# IR Optimization

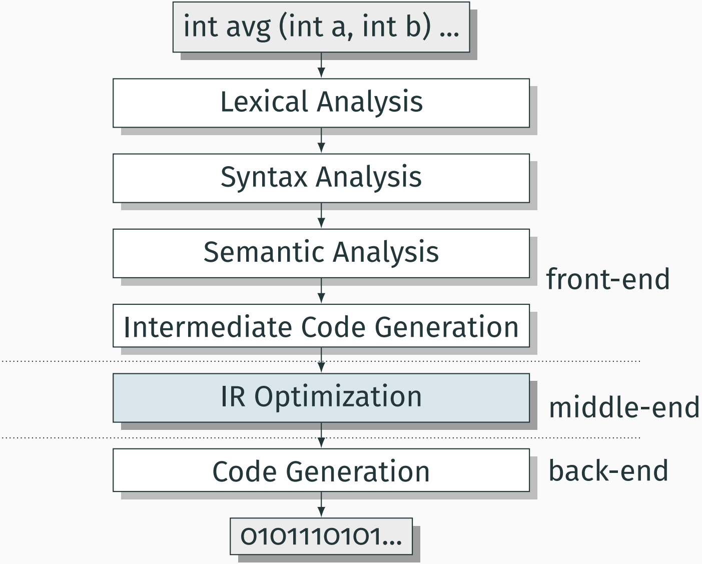
Ronghui Gu

Spring 2020

Columbia University

* Course website: https://www.cs.columbia.edu/ rgu/courses/4115/spring2019

# IR Optimization



int avg (int a, int b) ...

Lexical Analysis

Syntax Analysis

Semantic Analysis

front-end

Intermediate Code Generation

IR Optimization

middle-end

Code Generation

back-end

0101110101...

# 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

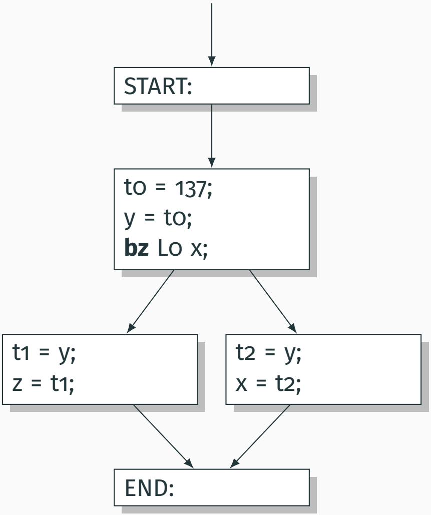
**IR optimization v.s. code optimization:**

`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;
}
```



START:

```
t0 = 137;
y = t0;
bz L0 x;
```

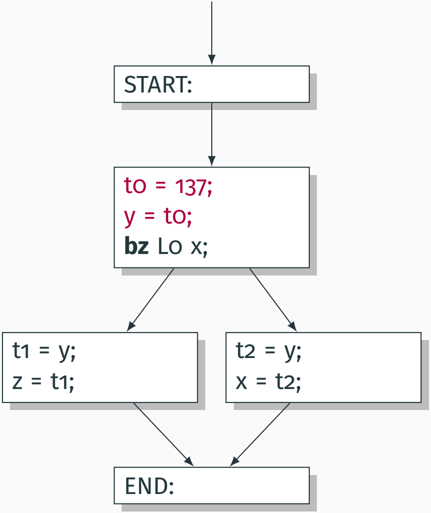
```
t1 = y;
z = t1;
```

