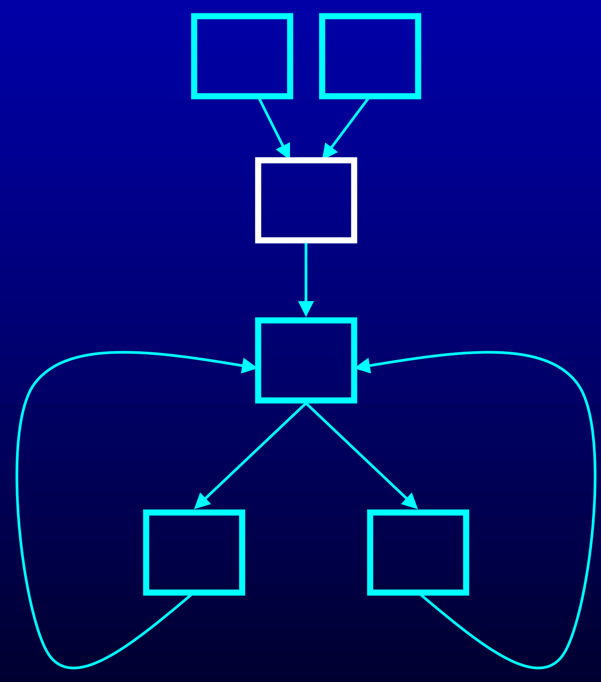


Two loops:
{1,2} and {1, 3}
Unioned: {1,2,3}

Loop Preheader

- Many optimizations stick code before loop
- Put a special node (loop preheader) before loop to hold this code





Loop Optimizations

- Now that we have the loop, can optimize it!
- Loop invariant code motion
  - Stick loop invariant code in the header

Detecting Loop Invariant Code

- A statement is invariant if operands are
  - Constant,
  - Have all reaching definitions outside loop, or
  - Have exactly one reaching definition, and that definition comes from an invariant statement
- Concept of exit node of loop
  - node with successors outside loop

Loop Invariant Code Detection Algorithm

for all statements in loop

    if operands are constant or have all reaching definitions outside loop, mark statement as invariant

do

    for all statements in loop not already marked invariant

        if operands are constant, have all reaching definitions outside loop, or have exactly one reaching definition from invariant statement then

