

# Local Optimizations

Lecture 9

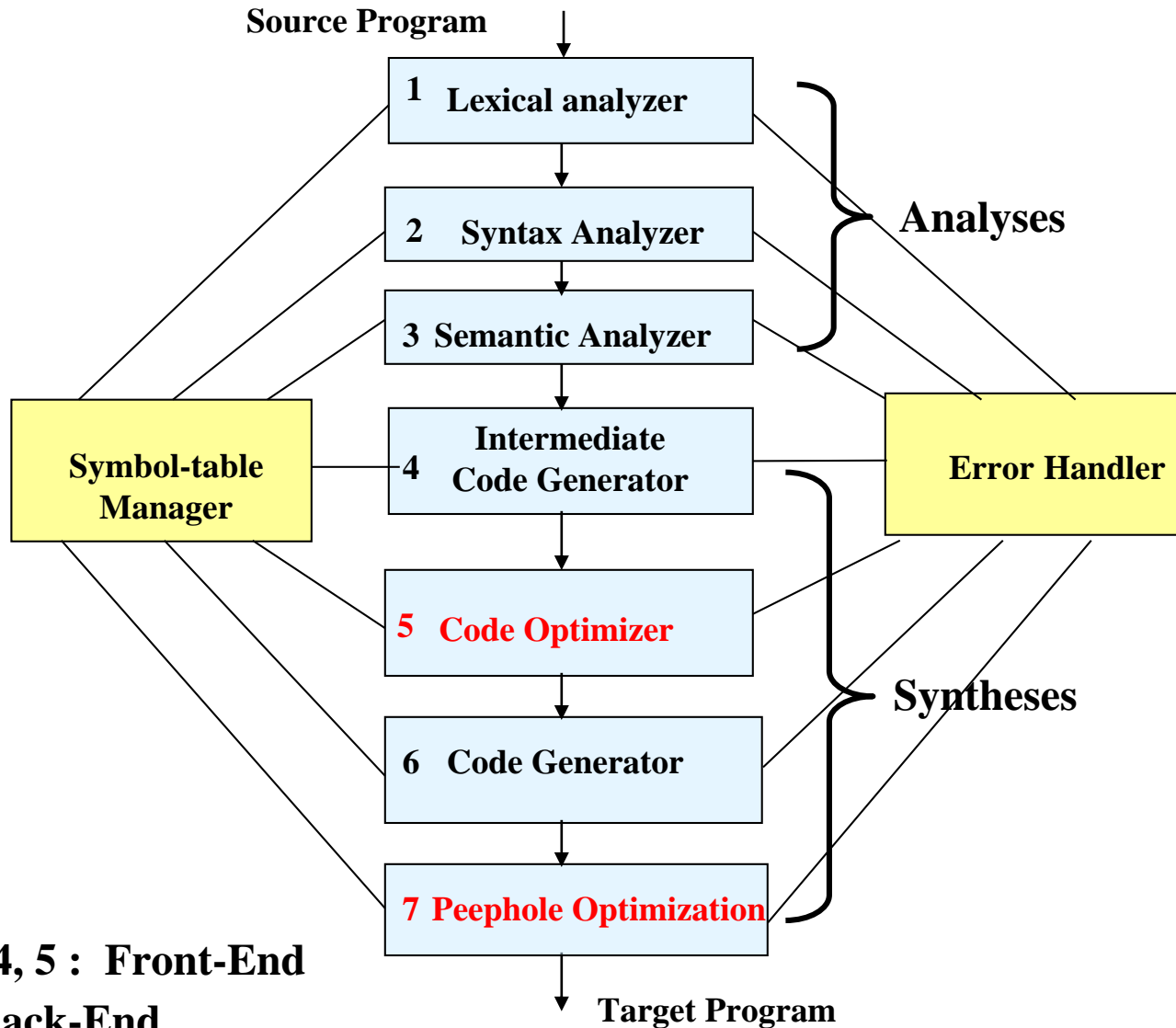
Part 1

# Announcements

---

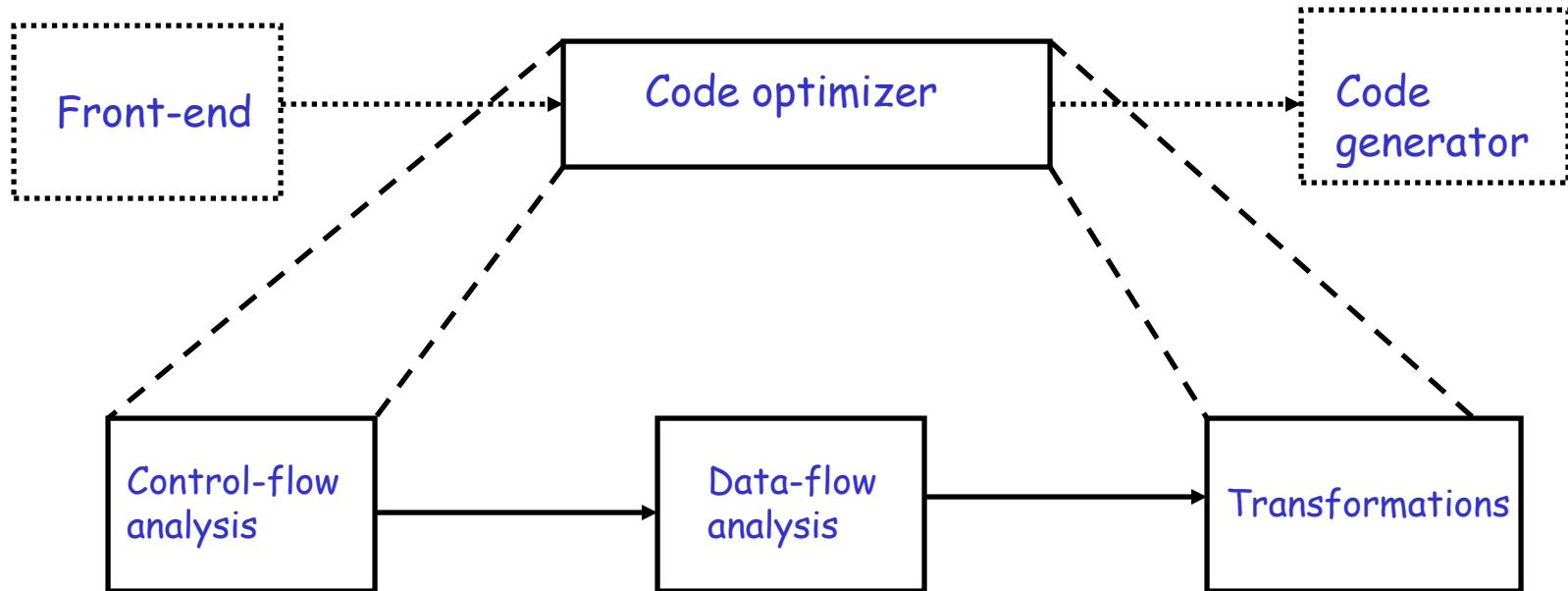
- WA 3
  - Due today (22 July) at 11:59pm
  
- WA 4
  - Assigned tomorrow
  
- PA 4
  - Assigned tomorrow

# Compiler Front-end and Back-end (Revisited)



# Organization of a Code Optimizer

---



# Optimization

---

- Most complexity in modern compilers is in the optimizer
  - Also by far the largest phase
- First, we need to discuss intermediate languages

# Why Intermediate Languages?

---

- When should we perform optimizations?
  - On AST
    - **Pro**: Machine independent
    - **Con**: Too high level
  - On assembly language
    - **Pro**: Exposes optimization opportunities
    - **Con**: Machine dependent
    - **Con**: Must reimplement optimizations when retargetting
  - On an intermediate language
    - **Pro**: Machine independent
    - **Pro**: Exposes optimization opportunities

# Intermediate Languages

---

- Intermediate language = high-level assembly
  - Uses register names, but has an unlimited number
  - Uses control structures like assembly language
  - Uses opcodes but some are higher level
    - Most opcodes correspond directly to assembly opcodes

