IR Optimization

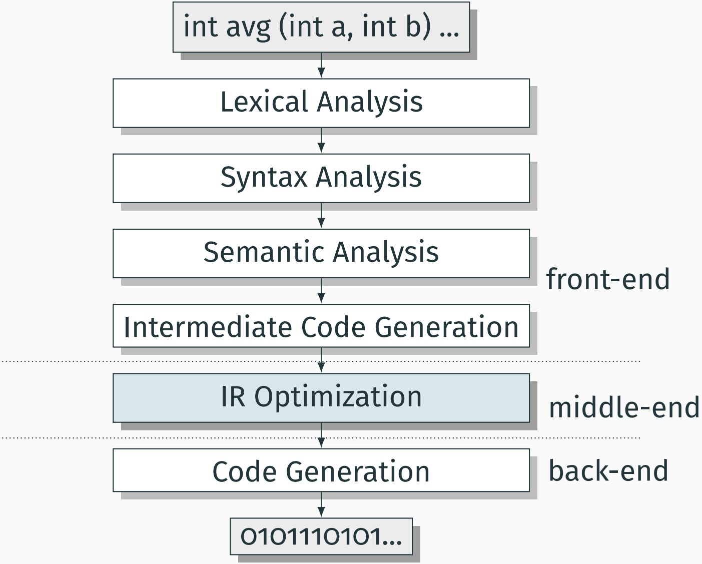
Ronghui Gu

Spring 2024

Columbia University

* Course website: https://verigu.github.io/4115Spring2024/

IR Optimization



IR Optimization

Goal

- Runtime
- Memory usage
- Power Consumption

Sources?

Optimizations from IR Generation

C code:

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

Three-Address:

```
_t0 = x + x;
_t1 = y;
b1 = _t0 < _t1;
_t2 = x + x;
_t3 = y;
b2 = _t2 == _t3;
_t4 = x + x;
_t5 = y;
b3 = _t5 < _t4;
```

Optimizations from IR Generation

C code:

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

Three-Address:

```
_t0 = x + x;
_t1 = y;
b1 = _t0 < _t1;
_t2 = x + x;
_t3 = y;
b2 = _t2 == _t3;
_t4 = x + x;
_t5 = y;
b3 = _t5 < _t4;
```

Optimizations from IR Generation

C code:

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

Three-Address:

```
_t0 = x + x;
_t1 = y;
b1 = _t0 < _t1;


b2 = _t0 == _t1;


b3 = _t0 < _t1;
```

Optimizations from Lazy Coders

C code:

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

Three-Address:

```
_L0:
    _t0 = y + z;
    _t1 = x < _t0;
    bz _L1 _t1;
    x = x - y;
    jmp _L0;
_L1:
```

Optimizations from Lazy Coders

C code:

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

Three-Address:

```
_L0:
    _t0 = y + z;
    _t1 = x < _t0;
    bz _L1 _t1;
    x = x - y;
    jmp _L0;
_L1:
```

Optimizations from Lazy Coders

C code:

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

Three-Address:

```
    _t0 = y + z;
_L0:
    _t1 = x < _t0;
    bz _L1 _t1;
    x = x - y;
    jmp _L0;
_L1:
```

IR Optimization Discussion

Optimal? Undecidable!

Soundness: semantics-preserving

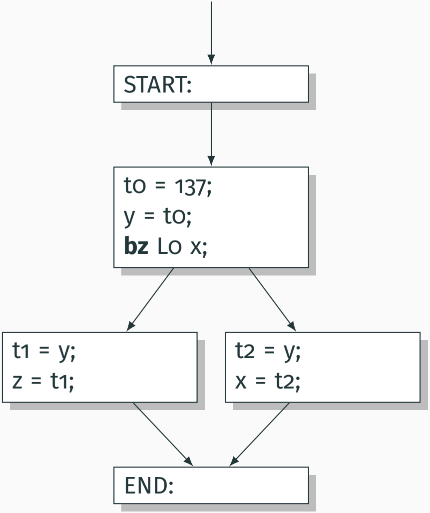
IR optimization v.s. code optimization:

`x * 0.5` ⇒ `x » 1`

Local optimization v.s. global optimization

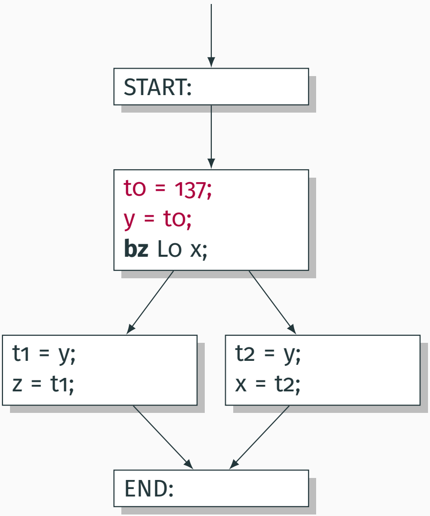
Local Optimization

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



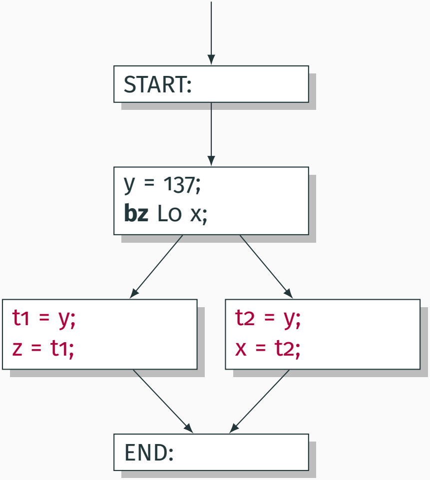
Local Optimization

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



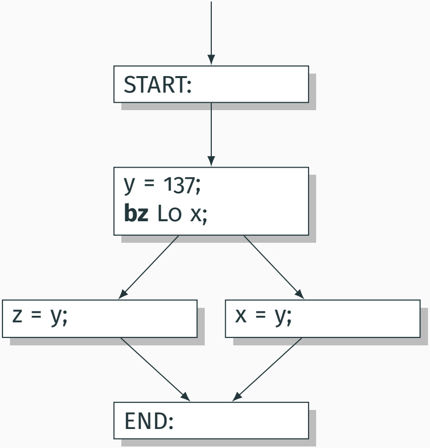
Local Optimization

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



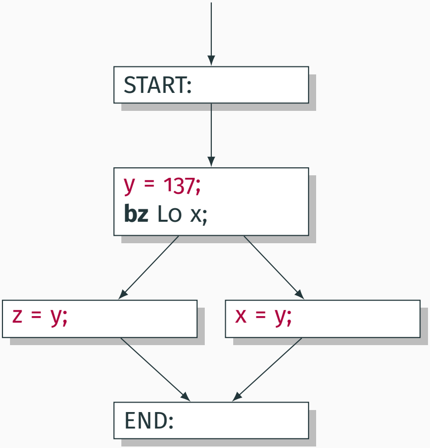
Local Optimization

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



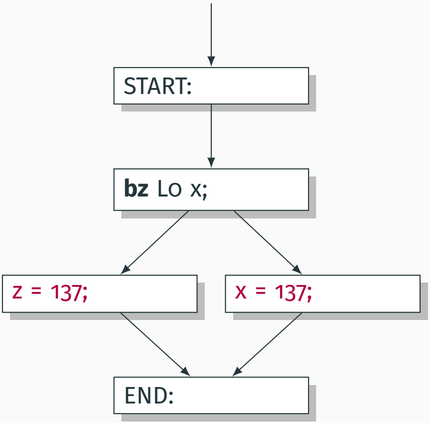
Global Optimization

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



Global Optimization

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



Local Optimization

Common Subexpression Elimination

Purpose: remove the duplicate computation of "`a op b`" in Three-Address code.

```
v1 = a op b

.  .  .

v2 = a op b
```

Common Subexpression Elimination

Purpose: remove the duplicate computation of “`a op b`” in Three-Address code.

```
v1 = a op b

.  .  .

v2 = a op b
```

If values of **`v1`**, **`a`**, and **`b`** have not changed, rewrite the code:

```
v1 = a op b

.  .  .

v2 = v1
```

Common Subexpression Elimination

C code:

```
int a;
int b;
int c;
a = 4;
c = a + b;
f(a + b);
```

Three-address code:

```
_t0 = 4;
a = _t0;
_t1 = a + b;
c = _t1;
_t2 = a + b;
param _t2;
call f;
```

