Principles of Compiler Construction

Lecturer: Change Huiyou

Note that most of these slides were created by:

Prof. Wen-jun LI
Dr. Zhong-mei SHU
Dr. Han LIN

Lecture 12. Code Optimization

1. Introduction
2. Local Optimization
3. Control-Flow Analysis and Loop Optimization
4. Data-Flow Analysis and Global Optimization

1. Introduction

- Terminology
  - Code optimization vs. code improvement
- Precondition
  - Semantics-preserving transformations
- Trade-off and consequence
  - Time efficiency vs. space efficiency
  - Compiler efficiency vs. target code efficiency

Optimization Levels

- Three levels of optimization
  - Source code
    - Manual, but the most effective.
  - Intermediate code
    - General and automatic.
    - Necessary even you write good source code.
  - Target code
    - Machine dependent (e.g. registers and pipelines).

Optimization Scopes

- Four scopes of optimization
  - Peephole optimization
    - Based on a sliding window, the smallest one.
  - Local optimization
    - Within a basic block.
  - Loop optimization
    - Within a loop.
  - Global optimization
    - The biggest scope.
    - In-Procedure vs. Inter-Procedure

```
sum = 0;
for (int i = 1; i <= 20; i++)   sum += A[i] * B[i];
```

An Example

B_1

(1)	sum	=	0
(2)	i	=	1

B_2

(3)	t_1	=	4 * i
(4)	t_2	=	addr(A) - 4
(5)	t_3	=	$t_2[t_1]$
(6)	t_4	=	4 * i
(7)	t_5	=	addr(B) - 4
(8)	t_6	=	$t_5[t_4]$
(9)	t_7	=	t_3 * t_6
(10)	sum	=	sum + t_7
(11)	i	=	i + 1
(12)	**if** i ≤ 20 **goto** B_2		

Common subexpressions

An Example: Eliminating Common Subexpressions

B_1

(1) sum = 0
(2) i = 1

B_2

(3) t_1 = 4 * i
(4) t_2 = addr(A) - 4
(5) t_3 = $t_2[t_1]$
(6) t_4 = t_1
(7) t_5 = addr(B) - 4
(8) t_6 = $t_5[t_4]$
(9) t_7 = t_3 * t_6
(10) sum = sum + t_7
(11) i = i + 1
(12) **if** i ≤ 20 **goto** B_2

Loop-invariant computations

Loop-invariant computations

An Example: Code Motion in Loop Optimization

B_1

(1)	sum	=	0
(2)	i	=	1
(4)	t_2	=	addr(A) - 4
(7)	t_5	=	addr(B) - 4

B_2

(3)	t_1	=	4 * i
(5)	t_3	=	$t_2[t_1]$
(6)	t_4	=	t_1
(8)	t_6	=	$t_5[t_4]$
(9)	t_7	=	t_3 * t_6
(10)	sum	=	sum + t_7
(11)	i	=	i + 1
(12)	**if** i ≤ 20 **goto** B_2		

Induction variable: t_1 and i remain in lock-step

An Example: Induction Variables and Reduction in Strength

B_1

(1) sum = 0
(2) i = 1
(4) t_2 = addr(A) - 4
(7) t_5 = addr(B) - 4
(3) t_1 = 4 * i

B_2

(5) t_3 = $t_2[t_1]$
(6) t_4 = t_1
(8) t_6 = $t_5[t_4]$
(9) t_7 = t_3 * t_6
(10) sum = sum + t_7
(11) i = i + 1
(3') t_1 = t_1 + 4
(12) **if** $i \leq 20$ **goto** B_2

Loop condition can be changed

An Example: Loop Condition Transformation

B_1

(1)	sum	=	0
(2)	i	=	1
(4)	t_2	=	addr(A) – 4
(7)	t_5	=	addr(B) – 4
(3)	t_1	=	4 * i

i is known as 1

B_2

(5)	t_3	=	$t_2[t_1]$
(6)	t_4	=	t_1
(8)	t_6	=	$t_5[t_4]$
(9)	t_7	=	$t_3 * t_6$
(10)	sum	=	sum + t_7
(11)	i	=	i + 1
(3')	t_1	=	$t_1 + 4$
(12)	**if** $t_1 \le 80$ **goto** B_2		

t_4 is known as t_1

i can be removed if not live on exit

An Example:
Constant and Copy Propagation

B_1

(1) $sum = 0$
(2) $i = 1$
(4) $t_2 = addr(A) - 4$
(7) $t_5 = addr(B) - 4$
(3) $t_1 = 4$

Suppose i is not live on exit

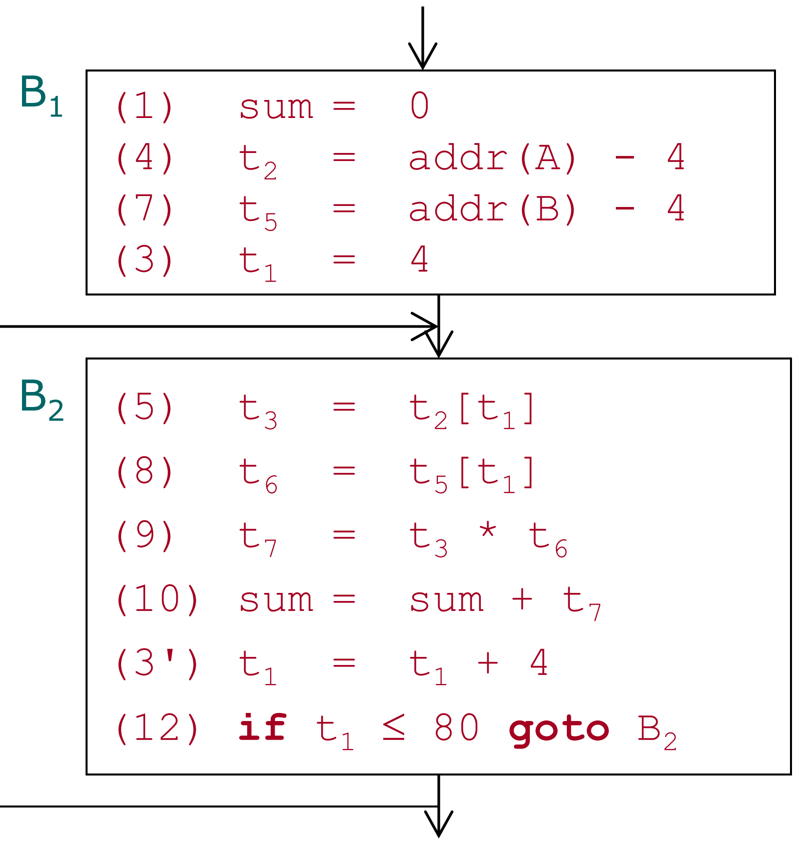
B_2

(5) $t_3 = t_2[t_1]$
(6) $t_4 = t_1$
(8) $t_6 = t_5[t_1]$
(9) $t_7 = t_3 * t_6$
(10) $sum = sum + t_7$
(11) $i = i + 1$
(3') $t_1 = t_1 + 4$
(12) **if** $t_1 \leq 80$ **goto** B_2

