CSE211: Compiler Design

Oct. 15, 2020

- **Topic**: Review of parsing with derivatives and IRs

- **Questions**:

  Questions/comments about derivatives and readings?

- δ_c *(re)*, where *re* is:
  - $re_{rhs} \cdot re_{lhs}$

    $\delta_c(re_{rhs}) \cdot re_{lhs}$ |

    if ε in re_{rhs} *then* $\delta_c(re_{lhs})$ *else* {}

Announcements

- Start of module 2
  - but with some review of parsing with derivatives

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

- Hopefully you have started and can come to office hours or discuss on canvas!

Homework notes

- in PLY, production rules cannot span multiple lines (unless it is a new option)

- there is a nonassoc option for associativity. When might we use that?

- What does C do?

- (1 == 0) false
- (1 == 0 == 0) true

CSE211: Compiler Design

Oct. 15, 2020

- **Topic**: Review of parsing with derivatives and IRs

- **Questions**:

  Questions/comments about derivatives and readings?

- δ_c (*re*), where *re* is:
  - $re_{rhs} \cdot re_{lhs}$

    $\delta_c(re_{rhs}) \cdot re_{lhs}$ |

    if ε in re_{rhs} *then* $\delta_c(re_{lhs})$ *else* {}

Regular expressions recursive definition

regular expression =

|{}

| {ε}

| ”a” (single character)

| re_{lhs} \/ re_{rhs}

| re_{lhs} . re_{rhs}

| $re_{starred}$ *

Regular expressions recursive definition

regular expression =

$|\{\}$

$|\ \{\varepsilon\}$

| *"a" (single character)*

$|\ re_{lhs} \setminus/\ re_{rhs}$

$|\ re_{lhs} \,.\, re_{rhs}$

$|\ re_{starred}\ *$

$re = \{\}$

$re = \{\varepsilon\}$

re = "a"

Regular expressions recursive definition

regular expression =

|{}

| {ε}

| ”a” (single character)

| re_{lhs} *\|* re_{rhs}

| re_{lhs} *.* re_{rhs}

| $re_{starred}$ *

re = “a.b”

Regular expressions recursive definition

regular expression =

|{}

| {ε}

| ”a” (single character)

| re_{lhs} *\|* re_{rhs}

| re_{lhs} *.* re_{rhs}

| $re_{starred}$ ***

re = “a.b”

=

re_{lhs} *.* re_{rhs}

“a” *“b”*

parse tree for a regular expression

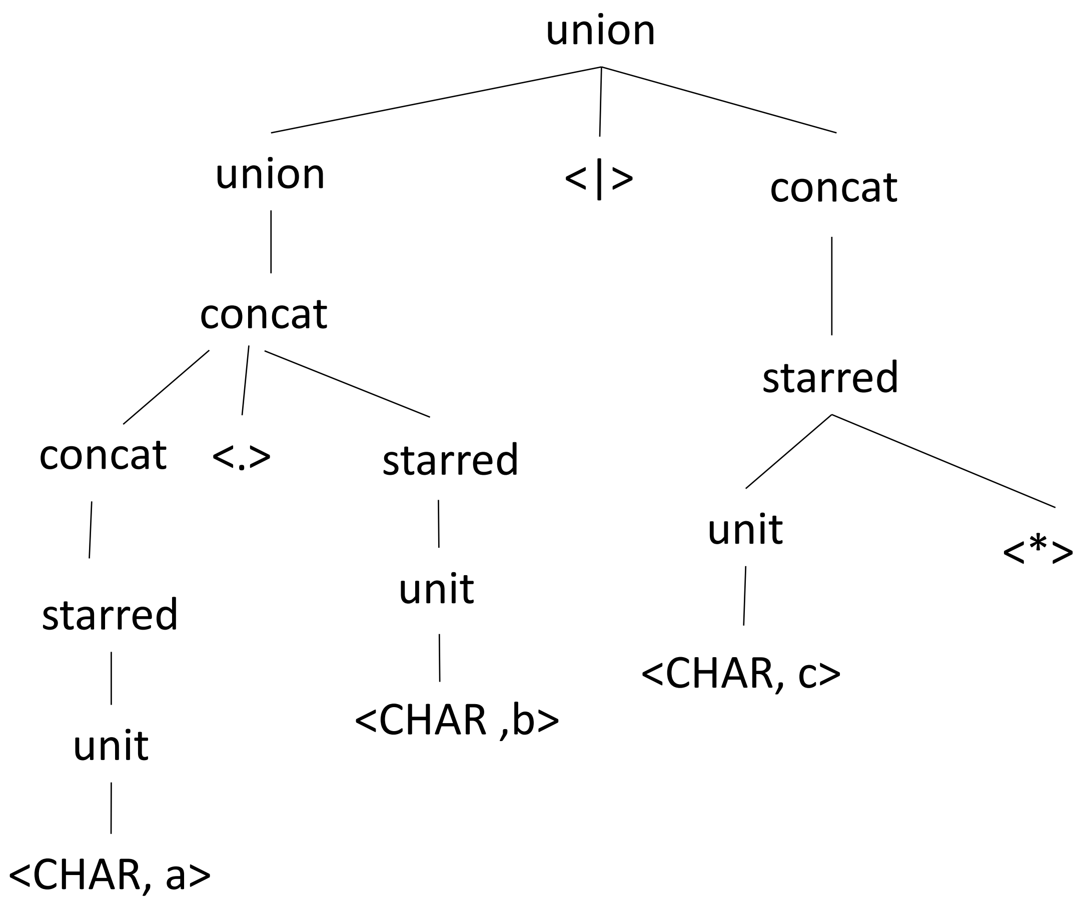
`input: a.b | c*`

<table>
<tr><th>Operator</th><th>Name</th><th>Productions</th></tr>
<tr><td>|</td><td>union</td><td>: union \| concat
| concat</td></tr>
<tr><td>.</td><td>concat</td><td>: concat . starred
| starred</td></tr>
<tr><td>*</td><td>starred</td><td>: starred *
| unit</td></tr>
<tr><td>()</td><td>unit</td><td>: (union)
| CHAR</td></tr>
</table>

parse tree for a regular expression

<table>
<tr><th>Operator</th><th>Name</th><th>Productions</th></tr>
<tr><td>|</td><td>union</td><td>: union \| concat
| concat</td></tr>
<tr><td>.</td><td>concat</td><td>: concat . starred
| starred</td></tr>
<tr><td>*</td><td>starred</td><td>: starred *
| unit</td></tr>
<tr><td>()</td><td>unit</td><td>: (union)
| CHAR</td></tr>
</table>

