COP-3402 Systems Software
11.12 Tue

P.S.: From this lecture on, the topics will not be implemented. Rest of the class will be advanced topics in compilers and program analysis.

IR Optimization
Goal: to improve IR generated by the previous step to take better advantage of resources. Storing and reading from memory and from the same memory location one line after another or having additional branches that you do not consider whether a branch is true or false. When we have the code generation algorithm, we could have to code better or/and faster for higher performance.

Our IR can be improved because;
(i) it introduces redundancy; high-level language features into IR often introduces subcomputations.
(ii) programmers are lazy; a loop can often be factored out of the loop.

Optimizations from IR Generation:

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

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

_t2 = x + x;            _t2 = x + x;
_t3 = y;                _t3 = y;
b2 = _t2 == _t3;        b2 = _t2 == _t3;        b2 = _t0 == _t1;

_t4 = x + x;            _t4 = x + x;
_t5 = y;                _t5 = y;
b3 = _t5 < _t4;    ⇒    b3 = _t5 < _t4;    ⇒    b3 = _t0 < _t1;
```

In this case, we have some redundancy in ou IR, shown with the red color. One optimization we can do is to remove those lines that cause redundancy.

Optimizations from Lazy Coders:

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

```
_L0:
    _t0 = y + z;
    _t1 = x < _t0;
    IfZ _t1 Goto _L1;
    x = x - y;
    Goto _L0;
_L1:
```

⇒

```
    _t0 = y + z;
_L0:
    _t1 = x < _t0;
    IfZ _t1 Goto _L1;
    x = x - y;
    Goto _L0;
_L1:
```

⇒

```
    _t0 = y + z;
_L0:
    _t1 = x < _t0;
    IfZ _t1 Goto _L1;
    x = x - y;
    Goto _L0;
_L1:
```

The expression x < y + z coming with the while is evaluated every time we run the loop. Since y + z never changes, we do not need to recompute it again inside the while-loop; therefore, we can move that outside of the loop.

Optimization means looking for an “optimal” piece of code for a program. A good optimizer
● Should never change the observable behavior of a program.
● Should produce IR that is as efficient as possible.
● Should not take too long to process inputs.

We do not want our compiler to introduce security vulnerabilities that have nothing to do with functional equivalent.

We want to optimize:
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)
Plus a lot more (minimize function calls, reduce usage of floating-point hardware, etc.)

IR Optimization vs Code Optimization:
Not any clear distinction for what belongs to IR opt. or code opt.
IR optimization try to perform simplifications that are valid across machines, whereas, code optimizations try to improve performance based on the specifics of the machine.

Analyzing a Program:
Being able to reason about properties of the program.
An analysis is called **sound** if it never asserts an incorrect fact about a program.

*Soundness.*Ex:

```
int x;
int y;
if (y<5)
        x = 137;
else
        x = 42;
print (x);          // At this point of the program, x is either 137 or 42. -> This is a sound.
                    // At this point of the program, x is either 137, 42, or 271. -> This is a sound.
                    //  At this point of the program, x holds some integer value. -> This is a sound.
```

Semantics-Preserving Optimizations

An optimization is semantics-preserving if it does not alter the semantics of the original program.

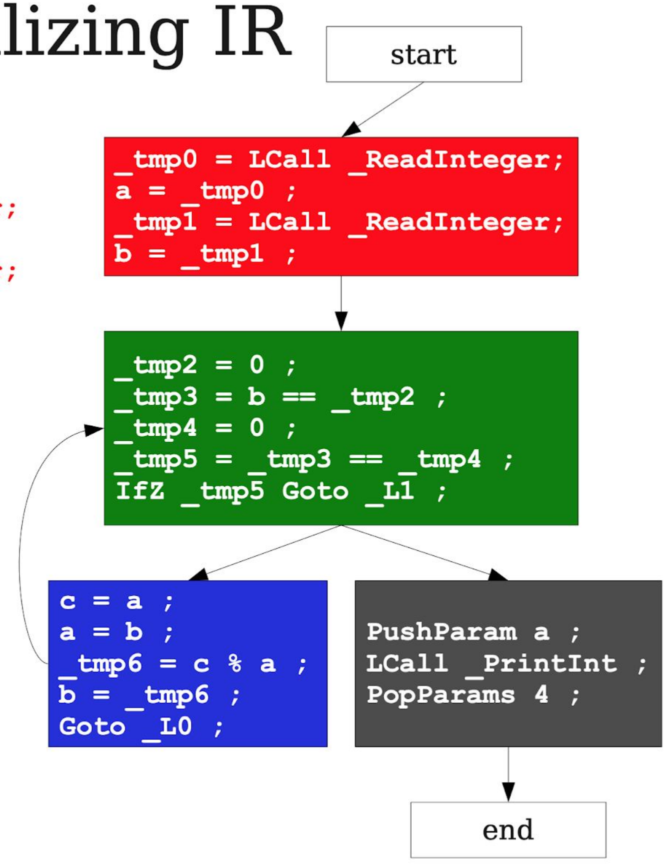
Ex: Eliminating unnecessary temporary variables.

Non-Ex: Replacing bubble sort with quicksort.

● Scanning uses regular expressions.
● Parsing uses Context-free Grammars.
● Semantic analysis uses proof systems and symbol tables.
● IR generation uses Abstract Syntax Trees.

Visualizing IR

