CSE211: Compiler Design

Oct. 20, 2020

- **Topic**: ASTs and 3 address code.
  Local value numbering

- **Questions**:

  Questions/comments about homework?

  What are some compiler optimizations you know about?

Announcements

- Module 2 has been revamped

- Homeworks:
  - Homework 2 will be posted on Oct. 22
  - Homework 1 is due on Oct. 29

- Thanks to those who have posted!

- Come to the LSD seminar!

Module 2

- This week:
  - 3 address code
  - local value numbering

- Next week:
  - Flow analysis

- Third week:
  - SSA
  - Homework overview

CSE211: Compiler Design

Oct. 20, 2020

- **Topic**: ASTs and 3 address code.
  Local value numbering

- **Questions**:

  Questions/comments about homework?

  What are some compiler optimizations you know about?

```
float hoist = z[const];
for (…) {
   x[i] = y[i] * hoist;
}
```

3 address code IR

- Each instruction consists of 3 “addresses”
  - Address here means a virtual register or value

- 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

- 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

- 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
- 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
- Other information:
    - Annotated
    - Typed
    - Alignment

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

3 address code IR

- Each instruction consists of 3 "addresses"
  - Address here means a virtual register or value

- Control flow: branches and labels:

```
br r0, label1, label2;

br label1;
```

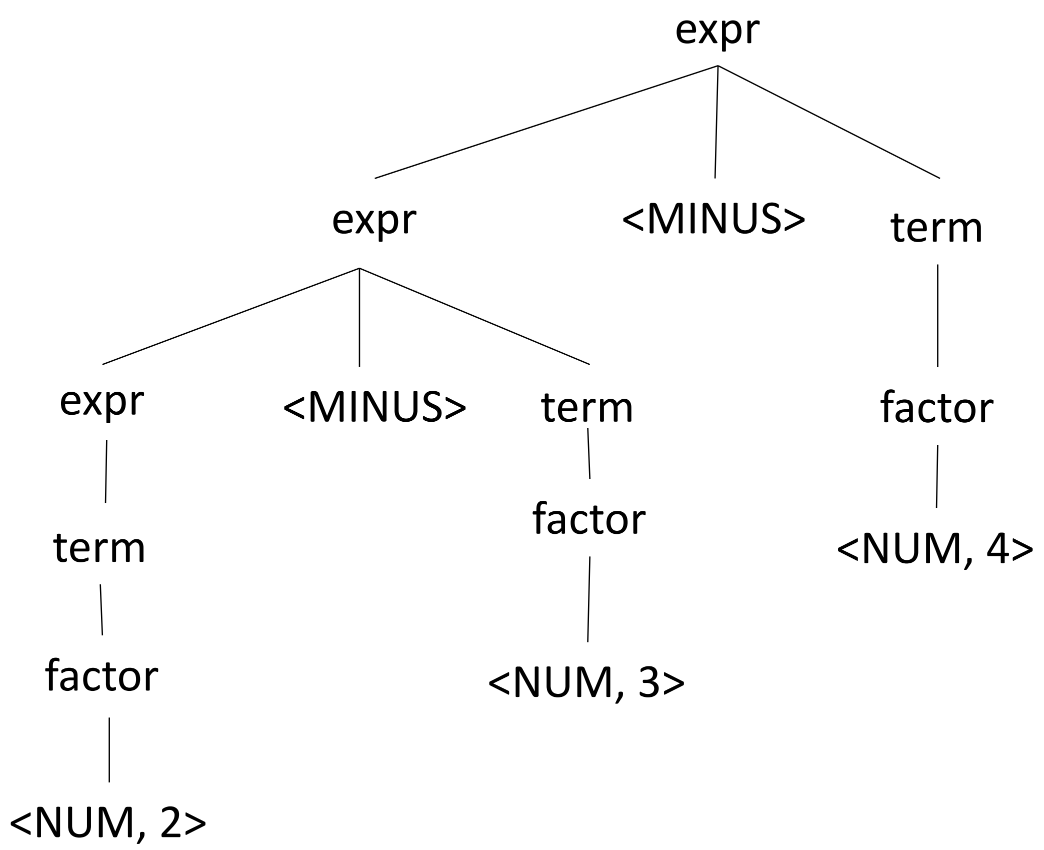
Creating 3 address code from AST

Abstract Syntax Trees

- Remember the expression parse tree

Operator	Name	Productions
+,-	Expr	: Expr + Term | Expr - Term | Term
*,/	Term	: Term * Pow : Term / Pow | Pow
^	Pow	: Factor ^ Pow | Factor
()	Factor	: (Expr) | NUM

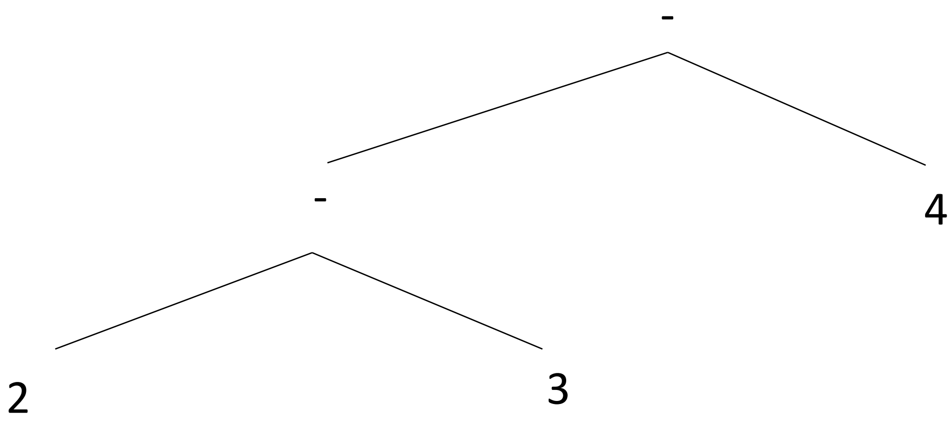
`input: 2-3-4`



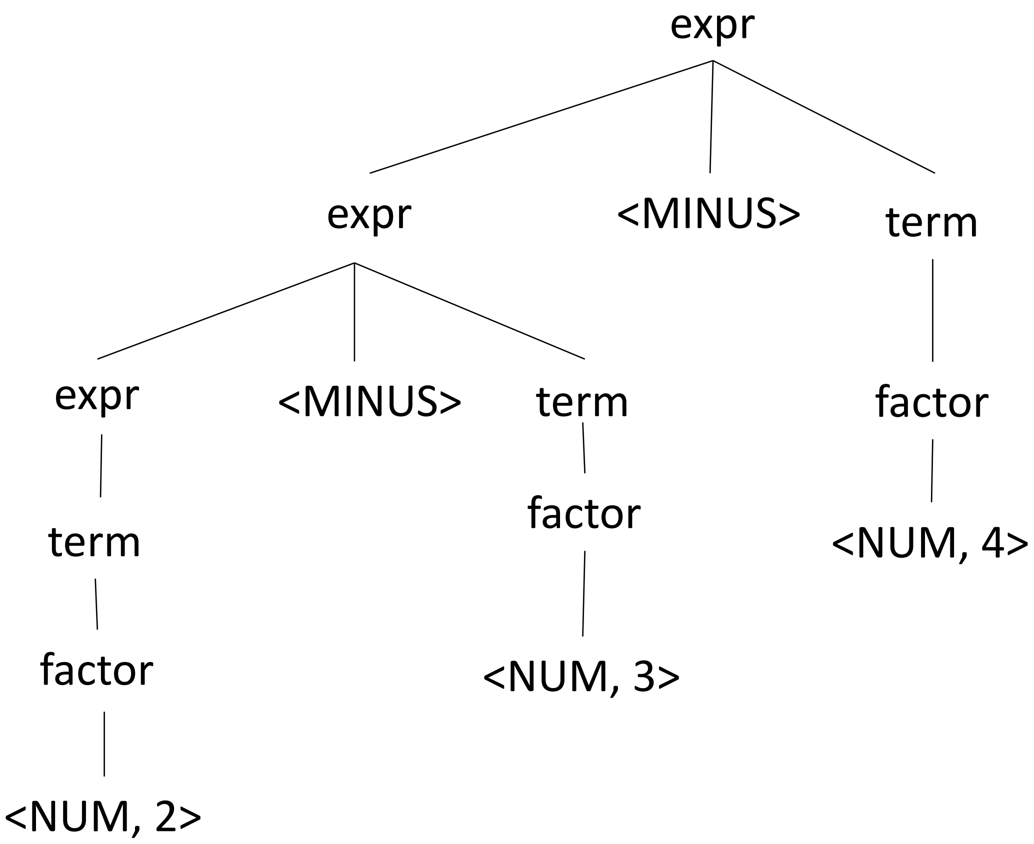
Abstract Syntax Trees

- Remember the expression parse tree

`input: 2-3-4`



Much more compact!

