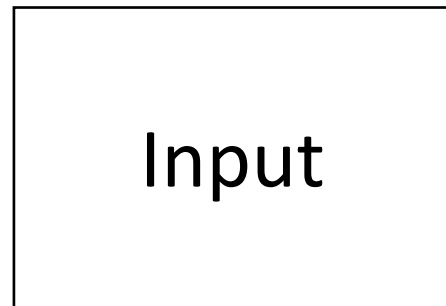


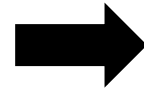
Slides from asynchronous
lecture

On to Module 2!

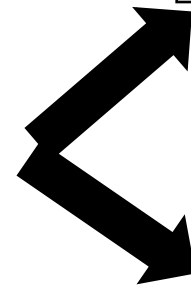
Optimizations and flow analysis



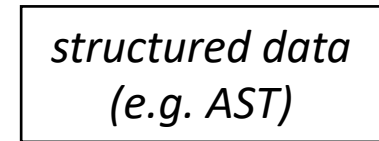
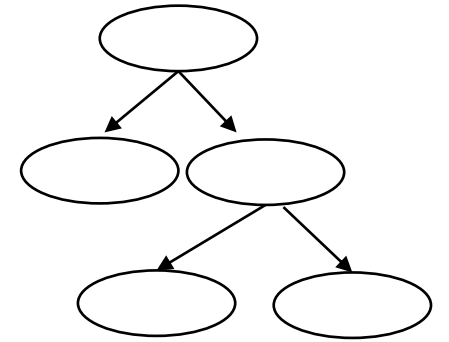
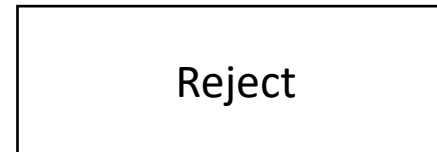
A string



*Language
Recognizer for
language L*

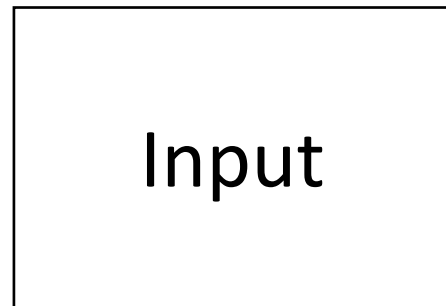


*continue to the rest
of compilation*

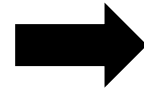


On to Module 2!

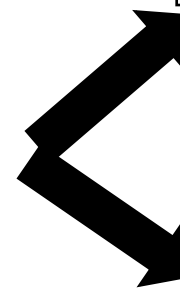
Optimizations and flow analysis



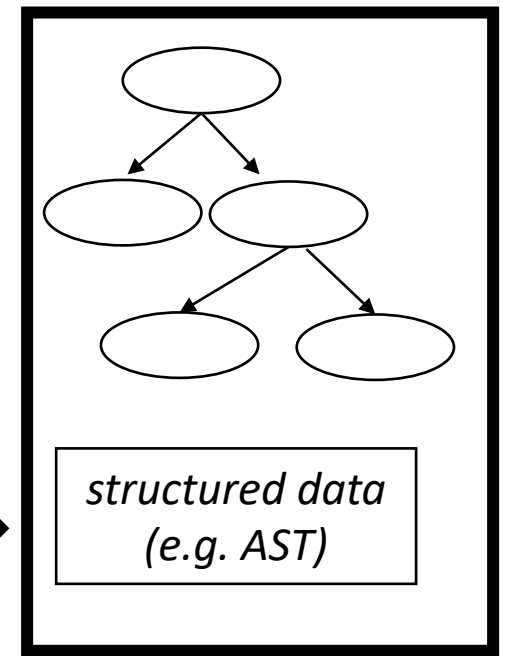
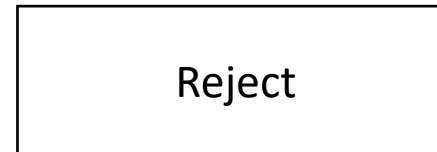
A string



*Language
Recognizer for
language L*



*continue to the rest
of compilation*



**Where most
optimizations
and flow analysis
happens!**

Intermediate representations (IRs)

- Intermediate step between human-accessible programming languages and horrible machine ISAs
- Ideal for analysis because:
 - More regularity than high-level languages (simple instructions)
 - Less constraints than ISA languages (virtual registers)
 - ***Machine-agnostic optimizations***
 - See Godbolt example

$$\begin{array}{l} x = y + z; \\ w = y + z; \end{array} \quad \longrightarrow \quad \begin{array}{l} x = y + z; \\ w = x; \end{array}$$

Different IRs

Many different IRs, each have different purposes

- Trees
 - Abstract syntax trees
 - Data-dependency trees
 - **Good for instruction scheduling**
- Textual
 - 3 address code
 - **Good for local value numberings, removing redundant expressions**
- Graphs
 - Control flow graphs
 - **Good for data flow analysis**

Different IRs

Many different IRs, each have different purposes

- Trees
 - Abstract syntax trees
 - Data-dependency trees
 - **Good for instruction scheduling**
- Textual
 - 3 address code
 - **Good for local value numberings, removing redundant expressions**
- Graphs
 - Control flow graphs
 - **Good for data flow analysis**

What are some examples of a modern compiler pipeline?

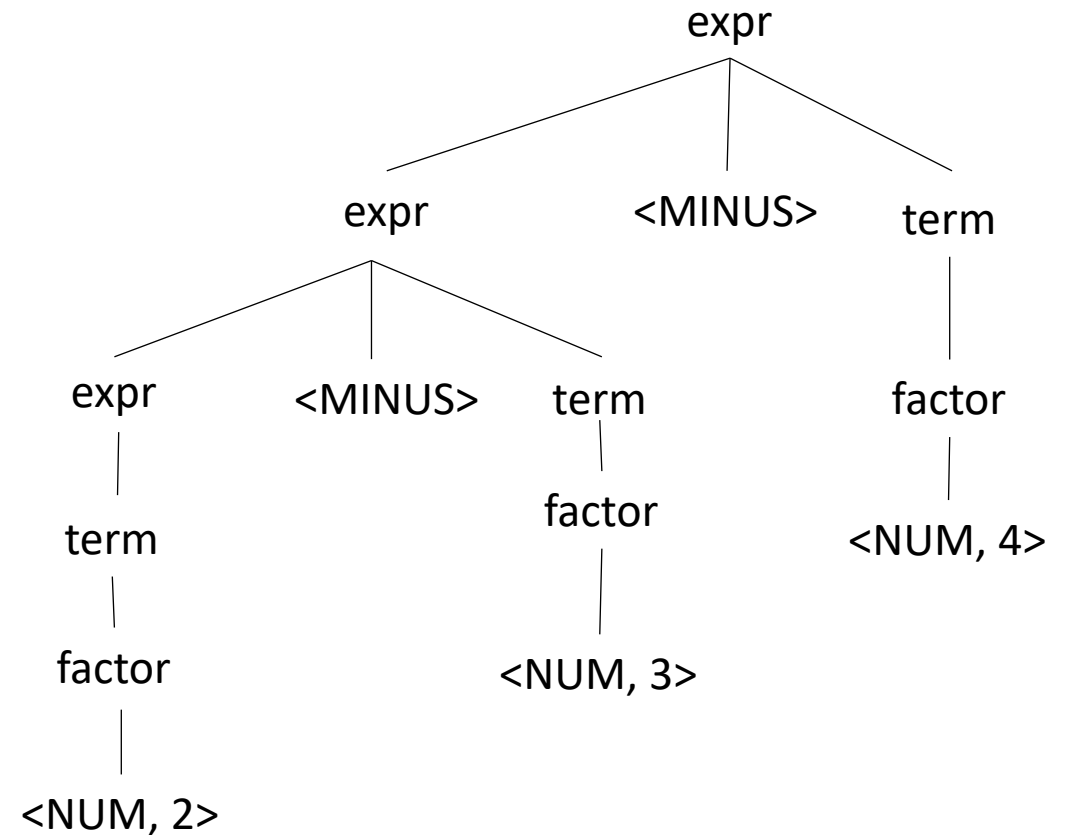
GPUs often have many IRs... why?

Abstract Syntax Trees

- Remember the expression parse tree

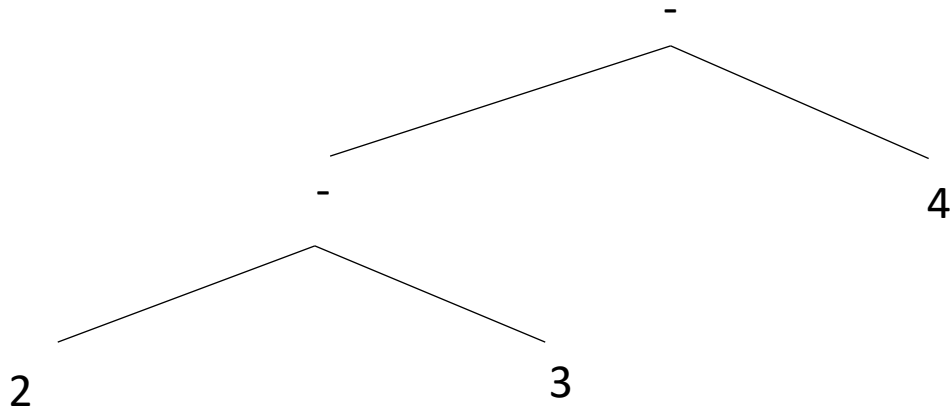
input: 2-3-4

Operator	Name	Productions
+,-	expr	: expr PLUS term expr MINUS term term
*,/	term	: term TIMES pow term DIV pow Pow
^	pow	: factor CARROT pow factor
()	factor	: LPAR expr RPAR NUM



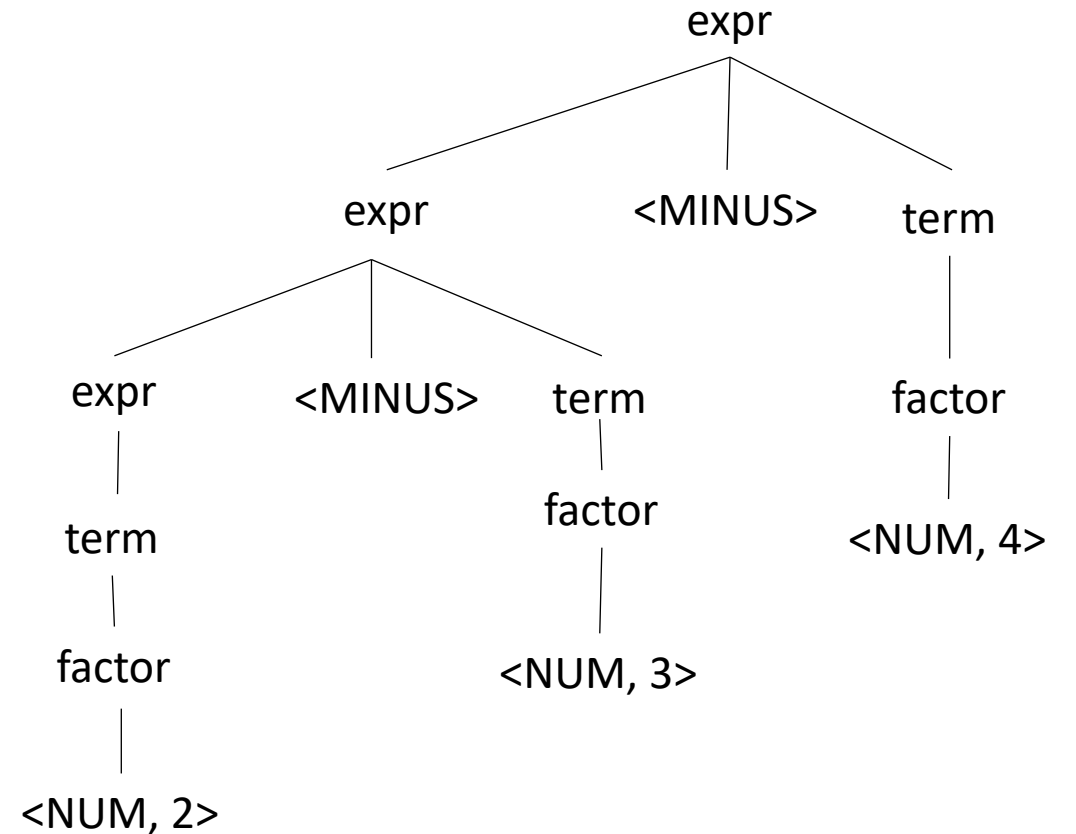
Abstract Syntax Trees

- Convert into an AST



Much more compact!

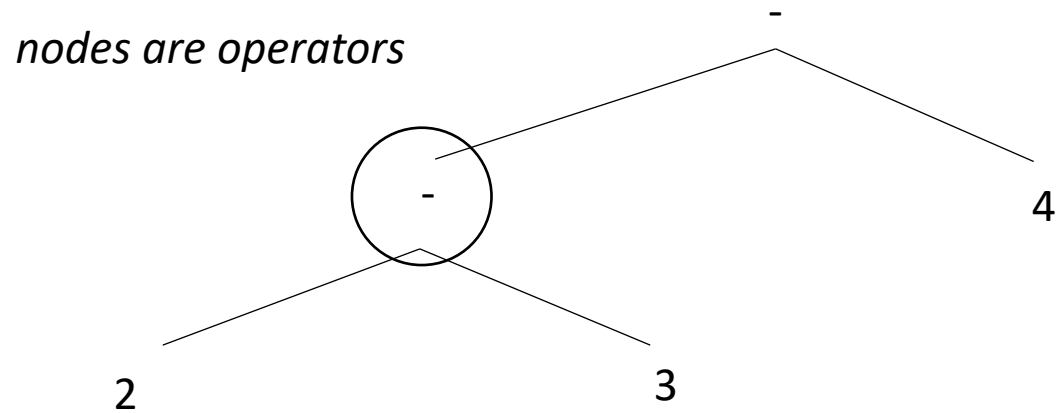
input: 2-3-4



Abstract Syntax Trees

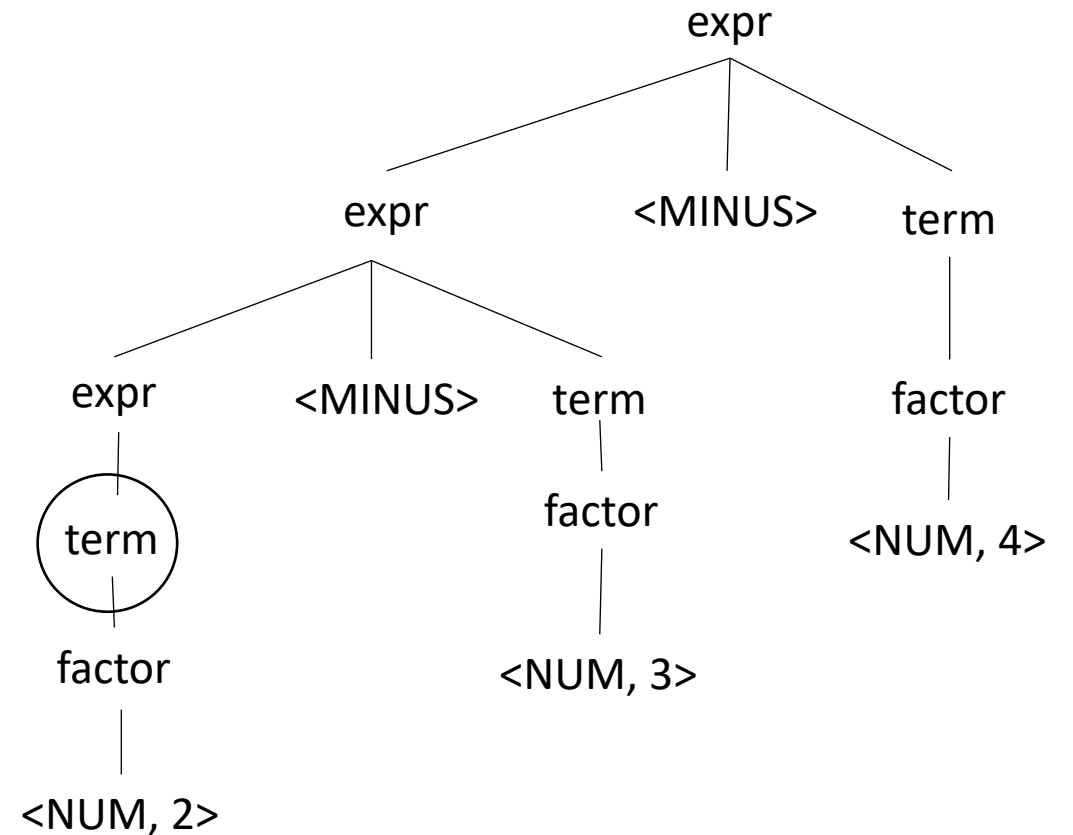
- Convert into an AST

input: 2-3-4



Much more compact!