Suppose t_4 is not live on exit

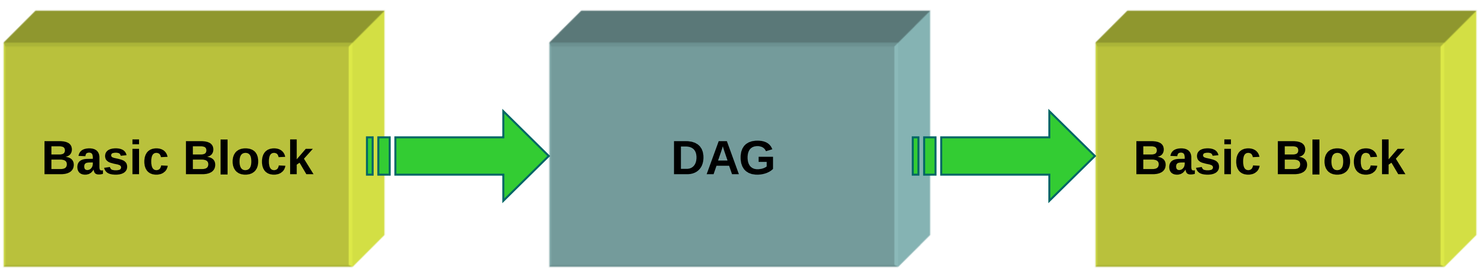
Suppose i is not live on exit

An Example:
Eliminating Redundant Operations



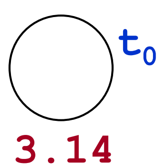
2. Local Optimization

- Transformations
  - Common subexpressions
  - Constant and copy propagation
  - Eliminating redundant operations



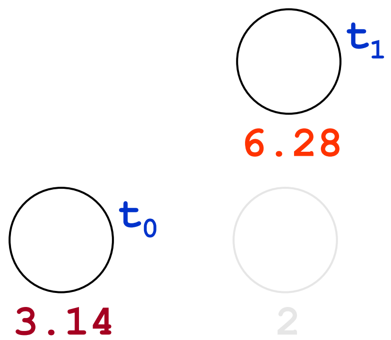
One More Example (1)

(1) $t_0 = 3.14$
(2) $t_1 = 2 * t_0$
(3) $t_2 = R + r$
(4) $A = t_1 * t_2$
(5) $B = A$
(6) $t_3 = 2 * t_0$
(7) $t_4 = R + r$
(8) $t_5 = t_3 * t_4$
(9) $t_6 = R - r$
(10) $B = t_5 * t_6$



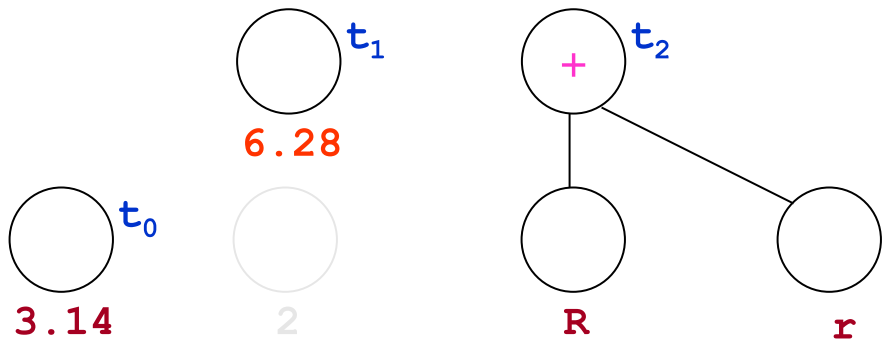
One More Example (2)

(1) $t_0 = 3.14$
(2) $t_1 = 2 * t_0$
(3) $t_2 = R + r$
(4) $A = t_1 * t_2$
(5) $B = A$
(6) $t_3 = 2 * t_0$
(7) $t_4 = R + r$
(8) $t_5 = t_3 * t_4$
(9) $t_6 = R - r$
(10) $B = t_5 * t_6$



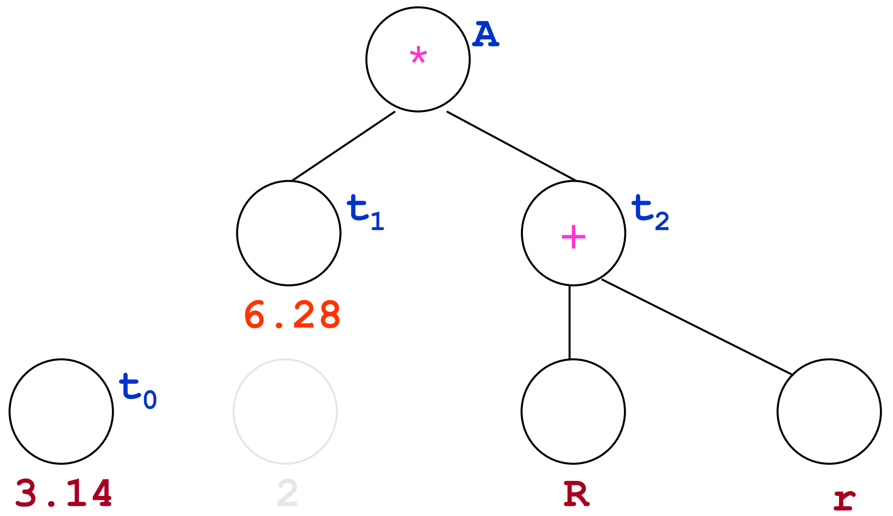
One More Example (3)

```
(1)  t0 = 3.14
(2)  t1 = 2 * t0
(3)  t2 = R + r
(4)  A  = t1 * t2
(5)  B  = A
(6)  t3 = 2 * t0
(7)  t4 = R + r
(8)  t5 = t3 * t4
(9)  t6 = R - r
(10) B  = t5 * t6
```



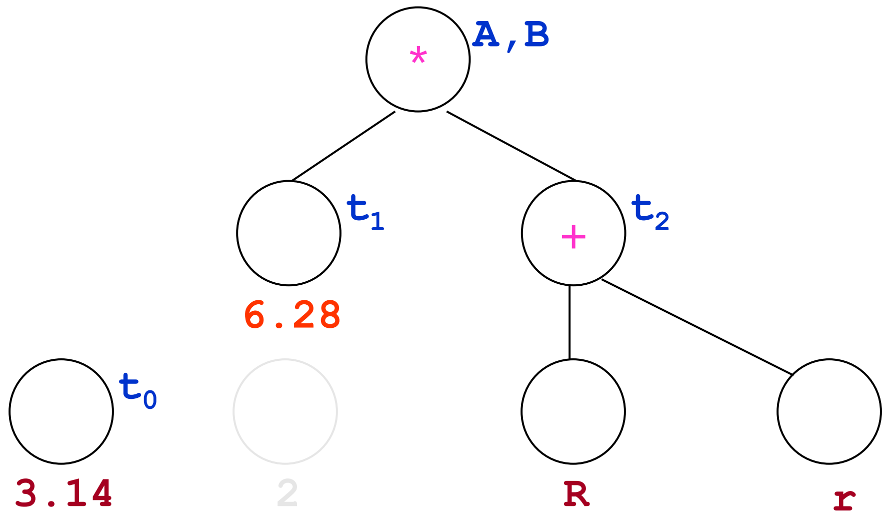
One More Example (4)

```
(1)   t0  =  3.14
(2)   t1  =  2 * t0
(3)   t2  =  R + r
(4)   A   =  t1 * t2
(5)   B   =  A
(6)   t3  =  2 * t0
(7)   t4  =  R + r
(8)   t5  =  t3 * t4
(9)   t6  =  R - r
(10)  B   =  t5 * t6
```



