PUNE VIDYARTHI GRIHA's

COLLEGE OF ENGINEERING, NASHIK.

" CODE OPTIMIZATION "

PREPARED BY :

PROF. ANAND N. GHARU

ASSISTANT PROFESSOR

COMPUTER DEPARTMENT

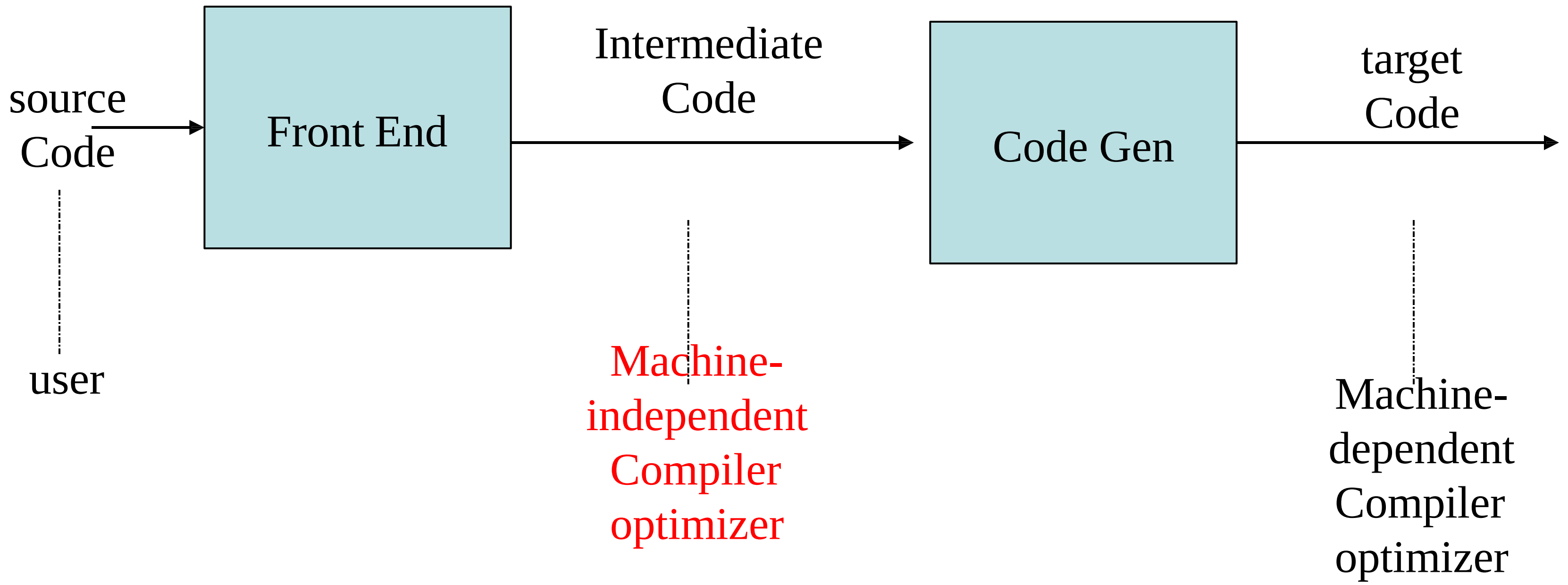
3/18/2019 SUBJECT – COMPILER (BE COMPUTER SPPU-2019)

CONTENTS :

Need for Optimization, local, global and loop optimization, Optimizing transformations, compile time evaluation, common sub-expression elimination, variable propagation, code movement, strength reduction, dead code elimination, DAG based local optimization, Introduction to global data flow analysis, Data flow equations and iterative data flow analysis.

Code Optimization :

- A main goal is to achieve a better performance


source Code
Front End
Intermediate Code
Code Gen
target Code
user
Machine-independent Compiler optimizer
Machine-dependent Compiler optimizer

What is optimization?

- In computing, **optimization** is the process of modifying a system to make some aspect of it work more efficiently or use fewer resources. For instance, a computer program may be optimized so that it executes more rapidly, or is capable of operating with less memory storage or other resources, or draw less power. The system may be a single computer program, a collection of computers or even an entire network such as the internet.

TYPES OF OPTIMIZATION

Optimization is of two types

1. Machine Dependent optimization
2. Machine independent

- The machine dependent optimization is based on characteristics of the target machine (the instruction set used and addressing modes used for the instructions) to generate efficient target code.

- The machine independent optimization is based on the characteristics of the programming languages for appropriate programming structure and usage of efficient arithmetic properties in order to reduce execution time

Code Optimization Techniques

Constant propagation

If the value of a variable is a constant, then replace the variable by the constant

It is not the constant definition, but a variable is assigned to a constant

The variable may not always be a constant

E.g.

```
N := 10;  C := 2;
for (i:=0; i<N; i++) { s := s + i*C; }
    ⇒ for (i:=0; i<10; i++) { s := s + i*2; }
If (C) go to … ⇒ go to …
```

The other branch, if any, can be eliminated by other optimizations

Requirement:

After a constant assignment to the variable

Code Optimization Techniques

Constant folding

In a statement x := y op z or x := op y

If y and z are constants

Then the value can be computed at compilation time

Example

#define M 10

x := 2 * M $\Rightarrow$ x := 20

If (M < 0) goto L $\Rightarrow$ can be eliminated

y := 10 * 5 $\Rightarrow$ y := 50

Difference: constant propagation and folding

Propagation: only substitute a variable by its assigned constant

Folding: Consider variables whose values can be computed at compilation time

and controls whose decision can be determined at compilation time

Code Optimization Techniques

Algebraic simplification

More general form of constant folding, e.g.,

$x + 0 \Rightarrow x$ $x - 0 \Rightarrow x$

$x * 1 \Rightarrow xx / 1 \Rightarrow x$

$x * 0 \Rightarrow 0$

Repeatedly apply the rules

$(y * 1 + 0) / 1 \Rightarrow y$

Strength reduction

Replace expensive operations

E.g., $x := x * 8 \Rightarrow x := x << 3$

Code Optimization Techniques

Copy propagation

Extension of constant propagation

After y is assigned to x, use y to replace x till x is assigned again

Example

```
    x := y;          ⇒      s := y * f(y)
    s := x * f(x)
```

Reduce the copying

If y is reassigned in between, then this action cannot be performed

Code Optimization Techniques

Common subexpression elimination

Example:

```
a := b + c                    a := b + c
c := b + c         ⇒          c := a
d := b + c                    d := b + c
```

Example in array index calculations

c[i+1] := a[i+1] + b[i+1]

During address computation, i+1 should be reused

Not visible in high level code, but in intermediate code

Code Optimization Techniques

Unreacheable code elimination

Construct the control flow graph

Unreachable code block will not have an incoming edge

After constant propagation/folding, unreachable branches can be eliminated

Dead code elimination

Ineffective statements

```
    x := y + 1          (immediately redefined, eliminate!)
    y := 5               ⇒          y := 5
    x := 2 * z          x := 2 * z
```

A variable is dead if it is never used after last definition

- Eliminate assignments to dead variables

Need to do data flow analysis to find dead variables

Code Optimization Techniques

Loop optimization Techniques

- Loop invariant detection and code motion
- Induction variable elimination
- Strength reduction in loops
- Loop unrolling
- Loop peeling
- Loop fusion

Code Optimization Techniques

Loop invariant detection and code motion

If the result of a statement or expression does not change within a loop, and it has no external side-effect Computation can be moved to outside of the loop

Example

```
for (i=0; i<n; i++) a[i] := a[i] + x/y;
Three address code
            for (i=0; i<n; i++) { c := x/y; a[i] := a[i] +
c; }
⇒ c := x/y;
    for (i=0; i<n; i++) a[i] := a[i] + c;
```

Code Optimization Techniques

Strength reduction in loops

Example

s := 0; for (i=0; i<n; i++) { v := 4 * i; s := s + v;)

⇒s := 0; for (i=0; i<n; i++) { v := v + 4; s := s + v;)

Induction variable elimination

If there are multiple induction variables in a loop, can eliminate the ones which are used only in the test condition

Example

s := 0; for (i=0; i<n; i++) { s := 4 * i; … } -- i is not referenced in loop

⇒ s := 0; e := 4*n; while (s < e) { s := s + 4; }

Code Optimization Techniques

Loop unrolling

- Execute loop body multiple times at each iteration
- Get rid of the conditional branches, if possible
- Allow optimization to cross multiple iterations of the loop
  - Especially for parallel instruction execution
- Space time tradeoff
  - Increase in code size, reduce some instructions

Loop peeling

- Similar to unrolling
- But unroll the first and/or last few iterations

Code Optimization Techniques

Loop fusion

Example

```
for i=1 to N do
        A[i] = B[i] + 1
endfor
for i=1 to N do
        C[i] = A[i] / 2
endfor
for i=1 to N do
        D[i] = 1 / C[i+1]
endfor
```

Before Loop Fusion

```
for i=1 to N do
    A[i] = B[i] + 1
    C[i] = A[i] / 2
    D[i] = 1 / C[i+1]
endfor
```

Is this correct?
Actually, cannot fuse the third loop

Principle sources of optimization

- Local optimization: within a basic block
- Global optimization: otherwise
- Mixed

PRINCIPLE SOURCES OF OPTIMIZATION

- Function Preserving Transformations :

i) Common Subexpression Elimination

ii)Copy Propogation

iii)Dead – code Elimination

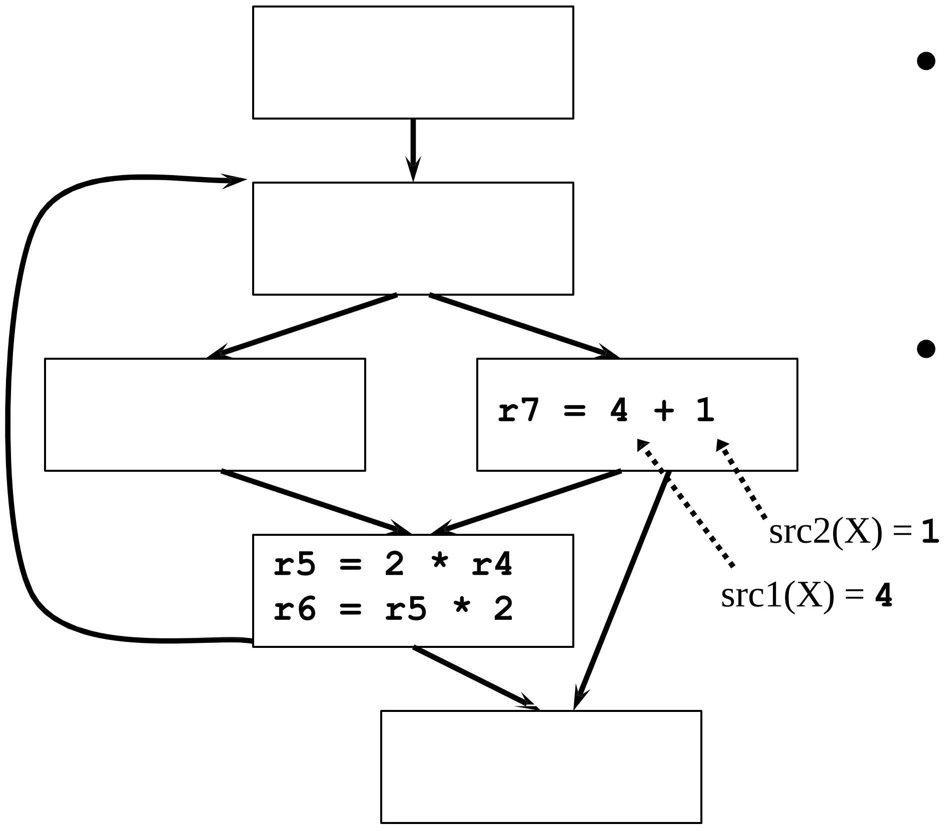
iv)Constant folding

v)Loop Optimization : Code Motion, Strength Reduction, Induction Variable Elimination

Classic Examples of Local and Global Code Optimizations

- Local
  - Constant folding
  - Constant combining
  - Strength reduction
  - Constant propagation
  - Common subexpression elimination
  - Backward copy propagation
- Global
  - Dead code elimination
  - Constant propagation
  - Forward copy propagation
  - Common subexpression elimination
  - Code motion
  - Loop strength reduction
  - Induction variable elimination

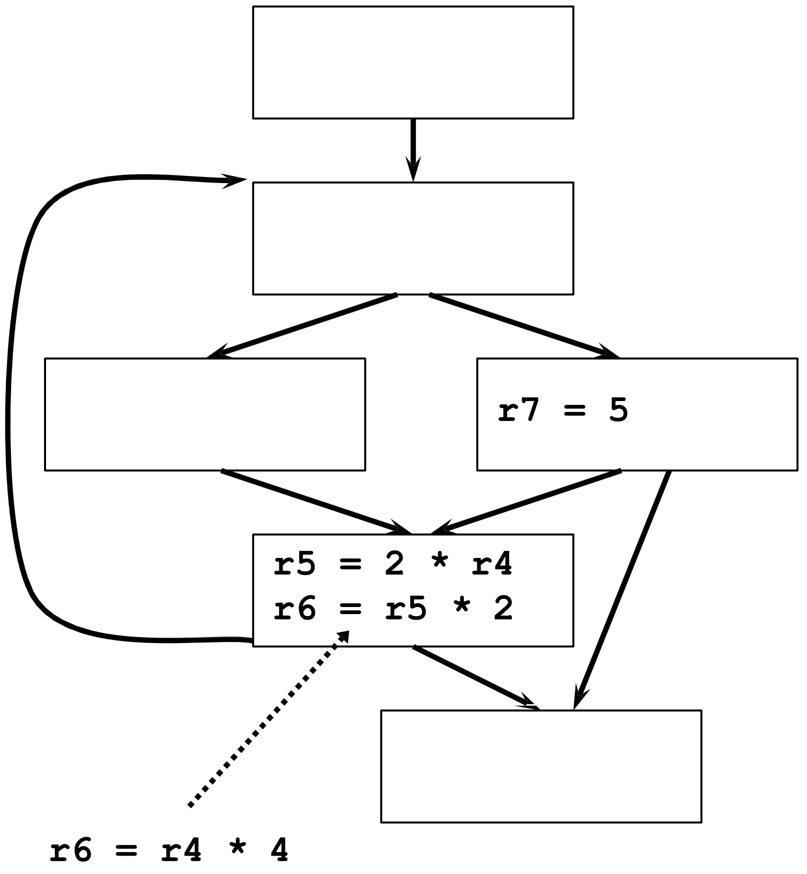
Local: Constant Folding



- Goal: eliminate unnecessary operations
- Rules:

