Compiler
(Optimization)

زینب زالی
دانشگاه صنعتی اصفهان

References: Dragon book, Alex Aiken compiler course

Optimization phase

Lexical analyzer
Syntax analyzer
Semantic ånalyzer
Intermediate code generation
Optimization
Code generation

- Most complexity of modern compilers is in optimization (usually the largest phase)
- Optimization on intermediate language:
  - Machine independent
  - Exposes optimization opportunities

Optimization goals

- Improve a program's resource utilization
  - Execution time (most often)
  - Code size
  - Network messages sent
  - Memory, disk, power, etc
- Optimization should not alter what the program computes

Block

- Basic block: Maximal sequence of instructions with
  - No labels: except at the first instruction
  - No jumps: except in the last instruction
- So:
  - Basic block is a single-entry, single-exit, straight-line code segment
  - Can not jump out of a basic block except at end
  - Can not jump into a basic block except at beginning

Block Example

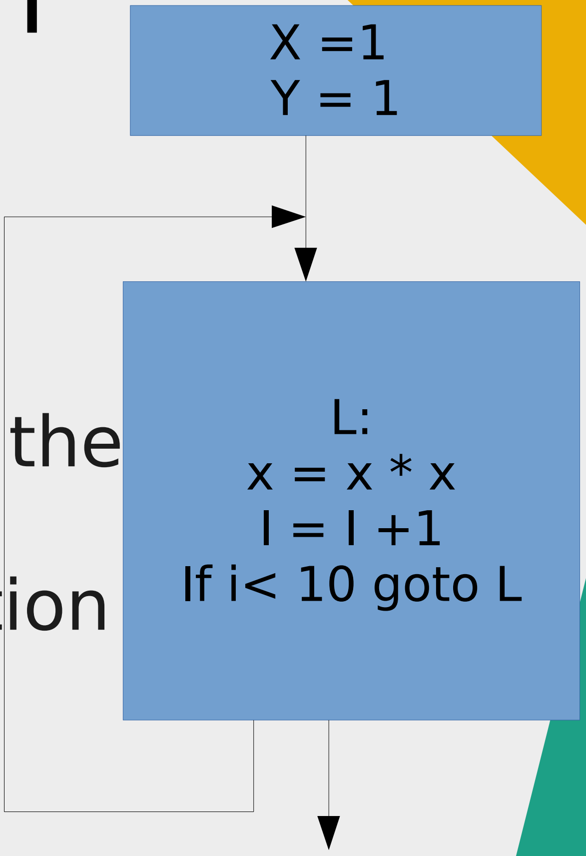
1. L:
2. t = 2 * x
3. w = t + x
4. if w>0 goto L'

W = 3 * x

Can we eliminate 2?

Control Flow Graph

- A directed graph with:
  - Basic blocks as nodes
  - An edge from block A to block B if the execution can pass from the last instruction in A to the first instruction in B
- The body of a method can be represented as a control-flow graph
- There is one initial node
- All return nodes are terminals



Optimization scope

- Local Optimization (most compilers do)
  - Apply to a basic block in isolation
- Global Optimization (many do)
  - Apply to a control flow graph in isolation (method body)
- Inter-procedural optimization (few do)
  - Apply across method boundries

Optimization challenges

- In practice often a conscious decision is made not to implement the fanciest optimization known.
- Why?
  - Some are hard to implement
  - Some are costly in compilation time
  - Some have low payoff
  - Many fancy optimizations are all three
- Maximum benefit for minimum cost

Algebraic simplification

- Some statements can be deleted
  - x = x + 0
  - x = x * 1
- Some statements can be simplified
  - x = x * 0 → x = 0
  - y = y ^ 2 → y = y * y
  - x = x * 8 → x << 3
  - x = x* 15 → t = x << 4; x = t – x

    (if << is faster than * in the target machine)

Constant folding

- Operations on constants can be computed at compile time
  - X = 2 + 3 → x = 5
  - If 2<0 jump L → can be deleted
- Dangerous constant folding
  - In Cross compilers:
    - Ex: 3.8 + 4.5 = 8.3 or 8.29999

Unreachable basic blocks

- Code that is unreachable from the initial block
  - Eg: basic blocks that are not target of any jump or fall through from a conditional
- Eliminating these blocks makes program smaller
  - And sometimes faster
    - Due to memory cache effects
    - Increase spatial locality

Unreachable basic blocks

- #define debug 0

  If (debug)

  .....