```
main:
    BeginFunc 40;
    _tmp0 = LCall _ReadInteger;
    a = _tmp0;
    _tmp1 = LCall _ReadInteger;
    b = _tmp1;
_L0:
    _tmp2 = 0;
    _tmp3 = b == _tmp2;
    _tmp4 = 0;
    _tmp5 = _tmp3 == _tmp4;
    IfZ _tmp5 Goto _L1;
    c = a;
    a = b;
    _tmp6 = c % a;
    b = _tmp6;
    Goto _L0;
_L1:
    PushParam a;
    LCall _PrintInt;
    PopParams 4;
    EndFunc;
```



```
_tmp0 = LCall _ReadInteger;
a = _tmp0 ;
_tmp1 = LCall _ReadInteger;
b = _tmp1 ;
```

```
_tmp2 = 0 ;
_tmp3 = b == _tmp2 ;
_tmp4 = 0 ;
_tmp5 = _tmp3 == _tmp4 ;
IfZ _tmp5 Goto _L1 ;
```

```
c = a ;
a = b ;
_tmp6 = c % a ;
b = _tmp6 ;
Goto _L0 ;
```

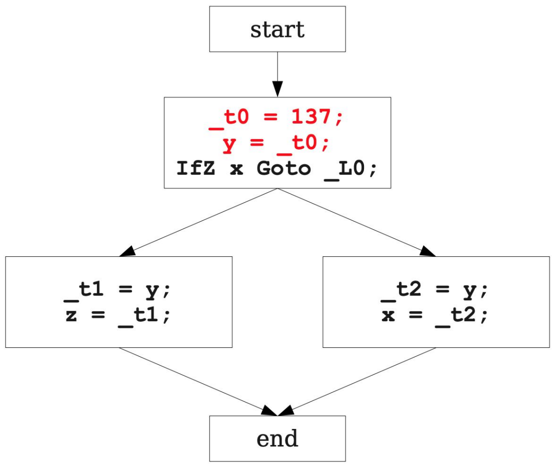
```
PushParam a ;
LCall _PrintInt ;
PopParams 4 ;
```

Control flow graph has basic blocks which are straight line code that do not have any branches in or out, and the edges of the graph are branching behavior.

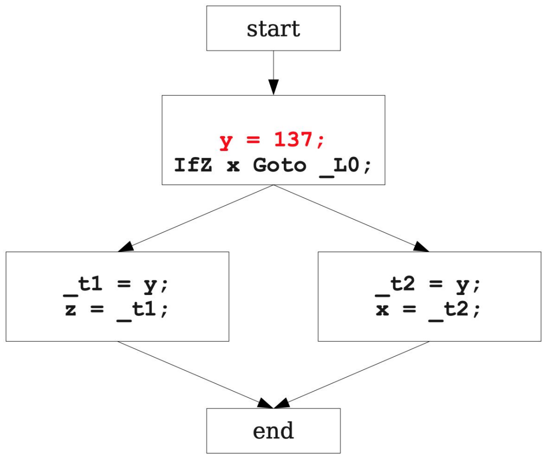
Basic Blocks:
It is a sequence of IR instructions where
● There is exactly one spot where control enters the sequence, which must be at the start of the sequence.
● There is exactly one spot where control leaves the sequence, which must be at the end of the sequence.

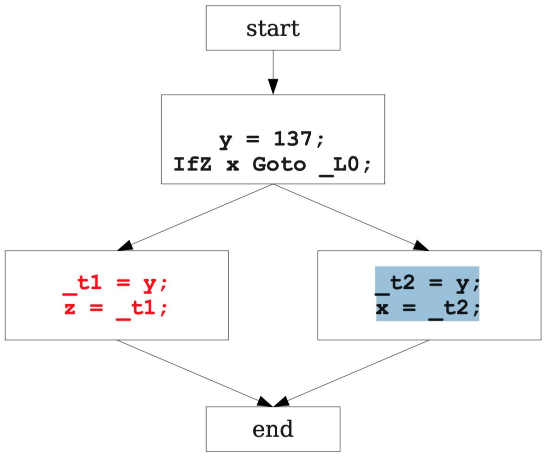
An optimization is **local** if it works on just a single basic block.