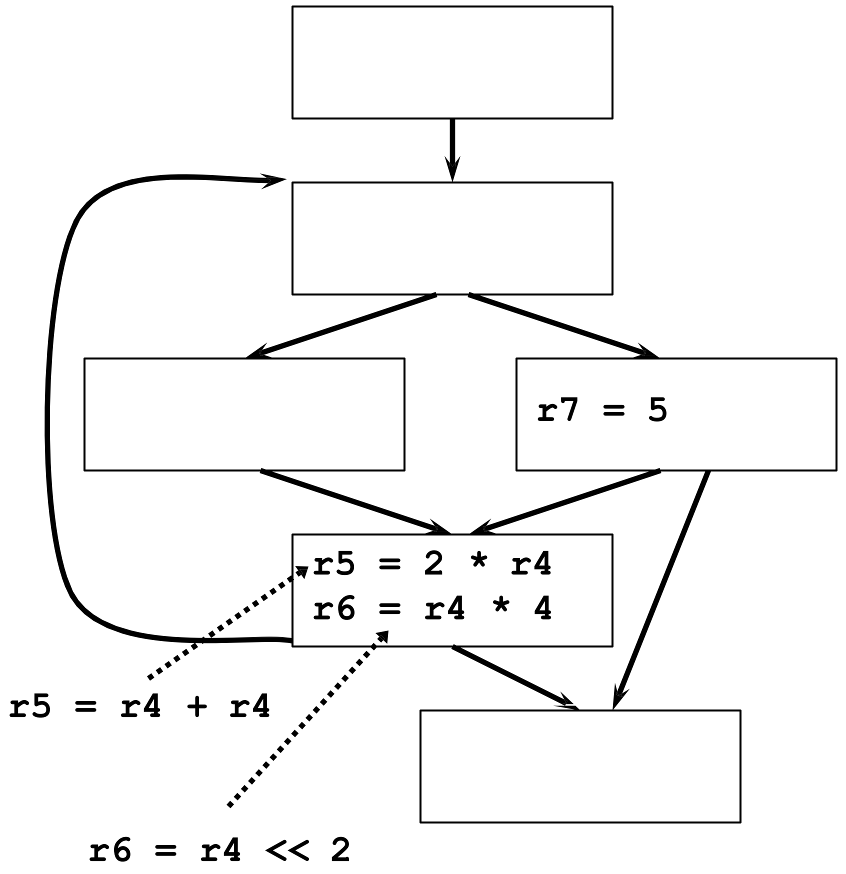
1. X is an arithmetic operation
2. If src1(X) and src2(X) are constant, then change X by applying the operation

Local: Constant Combining



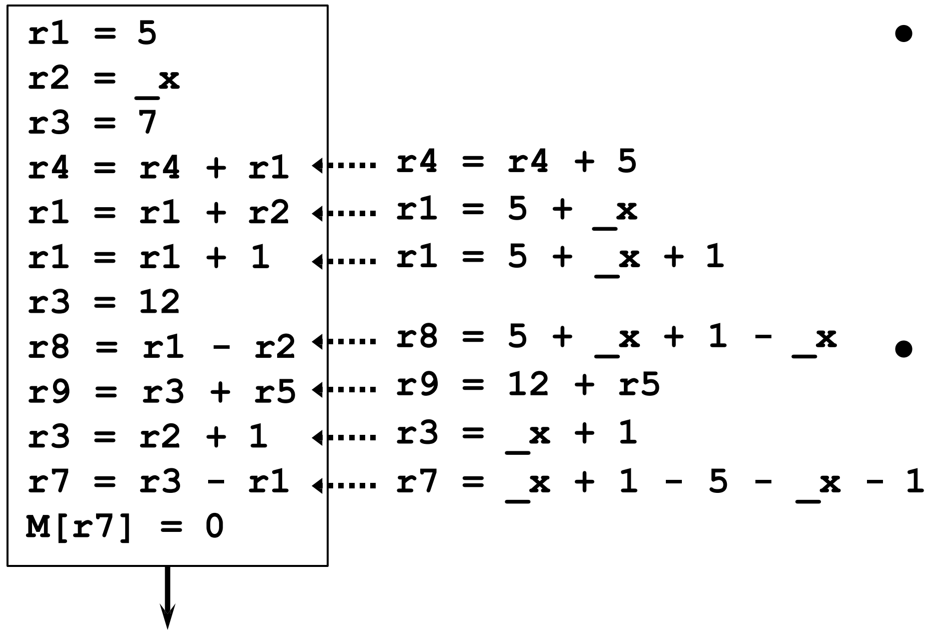
- Goal: eliminate unnecessary operations
  - First operation often becomes dead after constant combining
- Rules:
  1. Operations X and Y in same basic block
  2. X and Y have at least one literal src
  3. Y uses dest(X)
  4. None of the srcs of X have defs between X and Y (excluding Y)

Local: Strength Reduction



- Goal: replace expensive operations with cheaper ones
- Rules (common):
  1. X is an multiplication operation where src1(X) or src2(X) is a const 2^k integer literal
  2. Change X by using shift operation
  3. For k=1 can use add

Local: Constant Propagation



```
r1 = 5
r2 = _x
r3 = 7
r4 = r4 + r1  <····  r4 = r4 + 5
r1 = r1 + r2  <····  r1 = 5 + _x
r1 = r1 + 1   <····  r1 = 5 + _x + 1
r3 = 12
r8 = r1 - r2  <····  r8 = 5 + _x + 1 - _x
r9 = r3 + r5  <····  r9 = 12 + r5
r3 = r2 + 1   <····  r3 = _x + 1
r7 = r3 - r1  <····  r7 = _x + 1 - 5 - _x - 1
M[r7] = 0
```

- Goal: replace register uses with literals (constants) in a single basic block
- Rules:

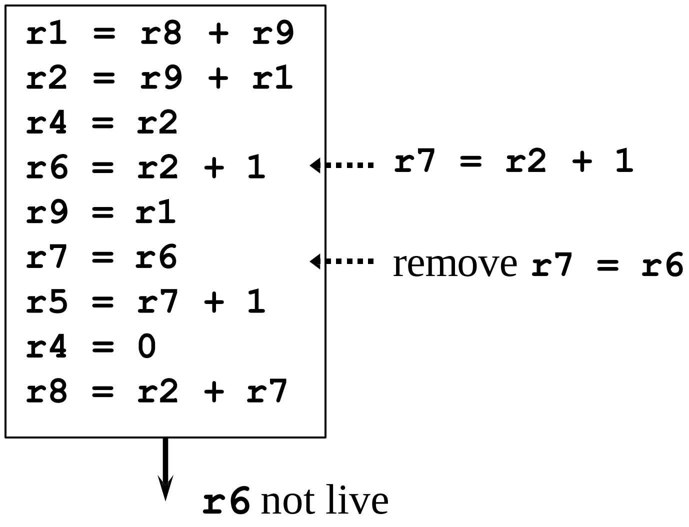
1. Operation X is a move to register with src1(X) literal
2. Operation Y uses dest(X)
3. There is no def of dest(X) between X and Y (excluding defs at X and Y)
4. Replace dest(X) in Y with src1(X)

Local: Common Subexpression Elimination (CSE)

```
r1 = r2 + r3
r4 = r4 + 1
r1 = 6
r6 = r2 + r3
r2 = r1 - 1
r5 = r4 + 1
r7 = r2 + r3
r5 = r1 - 1  <···· r5 = r2
```

- Goal: eliminate re-computations of an expression
  - More efficient code
  - Resulting moves can get copy propagated (see later)
- Rules:
  1. Operations X and Y have the same opcode and Y follows X
  2. src(X) = src(Y) for all srcs
  3. For all srcs, no def of a src between X and Y (excluding Y)
  4. No def of dest(X) between X and Y (excluding X and Y)
  5. Replace Y with move dest(Y) = dest(X)

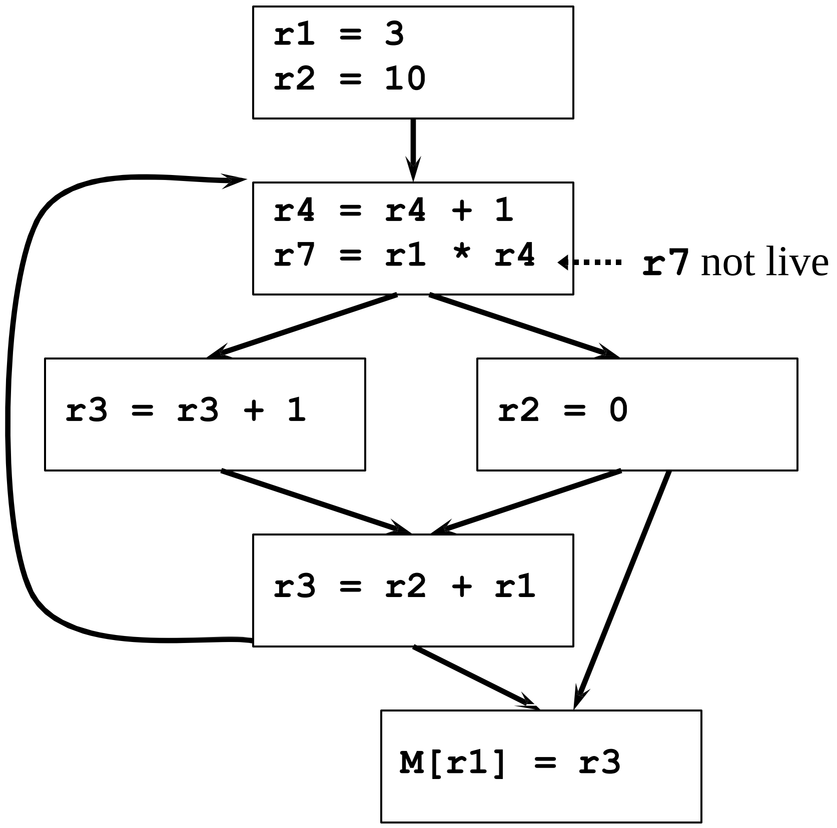
Local: Backward Copy Propagation



```
r1 = r8 + r9
r2 = r9 + r1
r4 = r2
r6 = r2 + 1
r9 = r1
r7 = r6
r5 = r7 + 1
r4 = 0
r8 = r2 + r7
```

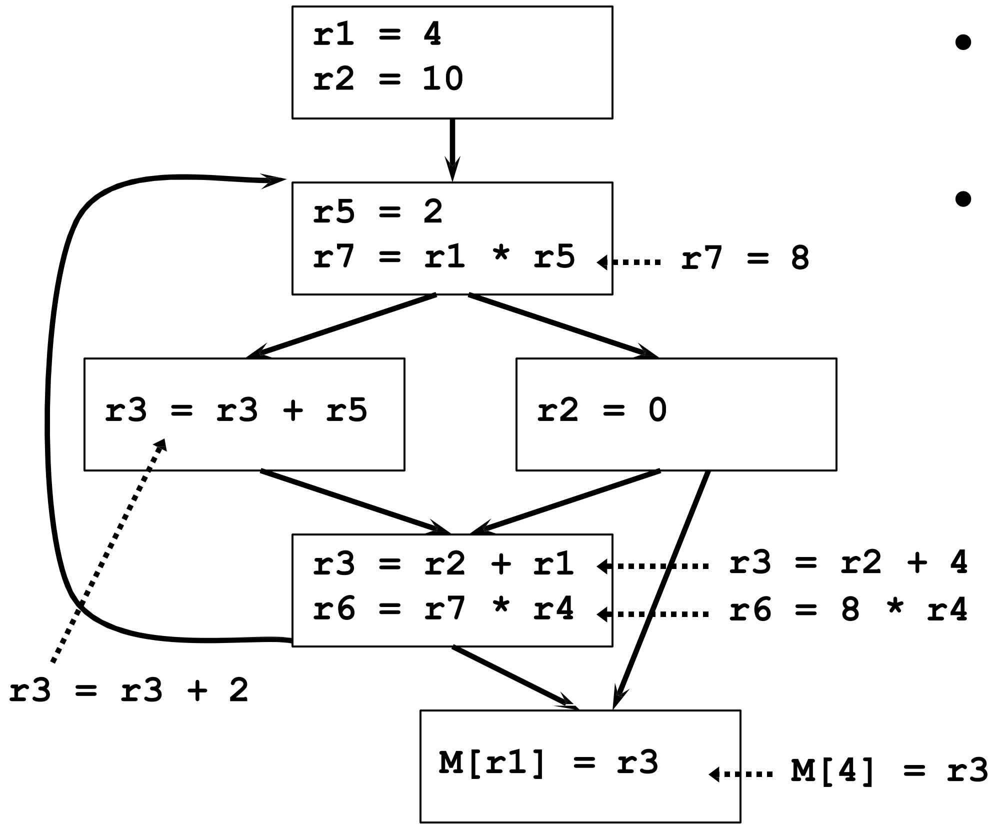
- Goal: propagate LHS of moves backward
  - Eliminates useless moves
- Rules (dataflow required)
  1. X and Y in same block
  2. Y is a move to register
  3. dest(X) is a register that is not live out of the block
  4. Y uses dest(X)
  5. dest(Y) not used or defined between X and Y (excluding X and Y)
  6. No uses of dest(X) after the first redef of dest(Y)
  7. Replace src(Y) on path from X to Y with dest(X) and remove Y

