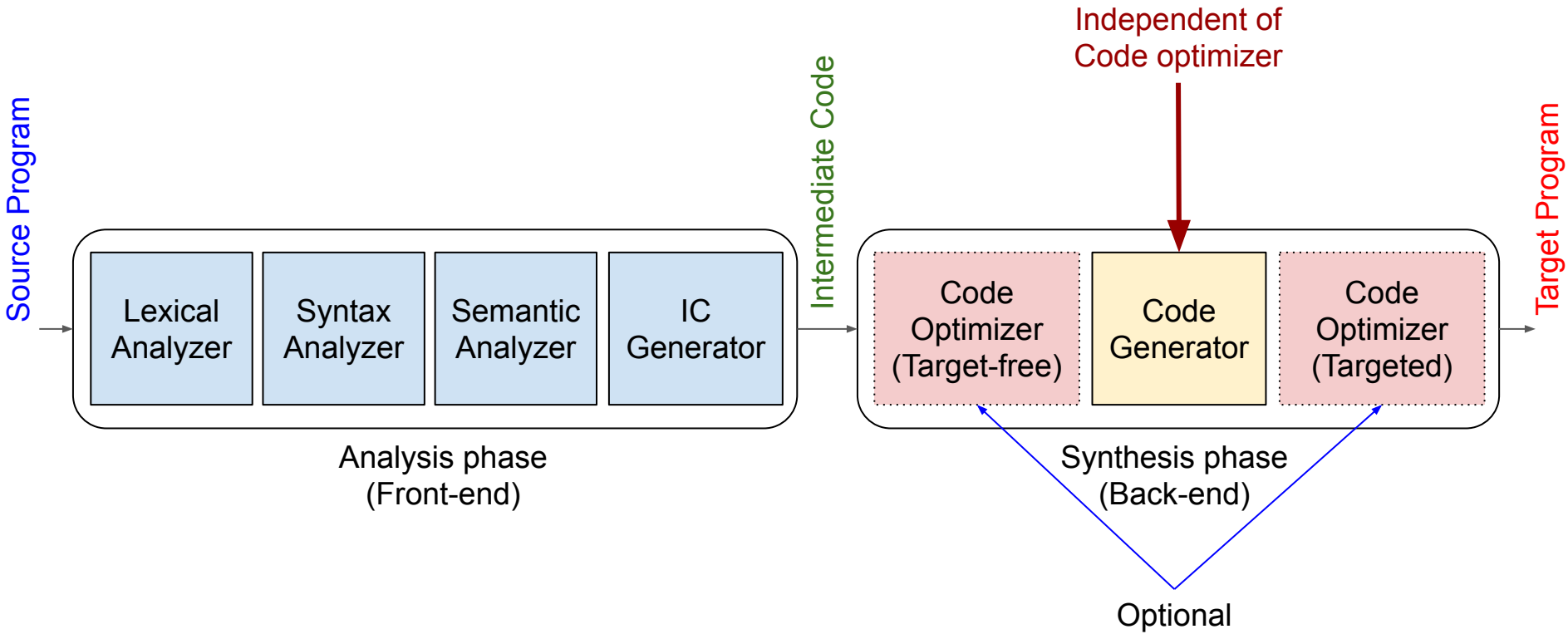
Code Optimization

Md Shad Akhtar
Assistant Professor
IIIT Dharwad

Where are we?


Source Program
Lexical Analyzer
Syntax Analyzer
Semantic Analyzer
IC Generator
Analysis phase
(Front-end)
Intermediate Code
Independent of
Code optimizer
Code Optimizer
(Target-free)
Code Generator
Code Optimizer
(Targeted)
Synthesis phase
(Back-end)
Optional
Target Program

Expectation from Code Optimization

- Improve the Intermediate code generated by the previous step to take better advantage of resources.
- Must preserve the semantic of the source program
- Should produce IR that is as efficient as possible.
- Should not take too long to process inputs.
- Should not introduce any errors

Why Code Optimization is required?

- Intermediate code generation introduces redundancy
- Programmer has not written an optimal code

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

$$
\begin{aligned}
t_0 &= x + x; \\
t_1 &= y; \\
b_1 &= t_0 < t_1; \\
\\
t_2 &= x + x; \\
t_3 &= y; \\
b_2 &= t_2 == t_3; \\
\\
t_4 &= x + x; \\
t_5 &= y; \\
b_3 &= t_4 > t_5;
\end{aligned}
$$

Why Code Optimization is required?

- Intermediate code generation introduces redundancy
- Programmer has not written an optimal code

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

$t_0 = x + x;$
$t_1 = y;$
$b_1 = t_0 < t_1;$

$b_2 = t_2 == t_3;$

$b_3 = t_4 > t_5;$

Why Code Optimization is required?

- Intermediate code generation introduces redundancy
- Programmer has not written an optimal code

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

L0:

$t_0 = y + z;$

$t_1 = x < t_0;$

IfZ t_1 goto L1;

$x = x - y;$

goto L0;

L1:

Why Code Optimization is required?

- Intermediate code generation introduces redundancy
- Programmer has not written an optimal code

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

$t_0 = y + z;$
L0:
$t_1 = x < t_0;$
IfZ t_1 goto L1;
$x = x - y;$
goto L0;
L1:

“Optimization”

- Optimization: Finding an optimal piece of code
- The term optimization is slightly misused here
- This is, in general, undecidable.
- Our goal will be code improvement rather than code optimization.

What are we optimizing?

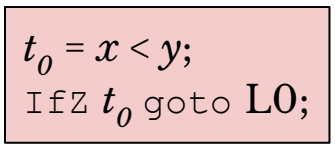
- Runtime
  - Make the program as fast as possible at the expense of time and power
- Memory usage
  - Generate the smallest possible executable at the expense of time and power
- Power consumption
  - Choose simple instructions at the expense of speed and memory usage
- Others in terms of coding
  - Minimize function calls,
  - Reduce use of floating-point hardware,
  - ..

Basic Blocks

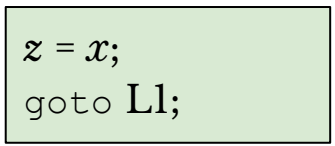
- A basic clock is a sequence of intermediate instructions where
  - Execution starts at the start of the sequence
  - Execution ends at the end of the sequence
  - Each instruction in between executed sequentially exactly once. (*NO Jump!!*)
- Instructions in a basic block are executed as a group.

```
if (x < y)
     z = x;
else
     z = y;
z = z * z;
```

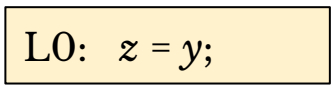
```
            t0 = x < y;
            IfZ t0 goto L0;
            z = x;
            goto L1;
L0:         z = y;
L1:         z = z * z;
```