Common Subexpression Elimination

C code:

```
int a;
int b;
int c;
a = 4;
c = a + b;
f(a + b);
```

Three-address code:

```
_t0 = 4;
a = _t0;
_t1 = a + b;
c = _t1;
_t2 = a + b;
param _t2;
call f;
```

Common Subexpression Elimination

C code:

```
int a;
int b;
int c;
a = 4;
c = a + b;
f(a + b);
```

Three-address code:

```
_t0 = 4;
a = _t0;
_t1 = a + b;
c = _t1;
_t2 = _t1;?
param _t2;
call f;
```

Do we need to replace `_t1` with `c`?

Common Subexpression Elimination

C code:

```
int a;
int b;
int c;
a = 4;
c = a + b;
f(a + b);
```

Three-address code:

```
_t0 = 4;
a = _t0;
_t1 = a + b;
c = _t1;
_t2 = _t1;?
param _t2;
call f;
```

Do we need to replace `_t1` with `c`? NO!

Copy Propagation

If we have

```
v1 = v2
```

then as long as **`v1`** and **`v2`** have not changed, we can rewrite

```
a = ...  v1 ...
```

as

```
a = ...  v2 ...
```

Copy Propagation

C code:

```
int a;
int b;
int c;
a = 4;
c = a + b;
f(a + b);
```

Three-address code:

```
_t0 = 4;
a = _t0;
_t1 = a + b;
c = _t1;
_t2 = _t1;
param _t2;
call f;
```

Copy Propagation

C code:

```
int a;
int b;
int c;
a = 4;
c = a + b;
f(a + b);
```

Three-address code:

```
_t0 = 4;
a = 4;
_t1 = a + b;
c = _t1;
_t2 = _t1;
param _t2;
call f;
```

Copy Propagation

C code:

```c
int a;
int b;
int c;
a = 4;
c = a + b;
f(a + b);
```

Three-address code:

```
_t0 = 4;
a = 4;
_t1 = a + b;
c = _t1;
_t2 = _t1;
param _t2;
call f;
```

Copy Propagation

C code:

```
int a;
int b;
int c;
a = 4;
c = a + b;
f(a + b);
```

Three-address code:

```
_t0 = 4;
a = 4;
_t1 = 4 + b;
c = _t1;
_t2 = _t1;
param _t2;
call f;
```

Copy Propagation

C code:

```
int a;
int b;
int c;
a = 4;
c = a + b;
f(a + b);
```

Three-address code:

```
_t0 = 4;
a = 4;
_t1 = 4 + b;
c = _t1;
_t2 = _t1;
param _t2;
call f;
```

Copy Propagation

C code:

```
int a;
int b;
int c;
a = 4;
c = a + b;
f(a + b);
```

Three-address code:

```
_t0 = 4;
a = 4;
_t1 = 4 + b;
c = _t1;
_t2 = _t1;
param _t1;
call f;
```

Dead Code Elimination

An assignment to a variable **v** is called dead if its value is never read anywhere.

Dead Code Elimination

C code:

```
int a;
int b;
int c;
a = 4;
c = a + b;
f(a + b);
```

Three-address code:

```
_t0 = 4;
a = 4;
_t1 = 4 + b;
c = _t1;
_t2 = _t1;
param _t1;
call f;
```

Dead Code Elimination

C code:

```
int a;
int b;
int c;
a = 4;
c = a + b;
f(a + b);
```

Three-address code:

```
_t0 = 4;
a = 4;
_t1 = 4 + b;
c = _t1;
_t2 = _t1;
param _t1;
call f;
```

Dead Code Elimination

C code:

```
int a;
int b;
int c;
a = 4;
c = a + b;
f(a + b);
```

Three-address code:

```
a = 4;
_t1 = 4 + b;
c = _t1;
_t2 = _t1;
param _t1;
call f;
```