*nodes are
production
rules*



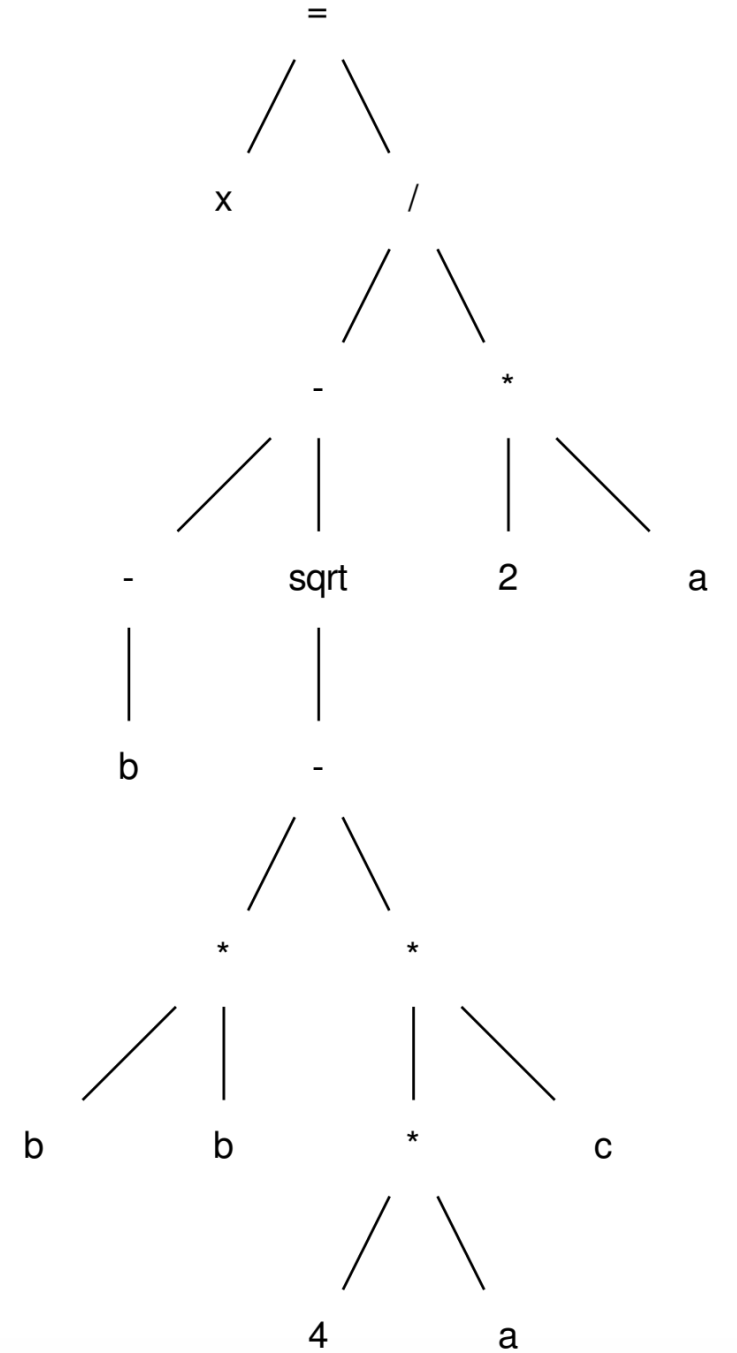
Abstract Syntax Trees

- Easier to see bigger trees, e.g. quadratic formula:

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

$$x = (-b - \text{sqrt}(b*b - 4 * a * c)) / (2*a)$$

$$x = (-b - \text{sqrt}(b*b - 4 * a * c)) / (2*a)$$



3 address code IR

- Each instruction consists of 3 “addresses”
 - Address here means a virtual register or value
 - unlimited virtual registers
- represented many ways:

```
rx = ry op rz;
```

```
r5 = r3 + r6;
```

```
r6 = r0 * r7;
```

3 address code IR

- Each instruction consists of 3 “addresses”
 - Address here means a virtual register or value
 - unlimited virtual registers
- represented many ways:

$rx \leftarrow ry \ op \ rz;$

$r5 \leftarrow r3 \ + \ r6;$

$r6 \leftarrow r0 \ * \ r7;$

3 address code IR

- Each instruction consists of 3 “addresses”
 - Address here means a virtual register or value
 - unlimited virtual registers
- represented many ways:

```
rx = op ry, rz;
```

```
r5 = add r3, r6;
```

```
r6 = mult r0, r7;
```

3 address code IR

- Each instruction consists of 3 “addresses”
 - Address here means a virtual register or value
 - unlimited virtual registers
- some instructions don't fit the pattern:

```
store ry, rz;
```

```
r5 = copy r3;
```

```
r6 = call(r0, r1, r2, r3...);
```

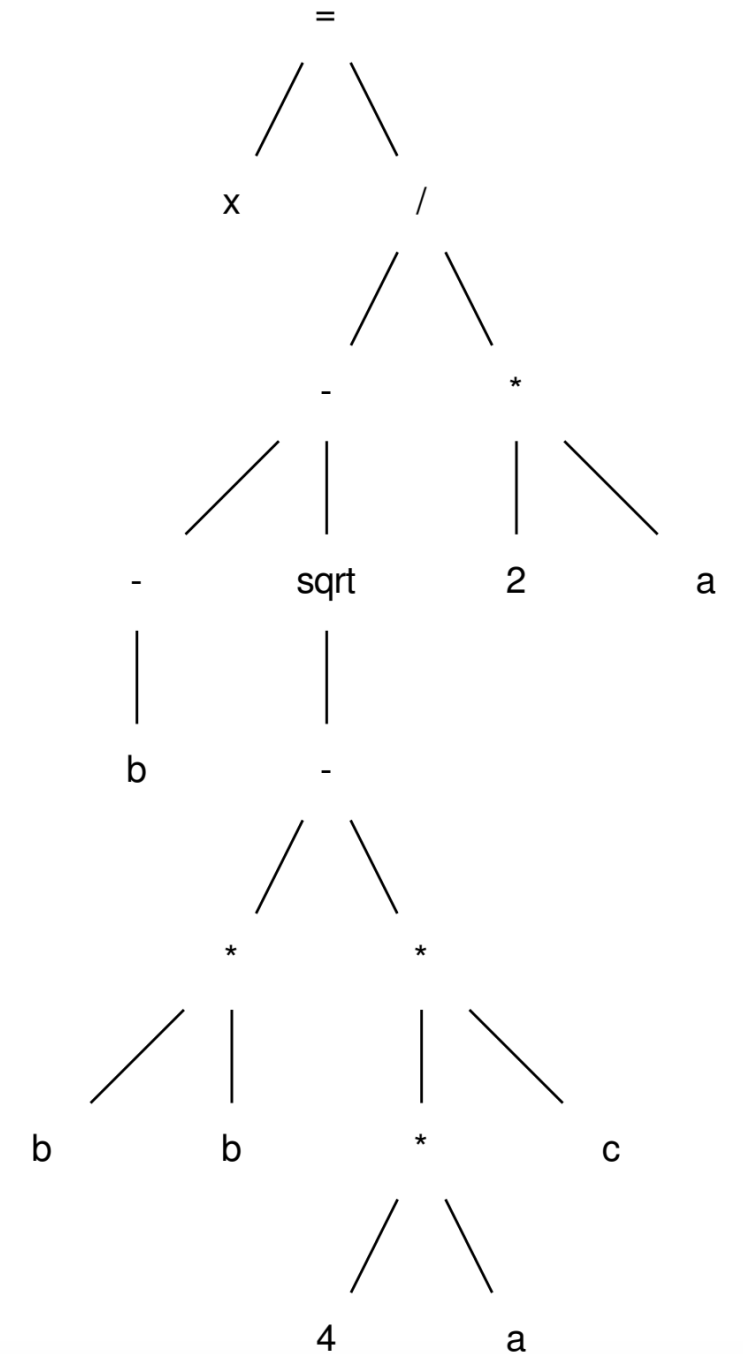
3 address code IR

- Each instruction consists of 3 “addresses”
 - Address here means a virtual register or value
 - unlimited virtual registers
- Other information:
 - Annotated
 - Typed
 - Alignment

```
r5 = r3 + r6; !dbg !22  
r6 = r0 *(int32) 67;  
store(r1,r2), aligned 8
```


Convert this code to 3 address code

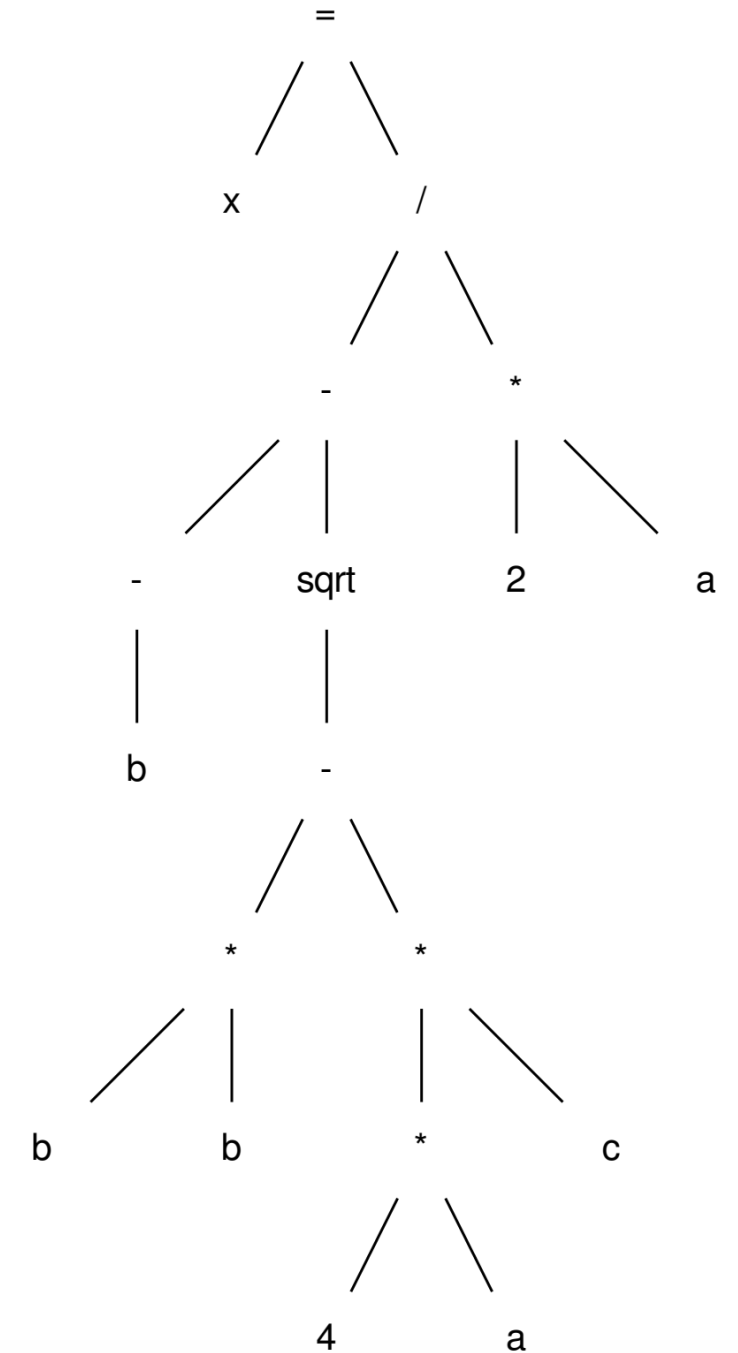
post-order traversal, creating virtual registers for each node