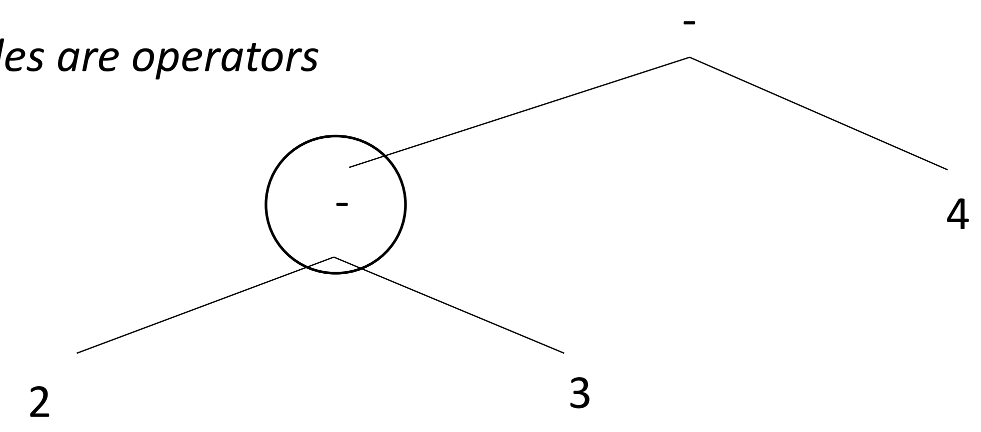


Abstract Syntax Trees

- Remember the expression parse tree

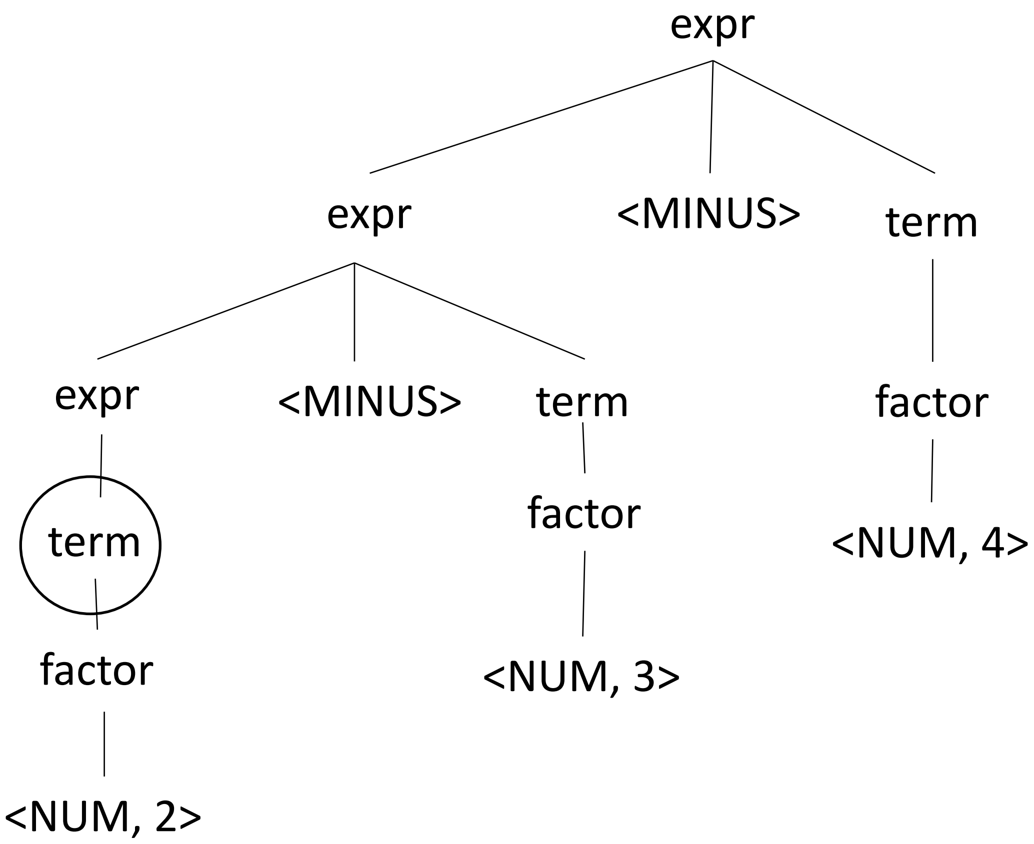
`input: 2-3-4`

nodes are operators



Much more compact!

nodes are production rules



Abstract Syntax Trees

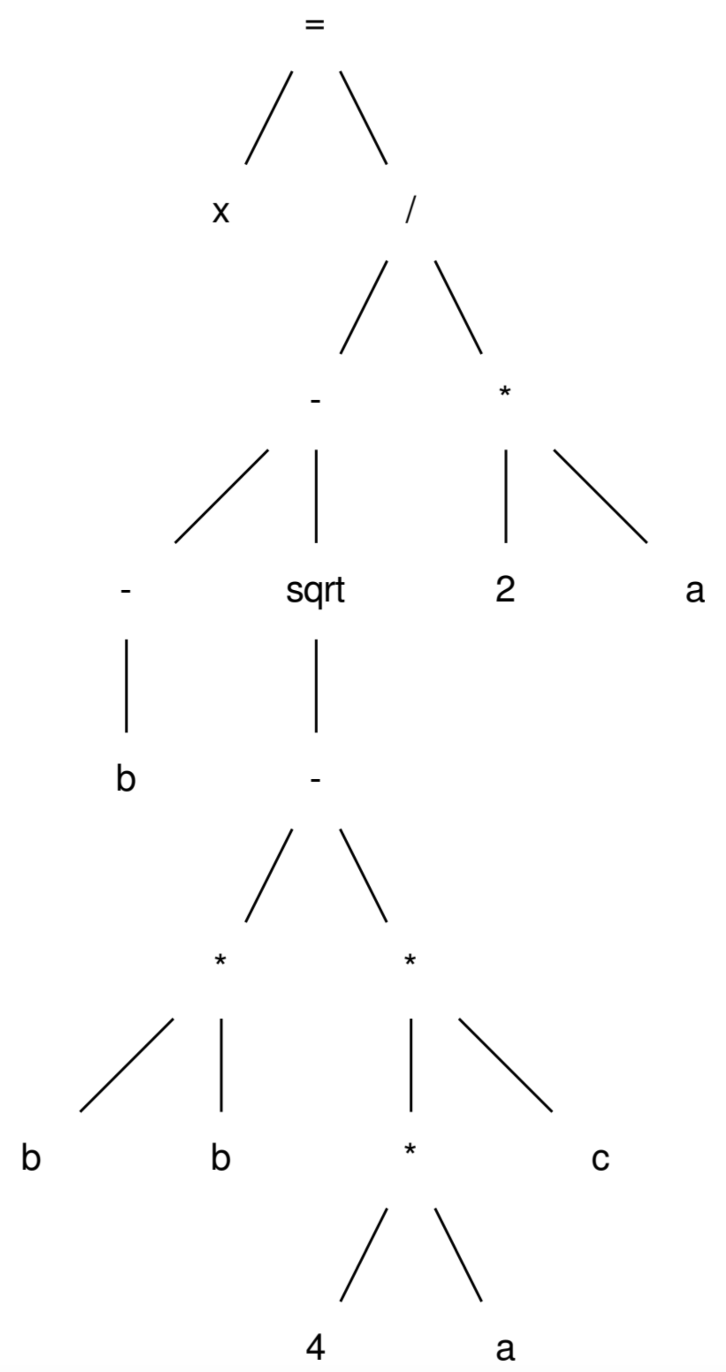
- Easier to see bigger trees, e.g. quadratic formula:

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

x = (-b - sqrt(b*b - 4 * a * c)) / (2*a)

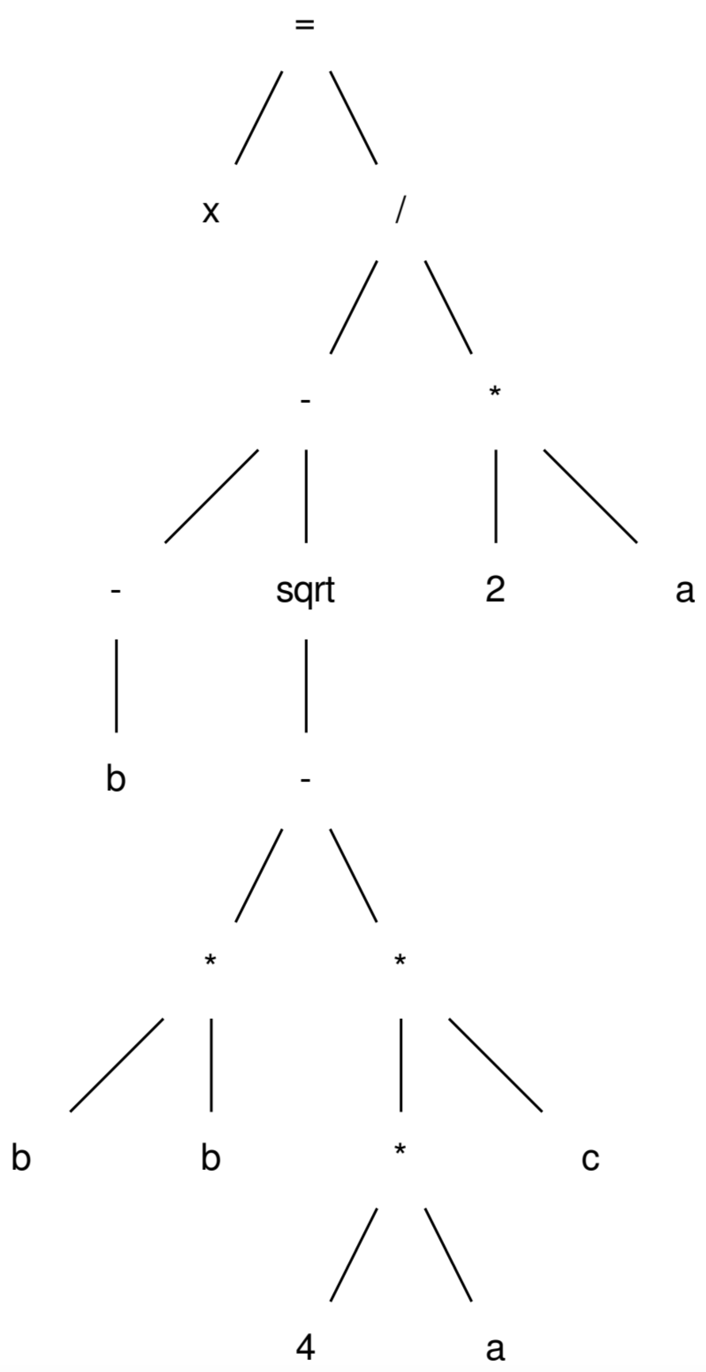
Thanks to Sreepathi Pai for the example idea!

x = (-b - sqrt(b*b - 4 * a * c)) / (2*a)