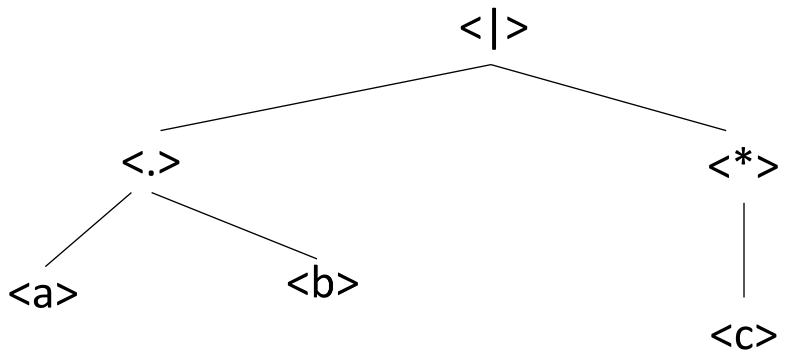
`input: a.b | c*`

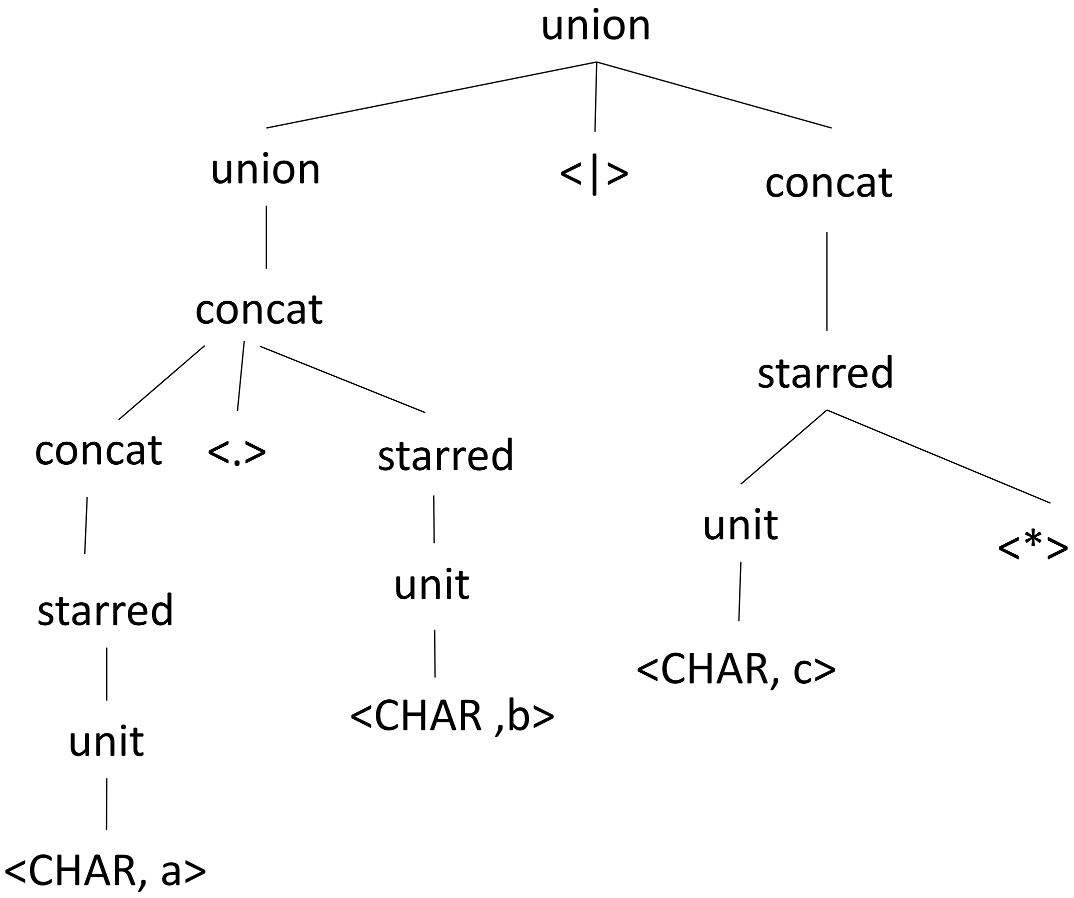


parse tree for a regular expression

input: a.b | c*

abstract syntax tree

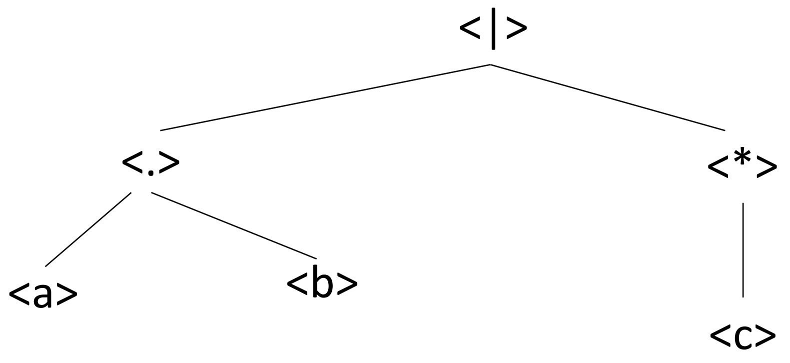




parse tree for a regular expression

`input: a.b | c*`

abstract syntax tree

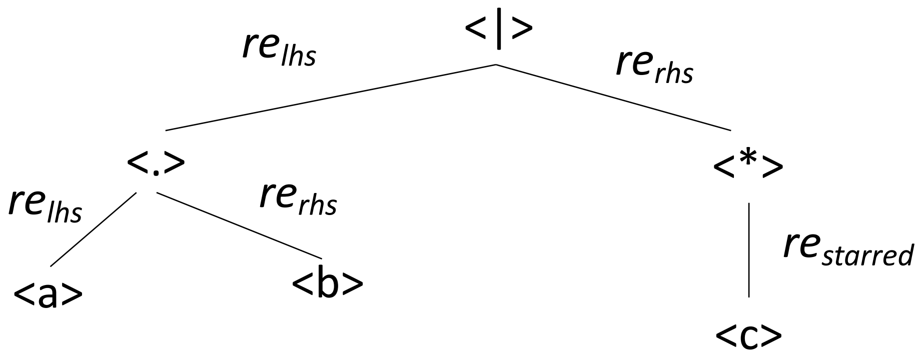


- *regular expression =*
  |{}
  | ε
  | a (single character)
  | re_{lhs} | re_{rhs}
  | re_{lhs} . re_{rhs}
  *| $re_{starred}$ **

parse tree for a regular expression

```
input: a.b | c*
```

abstract syntax tree

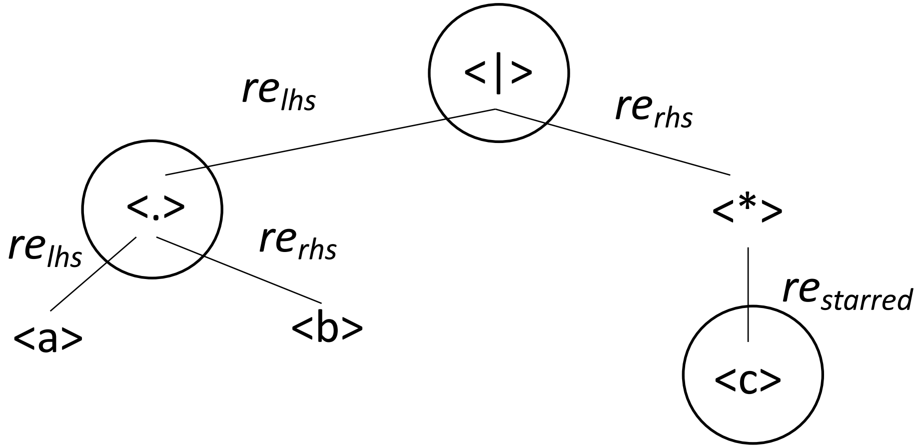


- *regular expression =*
  - *| {}*
  - *| ε*
  - *| a (single character)*
  - *|* re_{lhs} *|* re_{rhs}
  - *|* re_{lhs} *.* re_{rhs}
  - *|* $re_{starred}$ ***