Convert this code to 3 address code

post-order traversal, creating virtual registers for each node

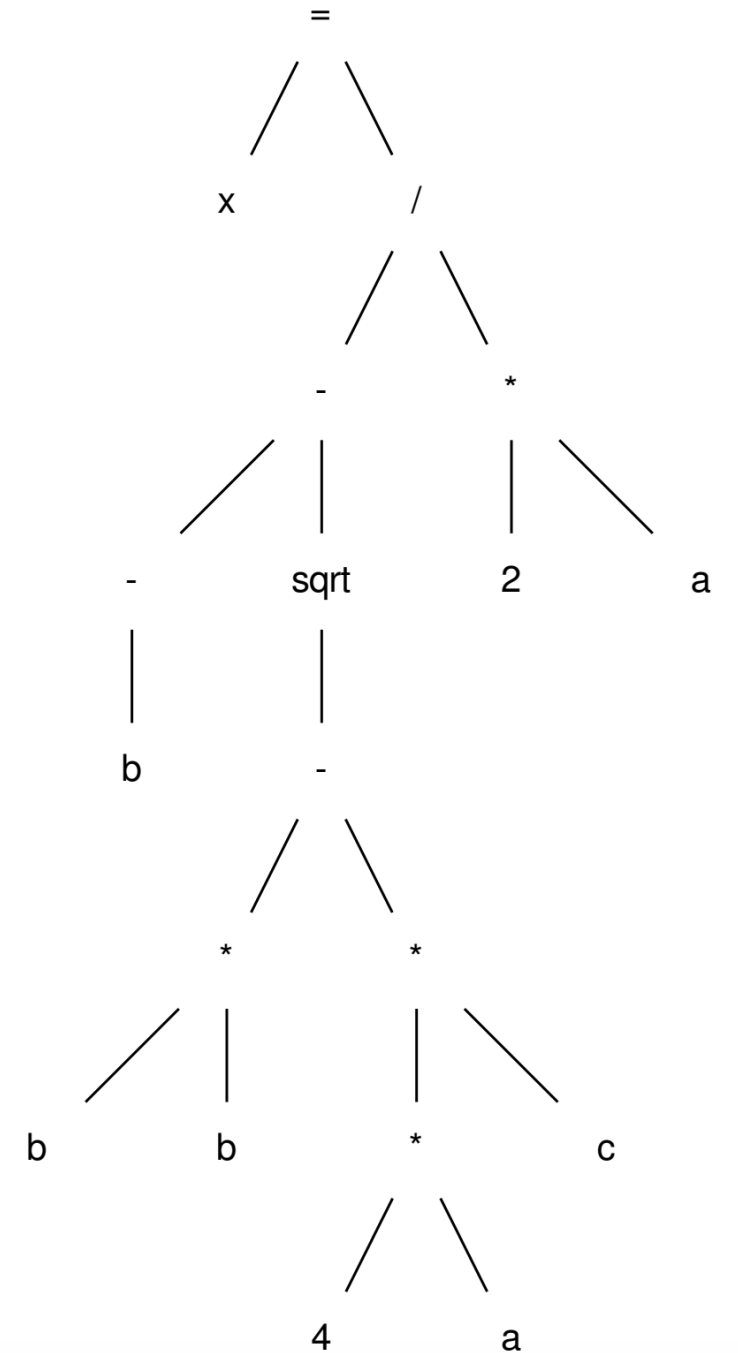
```
r0 = neg(b);
```



Convert this code to 3 address code

post-order traversal, creating virtual registers for each node

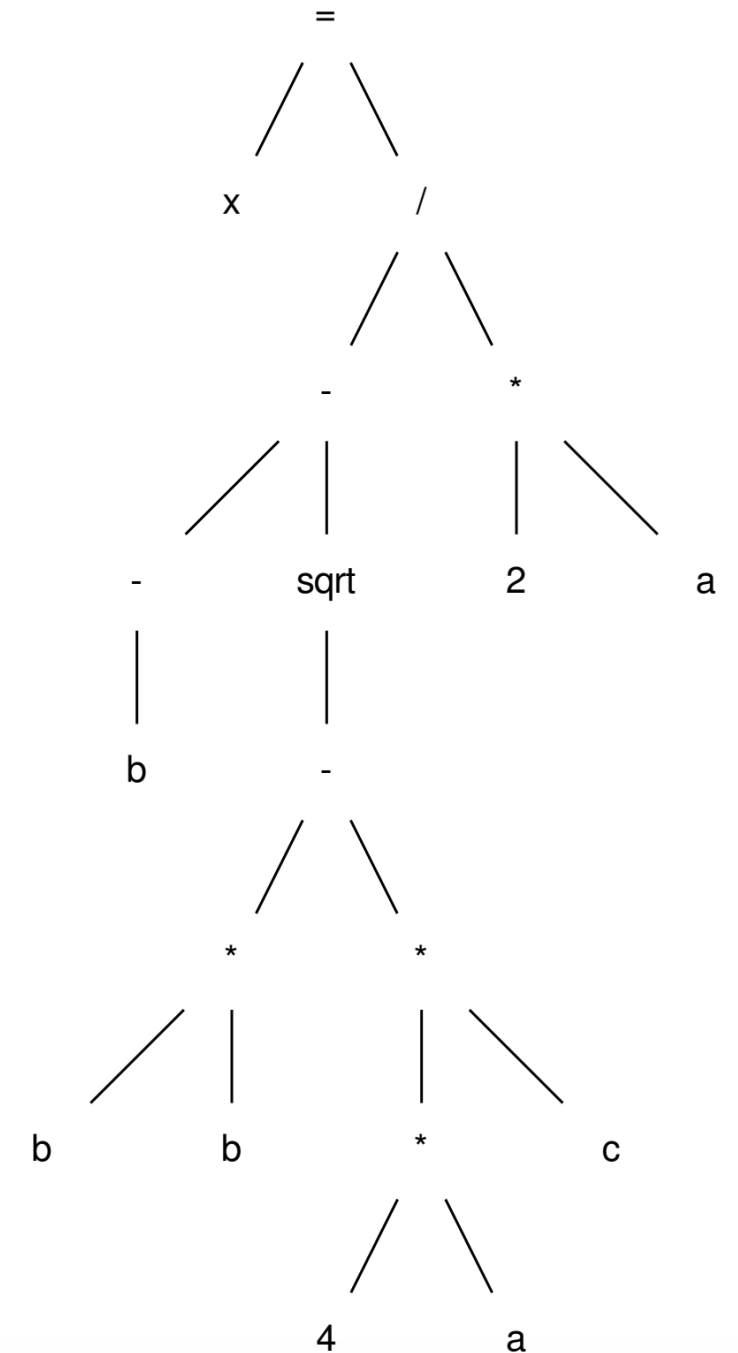
```
r0 = neg(b);  
r1 = b * b;
```



Convert this code to 3 address code

post-order traversal, creating virtual registers for each node

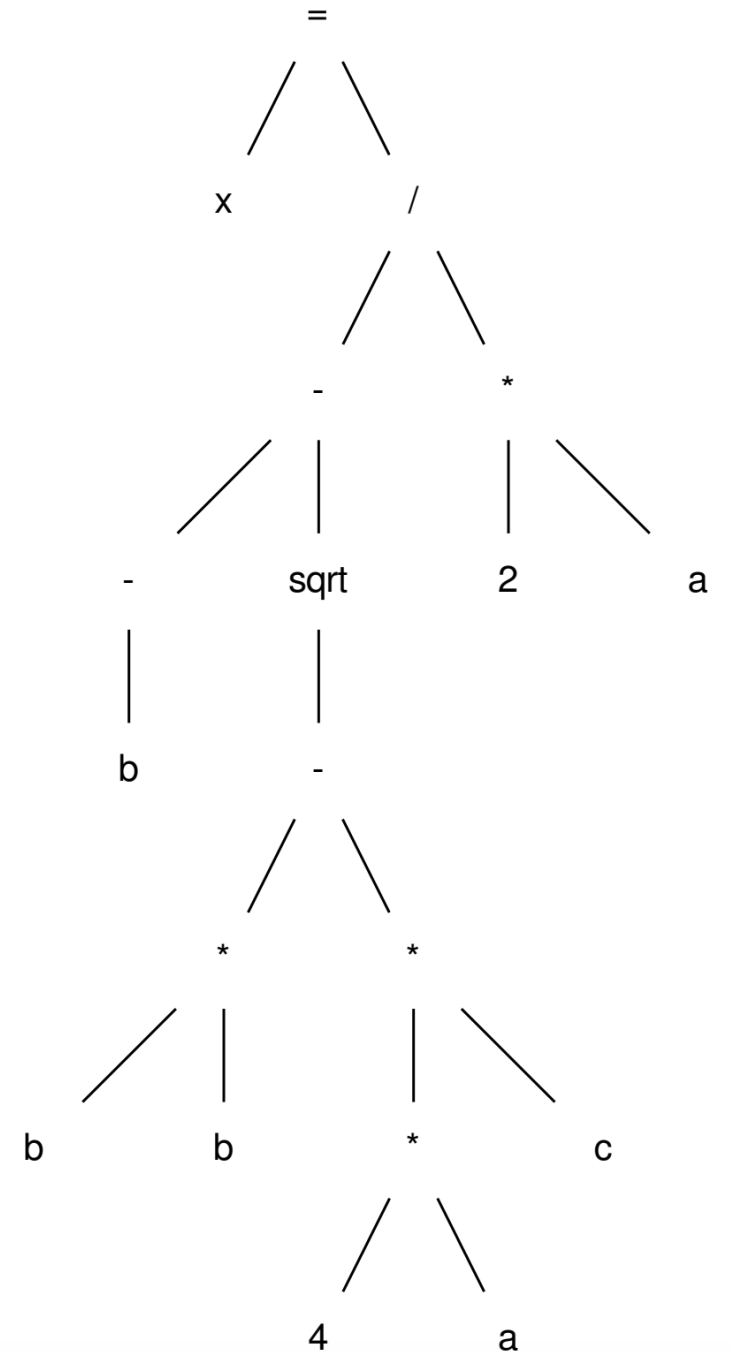
```
r0 = neg(b);  
r1 = b * b;  
r2 = 4 * a;
```



Convert this code to 3 address code

post-order traversal, creating virtual registers for each node

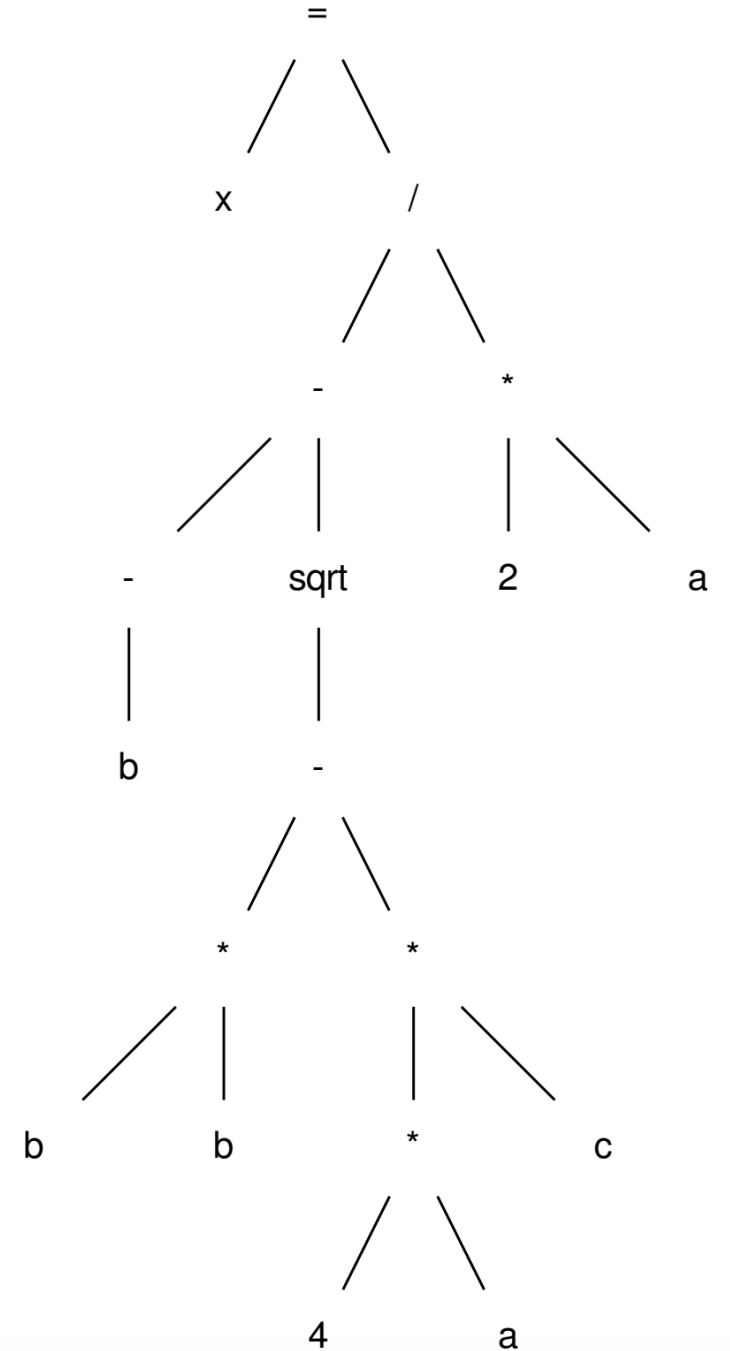
```
r0 = neg(b);  
r1 = b * b;  
r2 = 4 * a;  
r3 = r2 * c;
```



Convert this code to 3 address code

post-order traversal, creating virtual registers for each node

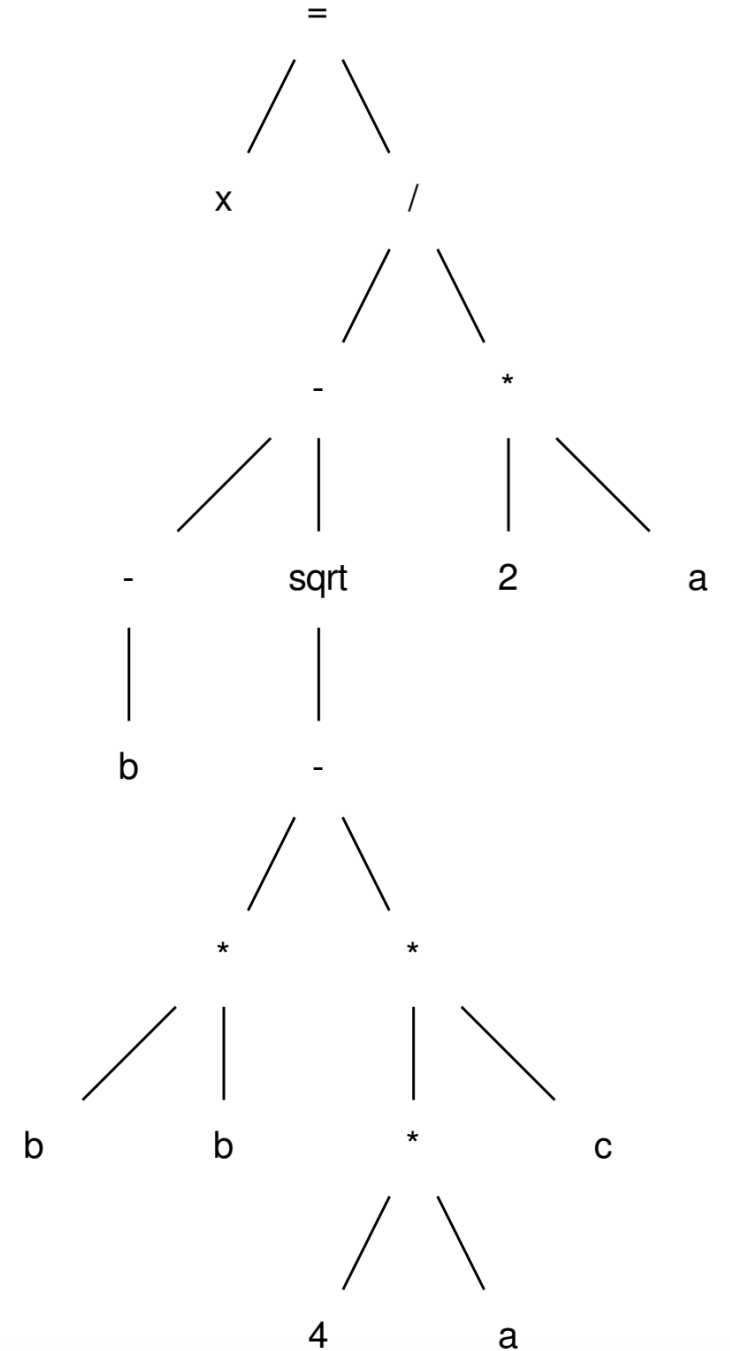
```
r0 = neg(b);  
r1 = b * b;  
r2 = 4 * a;  
r3 = r2 * c;  
r4 = r1 - r3;
```



Convert this code to 3 address code

post-order traversal, creating virtual registers for each node

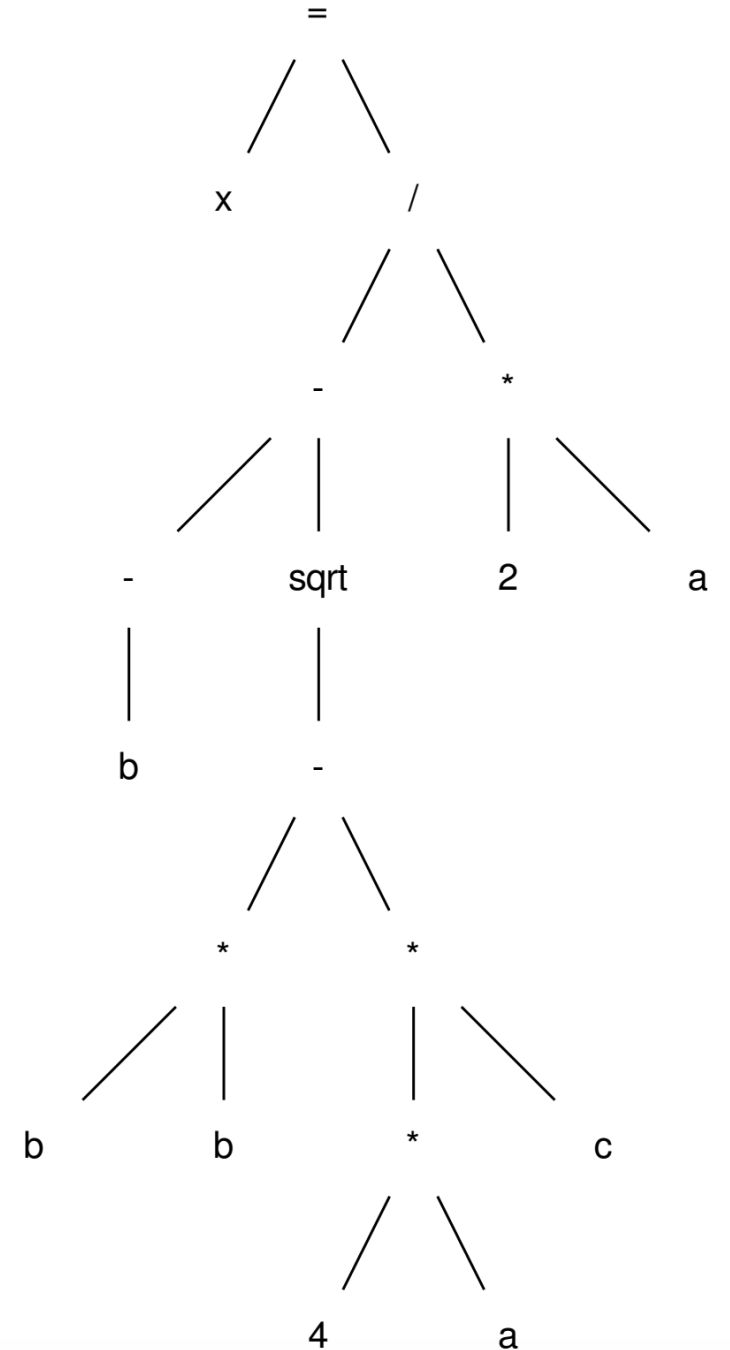
```
r0 = neg(b);  
r1 = b * b;  
r2 = 4 * a;  
r3 = r2 * c;  
r4 = r1 - r3;  
r5 = sqrt(r4);
```



Convert this code to 3 address code

post-order traversal, creating virtual registers for each node

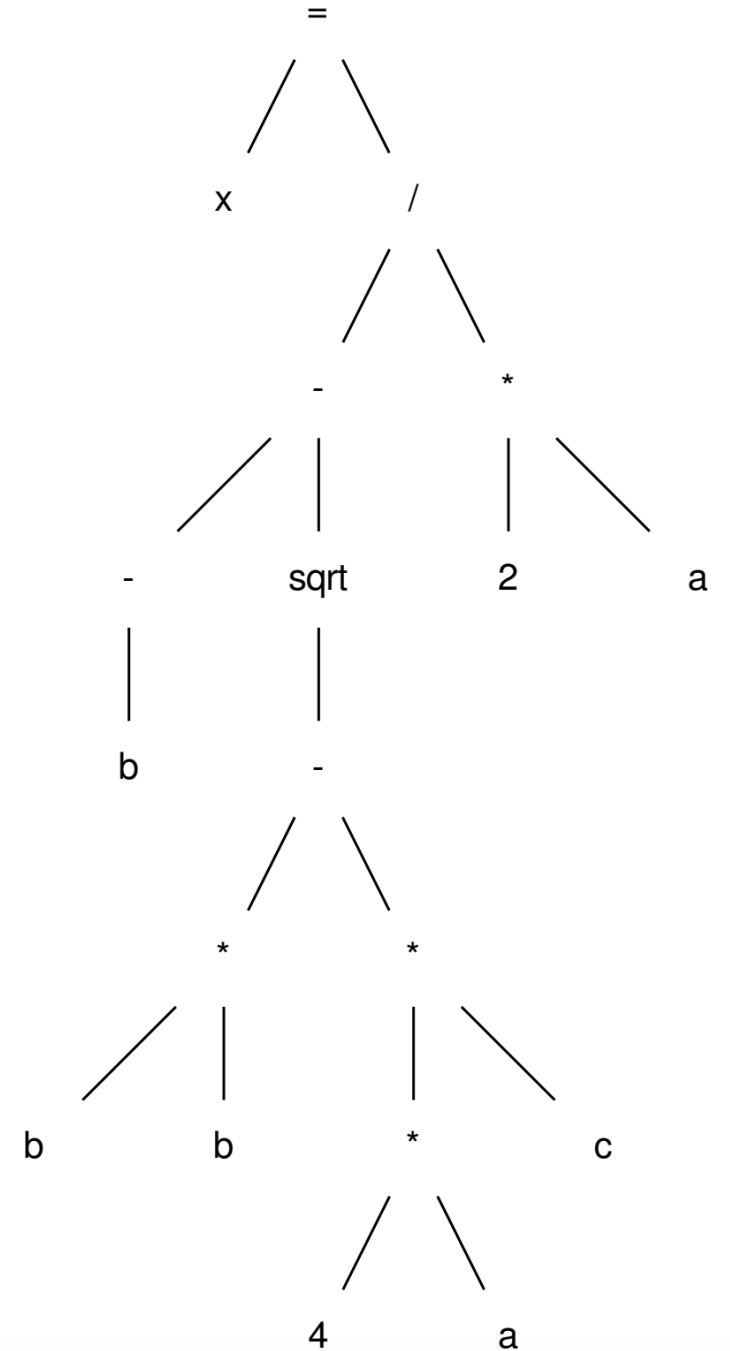
```
r0 = neg(b);  
r1 = b * b;  
r2 = 4 * a;  
r3 = r2 * c;  
r4 = r1 - r3;  
r5 = sqrt(r4);  
r6 = r0 - r5;
```



