CSE211: Compiler Design

Oct. 11, 2021

- **Topic**: Introduction to Module 2: optimizations!

- **Questions**:
  - *What sort of compiler optimizations do you know about?*
  - *What sort of intermediate representations do you know about?*

Announcements

- Homework 1 is out
  - Due on the 18th
  - One week!
- One more office hour:
  - Signup sheet: released sometime between 12 - 1 PM on Thursday
  - 10 minute slot
  - Remote or in-person
  - If you want a slot, but are unable to get one, please message me!

CSE211: Compiler Design

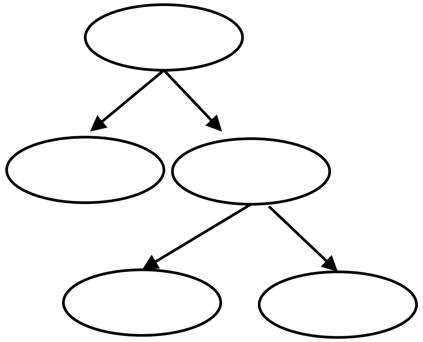
Oct. 11, 2021

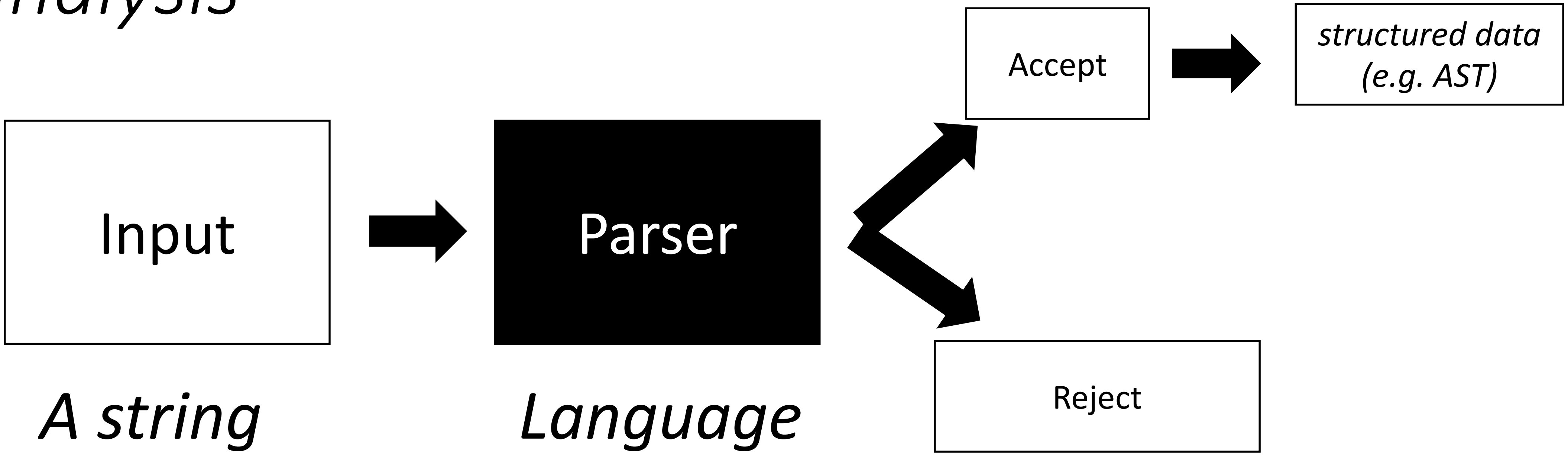
- **Topic**: Introduction to Module 2: optimizations!

- **Questions**:
  - *What sort of compiler optimizations do you know about?*
  - *What sort of intermediate representations do you know about?*

On to Module 2!
Optimizations and flow analysis

continue to the rest of compilation

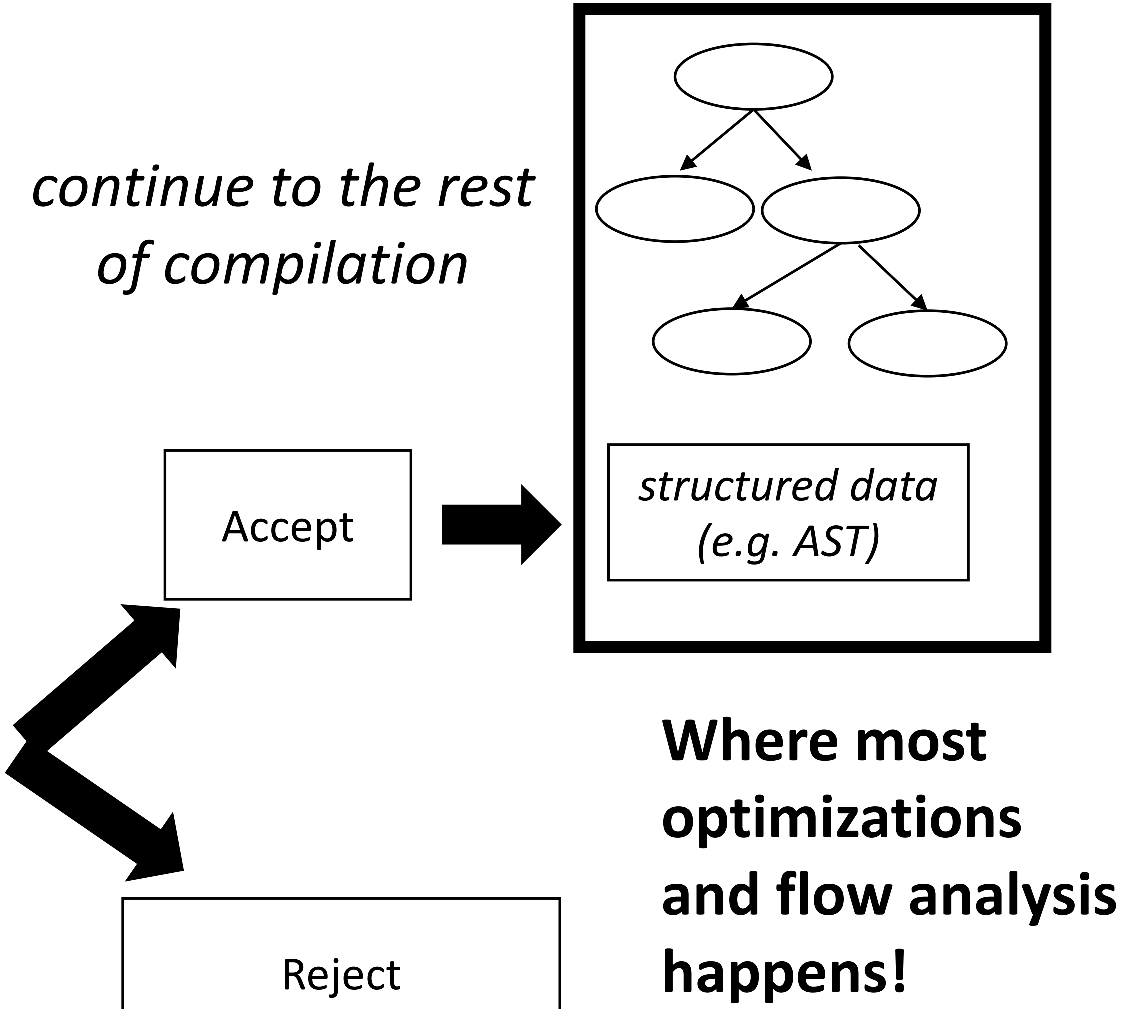


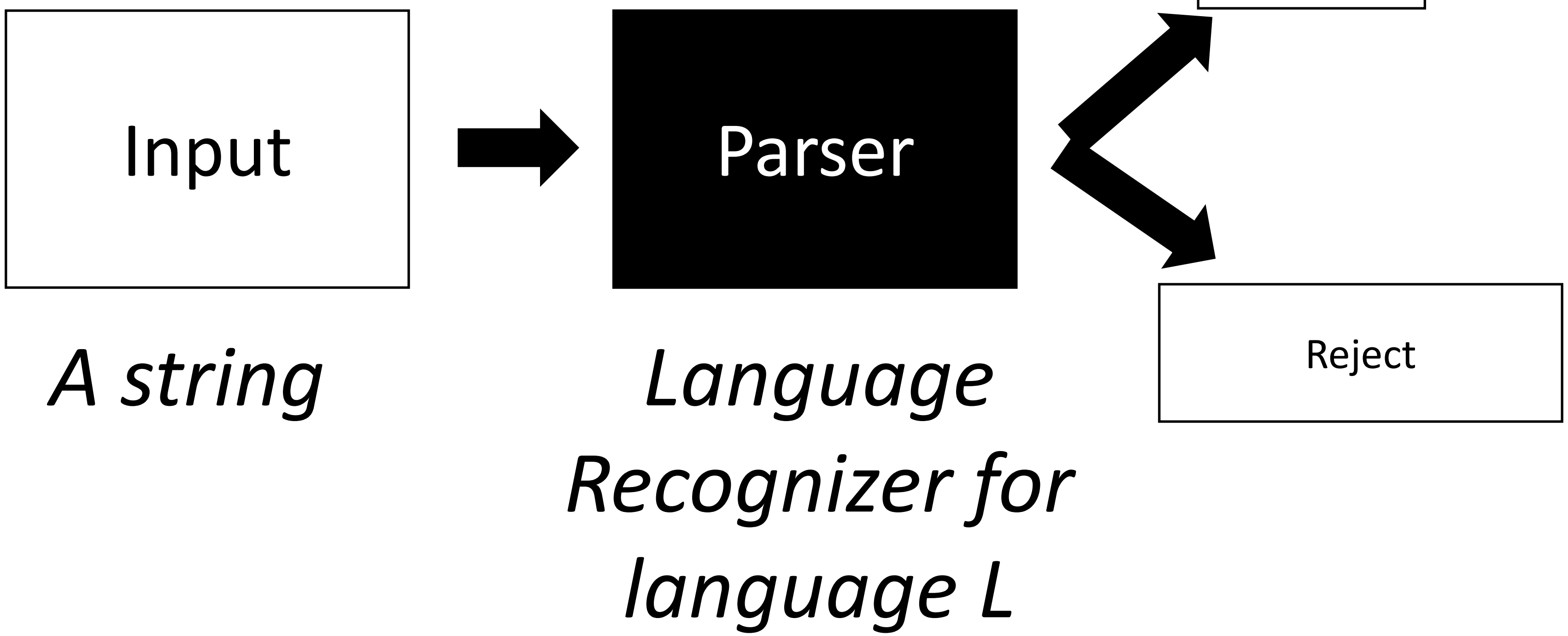


A string

Language Recognizer for language L

On to Module 2!
Optimizations and flow analysis





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

```
x = y + z;        x = y + z;
w = y + z;   →    w = x;
```

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?

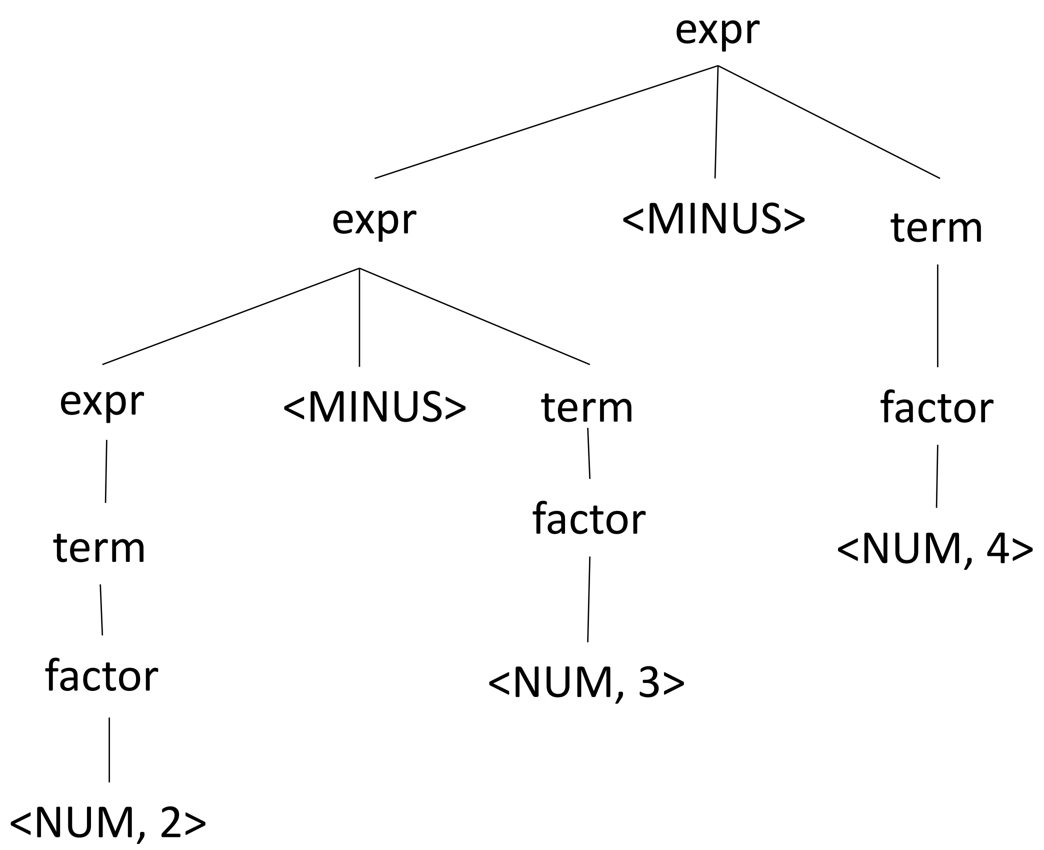
GPUs often have many IRs... why?