Convert this code to 3 address code

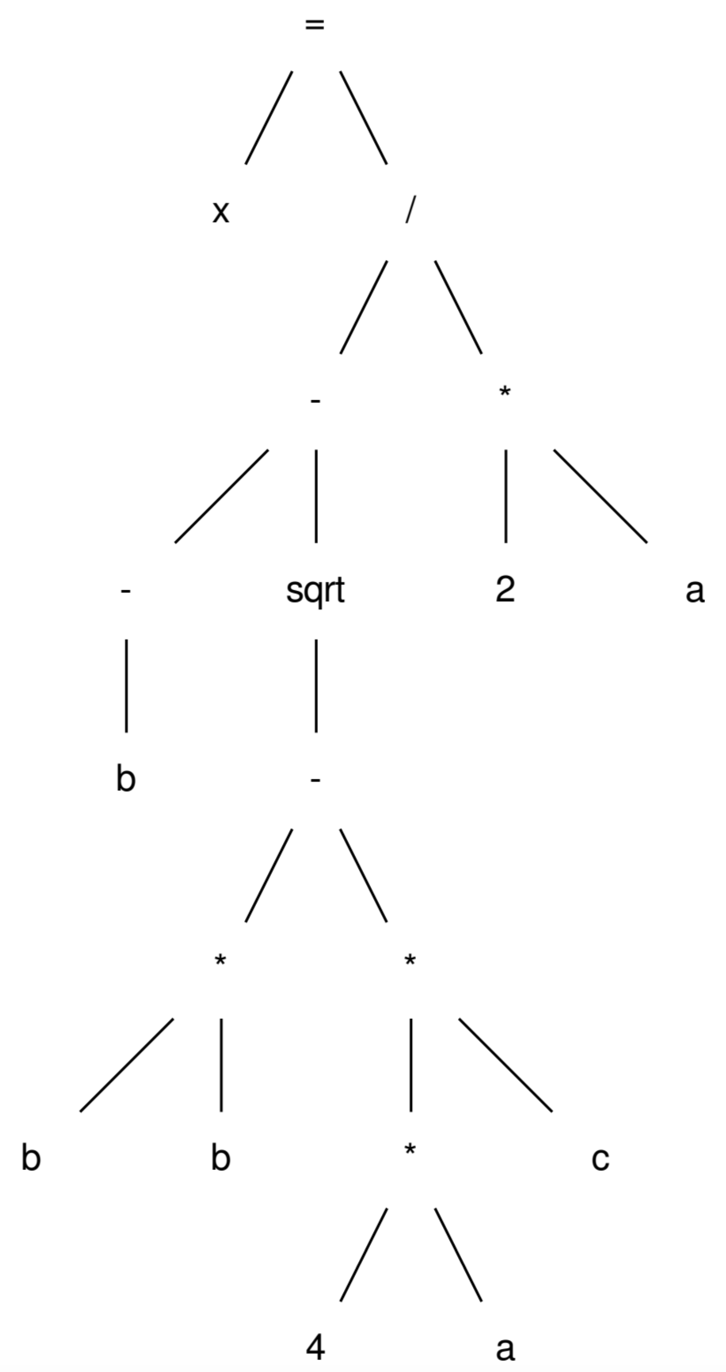
post-order traversal, creating virtual registers