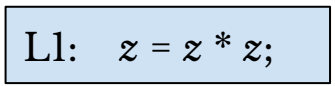
B1



B2



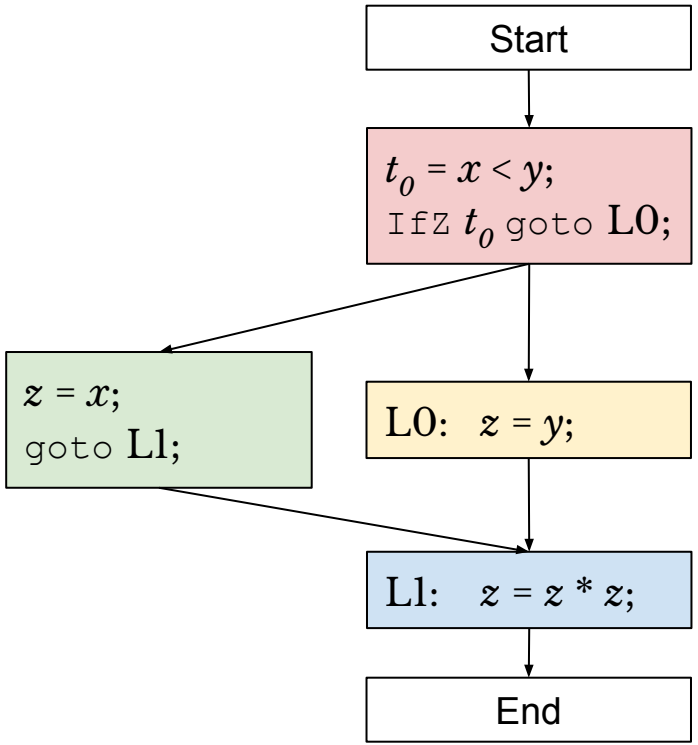
B3



B4

Control Flow Graph (CFG)

- A control-flow graph (CFG) is a graph of the basic blocks in a function.
- An edge from one basic block to another indicates that control can flow from the end of the first block to the start of the second block.
- There are two dedicated nodes for the start and end of a function.



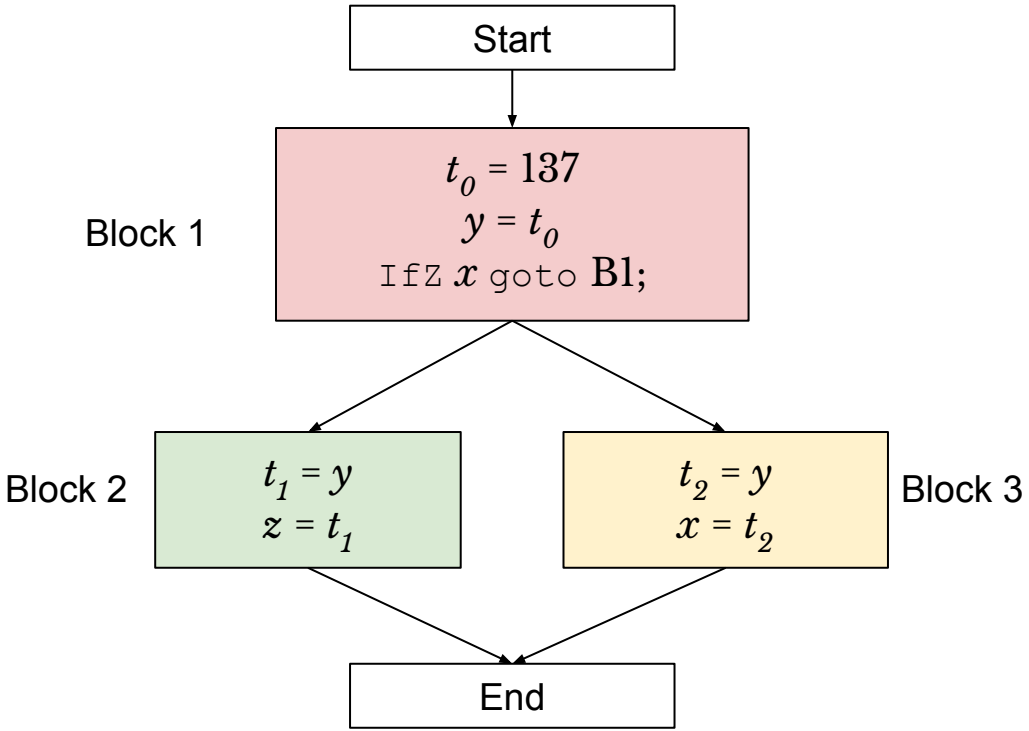
Types of Optimization

- **Local:** Optimization with in basic block
  - Common Subexpression Elimination
  - Copy Propagation
  - Dead code elimination
  - Others
    - Constant Folding:
      - Evaluate expressions at compile-time if they have a constant value
        - x = 4 * 5 ⇒ x = 20
    - Arithmetic Simplification(or Reduction in strength):
      - Replace “hard” operations with easier ones.
        - x = 4 * a ⇒ $x = a$ << 2
        - x^2 ⇒ $x * x$

- **Global:** Optimization across basic blocks, i.e., entire control flow graph

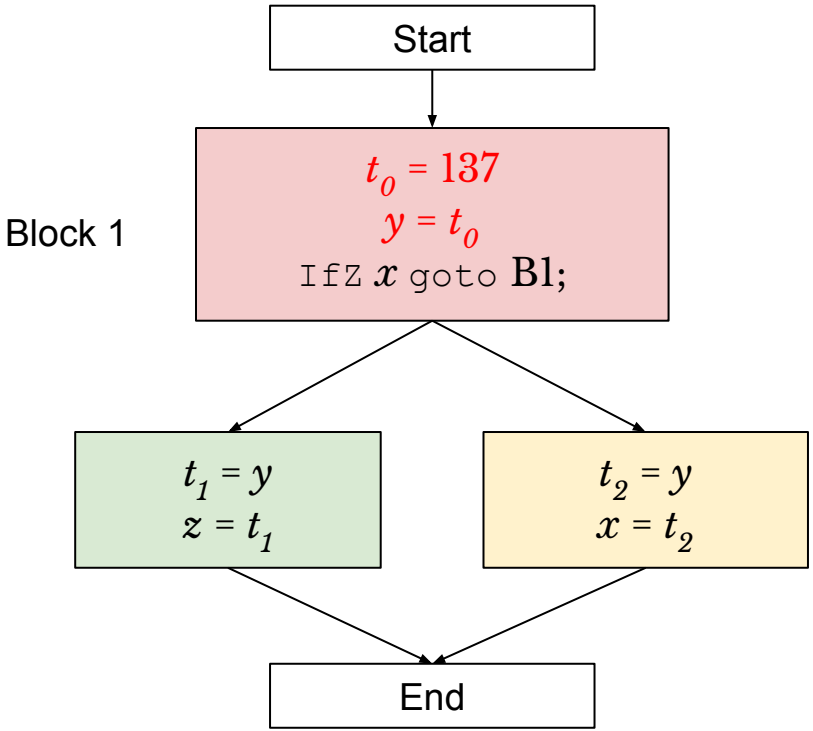
Local Optimization

```
int main()
{
        int x;
        int y;
        int z;
        y = 137;
        if (x == 0)
                z = y;
        else
                x = y;
}
```