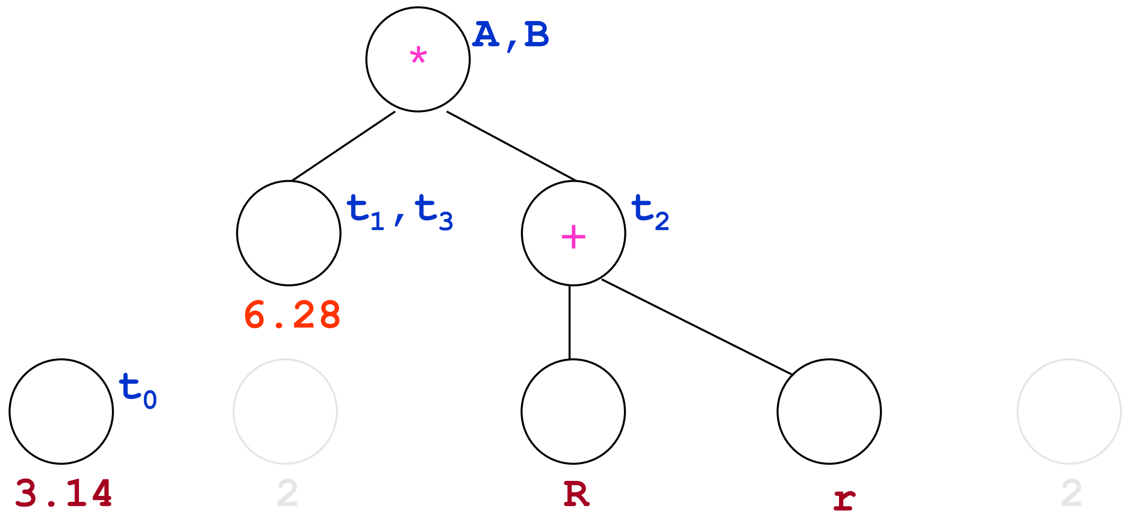
One More Example (5)

```
(1)   t0  =  3.14
(2)   t1  =  2 * t0
(3)   t2  =  R + r
(4)   A   =  t1 * t2
(5)   B   =  A
(6)   t3  =  2 * t0
(7)   t4  =  R + r
(8)   t5  =  t3 * t4
(9)   t6  =  R - r
(10)  B   =  t5 * t6
```



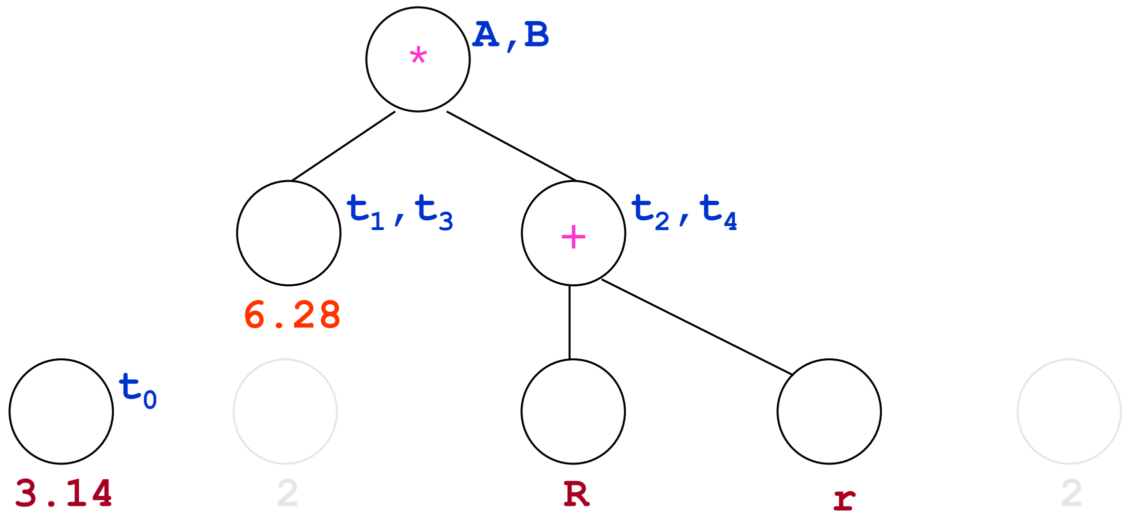
One More Example (6)

```
(1)  t0 = 3.14
(2)  t1 = 2 * t0
(3)  t2 = R + r
(4)  A  = t1 * t2
(5)  B  = A
(6)  t3 = 2 * t0
(7)  t4 = R + r
(8)  t5 = t3 * t4
(9)  t6 = R - r
(10) B  = t5 * t6
```



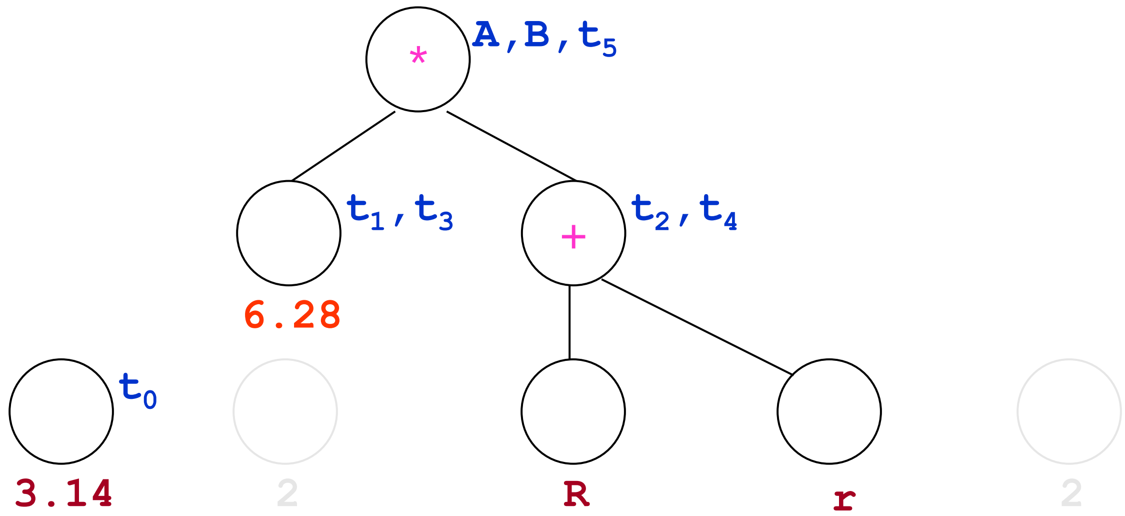
One More Example (7)

```
(1)   t0  =  3.14
(2)   t1  =  2 * t0
(3)   t2  =  R + r
(4)   A   =  t1 * t2
(5)   B   =  A
(6)   t3  =  2 * t0
(7)   t4  =  R + r
(8)   t5  =  t3 * t4
(9)   t6  =  R - r
(10)  B   =  t5 * t6
```



