Lecture 12

Code optimization

Local optimization

Hyosu Kim

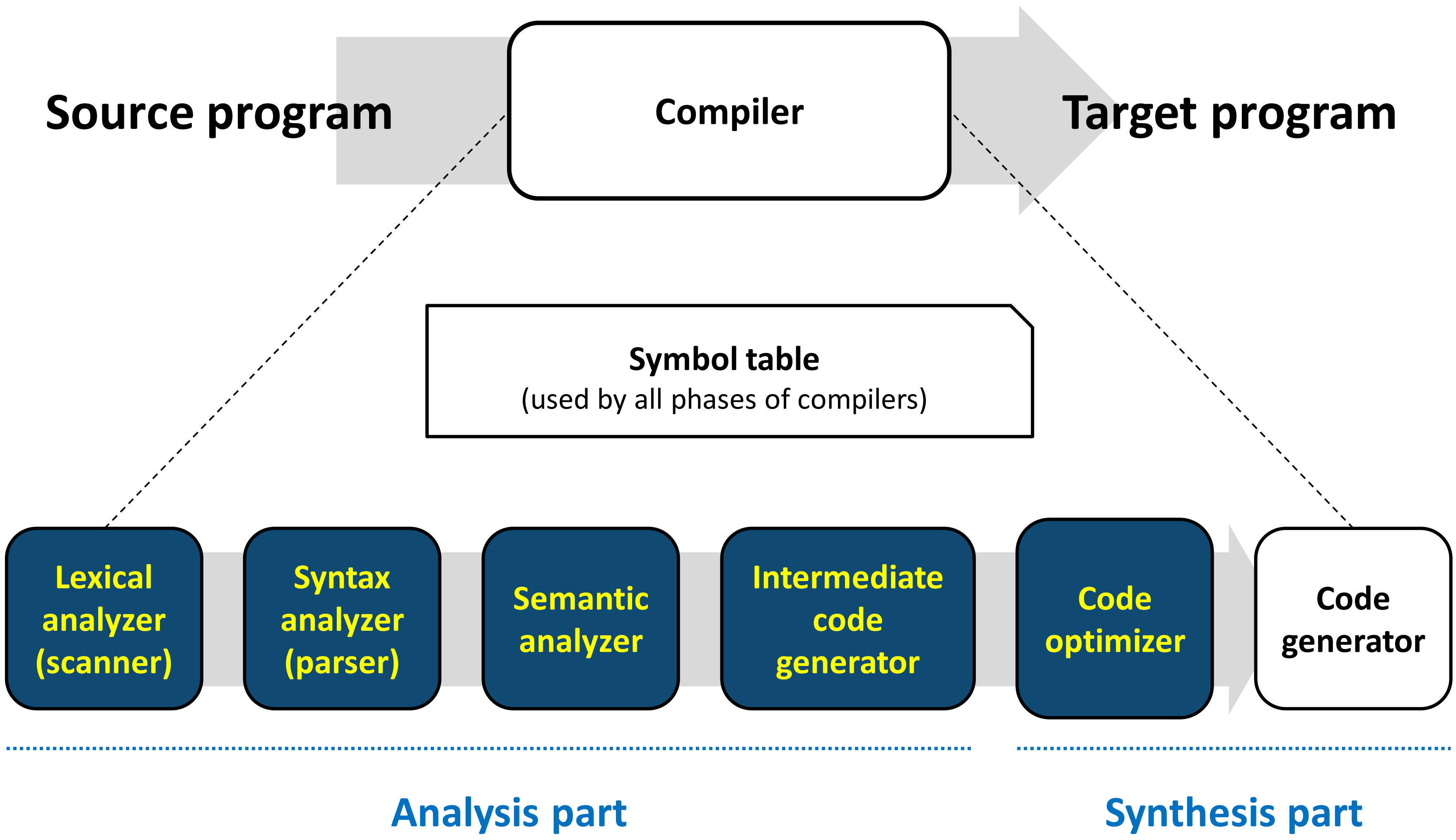
School of Computer Science and Engineering