Convert this code to 3 address code

post-order traversal, creating virtual registers for each node

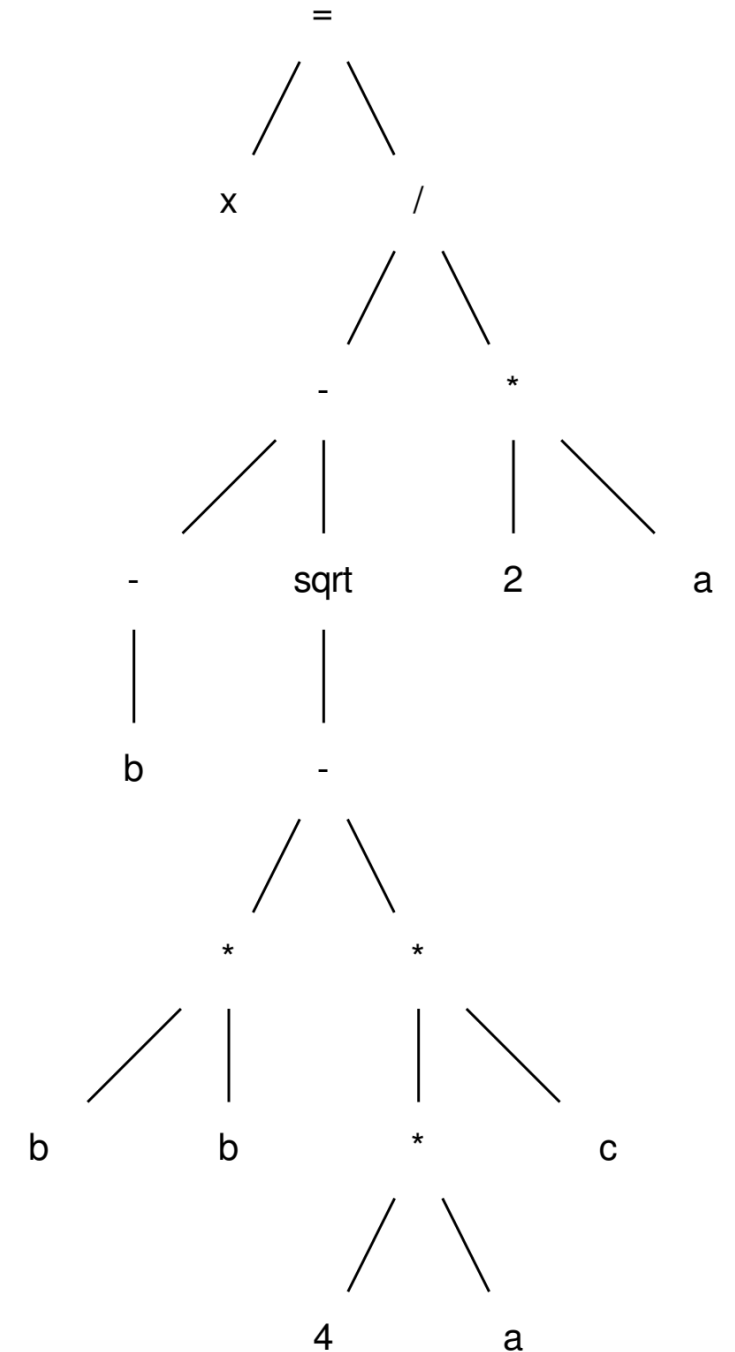
```
r0 = neg(b);  
r1 = b * b;  
r2 = 4 * a;  
r3 = r2 * c;  
r4 = r1 - r3;  
r5 = sqrt(r4);  
r6 = r0 - r5;  
r7 = 2 * a;
```



Convert this code to 3 address code

post-order traversal, creating virtual registers for each node

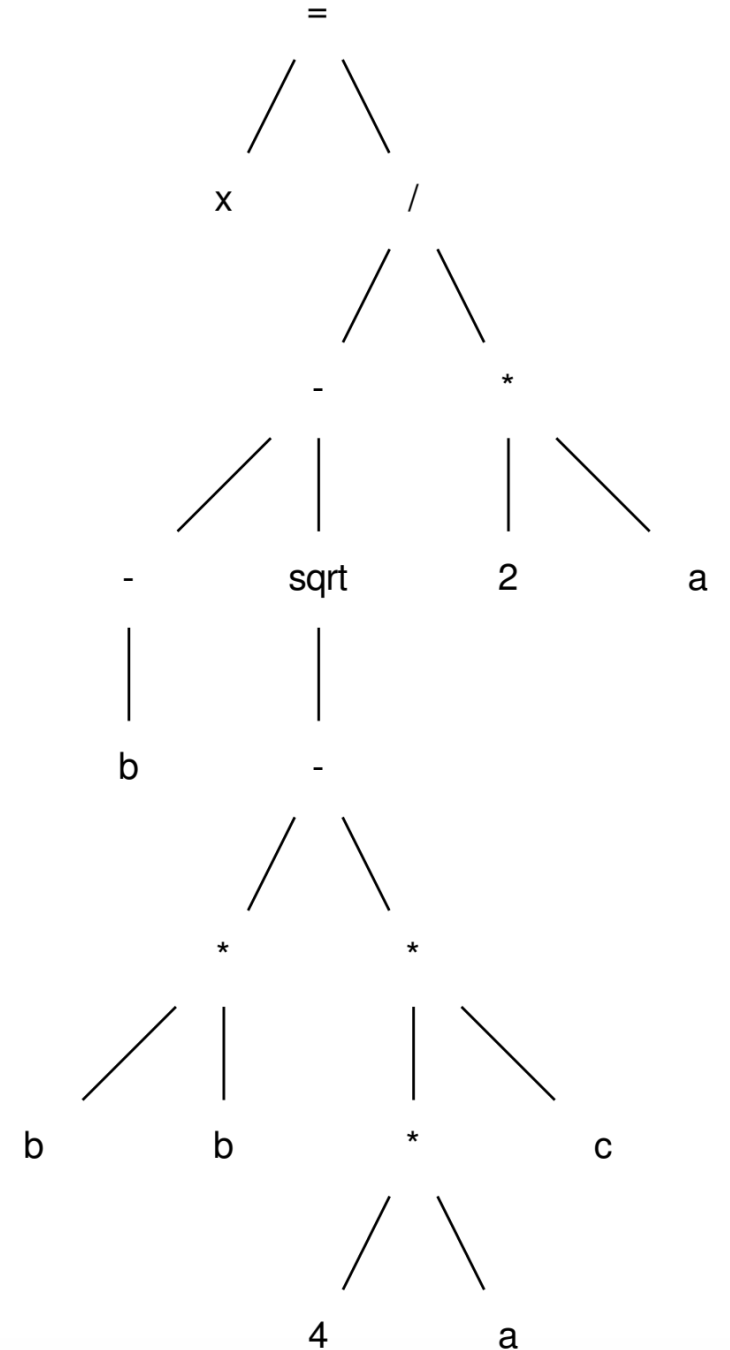
```
r0 = neg(b);  
r1 = b * b;  
r2 = 4 * a;  
r3 = r2 * c;  
r4 = r1 - r3;  
r5 = sqrt(r4);  
r6 = r0 - r5;  
r7 = 2 * a;  
r8 = r6 / r7;
```



Convert this code to 3 address code

post-order traversal, creating virtual registers for each node

```
r0 = neg(b);  
r1 = b * b;  
r2 = 4 * a;  
r3 = r2 * c;  
r4 = r1 - r3;  
r5 = sqrt(r4);  
r6 = r0 - r5;  
r7 = 2 * a;  
r8 = r6 / r7;  
x = r8;
```

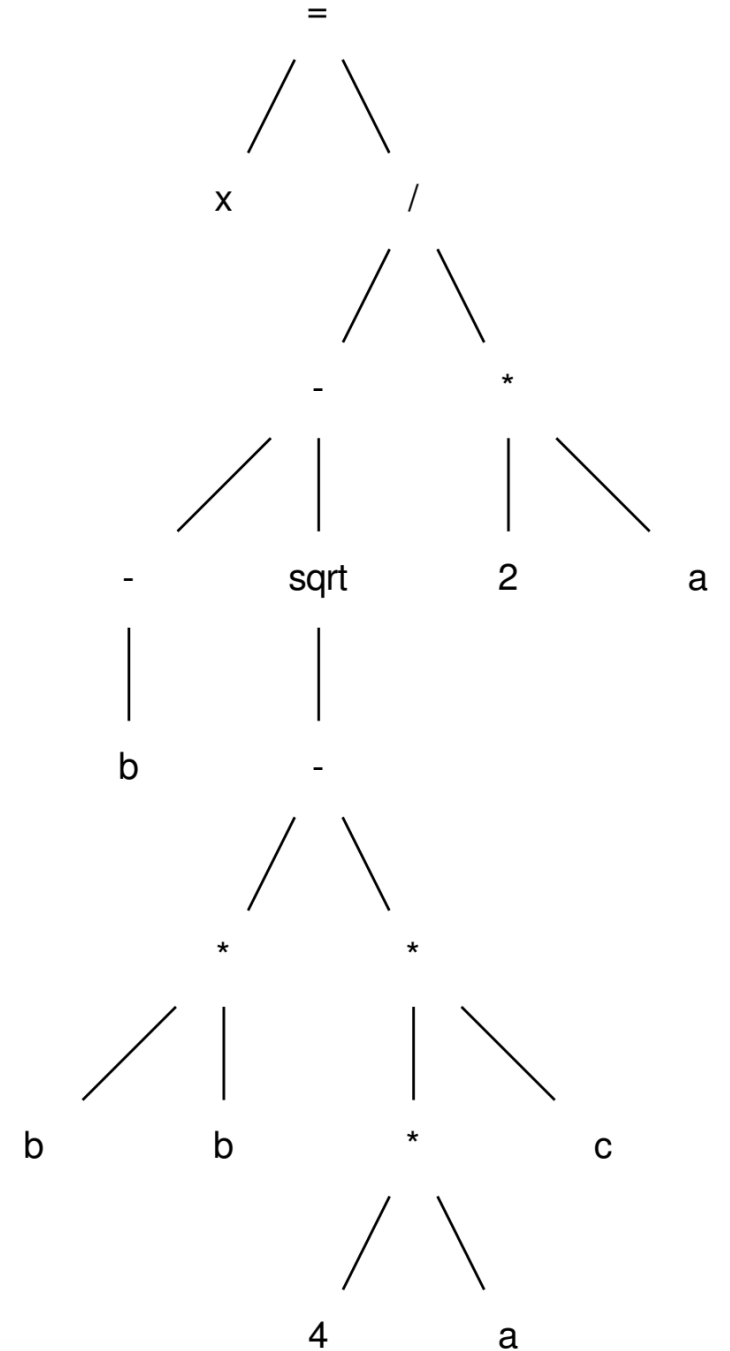


Convert this code to 3 address code

post-order traversal, creating virtual registers for each node

```
r0 = neg(b);  
r1 = b * b;  
r2 = 4 * a;  
r3 = r2 * c;  
r4 = r1 - r3;  
r5 = sqrt(r4);  
r6 = r0 - r5;  
r7 = 2 * a;  
r8 = r6 / r7;  
x = r8;
```

This is the exact code we'd see in LLVM!
See Godbolt example



What now?

We can more easily compile to machine code
OR

```
r0 = neg(b);  
r1 = b * b;  
r2 = 4 * a;  
r3 = r2 * c;  
r4 = r1 - r3;  
r5 = sqrt(r4);  
r6 = r0 - r5;  
r7 = 2 * a;  
r8 = r6 / r7;  
x  = r8;
```

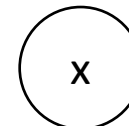
What now?

We can perform more optimizations, example:
by making a data-dependency graph (DDG)

```
r0 = neg(b);  
r1 = b * b;  
r2 = 4 * a;  
r3 = r2 * c;  
r4 = r1 - r3;  
r5 = sqrt(r4);  
r6 = r0 - r5;  
r7 = 2 * a;  
r8 = r6 / r7;  
x  = r8;
```

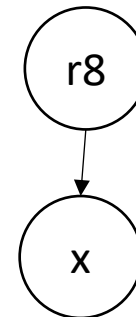
We can perform more optimizations, example:
by making a data-dependency graph (DDG)

```
r0 = neg(b);  
r1 = b * b;  
r2 = 4 * a;  
r3 = r2 * c;  
r4 = r1 - r3;  
r5 = sqrt(r4);  
r6 = r0 - r5;  
r7 = 2 * a;  
r8 = r6 / r7;  
x  = r8;
```



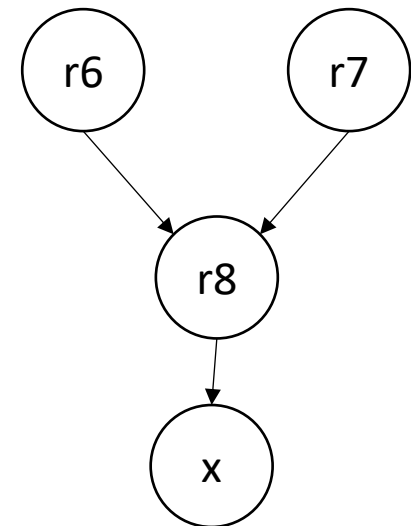
We can perform more optimizations, example:
by making a data-dependency graph (DDG)

```
r0 = neg(b);  
r1 = b * b;  
r2 = 4 * a;  
r3 = r2 * c;  
r4 = r1 - r3;  
r5 = sqrt(r4);  
r6 = r0 - r5;  
r7 = 2 * a;  
r8 = r6 / r7;  
x  = r8;
```



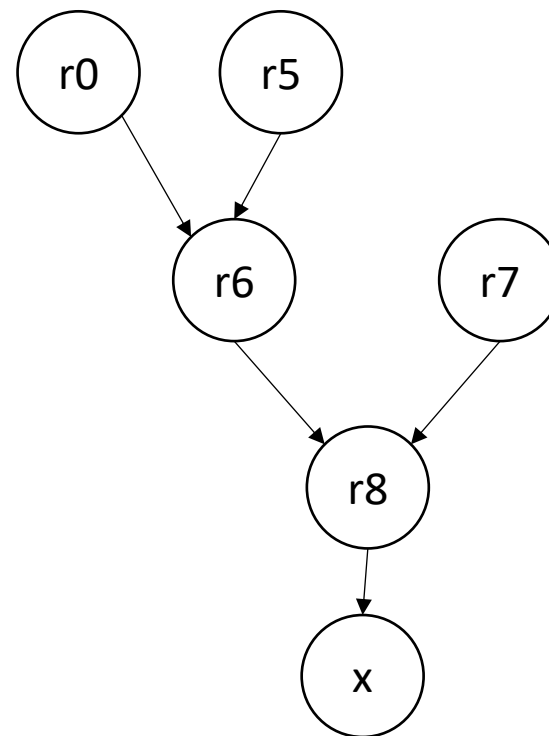
We can perform more optimizations, example:
by making a data-dependency graph (DDG)

```
r0 = neg(b);  
r1 = b * b;  
r2 = 4 * a;  
r3 = r2 * c;  
r4 = r1 - r3;  
r5 = sqrt(r4);  
r6 = r0 - r5;  
r7 = 2 * a;  
r8 = r6 / r7;  
x  = r8;
```



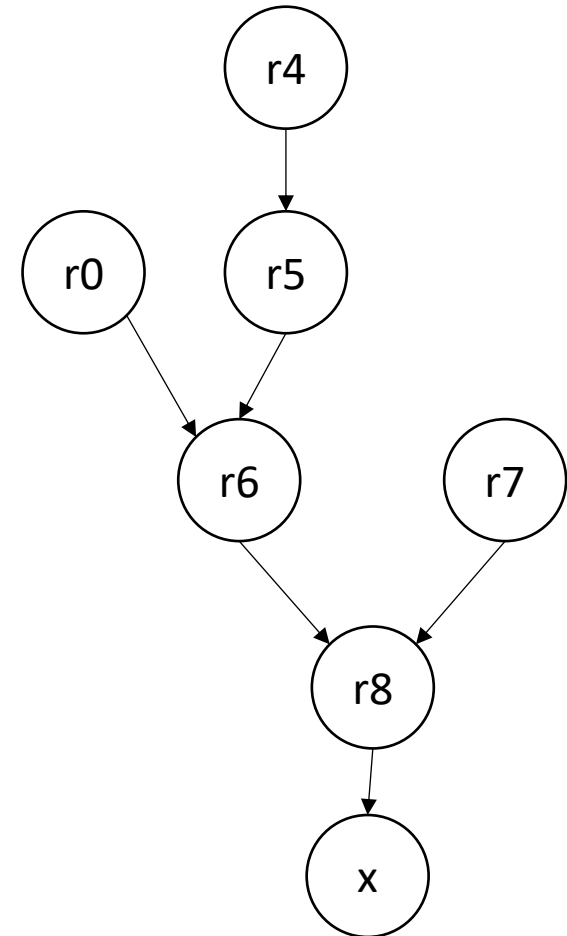
We can perform more optimizations, example:
by making a data-dependency graph (DDG)

```
r0 = neg(b);  
r1 = b * b;  
r2 = 4 * a;  
r3 = r2 * c;  
r4 = r1 - r3;  
r5 = sqrt(r4);  
r6 = r0 - r5;  
r7 = 2 * a;  
r8 = r6 / r7;  
x  = r8;
```