            mark statement as invariant

until find no more invariant statements

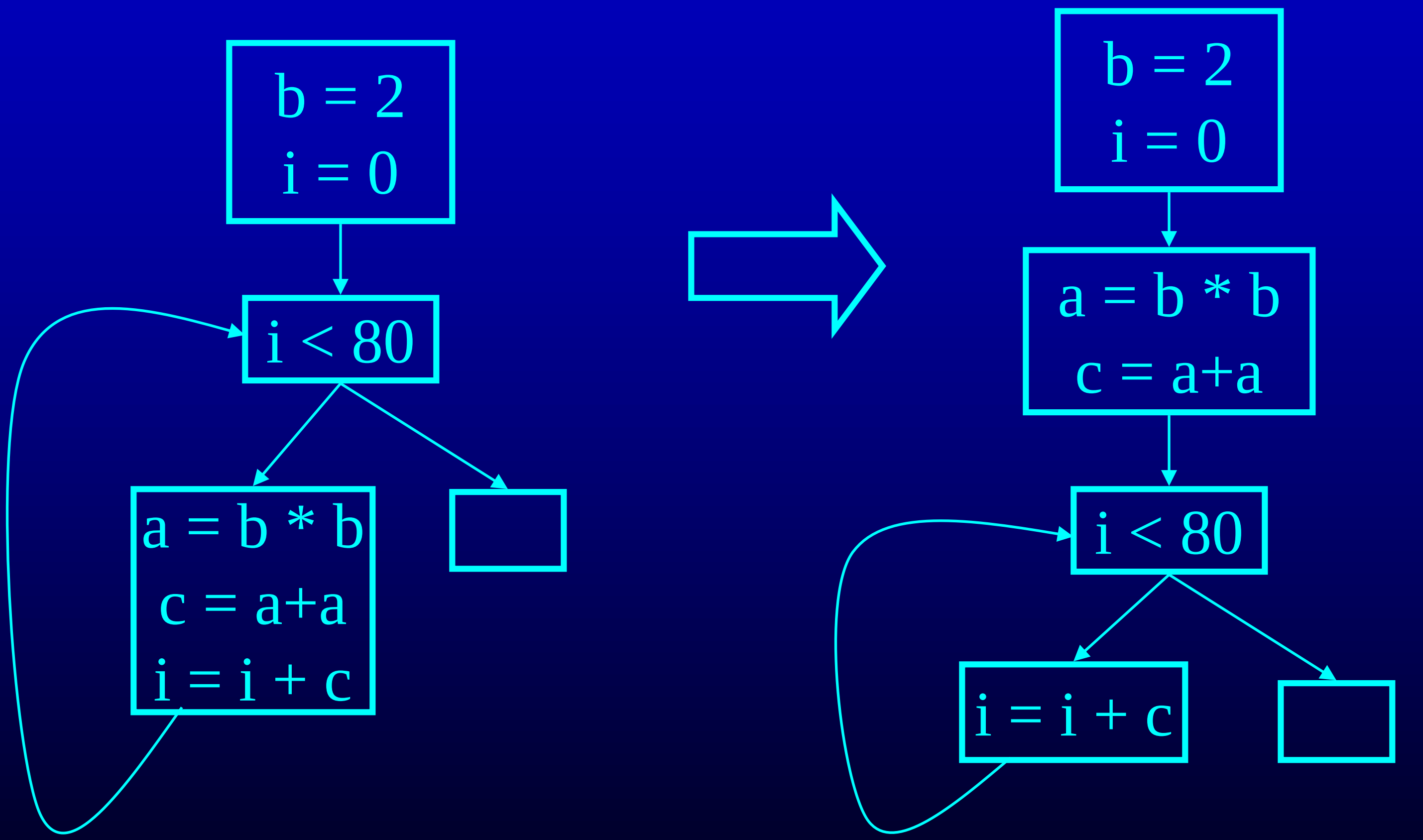
Loop Invariant Code Motion

- Conditions for moving a statement s:x:=y+z into loop header:
  - s dominates all exit nodes of loop
    - If it doesn't, some use after loop might get wrong value
    - Alternate condition: definition of x from s reaches no use outside loop (but moving s may increase run time)
  - No other statement in loop assigns to x
    - If one does, assignments might get reordered
  - No use of x in loop is reached by definition other than s
    - If one is, movement may change value read by use

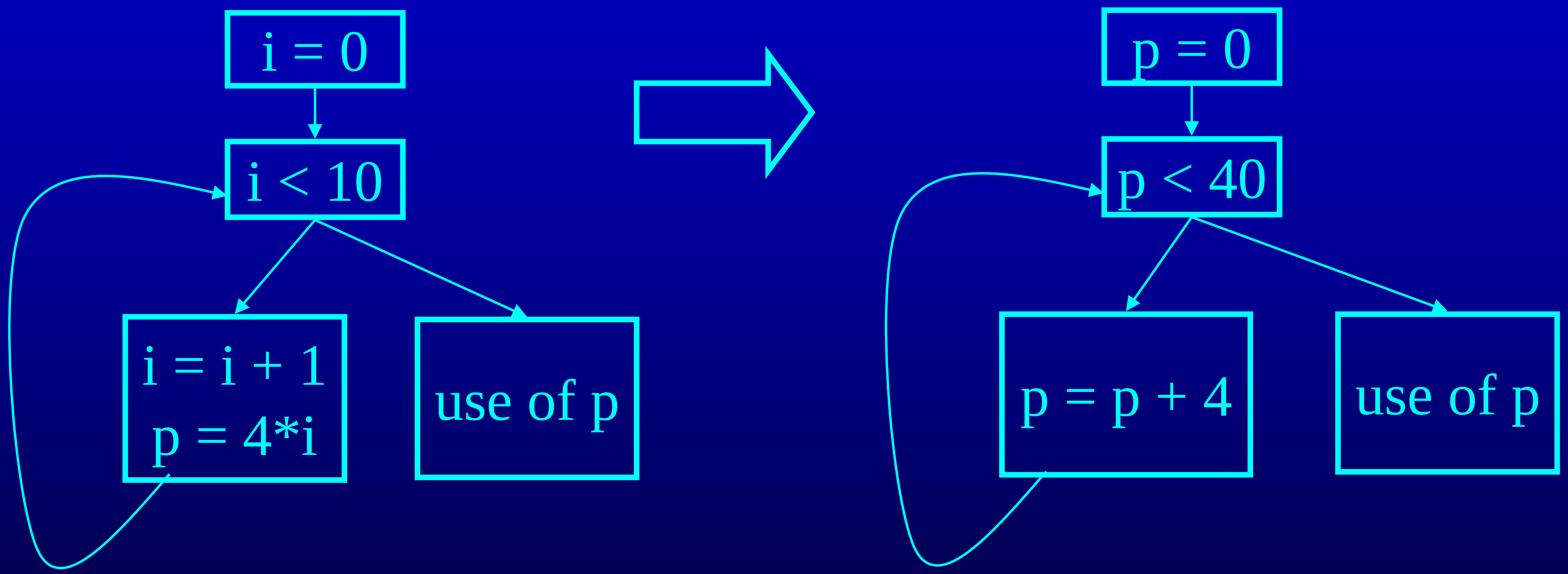
Order of Statements in Preheader

Preserve data dependences from original program

(can use order in which discovered by algorithm)