# Three-Address Intermediate Code

---

- Each instruction is of the form

$$x := y \text{ op } z$$

$$x := \text{op } y$$

- $y$  and  $z$  are registers or constants
- Common form of intermediate code
- The expression  $x + y * z$  is translated

$$t_1 := y * z$$

$$t_2 := x + t_1$$

- Each subexpression has a “name”



# Generating Intermediate Code

---

- Similar to assembly code generation
- But use any number of IL registers to hold intermediate results

# An Intermediate Language

---

$P \rightarrow S P \mid \varepsilon$

$S \rightarrow id := id \text{ op } id$

|  $id := op \ id$

|  $id := id$

|  $if \ id \ relop \ id \ goto \ L$

|  $L:$

|  $jump \ L$

- id's are register names
- Constants can replace id's
- Typical operators: +, -, \*

# Definition. Basic Blocks

---

- A basic block is a maximal sequence of instructions with:
  - no labels (except at the first instruction), and
  - no jumps (except in the last instruction)
- Idea:
  - Cannot jump into a basic block (except at beginning)
  - Cannot jump out of a basic block (except at end)
  - A basic block is a single-entry, single-exit, straight-line code segment

# Basic Block Example

---

- Consider the basic block
  1.  $L:$
  2.  $t := 2 * x$
  3.  $w := t + x$
  4.  $\text{if } w > 0 \text{ goto } L'$
- (3) executes only after (2)
  - We can change (3) to  $w := 3 * x$
  - Can we eliminate (2) as well?

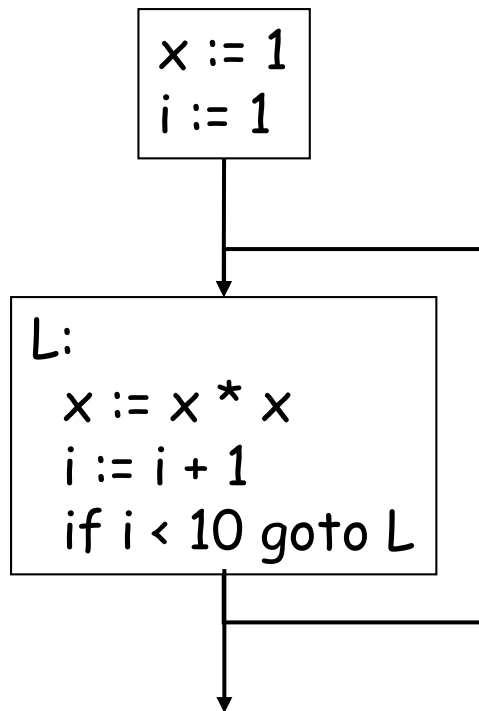
# Definition. Control-Flow Graphs

---

- A control-flow graph is 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
    - E.g., the last instruction in A is `jump LB`
    - E.g., execution can fall-through from block A to block B

# Example of Control-Flow Graphs

---



- The body of a method (or procedure) can be represented as a control-flow graph
- There is one initial node
- All “return” nodes are terminal

# Optimization Overview

---

- Optimization seeks to improve a program's resource utilization
  - Execution time (most often)
  - Code size
  - Memory usage
  - Disk access
- Optimization should not alter what the program computes
  - The answer must still be the same

# A Classification of Optimizations

---

- For languages like *C* and *Cool* there are three granularities of optimizations
  1. Local optimizations
    - Apply to a basic block in isolation
  2. Global optimizations
    - Apply to a control-flow graph (method body) in isolation
  3. Inter-procedural optimizations
    - Apply across method boundaries
- Most compilers do (1), many do (2), few do (3)



# Cost of Optimizations

---

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

# Local Optimizations

---

- The simplest form of optimizations
- No need to analyze the whole procedure body
  - Just the basic block in question
- Example: algebraic simplification

# Algebraic Simplification

---

- Some statements can be deleted

$x := x + 0$

$x := x * 1$

- Some statements can be simplified

$x := x * 0 \quad \Rightarrow \quad x := 0$

$y := y ** 2 \quad \Rightarrow \quad y := y * y$

$x := x * 8 \quad \Rightarrow \quad x := x \ll 3$

$x := x * 15 \quad \Rightarrow \quad t := x \ll 4; x := t - x$

(on some machines  $\ll$  is faster than  $*$ ; but not on all!)