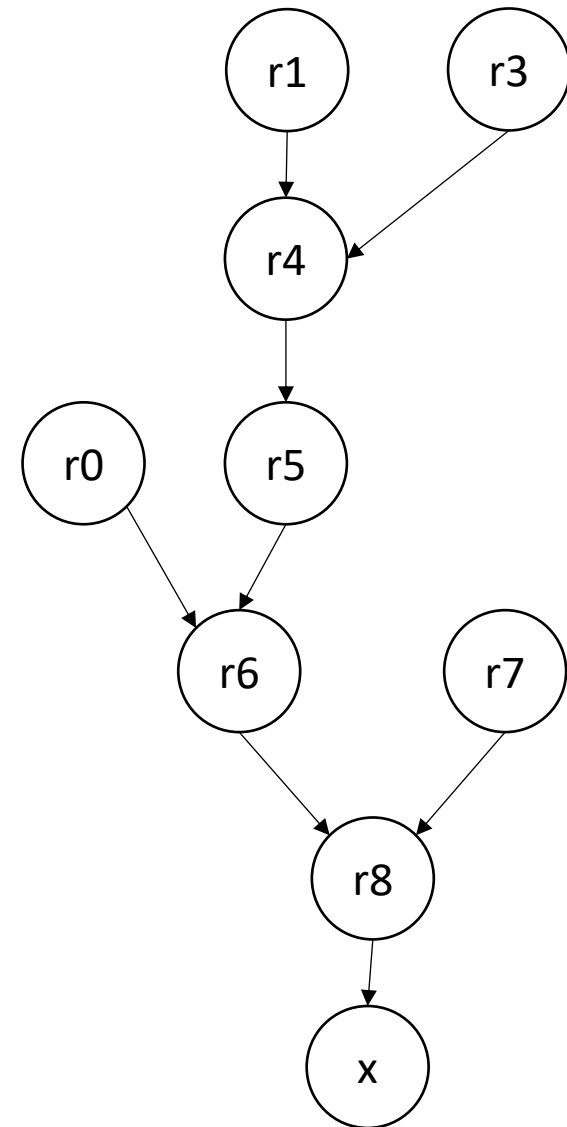
We can perform more optimizations, example:
by making a data-dependency graph (DDG)

```
r0 = neg(b);  
r1 = b * b;  
r2 = 4 * a;  
r3 = r2 * c;  
r4 = r1 - r3;  
r5 = sqrt(r4);  
r6 = r0 - r5;  
r7 = 2 * a;  
r8 = r6 / r7;  
x  = r8;
```



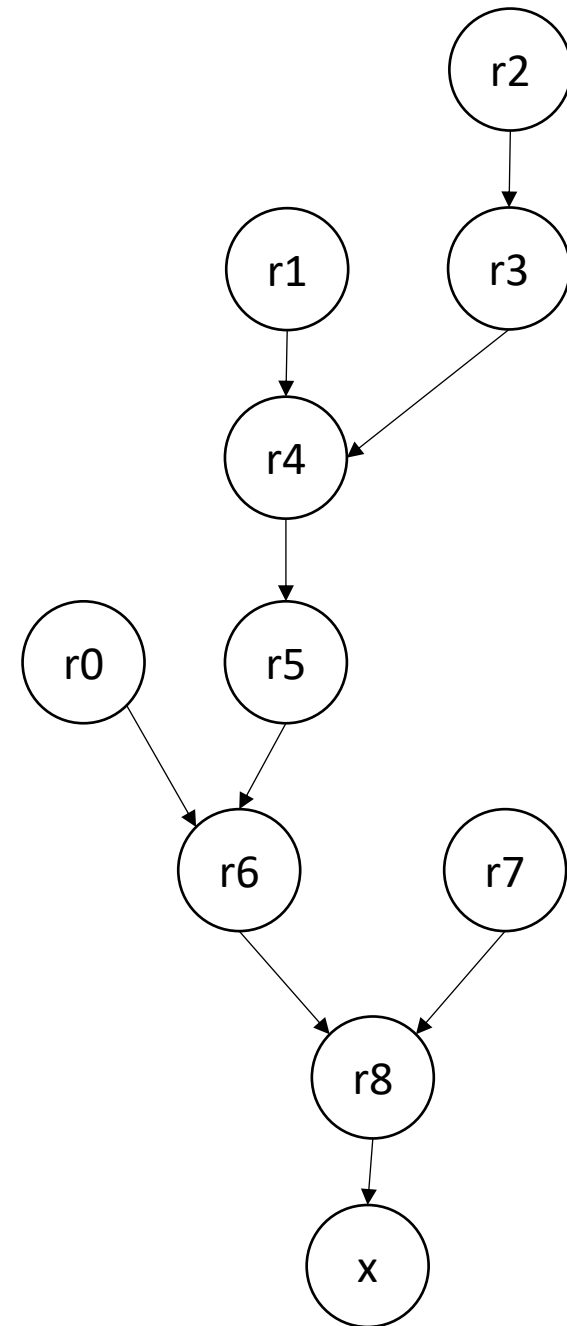
We can perform more optimizations, example:
by making a data-dependency graph (DDG)

```
r0 = neg(b);  
r1 = b * b;  
r2 = 4 * a;  
r3 = r2 * c;  
r4 = r1 - r3;  
r5 = sqrt(r4);  
r6 = r0 - r5;  
r7 = 2 * a;  
r8 = r6 / r7;  
x  = r8;
```



We can perform more optimizations, example:
by making a data-dependency graph (DDG)

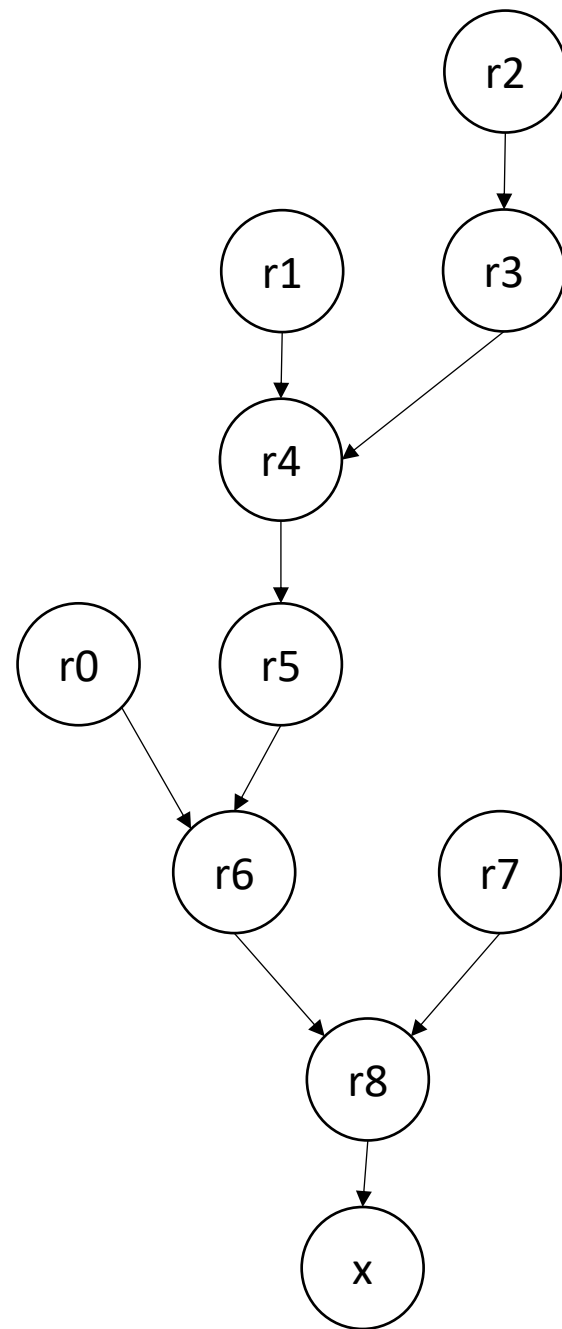
```
r0 = neg(b);  
r1 = b * b;  
r2 = 4 * a;  
r3 = r2 * c;  
r4 = r1 - r3;  
r5 = sqrt(r4);  
r6 = r0 - r5;  
r7 = 2 * a;  
r8 = r6 / r7;  
x  = r8;
```



We can perform more optimizations, example:
by making a data-dependency graph (DDG)

```
r0 = neg(b);  
r1 = b * b;  
r2 = 4 * a;  
r3 = r2 * c;  
r4 = r1 - r3;  
r5 = sqrt(r4);  
r6 = r0 - r5;  
r7 = 2 * a;  
r8 = r6 / r7;  
x  = r8;
```

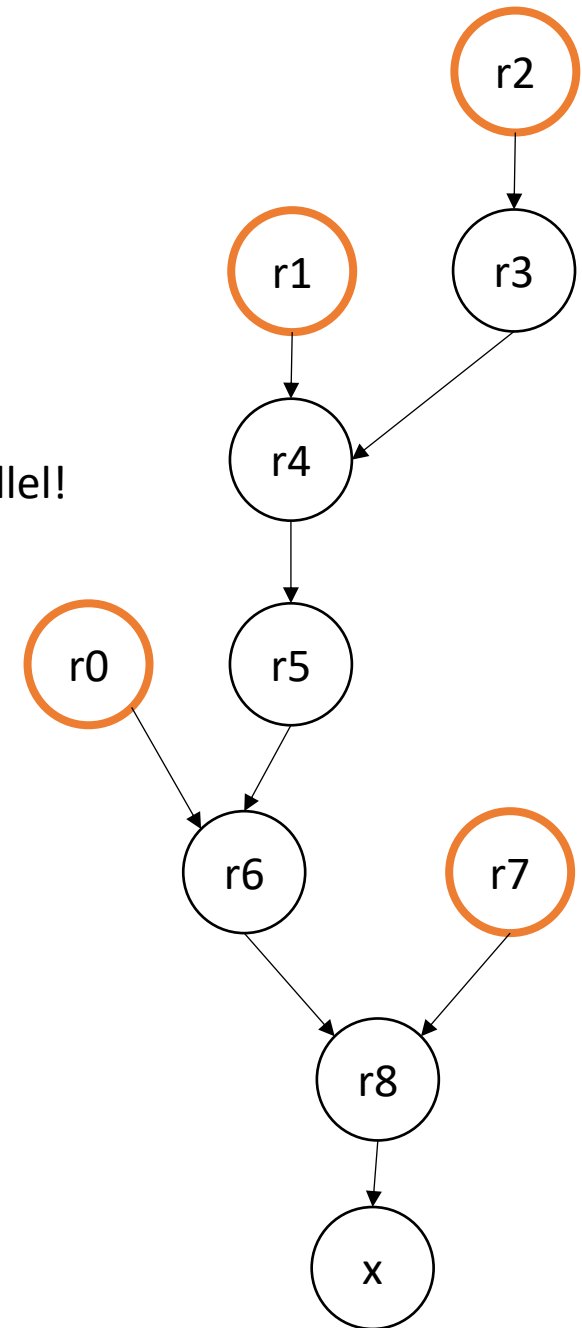
What can this tell us?



We can perform more optimizations, example:
by making a data-dependency graph (DDG)

```
r0 = neg(b);  
r1 = b * b;  
r2 = 4 * a;  
r3 = r2 * c;  
r4 = r1 - r3;  
r5 = sqrt(r4);  
r6 = r0 - r5;  
r7 = 2 * a;  
r8 = r6 / r7;  
x  = r8;
```

can be done in parallel!

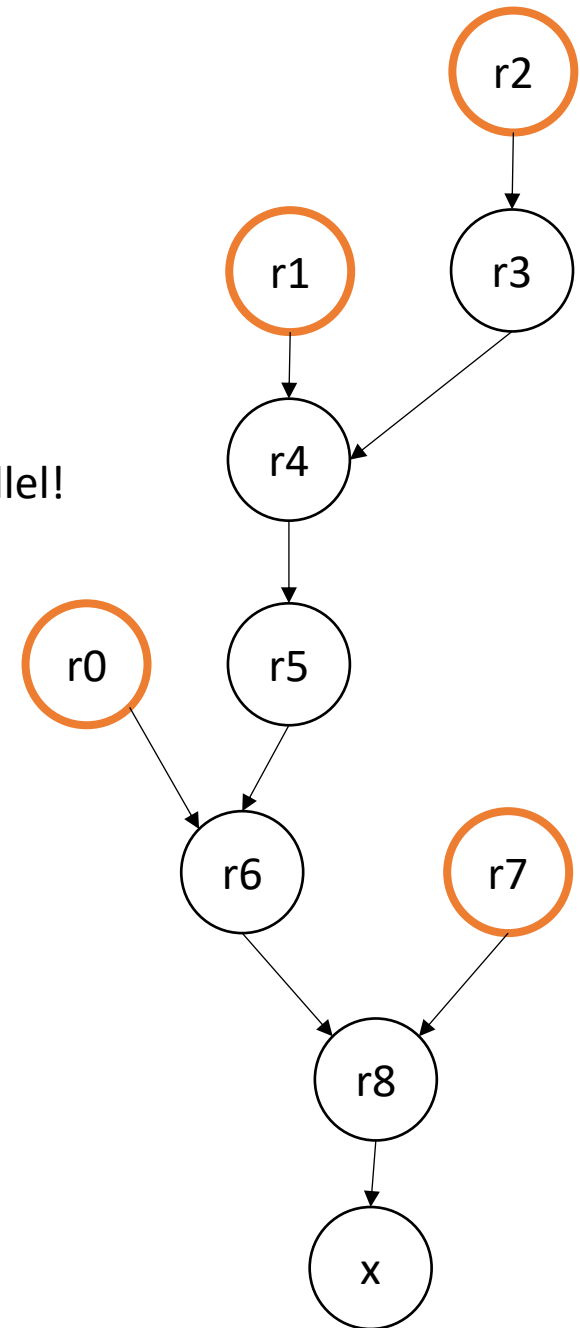


We can perform more optimizations, example:
by making a data-dependency graph (DDG)

```
r0 = neg(b);  
r1 = b * b;  
r2 = 4 * a;  
r3 = r2 * c;  
r4 = r1 - r3;  
r5 = sqrt(r4);  
r6 = r0 - r5;  
r7 = 2 * a;  
r8 = r6 / r7;  
x = r8;
```

Can be hoisted!

can be done in parallel!



We can perform more optimizations, example:
by making a data-dependency graph (DDG)

```
r0 = neg(b);
```

```
r1 = b * b;
```

```
r2 = 4 * a;
```

```
r3 = r2 * c;
```

```
r4 = r1 - r3;
```

```
r5 = sqrt(r4);
```

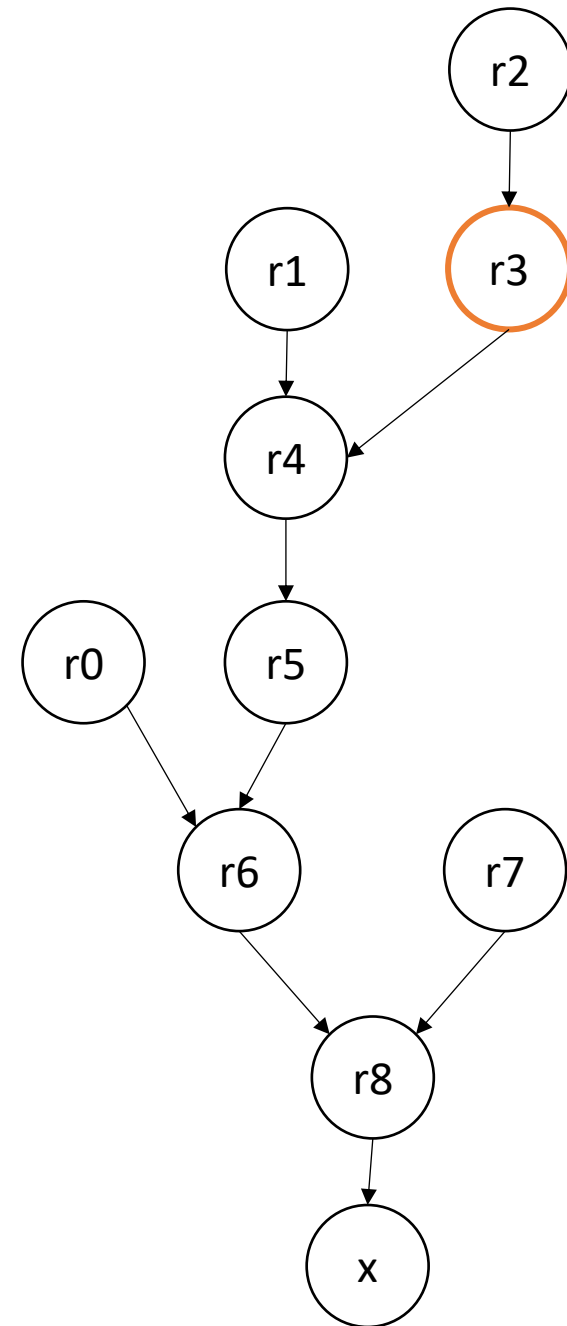
```
r6 = r0 - r5;
```

```
r7 = 2 * a;
```

```
r8 = r6 / r7;
```

```
x = r8;
```

should we hoist this one?