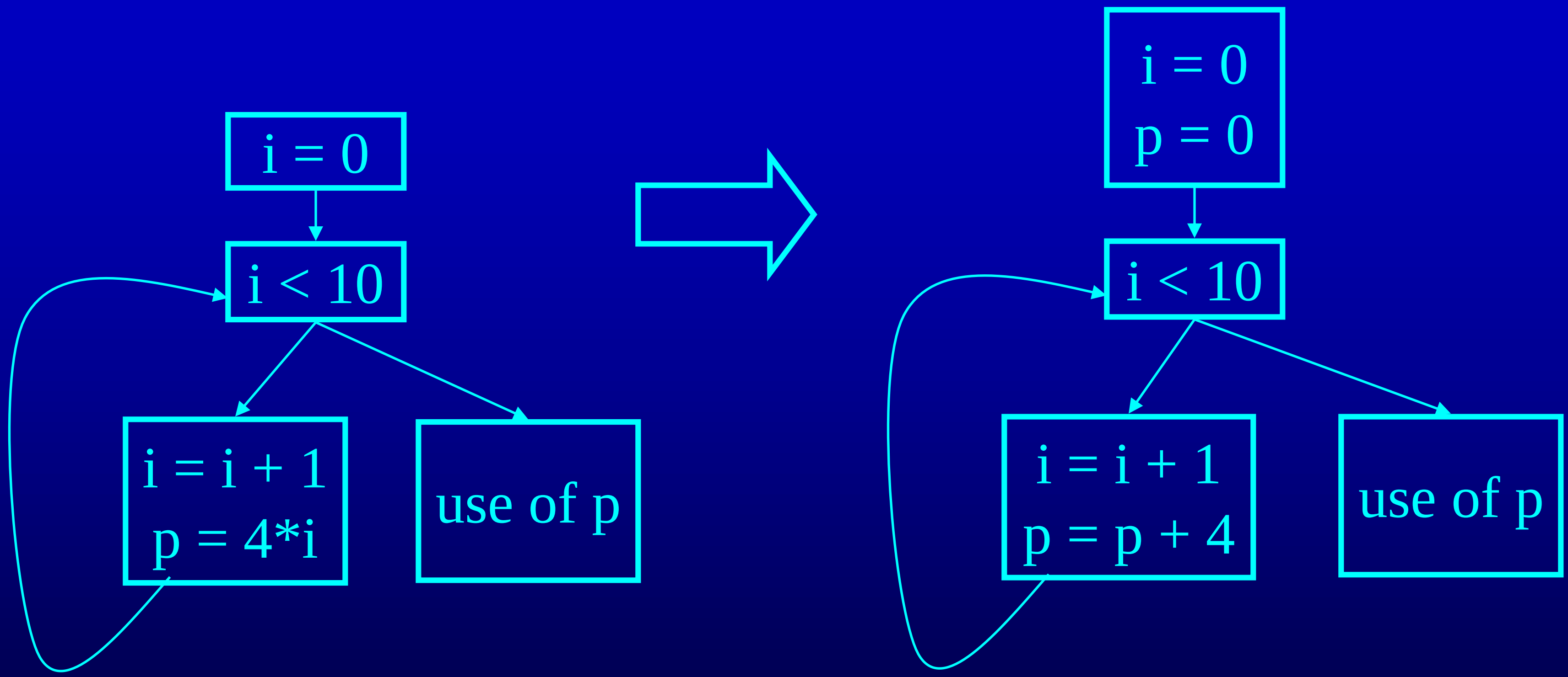
Induction Variable Elimination



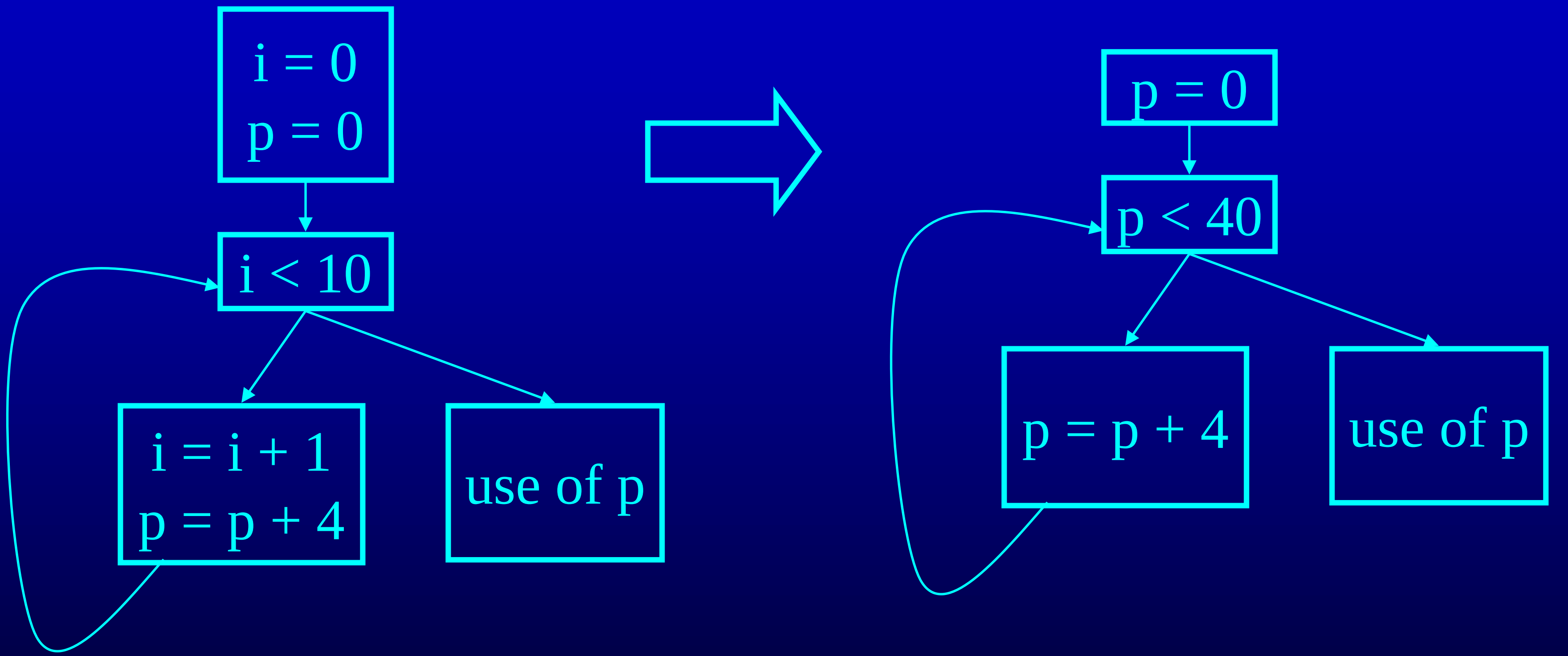
What is an Induction Variable?

- Base induction variable
  - Only assignments in loop are of form $i = i \pm c$
- Derived induction variables
  - Value is a linear function of a base induction variable
  - Within loop, $j = c*i + d$, where i is a base induction variable
  - Very common in array index expressions – an access to a[i] produces code like $p = a + 4*i$

Strength Reduction for Derived Induction Variables



Elimination of Superfluous Induction Variables



Three Algorithms

- Detection of induction variables
  - Find base induction variables
  - Each base induction variable has a family of derived induction variables, each of which is a linear function of base induction variable
- Strength reduction for derived induction variables
- Elimination of superfluous induction variables

Output of Induction Variable Detection Algorithm

- Set of induction variables
  - base induction variables
  - derived induction variables
- For each induction variable j, a triple <i,c,d>
  - i is a base induction variable
  - value of j is i*c+d
  - j belongs to family of i