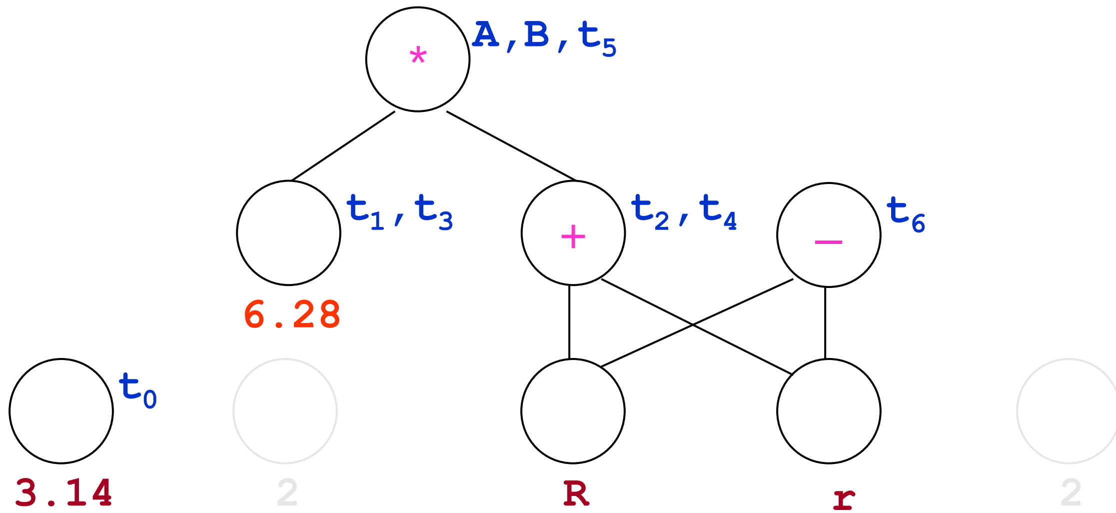
One More Example (8)

```
(1)  t0 = 3.14
(2)  t1 = 2 * t0
(3)  t2 = R + r
(4)  A  = t1 * t2
(5)  B  = A
(6)  t3 = 2 * t0
(7)  t4 = R + r
(8)  t5 = t3 * t4
(9)  t6 = R - r
(10) B  = t5 * t6
```



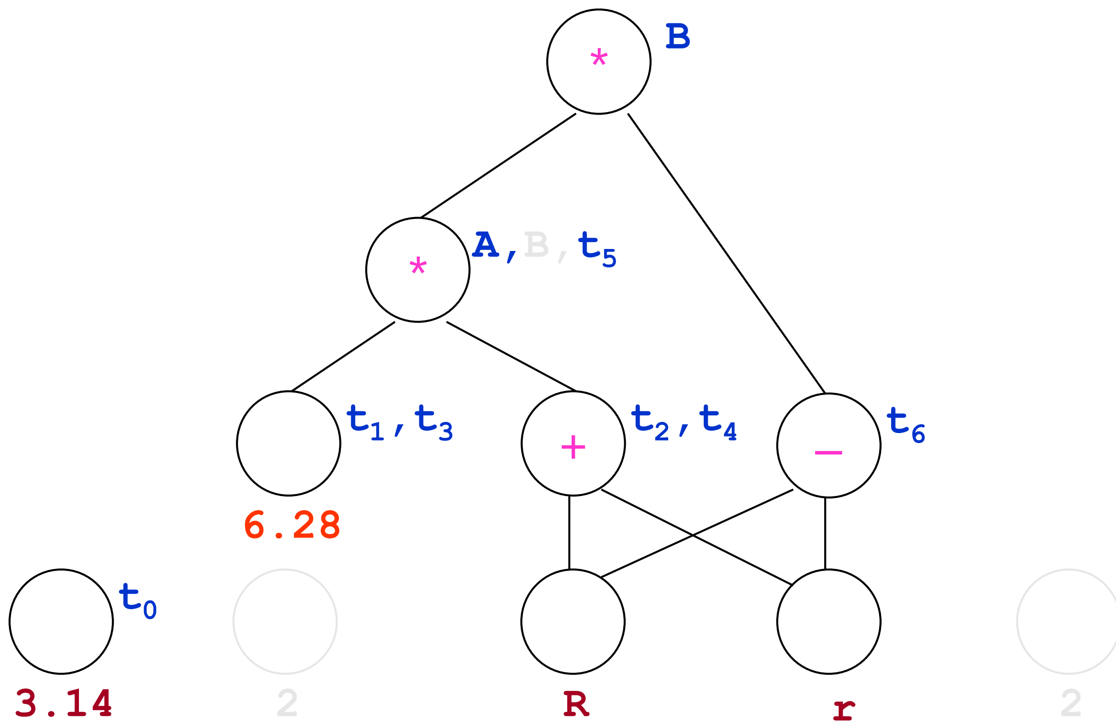
One More Example (9)

```
(1)   t0 = 3.14
(2)   t1 = 2 * t0
(3)   t2 = R + r
(4)   A  = t1 * t2
(5)   B  = A
(6)   t3 = 2 * t0
(7)   t4 = R + r
(8)   t5 = t3 * t4
(9)   t6 = R - r
(10)  B  = t5 * t6
```

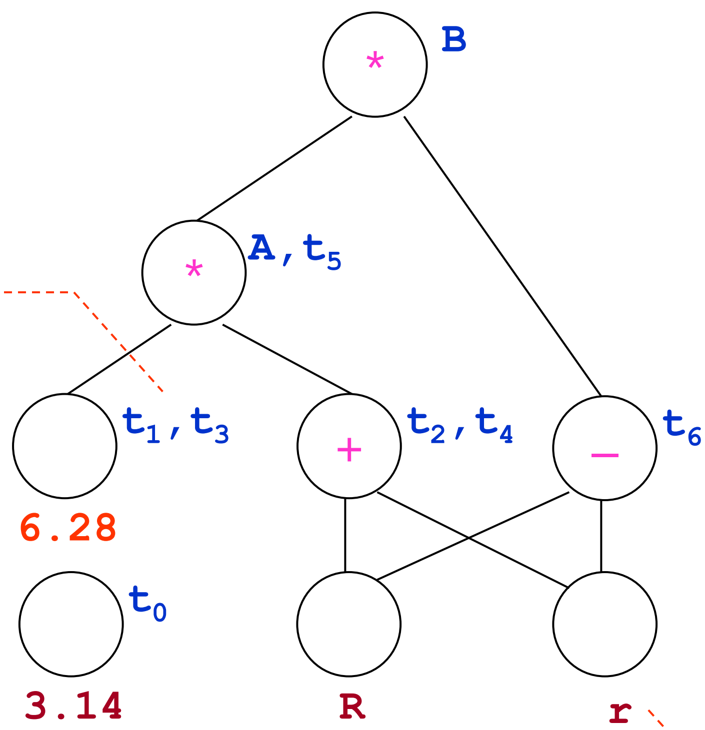


One More Example (10)

(1) $t_0 = 3.14$
(2) $t_1 = 2 * t_0$
(3) $t_2 = R + r$
(4) $A = t_1 * t_2$
(5) $B = A$
(6) $t_3 = 2 * t_0$
(7) $t_4 = R + r$
(8) $t_5 = t_3 * t_4$
(9) $t_6 = R - r$
(10) $B = t_5 * t_6$



One More Example (end)



A label indicates a value available to other blocks

A leaf indicates a value from outside the block

```
(1)  t0 = 3.14
(2)  t1 = 6.28
(3)  t3 = 6.28
(4)  t2 = R + r
(5)  t4 = t2
(6)  A  = 6.28 * t2
(7)  t5 = A
(8)  t6 = R - r
(9)  B  = A * t6
```