Convert this code to 3 address code

post-order traversal, creating virtual registers

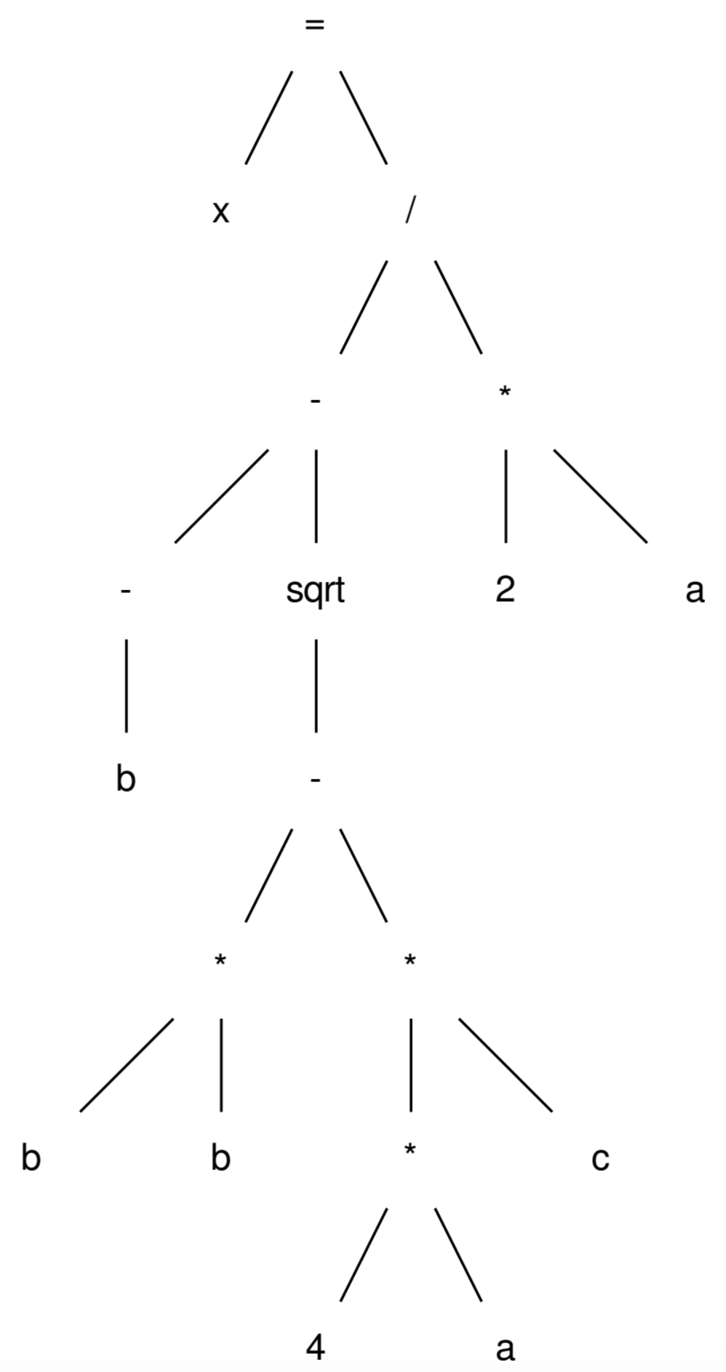
```
r0 = neg(b);
```



Convert this code to 3 address code

post-order traversal, creating virtual registers

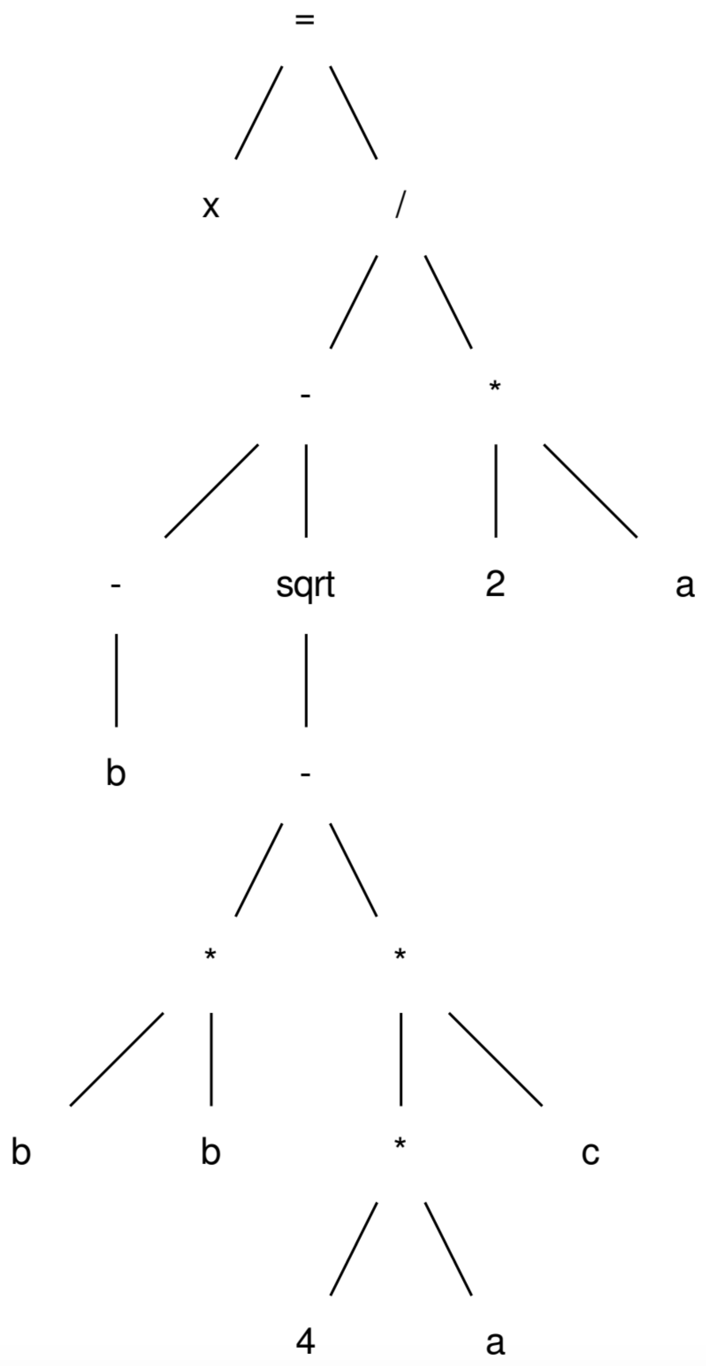
```
r0 = neg(b);
r1 = b * b;
```



Convert this code to 3 address code

post-order traversal, creating virtual registers

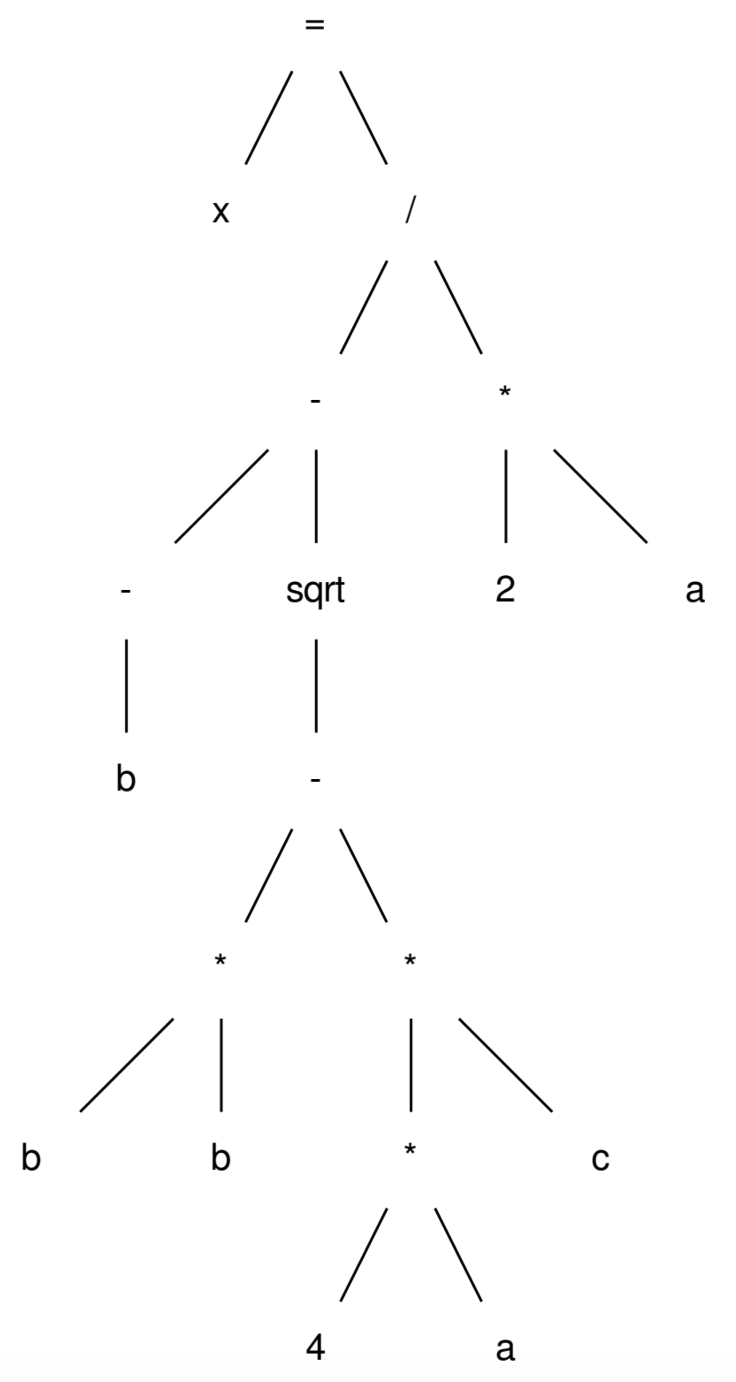
```
r0 = neg(b);
r1 = b * b;
r2 = 4 * a;
```



Convert this code to 3 address code

post-order traversal, creating virtual registers

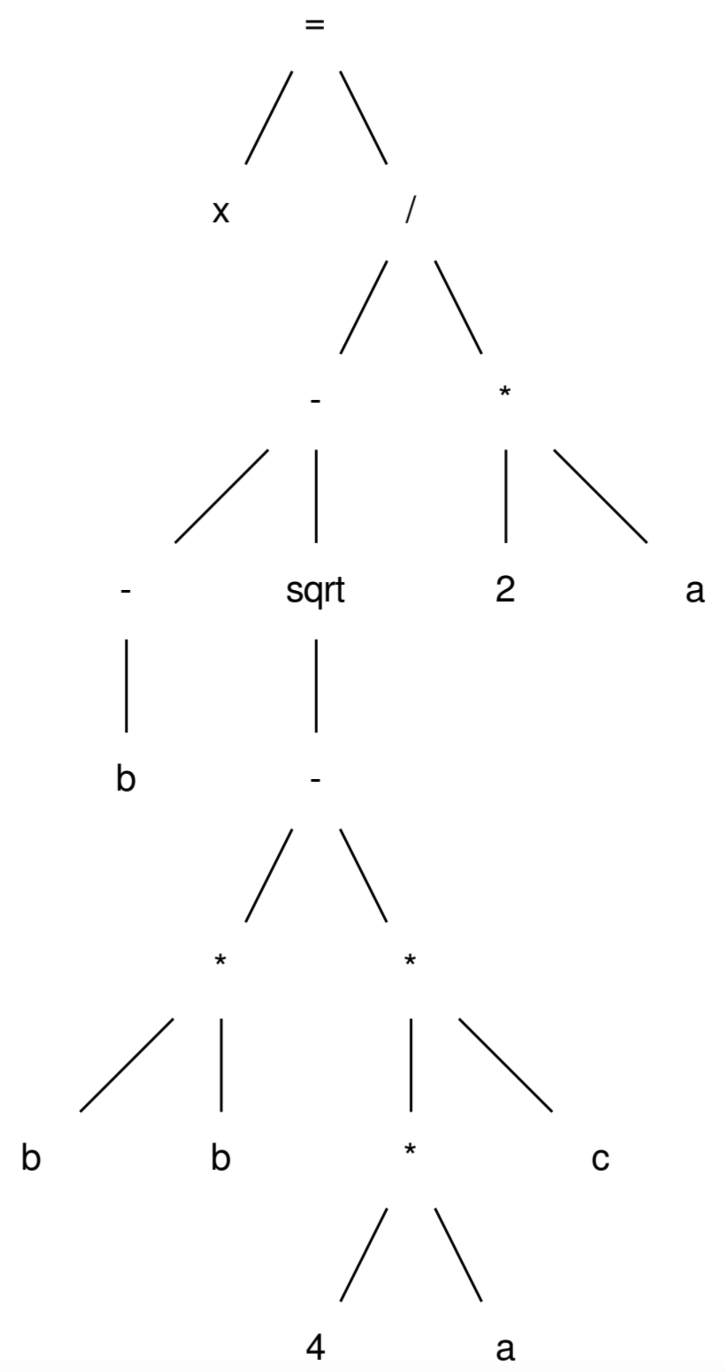
```
r0 = neg(b);
r1 = b * b;
r2 = 4 * a;
r3 = r2 * c;
```



Convert this code to 3 address code

post-order traversal, creating virtual registers

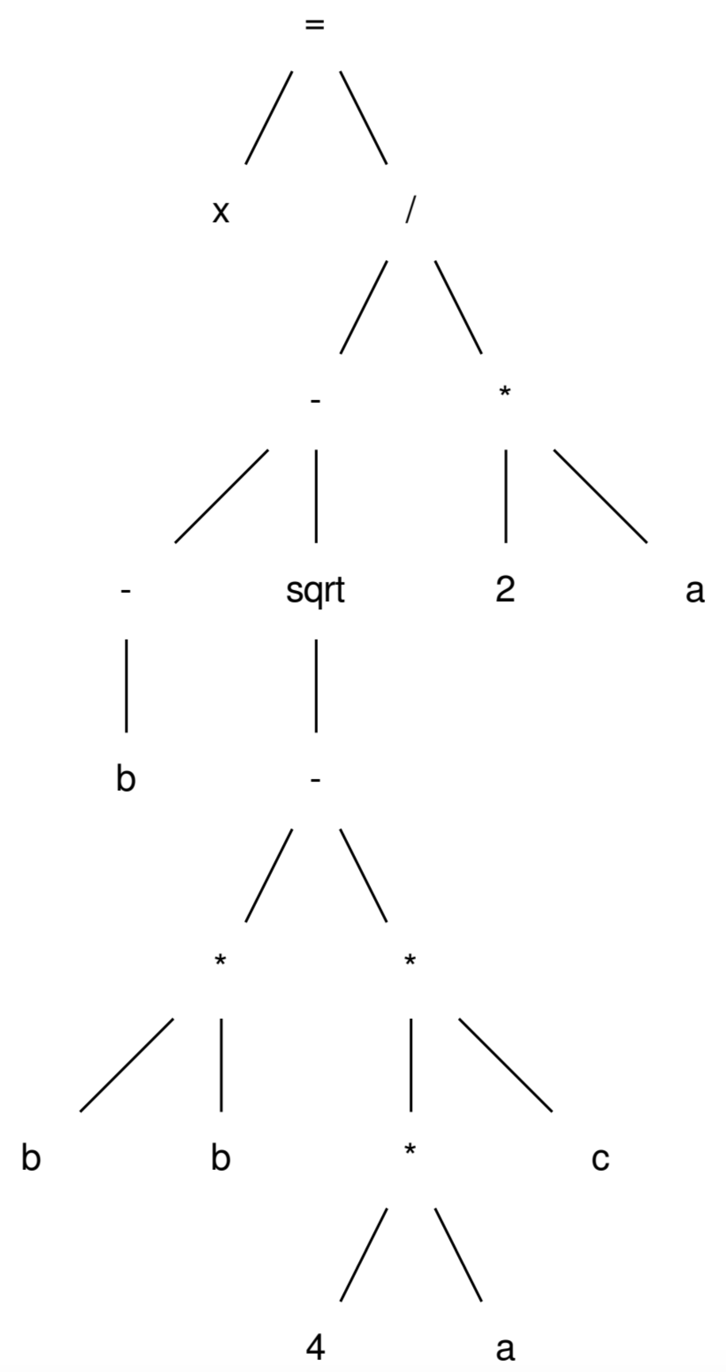
```
r0 = neg(b);
r1 = b * b;
r2 = 4 * a;
r3 = r2 * c;
r4 = r1 – r3;
```