Abstract Syntax Trees

- Remember the expression parse tree

`input: 2-3-4`

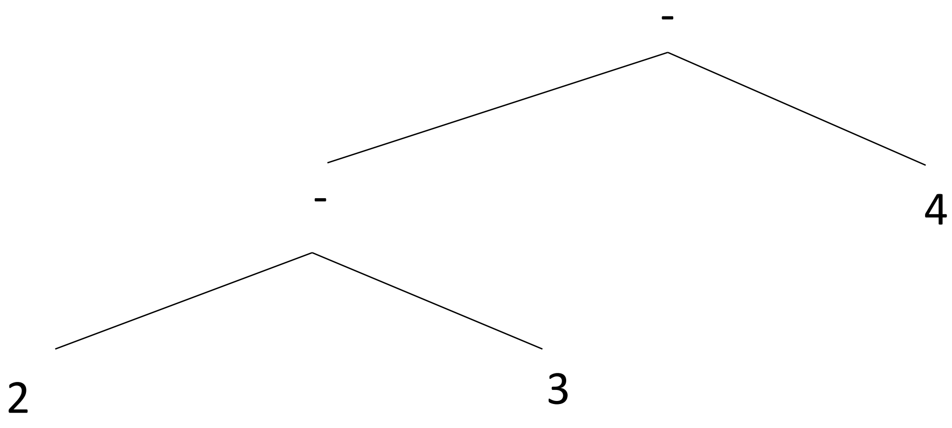
Operator	Name	Productions
+,-	expr	: expr PLUS term | expr MINUS term | term
*,/	term	: term TIMES pow | term DIV pow | Pow
^	pow	: factor CARROT pow | factor
()	factor	: LPAR expr RPAR | NUM



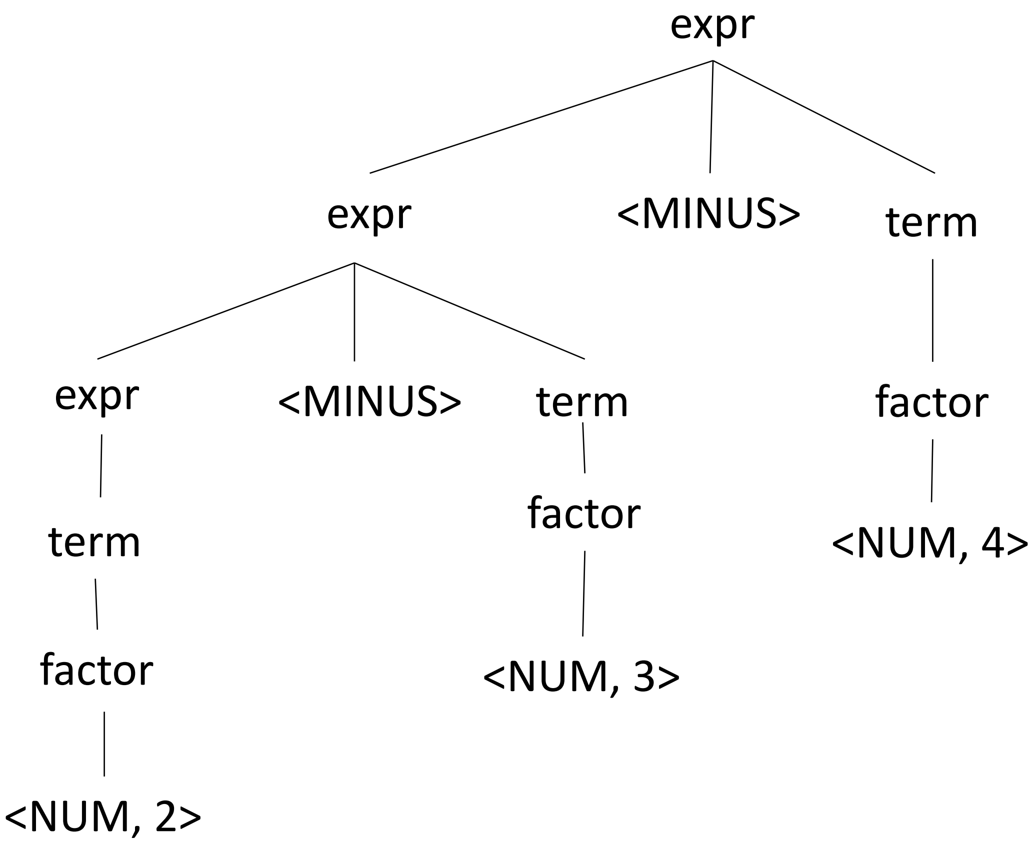
Abstract Syntax Trees

- Convert into an AST

`input: 2-3-4`



Much more compact!

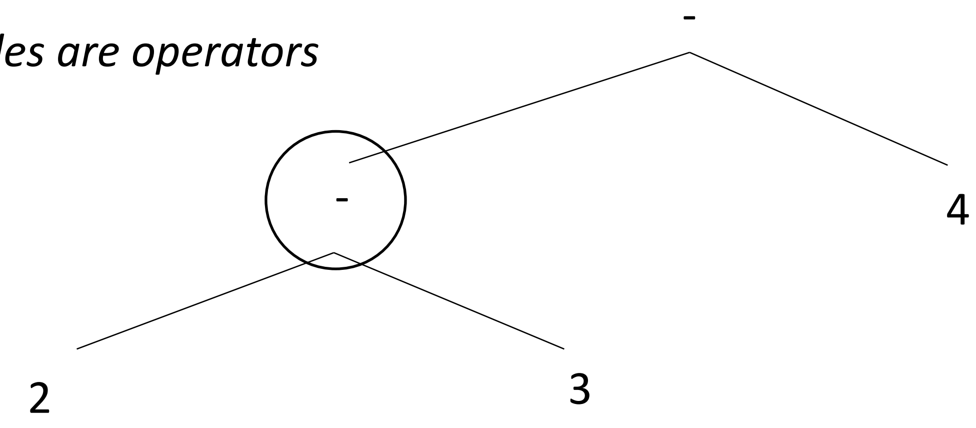


Abstract Syntax Trees

- Convert into an AST

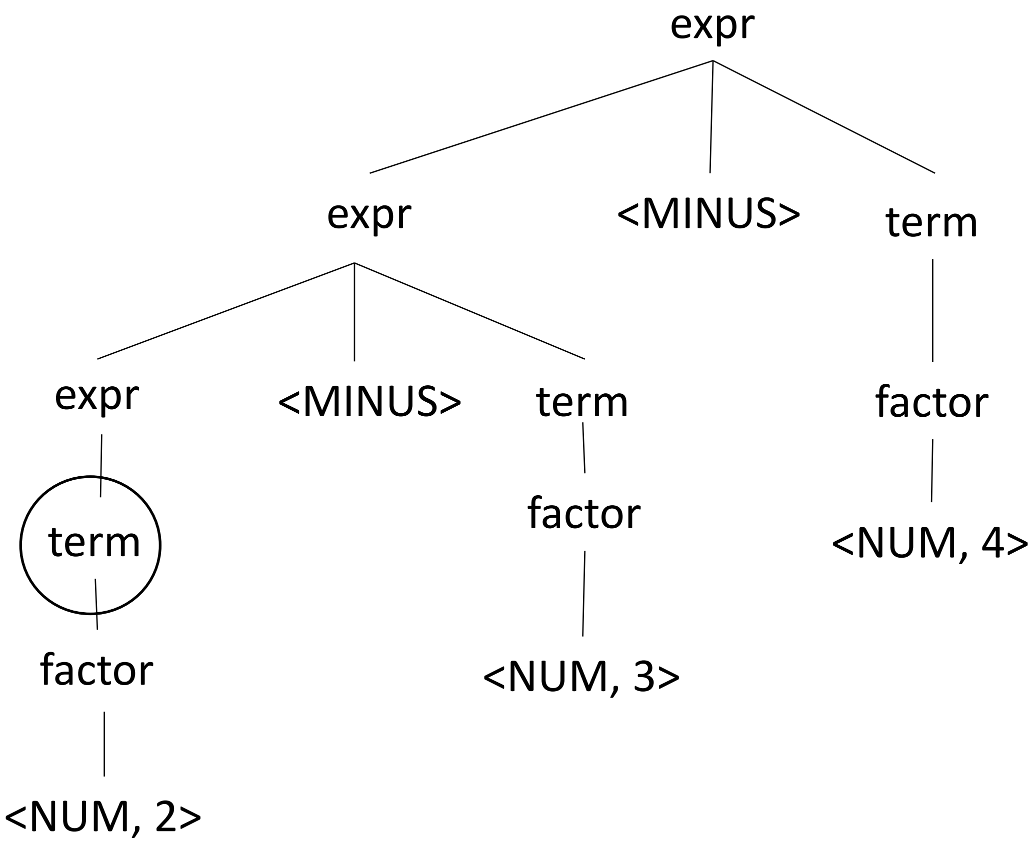
`input: 2-3-4`

nodes are operators



Much more compact!

nodes are production rules



Abstract Syntax Trees

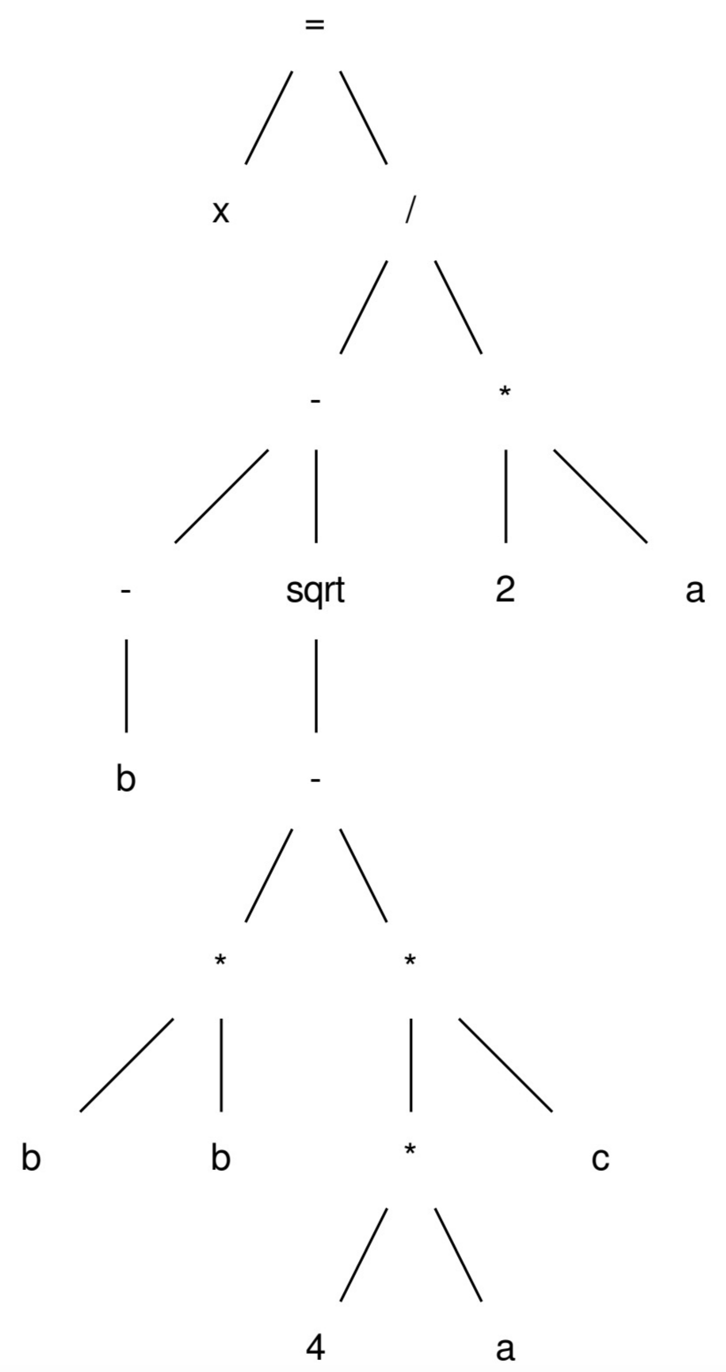
- Easier to see bigger trees, e.g. quadratic formula:

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

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

Thanks to Sreepathi Pai for the example!

x = (-b - 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 ← 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 = 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

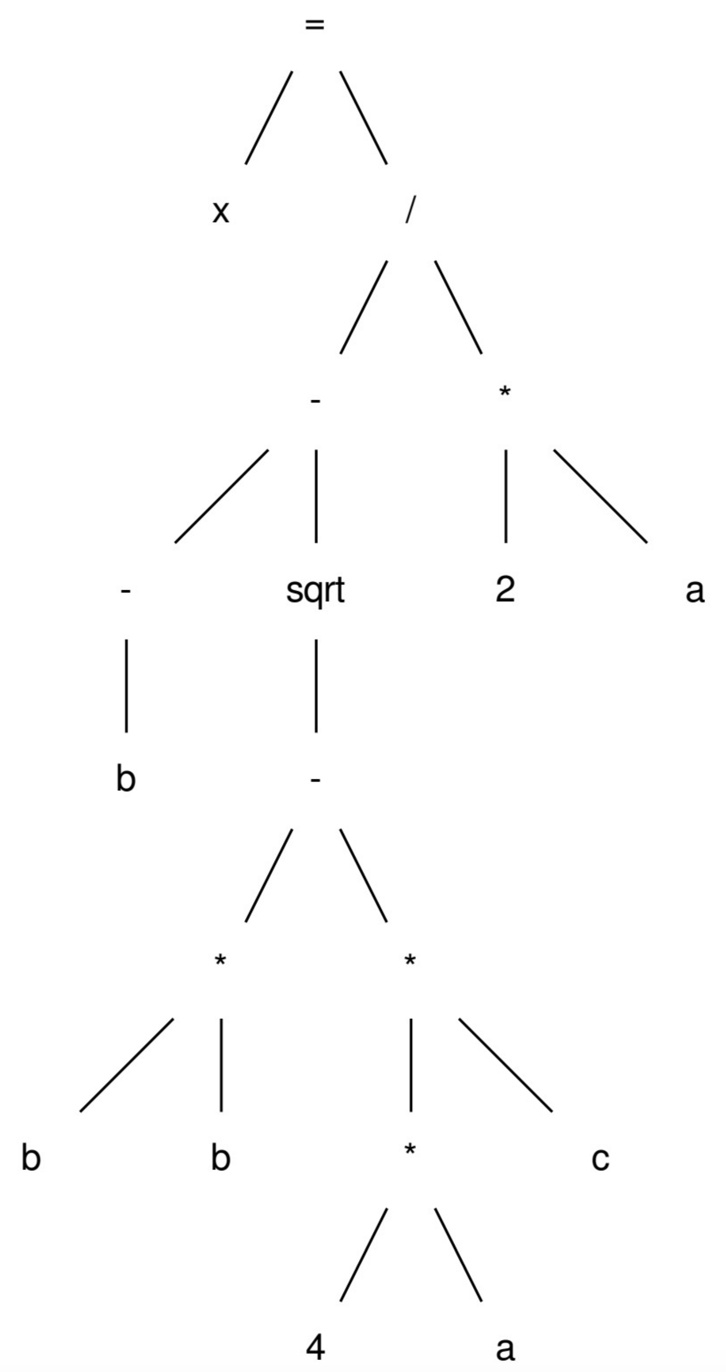
- Each instruction consists of 3 “addresses”
    - Address here means a virtual register or value
    - unlimited virtual registers

- 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

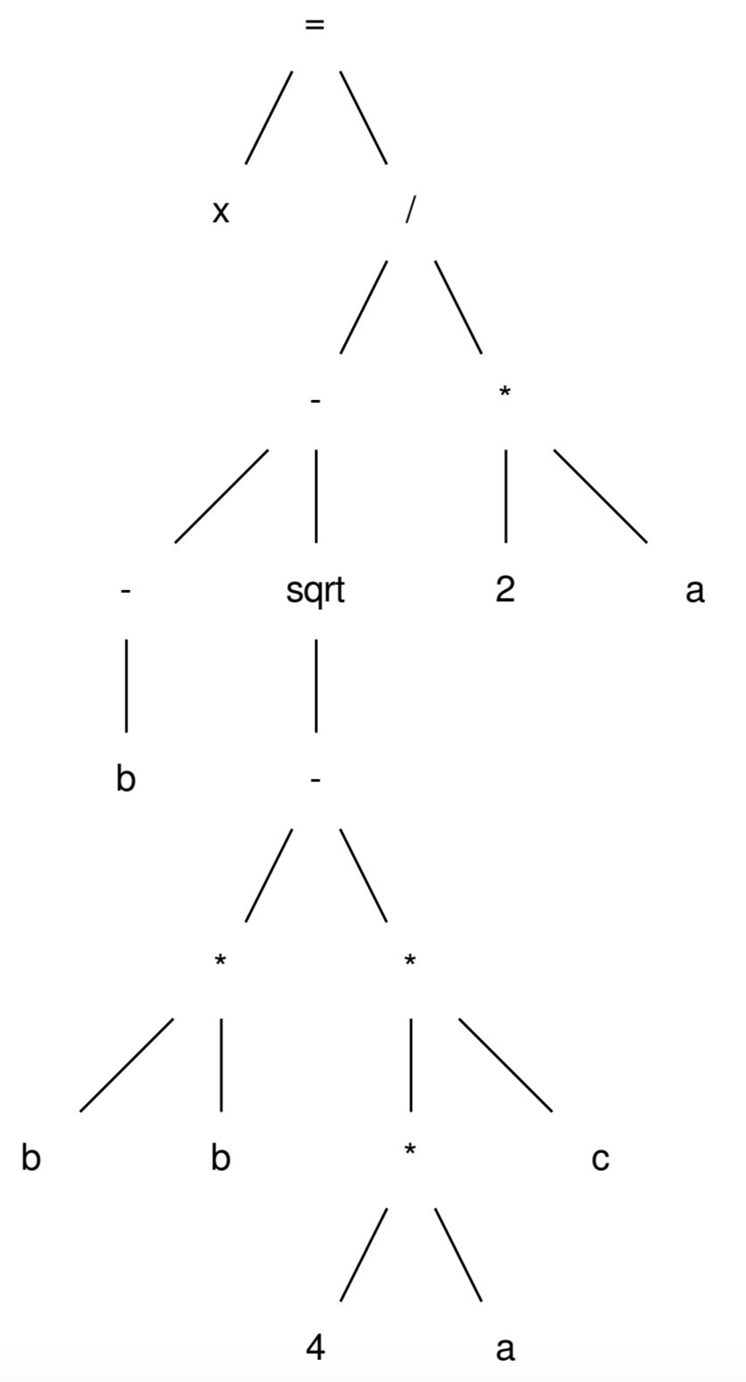
post-order traversal, creating virtual registers for each node