- Results of other optimizations

- Each register occurs only once in the left hand side of an assignment
  - x = z + y   b = z + y
  - a = x   →   a = b
  - x = 2 *x   x = 2 * b

Common sub expression elimination

- If
  - basic block is in single assignment form
  - a definition x = is the first use of x in a block
- Then
  - when two assignments have the same rhs they compute the same value
  - X = y +z   x = y + z
  - ....   →
  - W = y + z   w = x

Copy propagation

- If w = x appears in a block replace subsequent uses of w with uses of x
  - x = z + y → b = z + y
  - a = x → ~~a = b~~
  - x = 2 *x → x = 2 * b
- It is useful for enabling other optimizations
  - Constant folding
  - Dead code elimination

Example

- a = 5
- x = 2 * a
- y = x + 6
- t = x * y

→

a= 5
x = 10
y = 16
t = x << 4

Dead code elimination

- If
  - w =rhs appears in a basic block and w does not appear anywhere else in the program
- Then
  - statement w = rhs is dead and can be eliminated
- Dead = does not contribute to the program's result
  - x = z + y   b = z + y
  - a = x   →   ~~a = b~~
  - x = 2 *x   x = 2 * b

Consecutive optimization

- Each local optimization does little by itself
  - But performing one optimization enables another
- Optimizing compilers repeat optimizations until no improvement is possible
  - Or stopped at any point to limit compilation time.

Example

- Initial code:
  - a = x ^ 2 → x* x
  - b = 3
  - c = x
  - d = c * c
  - e = b *2
  - f = a + d
  - g = e*f

Example

- Initial code:
  - a = x ^ 2 → x* x
  - ~~b = 3~~
  - ~~c = x~~
  - ~~d = c * c → x * x → a~~
  - ~~e = b *2 → b <<1 → 3 << 1 → 6~~
  - f = a + d → f = a + a → 2 * a
  - g = e*f → g = 6 * f → 12*a

Peephole optimization

- Optimization applied directly to assembly code
- a sliding window of target instructions (called the peephole) and replacing instruction sequences within the peephole by a shorter or faster sequence, whenever possible.

Peephole optimization

- Ex1
  - Move $a, $b
  - Move $$b, $a
  - If the second line is not the target of a jump

Move $a, $b

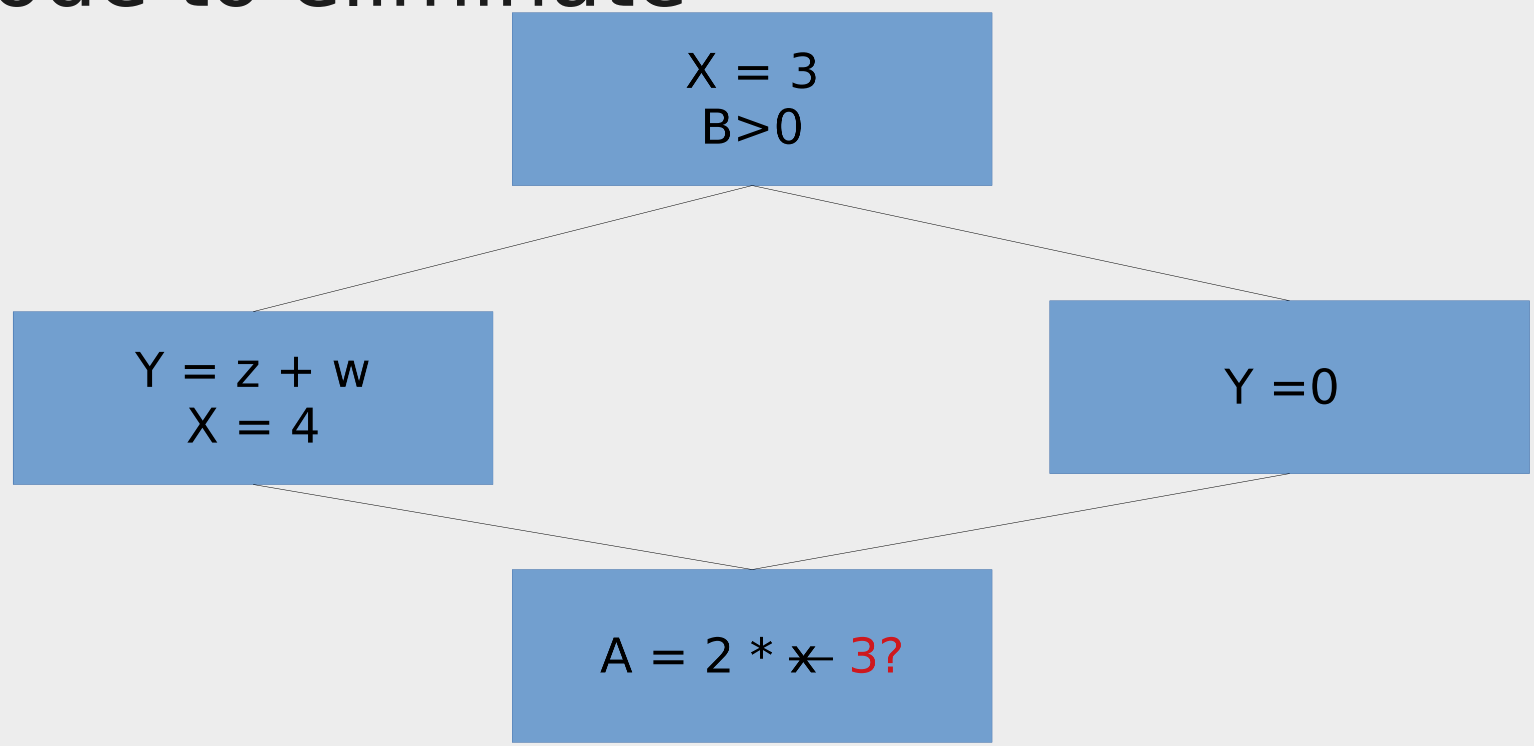
- Ex2
  - Add $a, $a i
  - Add $a, $a j

Add $a, $a i+j

- Ex3
  - Addiu $a, $b 0 → move $a, $b
  - Addiu $a $a 0 → eliminated
  - Move $a $a → eliminated