Chung-Ang University, Seoul, Korea

https://hcslab.cau.ac.kr

hskimhello@cau.ac.kr, hskim.hello@gmail.com

Overview



Source program
Compiler
Target program
Symbol table
(used by all phases of compilers)
Lexical analyzer (scanner)
Syntax analyzer (parser)
Semantic analyzer
Intermediate code generator
Code optimizer
Code generator
Analysis part
Synthesis part

Intermediate code optimizer

Improves the code generated by the intermediate code generator

for optimizing the runtime performance, memory usage, and power consumption

of the program, but preserving the semantics of the original program

An intermediate representation
e.g., three address code

→ **Intermediate code optimizer** →

An optimized intermediate representation

Intermediate code optimizer

Improves the code generated by the intermediate code generator

for optimizing the runtime performance, memory usage, and power consumption

of the program, but preserving the semantics of the original program

Why do we need optimization???

- Intermediate code is generated without considering optimization

  e.g., the code has many useless variables…

- Programmers write a poor code frequently

Intermediate code optimizer

Examples

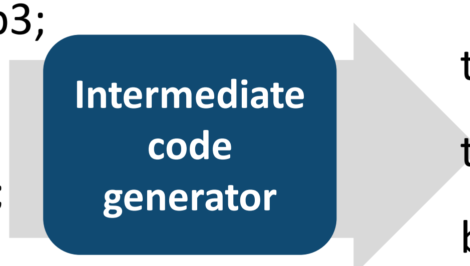
```
int x, y;

bool b1, b2, b3;

b1 = x + x < y;

b2 = x + x == y;

b3 = x + x > y;
```




```
t0 = x + x
t1 = t0 < y
b1 = t1

t2 = x + x
t3 = t2 == y
b2 = t3

t4 = x + x
t5 = t4 > y
b3 = t5
```

Intermediate code optimizer

Examples

```
int x, y;
bool b1, b2, b3;

b1 = x + x < y;
b2 = x + x == y;
b3 = x + x > y;
```

Intermediate code generator

```
t0 = x + x
t1 = t0 < y
b1 = t1

t2 = x + x
t3 = t2 == y
b2 = t3

t4 = x + x
t5 = t4 > y
b3 = t5
```

Intermediate code optimizer

```
t0 = x + x
b1 = t0 < y

b2 = t0 == y

b3 = t0 > y
```

Intermediate code optimizer

Examples

```
while (x < y + z) {
    x = x + 1;
}
```

Intermediate code generator

```
L0:
    t0 = y + z
    t1 = x < t0
    ifnot t1 goto L1
    t2 = x + 1
    x = t2
    goto L0
L1:
    …
```

Intermediate code optimizer

Intermediate code optimizer

Improves the code generated by the intermediate code generator

for optimizing the runtime performance, memory usage, and power consumption

of the program, but preserving the semantics of the original program

Optimizations can be also performed with machine-level code

- To improve performance based on the characteristics of specific machines

Intermediate code optimizations try to improve performance more generally

(independently of machines)

Basic blocks

A basic block is a maximal sequence of consecutive three address instructions

- A program can only enter the basic block through the first instruction in the block
- The program leaves the block without halting or branching at the last instruction in the block

```
int x, y, z;
y = 10 + 100;
if (x == 10)  z = y;
else  x = y;
y = 10;
```

Intermediate code generator

```
t0 = 10 + 100
y = t0
t1 = x == 10
if t1 goto L0
```

```
x = y
goto L1
```

```
L0:
   z = y
```

```
L1:
   y = 10
```