Global: Dead Code Elimination



- Goal: eliminate any operation who's result is never used
- Rules (dataflow required)
  1. X is an operation with no use in def-use (DU) chain, i.e. dest(X) is not live
  2. Delete X if removable (not a mem store or branch)
- Rules too simple!
  - Misses deletion of **`r4`**, even after deleting **`r7`**, since **`r4`** is live in loop
  - Better is to trace UD chains backwards from "critical" operations

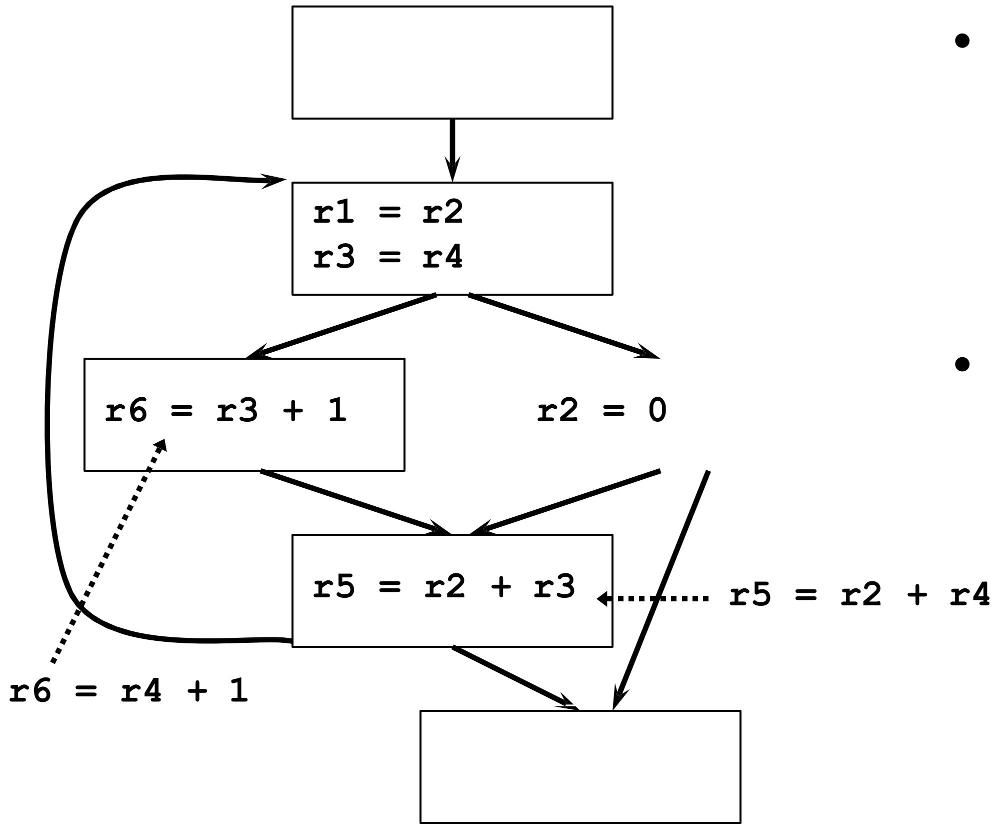
Global: Constant Propagation



- Goal: globally replace register uses with literals
- Rules (dataflow required)

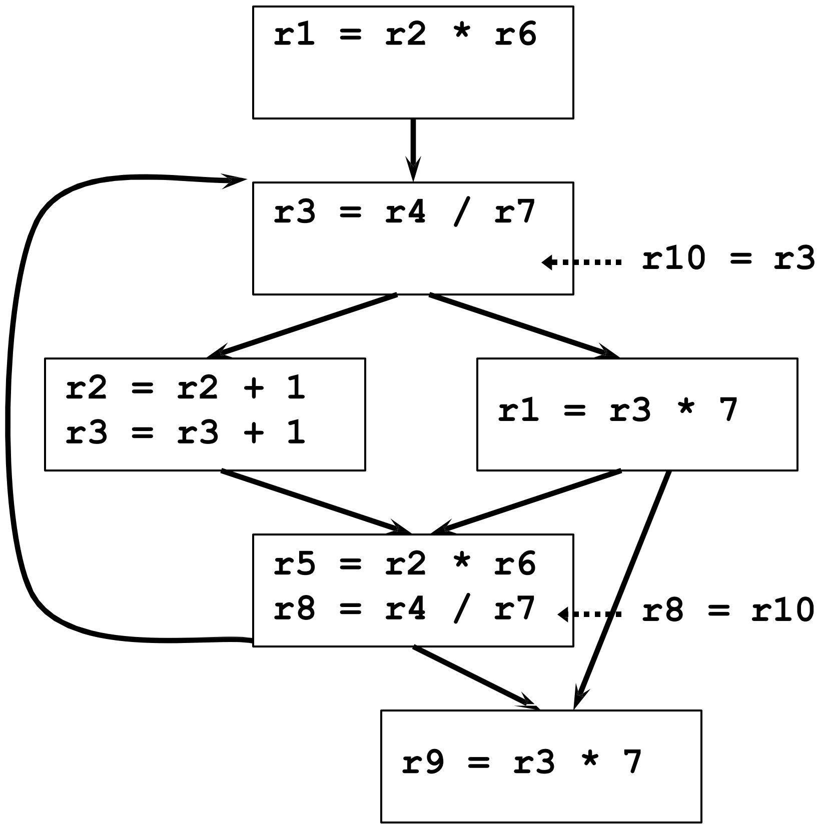
1. X is a move to a register with src1(X) literal
2. Y uses dest(X)
3. dest(X) has only one def at X for use-def (UD) chains to Y
4. Replace dest(X) in Y with src1(X)

Global: Forward Copy Propagation



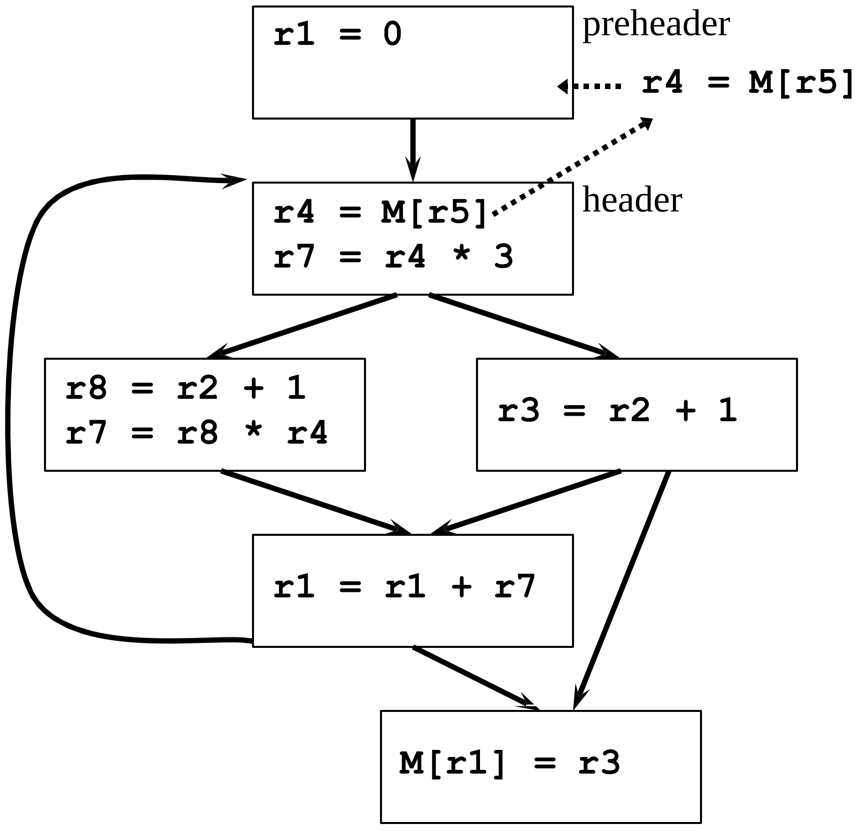
- Goal: globally propagate RHS of moves forward
  - Reduces dependence chain
  - May be possible to eliminate moves
- Rules (dataflow required)
  1. X is a move with src1(X) register
  2. Y uses dest(X)
  3. dest(X) has only one def at X for UD chains to Y
  4. src1(X) has no def on any path from X to Y
  5. Replace dest(X) in Y with src1(X)

Global: Common Subexpression Elimination (CSE)



- Goal: eliminate recomputations of an expression
- Rules:
  1. X and Y have the same opcode and X dominates Y
  2. src(X) = src(Y) for all srcs
  3. For all srcs, no def of a src on any path between X and Y (excluding Y)
  4. Insert rx = dest(X) immediately after X for new register rx
  5. Replace Y with move dest(Y) = rx

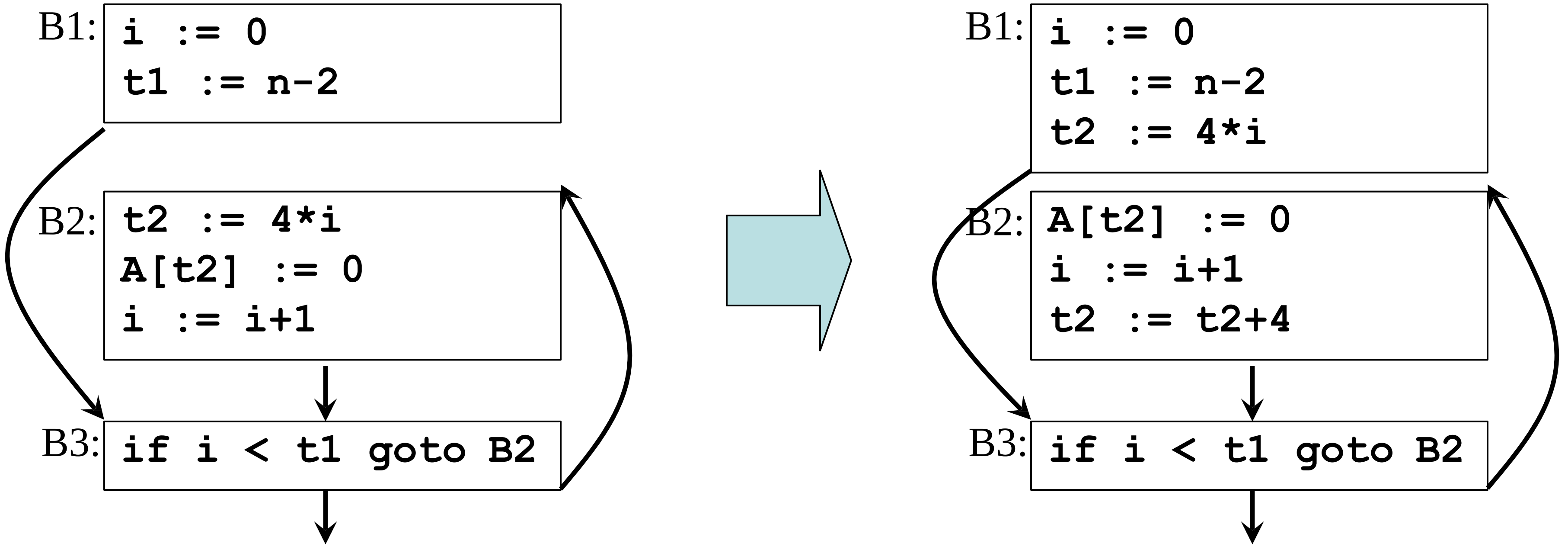
Global: Code Motion



- Goal: move loop-invariant computations to preheader
- Rules:

1. Operation X in block that dominates all exit blocks
2. X is the only operation to modify dest(X) in loop body
3. All srcs of X have no defs in any of the basic blocks in the loop body
4. Move X to end of preheader
5. Note 1: if one src of X is a memory load, need to check for stores in loop body
6. Note 2: X must be movable and not cause exceptions

Global: Loop Strength Reduction



```
B1: i  := 0
    t1 := n-2

B2: t2 := 4*i
    A[t2] := 0
    i  := i+1

B3: if i < t1 goto B2
```