Dataflow analysis

- How do we know is it OK to globally propagate constants? Or detect dead code to eliminate



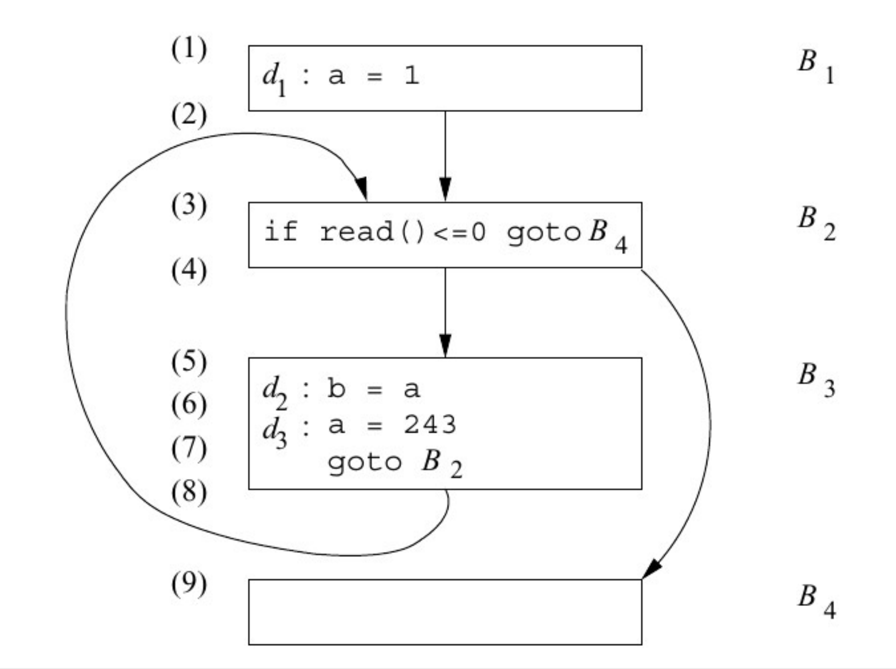
Dataflow analysis

- Checking the condition requires global dataflow analysis
  - An analysis of entire control-flow graph

Dataflow analysis

- In general, it is not possible to keep track of all the program states for all possible paths.
- In data-flow analysis, we do not distinguish among the paths taken to reach a program point.
- Moreover, we do not keep track of entire states;
- rather, we abstract out certain details, keeping only the data we need for the purpose of the analysis.

Data-flow analysis

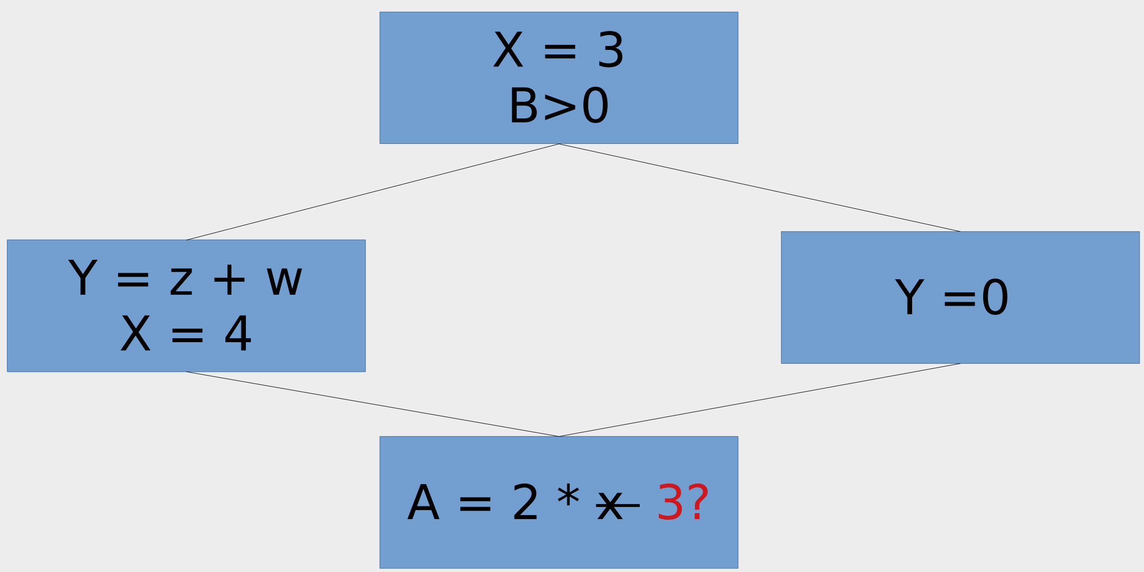


Constant propagation

- To replace a use of x by a constant k we must know:
  - On every path to the use of x, the last assignment to x is x = k