Lots of considerations in optimizing

- More on instruction scheduling later
 - Processor agnostic?
- Back to 3-address code
- We looked at expressions, but how about conditionals?

What about control flow?

- 3 address code typically contains a conditional branch:

```
br <reg>, <label0>, <label1>
```

if the value in <reg> is true, branch to <label0>, else branch to <label1>

```
br <label0>
```

unconditional branch

What about control flow?

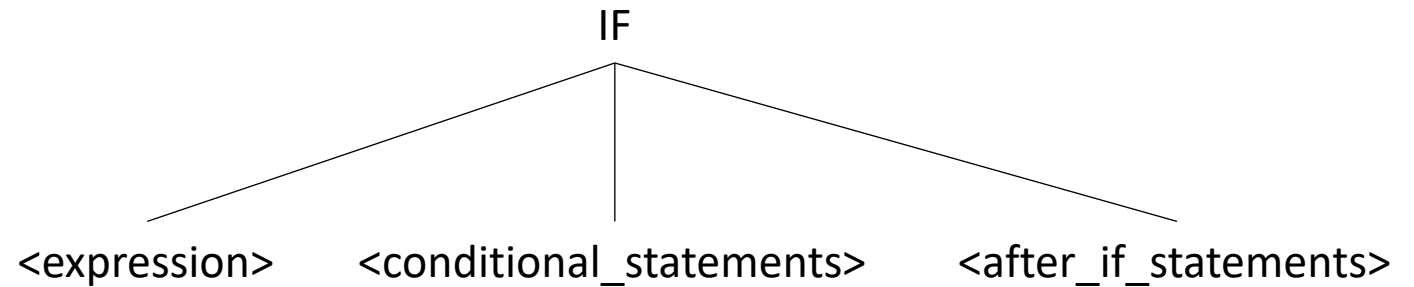
```
if (expr) {  
  // conditional statements  
}  
// after if statements
```

First, produce an AST

What about control flow?

```
if (expr) {  
  // conditional statements  
}  
// after if statements
```

Next lower to 3 address code



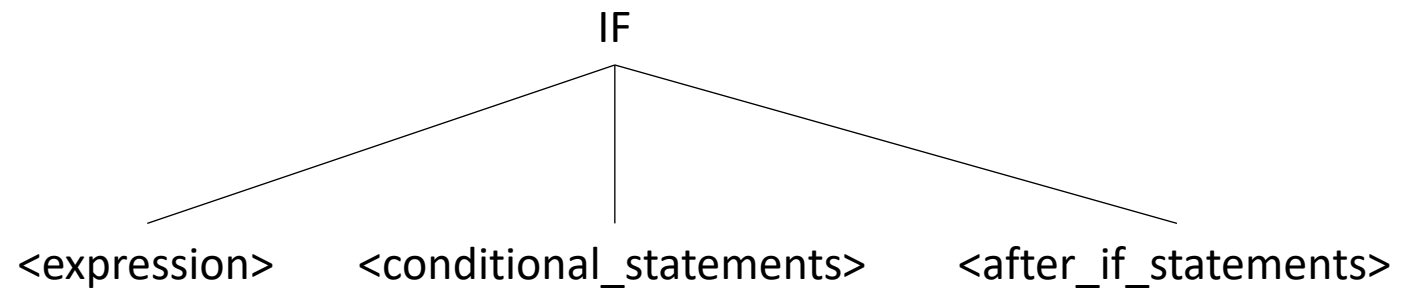
What about control flow?

```
if (expr) {  
    // conditional statements  
}  
// after if statements
```

```
r0 = <expression>;  
br r0, conditional_stmts, after_if;
```

```
conditional_stmts:  
<conditional_statements>;
```

```
after_if:  
<after_if_statements>;
```



What about control flow?

```
while (expr) {  
    // inside_loop_statements  
}  
// after_loop_statements
```

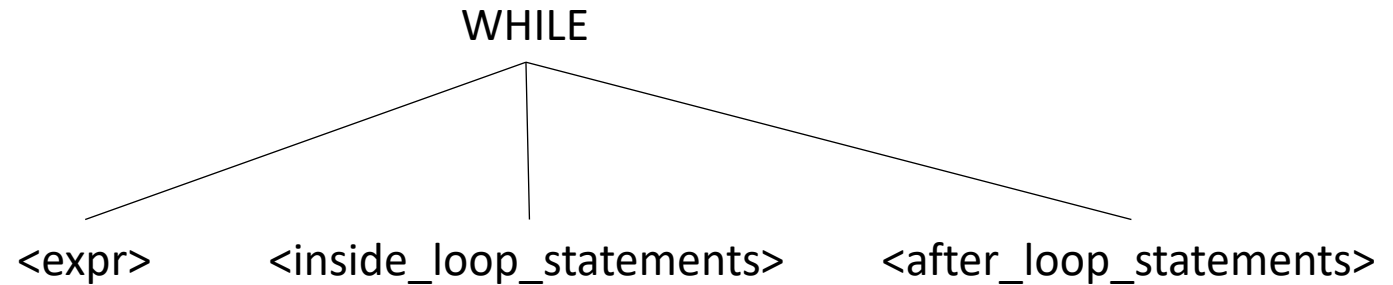
What about control flow?

```
while (expr) {  
  // inside_loop_statements  
}  
// after_loop_statements
```

First, produce an AST

What about control flow?

```
while (expr) {  
  // inside_loop_statements  
}  
// after_loop_statements
```



What about control flow?

```
while (expr) {  
    // inside_loop_statements  
}  
// after_loop_statements
```

beginning_label:

```
r0 = <expr>
```

```
br r0, inside_loop, after_loop;
```

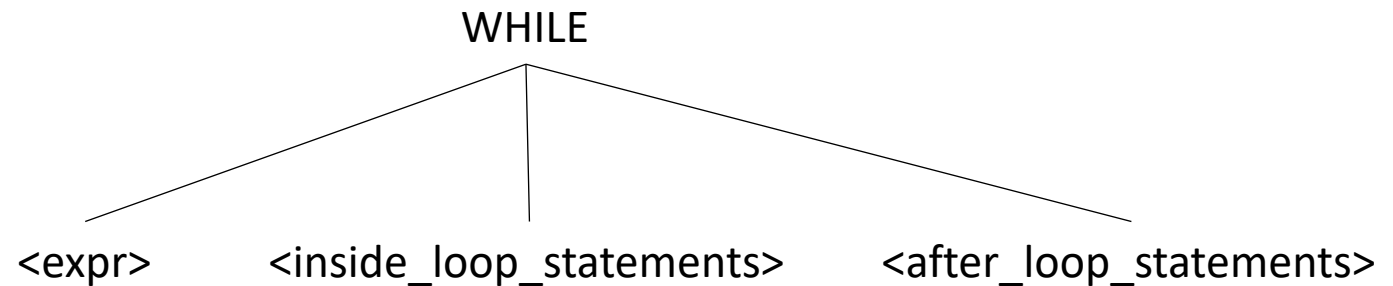
inside_loop:

```
<inside_loop_statements>
```

```
br beginning_label;
```

after_loop:

```
<after_loop_statements>
```

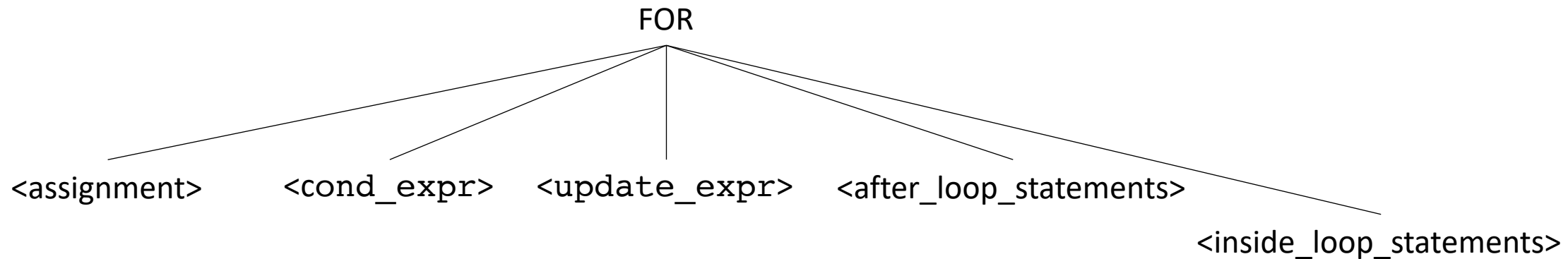


For loop

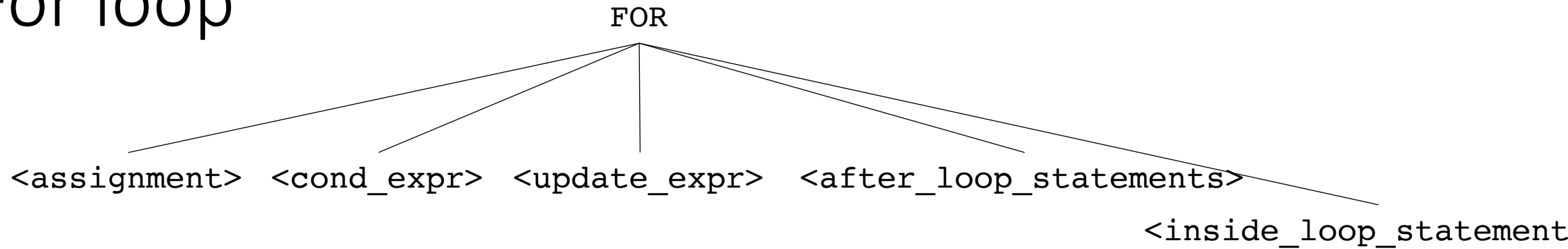
```
for (assignment; cond_expr; update_expr) {  
    // inside_loop_statements  
}  
// after_loop_statements
```

For loop

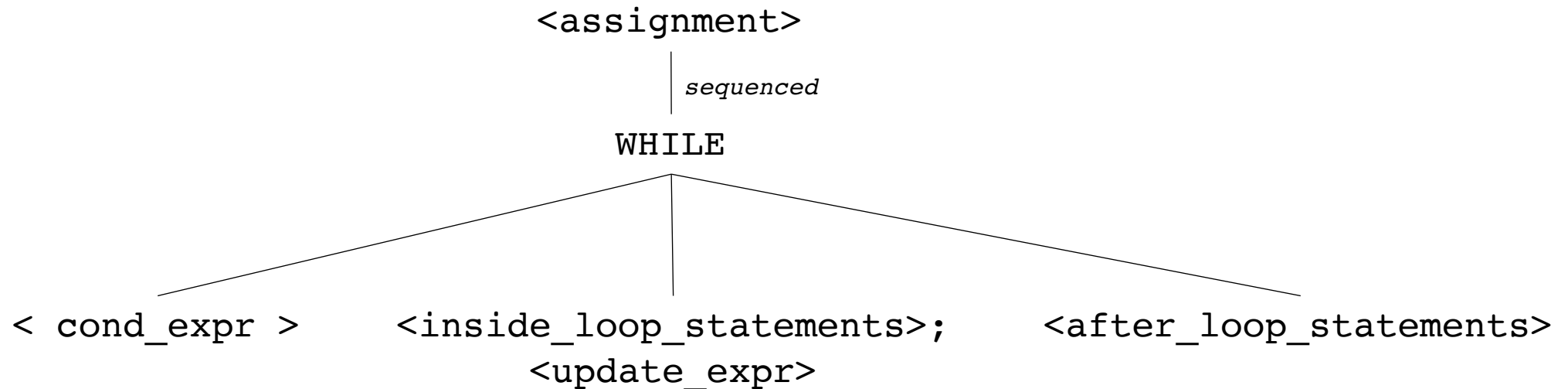
```
for (assignment; cond_expr; update_expr) {  
  // inside_loop_statements  
}  
// after_loop_statements
```



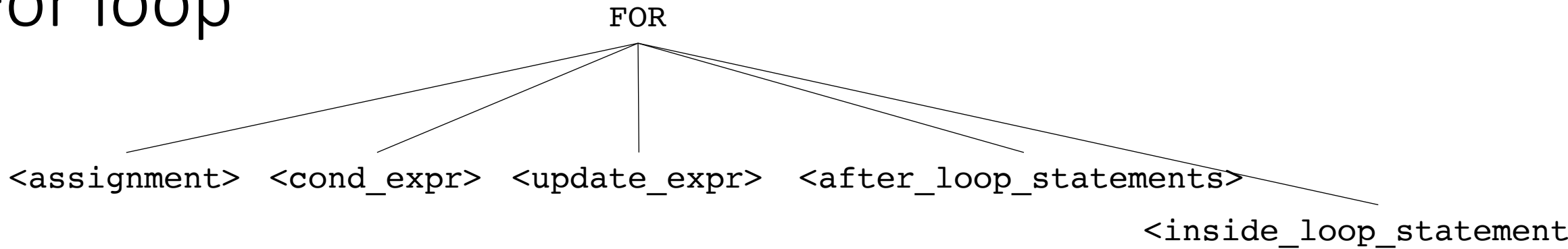
For loop



Can be de-sugared into a while loop:

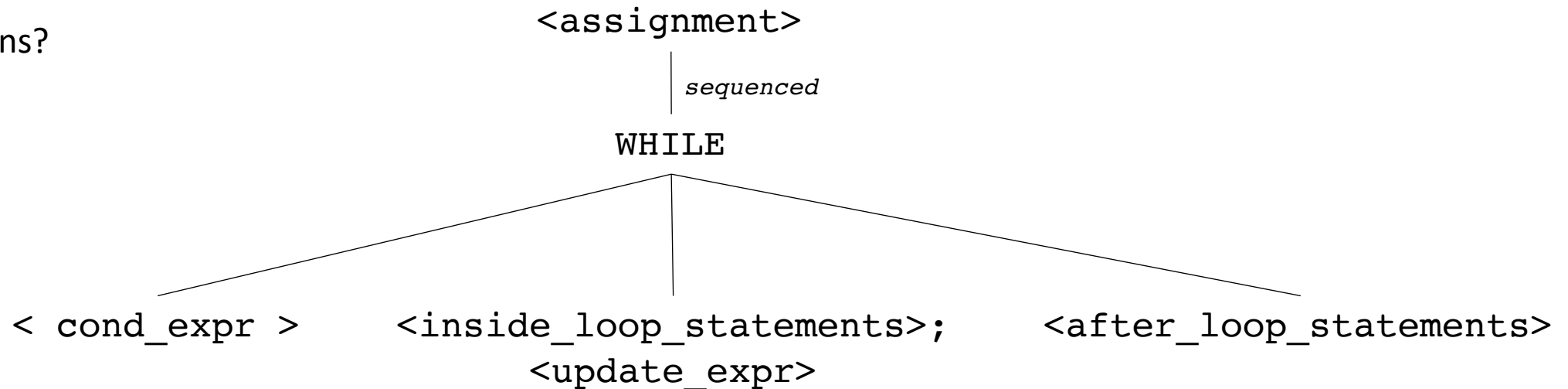


For loop



Can be de-sugared into a while loop:

Pros? Cons?



IR Program structure

- A sequence of 3 address instructions
- Programs can be split into **Basic Blocks**:
 - A sequence of 3 address instructions such that:
 - There is a single entry, single exit
- *Important property*: an instruction in a basic block can assume that all preceding instructions will execute

Single Basic Block

```
Label_x:  
op1;  
op2;  
op3;  
br label_z;
```

IR Program structure

- A sequence of 3 address instructions
- Programs can be split into **Basic Blocks**:
 - A sequence of 3 address instructions such that:
 - There is a single entry, single exit
- *Important property*: an instruction in a basic block can assume that all preceding instructions will execute

Single Basic Block

```
Label_x:  
op1;  
op2;  
op3;  
br label_z;
```

Two Basic Blocks

```
Label_x:  
op1;  
op2;  
op3;  
  
Label_y:  
op4;  
op5;
```


How might they appear in a high-level language? What are some examples?

IR Program structure

- A sequence of 3 address instructions
- Programs can be split into **Basic Blocks**:
 - A sequence of 3 address instructions such that:
 - There is a single entry, single exit

- *Important property*: an instruction in a basic block can assume that all preceding instructions will execute

Single Basic Block

```
Label_x:  
op1;  
op2;  
op3;  
br label_z;
```

Two Basic Blocks

```
Label_x:  
op1;  
op2;  
op3;  
  
Label_y:  
op4;  
op5;
```

IR Program structure

- A sequence of 3 address instructions
- Programs can be split into **Basic Blocks**:
 - A sequence of 3 address instructions such that:
 - There is a single entry, single exit
- *Important property*: an instruction in a basic block can assume that all preceding instructions will execute

How might they appear in a high-level language?