# Constant Folding

---

- Operations on constants can be computed at compile time
  - If there is a statement  $x := y \text{ op } z$
  - And  $y$  and  $z$  are constants
  - Then  $y \text{ op } z$  can be computed at compile time
- Example:  $x := 2 + 2 \Rightarrow x := 4$
- Example:  $\text{if } 2 < 0 \text{ jump } L$  can be deleted

# Flow of Control Optimizations

---

- Eliminate unreachable basic blocks:
  - Code that is unreachable from the initial block
    - E.g., basic blocks that are not the target of any jump or “fall through” from a conditional
- Why would such basic blocks occur?
- Removing unreachable code makes the program smaller
  - And sometimes also faster
    - Due to memory cache effects (increased spatial locality)

# Flow of Control Optimizations (Cont.)

---

- Why would unreachable basic blocks occur?
  - Debug mode
    - #define DEBUG 0
    - If (DEBUG) then ...
  - Libraries
  - Result of other optimizations

# Single Assignment Form

---

- Some optimizations are simplified if each register occurs only once on the left-hand side of an assignment
- Rewrite intermediate code in *single assignment form*

$x := z + y$		$b := z + y$
$a := x$	$\Rightarrow$	$a := b$
$x := 2 * x$		$x := 2 * b$

( $b$  is a fresh register)

# Common Subexpression 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

- Example:

$x := y + z$

$x := y + z$

...

$\Rightarrow$

...

$w := y + z$

$w := x$

(the values of  $x$ ,  $y$ , and  $z$  do not change in the ... code)



# Copy Propagation

---

- If  $w := x$  appears in a block, replace subsequent uses of  $w$  with uses of  $x$ 
  - Assumes single assignment form

- Example:

$b := z + y$		$b := z + y$
$a := b$	$\Rightarrow$	$a := b$
$x := 2 * a$		$x := 2 * b$

- Only useful for enabling other optimizations
  - Constant folding
  - Dead code elimination

# Copy Propagation and Constant Folding

---

- Example:

$a := 5$

$a := 5$

$x := 2 * a$

$\Rightarrow$

$x := 10$

$y := x + 6$

$y := 16$

$t := x * y$

$t := x \ll 4$

or

$t := 160$

# Copy Propagation and Dead Code Elimination

---

If

$w := rhs$  appears in a basic block

$w$  does not appear anywhere else in the program

Then

the statement  $w := rhs$  is dead and can be eliminated

- Dead = does not contribute to the program's result

Example: ( $a$  is not used anywhere else)

$x := z + y$	$b := z + y$	$b := z + y$	$b := z + y$
$a := x$	$\Rightarrow a := b$	$\Rightarrow a := b$	$\Rightarrow x := 2 * b$
$x := 2 * a$	$x := 2 * a$	$x := 2 * b$	

*turn to single  
assignment form*

*copy  
propagation*

*dead code  
elimination*

# Applying Local Optimizations

---

- Each local optimization does little by itself
- Typically optimizations interact
  - Performing one optimization enables another
- Optimizing compilers repeat optimizations until no improvement is possible
  - The optimizer can also be stopped at any point to limit compilation time

# An Example

---

- Initial code:

```
a := x ** 2
b := 3
c := x
d := c * c
e := b * 2
f := a + d
g := e * f
```

# An Example

---

- Algebraic optimization:

$a := x^{**} 2$

$b := 3$

$c := x$

$d := c * c$

$e := b * 2$

$f := a + d$

$g := e * f$

# An Example

---

- Algebraic optimization:

$a := x * x$

$b := 3$

$c := x$

$d := c * c$

$e := b \ll 1$

$f := a + d$

$g := e * f$

# An Example

---

- Copy propagation:

a := x \* x

b := 3

c := x

d := c \* c

e := b << 1

f := a + d

g := e \* f



# An Example

---

- Copy propagation:

a := x \* x

b := 3

c := x

d := x \* x

e := 3 << 1

f := a + d

g := e \* f

# An Example

---

- Constant folding:  
a := x \* x  
b := 3  
c := x  
d := x \* x  
e := 3 << 1  
f := a + d  
g := e \* f