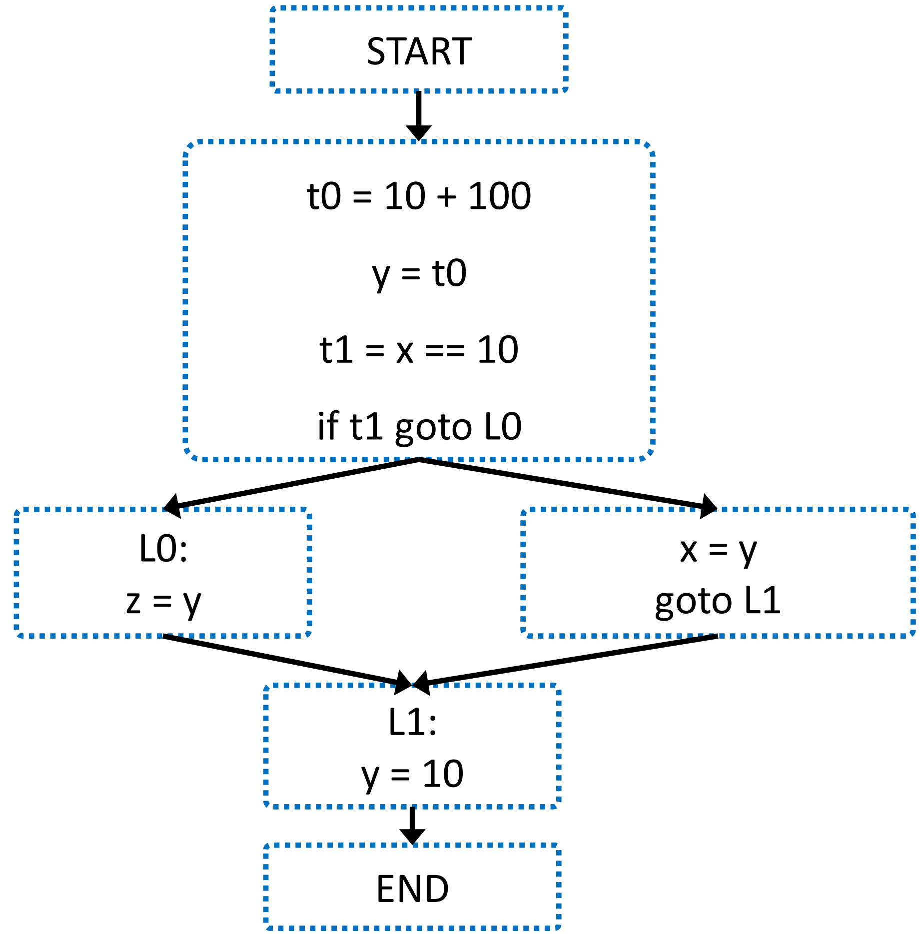
Basic blocks

Control flow graphs

A control flow graph is a graph of the basic blocks

- Edges indicates which blocks can follow which other blocks

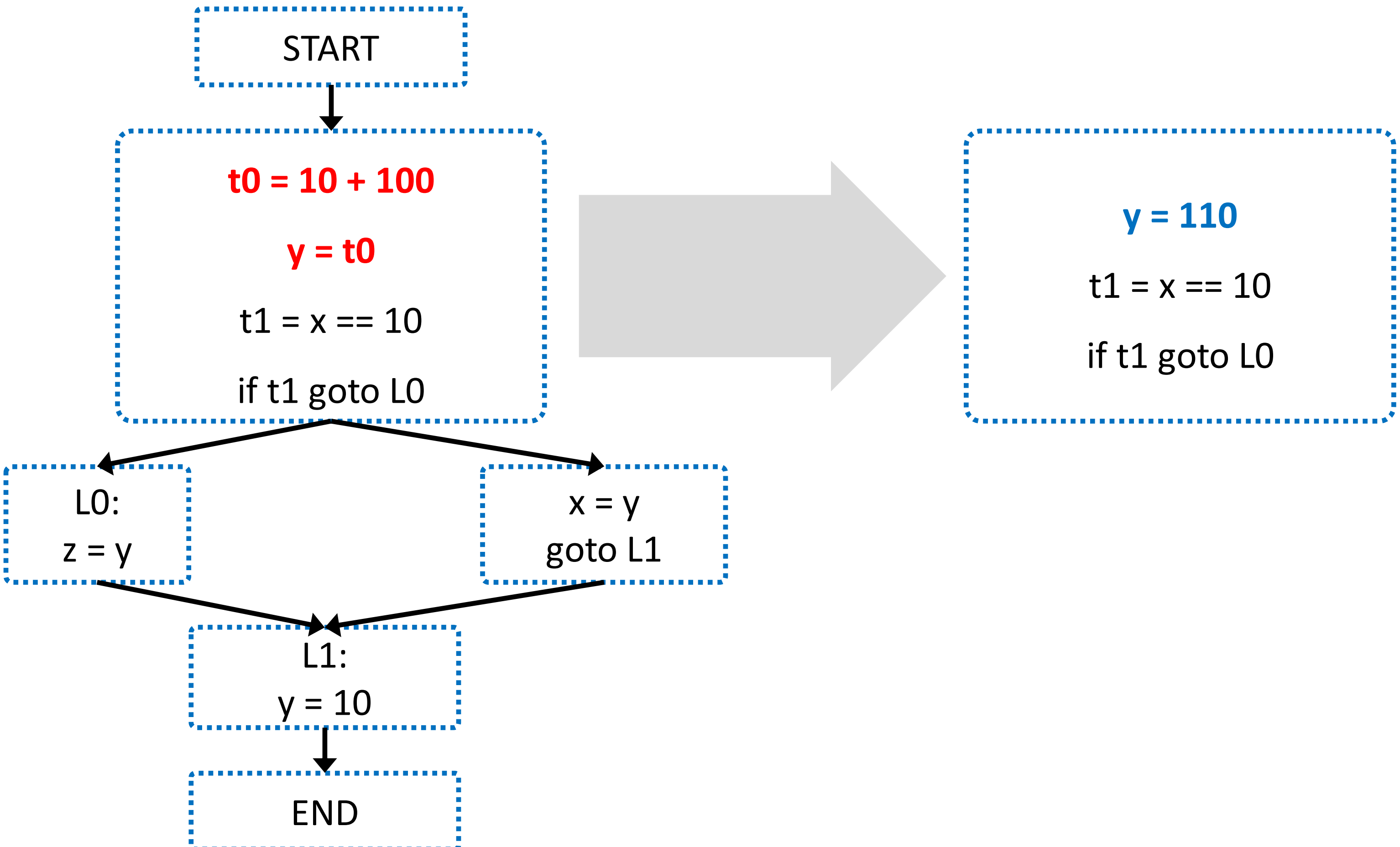
```
t0 = 10 + 100