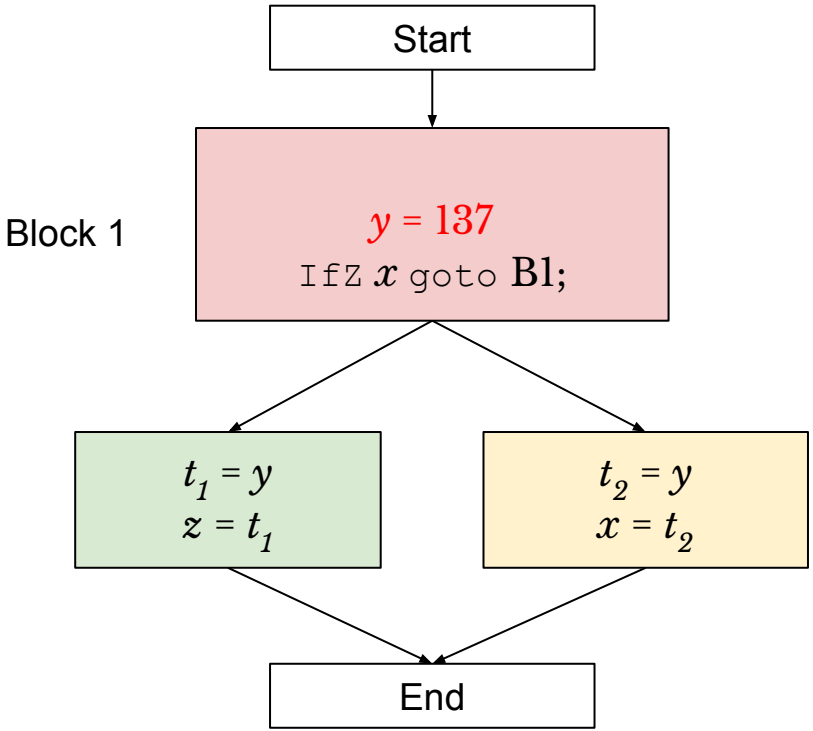
Local Optimization: Block 1

```
int main()
{
        int x;
        int y;
        int z;
        y = 137;
        if (x == 0)
                z = y;
        else
                x = y;
}
```



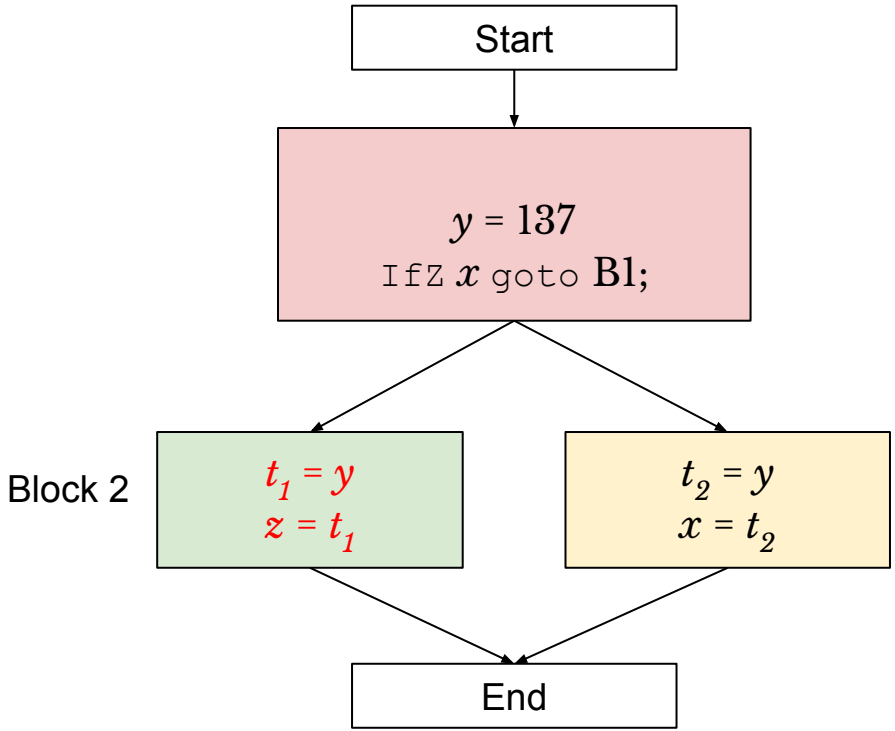
Local Optimization: Block 1

```
int main()
{
        int x;
        int y;
        int z;
        y = 137;
        if (x == 0)
                z = y;
        else
                x = y;
}
```



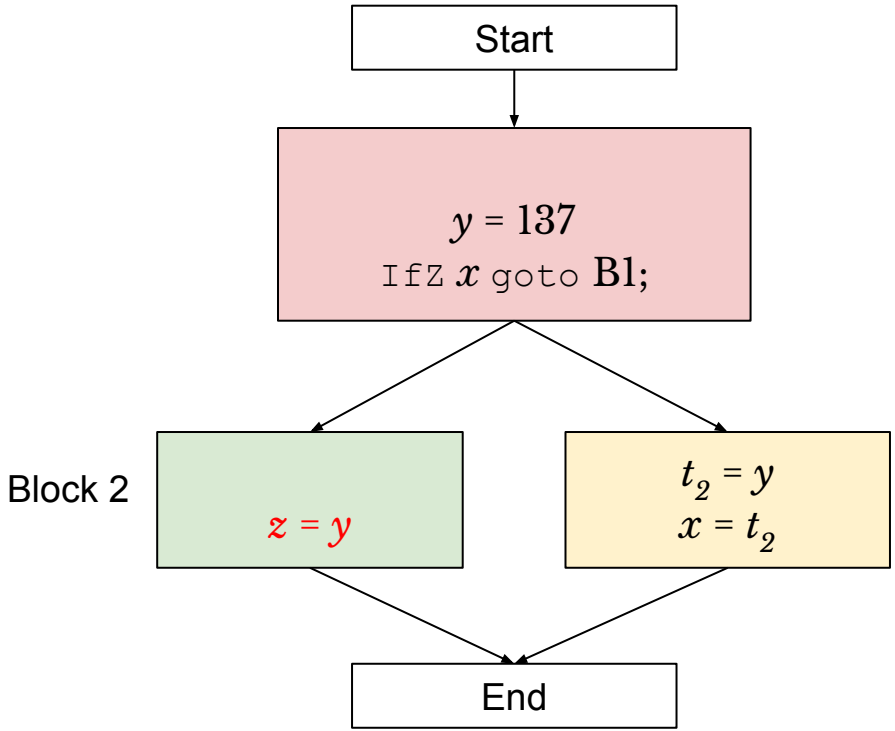
Local Optimization: Block 2

```
int main()
{
        int x;
        int y;
        int z;
        y = 137;
        if (x == 0)
                z = y;
        else
                x = y;
}
```