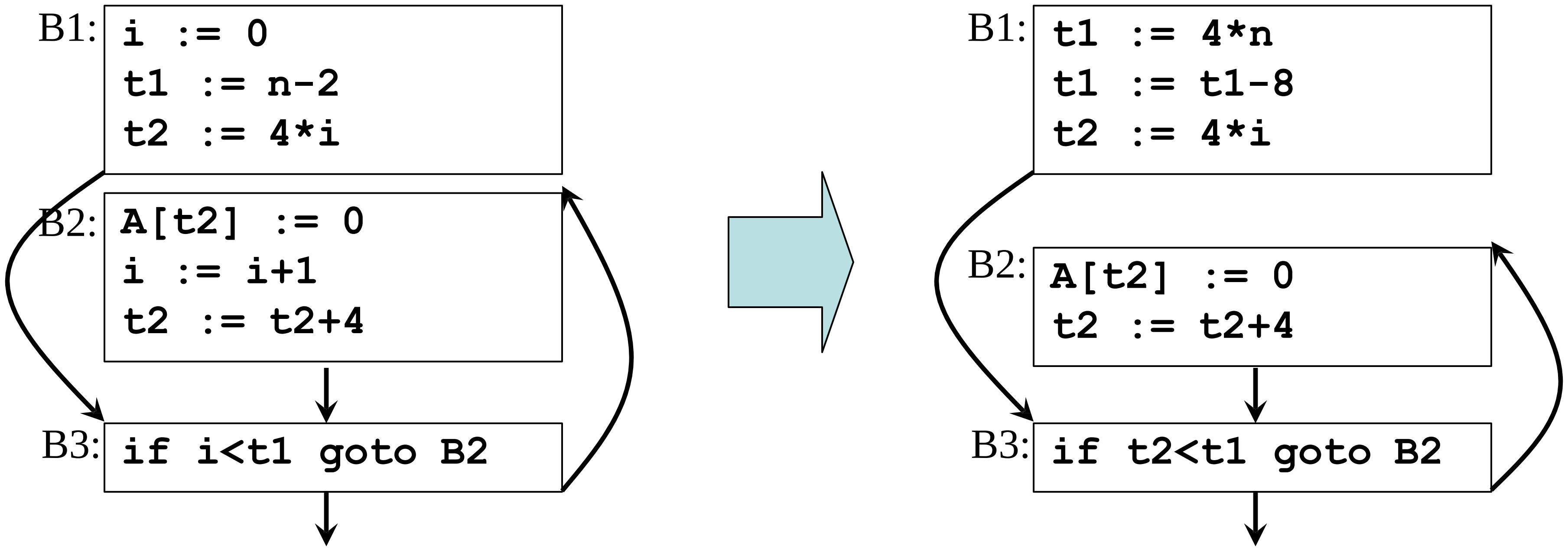
```
B1: i  := 0
    t1 := n-2
    t2 := 4*i

B2: A[t2] := 0
    i  := i+1
    t2 := t2+4

B3: if i < t1 goto B2
```

Replace expensive computations with *induction variables*

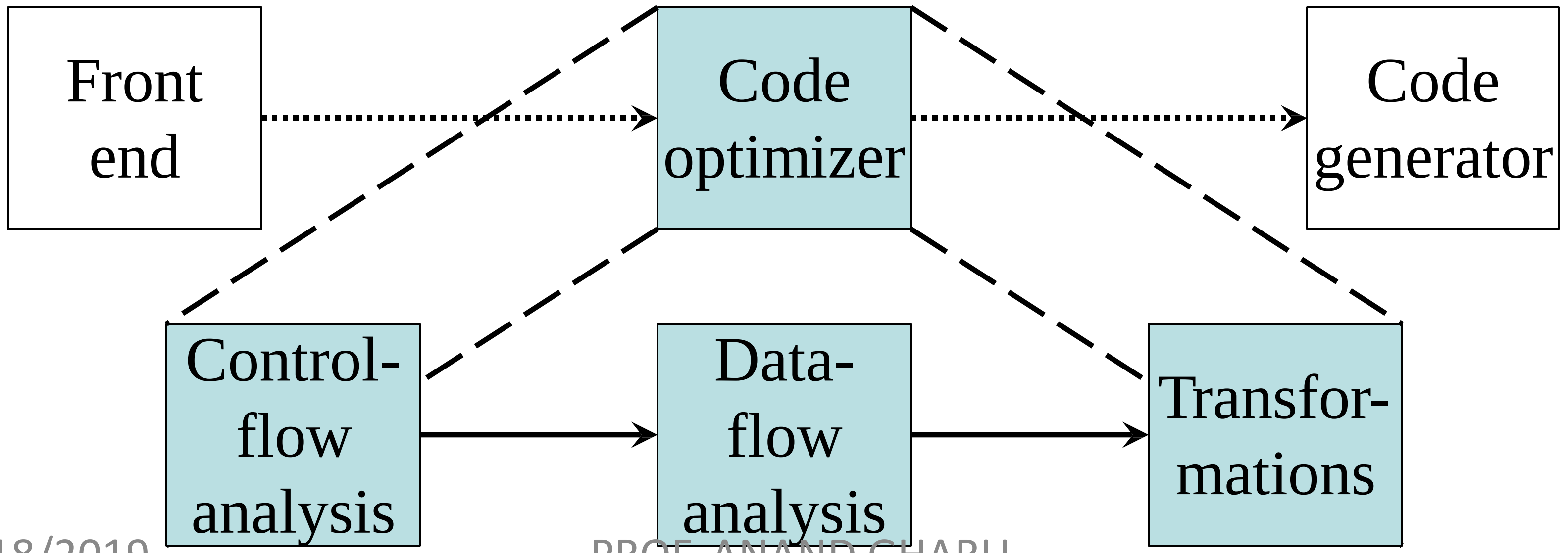
Global: Induction Variable Elimination



Replace induction variable in expressions with another

The Code Optimizer

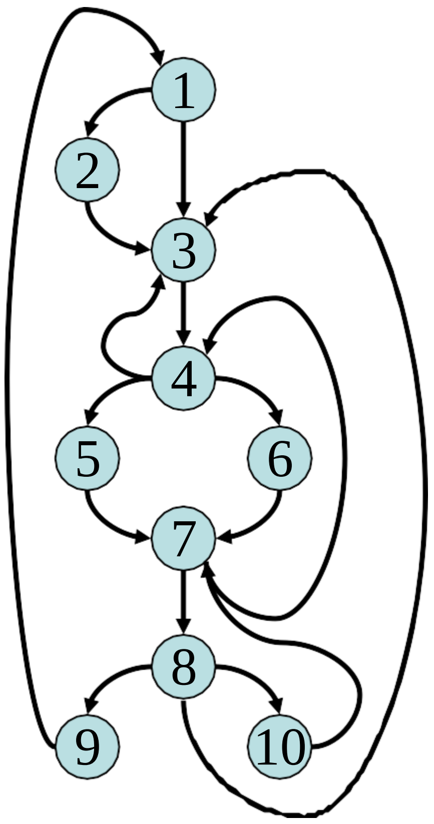
- Control flow analysis
- Data-flow analysis
- Transformations



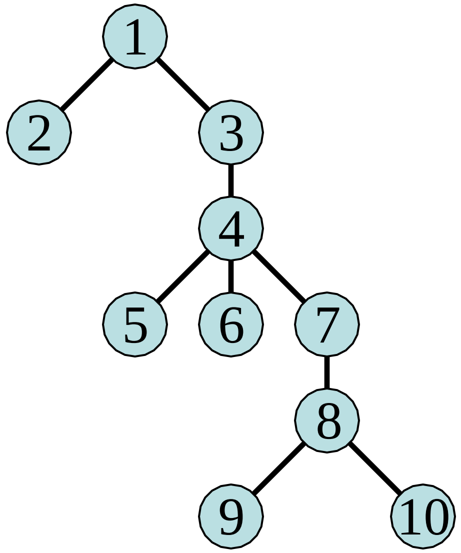
Determining Loops in Flow Graphs: Dominators

- Dominators: *d dom n*
  - Node *d* of a CFG *dominates* node *n* if *every* path from the initial node of the CFG to *n* goes through *d*
  - The loop entry dominates all nodes in the loop
- The *immediate dominator m* of a node *n* is the last dominator on the path from the initial node to *n*
  - If $d \neq n$ and *d dom n* then *d dom m*

Dominator Trees



CFG

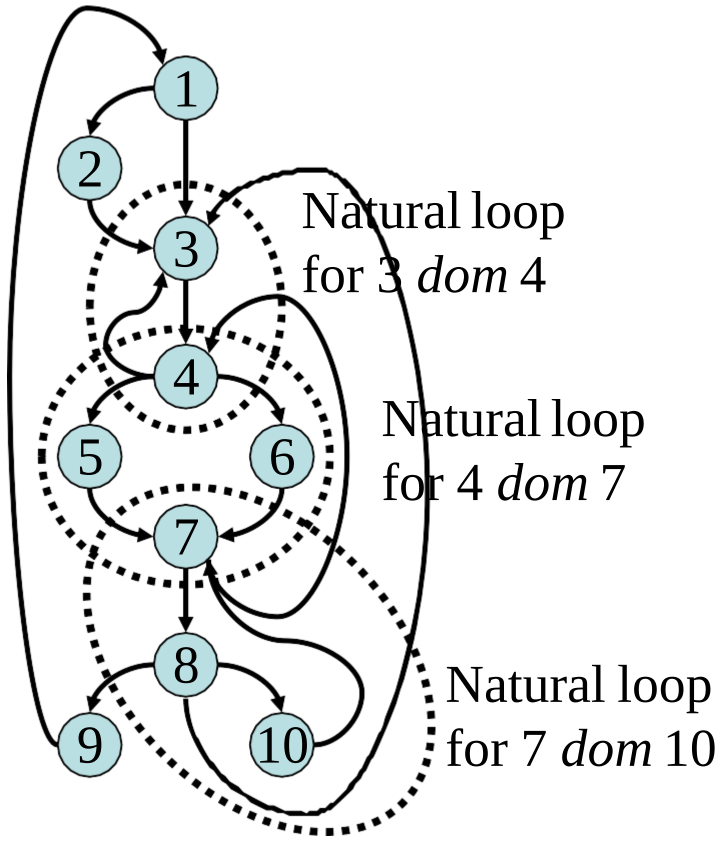


Dominator tree

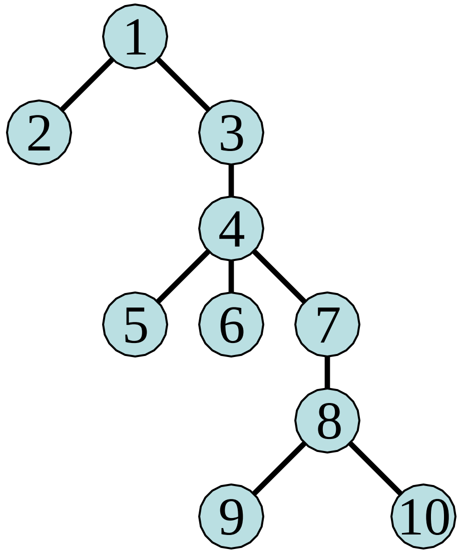
Natural Loops

- A *back edge* is is an edge $a \rightarrow b$ whose head b dominates its tail a
- Given a back edge $n \rightarrow d$
  - The *natural loop* consists of d plus the nodes that can reach n without going through d
  - The *loop header* is node d
- Unless two loops have the same header, they are disjoint or one is nested within the other
  - A nested loop is an *inner loop* if it contains no other loops

Natural Inner Loops Example

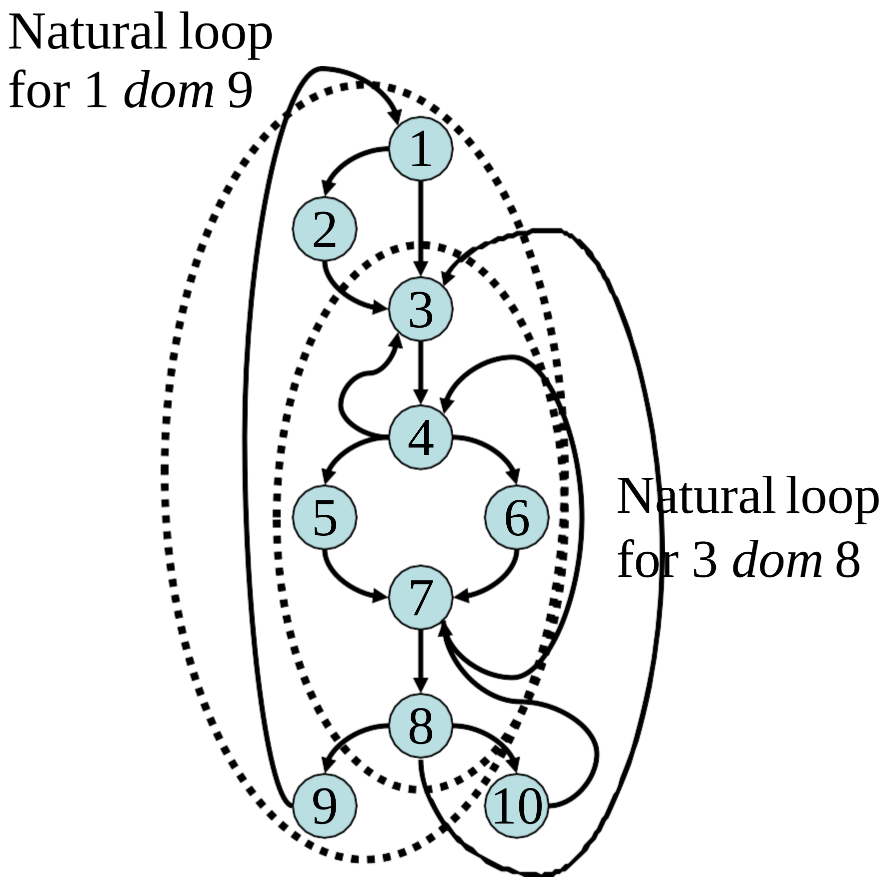

1
2
3
4
5
6
7
8
9
10
Natural loop for 3 *dom* 4
Natural loop for 4 *dom* 7
Natural loop for 7 *dom* 10