parse tree for a regular expression

`input: a.b | c*`

abstract syntax tree



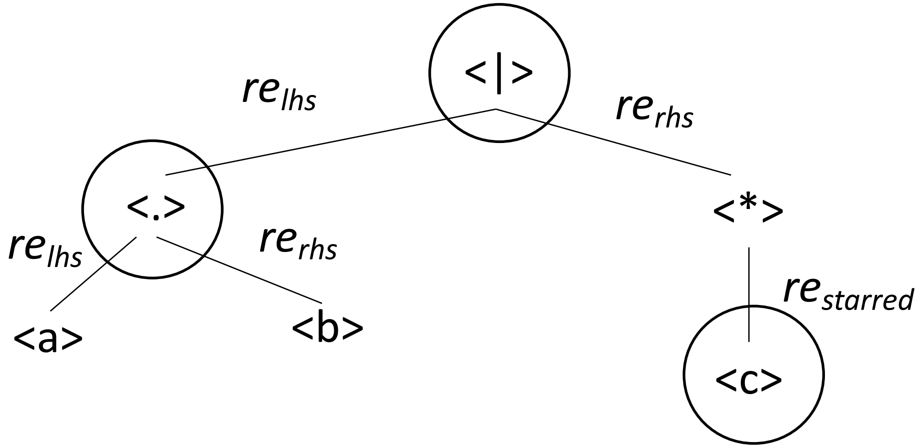
each node is
also a regular expression!

- *regular expression =*
  - *| {}*
  - *| ε*
  - *| a (single character)*
  - *| re_{lhs} | re_{rhs}*
  - *| re_{lhs} . re_{rhs}*
  - *| $re_{starred}$ **

parse tree for a regular expression

`input: a.b | c*`

abstract syntax tree



each node is
also a regular expression!

- *Check homework code to see AST construction*
- *Question: given a regular expression RE, how check if a string is in the language?*
- *parsing with derivatives!*

Language Derivatives Examples

- *L = {"aaa", "ab", "ba", "bba"}*
- $\delta_a (L)$ *= {"aa", "b"}*
- $\delta_{aa} (L)$ *= {"a"}*
- $\delta_b (L)$ *= {"a", "ba"}*
- $\delta_{ba} (L)$ *=*

Language Derivatives Examples

- $L = \{\text{“}aaa\text{”}, \text{“}ab\text{”}, \text{“}ba\text{”}, \text{“}bba\text{”}\}$
- $\delta_a (L) = \{\text{“}aa\text{”}, \text{“}b\text{”}\}$
- $\delta_{aa} (L) = \{\text{“}a\text{”}\}$
- $\delta_b (L) = \{\text{“}a\text{”}, \text{“}ba\text{”}\}$
- $\delta_{ba} (L) = \{\varepsilon\}$

Regular expressions are closed under derivatives

- Given a regular expression *re*, any derivative of *re* is also a regular expression

- *Let's try some!*

Regular expressions are closed under derivatives

- *re = "a"*
  L(re) = {"a"}

- δ_a*(re) = ""*

- δ_b*(re) = None*

Regular expressions are closed under derivatives

- *re* = "*a*"
- $\delta_a(re) = \{\varepsilon\}$
- $\delta_b(re) = \{\}$

Regular expressions are closed under derivatives

- *re = “a | b”*
  $L = \{“a”, “b”\}$

- $\delta_a(re) = “”$

- $\delta_b(re) = “”$

Regular expressions are closed under derivatives

- re = "$a \mid b$"
- $\delta_a(re) = \{\varepsilon\}$
- $\delta_b(re) = \{\varepsilon\}$

Regular expressions are closed under derivatives

- *re = "a.a | a.b"*
  L = {"ab", "aa"}

- δ_a*(re) = "b | a"*

- δ_b*(re) = None*

Regular expressions are closed under derivatives

- *re* = *"a.a | a.b"*

- $\delta_a(re)$ = *"a | b"*

- $\delta_b(re) = \{\}$

Regular expressions are closed under derivatives

- *re = “(a.b.c)*”*

  L = {“”, ”abc”, “abcabc”, “abcabcabc” …}

- δ_a*(re) = “b.c.(a.b.c)*”*

  δ_a*(L) = {“bc”, “bcabc”, “bcabcabc” …}*

Regular expressions are closed under derivatives

- *re = “(a.b.c)*”*

- δ_a*(re) = “b.c.(a.b.c)*”*

What is a method for computing the derivative?

Consider the base cases

- δ_c (*re)* = match re with:
  - {}
    return {}
  - $\{\varepsilon\}$
    return {}
  - *"a"* (single character)
    if "a" == c then return $\{\varepsilon\}$
    else return {}

- *regular expression =*
  | {}
  | ε
  | a (single character)
  | re_{lhs} | re_{rhs}
  | re_{lhs} . re_{rhs}
  *| $re_{starred}$ **

Derivative Recursive Cases

Consider the recursive cases:

- δ_c *(re)* = match re with:
  - re_{lhs} | re_{rhs}

    return $\delta_c(re_{lhs})$ | δ_c (re_{rhs})
  - $re_{starred}$*

    return $\delta_c(re_{starred})$. $re_{starred}$*
  - re_{lhs} . re_{rhs}

    return $\delta_c(re_{lhs})$. re_{rhs} |

    if ε in re_{lhs} *then* $\delta_c(re_{rhs})$ *else {}*

- *regular expression =*

  | {}

  | ε

  | a (single character)

  | re_{lhs} *|* re_{rhs}

  | re_{lhs} . re_{rhs}

  | $re_{starred}$ *

Derivative Recursive Cases

Consider the recursive cases:

- δ_c (*re*) = match re with:
  - re_{lhs} | re_{rhs}

    return $\delta_c(re_{lhs})$ | $\delta_c(re_{rhs})$

Example:

re = "a.a | a.b"

δ_a*(re) = "a | b"*

Derivative Recursive Cases

Consider the recursive cases:

- δ_c (*re*) = match re with:
  - $re_{starred}$*

    return $\delta_c(re_{starred})$. $re_{starred}$*

Example:

re = "(a.b.c)"*

δ_a*(re) = "b.c.(a.b.c)*"*

Derivative Recursive Cases

Consider the recursive cases:

- $\delta_c\,(re)$ = match re with:
  - $re_{lhs}\ .\ re_{rhs}$

return $\delta_c(re_{lhs})\ .\ re_{rhs}$ |

if ε in re_{lhs} *then* $\delta_c(re_{rhs})$ *else {}*

Example:

re = "a.b"

$\delta_a(re)$ = *"b"*

Derivative Recursive Cases

Consider the recursive cases:

- $\delta_c\,(re)$ = match re with:
  - $re_{lhs}\ .\ re_{rhs}$

return $\delta_c(re_{lhs})\ .\ re_{rhs}\ |$

$\text{if } \varepsilon \text{ in } re_{lhs} \text{ then } \delta_c(re_{rhs}) \text{ else } \{\}$

Example:

re = "(a.c)*.a.b"

$\delta_a(re)$ = "c.(a.c)*.a.b | b"

Nullable operator

- NULL(re) =

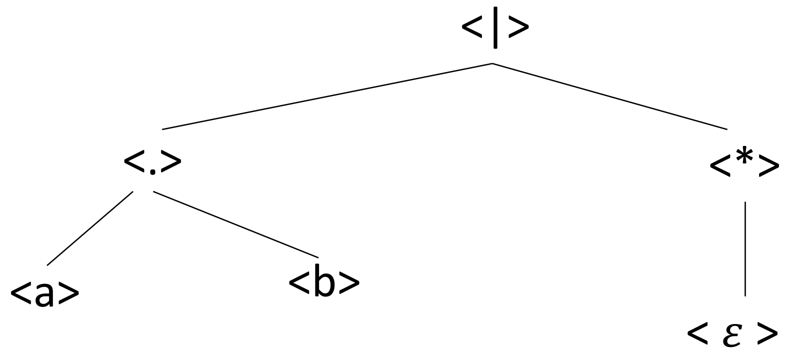
  $if\ \epsilon \in re$ then: *{ε}*
  else: {}

Nullable operator

- NULL(re) =

  if $\epsilon \in re$ then: *{ε}*
  else: {}

implement over a RE abstract syntax tree



- *regular expression =*

  | {}
  | ε
  | a (single character)
  | re_{lhs} *|* re_{rhs}
  | re_{lhs} *.* re_{rhs}
  | $re_{starred}$ ***

What is a method for computing NULL?