Constant propagation

- To replace a use of x by a constant k we must know:
  - On every path to the use of x, the last assignment to x is x= k



Global optimization

- The optimization depends on knowing a property X at a particular point in program execution
- Proving x at any point requires knowledge of the entire program
- Global dataflow analysis is a standard technique for solving problems with these characteristics
- Global constant propagation is one example of an optimization that requires global datafloa analysis

Constant propagation

- The analysis of a complicated program can be expressed as a combination of simple rules relating the change in information between adjacent statements

Constant propagation

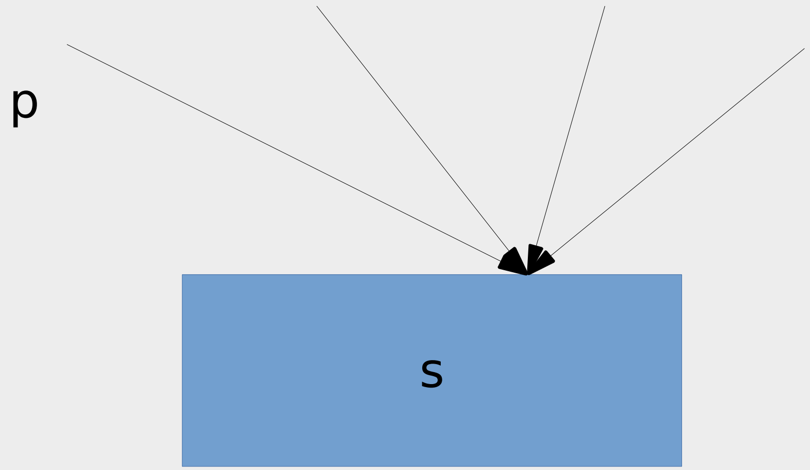
- To make the program precise, we associate one of the following values with x at every program point

Value	interpretation
⊥	This statement never executes
C	x = constant c
T	x is not a constant

Constant propagation

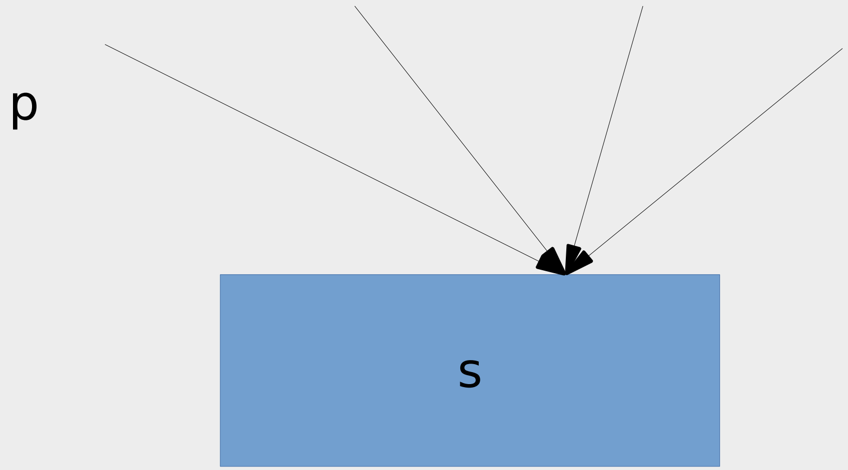
- For each statement s, we compute information about the value of x immediately before and after s
  - C(x, s, in) = value of x before s
  - C(x, s, out) = value of x after s
- In the following rules, let statement s have immediate predecessor statements $p_1, \ldots, p_n$