# An Example

---

- Constant folding:

$a := x * x$

$b := 3$

$c := x$

$d := x * x$

$e := 6$

$f := a + d$

$g := e * f$

# An Example

---

- Common subexpression elimination:

a := x \* x

b := 3

c := x

d := x \* x

e := 6

f := a + d

g := e \* f

# An Example

---

- Common subexpression elimination:

$a := x * x$

$b := 3$

$c := x$

$d := a$

$e := 6$

$f := a + d$

$g := e * f$

# An Example

---

- Copy propagation:

$a := x * x$

$b := 3$

$c := x$

$d := a$

$e := 6$

$f := a + d$

$g := e * f$

# An Example

---

- Copy propagation:

$a := x * x$

$b := 3$

$c := x$

$d := a$

$e := 6$

$f := a + a$

$g := 6 * f$

# An Example

---

- Dead code elimination:

a := x \* x

b := 3

c := x

d := a

e := 6

f := a + a

g := 6 \* f



# An Example

---

- Dead code elimination:

$a := x * x$

$f := a + a$

$g := 6 * f$

- This is the final form

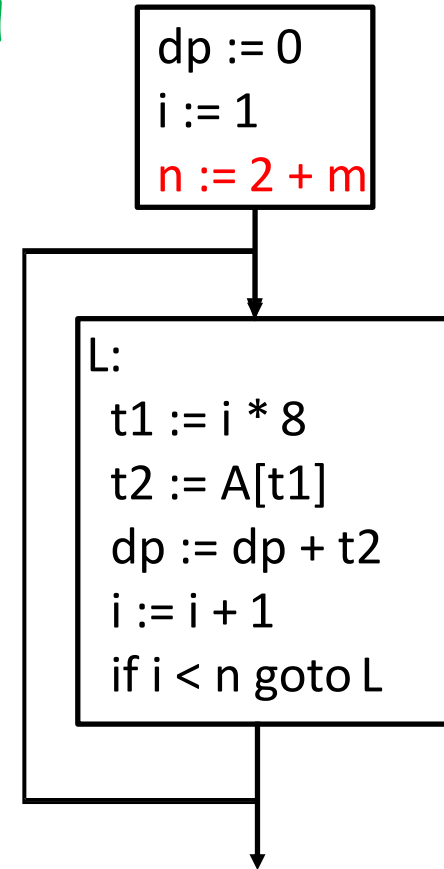
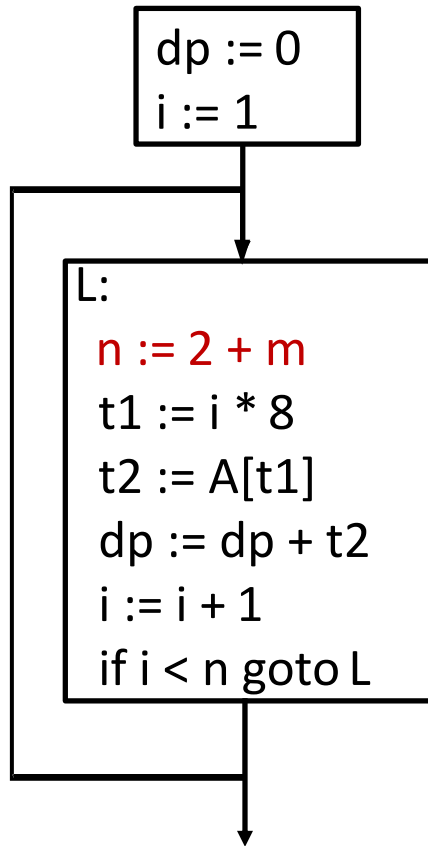
# Loop Optimization