y = t0
t1 = x == 10
if t1 goto L0
x = y
goto L1
L0:
   z = y
L1:
   y = 10
```



Types of intermediate code optimizations

An optimization is “local”

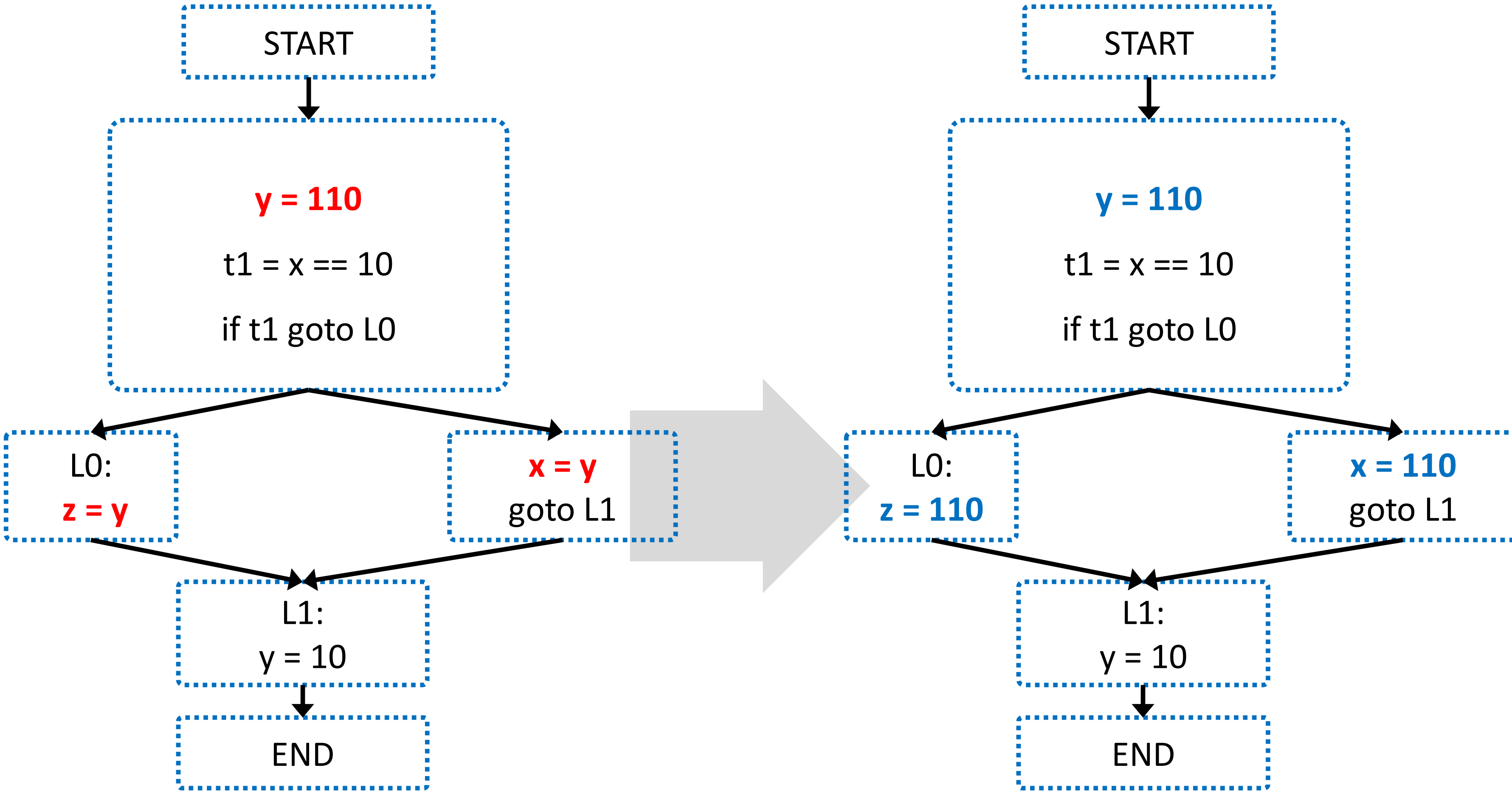
If it works on just a single basic block