Rule 1



If C(P_i, x, out)= T, for any i, then C(s, x, in) = T

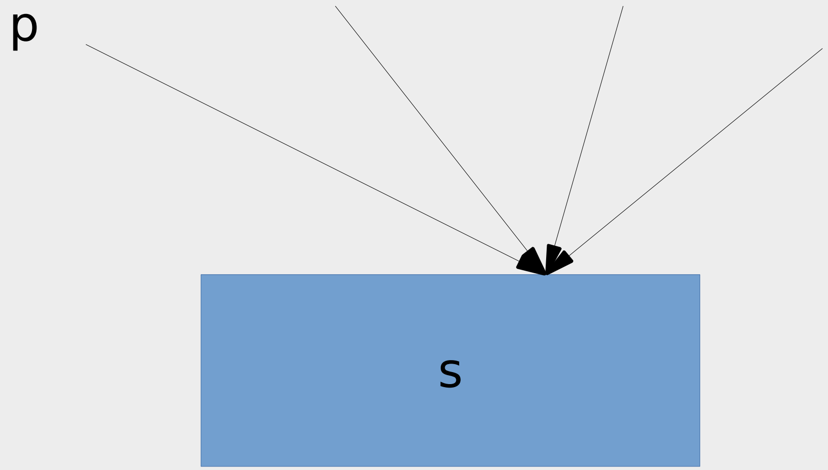
Rule 2



If $C(P_i, x, out) = c \;\&\; c(P_j, x, out) = d \;\&\; d <> c$

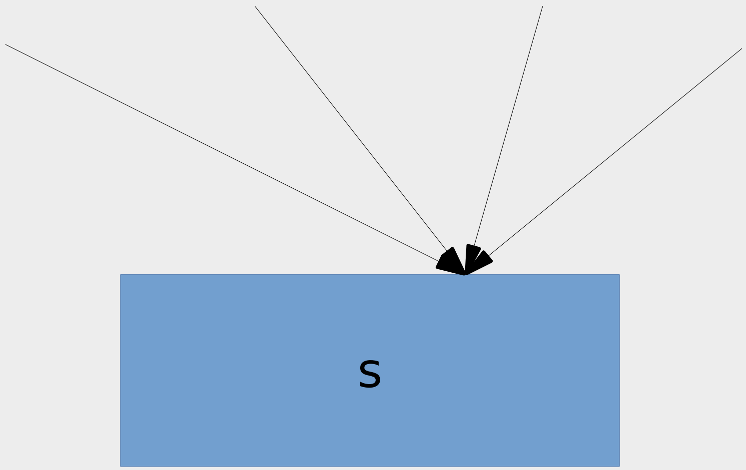
then $C(s, x, in) = T$

Rule 3



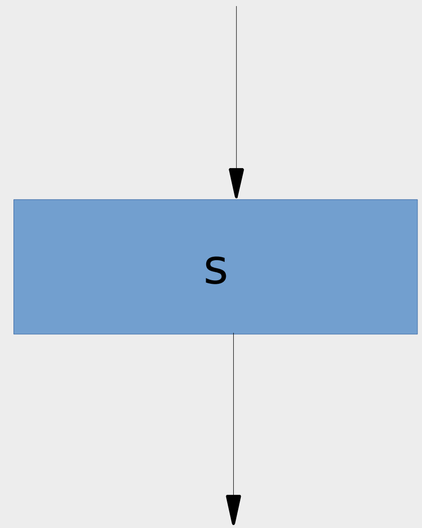
- If $C(P_i, x, out) = c$ or $\perp$ for all I, then $C(s, x, in)=c$

Rule 4



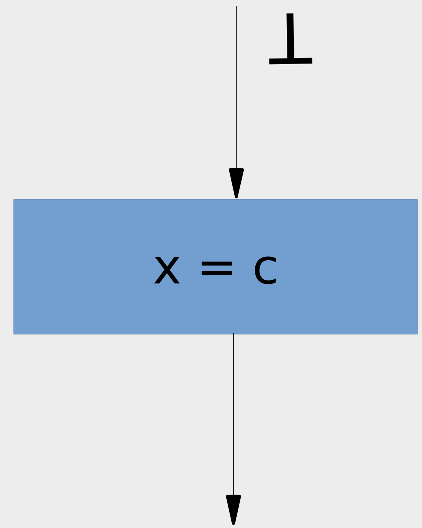
If $C(P_i, x, out) = \perp$ for all i, then $C(s, x, in) = \perp$

Rule 5



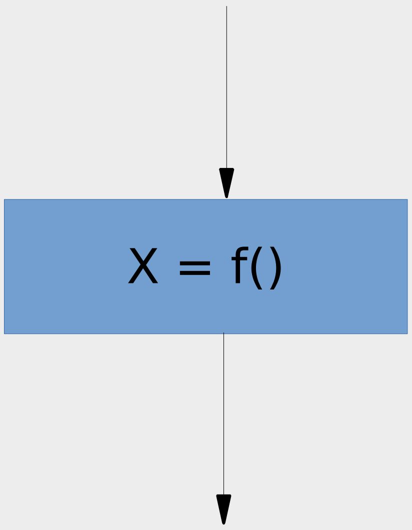
$$C(s, x, out) = \perp, \text{ if } C(s, x, in) = \perp$$

Rule 6



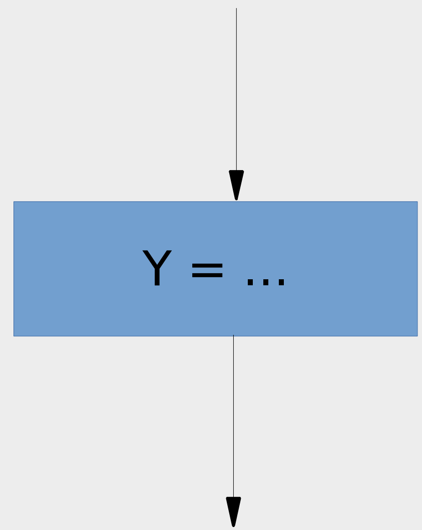
$C(x = c, x, out) = c$ if c is a constant