Four Basic Blocks

```
...  
if (x) {  
    ...  
}  
else {  
    ...  
}  
...
```

Two Basic Blocks

Single Basic Block

```
Label_x:  
op1;  
op2;  
op3;  
br label_z;
```

```
Label_x:  
op1;  
op2;  
op3;  
  
Label_y:  
op4;  
op5;
```

Optimization levels

- **Local optimizations:**
 - Optimizes an individual basic block
- **Regional optimizations:**
 - Combines several basic blocks
- **Global optimizations:**
 - operates across an entire procedure
 - what about across procedures?

Optimization levels

```
Label_0:  
x = a + b;  
y = a + b;
```

- **Local optimizations:**
 - Optimizes an individual basic block
- **Regional optimizations:**
 - Combines several basic blocks
- **Global optimizations:**
 - operates across an entire procedure
 - what about across procedures?

Optimization levels

- **Local optimizations:**

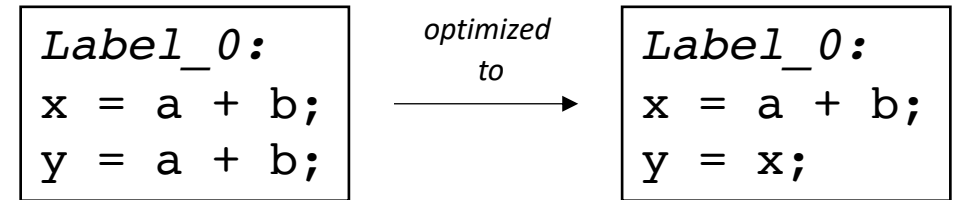
- Optimizes an individual basic block

- **Regional optimizations:**

- Combines several basic blocks

- **Global optimizations:**

- operates across an entire procedure
- what about across procedures?



Optimization levels

- **Local optimizations:**

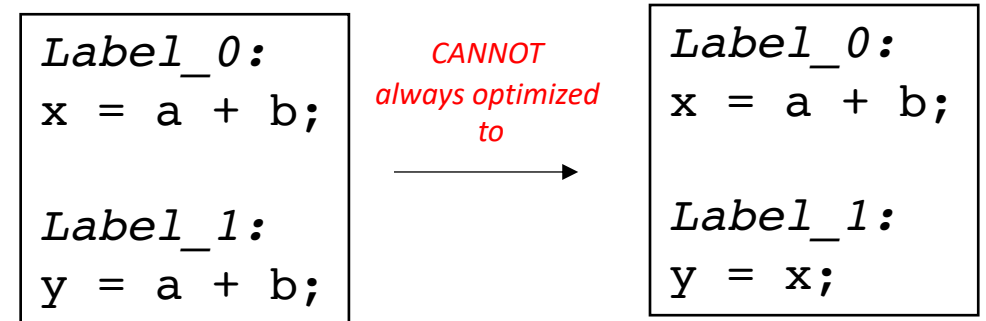
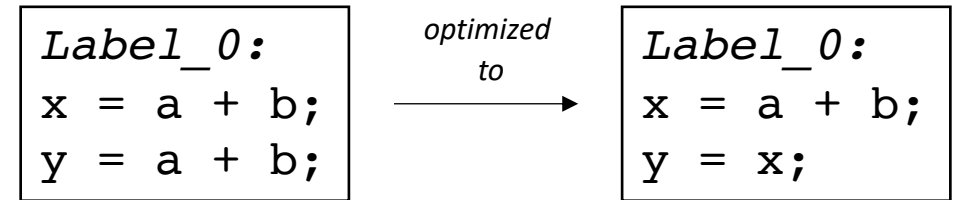
- Optimizes an individual basic block

- **Regional optimizations:**

- Combines several basic blocks

- **Global optimizations:**

- operates across an entire procedure
- what about across procedures?



Optimization levels

- **Local optimizations:**

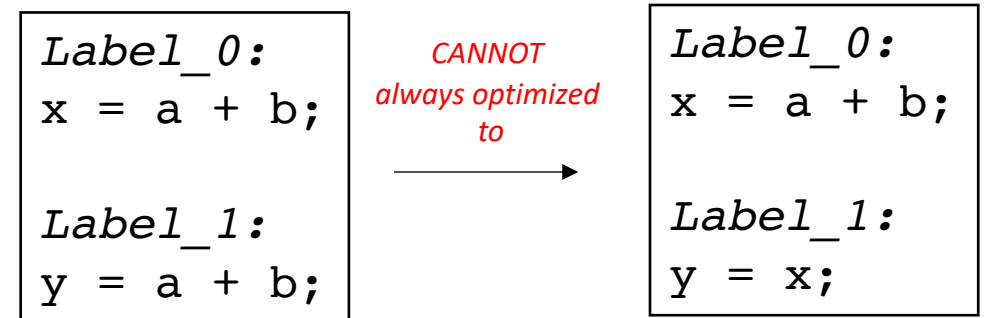
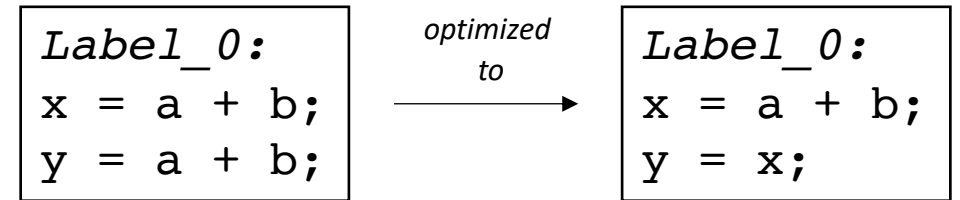
- Optimizes an individual basic block

- **Regional optimizations:**

- Combines several basic blocks

- **Global optimizations:**

- operates across an entire procedure
- what about across procedures?



*code could skip Label_0,
leaving x undefined!*

```
br Label_1;  
  
Label_0:  
x = a + b;  
  
Label_1:  
y = a + b;
```

Regional Optimization

```
...  
if (x) {  
    ...  
}  
else {  
    x = a + b;  
}  
y = a + b;  
...
```

*at a higher-level,
we cannot replace:
y = a + b.
with
y = x;*

Regional Optimization

```
...  
if (x) {  
    ...  
}  
else {  
    x = a + b;  
}  
y = a + b;  
...
```

*at a higher-level,
we cannot replace:
y = a + b.
with
y = x;*

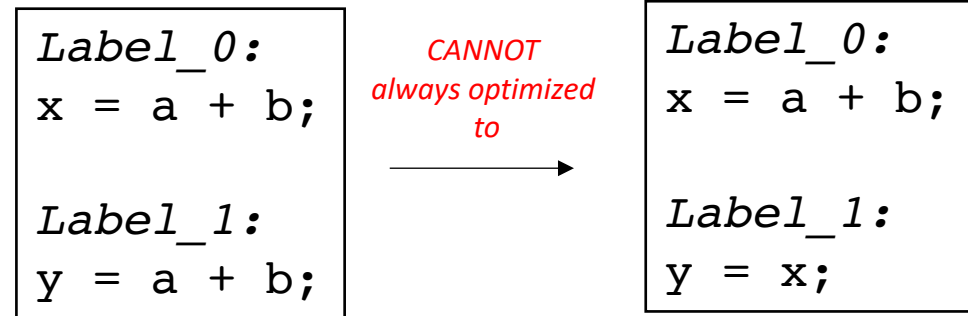
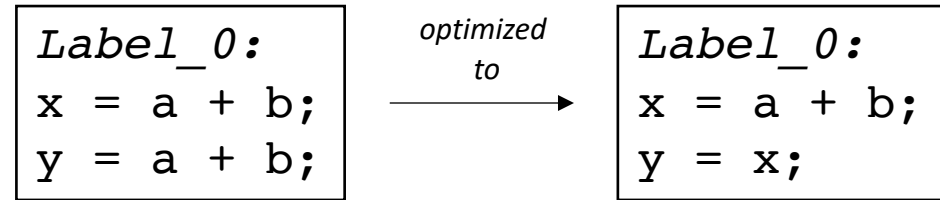
```
x = a + b;  
if (x) {  
    ...  
}  
else {  
    ...  
}  
y = a + b;  
...
```

*But if a and b are
not redefined, then
y = a + b;
can be replaced with
y = x;*

Local optimizations

- Optimizations that occur in a single basic block
 - What property can we exploit?

Local optimizations



*code could skip Label_0,
leaving x undefined!*

```
br Label_1;  
  
Label_0:  
x = a + b;  
  
Label_1:  
y = a + b;
```

Today's lecture: A local optimization

Local value numbering

- A local optimization over 3 address code
- Attempts to replace arithmetic operations (expensive) with copy instructions (cheap)
- Can be extended to a regional optimization using flow analysis
 - We will cover in later lectures.

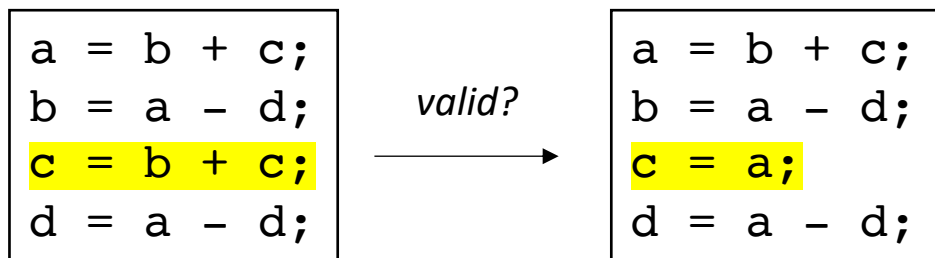
Local value numbering

- A local optimization over 3 address code
- Attempts to replace arithmetic operations (expensive) with copy instructions (cheap)
- Can be extended to a regional optimization using flow analysis
 - We will cover in later lectures.

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

Local value numbering

- A local optimization over 3 address code
- Attempts to replace arithmetic operations (expensive) with copy instructions (cheap)
- Can be extended to a regional optimization using flow analysis
 - We will cover in later lectures.



Local value numbering

- A local optimization over 3 address code
- Attempts to replace arithmetic operations (expensive) with copy instructions (cheap)
- Can be extended to a regional optimization using flow analysis
 - We will cover in later lectures.

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

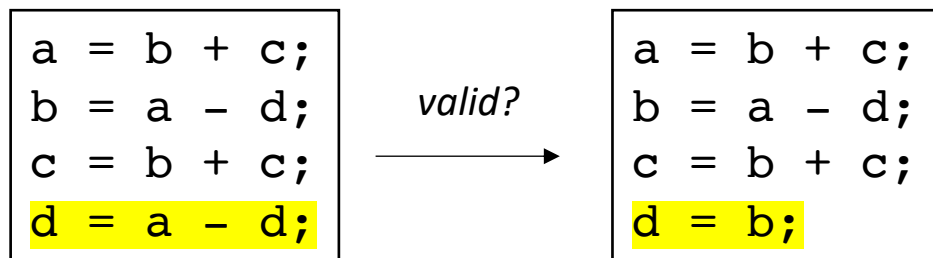
valid? →

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

No! Because b is redefined

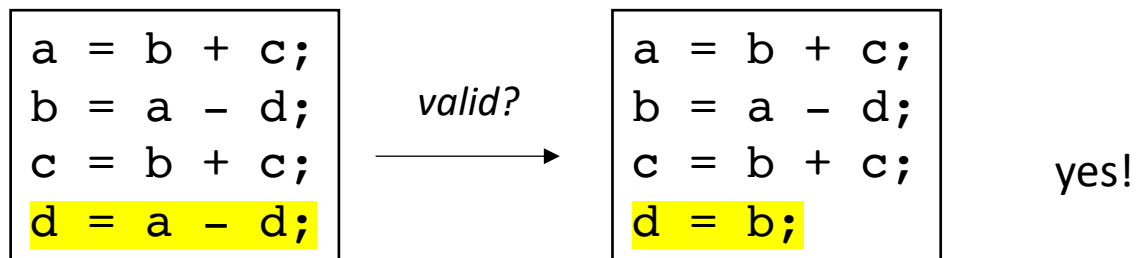
Local value numbering

- A local optimization over 3 address code
- Attempts to replace arithmetic operations (expensive) with copy instructions (cheap)
- Can be extended to a regional optimization using flow analysis
 - We will cover in later lectures.



Local value numbering

- A local optimization over 3 address code
- Attempts to replace arithmetic operations (expensive) with copy instructions (cheap)
- Can be extended to a regional optimization using flow analysis
 - We will cover in later lectures.



Local value numbering

Algorithm:

- Provide a number to each variable. Update the number each time the variable is updated.
- Keep a global counter; increment with new variables or assignments

```
a2 = b0 + c1;  
b4 = a2 - d3;  
c5 = b4 + c1;  
d6 = a2 - d3;
```

Global_counter = 7

Local value numbering

Algorithm:

- Provide a number to each variable. Update the number each time the variable is updated.
- Keep a global counter; increment with new variables or assignments

```
a2 = b0 + c1;  
b4 = a2 - d3;  
c5 = b4 + c1;  
d6 = a2 - d3;
```

Global_counter = 7

Local value numbering

Algorithm: Now that variables are numbered

- Iterate sequentially through instructions. Keep a hash table of the rhs (numbered variables and operation) mapped to their lhs.
- At each step, check to see if the rhs has already been computed.

```
a2 = b0 + c1;  
b4 = a2 - d3;  
c5 = b4 + c1;  
d6 = a2 - d3;
```

Local value numbering

Algorithm: Now that variables are numbered

- Iterate sequentially through instructions. Keep a hash table of the rhs (numbered variables and operation) mapped to their lhs.
- At each step, check to see if the rhs has already been computed.

→

a2 = b0 + c1;
b4 = a2 - d3;
c5 = b4 + c1;
d6 = a2 - d3;

H = {
}

Local value numbering

Algorithm: Now that variables are numbered

- Iterate sequentially through instructions. Keep a hash table of the rhs (numbered variables and operation) mapped to their lhs.
- At each step, check to see if the rhs has already been computed.

→

a2 = b0 + c1;
b4 = a2 - d3;
c5 = b4 + c1;
d6 = a2 - d3;

H = {
 "b0 + c1" : "a2",
}

Local value numbering

Algorithm: Now that variables are numbered

- Iterate sequentially through instructions. Keep a hash table of the rhs (numbered variables and operation) mapped to their lhs.
- At each step, check to see if the rhs has already been computed.

→

a2 = b0 + c1;
b4 = a2 - d3;
c5 = b4 + c1;
d6 = a2 - d3;

H = {
 "b0 + c1" : a2,
}

Local value numbering

Algorithm: Now that variables are numbered

- Iterate sequentially through instructions. Keep a hash table of the rhs (numbered variables and operation) mapped to their lhs.
- At each step, check to see if the rhs has already been computed.

→

a2 = b0 + c1;
b4 = a2 - d3;
c5 = b4 + c1;
d6 = a2 - d3;

H = {
 "b0 + c1" : "a2",
 "a2 - d3" : "b4",
}

Local value numbering

Algorithm: Now that variables are numbered

- Iterate sequentially through instructions. Keep a hash table of the rhs (numbered variables and operation) mapped to their lhs.
- At each step, check to see if the rhs has already been computed.

→

a2 = b0 + c1;
b4 = a2 - d3;
c5 = b4 + c1;
d6 = a2 - d3;

H = {
 "b0 + c1" : "a2",
 "a2 - d3" : "b4",
}

Local value numbering

Algorithm: Now that variables are numbered

- Iterate sequentially through instructions. Keep a hash table of the rhs (numbered variables and operation) mapped to their lhs.
- At each step, check to see if the rhs has already been computed.

→

a2 = b0 + c1;
b4 = a2 - d3;
c5 = b4 + c1;
d6 = a2 - d3;

H = {
"b0 + c1" : "a2",
"a2 - d3" : "b4",
}

mismatch due to numberings!

Local value numbering

Algorithm: Now that variables are numbered

- Iterate sequentially through instructions. Keep a hash table of the rhs (numbered variables and operation) mapped to their lhs.
- At each step, check to see if the rhs has already been computed.

→

a2 = b0 + c1;
b4 = a2 - d3;
c5 = b4 + c1;
d6 = a2 - d3;

H = {
 "b0 + c1" : "a2",
 "a2 - d3" : "b4",
 "b4 + c1" : "c5",
}

Local value numbering

Algorithm: Now that variables are numbered

- Iterate sequentially through instructions. Keep a hash table of the rhs (numbered variables and operation) mapped to their lhs.
- At each step, check to see if the rhs has already been computed.

→

a2 = b0 + c1;
b4 = a2 - d3;
c5 = b4 + c1;
d6 = a2 - d3;

H = {
 "b0 + c1" : "a2",
 "a2 - d3" : "b4",
 "b4 + c1" : "c5",
}

Local value numbering

Algorithm: Now that variables are numbered

- Iterate sequentially through instructions. Keep a hash table of the rhs (numbered variables and operation) mapped to their lhs.
- At each step, check to see if the rhs has already been computed.

→
a2 = b0 + c1;
b4 = a2 - d3;
c5 = b4 + c1;
d6 = b4;

H = {
"b0 + c1" : "a2",
"a2 - d3" : "b4",
"b4 + c1" : "c5",
}
match!

What else can we do?

What else can we do?

Consider this snippet:

```
a2 = c1 - b0;  
f4 = d3 * a2;  
c5 = b0 - c1;  
d6 = a2 * d3;
```


Commutative operations

What is the definition of commutative?