Convert this code to 3 address code

post-order traversal, creating virtual registers for each node

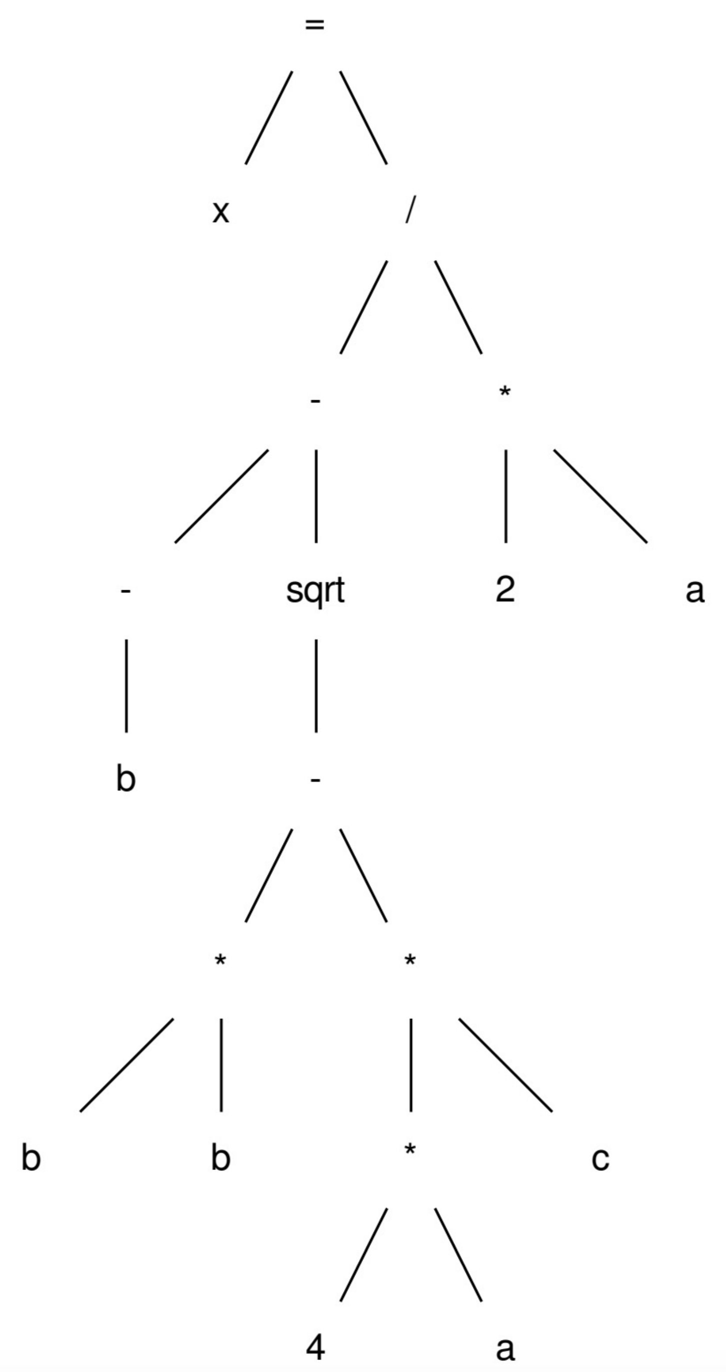
```
r0 = neg(b);
```



Convert this code to 3 address code

post-order traversal, creating virtual registers for each node

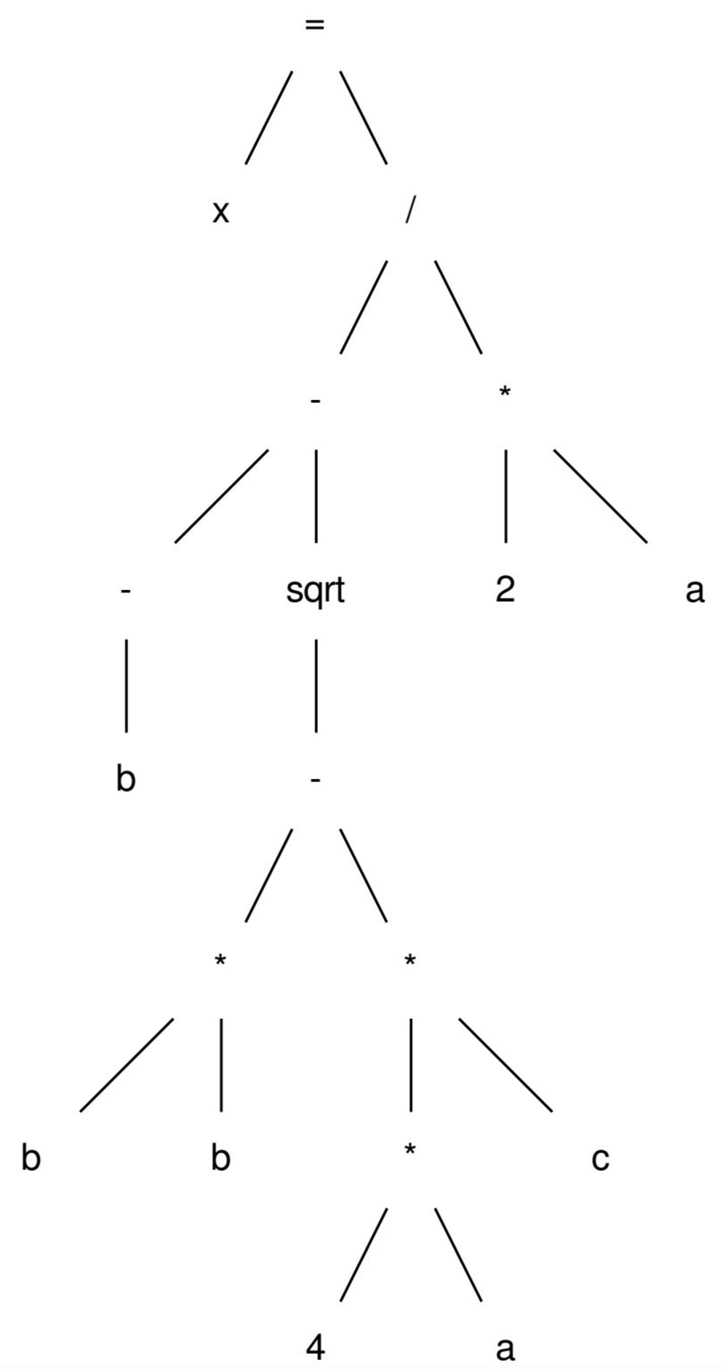
```
r0 = neg(b);
r1 = b * b;
```



Convert this code to 3 address code

post-order traversal, creating virtual registers for each node

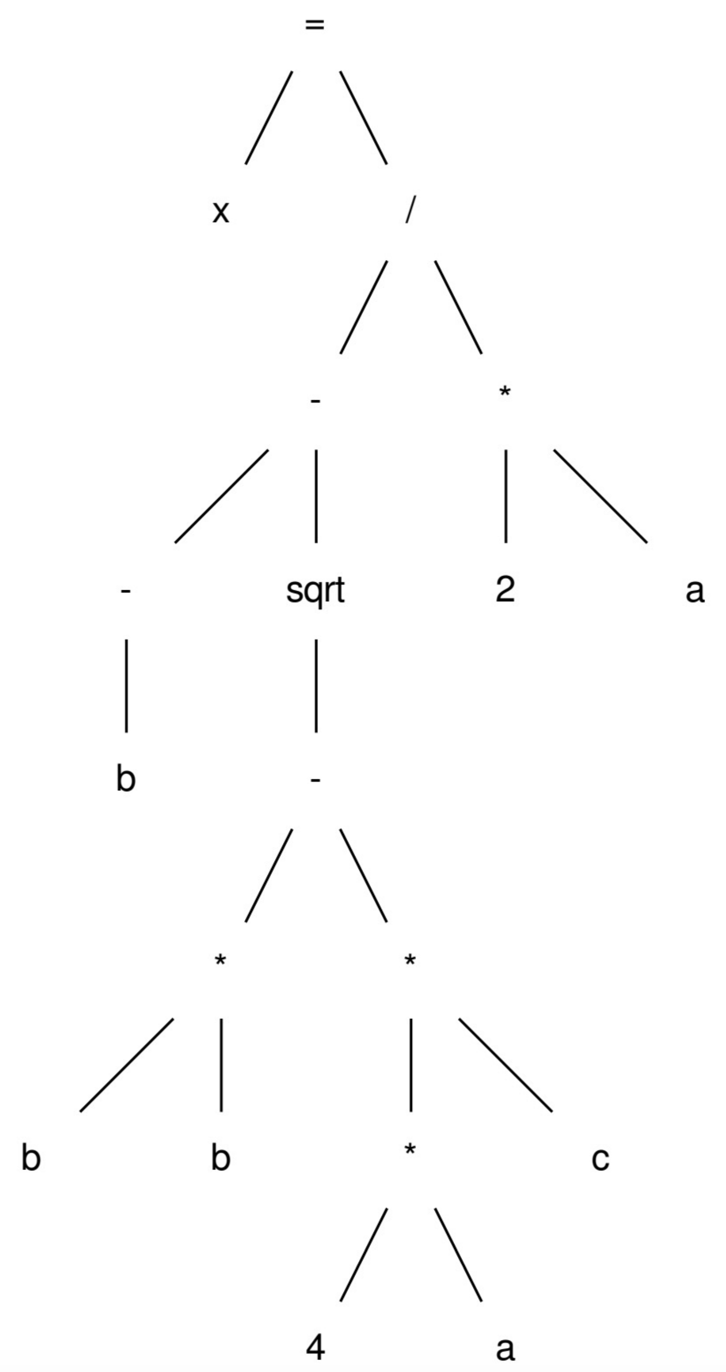
```
r0 = neg(b);
r1 = b * b;
r2 = 4 * a;
```



Convert this code to 3 address code

post-order traversal, creating virtual registers for each node

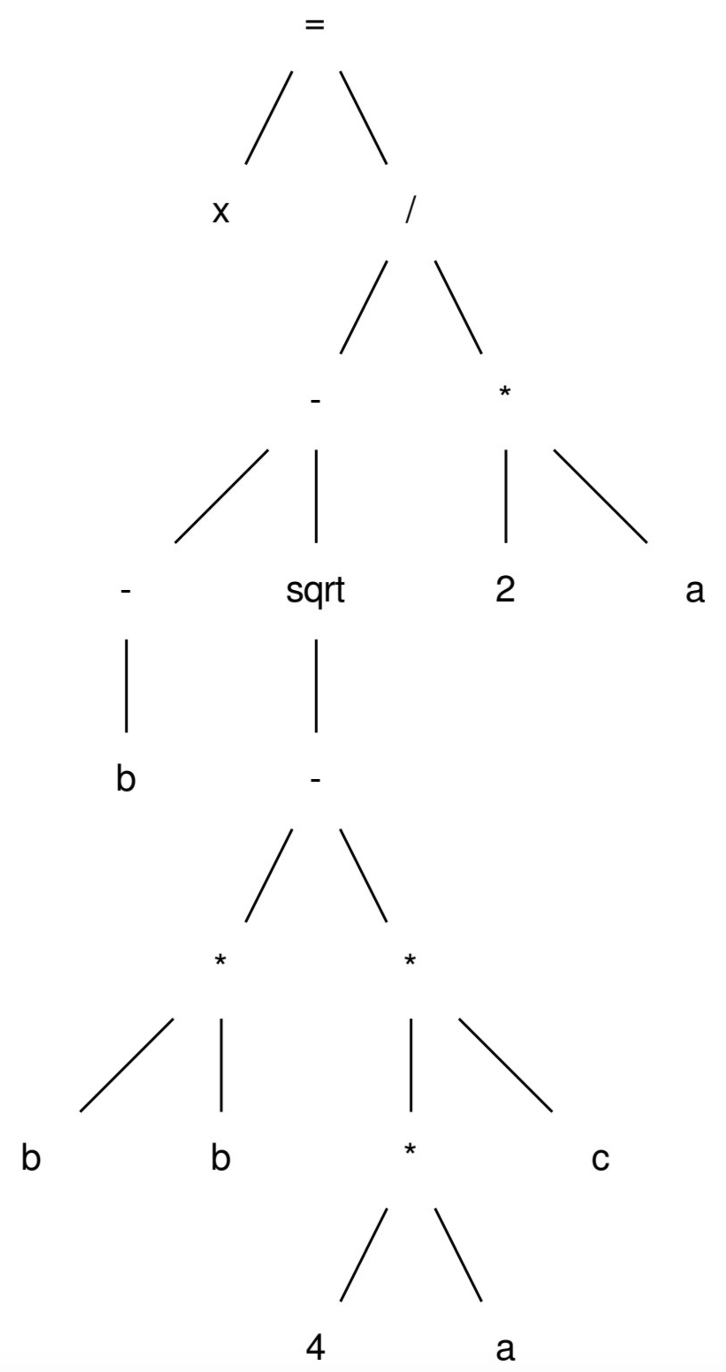
```
r0 = neg(b);
r1 = b * b;
r2 = 4 * a;
r3 = r2 * c;
```



Convert this code to 3 address code

post-order traversal, creating virtual registers for each node

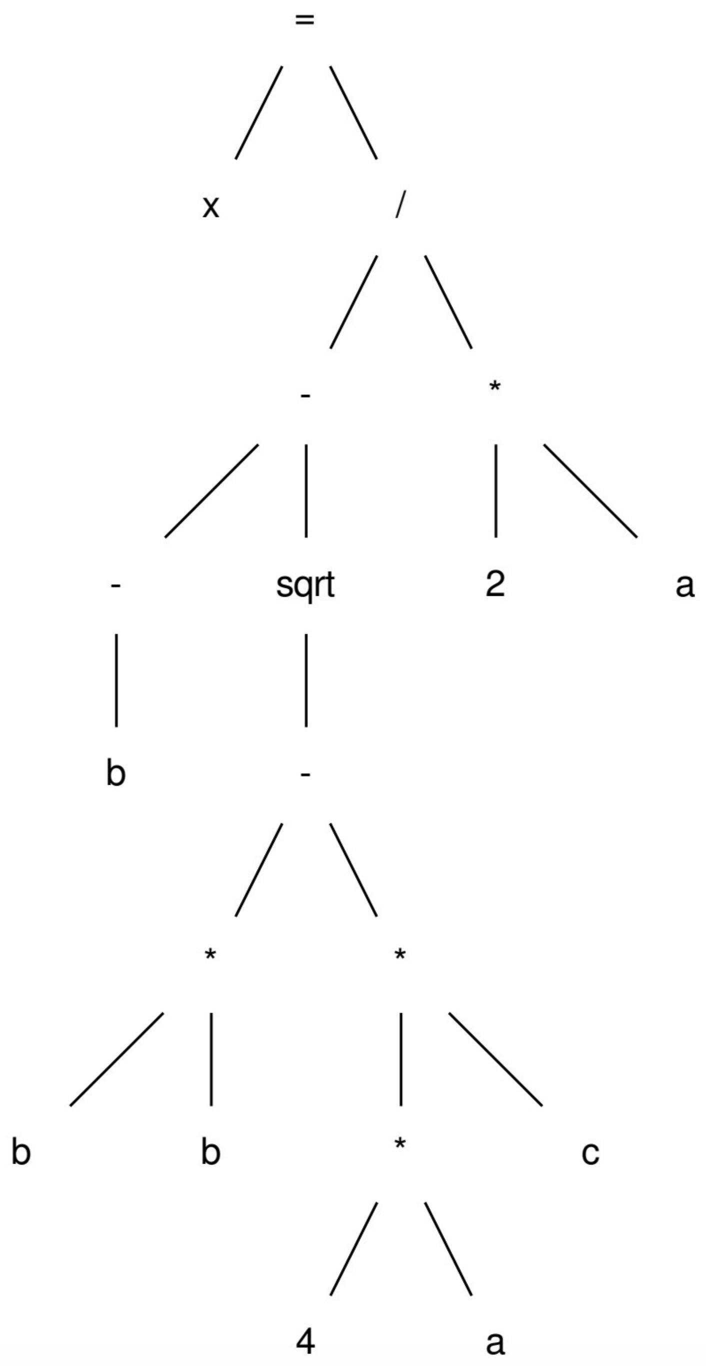
```
r0 = neg(b);
r1 = b * b;
r2 = 4 * a;
r3 = r2 * c;
r4 = r1 – r3;
```