---

1. Code Motion
2. Reduction in Strength
3. Induction Variables elimination

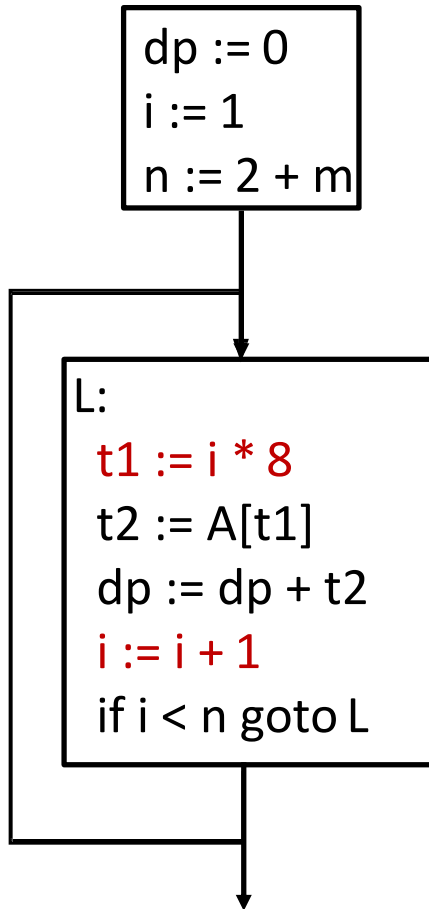
# Code Motion

- 'n := 2 + m' can be moved out of the loop

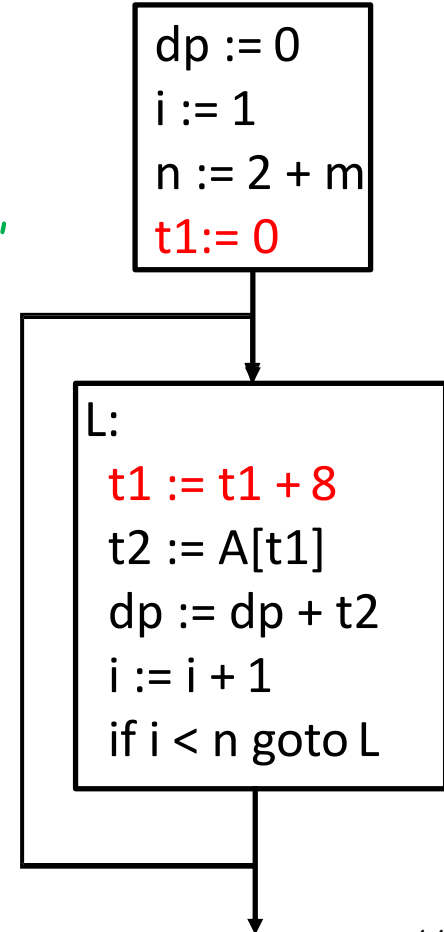


# Reduction in Strength

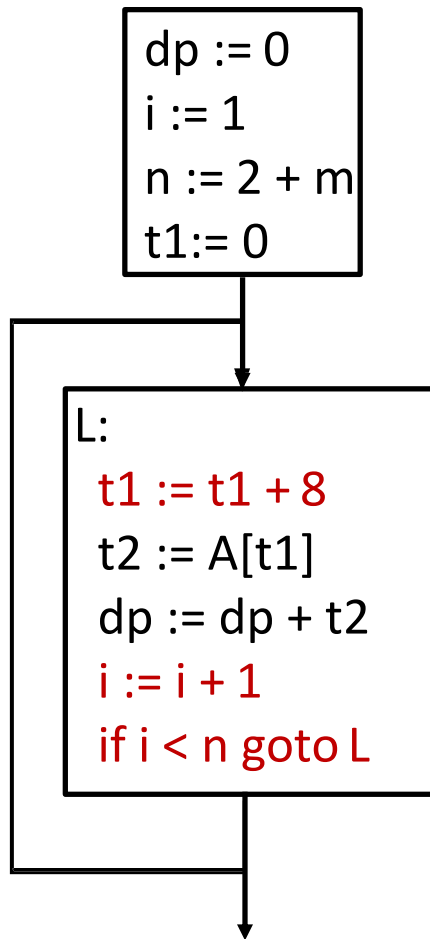
- $i$  is increased by 1
- $t1$  is increased by 8
- '\*' can be replaced by '+'