CFG

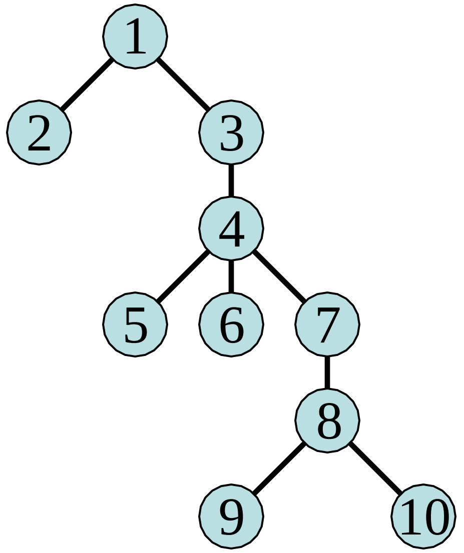


Dominator tree

Natural Outer Loops Example



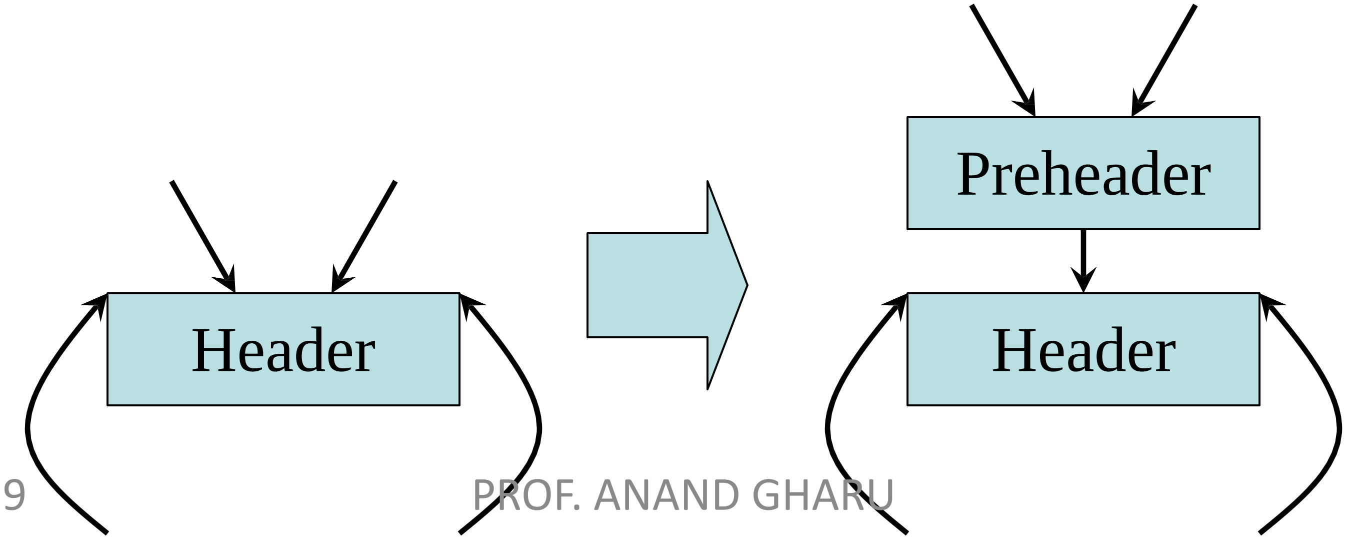
CFG



Dominator tree

Pre-Headers

- To facilitate loop transformations, a compiler often adds a *preheader* to a loop
- Code motion, strength reduction, and other loop transformations populate the preheader



Data Flow Analysis

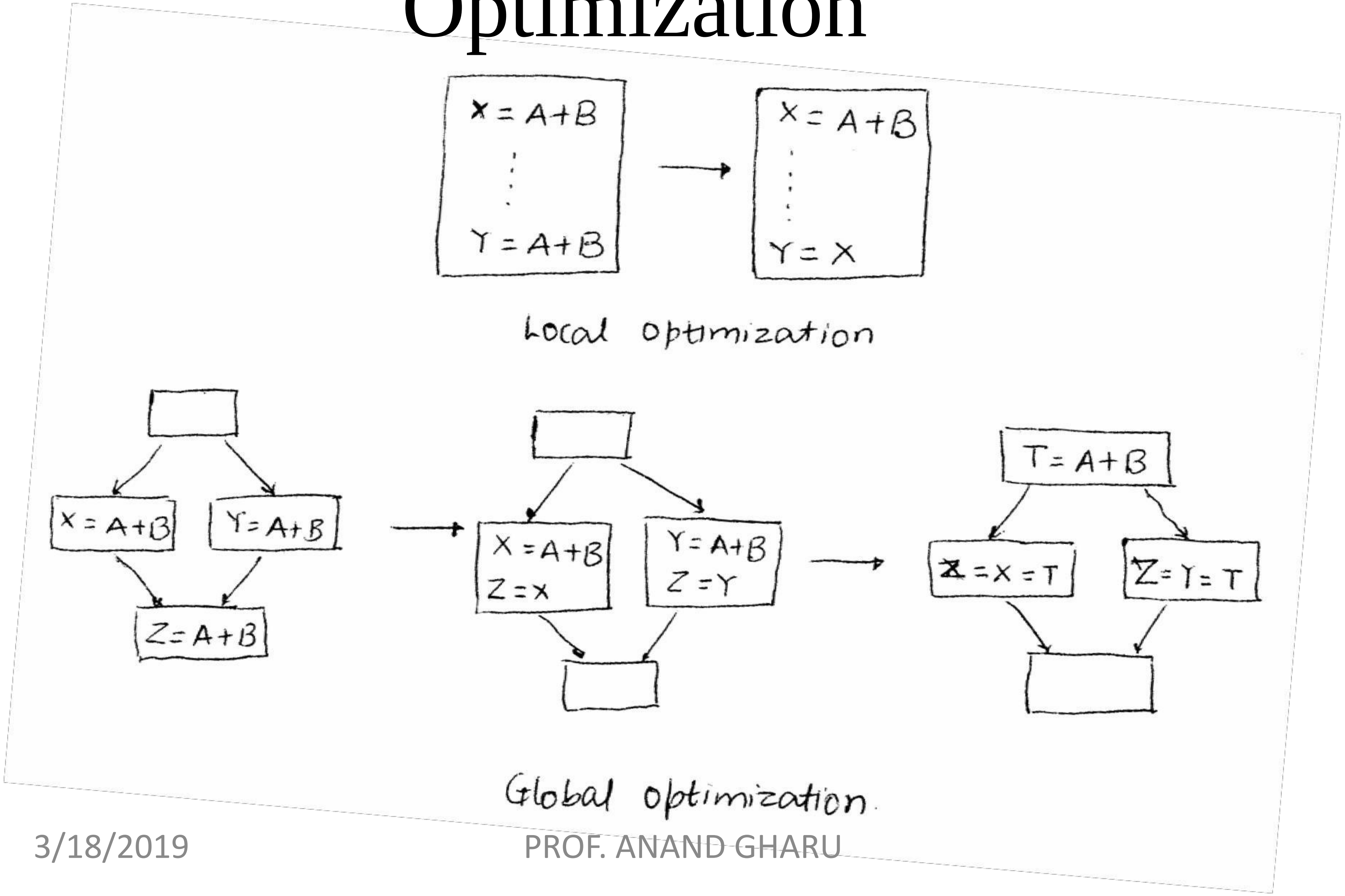
Data flow analysis is used to collect information about the flow of data values across basic blocks.

•Dominator analysis collected global information regarding the program’s structure

•For performing global code optimizations global information must be collected regarding values of program variables.

- Local optimizations involve statements from same basic block
- Global optimizations involve statements from different basic blocks → data flow analysis is performed to collect global information that drives global optimizations

Local and Global Optimization



Applications of Data Flow Analysis

- Applicability of code optimizations
- Symbolic debugging of code
- Static error checking
- Type inference
- …….

Global Data Flow Analysis

- These Analyze following properties :

1. Reaching Definitions : Constant Propagation, Dead code Elimination, copy propogation
2. Available Expressions : Common Subexpression Elimination
3. Live Variable Analysis : Register Allocation
4. Very Busy Expressions : Dead code Elimination , Register Allocation

Data Flow Equation

Data Flow Analysis is done by the equation :

$$Out[S] = gen[S] \cup (in[S] - kill[S])$$

i.e information at the end is either generated within the statement or at the beginning and not killed as control flows through the statement

1. Reaching Definitions

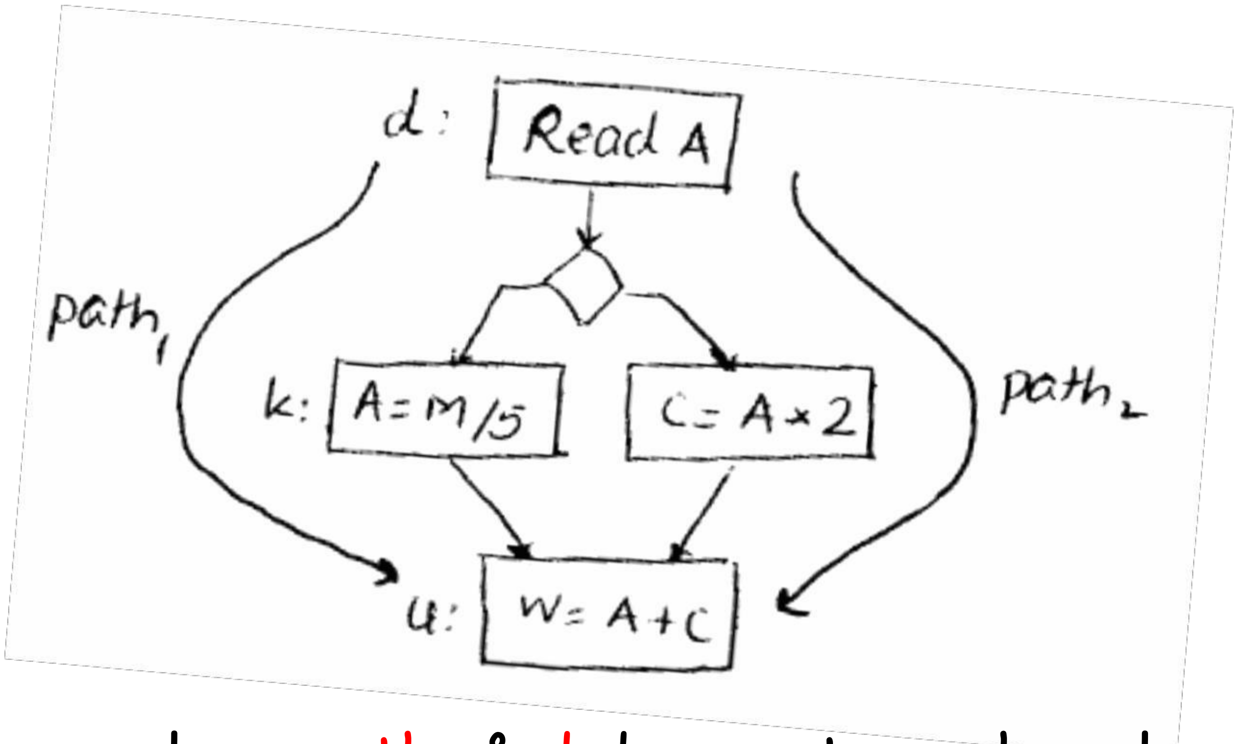
Definition d of variable v: a statement d that assigns a value to v.

Use of variable v: reference to value of v in an expression evaluation.

Definition d of variable v reaches a point p if there exists a path from immediately after d to p such that definition d is not killed along the path.

Definition d is killed along a path between two points if there exists an assignment to variable v along the path.

Example



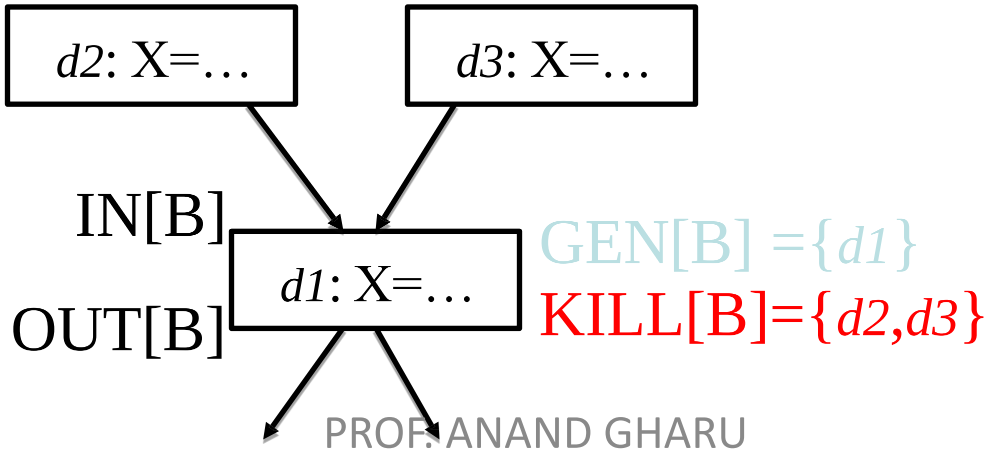
d reaches u along $path_2$ & d does not reach u along $path_1$

Since there exists a path from d to u along which d is not killed (i.e., $path_2$), d reaches u.