Dead Code Elimination

C code:

```
int a;
int b;
int c;
a = 4;
c = a + b;
f(a + b);
```

Three-address code:

```
_t1 = 4 + b;
c = _t1;
_t2 = _t1;
param _t1;
call f;
```

Dead Code Elimination

C code:

```
int a;
int b;
int c;
a = 4;
c = a + b;
f(a + b);
```

Three-address code:

```
_t1 = 4 + b;
_t2 = _t1;
param _t1;
call f;
```

Dead Code Elimination

C code:

```
int a;
int b;
int c;
a = 4;
c = a + b;
f(a + b);
```

Three-address code:

```
_t1 = 4 + b;
param _t1
call f;
```

For Comparison

C code:

```
int a;
int b;
int c;
a = 4;
c = a + b;
f(a + b);
```

Three-address code:

```
_t0 = 4;
a = _t0;
_t1 = a + b;
c = _t1;
_t2 = a + b;
param _t2;
call f;
```

Optimized code:

```
_t1 = 4 + b;
param _t1;
call f;
```

Other Types of Local Optimization

Arithmetic simplication:

- e.g., rewrite `x = 4 * a` as `x = a « 2`

Constant folding:

- e.g., rewrite `x = 4 * 5` as `x = 20`

Implementing Local Optimization

Optimizations and Analyses

Most optimizations are only possible given some analysis of the program's behavior.

In order to implement an optimization, we will talk about the corresponding program analyses.

Available Expressions

- Both common subexpression elimination and copy propagation depend on an analysis of the available expressions in a program.

Available Expressions

- Both common subexpression elimination and copy propagation depend on an analysis of the available expressions in a program.
- An expression is called available if some variable in the program holds the value of that expression.

Available Expressions

- Both common subexpression elimination and copy propagation depend on an analysis of the available expressions in a program.
- An expression is called available if some variable in the program holds the value of that expression.
- In common subexpression elimination, we replace an available expression requiring computation by the variable holding its value.

Available Expressions

- Both common subexpression elimination and copy propagation depend on an analysis of the available expressions in a program.
- An expression is called available if some variable in the program holds the value of that expression.
- In common subexpression elimination, we replace an available expression requiring computation by the variable holding its value.
- In copy propagation, we replace the use of a variable by the available expression it holds that does not require computation.

Finding Available Expressions

- Initially, no expressions are available

Finding Available Expressions

- Initially, no expressions are available
- Whenever we execute a statement

  `a = expr`

  - Any expression holding **`a`** is invalidated.
  - The expression **`a = expr`** becomes available.

Finding Available Expressions

- Initially, no expressions are available
- Whenever we execute a statement

  `a = expr`

  - Any expression holding **`a`** is invalidated.
  - The expression **`a = expr`** becomes available.
- Algorithm: Iterate across the basic block, beginning with the empty set of expressions and updating available expressions at each variable.

Example: Available Expressions

{ }

```
a = b;

c = b;

d = a + b;

e = a + b;

d = b;

f = a + b;
```

Example: Available Expressions

{ }

a = b;

{ a = b }

c = b;

d = a + b;

e = a + b;

d = b;

f = a + b;

Example: Available Expressions

{ }

a = b;

{ a = b }

c = b;

{ a = b, c = b }

d = a + b;

e = a + b;

d = b;

f = a + b;

Example: Available Expressions

{ }

a = b;

{ a = b }

c = b;

{ a = b, c = b }

d = a + b;

{ a = b, c = b, d = a + b }

e = a + b;

d = b;

f = a + b;

Example: Available Expressions

{ }

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;

f = a + b;

Example: Available Expressions

{ }

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;

Example: Available Expressions

{ }

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 }

Example: 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 }

Example: Common Subexpression Elimination

{ }

a = b;

{ a = b }

c = b;

{ 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 }

Example: Common Subexpression Elimination

{ }

a = b;

{ a = b }

c = b;

{ 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 = e;

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

Example: Common Subexpression Elimination

```
a = b;

c = b;

d = a + b;

e = d;

d = b;

f = e;
```

Example: Copy Propagation

{ }

a = b;

{ a = b }

c = b;

{ a = b, c = b }

d = a + b;

{ a = b, c = b, d = a + b }

e = d;

d = b;

f = e;

Example: Copy Propagation

{ }

a = b;

{ a = b }

c = b;

{ a = b, c = b }

d = a + b;

{ a = b, c = b, d = a + b }

e = d;

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

d = b;

f = e;

Example: Copy Propagation

{ }

a = b;

{ a = b }

c = b;