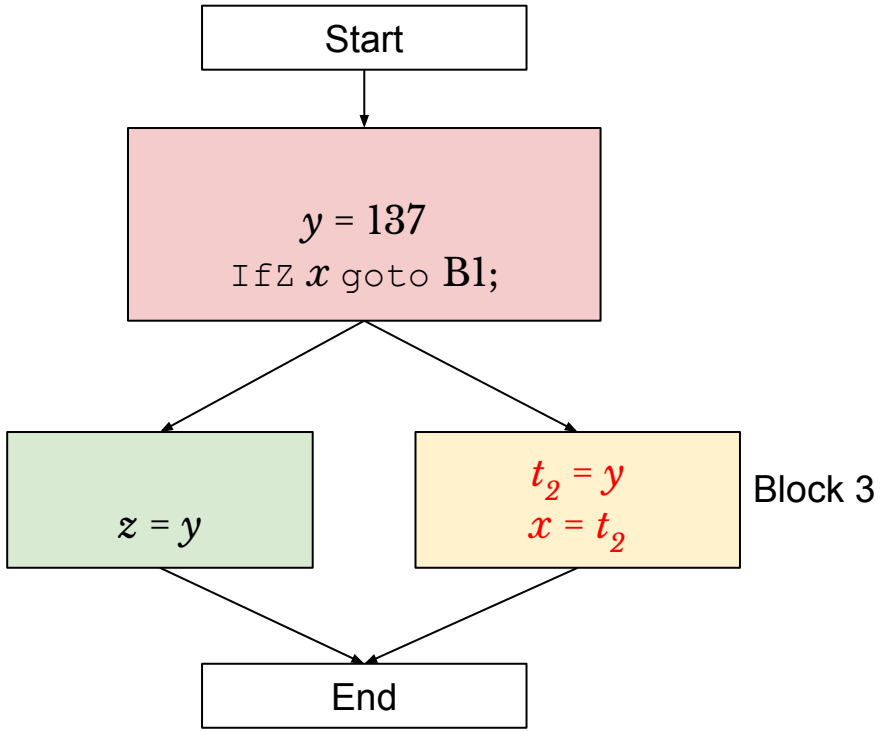
Local Optimization: Block 2

```
int main()
{
        int x;
        int y;
        int z;
        y = 137;
        if (x == 0)
                z = y;
        else
                x = y;
}
```



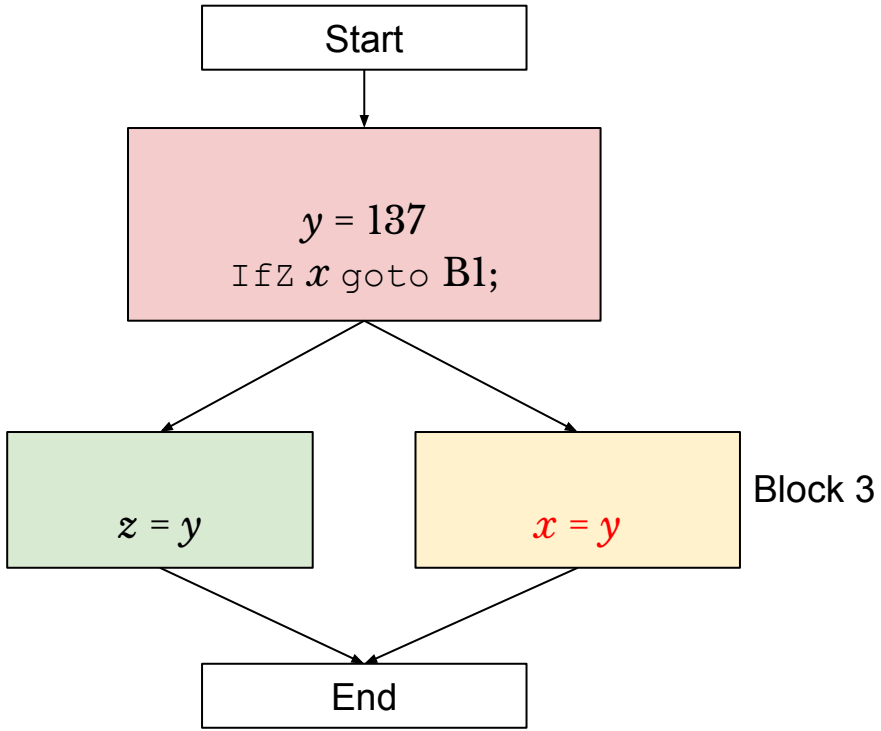
Local Optimization: Block 3

```
int main()
{
        int x;
        int y;
        int z;
        y = 137;
        if (x == 0)
                z = y;
        else
                x = y;
}
```



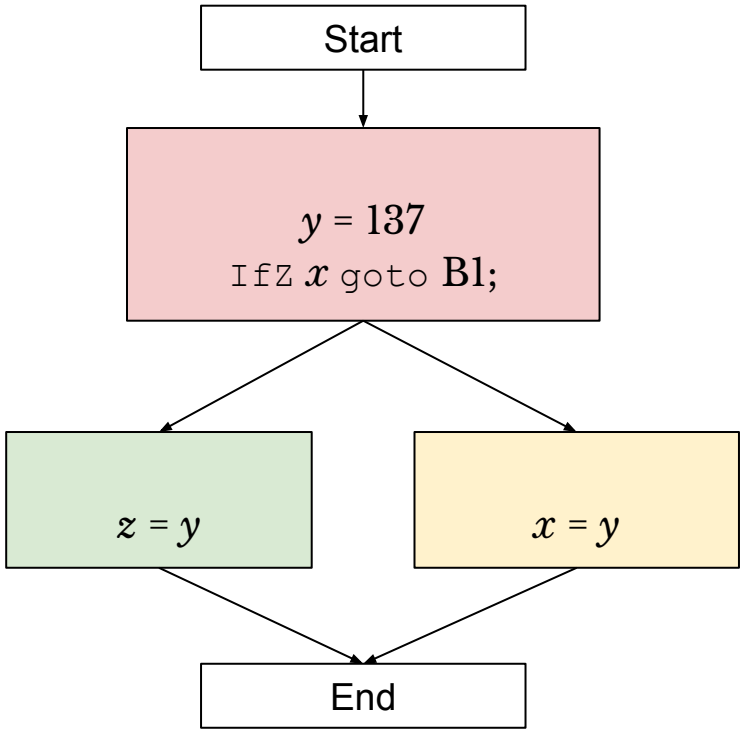
Local Optimization: Block 3

```
int main()
{
        int x;
        int y;
        int z;
        y = 137;
        if (x == 0)
                z = y;
        else
                x = y;
}
```