Types of intermediate code optimizations

An optimization is “global”

If it works on an entire control-flow graph



Local optimizations

Typical local optimization techniques

- Common sub expressions elimination
- Copy propagation
- Dead code elimination
- Arithmetic simplification
- Constant folding

Local optimizations

Common sub expressions elimination

- Let’s suppose that we have two variable assignments

v0 = a op b

…

v1 = a op b

- If the values of v0, a, and b have not changed between the two assignments, then we can rewrite the code as

v0 = a op b

…

v1 = v0

Local optimizations

Common sub expressions elimination

Example

t0 = x + x

y = t0 * x

t2 = x + x

t3 = t2 == y

if t3 goto L2

…

t0 = x + x

y = t0 * x

t2 = t0

t3 = t2 == y

if t3 goto L2

…

Local optimizations

Copy propagation

- Let's suppose that we have a variable assignment

v0 = v1

- As long as v0 and v1 are not reassigned, we can rewrite the following code

a = … v0 … ⇒ a = … v1 …

Local optimizations

Copy propagation

Example

t0 = x + x

y = t0 * x

t2 = t0

t3 = t2 == y

if t3 goto L2

…

t0 = x + x

y = t0 * x

t2 = t0

t3 = t0 == y

if t3 goto L2

…

Local optimizations

Dead code elimination

- A variable assignment is called dead if the value of that assignment is never used
- If there is a dead assignment, remove the code

Example

t0 = x + x		t0 = x + x
y = t0 * x		Y = t0 * x
t2 = t0	→	
t3 = t0 == y		t3 = t0 == y
if t3 goto L2		if t3 goto L2
…		…

Local optimizations

Arithmetic simplification

- Replace inefficient operations with more efficient one

Example

x = a * 2 → x = a << 1

Constant folding

- Computes expressions at compile-time if they have a constant value

Example

x = 10 * 2 → x = 20

Local optimizations

Practice

Optimize the following intermediate code!!

t0 = a * a

t1 = a * a

t2 = t0 + t1

t3 = t0 + t0

t4 = t3 + 1

…

(in later parts, t4 is used)

Local optimizations

Typical local optimization techniques

- Common sub expressions elimination
- Copy propagation
- Dead code elimination
- Arithmetic simplification
- Constant folding

How to implement these optimization techniques?