{ a = b, c = b }

d = a + b;

{ a = b, c = b, d = a + b }

e = d;

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

d = b;

{ a = b, c = b, d = b }

f = e;

Example: Copy Propagation

{ }

a = b;

{ a = b }

c = b;

{ a = b, c = b }

d = a + b;

{ a = b, c = b, d = a + b }

e = d;

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

d = b;

{ a = b, c = b, d = b }

f = e;

{ a = b, c = b, d = b, f = e }

Example: Copy Propagation

{ }

a = b;

{ a = b }

c = b;

{ a = b, c = b }

d = a + b;

{ a = b, c = b, d = a + b }

e = d;

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

d = b;

{ a = b, c = b, d = b }

f = e;

{ a = b, c = b, d = b, f=e }

Example: Copy Propagation

{ }

a = b;

{ a = b }

c = b;

{ a = b, c = b }

d = b + b;

{ a = b, c = b, d = a + b }

e = d;

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

d = b;

{ a = b, c = b, d = b }

f = e;

{ a = b, c = b, d = b, f=e }

Live Variables

- The analysis corresponding to dead code elimination is called liveness analysis.

Live Variables

- The analysis corresponding to dead code elimination is called liveness analysis.
- A variable is live at a point in a program if later in the program its value will be read before it is written to again.

Live Variables

- The analysis corresponding to dead code elimination is called liveness analysis.
- A variable is live at a point in a program if later in the program its value will be read before it is written to again.
- Dead code elimination works by computing liveness for each variable, then eliminating assignments to dead variables.

Computing Live Variables

- To know if a variable will be used at some point, we iterate across the statements in a block in reverse order.

Computing Live Variables

- To know if a variable will be used at some point, we iterate across the statements in a block in reverse order.
- Initially, some small set of values are known to be live (which ones depends on the particular program).

Computing Live Variables

- To know if a variable will be used at some point, we iterate across the statements in a 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 op c`**
  - If **a** is not alive after the statement, skip it.
  - Otherwise, If **a** is alive after the statement
    - 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.
  - (what if we have **`a = a op b`**?)

Example: Liveness Analysis

```
a = b;

c = a;

d = b + d;

e = d;

d = b;

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

Example: Liveness Analysis

```
a = b;

c = a;

d = b + d;

e = d;

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

Example: Liveness Analysis

```
a = b;

c = a;

d = b + d;

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

Example: Liveness Analysis

```
a = b;

c = a;