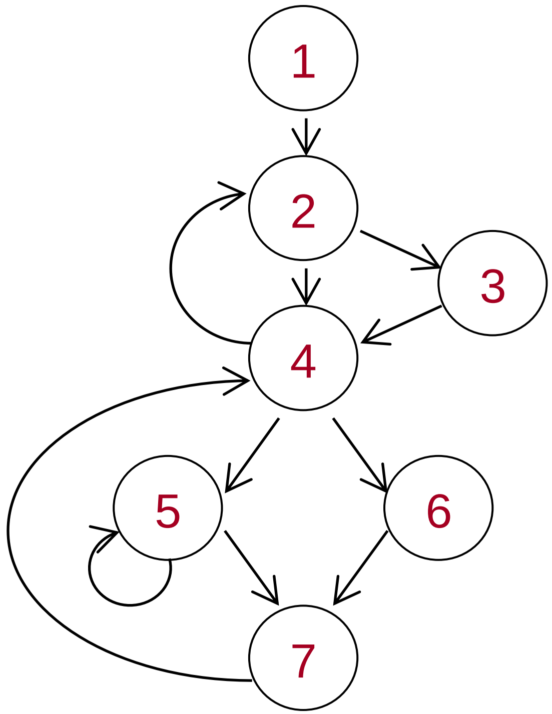
3. Control-Flow Analysis and Loop Optimization

- Three important subproblems
  - How to define a loop based on a flow graph ?
  - How to find a loop in a flow graph ?
  - How to optimize a loop ?

Define Loops Based on Flow Graphs

- A loop is a strongly connected subgraph with a unique entry (header).
  - Properties:
    - Strongly connected
    - Unique entry (destination of code motion)
    - A loop can be expressed as a sequence of nodes.

An Example



There are 3 loops:

{5}

{4, 5, 6, 7}

{2, 3, 4, 5, 6, 7}

They are NOT loops:

{2, 4}	Both 2 and 4 are entries
{2, 3, 4}	Both 2 and 4 are entries
{4, 5, 7}	Both 4 and 7 are entries
{4, 6, 7}	Both 4 and 7 are entries

Dominators

- Notations
  - m **DOM** n means m is a dominator of n.
  - D(n) is the set of all dominators of n.
    - D(n) = {m | m **DOM** n}
- Properties
  - The entry is a dominator of all nodes in the loop.
  - The binary relation **DOM** is a partial order
    - Reflective, transitive, and antisymmetric.

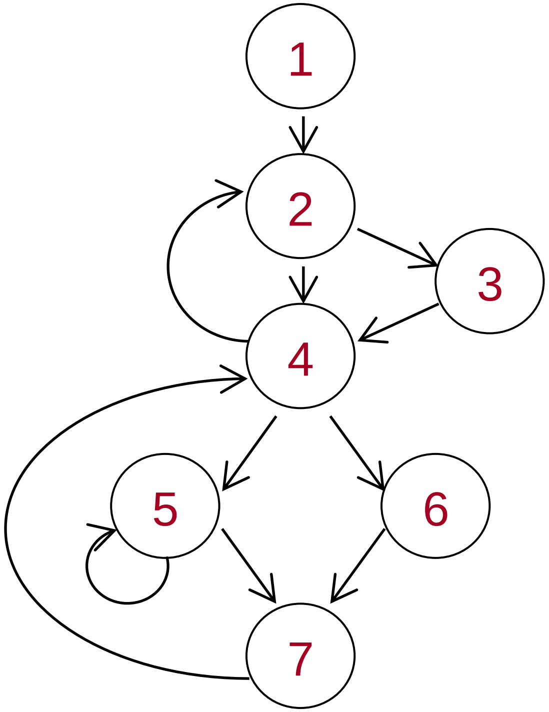
Algorithm to Calculate **D(n)**

- Input: flow graph $G = (N, E, n_0)$
  - N = set of nodes; E = set of edges; n_0 = entry.
- Algorithm

```
D(n_0) = {n_0};
foreach (n ∈ N - {n_0})  D(n) = N;
changed = true;
while (changed) {
    changed = false;
    foreach (n ∈ N - {n_0}) {
      newd = {n} ∪ (∩_{p∈PRE(n)} D(p));
      if (D(n) ≠ newd) {
        D(n) = newd; changed = true;
      }
    }
}
```

PRE = predecessor

The Previous Example

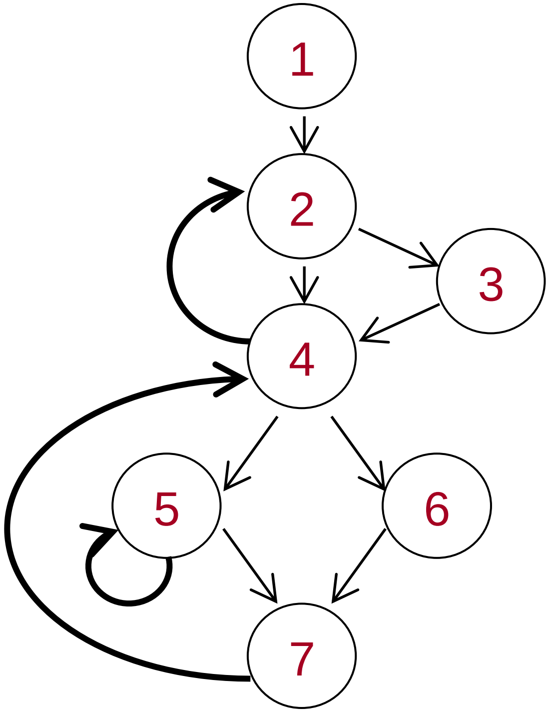


D(1) = {1}
D(2) = {1, 2}
D(3) = {1, 2, 3}
D(4) = {1, 2, 4}
D(5) = {1, 2, 4, 5}
D(6) = {1, 2, 4, 6}
D(7) = {1, 2, 4, 7}

Back Edges and Natural Loops

- Back edge
  - $a \rightarrow b$ is a back edge if $a \rightarrow b \in E \ \wedge \ b$ **DOM** a.
- Natural loop
  - A natural loop defined by a back edge $a \rightarrow b$ $= \{b\} \cup \{$nodes that can reach a without going through $b\}$

The Previous Example



There are 3 back edges:

- **5 → 5**
- **7 → 4**
- **4 → 2**

There are 3 natural loops:

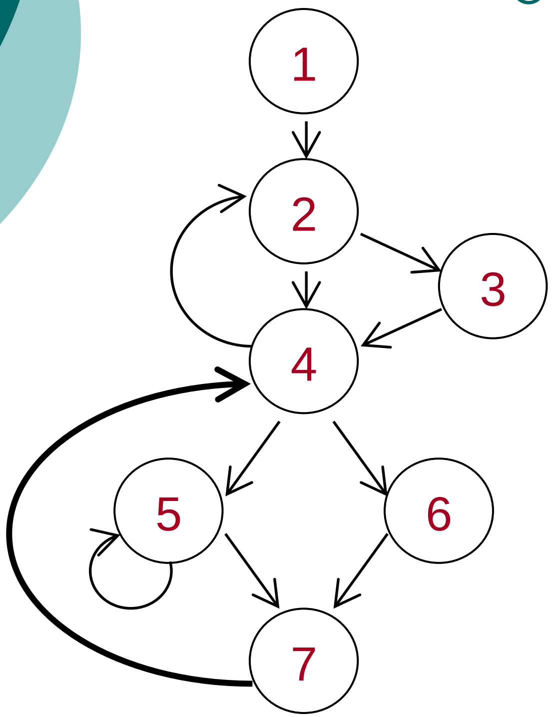
- **{5}** defined by **5 → 5**
- **{4, 5, 6, 7}** defined by **7 → 4**
- **{2, 3, 4, 5, 6, 7}** defined by **4 → 2**