Consider the base cases

- NULL(*re*) = match re with:
  - {}
    return {}
  - {ε}
    return {ε}
  - *"a"* (single character)
    return {}

- *regular expression =*
  $|\ \{\}$
  $|\ \varepsilon$
  $|\ a$ *(single character)*
  $|\ re_{lhs}\ |\ re_{rhs}$
  $|\ re_{lhs}\ .\ re_{rhs}$
  $|\ re_{starred}\ ^{*}$

What is a method for computing NULL?

Consider the recursive cases:

- NULL(*re*) = match re with:
  - re_{lhs} | re_{rhs}

    return NULL(re_{lhs}) | NULL(re_{rhs})

  - $re_{starred}$*

    return {ε}

  - re_{lhs} . re_{rhs}

    return NULL(re_{lhs}) . NULL(re_{rhs})

- *regular expression =*

  | {}

  | ε

  | a (single character)

  | re_{lhs} | re_{rhs}

  | re_{lhs} . re_{rhs}

  *| $re_{starred}$ **

Derivative Recursive Cases

Consider the recursive cases:

- δ_c *(re)* = match re with:
  - re_{lhs} | re_{rhs}

    return $\delta_c(re_{lhs})$ | δ_c (re_{rhs})
  - $re_{starred}$*

    return $\delta_c(re_{starred})$. $re_{starred}$*
  - re_{lhs} . re_{rhs}

    return $\delta_c(re_{lhs})$. re_{rhs} |

    if ε in re_{lhs} *then* $\delta_c(re_{rhs})$ *else {}*

- *regular expression =*
  | {}
  | ε
  | a (single character)
  | re_{lhs} *|* re_{rhs}
  | re_{lhs} *.* re_{rhs}
  | $re_{starred}$ ***

Derivative Recursive Cases

Consider the recursive cases:

- δ_c *(re)* = match re with:
  - re_{lhs} | re_{rhs}

    return $\delta_c(re_{lhs})$ | δ_c (re_{rhs})
  - $re_{starred}$*

    return $\delta_c(re_{starred})$. $re_{starred}$*
  - re_{lhs} . re_{rhs}

    return $\delta_c(re_{lhs})$. re_{rhs} |

    $NULL(re_{lhs})$. $\delta_c(re_{rhs})$

- *regular expression =*
  - | {}
  - | ε
  - | *a (single character)*
  - | re_{lhs} | re_{rhs}
  - | re_{lhs} . re_{rhs}
  - | $re_{starred}$ *

Parsing REs with derivative

given a function δ_c to compute the derivative of an RE, the NULL function, an RE *re*, and a string $s = c_1 . c_2 . c_3 \ldots$ (concat of characters)

Can we check if *re* matches *s*?

Parsing REs with derivative

given a function δ_c to compute the derivative of an RE, the NULL function, an RE *re*, and a string $s = c_1 \,.\, c_2 \,.\, c_3 \ldots$ (concat of characters)

Can we check if *re* matches *s*?

$L(re) = \{.. \; s \; ..\}$

Parsing REs with derivative

given a function δ_c to compute the derivative of an RE, the NULL function, an RE *re*, and a string $s = c_1 . c_2 . c_3 \ldots$ (concat of characters)

Can we check if *re* matches *s*?

δ_{c1} *(re)*

L(re) = {.. s ..}

$L(\delta_{c1}(\mathrm{re})) = \{..\ s[1:]\ ..\}$

Parsing REs with derivative

given a function δ_c to compute the derivative of an RE, the NULL function, an RE *re*, and a string $s = c_1 . c_2 . c_3 \ldots$ (concat of characters)

Can we check if *re* matches *s*?

	δ_{c1} *(re)*	δ_{c2} (δ_{c1} (re)) = $\delta_{c1,c2}$ *(re)*
L(re) = {.. s ..}		
	$L(\delta_{c1}$ (re)) = {.. s[1:] ..}	$L(\delta_{c1,c2}$ (re)) = {.. s[2:] ..}

Parsing REs with derivative

given a function δ_c to compute the derivative of an RE, the NULL function, an RE *re*, and a string $s = c_1 . c_2 . c_3 \ldots$ (concat of characters)

Can we check if *re* matches *s*?

	$\delta_{c1}(re)$	$\delta_{c2}(\delta_{c1}(\mathrm{re})) = \delta_{c1,c2}(re)$	$\delta_s(re)$
$L(re) = \{.. s ..\}$			
	$L(\delta_{c1}(\mathrm{re})) = \{.. s[1:] ..\}$	$L(\delta_{c1,c2}(\mathrm{re})) = \{.. s[2:] ..\}$	$\mathrm{L}(\delta_s(re)) = \{.. \varepsilon ..\}$

Parsing REs with derivative

given a function δ_c to compute the derivative of an RE, the NULL function, an RE *re*, and a string $s = c_1 . c_2 . c_3 ...$ (concat of characters)

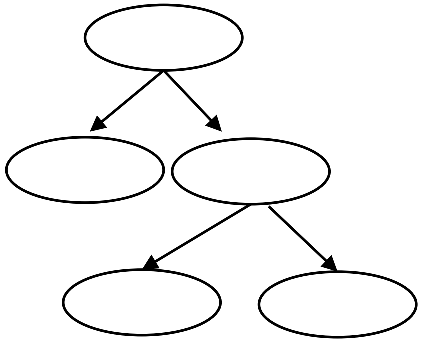
Can we check if *re* matches *s*?