Implementation of local optimizations

Available expression analysis

for common sub expressions elimination & copy propagation

- An expression is called available if variables in the expression hold an up-to-date value

Example

- After executing

  t0 = a * a

  t1 = a * a

  Available expressions = {t0 = a * a, t1 = a * a}

Implementation of local optimizations

Available expression analysis

Determines for each point in a program the set of available expressions

- Initially, no expressions are available
- Whenever we check a statement **a = operation (e.g., a = b + c)**
  - Any expression holding **a** is invalidated!!
  - The new expression **a = operation** becomes available

Three-address code	Available expressions
(before executing t0 = a* a)	{}
t0 = a * a	
(after t0 = a * a, before t1 = a * a)	{t0 = a * a}
t1 = a * a	
(after t1 = a * a, before t0 = a + t1)	{t0 = a * a, t1 = a * a}
t0 = a + t1	
(after t0 = a + t1)	{t1 = a * a, **t0 = a + t1**}

Implementation of local optimizations

Common subexpressions elimination with available expressions

Let's suppose that we currently check an expression **b = operation1**

and an expression **a = operation1** is in the set of available expressions

- The right-hand sides of two expressions are same

Three-address code	Available expressions
(before executing t1 = a * a)	{t0 = **a * a**}
t1 = **a * a**	

Implementation of local optimizations

Common subexpressions elimination with available expressions

Let's suppose that we currently check an expression **b = operation1**

and an expression **a = operation1** is in the set of available expressions

- The right-hand sides of two expressions are same

Three-address code	Available expressions
(before executing t1 = a * a)	{t0 = **a * a**}
t1 = **a * a**	

Then, replace the right-hand side of the current expression (b = **operation1**)

by the left-hand side of the corresponding available expression (**a** = operation1)

Three-address code	Available expressions
(before executing t1 = a * a)	**{t0 = a * a}**
~~t1 = a * a~~ **t1 = t0**	

Implementation of local optimizations

Copy propagation with available expressions

Let's suppose that we currently check an expression **c = operation with b**

and an expression **b = a** or **b = constant number** is in the set of available expressions

Three-address code	Available expressions
(before executing t1 = a * b)	{**b** = a}
t1 = a * **b**	

Implementation of local optimizations

Copy propagation with available expressions

Let's suppose that we currently check an expression **c = operation with b**

and an expression **b = a** or **b = constant number** is in the set of available expressions

Three-address code	Available expressions
(before executing t1 = a * b)	{**b** = a}
t1 = a * **b**	

Then, replace **b** in the right hand side of the current expression (c = operation with **b**)

by **a** (the right-hand side of the corresponding available expression) (b = **a**)

Three-address code	Available expressions
(before executing t1 = a * a)	{b = **a**}
t1 = a * ~~b~~ **a**	

Implementation of local optimizations

Example

Three-address code	Available expressions
	{}
t0 = a * a	
	{t0 = **a * a**}
t1 = **a * a**	
t2 = t0 + t1	
t0 = t0 + t0	
t3 = t0 + 1	

Implementation of local optimizations

Example

Three-address code	Available expressions
	{}
t0 = a * a	
	{t0 = a * a}
t1 = **t0**	
	{t0 = a * a, **t1** = t0}
t2 = t0 + **t1**	
t0 = t0 + t0	
t3 = t0 + 1	

Implementation of local optimizations

Example

Three-address code	Available expressions
	{}
t0 = a * a	
	{t0 = a * a}
t1 = **t0**	
	{t0 = a * a, t1 = t0}
t2 = t0 + **t0**	
	{t0 = a * a, t1 = t0, t2 = **t0 + t0**}
t0 = **t0 + t0**	
t3 = t0 + 1	

Implementation of local optimizations

Example

Three-address code	Available expressions
	{}
t0 = a * a	
	{t0 = a * a}
t1 = **t0**	
	{t0 = a * a, t1 = t0}
t2 = t0 + **t0**	
	{t0 = a * a, t1 = t0, t2 = t0 + t0}
t0 = **t2**	
	{~~t0 = a * a, t1 = t0, t2 = t0 + t0,~~ **t0** = t2}
t3 = **t0** + 1	