Rule 7



C(x = f(...), x, out)= T

Rule 8

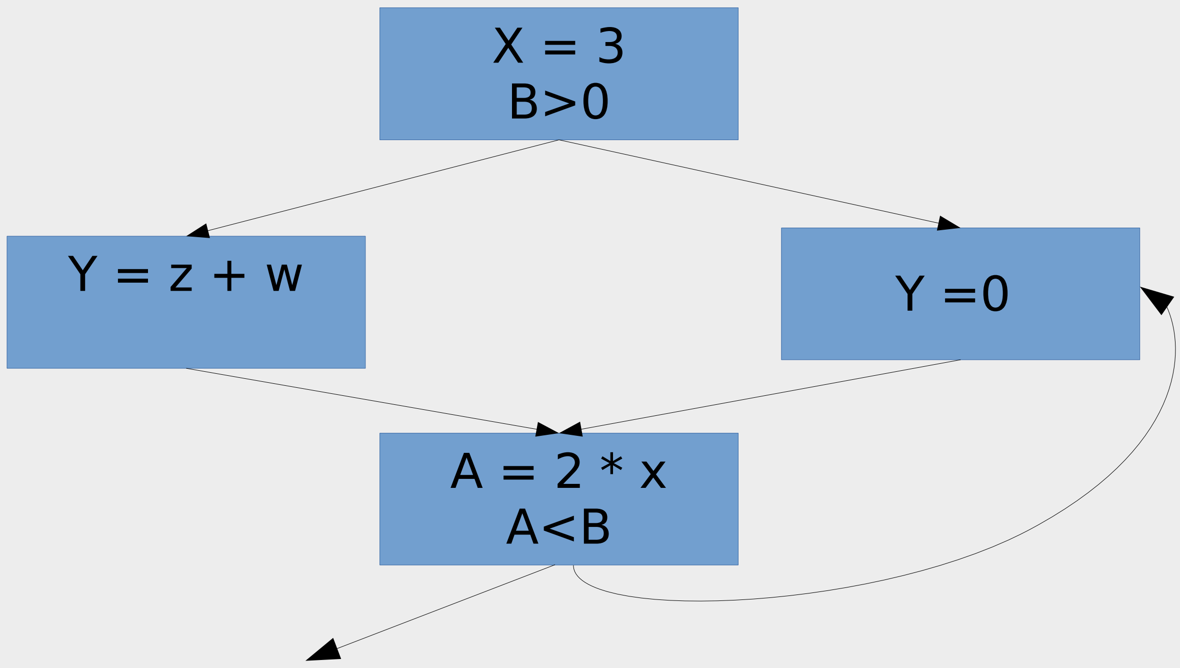


C(y= ..., x, out) = C(y=...,x,in) if x<>y

Rule 8

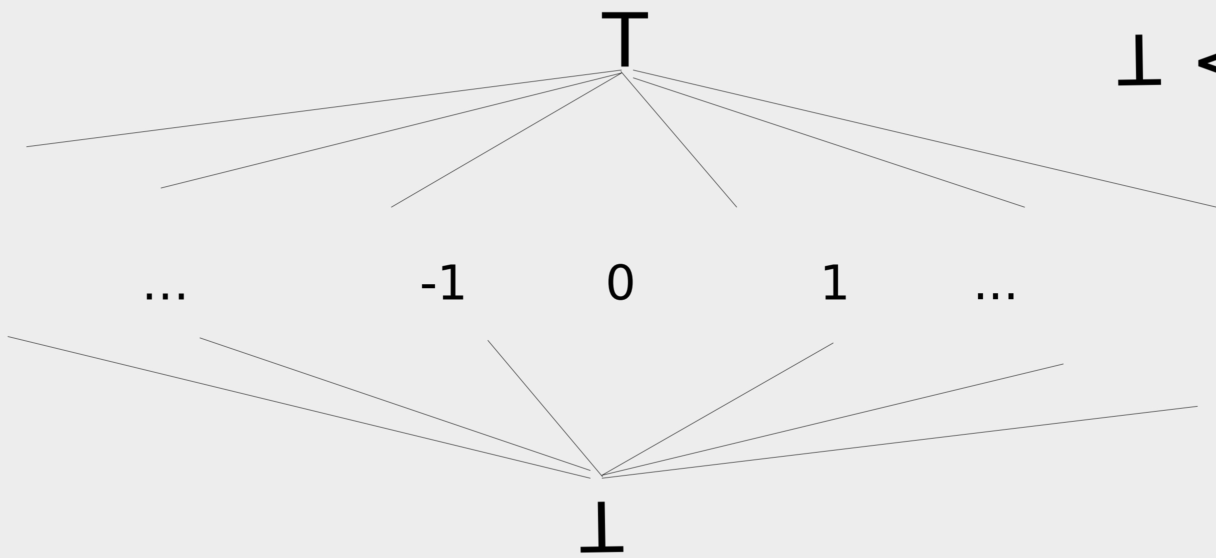
- For every entry s to the program, set c(s, x, in) = T
- Set C(s, x, in) = C(s, x, out) = $\perp$ everywhere else
- Repeat until all points satisfy 1-8:
  - Pick s not satisfying 1-8 and update using the appropriate rule