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

Example: Liveness Analysis

```
a = b;

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

Example: Liveness Analysis

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

Example: Liveness Analysis

{ b, d }

a = b;

{ b, d }

c = a;

{ b, d }

d = b + d;

{ b, d }

e = d;

{ b, e }

d = b;

{ d, e }

f = e + c;

{ d, e }

Example: Dead Code Elimination

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

Example: Dead Code Elimination

{ b, d }

d = b + d;

e = d;

d = b;

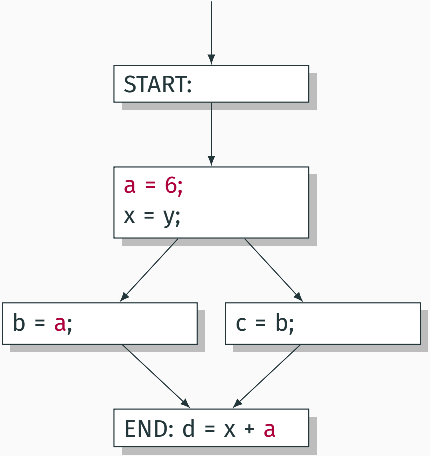
{ d, e }

Global Optimization

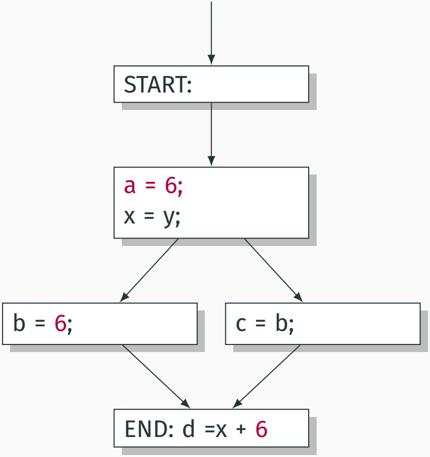
Global Constant Propagation

Replace each variable that is known to be a constant value with the constant.

Global Constant Propagation



Global Constant Propagation



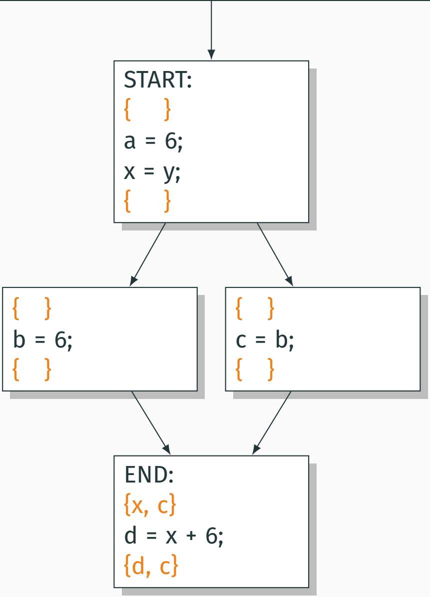
Global Dead Code Elimination

- Local dead code elimination needed to know what variables were live on exit from a basic block.
- This information can only be computed as part of a global analysis.
- How do we modify our liveness analysis to handle a CFG?

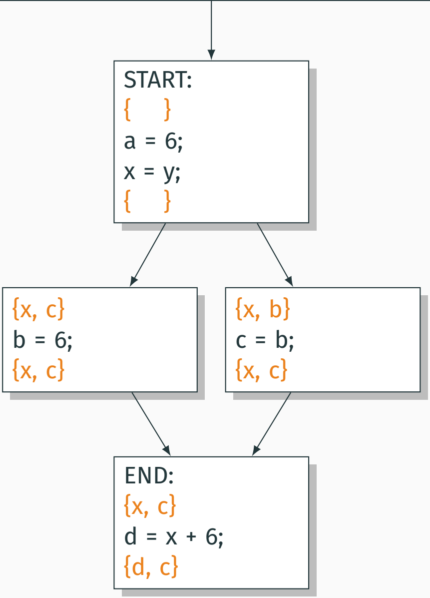