Implementation of local optimizations

Example

Three-address code	Available expressions
	{}
t0 = a * a	
	{t0 = a * a}
t1 = **t0**	
	{t0 = a * a, t1 = t0}
t2 = t0 + **t0**	
	{t0 = a * a, t1 = t0, t2 = t0 + t0}
t0 = **t2**	
	{t0 = t2}
t3 = **t2** + 1	
	{t0 = t2, t3 = t2 + 1}

Implementation of local optimizations

Live variable analysis

for dead code elimination

- A variable is called a live variable if it holds a value that will be needed in the future

Three-address code	Live variables
	{b}
a = b;	
	{a, b}
c = a + b;	
	{a, b}
d = a;	
(the value of b and d will be used in the future)	{b, d}

Implementation of local optimizations

Live variable analysis

To know whether a variable will be used in the future or not,

checks the statements in a basic block in a reverse order

- Initially, some small set of variables are known to be live (e.g., variables will be used in the next block)
- Just before executing the statement **a = … b …**
  - **a is not alive** because its value will be newly overwritten
  - **b is alive** because it will be used

Three-address code	Live variables
d = a + c;	
(the value of b and d will be used in the future)	{b, d}

Implementation of local optimizations

Live variable analysis

To know whether a variable will be used in the future or not,

checks the statements in a basic block in a reverse order

- Initially, some small set of variables are known to be live (e.g., variables will be used in the next block)
- Just before executing the statement **a = … b …**
    - **a is not alive** because its value will be newly overwritten
    - **b is alive** because it will be used

Three-address code	Live variables
(the value of a, b, c will be used in the future)	{a, b, c}
d = a + c;	
(the value of b and d will be used in the future)	{b, d}

Implementation of local optimizations

Live variable analysis

To know whether a variable will be used in the future or not,

checks the statements in a basic block in a reverse order

- Initially, some small set of variables are known to be live (e.g., variables will be used in the next block)
- Just before executing the statement **a = … b …**
  - **a is not alive** because its value will be newly overwritten
  - **b is alive** because it will be used

Three-address code	Live variables
	{a, b, c, d}
a = a + c;	
(the value of b and d will be used in the future)	{b, d}

Implementation of local optimizations

Dead code elimination with live variables

Let's suppose that we currently check an expression **b = operation1**

and **b** is not a live variable after this assignment

Three-address code	Live variables
t1 = a * a	
(after executing t1 = a * a)	{a}

Then, eliminate the assignment statement

Three-address code	Live variables
~~**t1 = a * a**~~	
(after executing t1 = a * a)	{a}

Implementation of local optimizations

Example

Three-address code	Live variables
t0 = a * a	
t1 = t0	
t2 = t0 + t0	
t0 = t2	
t3 = t2 + 1	
(t3 will be used in the future)	{t3}

Implementation of local optimizations

Example

Three-address code	Live variables
t0 = a * a	
t1 = t0	
t2 = t0 + t0	
t0 = t2	
	{t2}
t3 = t2 + 1	
(t3 will be used in the future)	{t3}

Implementation of local optimizations

Example

Three-address code	Live variables
t0 = a * a	
t1 = t0	
	{t0}
t2 = t0 + t0	
~~t0 = t2~~	
	{t2}
t3 = t2 + 1	
(t3 will be used in the future)	{t3}

Implementation of local optimizations

Example

Three-address code	Live variables
	{a}
t0 = a * a	
~~t1 = t0~~	
	{t0}
t2 = t0 + t0	
~~t0 = t2~~	
	{t2}
t3 = t2 + 1	
(t3 will be used in the future)	{t3}

Summary: intermediate code optimizer

Improves the code generated by the intermediate code generator

for optimizing the runtime performance, memory usage, and power consumption of the program, but preserving the semantics of the original program

An intermediate representation
e.g., three address code

Intermediate code optimizer

An optimized intermediate representation

Types of optimizations

- Local optimizations
- Global optimizations

Summary: local optimizations

Typical local optimization techniques

- Common sub expressions elimination **=> through available expression analysis**
- Copy propagation **=> through available expression analysis**
- Dead code elimination **=> through live variable analysis**
- Arithmetic simplification
- Constant folding