Computing Reaching Definitions

At each program point p, we compute the set of definitions that reach point p.

Reaching definitions are computed by solving a system of equations (data flow equations).



Data Flow Equations

IN[B]: Definitions that reach B's entry.
OUT[B]: Definitions that reach B's exit.

$$IN[B] = \bigcup_{p \in pred(B)} OUT(p)$$

$$OUT(B) = GEN(B) \cup (IN(B) - KILL(B))$$

GEN[B]: Definitions within B that reach the end of B.
KILL[B]: Definitions that never reach the end of B due to redefinitions of variables in B.

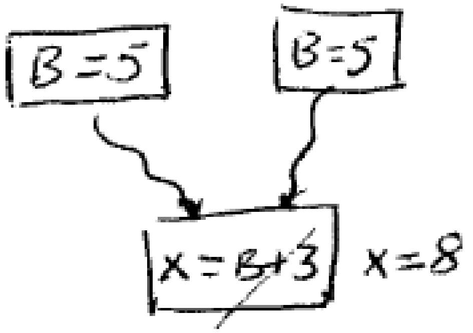
Reaching Definitions Contd.

- Forward problem – information flows forward in the direction of edges.
- May problem – there is a path along which definition reaches a point but it does not always reach the point.

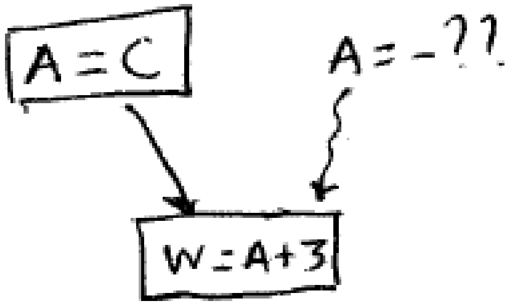
Therefore in a May problem the meet operator is the Union operator.

Applications of Reaching Definitions

- Constant Propagation/folding



- Copy Propagation



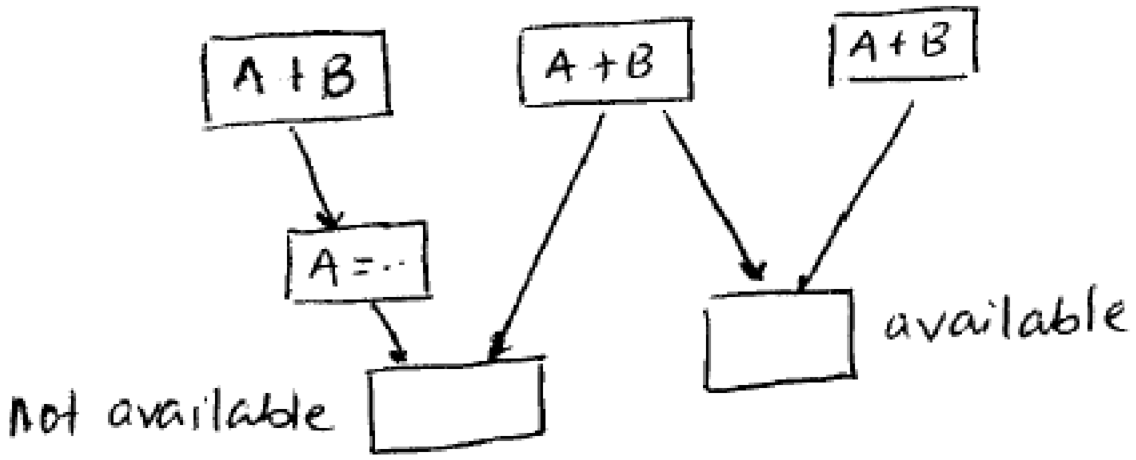
2. Available Expressions

An expression is generated at a point if it is computed at that point.

An expression is killed by redefinitions of operands of the expression.

An expression A+B is available at a point if every path from the start node to the point evaluates A+B and after the last evaluation of A+B on each path there is no redefinition of either A or B (i.e., A+B is not killed).

Available Expressions



Available expressions problem computes: at each program point the set of expressions available at that point.

Data Flow Equations

IN[B]: Expressions available at B's entry.
OUT[B]: Expressions available at B's exit.

$$IN[B] = \bigcap_{P \in pred(B)} OUT(P)$$

$$OUT[B] = GEN[B] \cup (IN[B] - KILL[B])$$

GEN[B]: Expressions computed within B that are available at the end of B.
KILL[B]: Expressions whose operands are redefined in B.

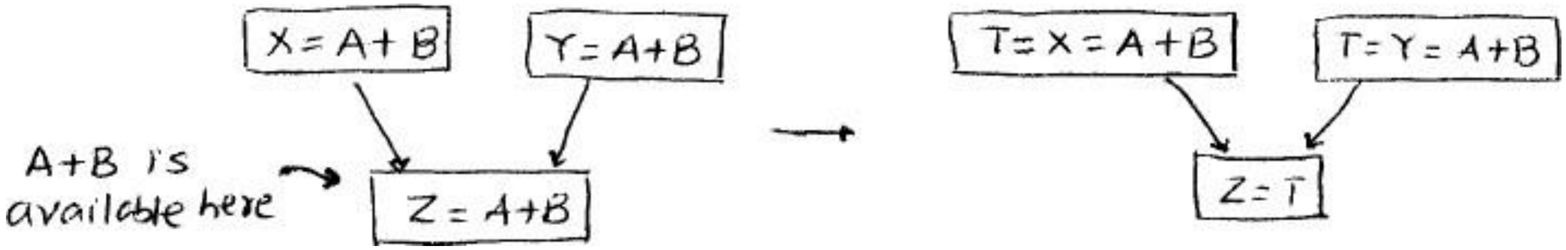
Available Expressions Contd.

- Forward problem – information flows forward in the direction of edges.
- Must problem – expression is definitely available at a point along all paths.

Therefore in a Must problem the meet operator is the Intersection operator.

Applications of Available Expressions

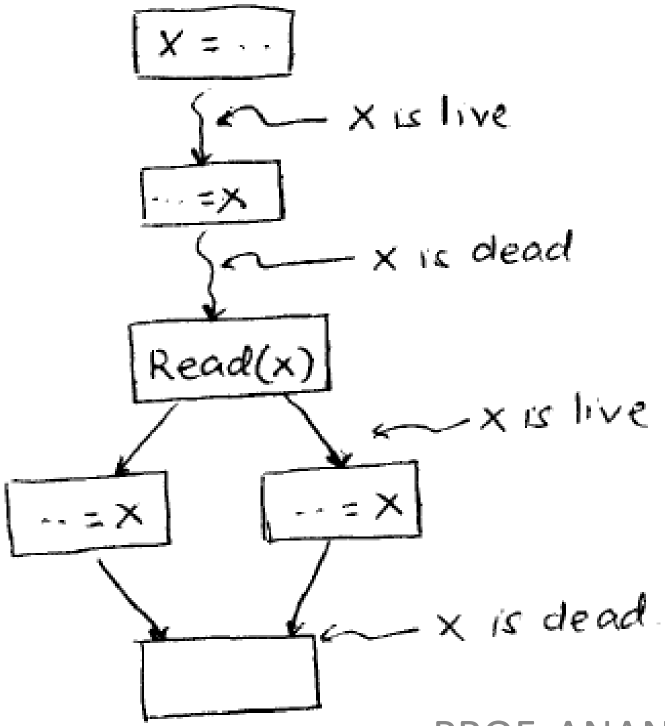
- Common Subexpression Elimination



3. Live Variable Analysis

A path is X-clear is it contains no definition of X.

A variable X is live at point p if there exists a X-clear path from p to a use of X; otherwise X is dead at p.



Live Variable Analysis Computes:

At each program point p identify the set of variables that are live at p.

Data Flow Equations

IN[B]: Variables live at B's entry.
OUT[B]: Variables live at B's exit.

$$OUT[B] = \bigcup_{S \in Succ(B)} IN[S]$$

$$IN[B] = GEN[B] \cup (OUT[B] - KILL[B])$$

GEN[B]: Variables that are used in B prior to their definition in B.
KILL[B]: Variables definitely assigned value in B before any use of that variable in B.

Live Variables Contd.

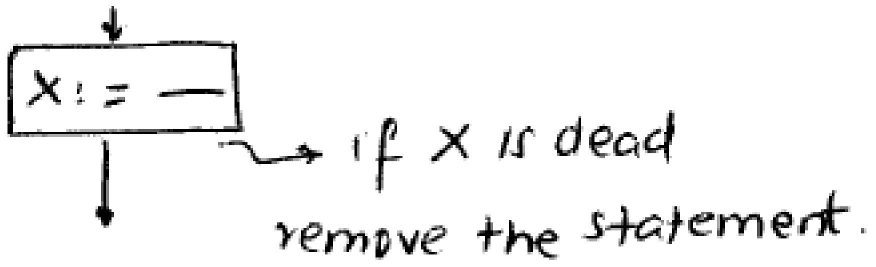
- Backward problem – information flows backward in reverse of the direction of edges.
- May problem – there exists a path along which a use is encountered.

Therefore in a May problem the meet operator is the Union operator.

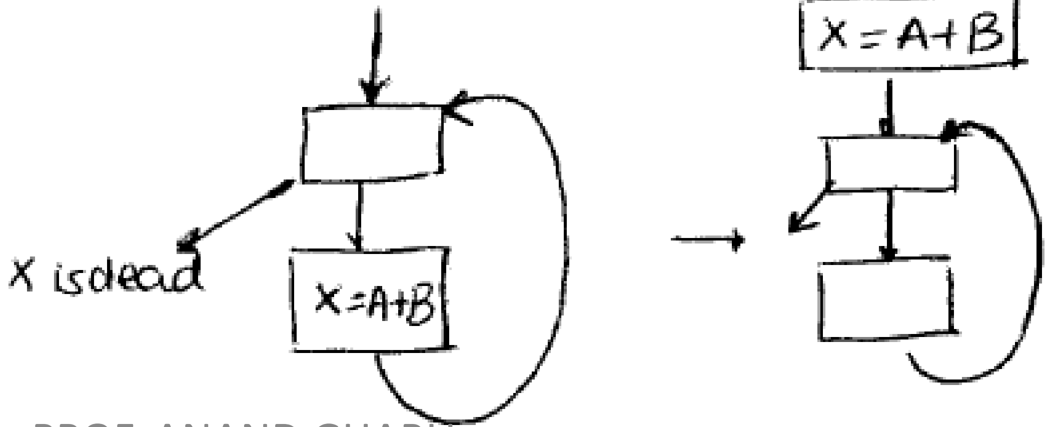
Applications of Live Variables

- Register Allocation

- Dead Code Elimination

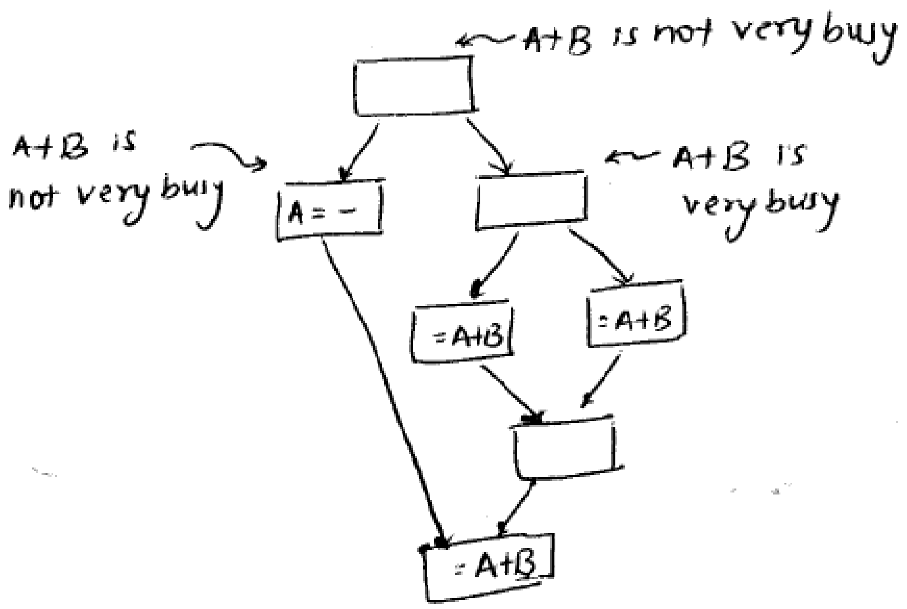


- Code Motion Out of Loops

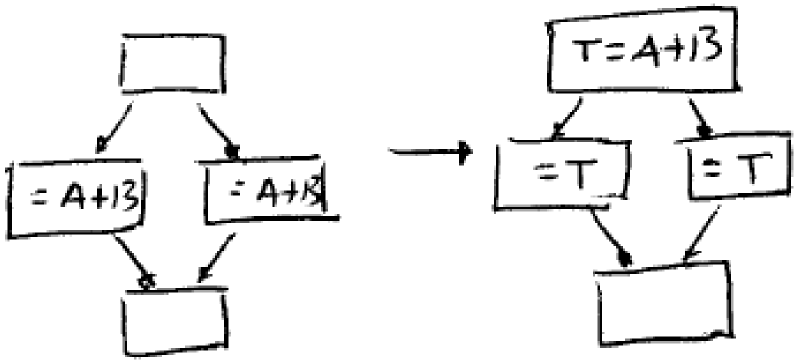


4. Very Busy Expressions

A expression A+B is very busy at point p if for all paths starting at p and ending at the end of the program, an evaluation of A+B appears before any definition of A or B.