Convert this code to 3 address code

post-order traversal, creating virtual registers for each node

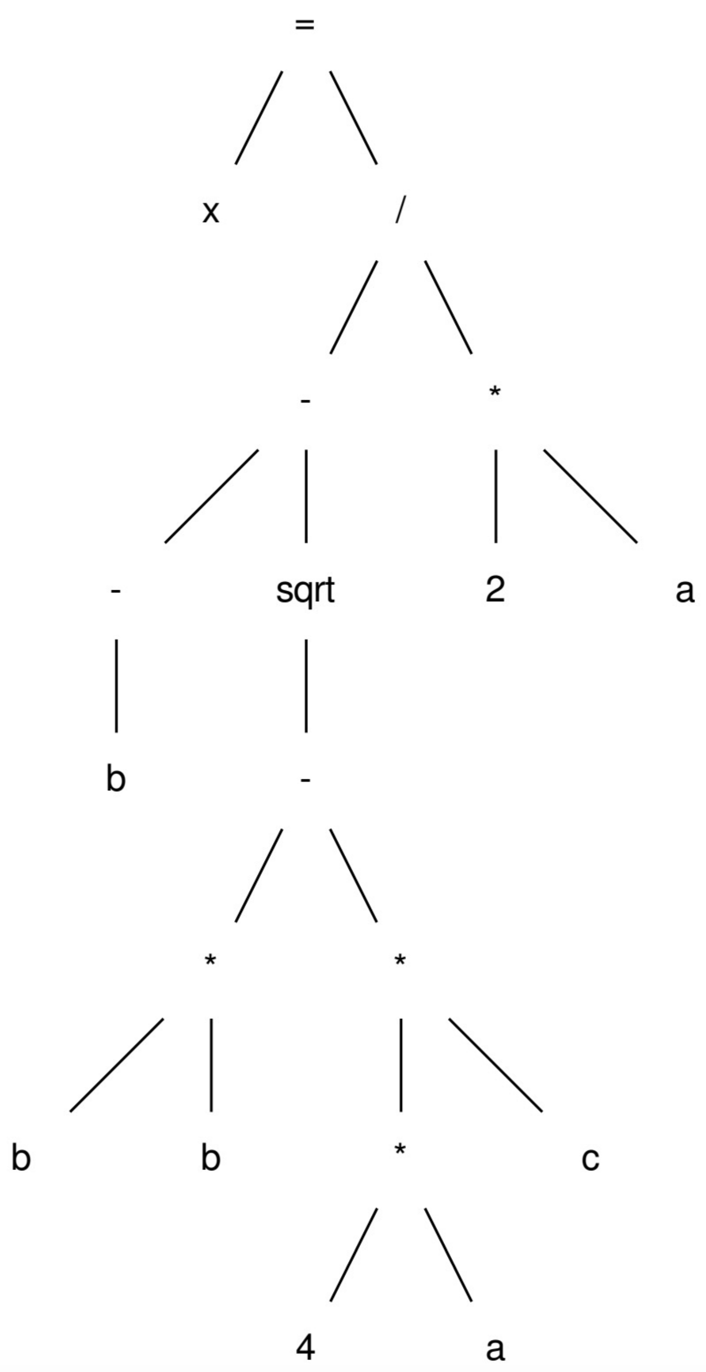
```
r0 = neg(b);
r1 = b * b;
r2 = 4 * a;
r3 = r2 * c;
r4 = r1 – r3;
r5 = sqrt(r4);
```



Convert this code to 3 address code

post-order traversal, creating virtual registers for each node

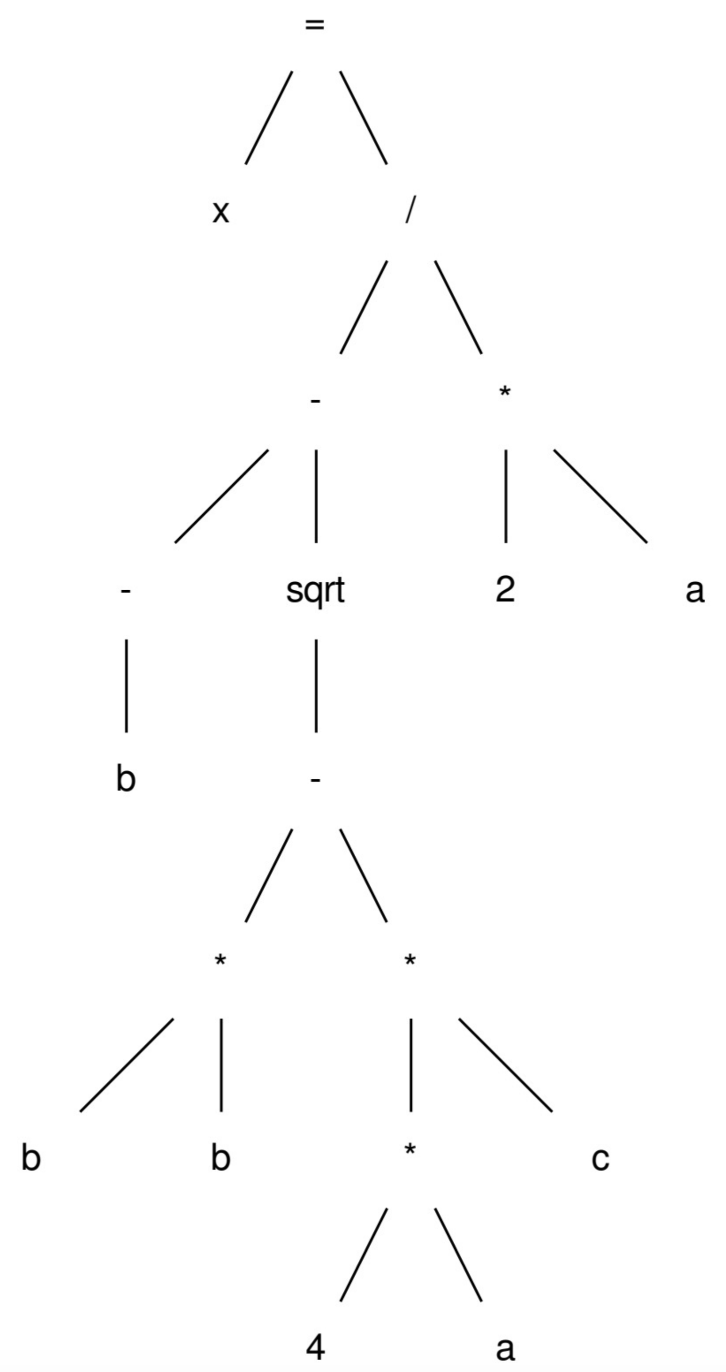
```
r0 = neg(b);
r1 = b * b;
r2 = 4 * a;
r3 = r2 * c;
r4 = r1 – r3;
r5 = sqrt(r4);
r6 = r0 – r5;
```



Convert this code to 3 address code

post-order traversal, creating virtual registers for each node

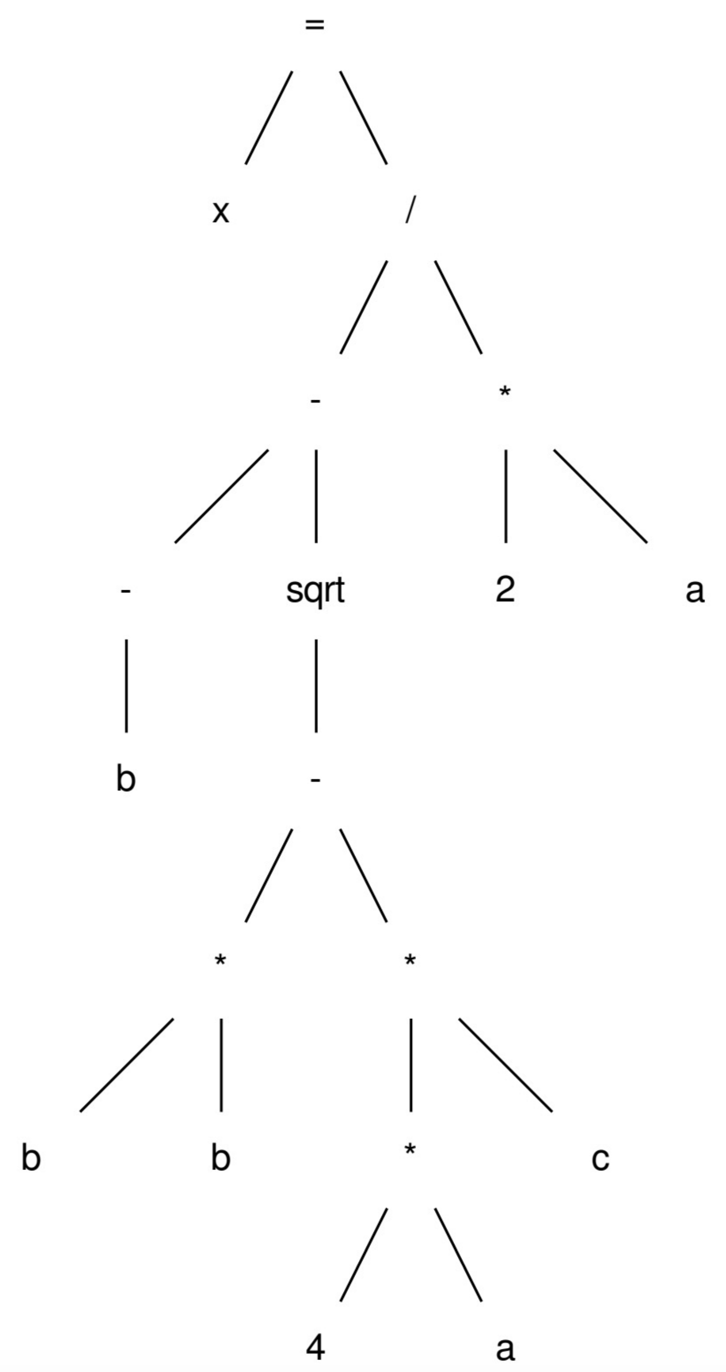
```
r0 = neg(b);
r1 = b * b;
r2 = 4 * a;
r3 = r2 * c;
r4 = r1 – r3;
r5 = sqrt(r4);
r6 = r0 – r5;
r7 = 2 * a;
```



Convert this code to 3 address code

post-order traversal, creating virtual registers for each node

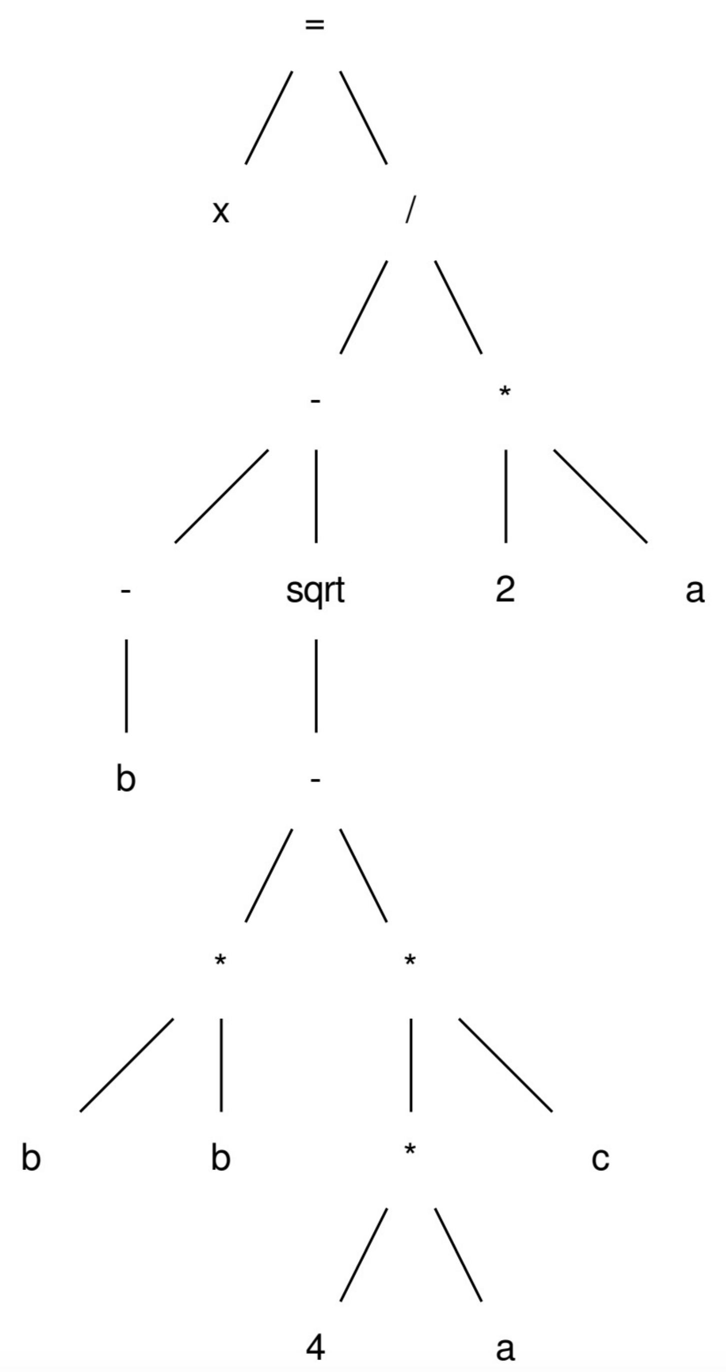
```
r0 = neg(b);
r1 = b * b;
r2 = 4 * a;
r3 = r2 * c;
r4 = r1 – r3;
r5 = sqrt(r4);
r6 = r0 – r5;
r7 = 2 * a;
r8 = r6 / r7;
```