Global Dead Code Elimination

- In a local analysis, each statement has exactly one predecessor.
- In a global analysis, each statement may have multiple predecessors.
- A global analysis must combine information from all predecessors of a basic block.

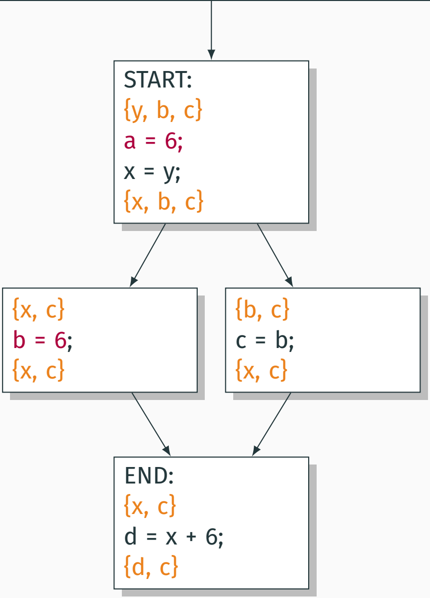
Global Dead Code Elimination



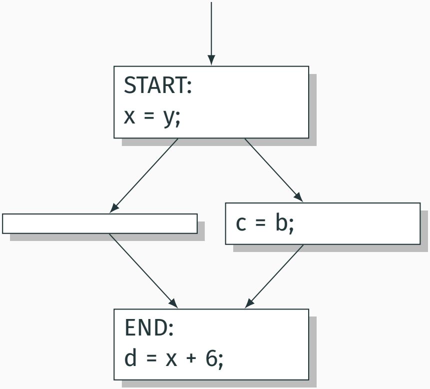
Global Dead Code Elimination



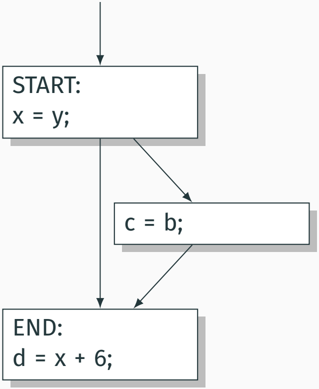
Global Dead Code Elimination



Global Dead Code Elimination



Global Dead Code Elimination



Global Dead Code Elimination with Loops

- Up to this point, we've considered loop-free CFGs, which have only finitely many possible paths.

Global Dead Code Elimination with Loops

- Up to this point, we've considered loop-free CFGs, which have only finitely many possible paths.
- Not all possible loops in a CFG can be realized in the actual program.
- Sound approximation: Assume that every possible path through the CFG corresponds to a valid execution.
  - Includes all realizable paths, but some additional paths as well.
  - May make our analysis less precise (but still sound).
  - Makes the analysis feasible; we'll see how later.