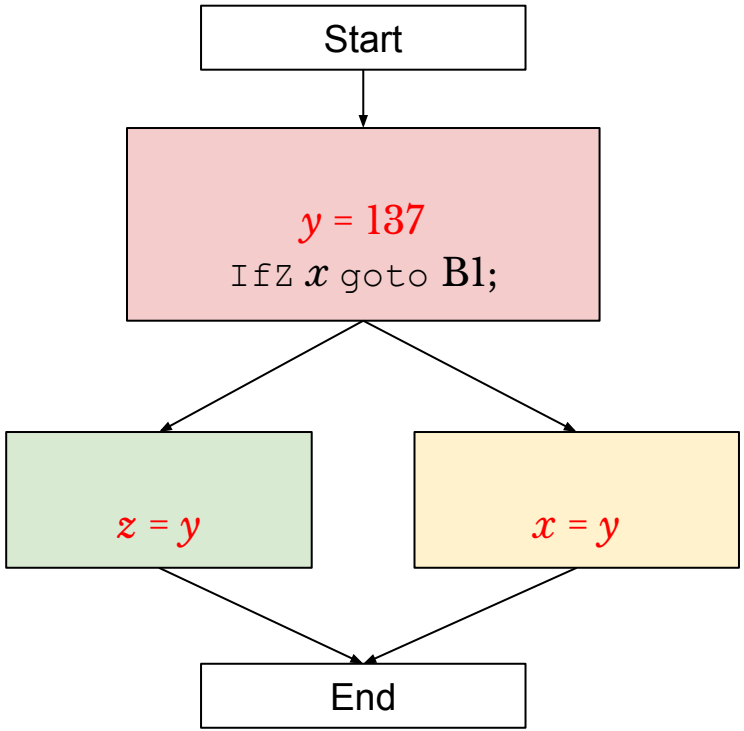
Local Optimization: Complete

```
int main()
{
        int x;
        int y;
        int z;
        y = 137;
        if (x == 0)
                z = y;
        else
                x = y;
}
```



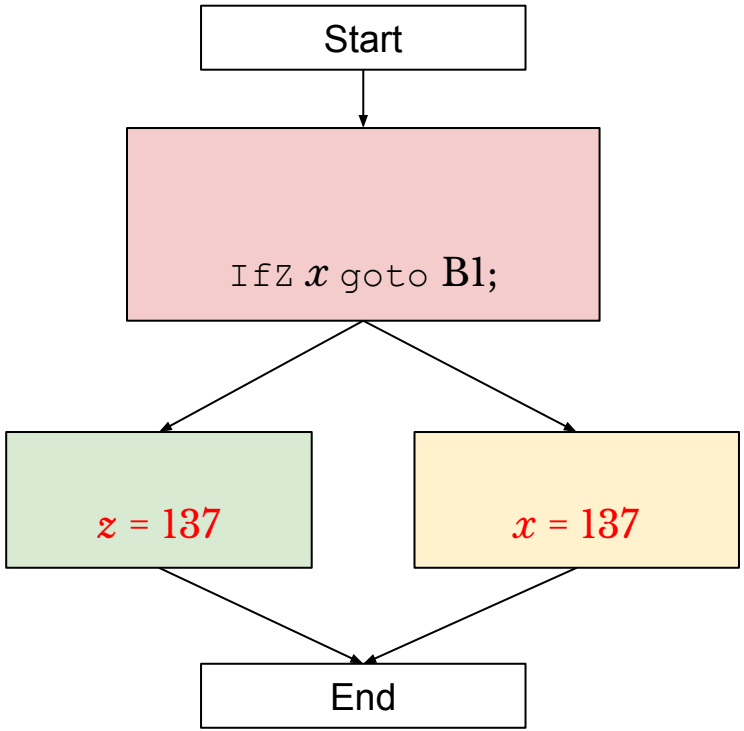
Global Optimization

```
int main()
{
        int x;
        int y;
        int z;
        y = 137;
        if (x == 0)
                z = y;
        else
                x = y;
}
```



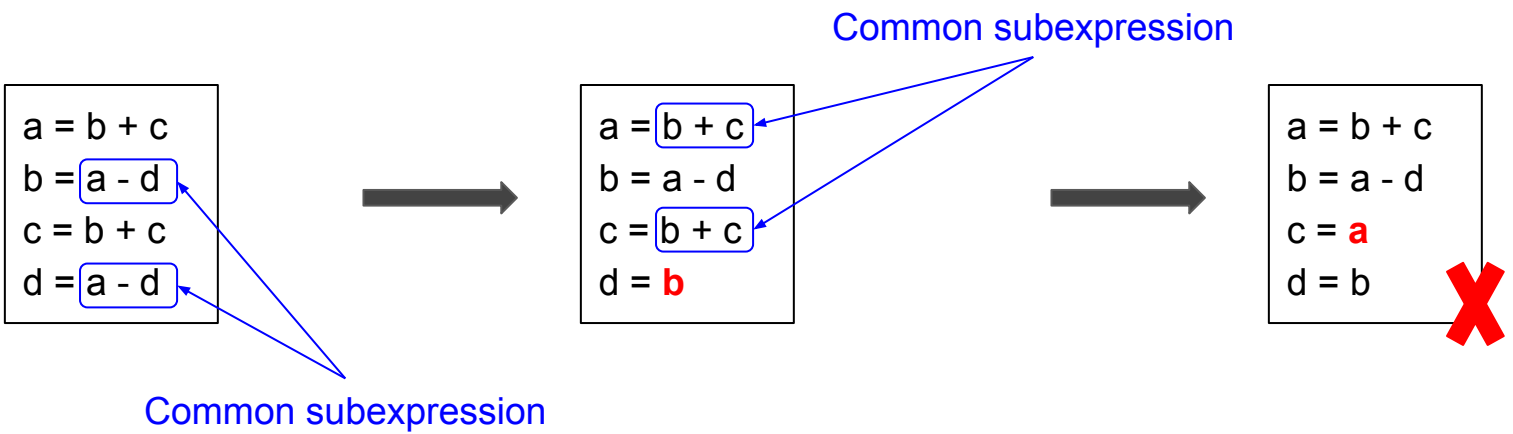
Global Optimization

```
int main()
{
        int x;
        int y;
        int z;
        y = 137;
        if (x == 0)
                z = y;
        else
                x = y;
}
```



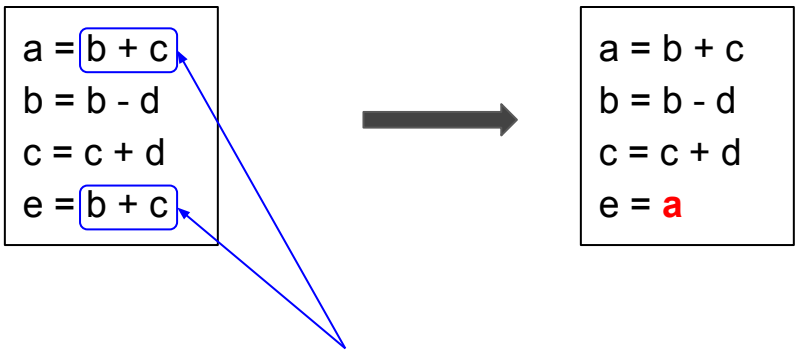
Local Optimization

- Common Subexpression Elimination
  - If we have x = a + b, and
    y = a + b as long as values of x, a, and b are not changed in between
    - We can write y = x
  - Example 1