Find Loops by Back Edges

- Input: back edge $n \rightarrow d$.
- Algorithm:

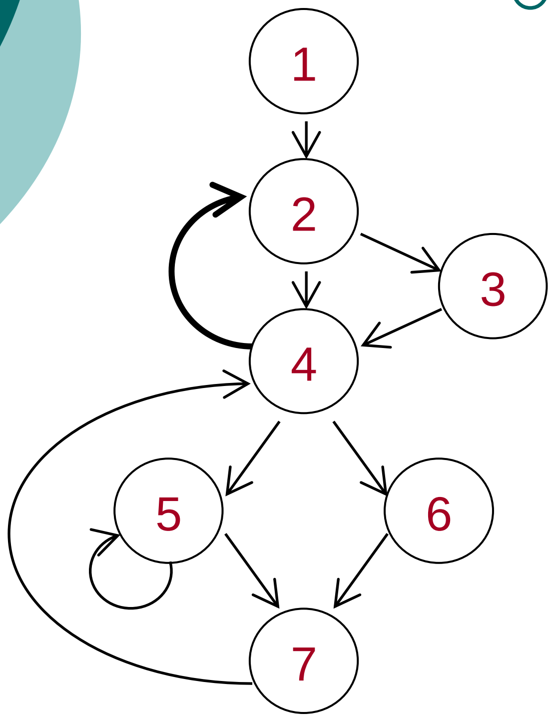
```
void insert(Node m) {
   if (m ∉ loop) {   // d will not be pushed
      loop = loop ∪ {m};
      stack.push(m)
   }
}
void main() {
   Stack stack = new Stack();
   loop = {d};   // PRE(d) will not be added
   insert(n);
   while (stack.notEmpty()) {
      m = stack.pop();
      foreach (p ∈ PRE(m))  insert(p)
   }
}
```

Example #1



- Given the back edge 7 → 4
  1. initialize: loop = {4, 7}, stack = [7].
  2. pop 7; insert 5 and 6;
     loop = {4, 7, 5, 6}, stack = [5, 6].
  3. pop 6; insert 4 (already in loop);
     loop has no change, stack = [5].
  4. pop 5; insert 4 and 5 (both in loop);
     loop has no change, stack = [].
  5. result: loop = {4, 7, 5, 6}.

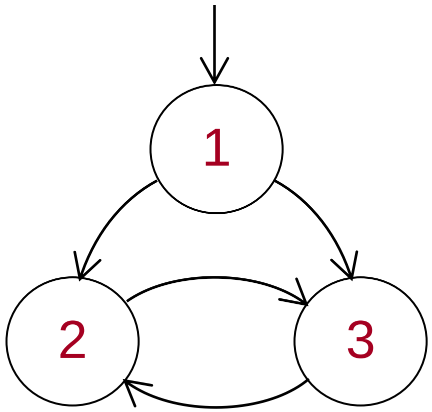
Example #2



- Given the back edge $4 \rightarrow 2$
  1. initialize: loop = {2, 4}, stack = [4].
  2. pop 4; insert 2, 3 and 7 (2 already in loop); loop = {2, 4, 3, 7}, stack = [3, 7].
  3. pop 7; insert 5 and 6; loop = {2, 4, 3, 7, 5, 6}, stack = [3, 5, 6].
  4. pop 6; insert 4 (already in loop); loop had no change, stack = [3, 5].
  5. pop 5; insert 4 and 5 (both in loop); loop has no change, stack = [3].
  6. pop 3; insert 2 (already in loop); loop has no change, stack = [].
  7. result: loop = {2, 4, 3, 7, 5, 6}.

Properties of Natural Loops

- Natural loops do not cover all of loops in common sense
  - E.g. there is no back edge in the following flow graph, but it does have a loop in common sense: {2, 3}.
  - Only in a reducible flow graph, can the back edges find all loops.



- Reducible flow graph
  - After removing all back edges, the subgraph is acyclic.
  - In a reducible flow graph, the only entry to a loop is the header.
  - A flow graph generated from a structured program is commonly reducible.

Loop Optimization: Code Motion

- Target of code motion
  - Following the header of the loop.
- What code can be moved ?

  For an instruction x = y **op** z,

  1. It is a loop-invariant operation.
     - All possible definitions of y and z are outside the loop, including constants, or (recursively)
     - Defined by loop-invariant values.
  2. No other statement in the loop defines x.
  3. All uses of x in the loop are defined by it.

Loop Optimization: Reducing Strength and Eliminating Induction Variables

- Basic induction variable
  - $i = i \pm C$
    - Unique assignment to i in the loop.
    - C is loop-invariant.
- Family of induction variables
  - $j = C_1 * i \pm C_2$
    - Both C_1 and C_2 are loop-invariant.
- Motivation
  - Substitute i with some j in the family.
    - The multiplication of j can be removed.
    - Then i can be eliminated.
  - Specially effective to indexing variables.

An Example:
Family of Induction Variables

Only one loop:

{ $\mathbf{B_2}$ }

Basic induction variable:

i

Family of induction variables:

$\mathbf{t_1} = (i, 4, 0) = 4 * i + 0$

$\mathbf{t_4} = (i, 4, 0) = 4 * i + 0$

B_1

```
(1)  sum =  0
(2)  i   =  1
```

B_2

```
(3)  t1  =  4 * i
(4)  t2  =  addr(A) - 4
(5)  t3  =  t2[t1]
(6)  t4  =  4 * i
(7)  t5  =  addr(B) - 4
(8)  t6  =  t5[t4]
(9)  t7  =  t3 * t6
(10) sum =  sum + t7
(11) i   =  i + 1
(12) if i ≤ 20 goto B2
```

An Example: Strength Reduction (1)

Create a new variable **j'** (e.g. **t_1'** and **t_4'**) for each induction variable in family **$j = C_1 * i \pm C_2$** (e.g. **t_1** and **t_4**).

Initialize new variables at the end of the preheader:

$j' = C_1 * i$

$j' = j' + C_2$ // only if $C_2 \neq 0$

B_1

```
(1)  sum = 0
(2)  i   = 1
(2a) t1' = 4 * i
(2b) t4' = 4 * i
```

B_2

```
(3)  t1  = 4 * i
(4)  t2  = addr(A) - 4
(5)  t3  = t2[t1]
(6)  t4  = 4 * i
(7)  t5  = addr(B) - 4
(8)  t6  = t5[t4]
(9)  t7  = t3 * t6
(10) sum = sum + t7
(11) i   = i + 1
(12) if i ≤ 20 goto B2
```

An Example: Strength Reduction (2)

Change the definition of each induction variable (e.g. $\mathbf{t_1}$ and $\mathbf{t_4}$):

j = j'

B_1

(1) sum = 0
(2) i = 1
(2a) $t_1' = 4 * i$
(2b) $t_4' = 4 * i$

B_2

(3) $t_1 = t_1'$
(4) $t_2 = addr(A) - 4$
(5) $t_3 = t_2[t_1]$
(6) $t_4 = t_4'$
(7) $t_5 = addr(B) - 4$
(8) $t_6 = t_5[t_4]$
(9) $t_7 = t_3 * t_6$
(10) sum = sum + t_7
(11) i = i + 1
(12) **if** $i \le 20$ **goto** B_2