Convert this code to 3 address code

post-order traversal, creating virtual registers

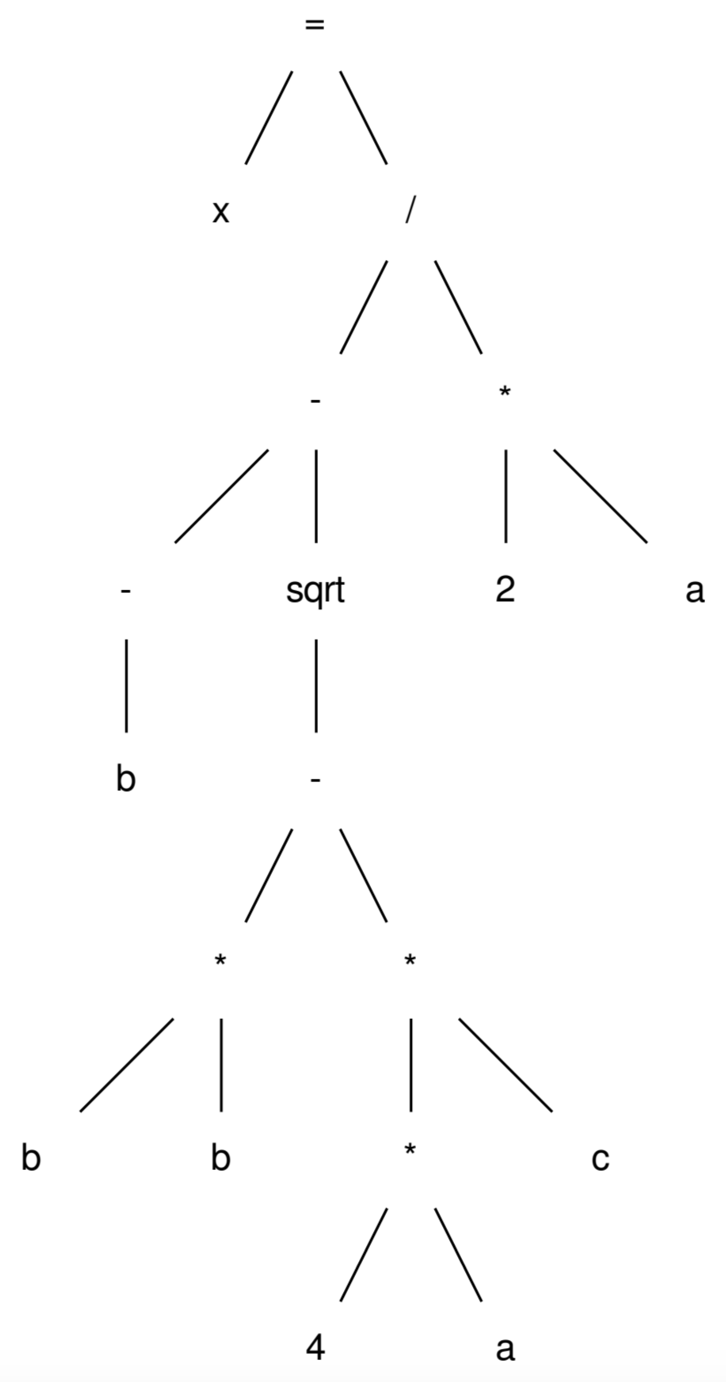
```
r0 = neg(b);
r1 = b * b;
r2 = 4 * a;
r3 = r2 * c;
r4 = r1 – r3;
r5 = sqrt(r4);
```



Convert this code to 3 address code

post-order traversal, creating virtual registers

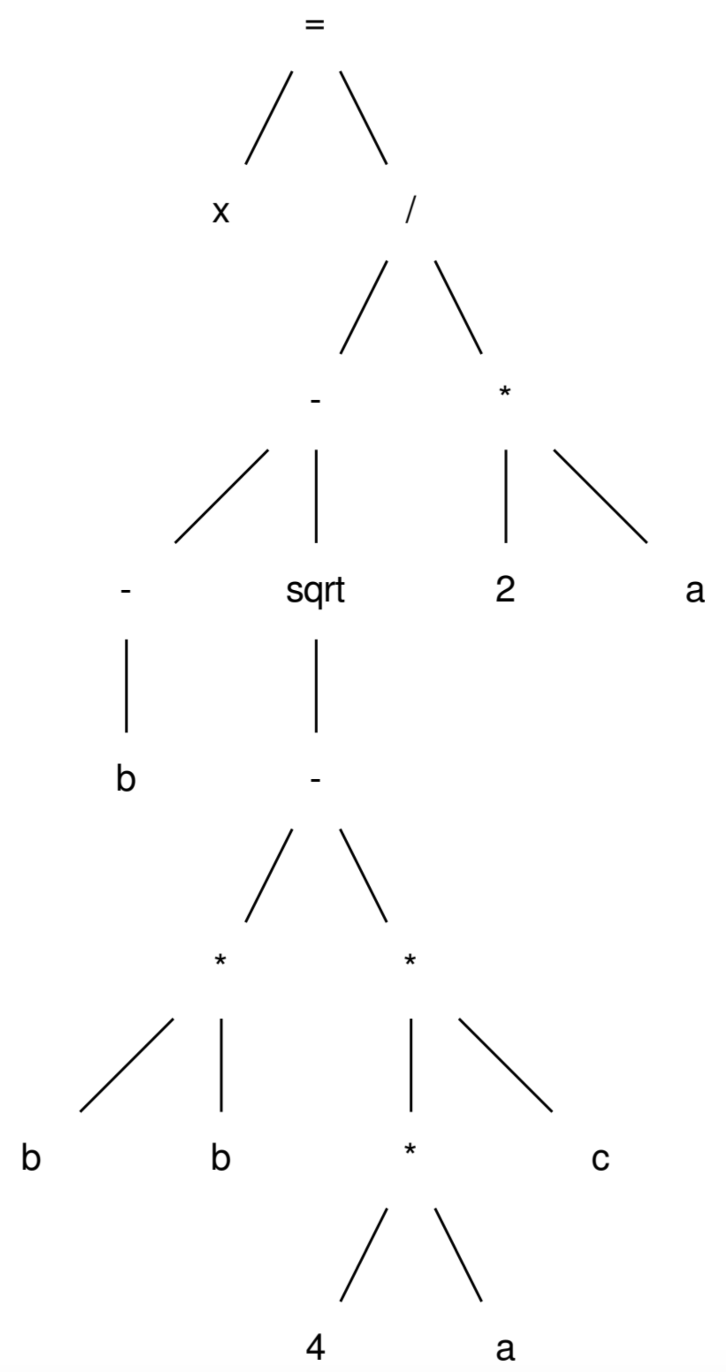
```
r0 = neg(b);
r1 = b * b;
r2 = 4 * a;
r3 = r2 * c;
r4 = r1 – r3;
r5 = sqrt(r4);
r6 = r0 – r5;
```



Convert this code to 3 address code

post-order traversal, creating virtual registers

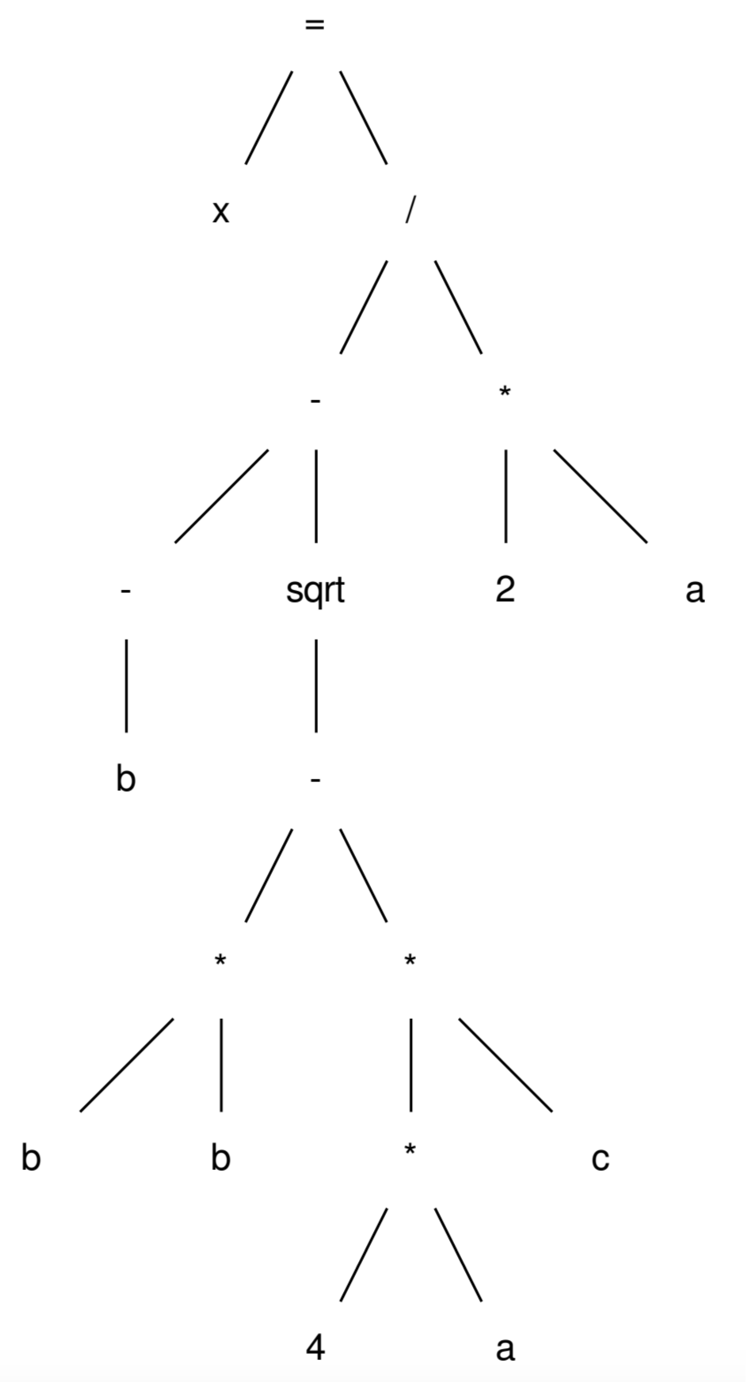
```
r0 = neg(b);
r1 = b * b;
r2 = 4 * a;
r3 = r2 * c;
r4 = r1 – r3;
r5 = sqrt(r4);
r6 = r0 – r5;
r7 = 2 * a;
```