An optimization is **global** if it works on an entire control-flow graph.

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

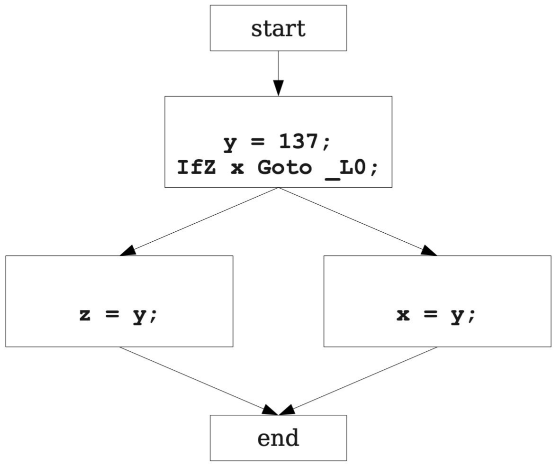

start
_t0 = 137;
y = _t0;
IfZ x Goto _L0;
_t1 = y;
z = _t1;
_t2 = y;
x = _t2;
end

⇒

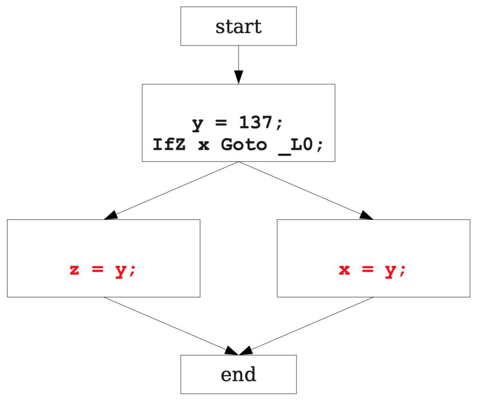

start
y = 137;
IfZ x Goto _L0;
_t1 = y;
z = _t1;
_t2 = y;
x = _t2;
end


start
y = 137;
IfZ x Goto _L0;
_t1 = y;
z = _t1;
_t2 = y;
x = _t2;
end

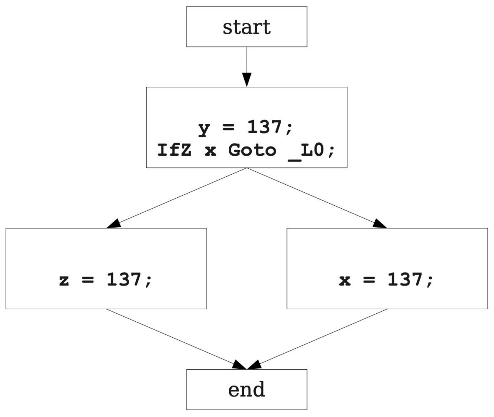
⇒


start
y = 137;
IfZ x Goto _L0;
z = y;
x = y;
end

Global Optimizations:



⇒



Common Subexpression Elimination:

Object x;
int a;
int b;
int c;
x = new Object;
a = 4;
c = a + b;
x.fn(a+b);

```
_tmp0 = 4 ;
PushParam _tmp0 ;
_tmp1 = LCall _Alloc ;
PopParams 4 ;
_tmp2 = Object ;
*(_tmp1) = _tmp2 ;
x = _tmp1 ;
_tmp3 = 4 ;
a = _tmp3 ;
_tmp4 = a + b ;
c = _tmp4 ;
_tmp5 = a + b ;
_tmp6 = *(x) ;
_tmp7 = *(_tmp6) ;
PushParam _tmp5 ;
PushParam x ;
ACall _tmp7 ;
PopParams 8 ;
```

⇒

```
_tmp0 = 4 ;
PushParam _tmp0 ;
_tmp1 = LCall _Alloc ;
PopParams 4 ;
_tmp2 = Object ;
*(_tmp1) = _tmp2 ;
x = _tmp1 ;
_tmp3 = 4 ;
a = _tmp3 ;
_tmp4 = a + b ;
c = _tmp4 ;
_tmp5 = _tmp4 ;
_tmp6 = *(x) ;
_tmp7 = *(_tmp6) ;
PushParam _tmp5 ;
PushParam x ;
ACall _tmp7 ;
PopParams 8 ;
```

⇒