An Example: Strength Reduction (3)

Add linear assignments to new variables following the unique definition of basic statement variable ($i = i \pm C$):

$$\mathbf{t = C_1 * C}$$

$$\mathbf{j' = j' \pm t}$$

If C == ±1, only one statement need to be added:

$$\mathbf{j' = j' \pm C_1}$$

B_1

(1) $sum = 0$
(2) $i = 1$
(2a) $t_1' = 4 * i$
(2b) $t_4' = 4 * i$

B_2

(3) $t_1 = t_1'$
(4) $t_2 = addr(A) - 4$
(5) $t_3 = t_2[t_1]$
(6) $t_4 = t_4'$
(7) $t_5 = addr(B) - 4$
(8) $t_6 = t_5[t_4]$
(9) $t_7 = t_3 * t_6$
(10) $sum = sum + t_7$
(11) $i = i + 1$
(11a) $t_1' = t_1' + 4$
(11b) $t_4' = t_4' + 4$
(12) **if** $i \leq 20$ **goto** B_2

An Example: Eliminate Dead Induction Variables

If induction variable j is not live on exit,

change the use of j to j' (e.g. change reference from $\mathbf{t_1}$ and $\mathbf{t_4}$ to $\mathbf{t_1}$' and $\mathbf{t_4}$'),

and then remove the definition of j (e.g. $\mathbf{t_1}$ and $\mathbf{t_4}$).

B_1

```
(1)   sum =  0
(2)   i   =  1
(2a)  t1' = 4 * i
(2b)  t4' = 4 * i
```

B_2

```
(3)   t1  =  t1'
(4)   t2  =  addr(A) - 4
(5)   t3  =  t2[t1']
(6)   t4  =  t4'
(7)   t5  =  addr(B) - 4
(8)   t6  =  t5[t4']
(9)   t7  =  t3 * t6
(10)  sum =  sum + t7
(11)  i   =  i + 1
(11a) t1' =  t1' + 4
(11b) t4' =  t4' + 4
(12)  if i ≤ 20 goto B2
```

An Example:
Change Loop Condition

B_1

```
(1)   sum =  0
(2)   i   =  1
(2a)  t1' = 4 * i
(2b)  t4' = 4 * i
```

B_2

```
(3)   t1  =  t1'
(4)   t2  =  addr(A) - 4
(5)   t3  =  t2[t1']
(6)   t4  =  t4'
(7)   t5  =  addr(B) - 4
(8)   t6  =  t5[t4']
(9)   t7  =  t3 * t6
(10)  sum =  sum + t7
(11)  i   =  i + 1
(11a) t1' =  t1' + 4
(11b) t4' =  t4' + 4
(12)  if i ≤ 20 goto B2
(12a) R = 4 * 20
(12b) if t1' ≤ R goto B2
```

Pick a new induction variable from the family (say **t_1'**), then change the loop condition.

An Example:
Remove Basic Induction Variable

If the basic induction variable i is not live on exit, the definition of i can be removed.

B1

```
(1)   sum =  0
(2)   i   =  1
(2a)  t1' = 4 * i
(2b)  t4' = 4 * i
```

B2

```
(3)   t1  =  t1'
(4)   t2  =  addr(A) - 4
(5)   t3  =  t2[t1']
(6)   t4  =  t4'
(7)   t5  =  addr(B) - 4
(8)   t6  =  t5[t4']
(9)   t7  =  t3 * t6
(10)  sum =  sum + t7
(11)  i   =  i + 1
(11a) t1' =  t1' + 4
(11b) t4' =  t4' + 4
(12)  if i ≤ 20 goto B2
(12a) R = 4 * 20
(12b) if t1' ≤ R goto B2
```

An Example: After Loop Optimization

The optimized code facilitate further local optimization.

B_1

```
(1)   sum =  0
(2)   i   =  1
(2a)  t1' = 4 * i
(2b)  t4' = 4 * i
```

B_2

```
(4)   t2  =  addr(A) - 4
(5)   t3  =  t2[t1']
(7)   t5  =  addr(B) - 4
(8)   t6  =  t5[t4']
(9)   t7  =  t3 * t6
(10)  sum =  sum + t7
(11a) t1' =  t1' + 4
(11b) t4' =  t4' + 4
(12a) R = 4 * 20
(12b) if t1' ≤ R goto B2
```

4. Data-Flow Analysis and Global Optimization

- Collect information about data flows
  - How a variable is assigned (**definition**) ?
  - How a variable is referred (**use**) ?
- Control-flow vs. data-flow
  - Control-flow analysis: basic blocks are considered as **black** boxes.
  - Data-flow analysis: basic blocks are considered as **white** boxes.

Where Global Information Are Needed ?

- Local optimization
  - Assignments to a variable can be removed if the variable is never used.
- Loop optimization: code motion
  - Determine loop-invariant operations according to the definitions of variables.
  - Code motion requires the operation is the unique definition in the loop.
  - Code motion also requires the defined variable is not live on the exit of the loop.
- Loop optimization: induction variable elimination
  - induction variables can be removed if it is not used outside the loop.
- Code generation
  - Information on liveness on exit facilitate register utilization.

What Global Information Are Needed ?

- Definition
  - All assignments (sources) of a R-value in a statement.
- Use
  - All possible use of an L-value in a statement.
- Liveness
  - Will the variable be referred as a R-value after a statement.

Basic Concepts

- Points in a flow graph
  - Between two adjacent statements.
  - Before the first and after the last statement.
- Definition of a variable **x**
  - A statement that (may) assign(s) a value to **x**.
  - L-value.
- Use of a variable **x**
  - A statement that refers **x** as an operand.
  - R-value.

between statements

statement

statement

Basic Concepts (cont')

- Definition **d** reaches a point **p**
  - There exists a path from the point immediately following **d** to **p**, such that **d** is not "killed" along the path.
  - While we use **x** immediately following **p**, the value of **x** may be determined by **d**.

Ud-Chains vs. Du-Chains

- Ud-chain: the use-definition chain of a variable **x** in a use statement **s**
  - Set of definitions of **x** that can reach **s**.
  - Useful for finding loop-invariants.
  - Also for global constant folding.
- Du-chain: the definition-use chain of a variable **x** in a definition statement **s**
  - Set of uses of **x** that can be reached from **s**.
  - Useful for eliminating induction variables in loop optimization.
  - Also for finding family of induction variables.

Reaching Definition Analysis

- Forward data-flow equation

  $$\texttt{out[B]} = (\texttt{in[B]} - \texttt{kill[B]}) \cup \texttt{gen[B]}$$

  $$\texttt{in[B]} = \bigcup_{p \in \texttt{PRE(B)}} \texttt{out[p]}$$

  - in[B]: ud-chain before the entry of B.
  - out[B]: ud-chain after the exit of B.
  - gen[B]: all definitions in B that can reach the exit of B.
  - kill[B]: all definitions outside B that are killed by B.