Convert this code to 3 address code

post-order traversal, creating virtual registers for each node

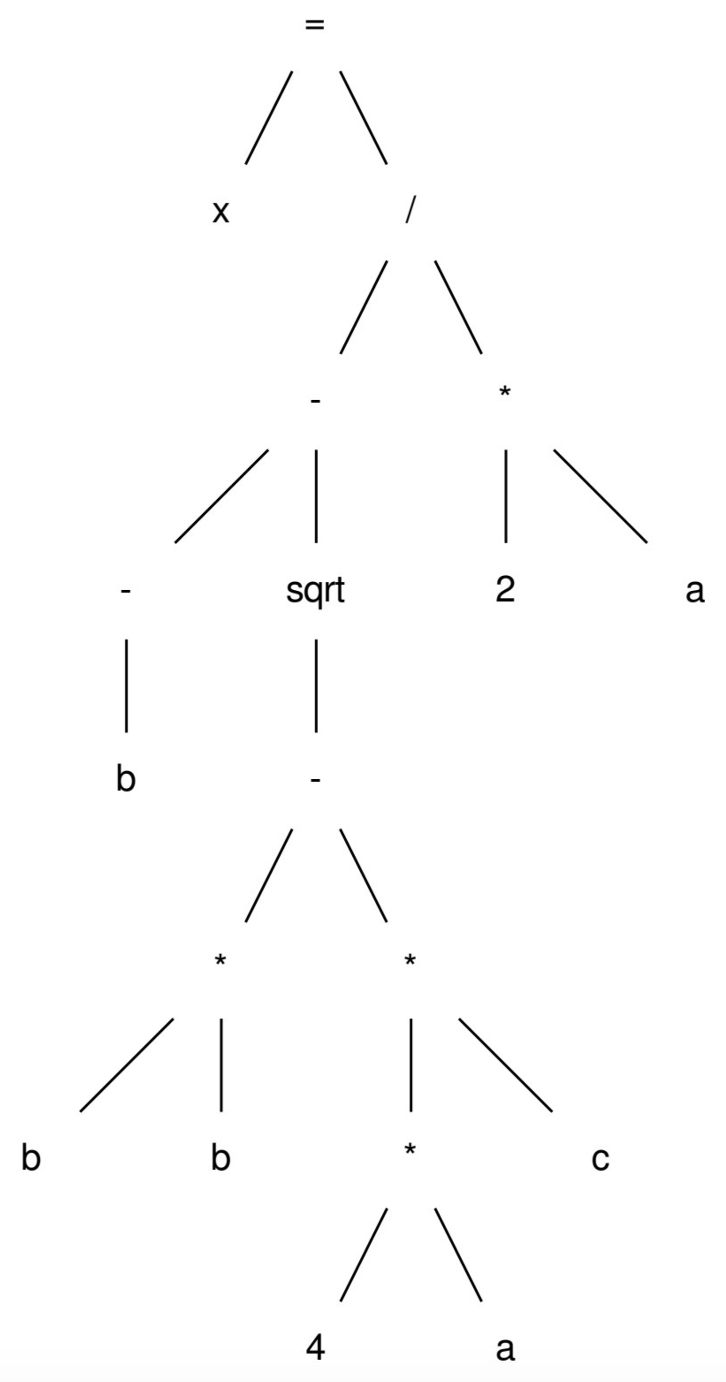
```
r0 = neg(b);
r1 = b * b;
r2 = 4 * a;
r3 = r2 * c;
r4 = r1 – r3;
r5 = sqrt(r4);
r6 = r0 – r5;
r7 = 2 * a;
r8 = r6 / r7;
x  = r8;
```



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;
```

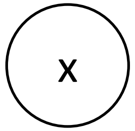
What now?

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;
```

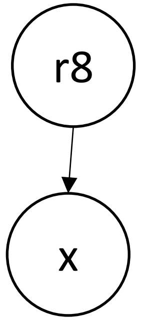
We can perform more optimizations, example:
by making a data-dependency graph (DDG)

```
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;
```



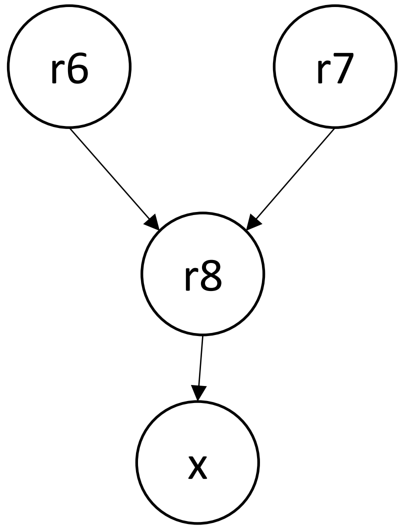
We can perform more optimizations, example:
by making a data-dependency graph (DDG)

```
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;
```



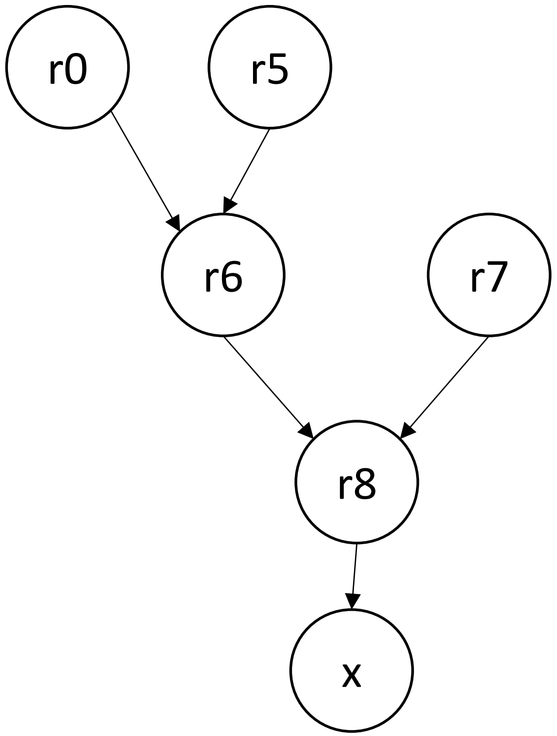
We can perform more optimizations, example:
by making a data-dependency graph (DDG)

```
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;
```



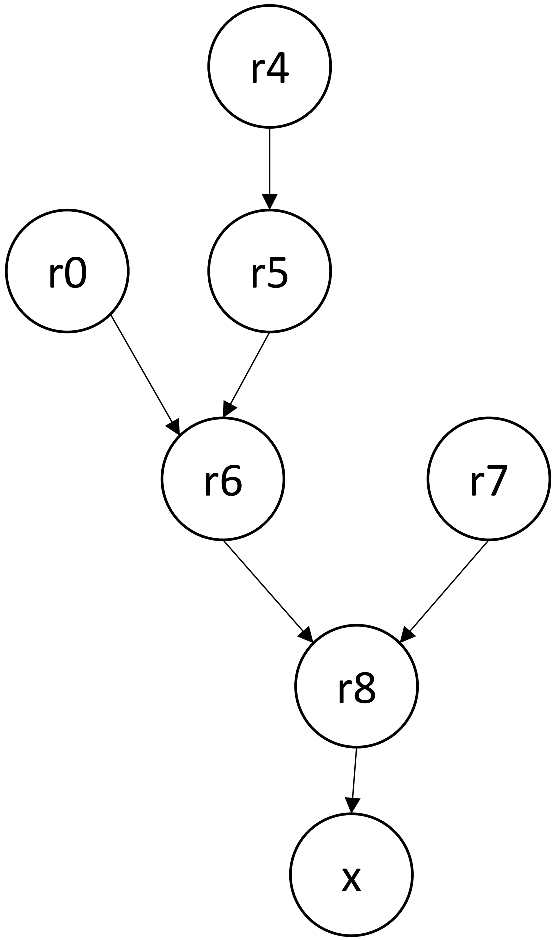
We can perform more optimizations, example:
by making a data-dependency graph (DDG)

```
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;
```



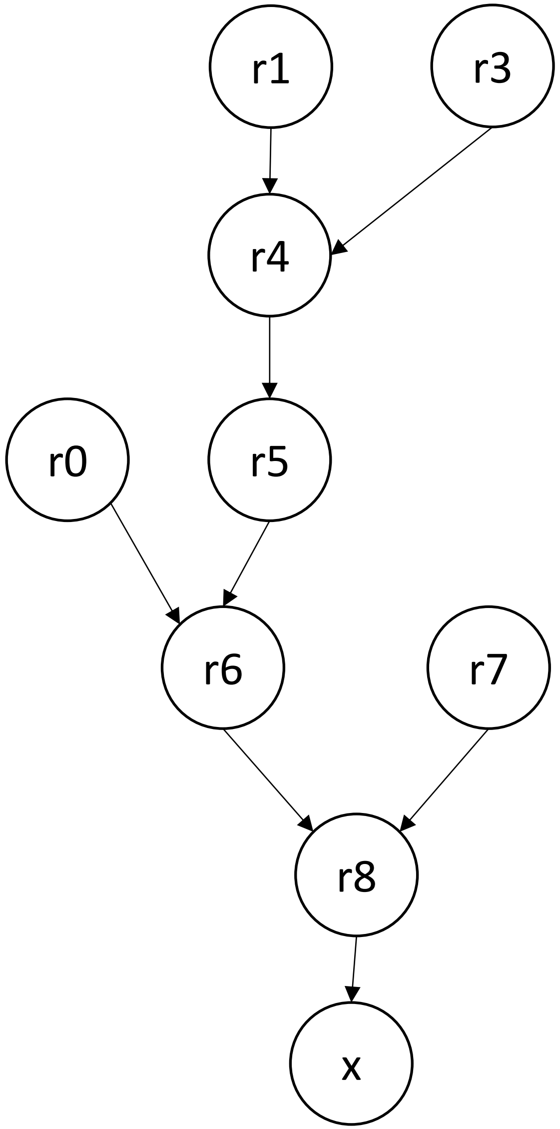
We can perform more optimizations, example:
by making a data-dependency graph (DDG)

```
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;
```