```
t2 = y;
x = t2;
```

END:

# Local Optimization

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



START:

t0 = 137;
y = t0;
**bz** L0 x;

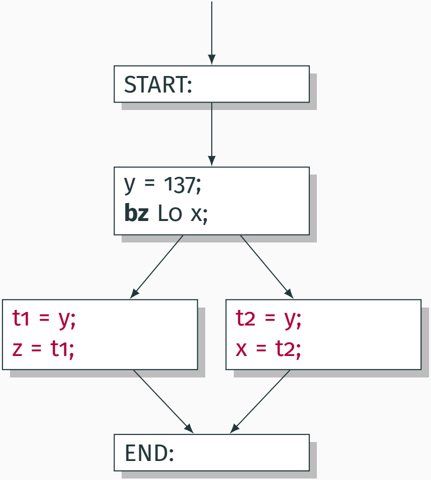
t1 = y;
z = t1;

t2 = y;
x = t2;

END:

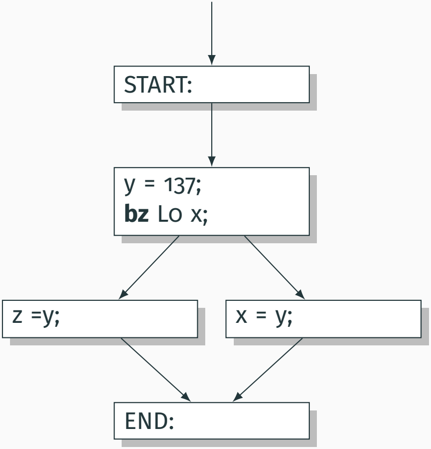
# Local Optimization

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



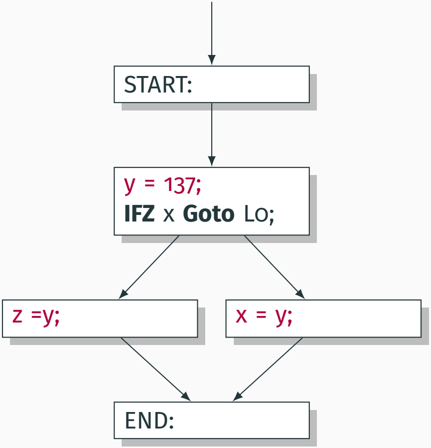
# Local Optimization

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



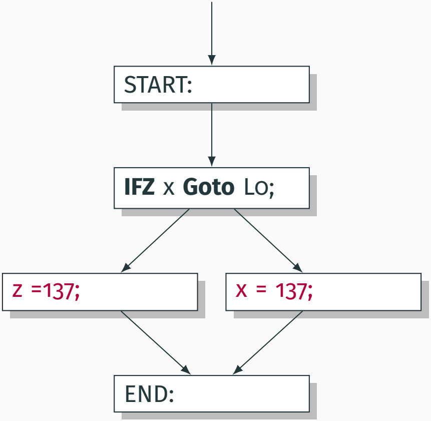
# Global Optimization

```
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;
}
```



START:

**IFZ** x **Goto** L0;

z =137;

x = 137;

END:

# Local Optimization

# Common Subexpression Elimination

```
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;
```

# 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:

```
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:

```
_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;
```

# Computing Live Variables (for dead code elimination)

Initially, some small set of values are known to be live.

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.
- what if we have `a = b + a`?

# Computing Live Variables (for dead code elimination)

```
  a = b;
  c = a;
d = a + b;
  e = d;

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

# Computing Live Variables (for dead code elimination)

```
a = b;
c = a;
d = a + b;
e = d;

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

# Computing Live Variables (for dead code elimination)

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

# Computing Live Variables (for dead code elimination)

a = b;

c = a;

d = a + b;

~~e = d;~~

{ a, b }

d = a;

~~f = e;~~

{b, d}

# Computing Live Variables (for dead code elimination)

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

# Computing Live Variables (for dead code elimination)

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

# Computing Live Variables (for dead code elimination)

{ b }

a = b;

~~c = a;~~

~~d = a + b;~~

~~e = d;~~

{ a, b }

d = a;

~~f = e;~~

{b, d}

# Computing Available Expressions

An expression is called available if some variable in the program holds the value of that expression.

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

Initially, no expressions are available.

When we see the statement `a = b + c`:

- Any expression holding `a` is invalidated.
- The expression `a = b + c` becomes available.

# Computing Available Expressions

```
       { }
    a = b;

    c = b;

  d = a + b;

  e = a + b;

    d = b;

  f = a + b;
```

# Computing Available Expressions

```
  { }
a = b;
{ a=b }
c = b;

d = a + b;

e = a + b;

d = b;

f = a + b;
```

# Computing Available Expressions

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

e = a + b;

   d = b;

f = a + b;
```

# Computing 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;
```

# Computing 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;
```

# Computing 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;
```