Induction Variable Detection Algorithm

Scan loop to find all base induction variables

do

- Scan loop to find all variables k with one assignment of form k = j*b where j is an induction variable with triple <i,c,d>
  - make k an induction variable with triple <i,c*b,d*b>
- Scan loop to find all variables k with one assignment of form k = j±b where j is an induction variable with triple <i,c,d>
  - make k an induction variable with triple <i,c,b±d>

until no more induction variables found

Strength Reduction Algorithm

for all derived induction variables j with triple <i,c,d>

Create a new variable s

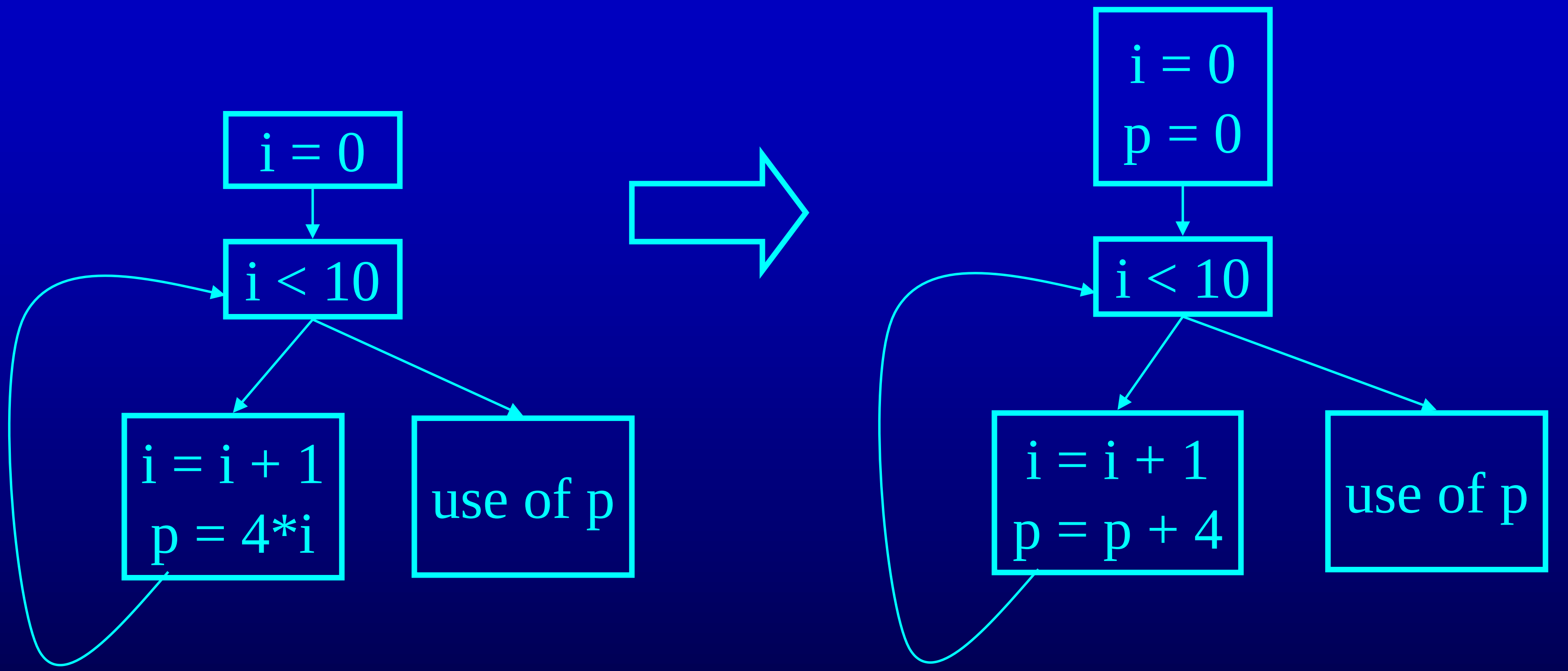
Replace assignment j = i*c+d with j = s

Immediately after each assignment i = i + e, insert statement s = s + c*e (c*e is constant)

place s in family of i with triple <i,c,d>

Insert s = c*i+d into preheader

Strength Reduction for Derived Induction Variables



Induction Variable Elimination

Choose a base induction variable i such that

- only uses of i are in
  - termination condition of the form $i < n$
  - assignment of the form $i = i + m$

Choose a derived induction variable k with <i,c,d>

- Replace termination condition with $k < c*n+d$

Why?

$$k = i*c+d \Rightarrow i < n \Leftrightarrow i*c < c*n \Leftrightarrow i*c+d < c*n+d$$
$$\Leftrightarrow k < c*n+d$$

Induction Variable Wrapup

- There is lots more to induction variables
  - more general classes of induction variables
  - more general transformations involving induction variables

Summary

- Wide range of analyses and optimizations
- Dataflow Analyses and Corresponding Optimizations
  - reaching definitions, constant propagation
  - live variable analysis, dead code elimination
- Induction variable analyses and optimizations
  - Strength reduction
  - Induction variable elimination
  - Important because of time spent in loops