```
_tmp0 = 4 ;
PushParam _tmp0 ;
_tmp1 = LCall _Alloc ;
PopParams 4 ;
_tmp2 = Object ;
*(_tmp1) = _tmp2 ;
x = _tmp1 ;
_tmp3 = 4 ;
a = _tmp3 ;
_tmp4 = a + b ;
c = _tmp4 ;
_tmp5 = _tmp4 ;
_tmp6 = *(x) ;
_tmp7 = *(_tmp6) ;
PushParam _tmp5 ;
PushParam x ;
ACall _tmp7 ;
PopParams 8 ;
```

⇒

```
_tmp0 = 4 ;
PushParam _tmp0 ;
_tmp1 = LCall _Alloc ;
PopParams 4 ;
_tmp2 = Object ;
*(_tmp1) = _tmp2 ;
x = _tmp1 ;
_tmp3 = _tmp0 ;
a = _tmp3 ;
_tmp4 = a + b ;
c = _tmp4 ;
_tmp5 = _tmp4 ;
_tmp6 = *(x) ;
_tmp7 = *(_tmp6) ;
PushParam _tmp5 ;
PushParam x ;
ACall _tmp7 ;
PopParams 8 ;
```

⇒

```
_tmp0 = 4 ;
PushParam _tmp0 ;
_tmp1 = LCall _Alloc ;
PopParams 4 ;
_tmp2 = Object ;
*(_tmp1) = _tmp2 ;
x = _tmp1 ;
_tmp3 = _tmp0 ;
a = _tmp3 ;
_tmp4 = a + b ;
c = _tmp4 ;
_tmp5 = c ;
_tmp6 = *(x) ;
_tmp7 = *(_tmp6) ;
PushParam _tmp5 ;
PushParam x ;
ACall _tmp7 ;
PopParams 8 ;
```

Copy Propagation:

If we have two variable assignments

v1 = a op b

…

v2 = a op b

and the values of v1 , a, and b have not changed between the assignments, rewrite the code as

v1 = a op b

…

v2 = v1

● Eliminates useless recalculation.

● Paves the way for later optimizations.

Dead Code Elimination:

● An assignment to a variable v is called dead if the value of that assignment is never read anywhere.

● Dead code elimination removes dead assignments from IR.

● Determining whether an assignment is dead depends on what variable is being assigned to and when it's being assigned.

```
_tmp0 = 4 ;                 _tmp0 = 4 ;
PushParam _tmp0 ;           PushParam _tmp0 ;
_tmp1 = LCall _Alloc ;      _tmp1 = LCall _Alloc ;
PopParams 4 ;               PopParams 4 ;
_tmp2 = Object ;            _tmp2 = Object ;
*(_tmp1) = _tmp2 ;          *(_tmp1) = _tmp2 ;
x = _tmp1 ;
_tmp3 = 4 ;
a = 4 ;
_tmp4 = _tmp0 + b ;         _tmp4 = _tmp0 + b ;
c = _tmp4 ;
_tmp5 = _tmp4 ;
_tmp6 = _tmp2 ;
_tmp7 = *(_tmp2) ;          _tmp7 = *(_tmp2) ;
PushParam _tmp4 ;           PushParam _tmp4 ;
PushParam _tmp1 ;           PushParam _tmp1 ;
ACall _tmp7 ;               ACall _tmp7 ;
PopParams 8 ;               PopParams 8 ;
```
⇒

Arithmetic Simplifications:
Instead of x = 4 * a; write as x = a<<2

Constant Folding:
Instead of x = 4 * 5; write as x = 20

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 }
```

```
                        { }
                       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 }
```
⇒

```
                        { }
                       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 }
```
⇒

```
                        { }
                       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 }
```
⇒

⇒

```
   a = b;

   c = a;

 d = a + b;

   e = d;

   d = a;

   f = e;
```

Liveness Analysis:
● 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.

Ex: Assume b, d is live and we trace up. The letters shown in red colors must be live.

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

Dead Code Elimination:
Ex: f and c were not in the live set; therefore, they were eliminated

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

⇒

```
  a = b;


d = a + b;

  e = d;

  d = a;
```

Liveness Analysis II

```
   { b }
   a = b;

 { a, b }

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

Dead Code Elimination

```
   { b }
   a = b;

 { a, b }

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

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

Liveness Analysis III

```
   {b}
  a = b;

  {a, b}

d = a + b;

  {a, b}

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

Dead Code Elimination

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

{a, b}



{a, b}

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