Application:
Code Size Reduction



Compute for each program point the set of very busy expressions at the point.

Data Flow Equations

IN[B]: Expressions very busy at B's entry.
OUT[B]: Expressions very busy at B's exit.

$$OUT[B] = \bigcap_{S \in Suc(B)} IN[S]$$

$$IN[B] = GEN[B] \cup (OUT[B] - KILL[B])$$

GEN[B]: Expression computed in B and variables used in the expression are not redefined in B prior to expression's evaluation in B.
KILL[B]: Expressions that use variables that are redefined in B.

Very Busy Expressions Contd.

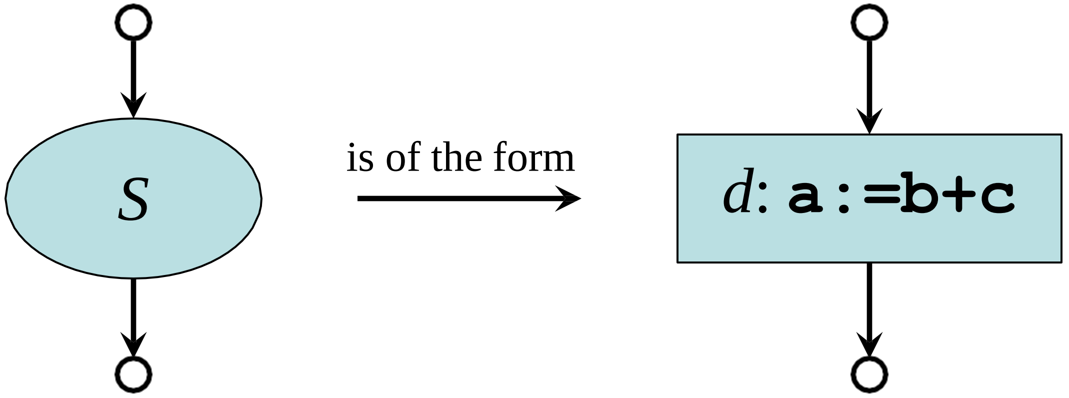
- Backward problem – information flows backward in reverse of the direction of edges.
- Must problem – expressions must be computed along all paths.

Therefore in a Must problem the meet operator is the Intersection operator.

Summary

	May/Union	Must/Intersection
Forward	Reaching Definitions	Available Expressions
Backward	Live Variables	Very Busy Expressions

Reaching Definitions



Then, the data-flow equations for *S* are:

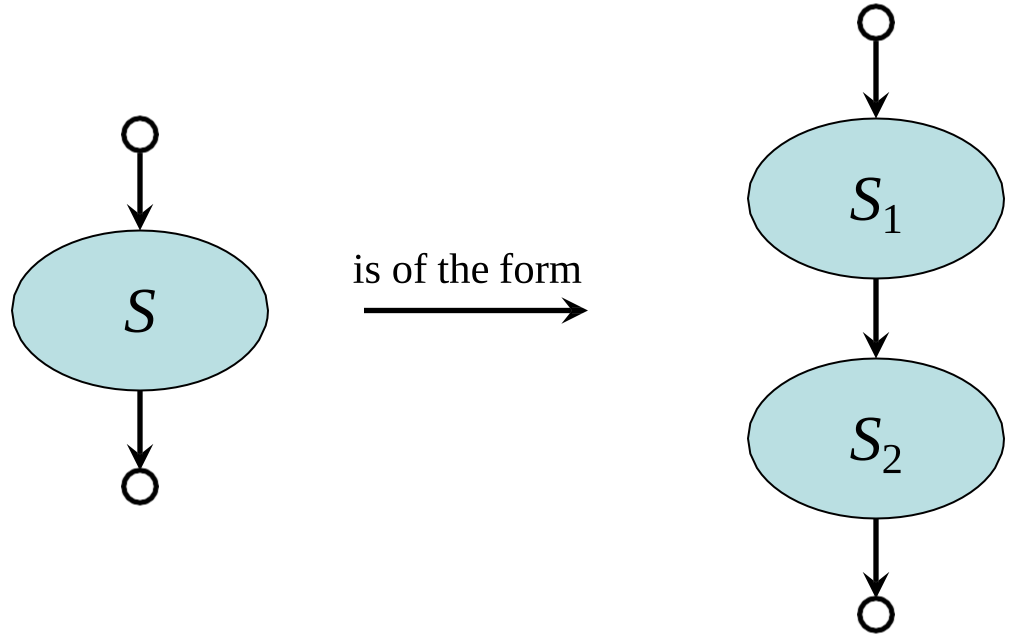
$gen[S] \quad = \{d\}$

$kill[S] \quad = D_{\mathtt{a}} - \{d\}$

$out[S] \quad = gen[S] \cup (in[S] - kill[S])$

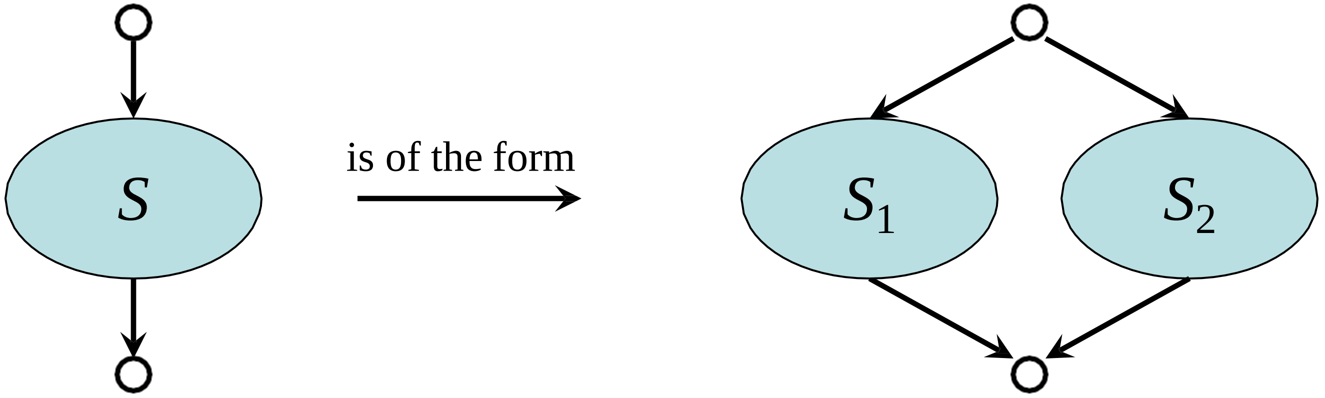
where $D_{\mathtt{a}}$ = all definitions of **a** in the region of code

Reaching Definitions



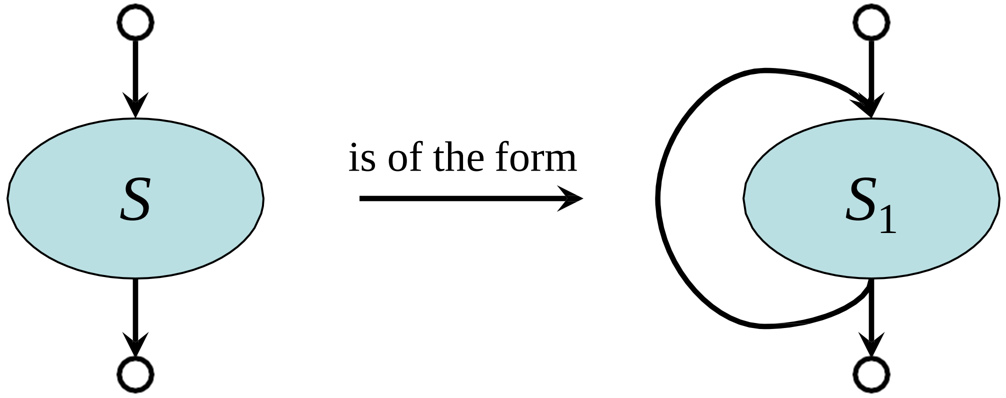
$$
\begin{aligned}
gen[S] &= gen[S_2] \cup (gen[S_1] - kill[S_2]) \\
kill[S] &= kill[S_2] \cup (kill[S_1] - gen[S_2]) \\
in[S_1] &= in[S] \\
in[S_2] &= out[S_1] \\
out[S] &= out[S_2]
\end{aligned}
$$

Reaching Definitions



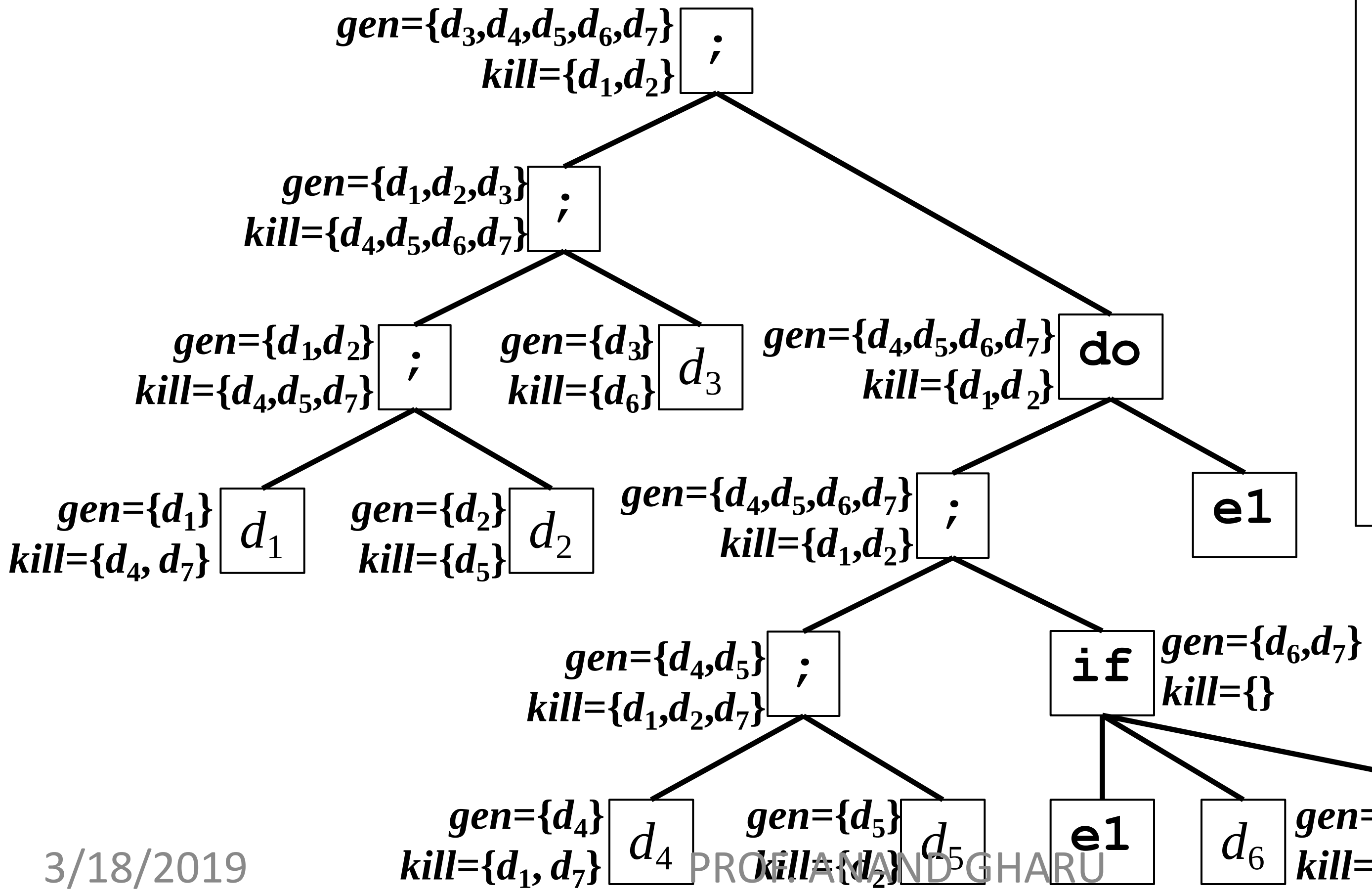
$$
\begin{aligned}
gen[S] &= gen[S_1] \cup gen[S_2] \\
kill[S] &= kill[S_1] \cap kill[S_2] \\
in[S_1] &= in[S] \\
in[S_2] &= in[S] \\
out[S] &= out[S_1] \cup out[S_2]
\end{aligned}
$$