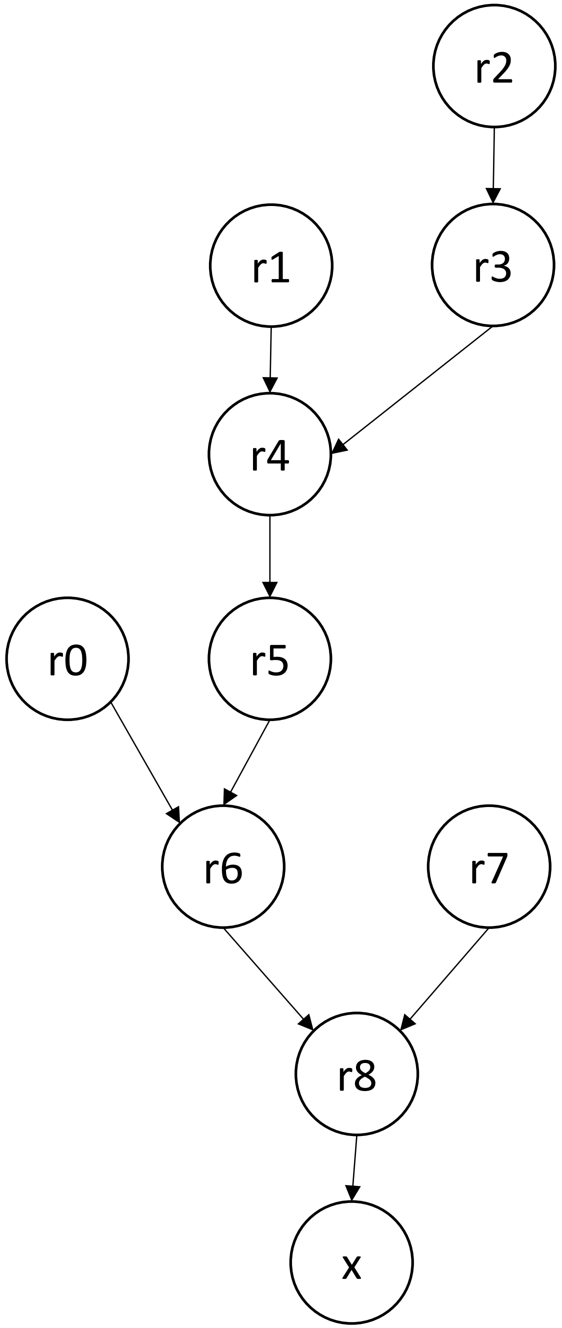
We can perform more optimizations, example: by making a data-dependency graph (DDG)

```
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)

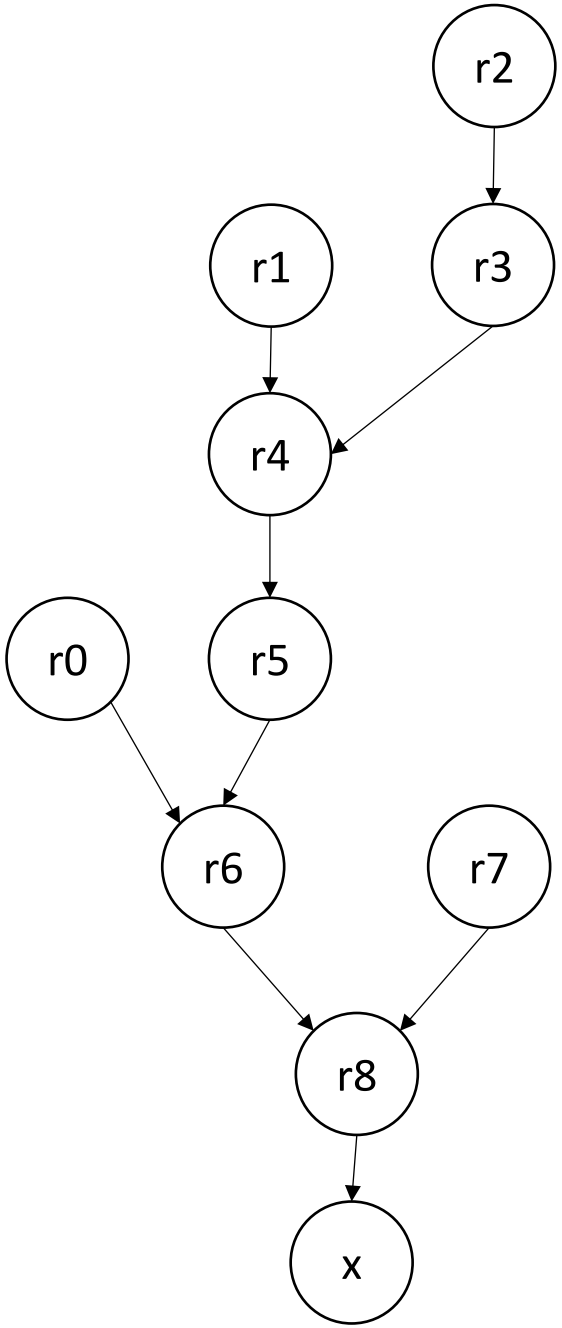
```
r0 = neg(b);
r1 = b * b;
r2 = 4 * a;
r3 = r2 * c;
r4 = r1 – r3;
r5 = sqrt(r4);
r6 = r0 – r5;
r7 = 2 * a;
r8 = r6 / r7;
x  = r8;
```



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;
```

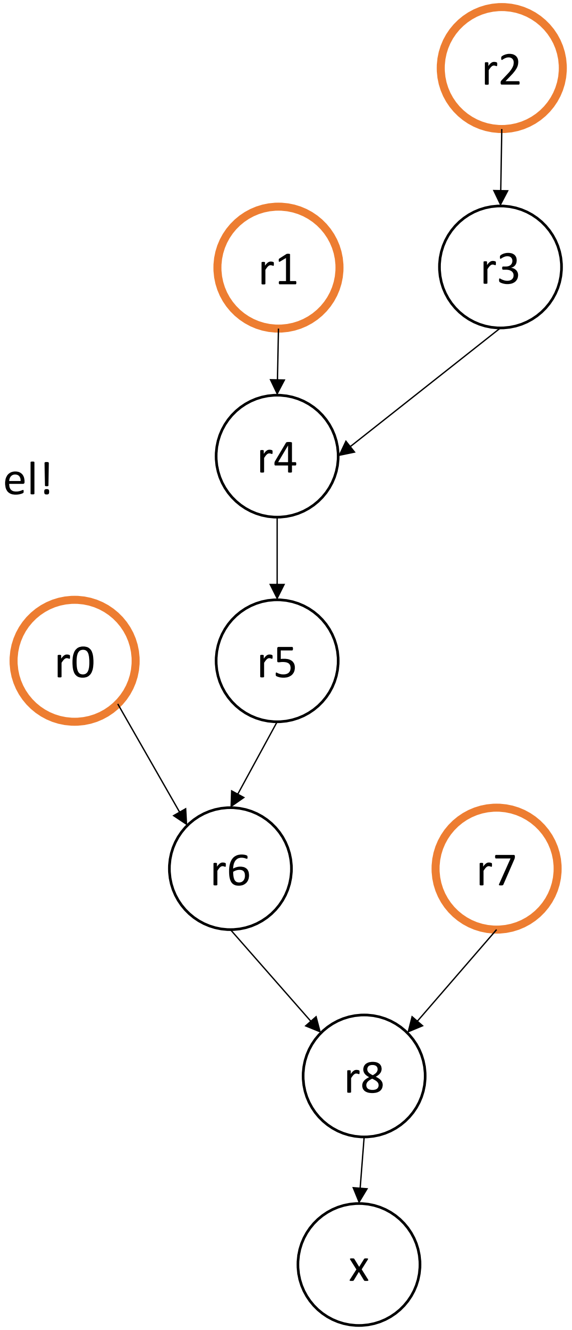
What can this tell us?



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!

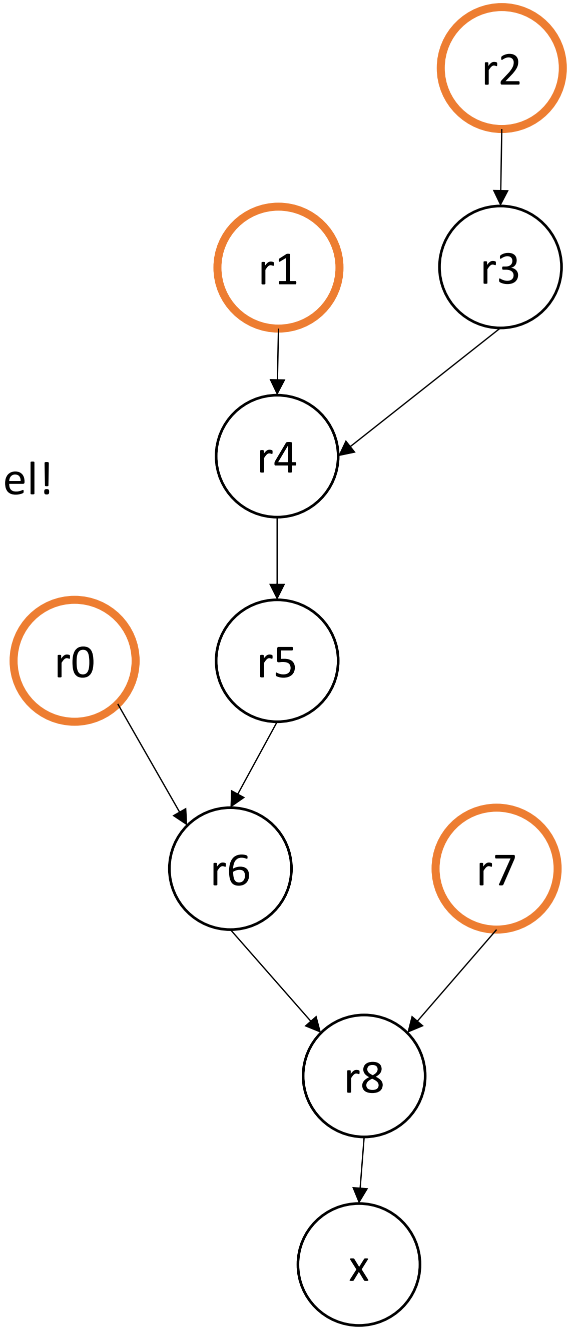


We can perform more optimizations, example:
by making a data-dependency graph (DDG)

```
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!

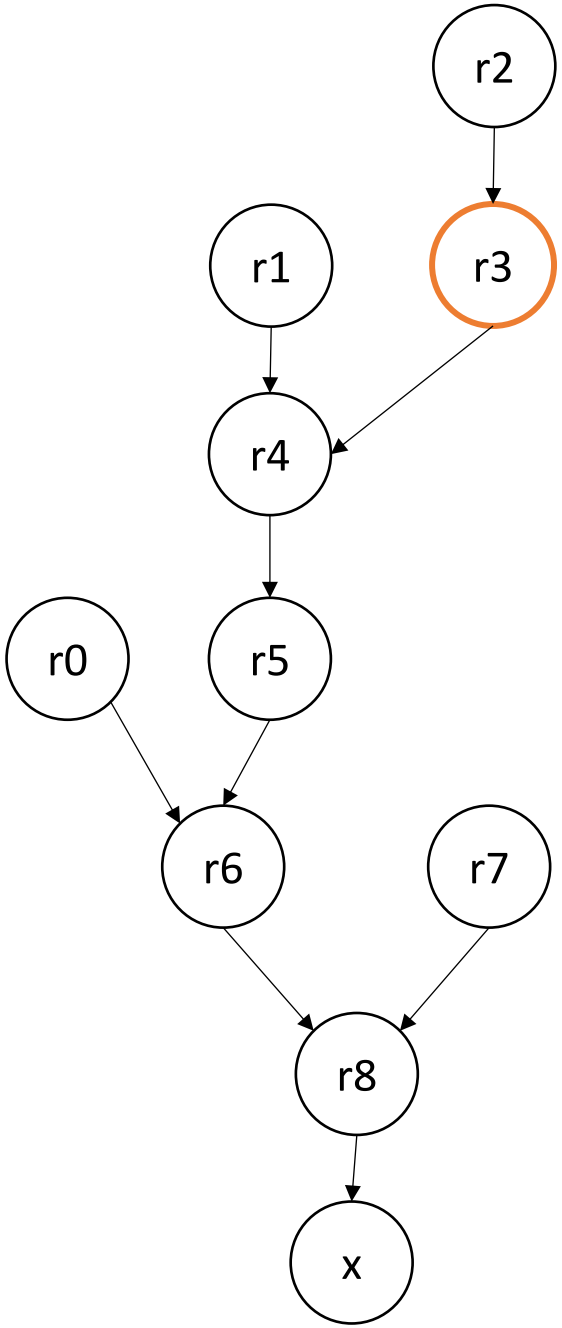
can be done in parallel!



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?

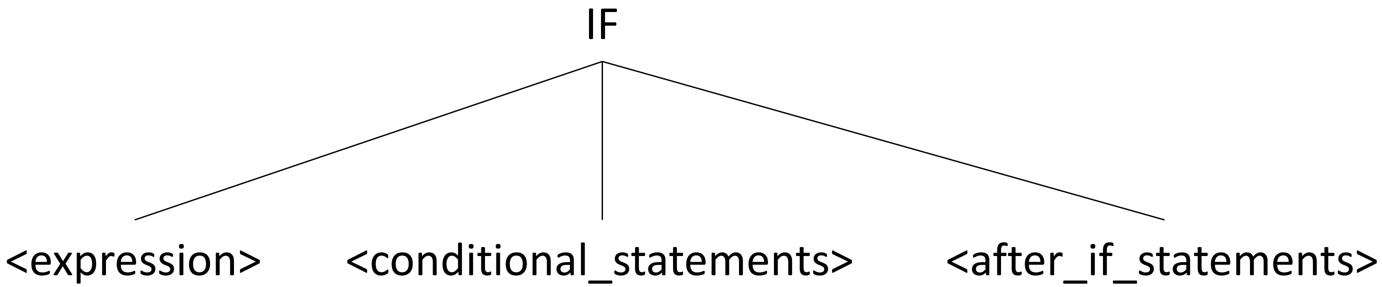
```
if (expr) {
 // conditional statements
}
// after if statements
```

First, produce an AST

What about control flow?

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

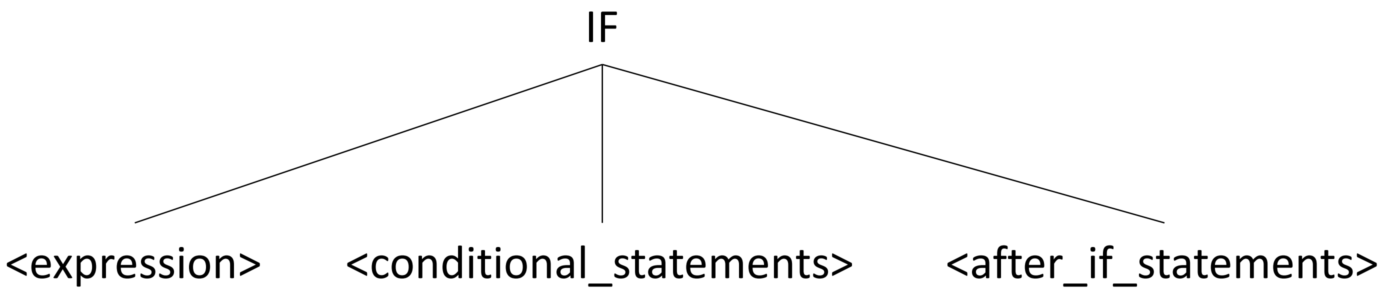
Next lower to 3 address code



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
```