Convert this code to 3 address code

post-order traversal, creating virtual registers

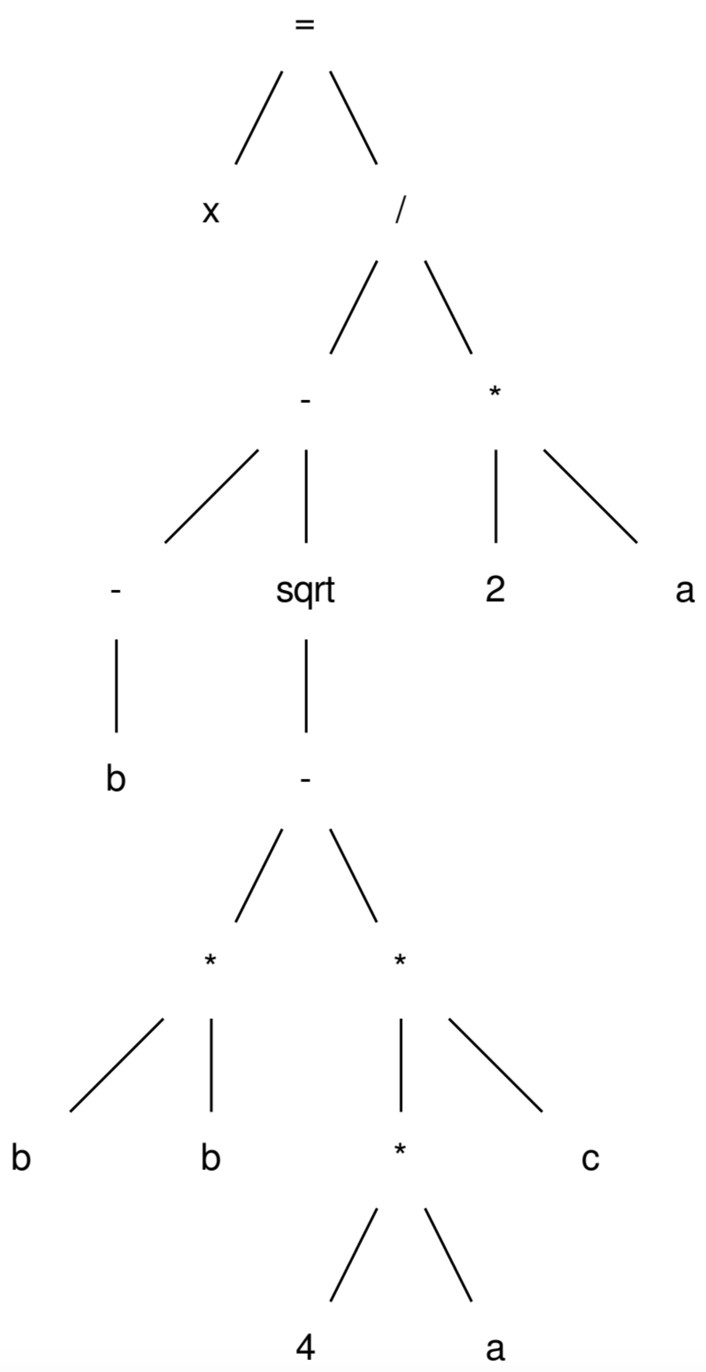
```
r0 = neg(b);
r1 = b * b;
r2 = 4 * a;
r3 = r2 * c;
r4 = r1 – r3;
r5 = sqrt(r4);
r6 = r0 – r5;
r7 = 2 * a;
r8 = r6 / r7;
```



Convert this code to 3 address code

post-order traversal, creating virtual registers

```
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;
```

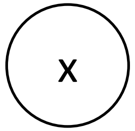


3 address code use-case

What now?

We can make a data-dependency graph (DDG)

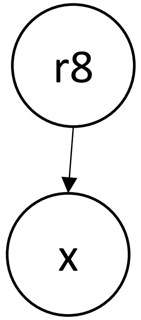
```
r0 = neg(b);
r1 = b * b;
r2 = 4 * a;
r3 = r2 * c;
r4 = r1 – r3;
r5 = sqrt(r4);
r6 = r0 – r5;
r7 = 2 * a;
r8 = r6 / r7;
x  = r8;
```



What now?

We can make a data-dependency graph (DDG)

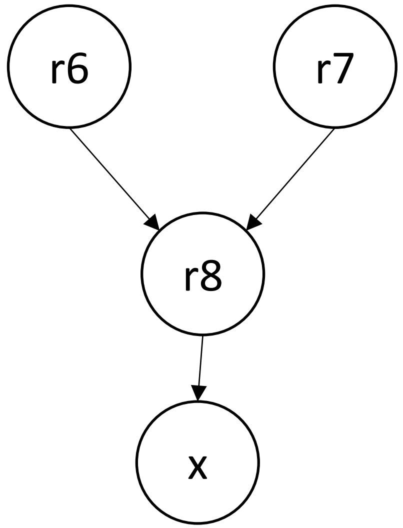
```
r0 = neg(b);
r1 = b * b;
r2 = 4 * a;
r3 = r2 * c;
r4 = r1 – r3;
r5 = sqrt(r4);
r6 = r0 – r5;
r7 = 2 * a;
r8 = r6 / r7;
x  = r8;
```



What now?

We can make a data-dependency graph (DDG)

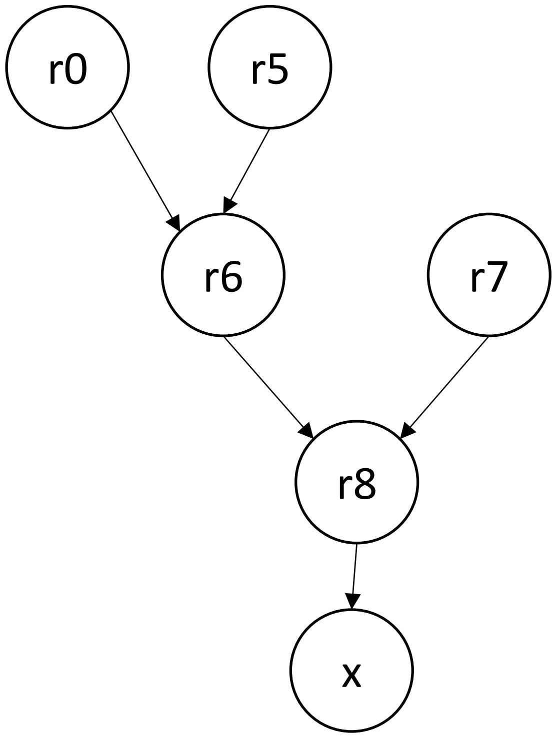
```
r0 = neg(b);
r1 = b * b;
r2 = 4 * a;
r3 = r2 * c;
r4 = r1 – r3;
r5 = sqrt(r4);
r6 = r0 – r5;
r7 = 2 * a;
r8 = r6 / r7;
x  = r8;
```



What now?

We can make a data-dependency graph (DDG)

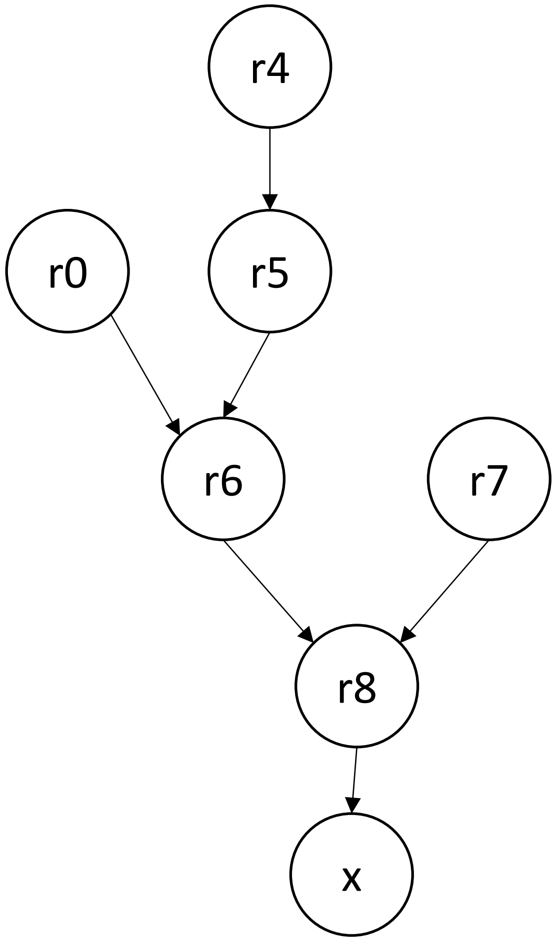
```
r0 = neg(b);
r1 = b * b;
r2 = 4 * a;
r3 = r2 * c;
r4 = r1 – r3;
r5 = sqrt(r4);
r6 = r0 – r5;
r7 = 2 * a;
r8 = r6 / r7;
x  = r8;
```



What now?

We can make a data-dependency graph (DDG)

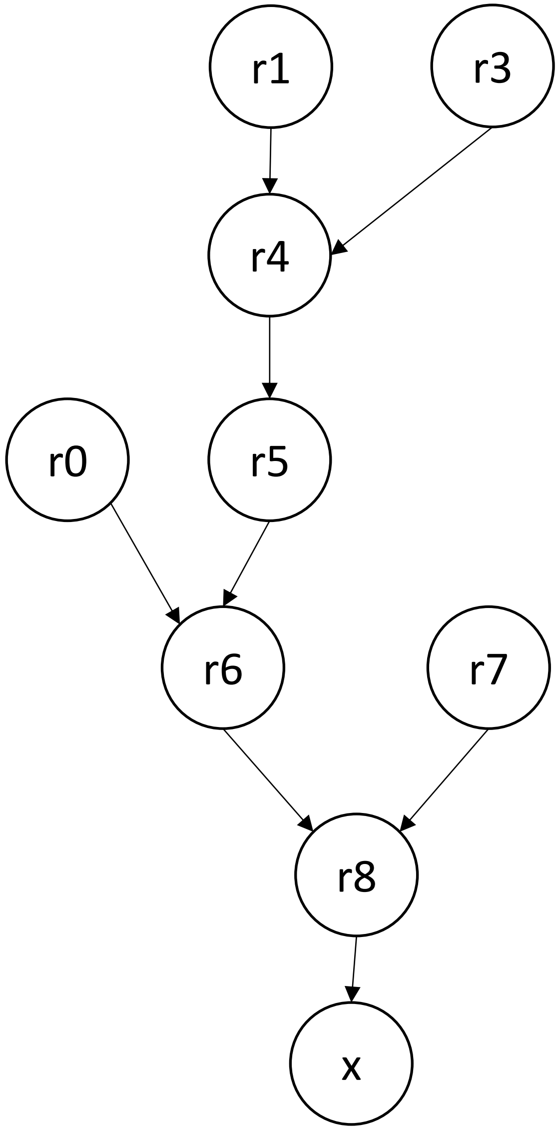
```
r0 = neg(b);
r1 = b * b;
r2 = 4 * a;
r3 = r2 * c;
r4 = r1 – r3;
r5 = sqrt(r4);
r6 = r0 – r5;
r7 = 2 * a;
r8 = r6 / r7;
x  = r8;
```



What now?