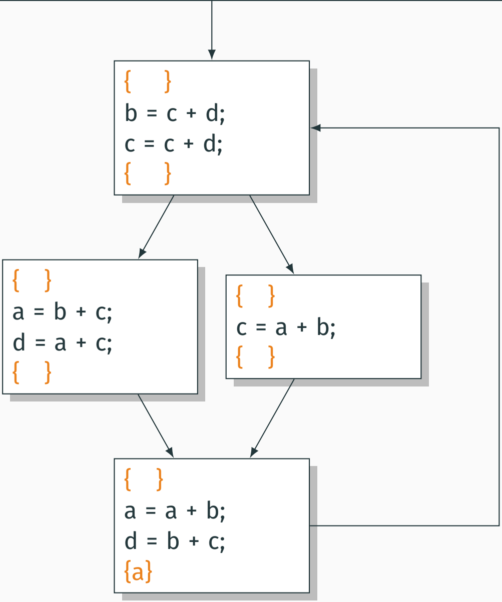
Global Dead Code Elimination with Loops

- In a local analysis, there is always a well-defined first statement to begin processing.
- In a global analysis with loops, every basic block might depend on every other basic block.

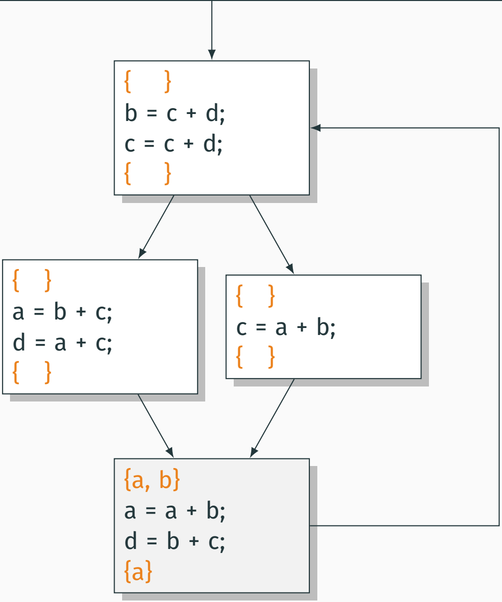
Global Dead Code Elimination with Loops

- In a local analysis, there is always a well-defined first statement to begin processing.
- In a global analysis with loops, every basic block might depend on every other basic block.
- To fix this, we need to assign initial values to all of the blocks in the CFG

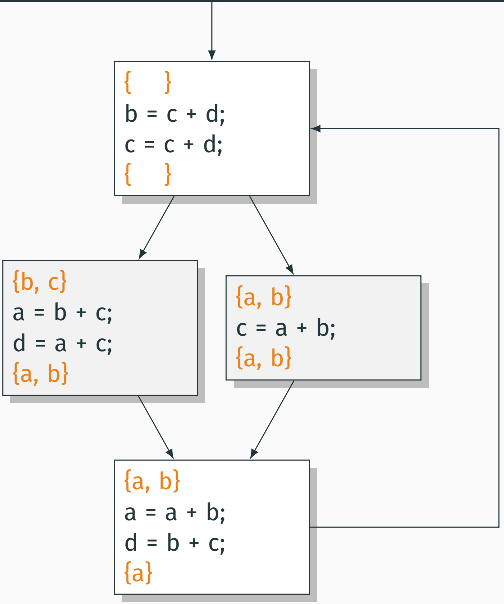
Global Dead Code Elimination with Loops



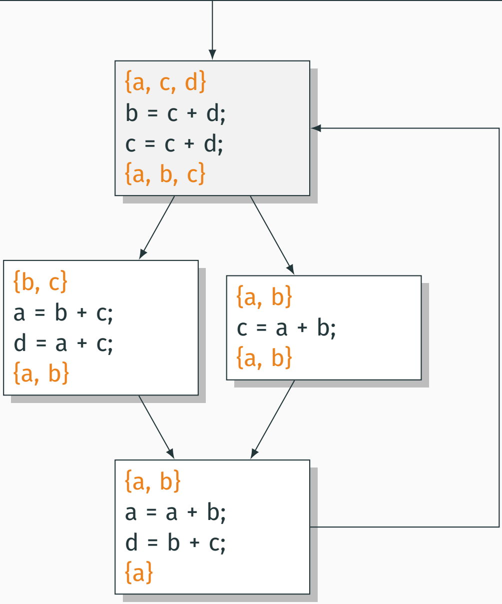
Global Dead Code Elimination with Loops



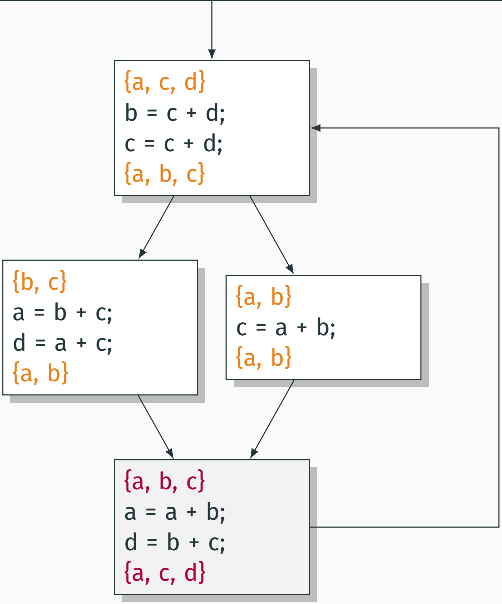
Global Dead Code Elimination with Loops



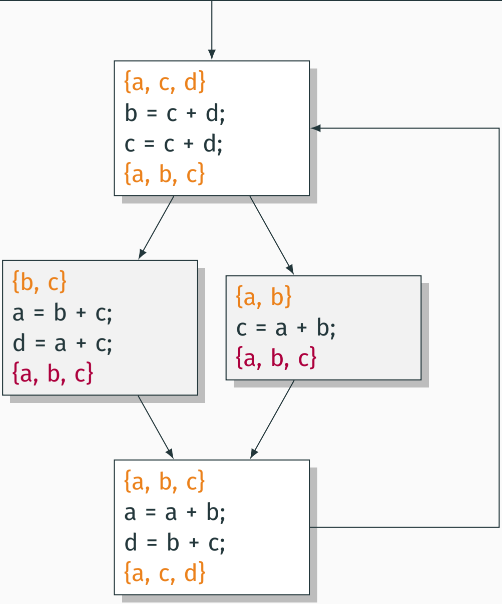
Global Dead Code Elimination with Loops



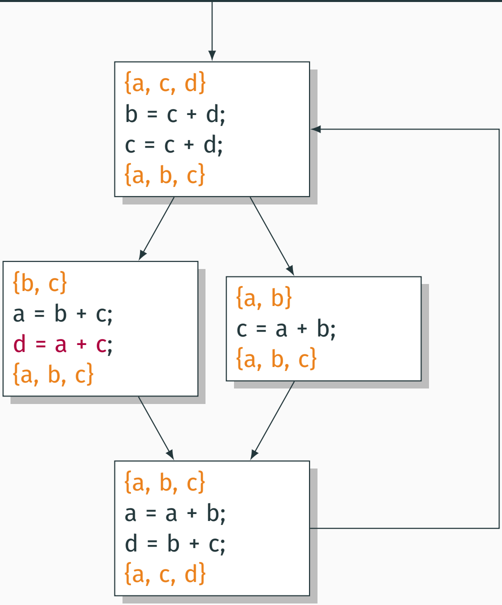
Global Dead Code Elimination with Loops



Global Dead Code Elimination with Loops



Global Dead Code Elimination with Loops



Global Dead Code Elimination with Loops