⇒

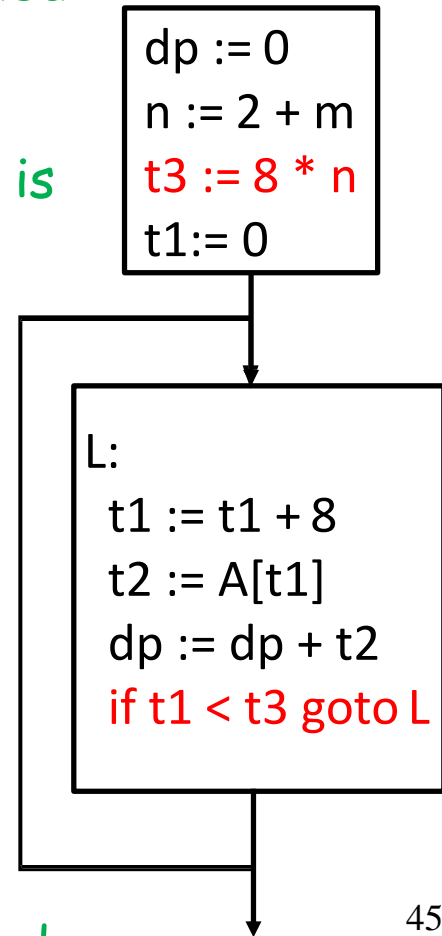


# Induction Variables Elimination



- *i* and *t1* here are regarded as 'induction variables'
- *i* can be removed and *t1* is used instead

⇒

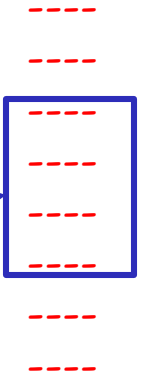


- '*i* := *i* + 1' is then a dead code and can be removed

# Peephole Optimizations on Assembly Code

---

- These optimizations work on intermediate code
  - Target independent
  - But they can be applied on assembly language also
- Peephole optimization is effective for improving assembly code
  - The “peephole” is a short sequence of (usually contiguous) instructions
  - The optimizer replaces the sequence with another equivalent one (but faster)



# Peephole Optimizations (Cont.)

---

- Write peephole optimizations as replacement rules

$$i_1, \dots, i_n \rightarrow j_1, \dots, j_m$$

where the rhs is the improved version of the lhs

- Example:

`move $a $b, move $b $a`  $\rightarrow$  `move $a $b`

- Works if `move $b $a` is not the target of a jump

- Another example

`addiu $a $a i, addiu $a $a j`  $\rightarrow$  `addiu $a $a i+j`

# Peephole Optimizations (Cont.)

---

3AC

```
x := y + 1;  
z := x + 2;
```

Machine Code

```
move $a y  
addiu $a $a 1  
move x $a  
move $a x  
addiu $a $a 2  
move $a z
```

<- This move statement is not needed



# Peephole Optimizations (Cont.)

---

Use of specialized instructions

```
move $a a  
addiu $a $a 1  
move a $a
```

} ⇒ increment a

# Peephole Optimizations (Cont.)

---

Some machine codes can be deleted

```
multu $a $a 1  
addiu $a $a 0
```

Using shift to left instead of multiplication by powers of 2

Using shift to right instead of division into powers of 2

# Local Optimizations: Notes

---

- Intermediate code is helpful for many optimizations
- Many simple optimizations can still be applied on assembly language
- “Program optimization” is grossly misnamed
  - Code produced by “optimizers” is not optimal in any reasonable sense
  - “Program improvement” is a more appropriate term
- Next time: global optimizations

# Question?

---

Which of the following are valid local optimizations for the given basic block? Assume that only  $g$  and  $x$  are referenced outside of this basic block.

- Copy propagation: Line 4 becomes  $d := a * b$ .
- Common subexpression elimination: Line 5 becomes  $e := d$ .
- Dead code elimination: Line 3 is removed.
- After many rounds of valid optimizations, the entire block can be reduced to  $g := 5$ .

```
1  a := 1
2  b := 3
3  c := a + x
4  d := a * 3
5  e := b * 3
6  f := a + b
7  g := e - f
```