We can make a data-dependency graph (DDG)

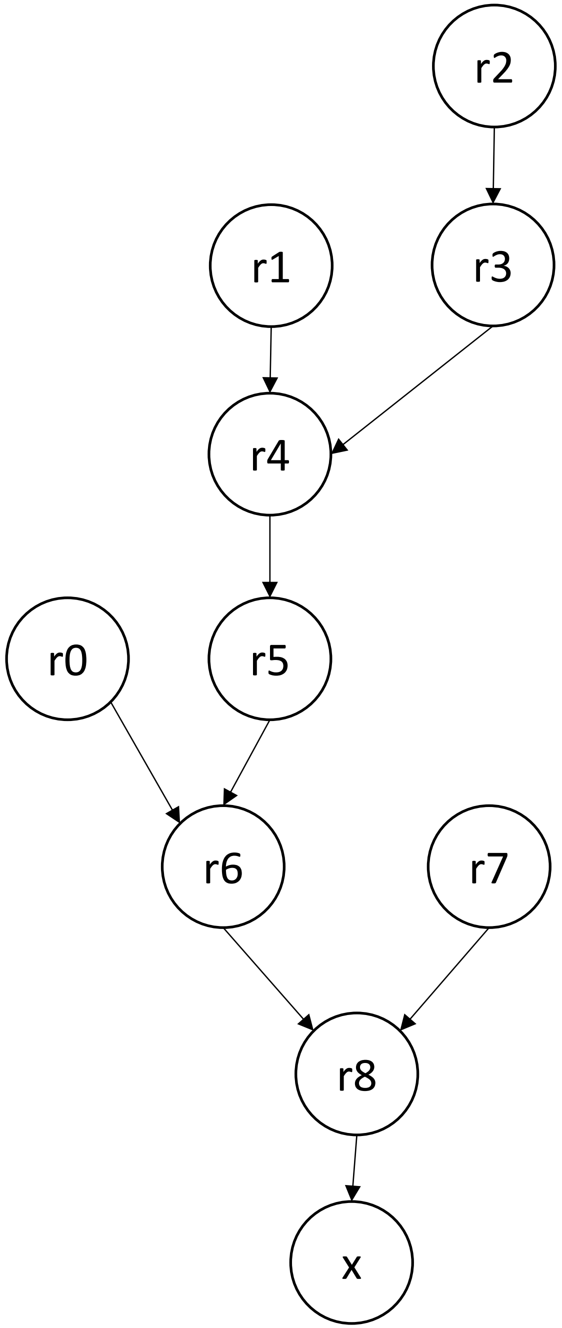
```
r0 = neg(b);
r1 = b * b;
r2 = 4 * a;
r3 = r2 * c;
r4 = r1 – r3;
r5 = sqrt(r4);
r6 = r0 – r5;
r7 = 2 * a;
r8 = r6 / r7;
x  = r8;
```



What now?

We can make 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;
```

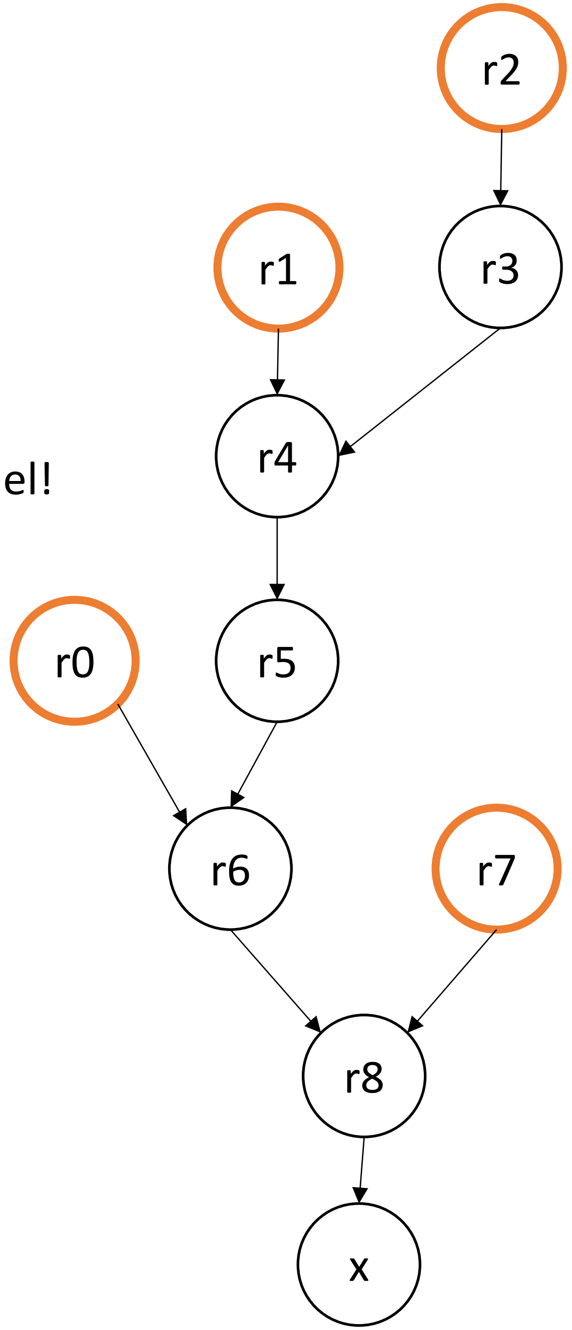


What now?

We can make 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!



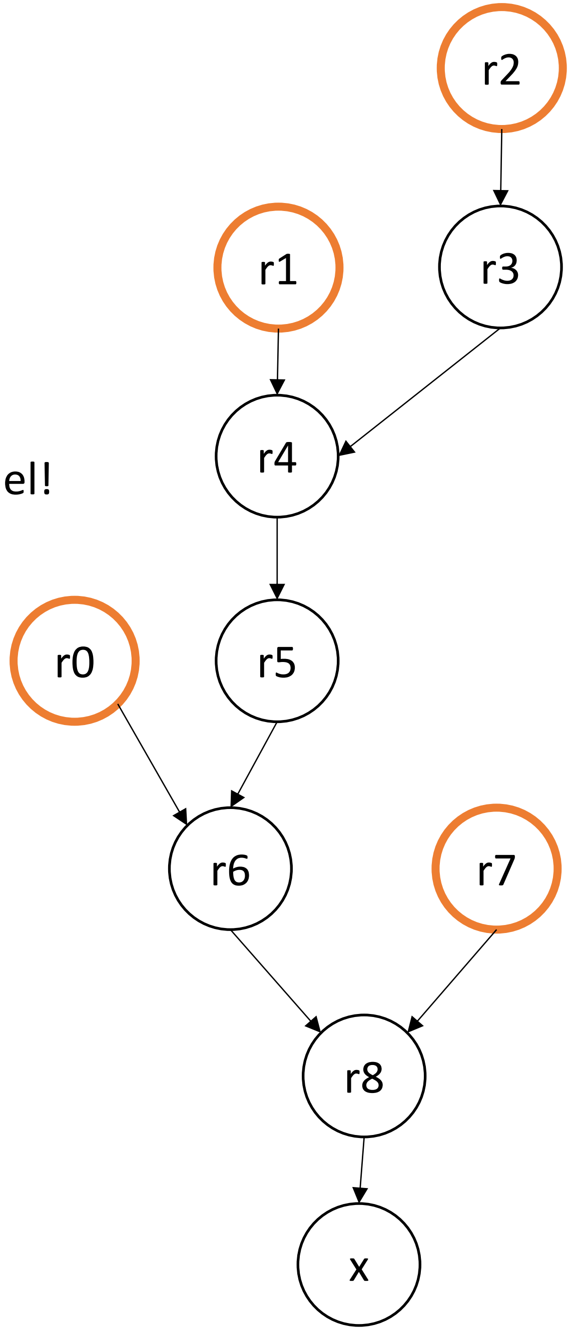
What now?

We can make 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!

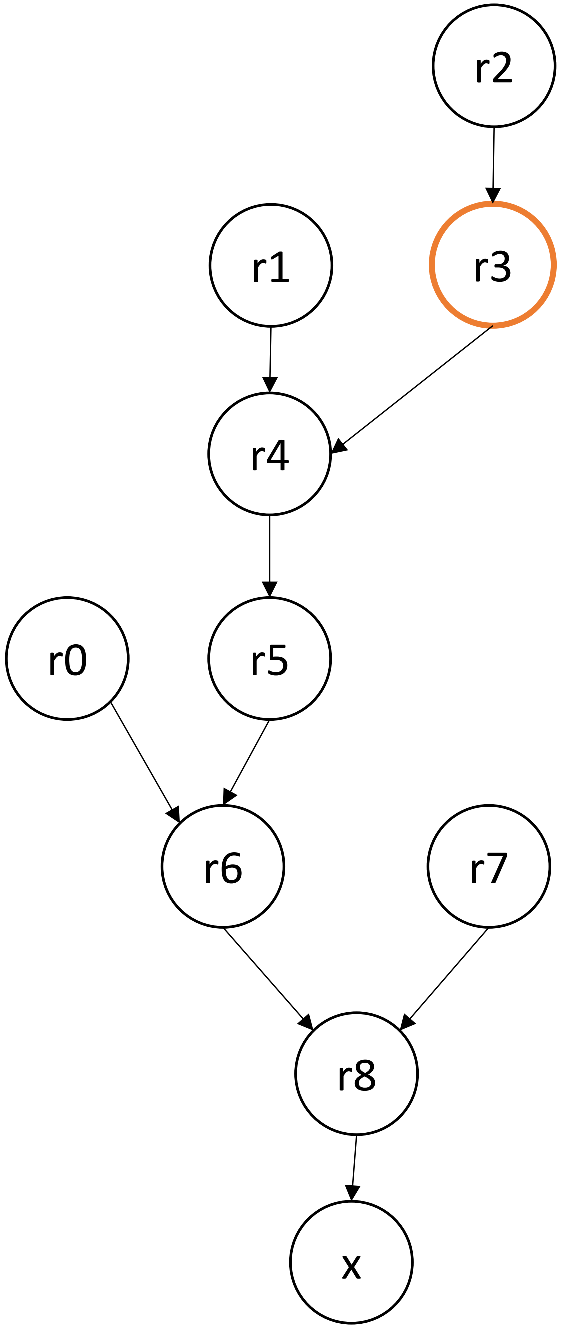


What now?

We can make 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?



back to 3 address code

```
x = expr0;
y = expr1;
z = expr2;
```

- Convert each expression to an AST.
- Convert each AST to 3 address code.
- Sequence each expression.

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>
```