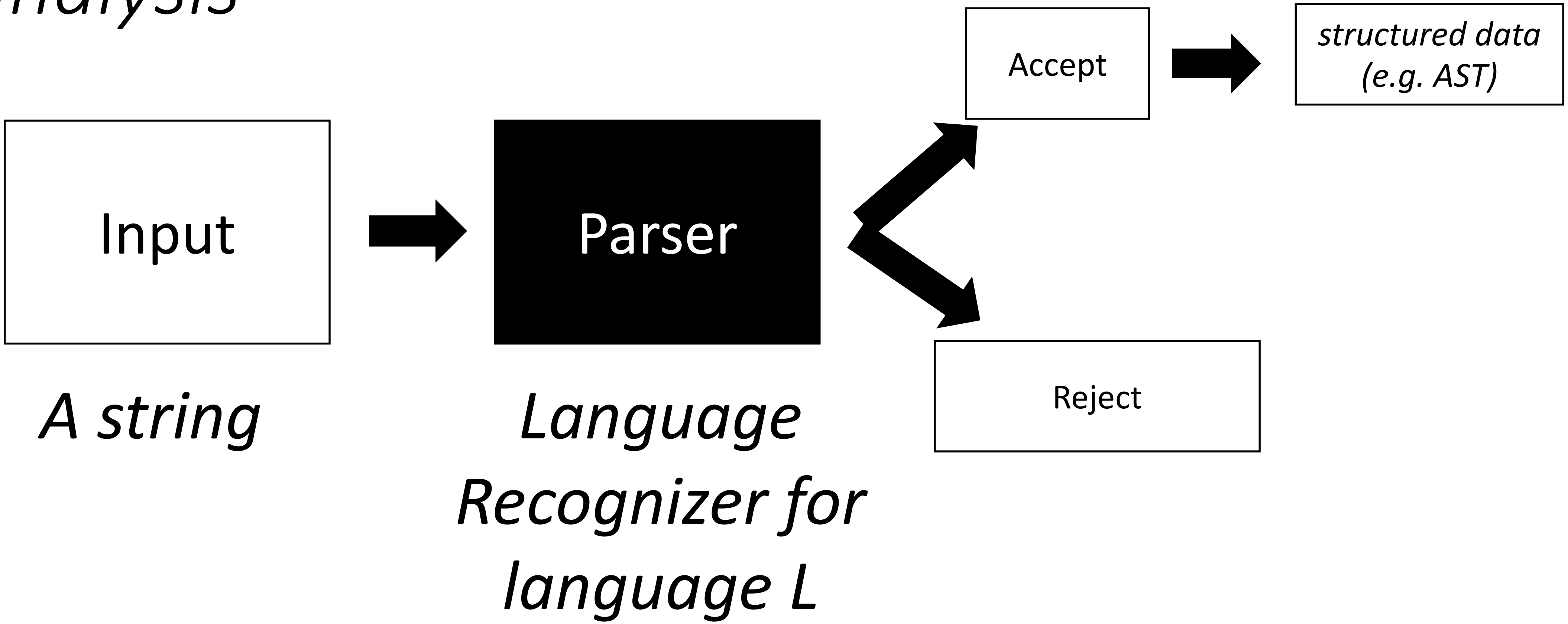
If this is true,
Then *re matches* s

	$\delta_{c1}(re)$	$\delta_{c2}(\delta_{c1}(\text{re})) = \delta_{c1,c2}(re)$	$\delta_s(re)$	
$L(re) = \{.. \ s \ ..\}$				$NULL(\delta_s(re)) == \{\varepsilon\}$
	$L(\delta_{c1}(\text{re})) = \{.. \ s[1:] \ ..\}$	$L(\delta_{c1,c2}(\text{re})) = \{.. \ s[2:] \ ..\}$	$L(\delta_s(re)) = \{.. \ \varepsilon \ ..\}$	