Why we need ⊥ ?



ordering

- T is the greatest value, ⊥ is the least
  - All constants are in between and incomparable
- Let lub be the least-upper bound in this ordering
- Rules 1-4 can be written using lub:
  - C(s,x,in)=lub{C(p,x,out) | p is a predecessor of s }



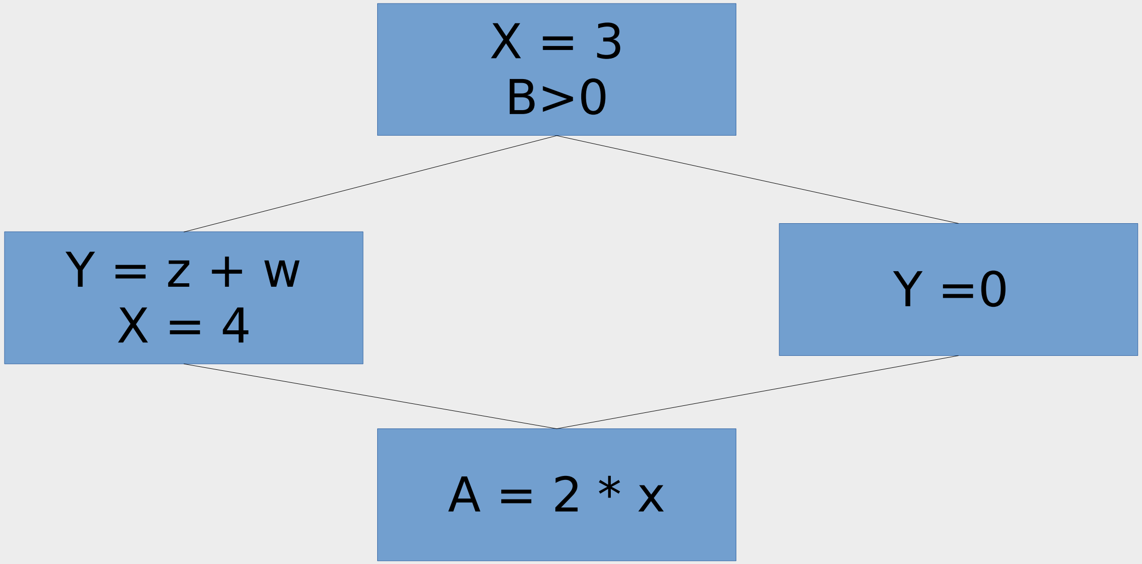
$\perp < c < T$

ordering

- Simply saying “repeat until nothing changess” doesn’t guarantee that eventually nothing changes
- The use of lub explains why the algorithm terminates
  - Values start as $\perp$ and only increase
  - $\perp$ Can change to a constant, and a constant to T
  - Thus, C(s, x, _) can change at most twice
- Thus the constant propagation algorithm is linear in program size

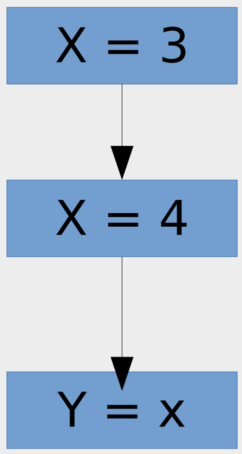
Dead code elimination

- Once constants have been globally propagated, we would like to eliminate dead code



Dead code elimination

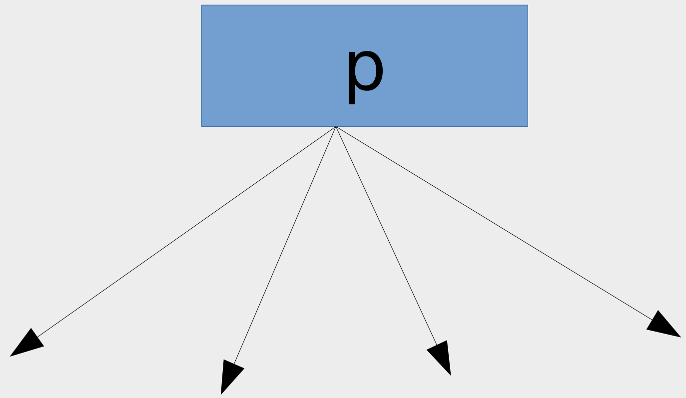
- The first value of x is dead (never used)
- The second value of x is live (may be used)
- Liveness is an important concept
- A variable x is live at statement s if
  - There exists a statement s' that uses x
  - There is a path from s to s'
  - That path has no intervening assignment to x