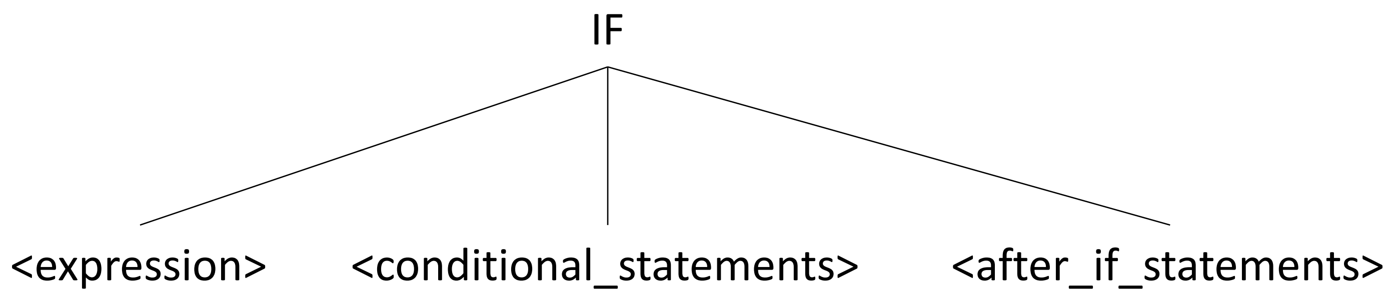
unconditional branch

What about control flow?

r0 = <expression>;
br r0, inside_if, after_if;

inside_if:
<conditional_statements>;

after_if:
<after_if_statements>;



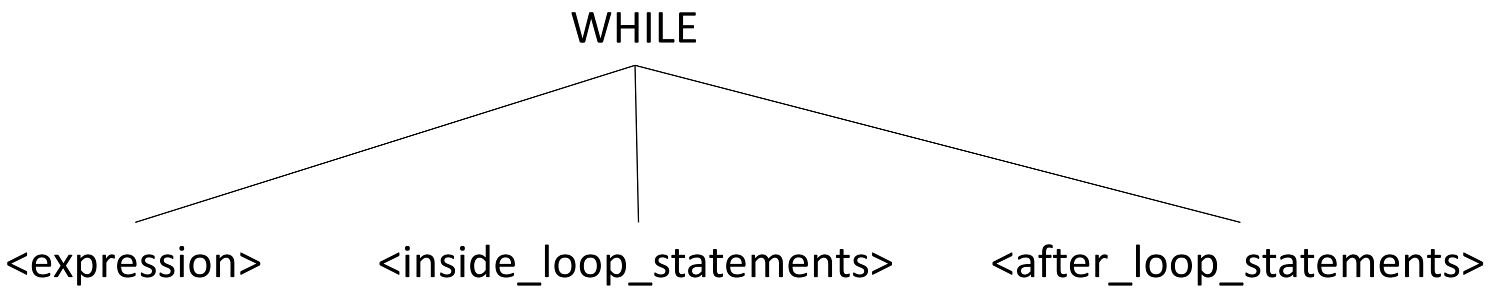
What about control flow?

beginning_label:
r0 = <expression>

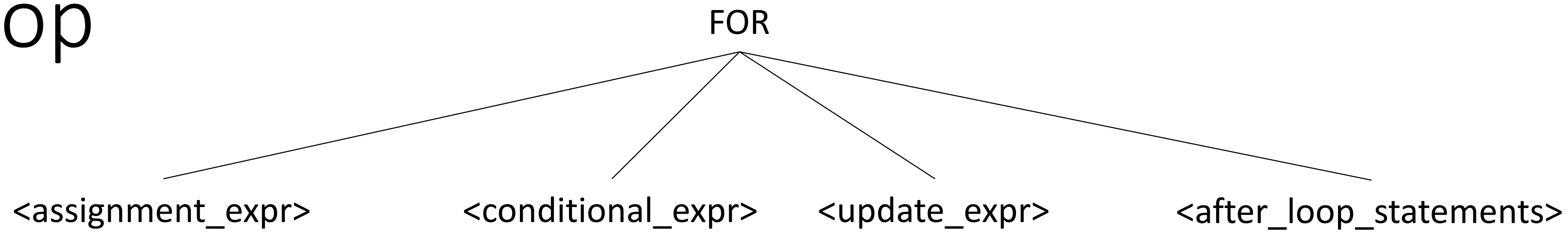
br r0, inside_loop, after_loop;

inside_loop:
<inside_loop_statements>
br beginning_label;

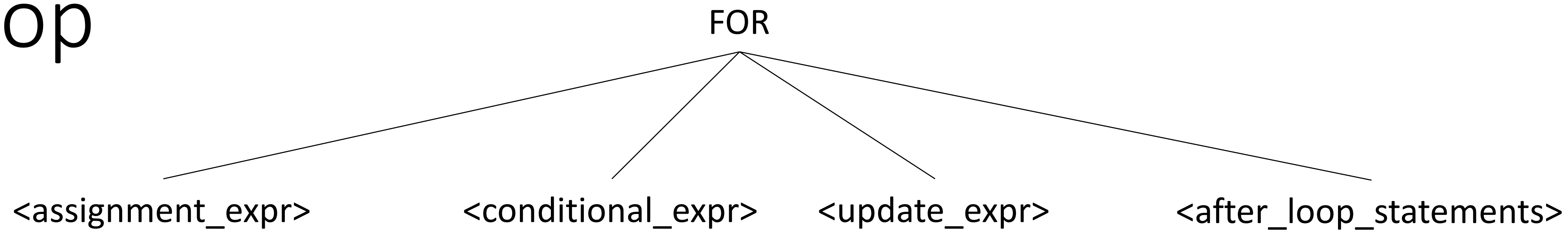
after_loop:
<after_loop_statements>



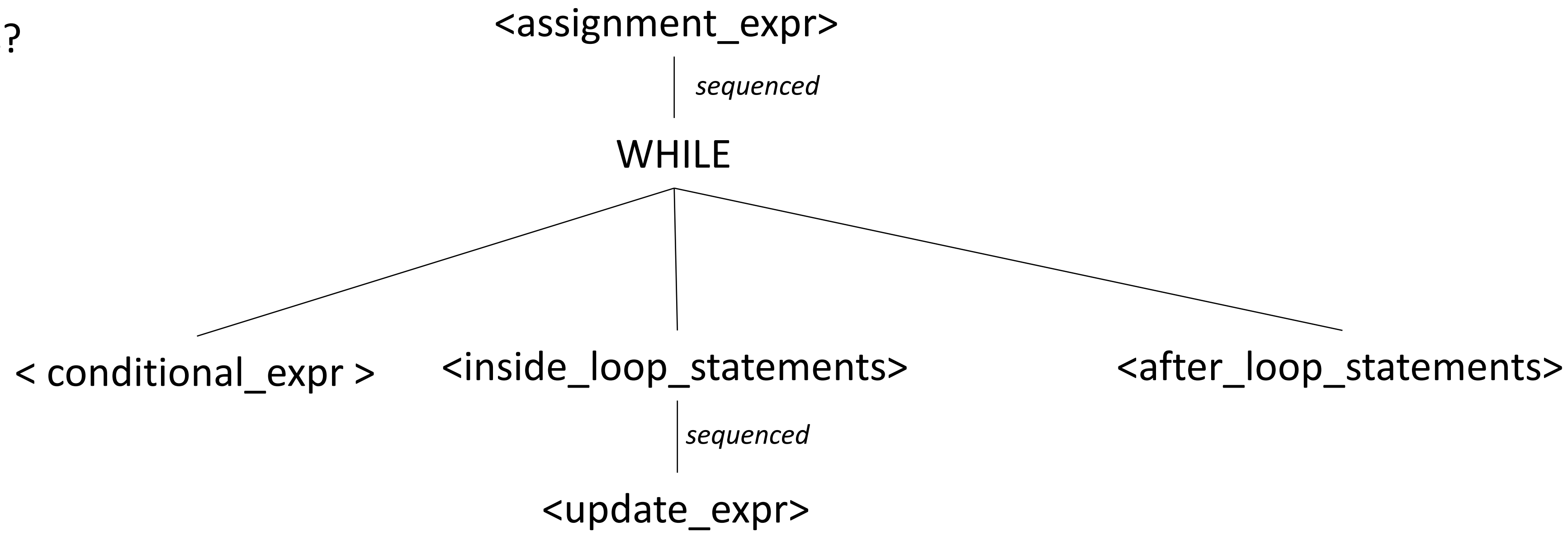
For loop


FOR
<assignment_expr>
<conditional_expr>
<update_expr>
<after_loop_statements>

For loop



Pros and 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 preceeding instructions will execute

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 preceeding 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 preceeding 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

How might they appear in a high-level language?

- 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 preceeding 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 preceeding instructions will execute

How might they appear in a high-level language?

Four Basic Blocks

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

Single Basic Block

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

Two Basic Blocks

```
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

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

optimized to

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

- 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?

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

optimized to →

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

```
Label_0:
x = a + b;

Label_1:
y = a + b;
```

CANNOT always optimized to →

```
Label_0:
x = a + b;

Label_1:
y = x;
```

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?

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

optimized to →

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

```
Label_0:
x = a + b;

Label_1:
y = a + b;
```

CANNOT always optimized to →

```
Label_0:
x = a + b;

Label_1:
y = x;
```

code could skip Label_0, leaving x undefined!

```
br Label_1;

Label_0:
x = a + b;

Label_1:
y = a + b;
```

Optimization levels

```
…
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;

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

optimized to →

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

```
Label_0:
x = a + b;

Label_1:
y = a + b;
```

CANNOT always optimized to →

```
Label_0:
x = a + b;

Label_1:
y = x;
```

code could skil Label_0

```
br Label_1;

Label_0:
x = a + b;

Label_1:
y = a + b;
```

Moving on to a concrete optimization algorithm

Local Value Numbering

- Local optimization
  - can be extended with the help of flow analysis
- Aims to remove redundant arithmetic instructions

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

Local Value Numbering

- Local optimization
  - can be extended with the help of flow analysis

- Aims to remove redundant arithmetic instructions

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

valid? →

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

Local Value Numbering

- Local optimization
  - can be extended with the help of flow analysis
- Aims to remove redundant arithmetic instructions

```
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

- Local optimization
  - can be extended with the help of flow analysis

- Aims to remove redundant arithmetic instructions

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

valid? →

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

Yes!

Local Value Numbering

- Algorithm:

- Provide a number to each variable. Update the number each time the variable is updated.

- Several different implementations. I 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.

- Several different implementations. I keep a global counter; increment with new variables or assignments

```
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;
```

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!

Next week

- Local value numbering continued:
  - commutative operations
  - register usage

- Introduction to flow analysis:
  - How to create extended basic blocks for local analysis