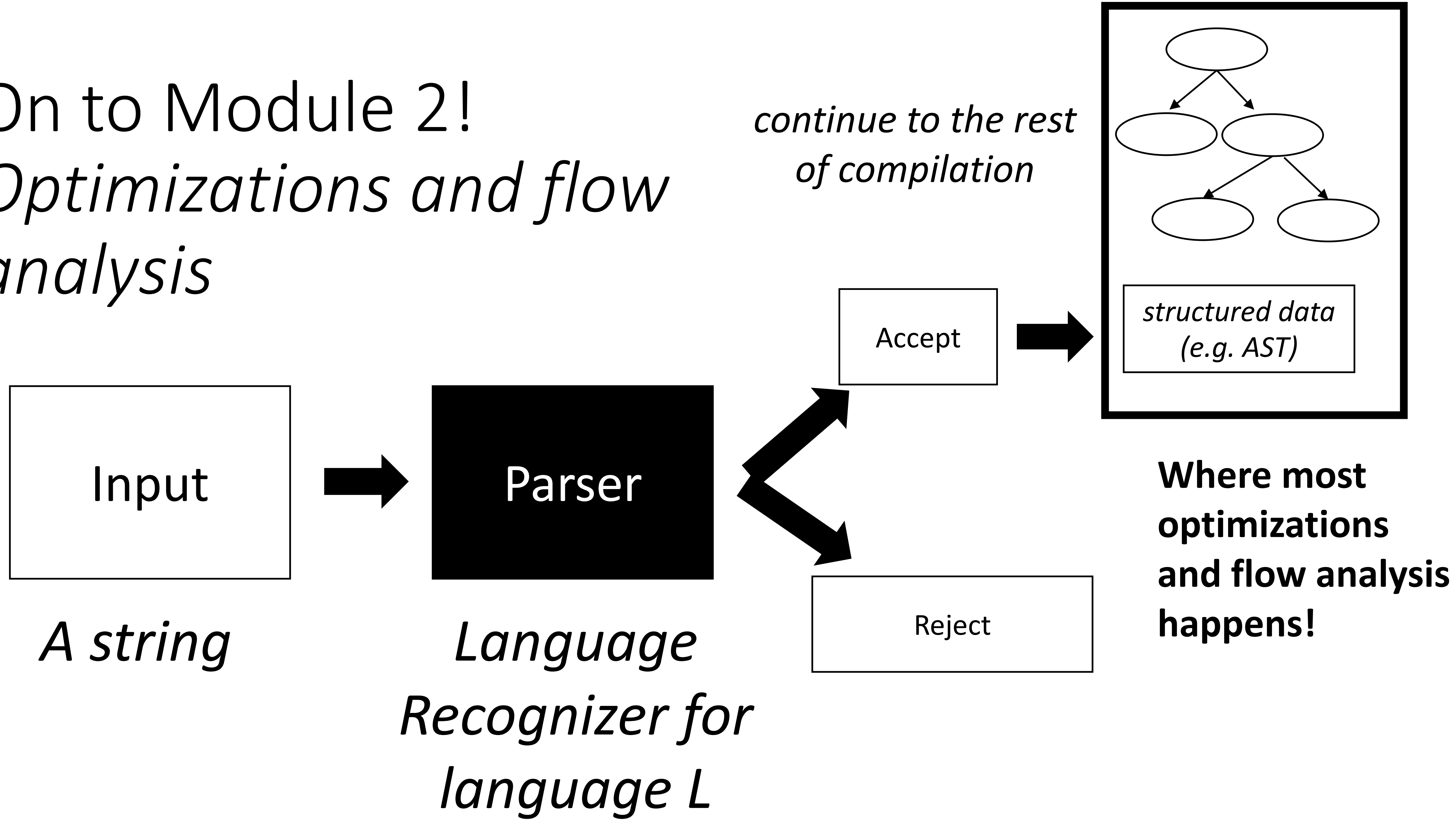
Code overview

On to Module 2!
Optimizations and flow analysis





On to Module 2!
Optimizations and flow analysis



Intermediate representations

- 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

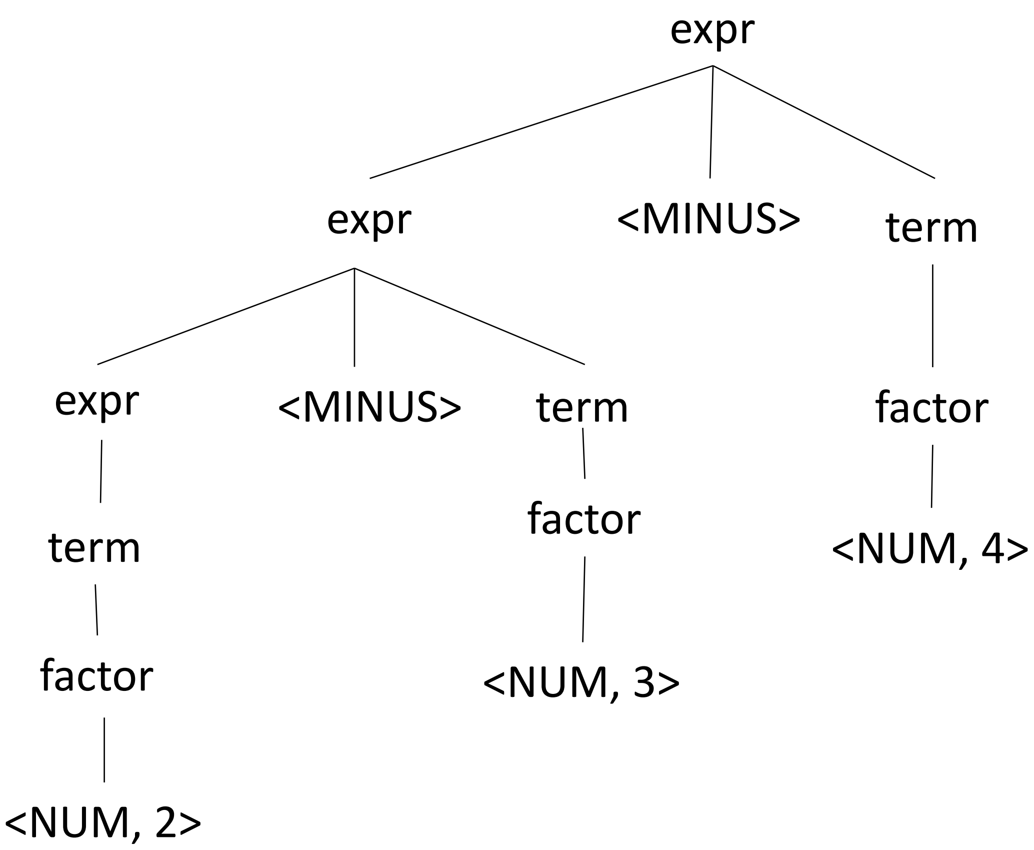
- Trees
  - Abstract syntax trees
  - Good for instruction scheduling
- Textual
  - 3 address code, e.g. LLVM IR
  - Good for local value numberings, removing redundant expressions
- Graphs
  - Control flow graphs
  - Good for data flow analysis

Abstract Syntax Trees

- Remember the expression parse tree

`input: 2-3-4`

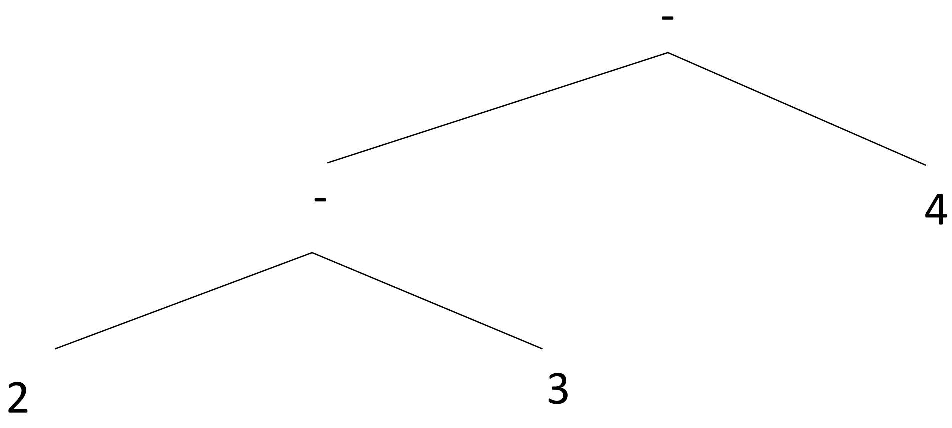
Operator	Name	Productions
+,-	Expr	: Expr + Term | Expr - Term | Term
*,/	Term	: Term * Pow : Term / Pow | Pow
^	Pow	: Factor ^ Pow | Factor
()	Factor	: (Expr) | NUM



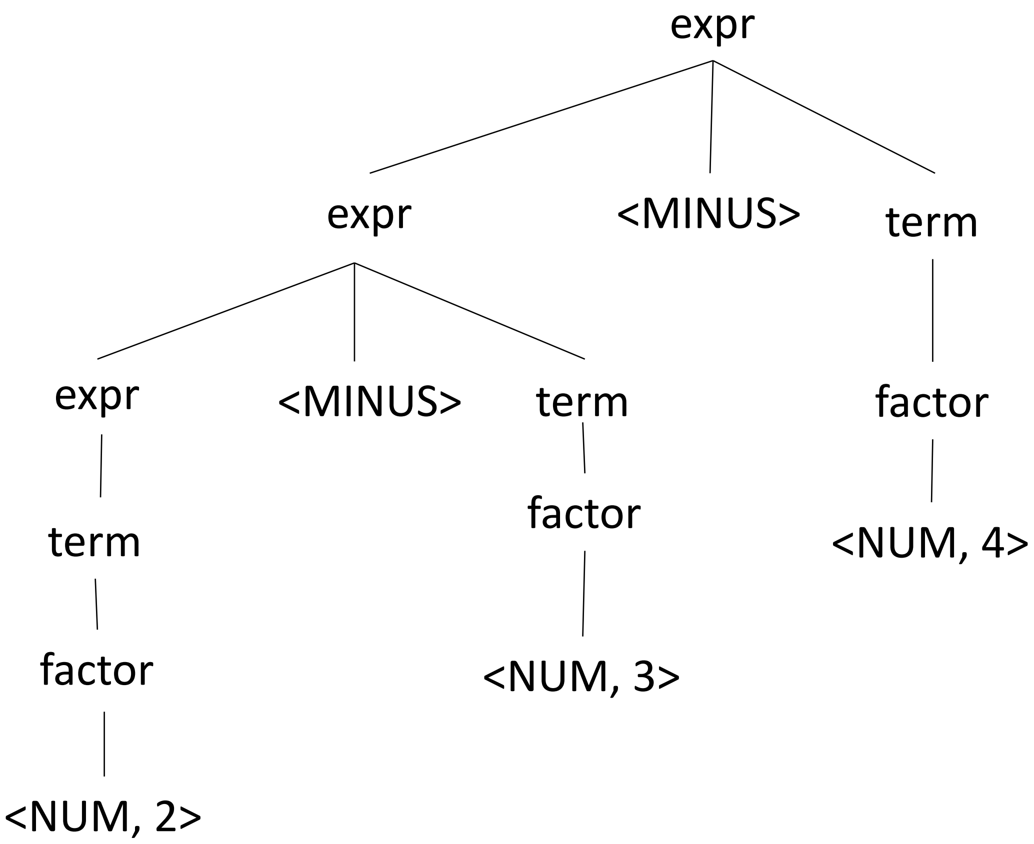
Abstract Syntax Trees

- Remember the expression parse tree

`input: 2-3-4`



Much more compact!