Local Optimization

- Common Subexpression Elimination
  - Example 2

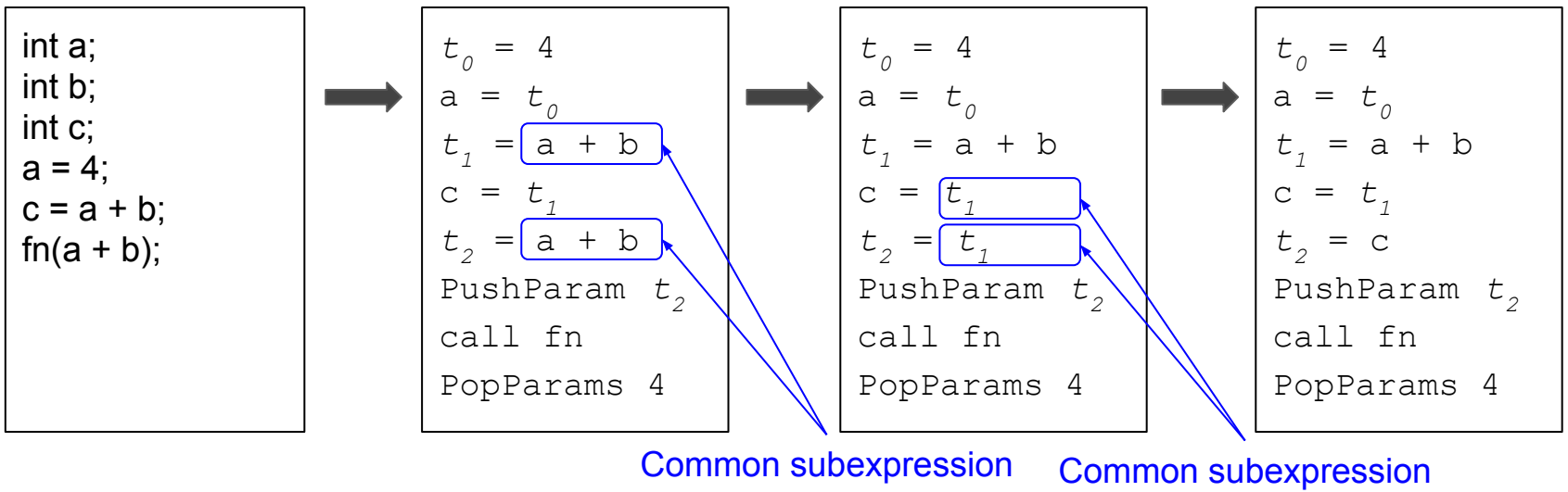


But the variables b & c have been modified in between!

However, observe that both the first and last instruction will result in same value.