# Computing 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 }
```

# Computing Available Expressions

```
{ }
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 }
```

# Other Types of Local Optimization

Arithmetic simplication:

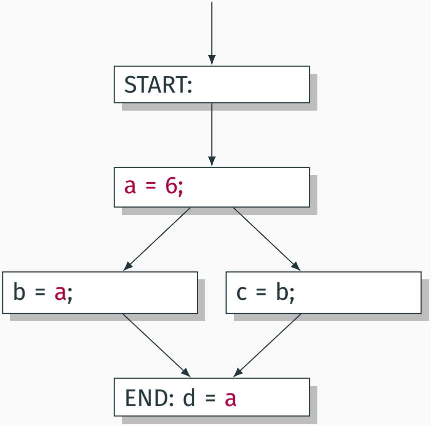
- e.g., rewrite `x = 4 * a` as `x = a « 2`

Constant folding:

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

# Global Optimization

# Global Constant Propagation



START:

a = 6;

b = a;

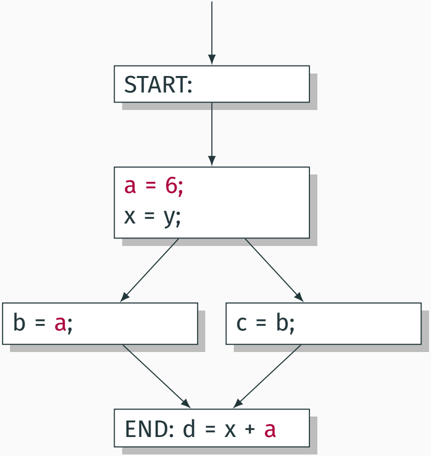
c = b;

END: d = a

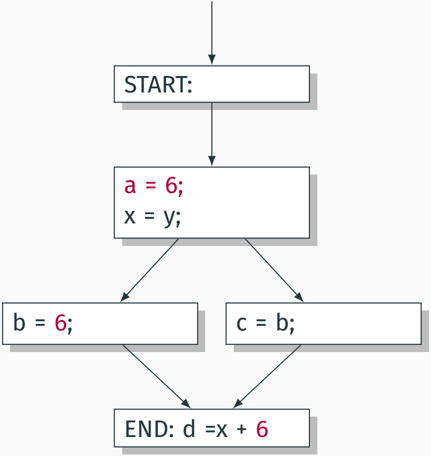
# Global Constant Propagation

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

# Global Constant Propagation



# Global Constant Propagation



START:

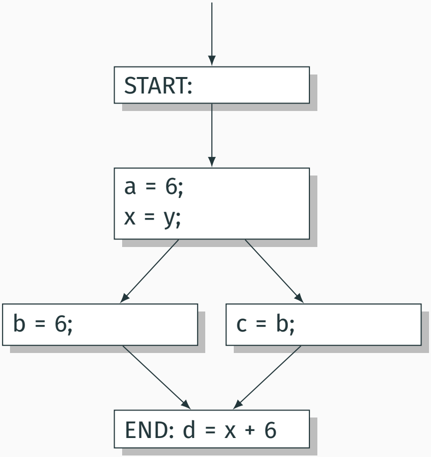
a = 6;
x = y;

b = 6;

c = b;

END: d =x + 6

# Global Dead Code Elimination



START:

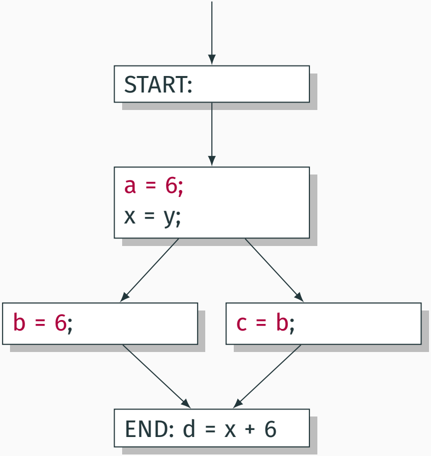
a = 6;
x = y;

b = 6;

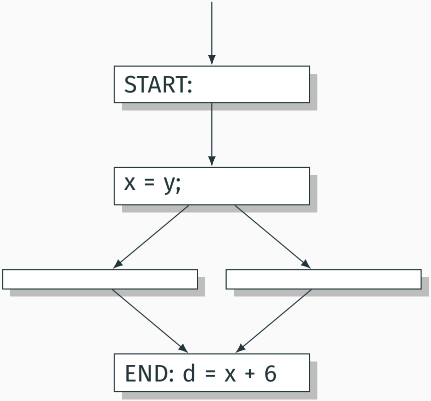
c = b;

END: d = x + 6

# Global Dead Code Elimination



# Global Dead Code Elimination


START:
x = y;
END: d = x + 6