Commutative operations

What is the definition of commutative?

$$x \text{ OP } y == y \text{ OP } x$$

What operators are commutative? Which ones are not?

Adding commutativity to local value numbering

- For commutative operators (e.g. + *), the analysis should consider a deterministic order of operands.
- You can use variable numbers or lexicographical order

Local value numbering: commutative operations

Algorithm optimization:

- for commutative operations, re-order operands into a deterministic order

→

a2	=	c1	-	b0;
f4	=	d3	*	a2;
c5	=	b0	-	c1;
d6	=	a2	*	d3;

H = {
}

Local value numbering: commutative operations

Algorithm optimization:

- for commutative operations, re-order operands into a deterministic order

cannot re-order because - is not commutative

→

a2	=	c1	-	b0;
f4	=	d3	*	a2;
c5	=	b0	-	c1;
d6	=	a2	*	d3;

H = {
 "b0 - c1" : "c5",
 "a2 * d3" : "d6",
}

Local value numbering: commutative operations

Algorithm optimization:

- for commutative operations, re-order operands into a deterministic order

→

a2	=	c1	-	b0;
f4	=	d3	*	a2;
c5	=	b0	-	c1;
d6	=	a2	*	d3;

H = {
 "b0 - c1" : "c5",
 "a2 * d3" : "d6",
}

Local value numbering: commutative operations

Algorithm optimization:

- for commutative operations, re-order operands into a deterministic order

→

a2 = c1 - b0;
f4 = d3 * a2;
c5 = b0 - c1;
d6 = a2 * d3;

re-ordered because a2 < d3 lexicographically

```
H = {  
    "c1 - b0" : "a2",  
    "a2 * d3" : "f4",  
}
```

Local value numbering: commutative operations

Algorithm optimization:

- for commutative operations, re-order operands into a deterministic order

→

a2 = c1 - b0;
f4 = d3 * a2;
c5 = b0 - c1;
d6 = a2 * d3;

```
H = {  
    "c1 - b0" : "a2",  
    "a2 * d3" : "f4",  
}
```


Local value numbering: commutative operations

Algorithm optimization:

- for commutative operations, re-order operands into a deterministic order

→

a2 = c1 - b0;
f4 = d3 * a2;
c5 = b0 - c1;
d6 = a2 * d3;

H = {
 "b0 - c1" : "c5",
 "a2 * d3" : "f4",
 "c1 - b0" : "a2",
}

Local value numbering: commutative operations

Algorithm optimization:

- for commutative operations, re-order operands into a deterministic order

```
a2 = c1 - b0;  
f4 = d3 * a2;  
c5 = b0 - c1;  
→ d6 = a2 * d3;
```

```
H = {  
    "c1 - b0" : "a2",  
    "a2 * d3" : "f4",  
    "b0 - c1" : "c5",  
}
```

Local value numbering: commutative operations

Algorithm optimization:

- for commutative operations, re-order operands into a deterministic order

```
a2 = c1 - b0;  
f4 = d3 * a2;  
c5 = b0 - c1;  
→ d6 = f4;
```

```
H = {  
    "c1 - b0" : "a2",  
    "a2 * d3" : "f4",  
    "b0 - c1" : "c5",  
}
```

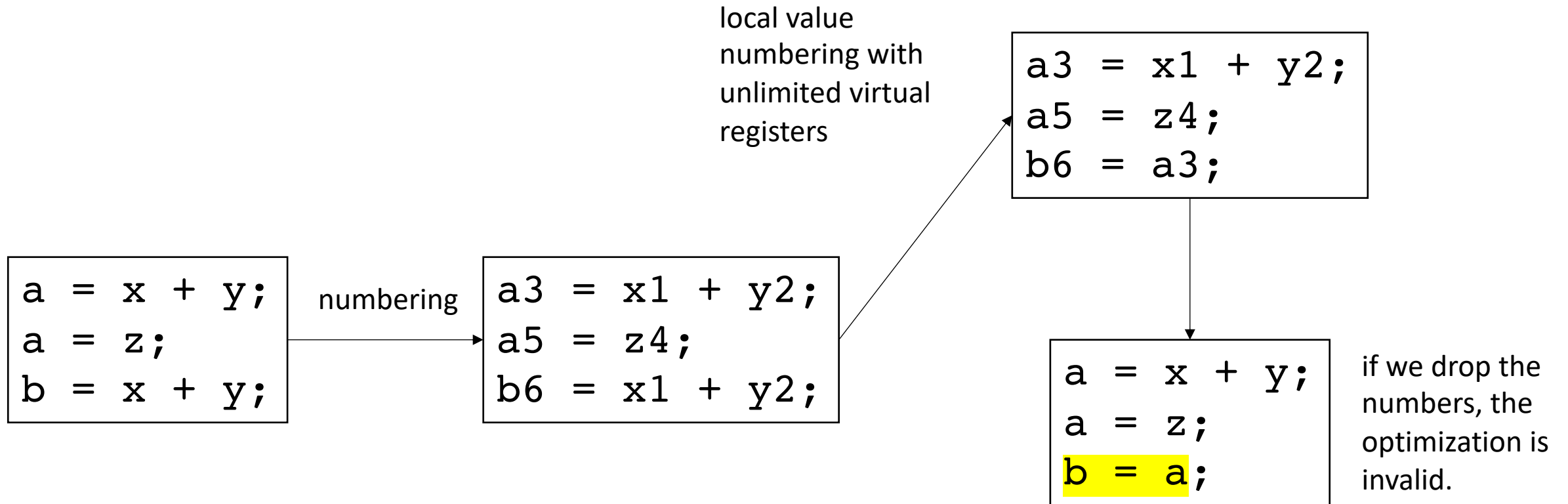
Other considerations?

Local value numbering w/out adding registers

- We've assumed we have access to an unlimited number of virtual registers.
- In some cases we may not be able to add virtual registers
 - If an expensive register allocation pass has already occurred.
- New constraint:
 - We need to produce a program such that variables without the numbers is still valid.

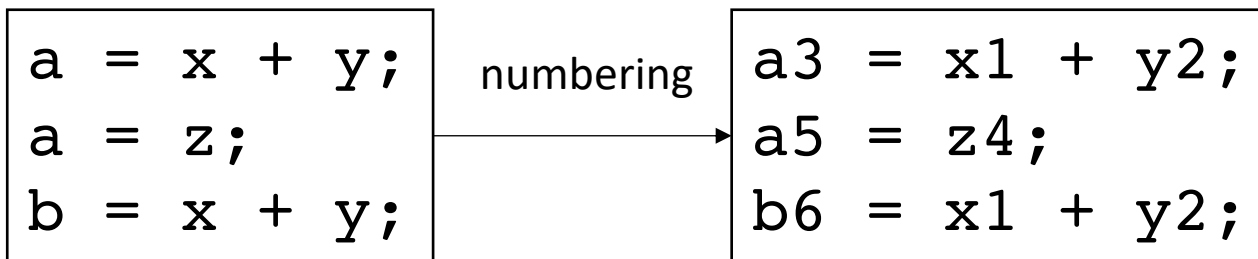
Local value numbering w/out adding registers

- Example:



Local value numbering w/out adding registers

- Solutions?



Local value numbering w/out adding registers

- Keep another hash table to keep the current variable number

```
a = x + y;  
a = z;  
b = x + y;  
c = x + y;
```


Local value numbering w/out adding registers

- Keep another hash table to keep the current variable number

```
a = x + y;  
a = z;  
b = x + y;  
c = x + y;
```

We cannot optimize the first line, but we can optimize the second

Local value numbering w/out adding registers

- Keep another hash table to keep the current variable number

```
a = x + y;  
a = z;  
b = x + y;  
c = x + y;
```

Local value numbering w/out adding registers

- Keep another hash table to keep the current variable number

```
a = x + y;  
a = z;  
b = x + y;  
c = x + y;
```

First we number

Local value numbering w/out adding registers

- Keep another hash table to keep the current variable number

```
a3 = x1 + y2;  
a5 = z4;  
b6 = x1 + y2;  
c7 = x1 + y2;
```

Local value numbering w/out adding registers

- Keep another hash table to keep the current variable number

```
Current_val = {  
}
```

→

```
a3 = x1 + y2;  
a5 = z4;  
b6 = x1 + y2;  
c7 = x1 + y2;
```

```
H = {  
}
```

Local value numbering w/out adding registers

- Keep another hash table to keep the current variable number

```
Current_val = {  
    "a" : 3,  
}
```

```
H = {  
    "x1 + y2" : "a3",  
}
```

→

```
a3 = x1 + y2;  
a5 = z4;  
b6 = x1 + y2;  
c7 = x1 + y2;
```

Local value numbering w/out adding registers

- Keep another hash table to keep the current variable number

→

<pre>a3 = x1 + y2; a5 = z4; b6 = x1 + y2; c7 = x1 + y2;</pre>

```
Current_val = {  
    "a" : 3,  
}
```

```
H = {  
    "x1 + y2" : "a3",  
}
```

Local value numbering w/out adding registers

- Keep another hash table to keep the current variable number

→

<pre>a3 = x1 + y2; a5 = z4; b6 = x1 + y2; c7 = x1 + y2;</pre>

```
Current_val = {  
    "a" : 5,  
}
```

```
H = {  
    "x1 + y2" : "a3",  
}
```


Local value numbering w/out adding registers

- Keep another hash table to keep the current variable number

```
Current_val = {  
    "a" : 5,  
}
```

```
H = {  
    "x1 + y2" : "a3",  
}
```

→

<pre>a3 = x1 + y2; a5 = z4; b6 = x1 + y2; c7 = x1 + y2;</pre>

Local value numbering w/out adding registers

- Keep another hash table to keep the current variable number

→

<pre>a3 = x1 + y2; a5 = z4; b6 = x1 + y2; c7 = x1 + y2;</pre>

```
Current_val = {  
    "a" : 5,  
}
```

```
H = {  
    "x1 + y2" : "a3",  
}
```

Local value numbering w/out adding registers

- Keep another hash table to keep the current variable number

→

<pre>a3 = x1 + y2; a5 = z4; b6 = x1 + y2; c7 = x1 + y2;</pre>

```
Current_val = {  
    "a" : 5,  
    "b" : 6  
}  
  
H = {  
    "x1 + y2" : "b6",  
}
```

Local value numbering w/out adding registers

- Keep another hash table to keep the current variable number

→

a3 = x1 + y2;
a5 = z4;
b6 = x1 + y2;
c7 = x1 + y2;

```
Current_val = {  
    "a" : 5,  
    "b" : 6  
}  
  
H = {  
    "x1 + y2" : "b6",  
}
```

Local value numbering w/out adding registers

- Keep another hash table to keep the current variable number

→

a3 = x1 + y2;
a5 = z4;
b6 = x1 + y2;
c7 = x1 + y2;

```
Current_val = {  
    "a" : 5,  
    "b" : 6  
}  
  
H = {  
    "x1 + y2" : "b6",  
}
```

Local value numbering w/out adding registers

- Keep another hash table to keep the current variable number

→

<pre>a3 = x1 + y2; a5 = z4; b6 = x1 + y2; c7 = b6;</pre>
--

```
Current_val = {  
    "a" : 5,  
    "b" : 6  
}  
  
H = {  
    "x1 + y2" : "b6",  
}
```

Anything else we can add to local value numbering?

Anything else we can add to local value numbering?

- Final heuristic: keep sets of possible values

Local value numbering: value sets

- Final heuristic: keep sets of possible values

```
Current_val = {  
}
```

```
a = x + y;  
b = x + y;  
a = z;  
c = x + y;
```

```
H = {  
}
```

Local value numbering: value sets

- Final heuristic: keep sets of possible values

```
Current_val = {  
}
```

```
a3 = x1 + y2;  
b4 = x1 + y2;  
a6 = z5;  
c7 = x1 + y2;
```

```
H = {  
}
```

Local value numbering: value sets

- Final heuristic: keep sets of possible values

→ `a3 = x1 + y2;
b4 = a3;
a6 = z5;
c7 = x1 + y2;`

```
Current_val = {  
    "a" : 6,  
    "b" : 4  
}
```

```
H = {  
    "x1 + y2" : "a3"  
}
```

Local value numbering: value sets

- Final heuristic: keep sets of possible values

→

<pre>a3 = x1 + y2; b4 = a3; a6 = z5; c7 = x1 + y2;</pre>
--

```
Current_val = {  
    "a" : 6,  
    "b" : 4  
}
```

```
H = {  
    "x1 + y2" : "a3"  
}
```

Local value numbering: value sets

- Final heuristic: keep sets of possible values

→

<pre>a3 = x1 + y2; b4 = a3; a6 = z5; c7 = x1 + y2;</pre>
--

```
Current_val = {  
    "a" : 6,  
    "b" : 4  
}
```

```
H = {  
    "x1 + y2" : "a3"  
}
```

but we could have
replaced it with b4!

Local value numbering: value sets

- Final heuristic: keep sets of possible values

```
Current_val = {  
    "a" : 3,  
}
```

rewind to
this point

```
a3 = x1 + y2;  
b4 = x1 + y2;  
a6 = z5;  
c7 = x1 + y2;
```

```
H = {  
    "x1 + y2" : "a3"  
}
```

Local value numbering: value sets

- Final heuristic: keep sets of possible values

→
a3 = x1 + y2;
b4 = a3;
a6 = z5;
c7 = x1 + y2;

```
Current_val = {  
    "a" : 3,  
    "b" : 4  
}
```

```
H = {  
    "x1 + y2" : ["a3", "b4"],  
}
```

hash a list of possible values

Local value numbering: value sets

- Final heuristic: keep sets of possible values

fast forward
again



```
a3 = x1 + y2;  
b4 = a3;  
a6 = z5;  
c7 = x1 + y2;
```

```
Current_val = {  
    "a" : 6,  
    "b" : 4  
}
```

```
H = {  
    "x1 + y2" : ["a3", "b4"],  
}
```


Local value numbering: value sets

- Final heuristic: keep sets of possible values

fast forward
again



```
a3 = x1 + y2;  
b4 = a3;  
a6 = z5;  
c7 = b4;
```

```
Current_val = {  
    "a" : 6,  
    "b" : 4  
}  
  
H = {  
    "x1 + y2" : ["a3", "b4"],  
}
```

Local value numbering: Memory

- Consider a 3 address code that allows memory accesses

```
a[i] = x[j] + y[k];  
b[i] = x[j] + y[k];
```

is this transformation allowed?
No!

```
a[i] = x[j] + y[k];  
b[i] = a[i];
```

only if the compiler can prove that a does not alias x and y

In the worst case, every time a memory location is updated, the compiler must update the value for all pointers.

Local value numbering: Memory

- How to number:
 - Number each pointer/index pair

```
(a[i],3) = (x[j],1) + (y[k],2);  
b[i] = x[j] + y[k];
```

Local value numbering: Memory

- How to number:
 - Number each pointer/index pair
 - Any pointer/index pair that might alias must be incremented at each instruction

```
(a[i], 3) = (x[j], 1) + (y[k], 2);  
(b[i], 6) = (x[j], 4) + (y[k], 5);
```

Local value numbering: Memory

- How to number:
 - Number each pointer/index pair
 - Any pointer/index pair that might alias must be incremented at each instruction

```
(a[i], 3) = (x[j], 1) + (y[k], 2);  
(b[i], 6) = (x[j], 4) + (y[k], 5);
```

compiler analysis:

```
can we trace a, x, y to  
a = malloc(...);  
x = malloc(...);  
y = malloc(...);
```

```
// a, x, y are never overwritten
```

Local value numbering: Memory

- How to number:
 - Number each pointer/index pair
 - Any pointer/index pair that might alias must be incremented at each instruction

```
(a[i], 3) = (x[j], 1) + (y[k], 2);  
(b[i], 6) = (x[j], 1) + (y[k], 2);
```

in this case we do not have to update the number

compiler analysis:

can we trace a, x, y to

```
a = malloc(...);
```

```
x = malloc(...);
```

```
y = malloc(...);
```

```
// a, x, y are never overwritten
```

Local value numbering: Memory

- How to number:
 - Number each pointer/index pair
 - Any pointer/index pair that might alias must be incremented at each instruction

```
(a[i], 3) = (x[j], 1) + (y[k], 2);  
(b[i], 6) = (x[j], 4) + (y[k], 5);
```

programmer annotations can also tell the compiler that no other pointer can access the memory pointed to by a

Local value numbering: Memory

- How to number:
 - Number each pointer/index pair
 - Any pointer/index pair that might alias must be incremented at each instruction

```
(a[i], 3) = (x[j], 1) + (y[k], 2);  
(b[i], 6) = (x[j], 4) + (y[k], 5);
```

in this case we do not have to update the number

`restrict a`

programmer annotations can also tell the compiler that no other pointer can access the memory pointed to by a

Local value numbering: Memory

- How to number:
 - Number each pointer/index pair
 - Any pointer/index pair that might alias must be incremented at each instruction

```
(a[i], 3) = (x[j], 1) + (y[k], 2);  
(b[i], 6) = (a[i], 3);
```

Optimizing over wider regions

- Local value numbering operated over just one basic block.
- We want optimizations that operate over several basic blocks (a region), or across an entire procedure (global)
- For this, we need Control Flow Graphs and Flow Analysis