Reaching Definitions



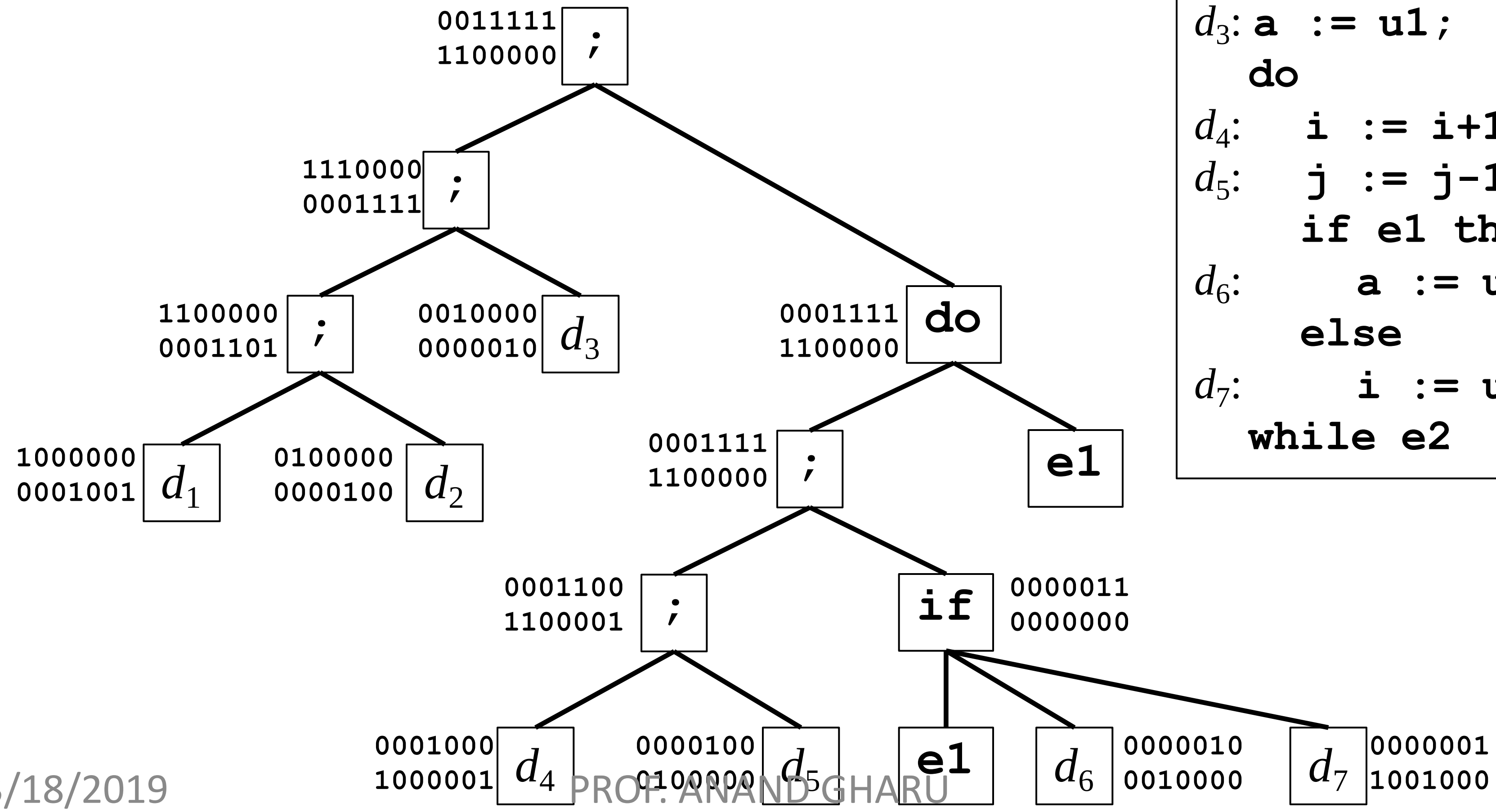
$$
\begin{aligned}
gen[S] &= gen[S_1] \\
kill[S] &= kill[S_1] \\
in[S_1] &= in[S] \cup gen[S_1] \\
out[S] &= out[S_1]
\end{aligned}
$$

Example Reaching Definitions


gen={d_3,d_4,d_5,d_6,d_7} kill={d_1,d_2}
;
gen={d_1,d_2,d_3} kill={d_4,d_5,d_6,d_7}
;
gen={d_1,d_2} kill={d_4,d_5,d_7}
;
gen={d_3} kill={d_6}
d_3
gen={d_4,d_5,d_6,d_7} kill={d_1,d_2}
do
gen={d_1} kill={d_4, d_7}
d_1
gen={d_2} kill={d_5}
d_2
gen={d_4,d_5,d_6,d_7} kill={d_1,d_2}
;
e1
gen={d_4,d_5} kill={d_1,d_2,d_7}
;
if
gen={d_6,d_7} kill={}
gen={d_4} kill={d_1, d_7}
d_4
gen={d_5} kill={d_2}
d_5
e1
d_6
gen={d_6} kill={d_3}
d_7
gen={d_7} kill={d_1,d_4}

```
d1: i  := m-1;
d2: j  := n;
d3: a  := u1;
    do
d4:     i  := i+1;
d5:     j  := j-1;
        if e1 then
d6:         a  := u2
        else
d7:         i  := u3
    while e2
```

Using Bit-Vectors to Compute Reaching Definitions

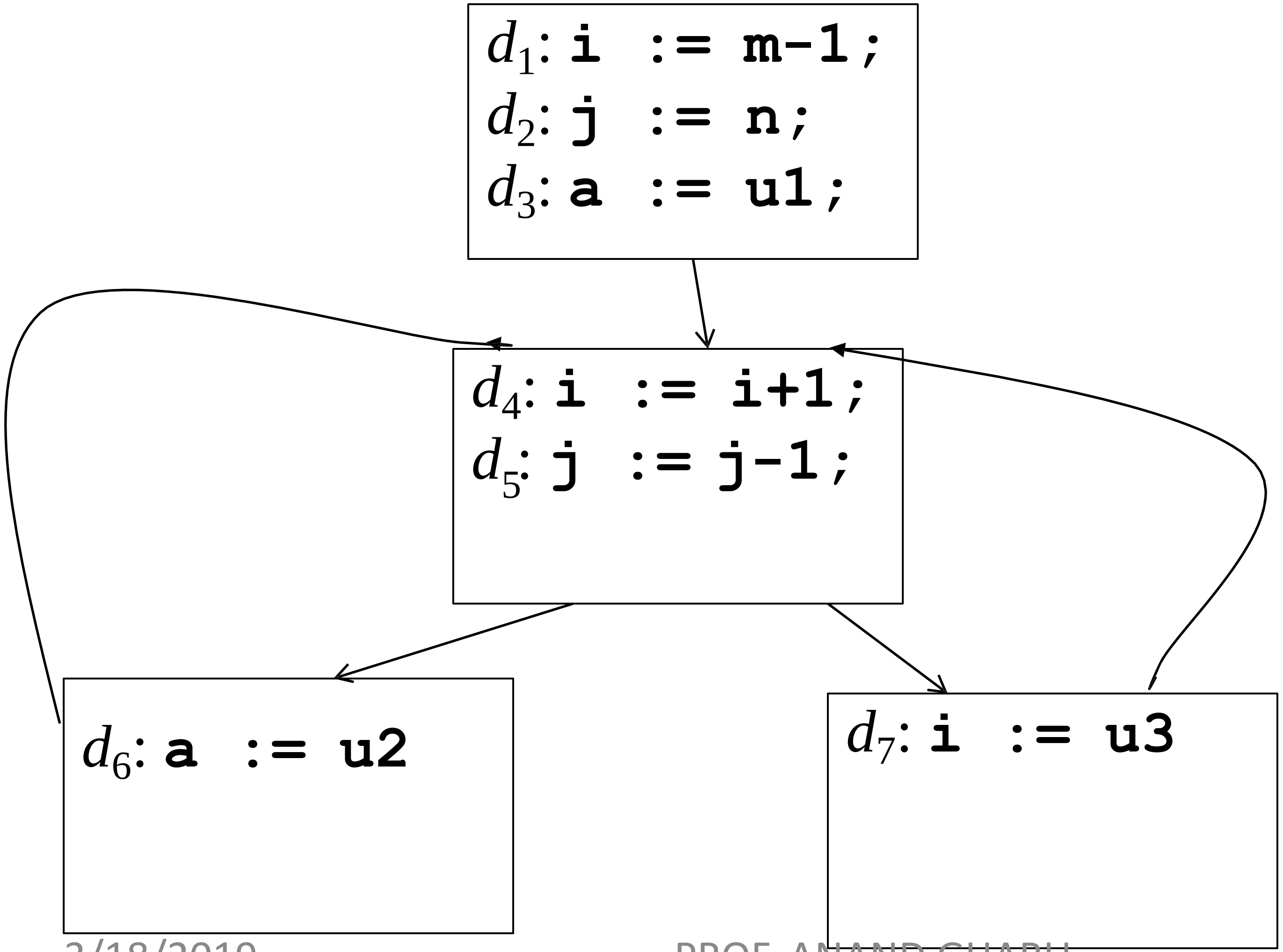


```
d1: i := m-1;
d2: j := n;
d3: a := u1;
    do
d4:     i := i+1;
d5:     j := j-1;
        if e1 then
d6:         a := u2
        else
d7:         i := u3
    while e2
```

Computation of *in* and *out* sets …for reaching definitions

```
for all basic blocks BB      $in$(BB) = $\varnothing$
for all basic blocks BB    $out$(BB) = gen(BB)
change = true
while (change) do
    change = false
    for each basic block BB, do
        $old\_out$ = $out$(BB)
        $in$(BB) = $\bigcup$($out$(Y)) for all predecessors Y of BB
        $out$(BB) = $gen$(BB) + ($in$(BB) – $kill$(BB))
        if ($old\_out$ != $out$(BB)) then change = true
    endfor
endfor
```

Example Reaching Definitions



```
d1: i := m-1;
d2: j := n;
d3: a := u1;
   do
d4:   i := i+1;
d5:   j := j-1;
      if e1 then
d6:     a := u2
      else
d7:     i := u3
   while e2
```

in(BB) = U(*out*(Y)) for all predecessors Y of BB

out(BB) = *gen*(BB) U (*in*(BB) – *kill*(BB))

Blocks	Initial		Pass1		Pass2	
	In	Out	In	Out	In	Out
B1	0000000	1110000	0000000	1110000	0000000	1110000
B2	0000000	0001100	1110011	0011110	1111111	0011110
B3	0000000	0000010	0011110	0001110	0011110	0001110
B4	0000000	0000001	0011110	0010111	0011110	0010111

Since out[B] of pass1 = out[B] of pass2 ….We Stop

Pass2	
In	Out
0000000	1110000
1111111	0011110
0011110	0001110
0011110	0010111

Thus we can finally say that at the end of block 4 the final definitions reaching are

d3, d5, d6, d7

Computation of *in* and *out* sets …for available expressions

```
in(B1) = ∅
out(B1) = gen(B1)
for  B ≠ B1 do out[B] := U - kill[B]
change = true
while (change) do
    change = false
    for B ≠ B1, do begin
        in(B) = ∩ (out(P)) for all predecessors P of BB
        old_out = out(B)
        out(B) = gen(B) U  (in(B) – kill(B))
        if (old_out != out(BB)) then change = true
    endfor
endfor
```

Live Variable (Liveness) Analysis

- Liveness: For each point p in a program and each variable *y*, determine whether *y* can be used before being redefined, starting at p.

- Attributes
  - ***use*** = set of variable used in the B prior to its definition
  - ***def*** = set of variables defined in B prior to any use of the variable
  - ***in*** = set of variables that are live at the entry point of a B
  - ***out*** = set of variables that are live at the exit point of a B

Live Variable (Liveness) Analysis

- Data flow equations:

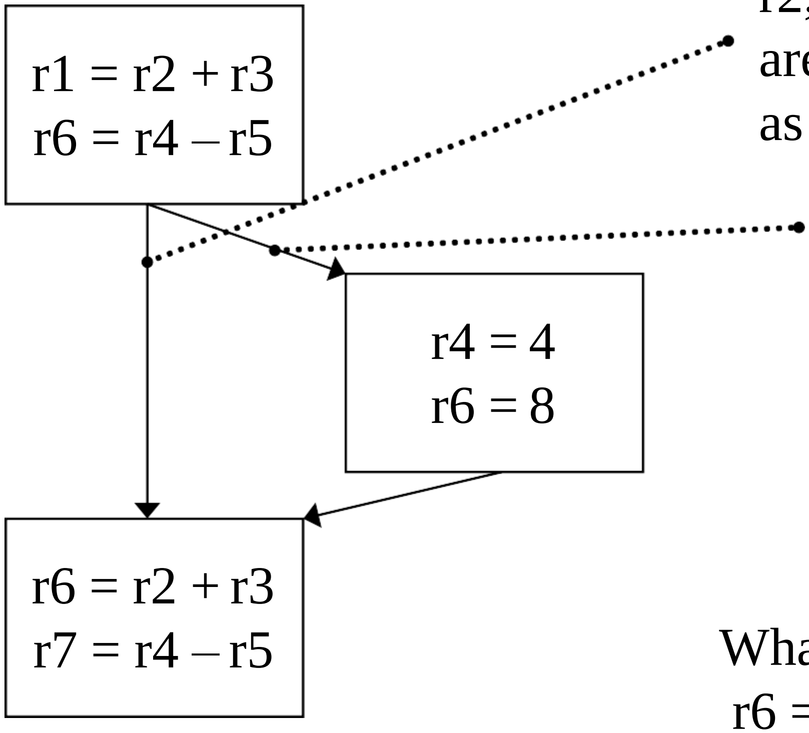
$$in[B] = use[B] \bigcup (out[B] - def[B])$$

$$out[B] = \bigcup_{S=succ(B)} in[S]$$

  - 1st Equation: a var is live, *coming in* the block, if either
    - •it is used before redefinition in B

      or
    - •it is live coming out of B and is not redefined in B
  - 2nd Equation: a var is live *coming out* of B, iff it is live coming in to one of its successors.

Example: Liveness



r2, r3, r4, r5 are all live as they are consumed later, r6 is dead as it is redefined later

r4 is dead, as it is redefined. So is r6. r2, r3, r5 are live

What does this mean?
r6 = r4 – r5 is useless,
it produces a dead value !!
Get rid of it!

THANK YOU!!!!!!!!!!

MY BLOG : *anandgharu.wordpress.com*