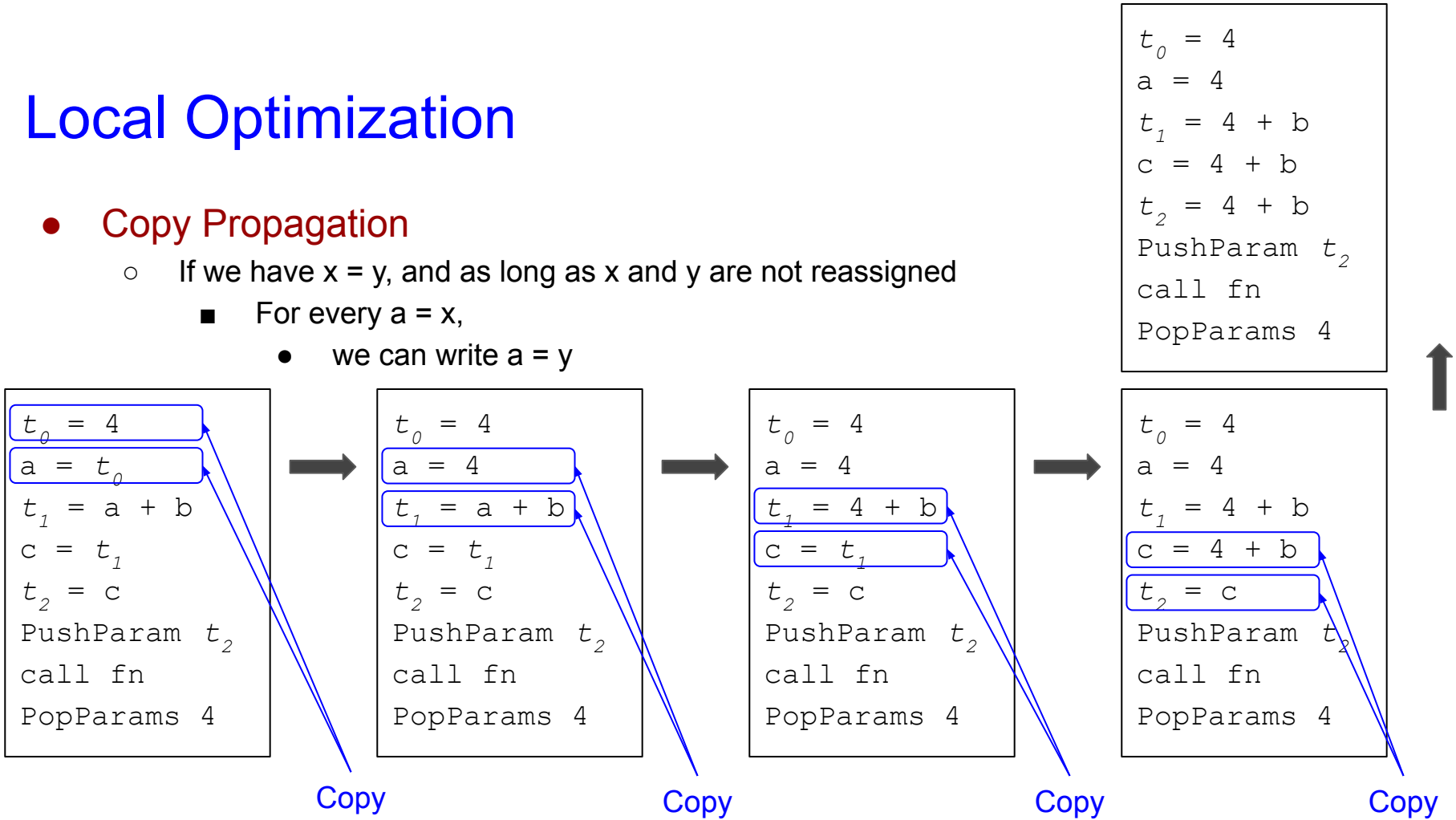
Local Optimization

- Common Subexpression Elimination
  - Example 3



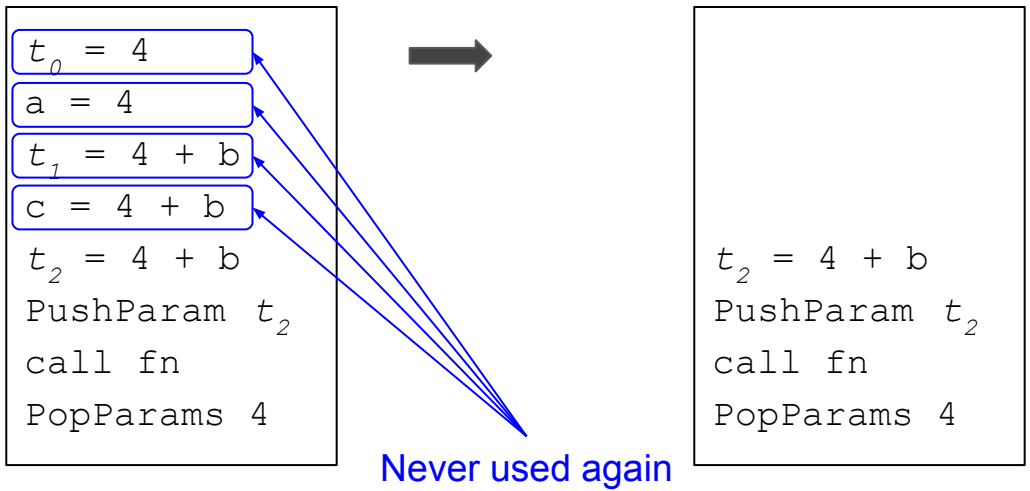
Local Optimization

- Copy Propagation
  - If we have x = y, and as long as x and y are not reassigned
    - For every a = x,
      - we can write a = y



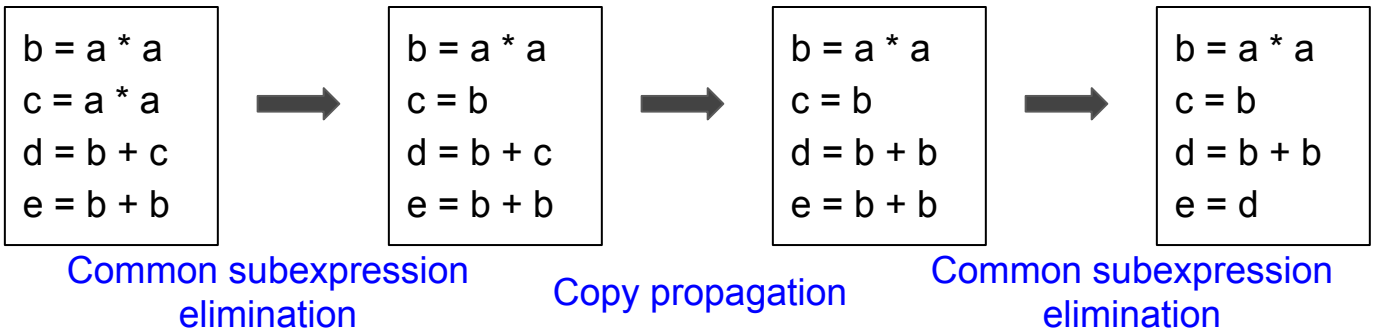
Local Optimization

- Dead Code Elimination
  - If a variable is never read anywhere after the assignment
    - Then, we can eliminate the assignment instruction



Local Optimization

- We may have to apply these optimizations again and again, i.e.,
  - One optimation step may leads to another issue



Implementations of Local optimization

Available Expression

- An *expression is available* if some variable holds the value of that expression.
- Steps
  - Initially, no expressions are available.
  - Whenever we execute a statement a = b + c:
    - Any expression holding a is invalidated.
    - The expression a = b + c becomes available.
  - Iterate across the basic block, beginning with the empty set of expressions and updating available expressions at each variable.
- Application
  - Common subexpression elimination: Replace an available expression by the variable holding its value.
  - Copy propagation, we replace the use of a variable by the available expression it holds.

Available Expression

```
{}
a = b;
{a = b;}
c = b;
{a = b; c = b;}
d = a + b;
{a = b; c = b; d = a + b;}
e = a + b;
{a = b; c = b; d = a + b; e = a + b;}
d = b;
{a = b; c = b; d = b; e = a + b;}
f = a + b;
{a = b; c = b; d = b; e = a + b; f = a + b;}
```

Common Subexpression Elimination

```
{}
a = b;
{a = b;}
c = b;
{a = b; c = b;}
d = a + b;
{a = b; c = b; d = a + b;}
e = a + b;
{a = b; c = b; d = a + b; e = a + b;}
d = b;
{a = b; c = b; d = b; e = a + b;}
f = a + b;
{a = b; c = b; d = b; e = a + b; f = a + b;}
```

Common Subexpression Elimination

```
{}
a = b;
{a = b;}
c = b;
{a = b; c = b;}
d = a + b;
{a = b; c = b; d = a + b;}
e = a + b;
{a = b; c = b; d = a + b; e = a + b;}
d = b;
{a = b; c = b; d = b; e = a + b;}
f = a + b;
{a = b; c = b; d = b; e = a + b; f = a + b;}
```

Common Subexpression Elimination

```
{}
a = b;
{a = b;}
c = a;
{a = b; c = b;}
d = a + b;
{a = b; c = b; d = a + b;}
e = a + b;
{a = b; c = b; d = a + b; e = a + b;}
d = b;
{a = b; c = b; d = b; e = a + b;}
f = a + b;
{a = b; c = b; d = b; e = a + b; f = a + b;}
```

Common Subexpression Elimination

```
{}
a = b;
{a = b;}
c = a;
{a = b; c = b;}
d = a + b;
{a = b; c = b; d = a + b;}
e = a + b;
{a = b; c = b; d = a + b; e = a + b;}
d = b;
{a = b; c = b; d = b; e = a + b;}
f = a + b;
{a = b; c = b; d = b; e = a + b; f = a + b;}
```

Common Subexpression Elimination

```
{}
a = b;
{a = b;}
c = a;
{a = b; c = b;}
d = a + b;
{a = b; c = b; d = a + b;}
e = d;
{a = b; c = b; d = a + b; e = a + b;}
d = b;
{a = b; c = b; d = b; e = a + b;}
f = a + b;
{a = b; c = b; d = b; e = a + b; f = a + b;}
```

Common Subexpression Elimination

```
{}
a = b;
{a = b;}
c = a;
{a = b; c = b;}
d = a + b;
{a = b; c = b; d = a + b;}
e = d;
{a = b; c = b; d = a + b; e = a + b;}
d = b;
{a = b; c = b; d = b; e = a + b;}
f = a + b;
{a = b; c = b; d = b; e = a + b; f = a + b;}
```

Common Subexpression Elimination

```
{}
a = b;
{a = b;}
c = a;
{a = b; c = b;}
d = a + b;
{a = b; c = b; d = a + b;}
e = d;
{a = b; c = b; d = a + b; e = a + b;}
d = a;
{a = b; c = b; d = b; e = a + b;}
f = a + b;
{a = b; c = b; d = b; e = a + b; f = a + b;}
```

Common Subexpression Elimination

```
{}
a = b;
{a = b;}
c = a;
{a = b; c = b;}
d = a + b;
{a = b; c = b; d = a + b;}
e = d;
{a = b; c = b; d = a + b; e = a + b;}
d = a;
{a = b; c = b; d = b; e = a + b;}
f = a + b;
{a = b; c = b; d = b; e = a + b; f = a + b;}
```

Common Subexpression Elimination

```
{}
a = b;
{a = b;}
c = a;
{a = b; c = b;}
d = a + b;
{a = b; c = b; d = a + b;}
e = d;
{a = b; c = b; d = a + b; e = a + b;}
d = a;
{a = b; c = b; d = b; e = a + b;}
f = e;
{a = b; c = b; d = b; e = a + b; f = a + b;}
```

Common Subexpression Elimination

```
a = b;

c = a;

d = a + b;

e = d;

d = a;

f = e;
```

Live Variables

- A *variable is live at a point* in a program if later in the program its value will be read before it is written again.
- Steps
  - Iterate across the statements in a basic block in reverse order.
  - Initially, some small set of values are known to be live (which ones depends on the particular
  - program).
  - When we see the statement a = b + c:
    - Just before the statement, a is not alive, since its value is about to be overwritten.
    - Just before the statement, both b and c are alive, since we're about to read their values.
- Application
  - Dead code elimination: Eliminating assignments to dead variables.