Dead code elimination

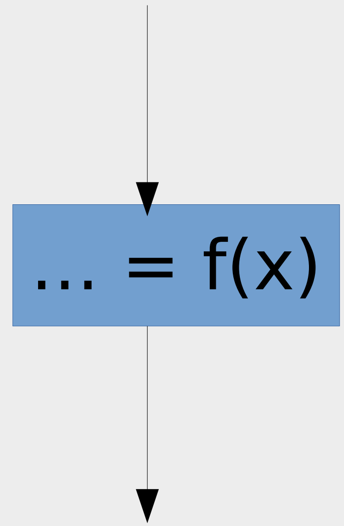
- We can express liveness in terms of information transferred between adjacent statements, just as in copy propagation
- Liveness is simpler that constant propagation it is a boolean property

Rule 1



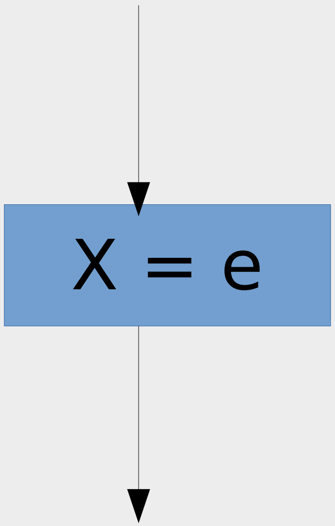
$$L(p, x, out) = \vee \{ L(s,x, in) \mid s \text{ a successor of } p\}$$

Rule 2



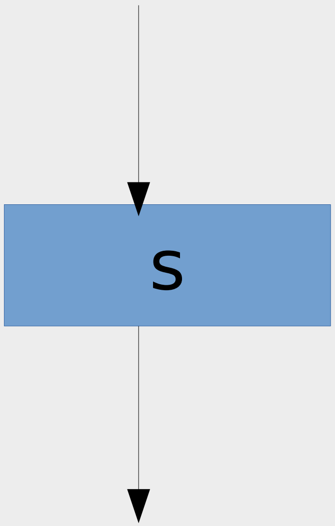
- L(s, x, in) = true if s refers to x on the rhs

Rule 3



- L(x = e, x, in) = false if e does not refer to x

Rule 4

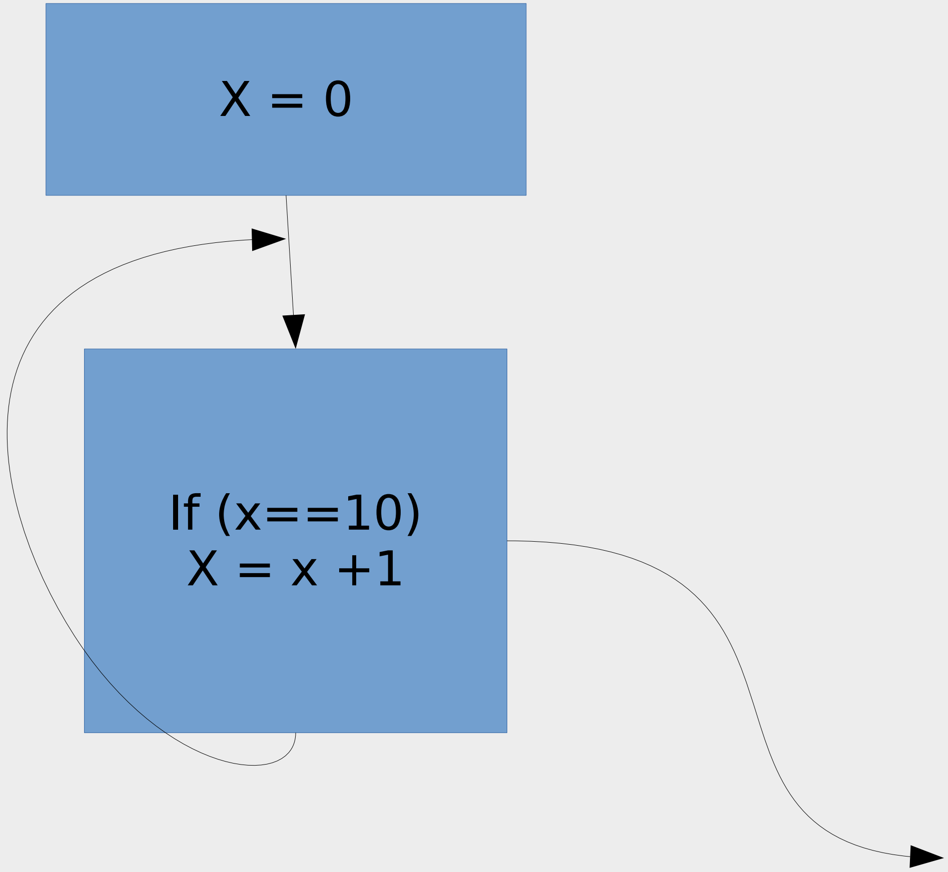


- L(s , x, in) = L(s , x, out) if s does not refer to x

Dead code elimination

- Let all L(...) = false initially
- Repeat until all statements s satisfy rules 1-4
  - Pick s where one of 1-4 does not hold and update using the appropriate rule

Dead code elimination



- A value can change from false to true, but not the other way around
- Each value can change only once, so termination is guaranteed
- Once the analysis is computed, it is simple to eliminate dead code