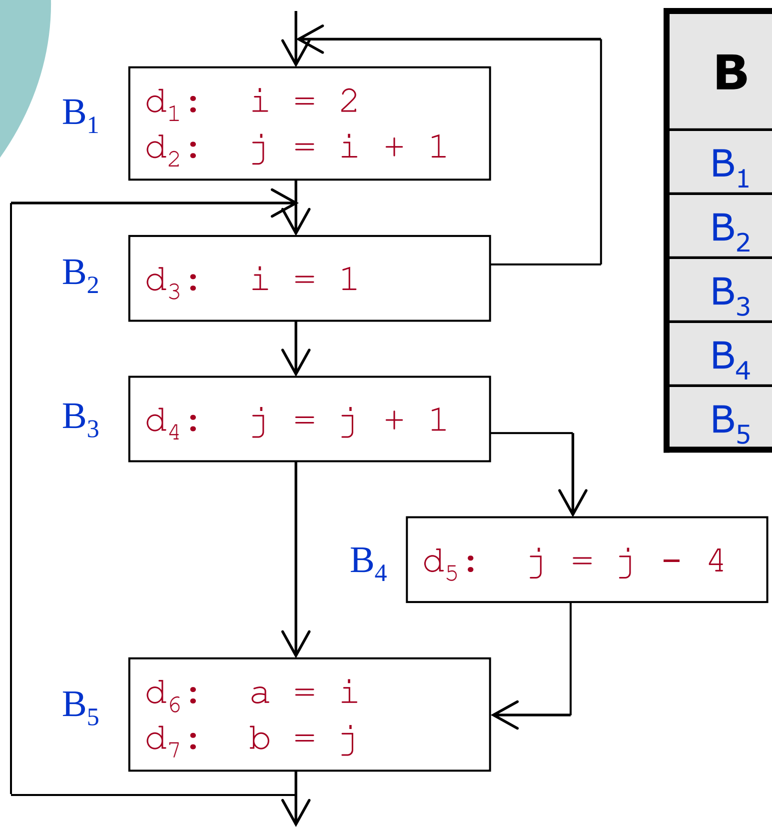
Construction of Ud-Chains

- **Input**: gen[] and kill[]; **Output**: in[] and out[].
- **Algorithm**

```
for (int i = 1; i <= n; i++) {     // initialize
    in[Bi] = ∅;  out[Bi] = gen[Bi];
}
changed = true;
while (changed) {     // iterative
    changed = false;
    for (i = 1; i <= n; i++) {
      newIn = ∪p∈PRE[Bi] out[p];
      if (newIn ≠ in[Bi]) {
          changed = true;
          in[Bi] = newIn;
          out[Bi] = (in[Bi] - kill[Bi]) ∪ gen[Bi];
      }
    }
}
```

An Example:
(1) gen[ ] and kill[ ] is known

- Only variable **i** and **j** are considered



B	gen[B]		kill[B]	
	Set	Vector	Set	Vector
B_1	$\{d_1, d_2\}$	1100000	$\{d_3, d_4, d_5\}$	0011100
B_2	$\{d_3\}$	0010000	$\{d_1\}$	1000000
B_3	$\{d_4\}$	0001000	$\{d_2, d_5\}$	0100100
B_4	$\{d_5\}$	0000100	$\{d_2, d_4\}$	0101000
B_5	$\varnothing$	0000000	$\varnothing$	0000000

An Example:
(2) Iterations of in[ ] and out[ ]

- Depth-first visit: B_1, B_2, B_3, B_4 and B_5

B	**Init**		**1^{st}**		**2^{nd}**		**3^{rd}**		**4^{th}**	
	in[B]	out[B]	in[B]	out[B]	in[B]	out[B]	in[B]	out[B]	in[B]	out[B]
B_1	0000000	1100000	**0010000**	**1100000**	**0110000**	**1100000**	**0111100**	**1100000**	0111100	1100000
B_2	0000000	0010000	**1100000**	**0110000**	**1111100**	**0111100**	1111100	0111100	1111100	0111100
B_3	0000000	0001000	**0110000**	**0011000**	**0111100**	**0011000**	0111100	0011000	0111100	0011000
B_4	0000000	0000100	**0011000**	**0010100**	0011000	0010100	0011000	0010100	0011000	0010100
B_5	0000000	0000000	**0011100**	**0011100**	0011100	0011100	0011100	0011100	0011100	0011100

An Example:
(3) Construction of Ud-Chains

- Compute ud-chains with in[B].
  - If **s.x** has definitions before **s** in B, ud-chain of **s.x** is a singleton (definition nearest to **s**).
  - Otherwise, ud-chain of **s.x** is all definitions of **x** in in[B].
- Result ud-chains
  - Variable i at definition d_2: $\{d_1\}$
  - Variable j at definition d_4: $\{d_2, d_4, d_5\}$
  - Variable j at definition d_5: $\{d_4\}$
  - Variable i at definition d_6: $\{d_3\}$
  - Variable j at definition d_7: $\{d_4, d_5\}$

d_3 is the definition of **i**, not **j** !

d_4 and d_5 are the definitions of **j**, not **i** !

Global Constant Propagation and Folding Based on Ud-Chains

```
changed = true;
while (changed) {
   changed = false;
   foreach (statement [S: x = ...]) {
     foreach (operand S.y) {  // constant propagation
       if (S.y.ud-chain has only one i and i is [y = CONST]) {
          replace all S.y with CONST;
          changed = true;
       }
     }
     if (S has op and each operand is CONST) { // folding
       let C = result of constant operation;
       replace S with [x = C];
       changed = true;
     }
   }
}
```

More Data-Flow Equations: Available Expressions

- **Forward** data-flow equation

  `out[B] = (in[B] - E_kill[B])` $\cup$ `E_gen[B]`

  `in[B] = iif(B == ENTRY,` $\varnothing$`,` $\bigcap_{p \in PRE(B)}$ `out[p])`

  - in[B]: available expressions before B.
  - out[B]: available expressions after B.
  - E_gen[B]: expressions generated by B.
  - E_kill[B]: expressions killed by B.

- Motivation
  - Available expression E = X op Y at **s** is the last evaluation of **E** from entry point to **s**, and no redefinition of **X** and **Y** after the definition of **E**.
  - Useful: global common expression elimination.

More Data-Flow Equations: Liveness Analysis

- **Backward** data-flow equation

  $$\texttt{in[B]} = (\texttt{out[B]} - \texttt{def[B]}) \cup \texttt{use[B]}$$

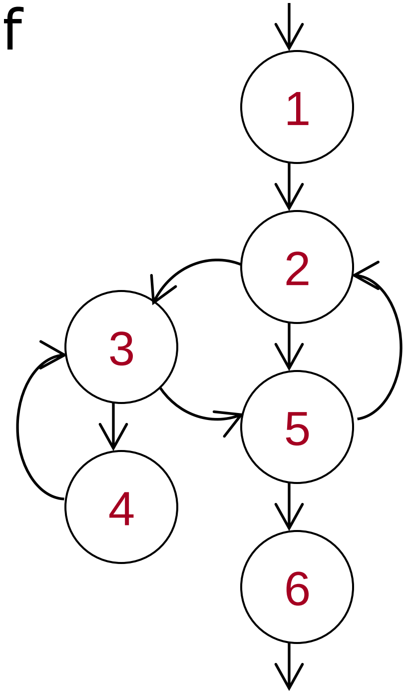
  $$\texttt{out[B]} = \bigcup_{s \in \texttt{SUCC(B)}} \texttt{in[s]}$$

  - in[B]: live variables before B.
  - out[B]: live variables after B.
  - use[B]: live variables generated by B.
  - def[B]: live variables killed by B.

SUCC = successor

Exercise 12.1

- Given the following flow graph:
  - Compute the dominators of all nodes.
  - Find all back edges in the flow graph.
  - Find all natural loops defined by each back edge.



Enjoy the Course!