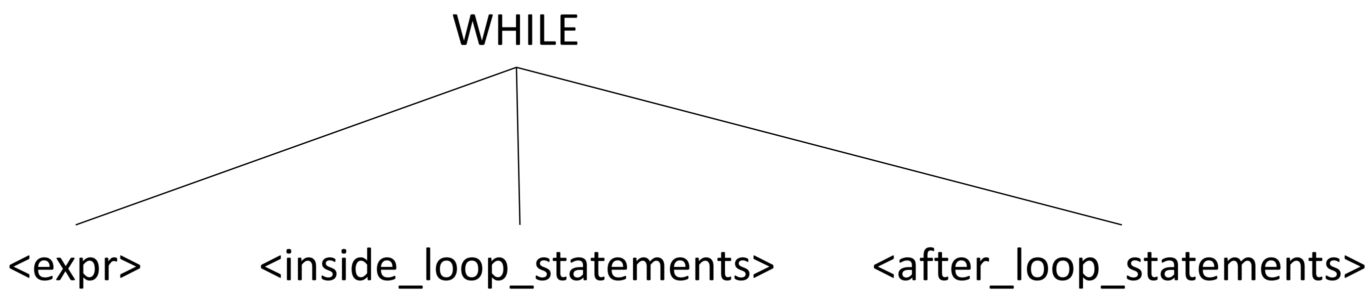
What about control flow?

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

First, produce an AST

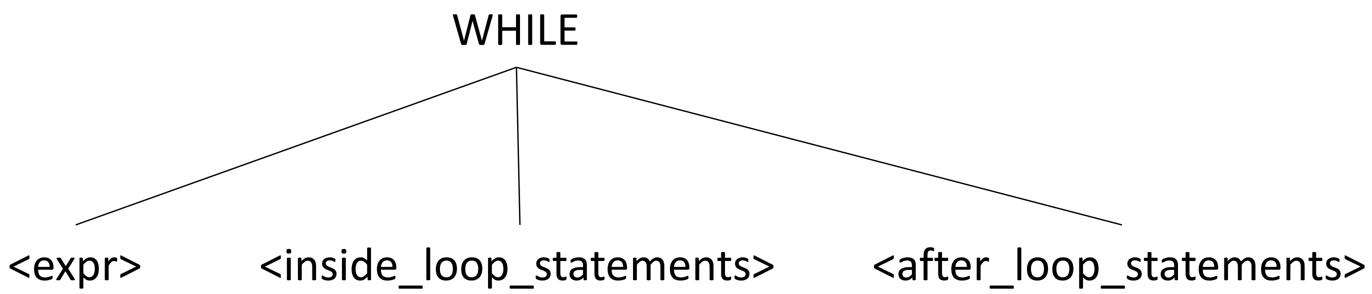
What about control flow?

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



What about control flow?

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



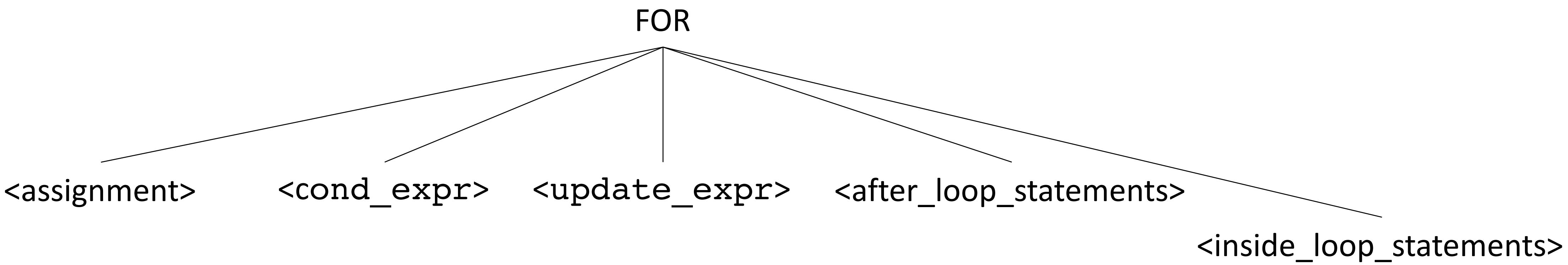

```
beginning_label:
r0 = <expr>

br r0, inside_loop, after_loop;

inside_loop:
<inside_loop_statements>
br beginning_label;

after_loop:
<after_loop_statements>
```

For loop

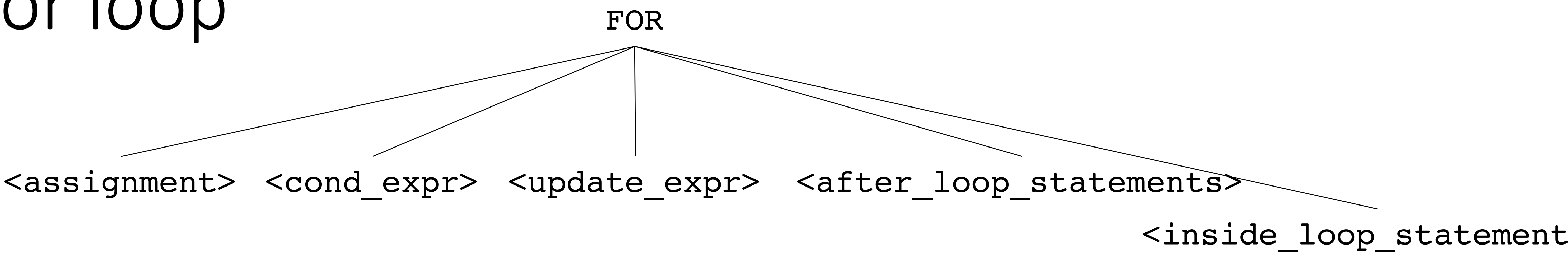
```
for (assignment; cond_expr; update_expr) {
 // inside_loop_statements
}
// after_loop_statements
```

For loop

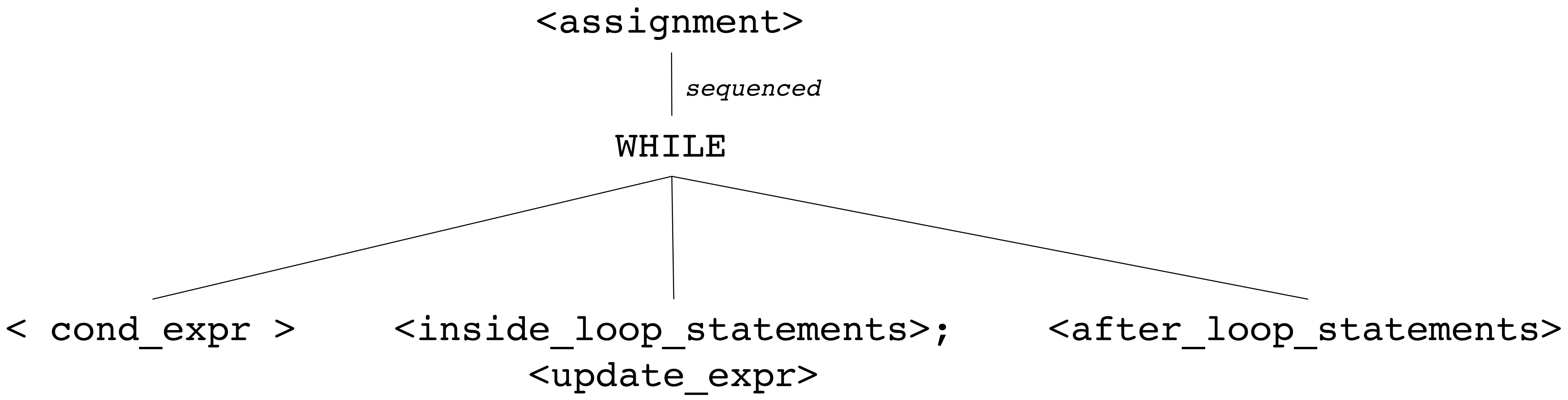
```
for (assignment; cond_expr; update_expr) {
 // inside_loop_statements
}
// after_loop_statements
```



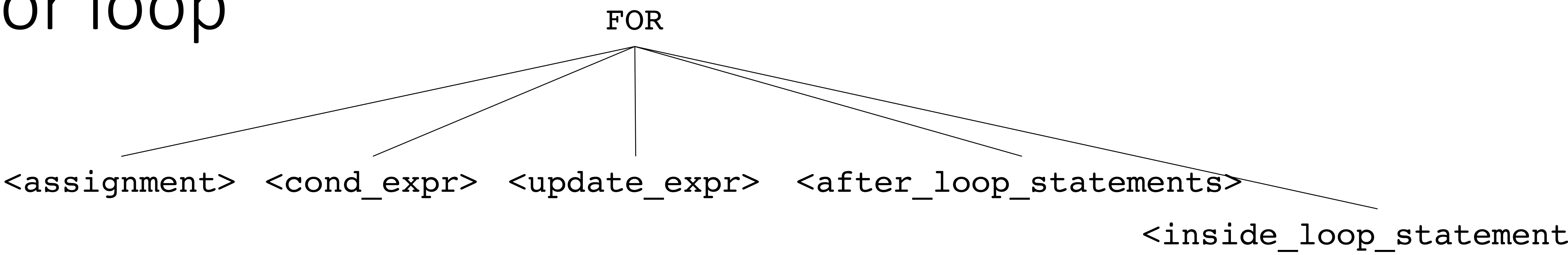
For loop



Can be de-sugared into a while loop:

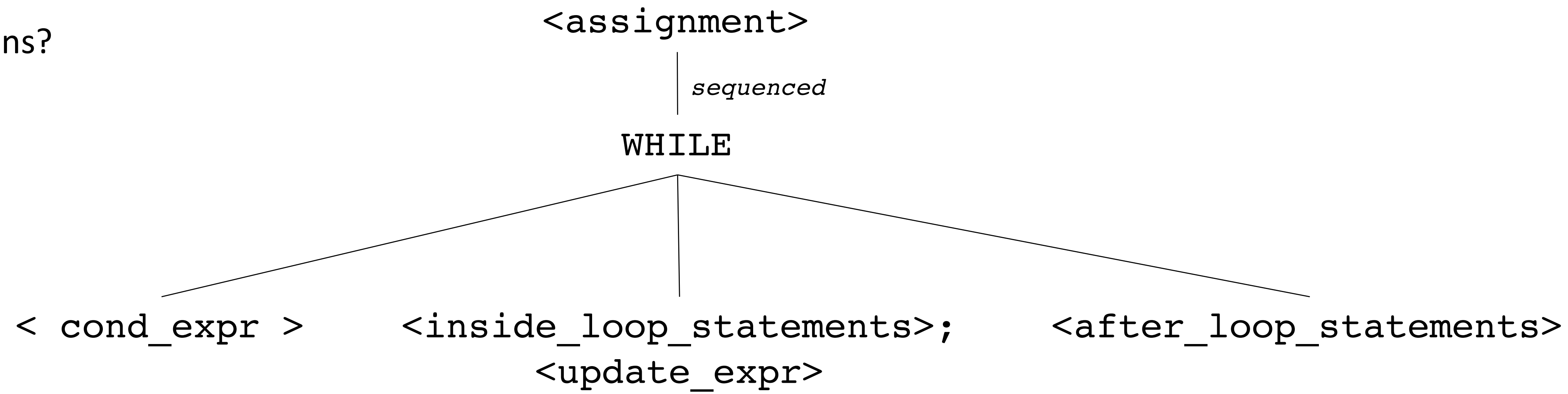


For loop



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;
```

IR Program structure

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

- 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

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 preceding instructions will execute

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;
```

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;

Next Wednesday

- A basic-block local optimization
  - local value numbering

- Friday: Control flow graphs and intra-block analysis

- Work on the homework! Thanks for all the discussion and patience!
  - I am still working on tuning the assignments for this class
  - Please give feedback!