Live Variables

```
a = b;

c = a;

d = a + b;

e = d;

d = a;

f = e;
```

{ } → May contains some variables

Live Variables

```
a = b;

c = a;

d = a + b;

e = d;

d = a;
  {e}
f = e;
  {}
```

Live Variables

```
a = b;

c = a;

d = a + b;

e = d;
{a, e}
d = a;
  {e}
f = e;
  {}
```

Live Variables

```
a = b;

c = a;

d = a + b;
{a, d}
e = d;
{a, e}
d = a;
  {e}
f = e;
  {}
```

Live Variables

```
   a = b;

   c = a;
   {a, b}
d = a + b;
   {a, d}
   e = d;
   {a, e}
   d = a;
     {e}
   f = e;
     {}
```

Live Variables

```
a = b;
{a, b}
c = a;
{a, b}
d = a + b;
{a, d}
e = d;
{a, e}
d = a;
{e}
f = e;
{}
```

Live Variables

```
  {b}
a = b;
{a, b}
c = a;
{a, b}
d = a + b;
{a, d}
e = d;
{a, e}
d = a;
  {e}
f = e;
  {}
```

Dead Code Elimination

```
  {b}
a = b;
{a, b}
c = a;
{a, b}
d = a + b;
{a, d}
e = d;
{a, e}
d = a;
  {e}
f = e;
  { }     → f may or may not be here
```

Dead Code Elimination

```
  {b}
a = b;
{a, b}
c = a;
{a, b}
d = a + b;
{a, d}
e = d;
{a, e}
d = a;
  {e}
f = e;
  {}
```

Dead Code Elimination

```
  {b}
a = b;
{a, b}
c = a;
{a, b}
d = a + b;
{a, d}
e = d;
{a, e}
d = a;
  {e}


  {}
```

Dead Code Elimination

```
  {b}
a = b;
{a, b}
c = a;
{a, b}
d = a + b;
{a, d}
e = d;
{a, e}
d = a;
  {e}


  {}
```

Dead Code Elimination

```
  {b}
a = b;
{a, b}
c = a;
{a, b}
d = a + b;
{a, d}
e = d;
{a, e}

{e}

{}
```

Dead Code Elimination

```
  {b}
a = b;
{a, b}
c = a;
{a, b}
d = a + b;
{a, d}
e = d;
{a, e}

{e}

{}
```

Dead Code Elimination

```
  {b}
a = b;
{a, b}

{a, b}
d = a + b;
{a, d}
e = d;
{a, e}

{e}

{}
```

Dead Code Elimination

```
a = b;


d = a + b;

e = d;
```

Global Optimization

- Global common subexpression elimination
- Global copy propagation
- Global dead code elimination
- Loop optimizations
  - Code Motion
    - If a is not updated within loop

$$\text{while } (i <= a - 2)$$
$$\Rightarrow \quad t = a - 2;$$
$$\text{while } (i <= t)$$

  - Strength reduction
    - If i is a loop variable

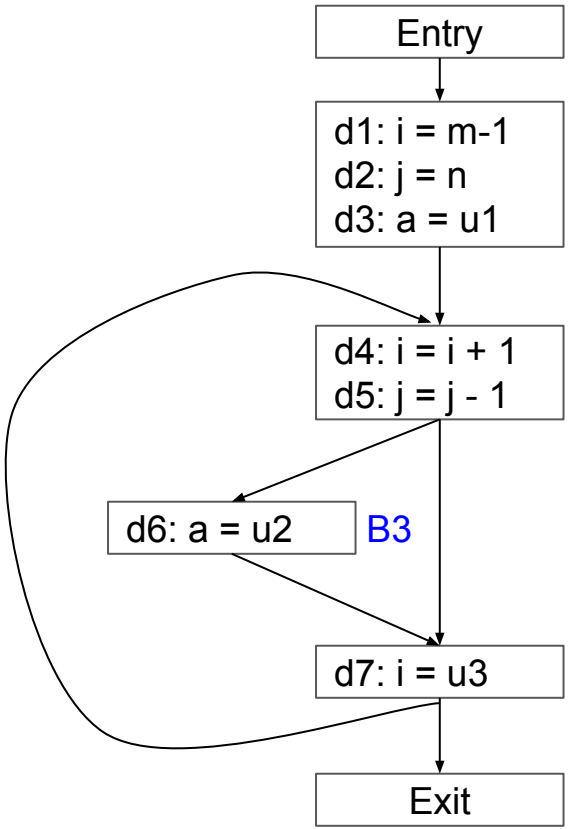
$$a = 3 * i;$$
$$\Rightarrow \quad t = t + 3;$$
$$a = t;$$

Reaching definition

- Point
  - For each statement in flow graph, we have a point p_1 just before it and a point p_2 right after it.
- Path (or Execution path)
  - A path from point p_1 to point p_n is the sequence of points $p_1, p_2, ..., p_i$ such that $i = 1, 2, ..., n-1$
- Definition
  - A definition of a variable x is a statement that assigns a value to x.
- Reaching definition
  - A definition d ***reaches*** a point p if there is a path from the point immediately following d to p, such that d is ***not killed*** along the path.
    - A definition of a variable x is ***killed*** if there is any other definition of x along the path.

Reaching definition

The *gen* set: All definitions inside the block that are visible immediately after the block.
The *kill* set: Union of all definitions killed by the individual statements.



gen_{B1} = {d1, d2, d3}
$kill_{B1}$ = {d4, d5, d6, d7}

gen_{B2} = {d4, d5}
$kill_{B2}$ = {d1, d2, d7}

gen_{B3} = {d6}
$kill_{B3}$ = {d3}

gen_{B4} = {d7}
$kill_{B4}$ = {d1, d4}

Summary

- Objective and Motivation of code optimization
- Basic blocks
- Control Flow Graph
- Local and Global optimizations
- Constant Folding
- Strength Reduction
- Common Subexpression Elimination
- Copy Propagation
- Dead Code Elimination
- Code Motion
- Available expression, Live variables, and Reaching definitions