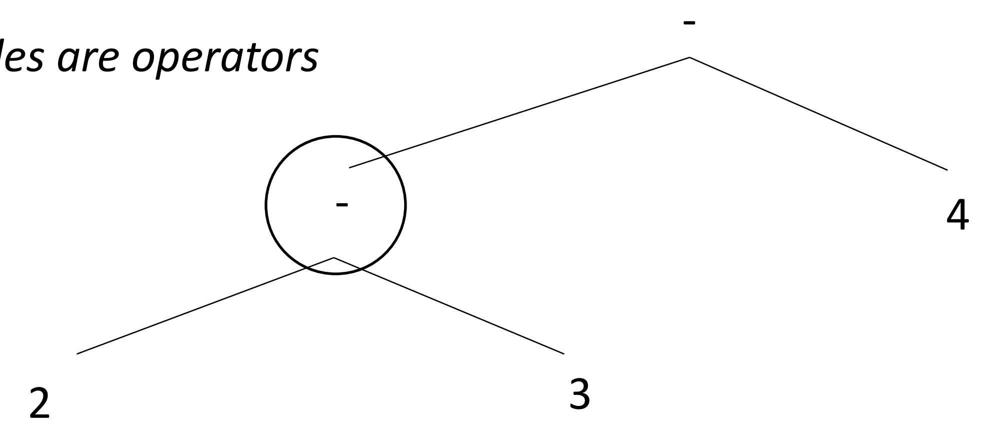


Abstract Syntax Trees

`input: 2-3-4`

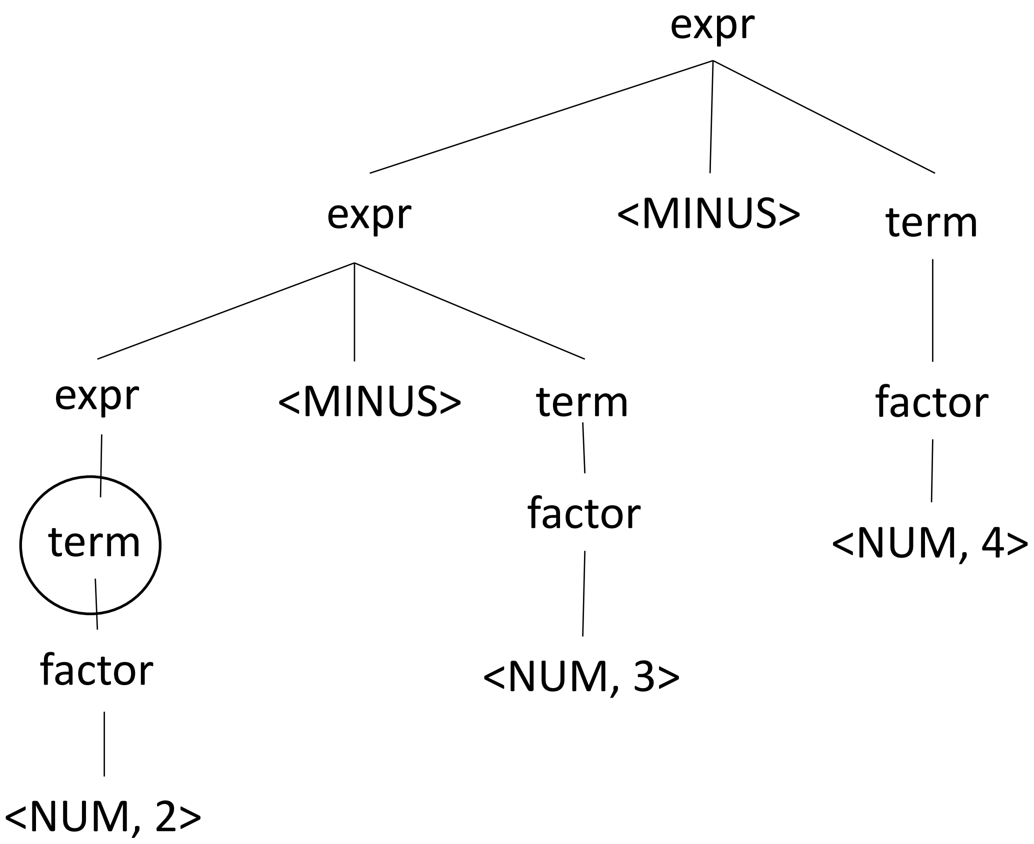
- Remember the expression parse tree

nodes are operators



Much more compact!

nodes are production rules



Abstract Syntax Trees

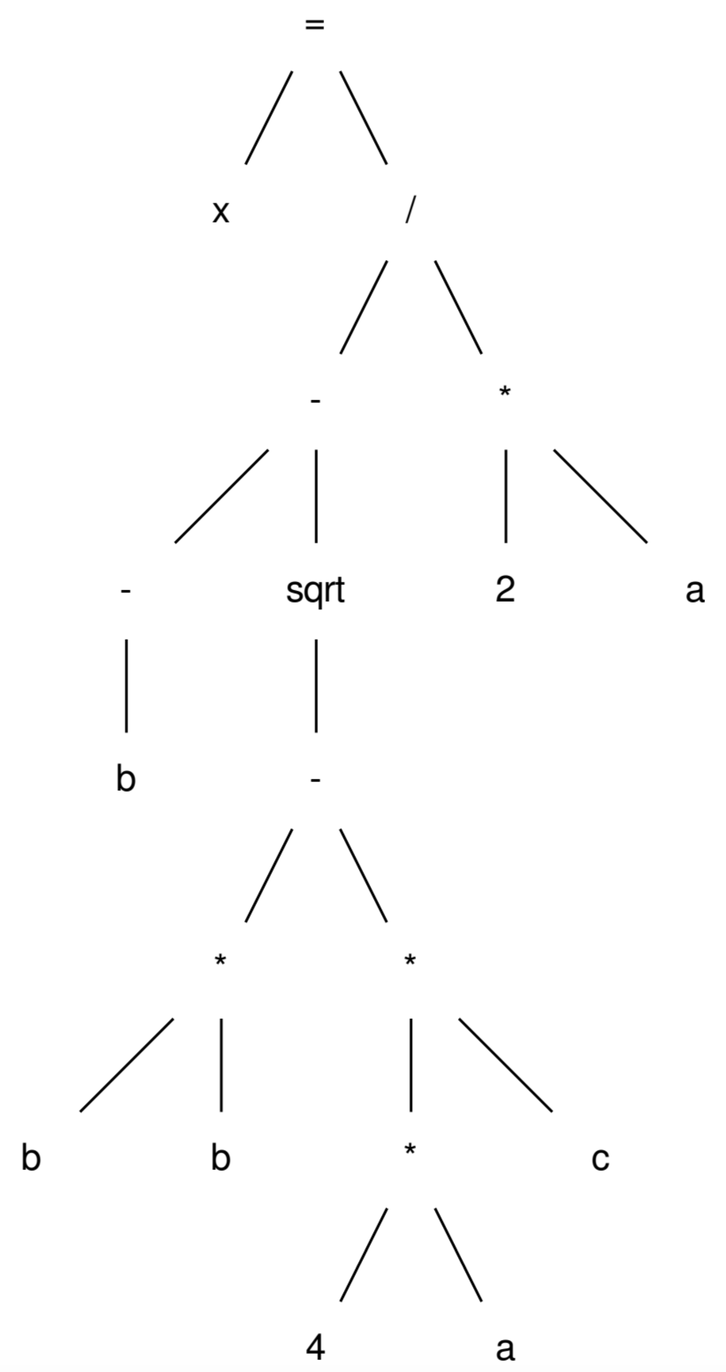
- Easier to see bigger trees, e.g. quadratic formula:

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

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)



3 Address IR

Powerful IR

- Close to machine instructions
- Uses virtual registers

- All instructions are of the form:

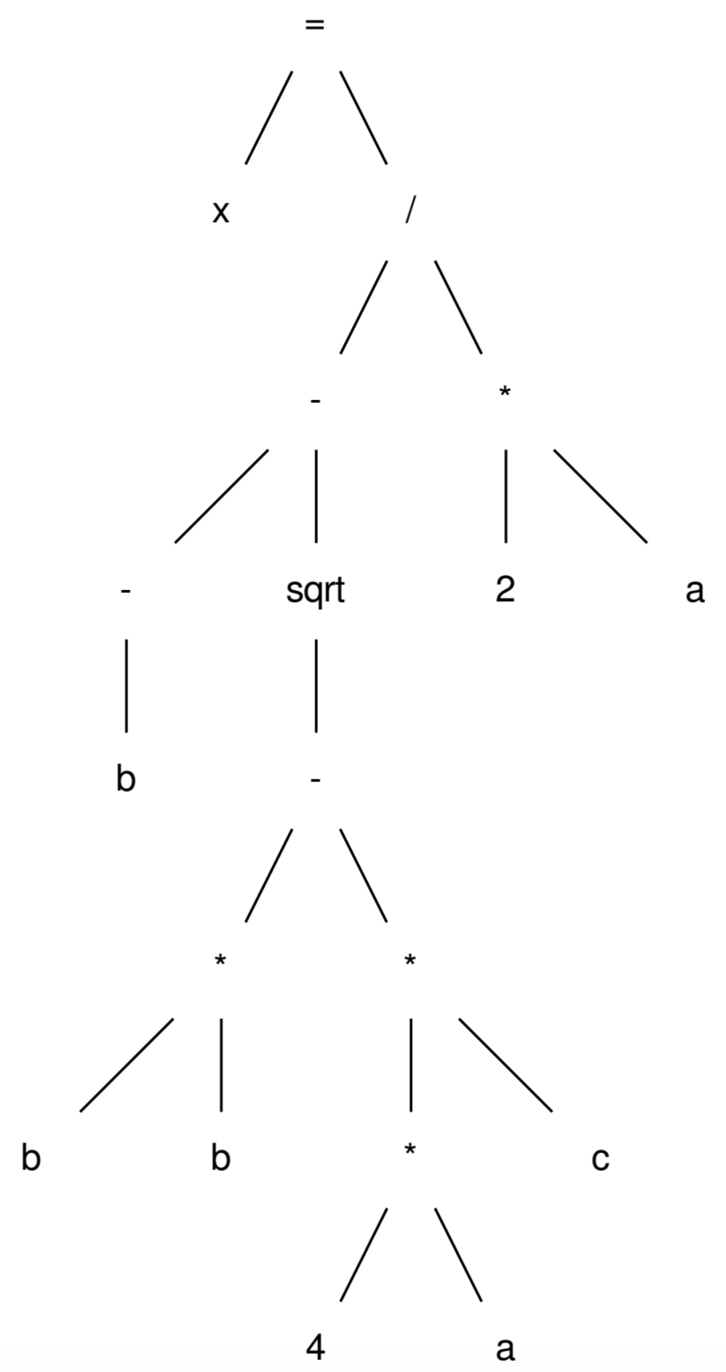
```
result = op1 OP op2
```

Special instructions take 1 op

```
result = load(op1)
```

Convert this code to 3 address code

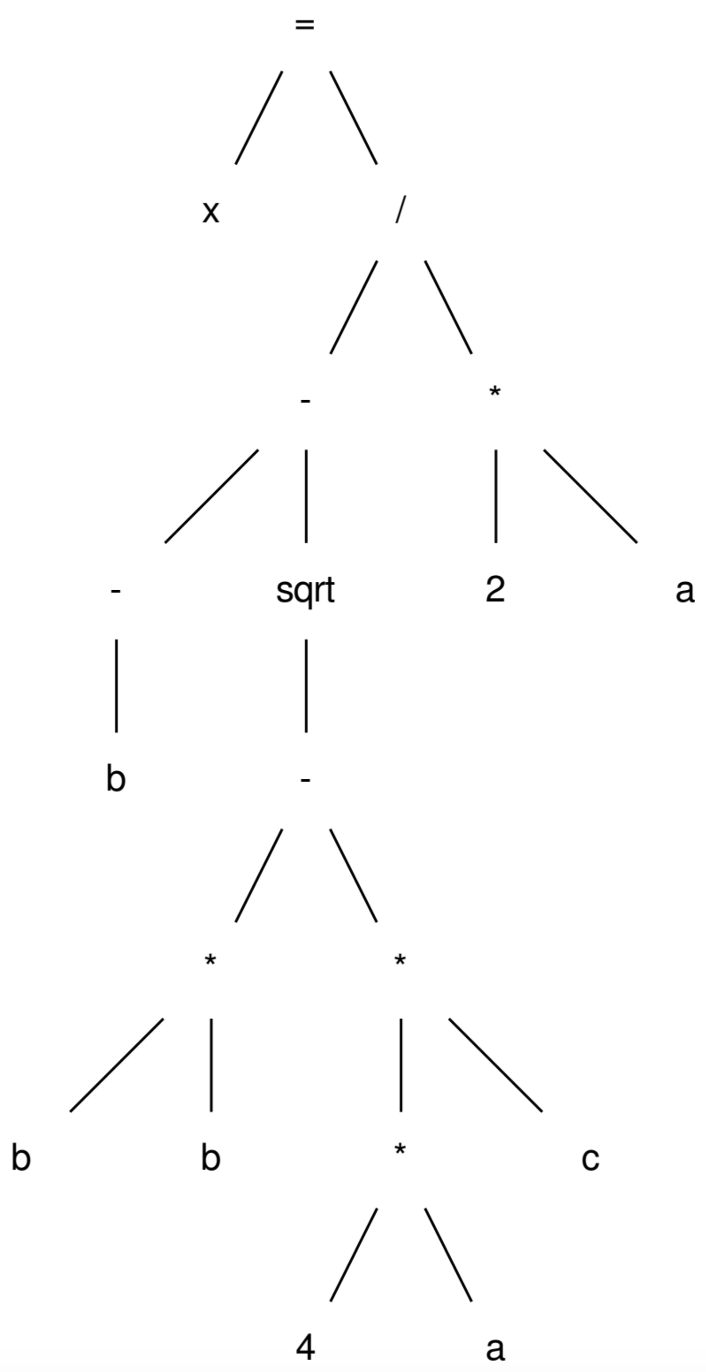
post-order traversal, creating virtual registers



Convert this code to 3 address code

post-order traversal, creating virtual registers

```
r0 = neg(b);
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?

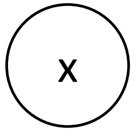
We can make a data-dependency graph (DDG)

```
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)

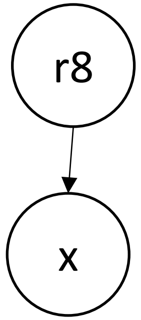
```
r0 = neg(b);
r1 = b * b;
r2 = 4 * a;
r3 = r2 * c;
r4 = r1 – r3;
r5 = sqrt(r4);
r6 = r0 – r5;
r7 = 2 * a;
r8 = r6 / r7;
x  = r8;
```



What now?

We can make a data-dependency graph (DDG)

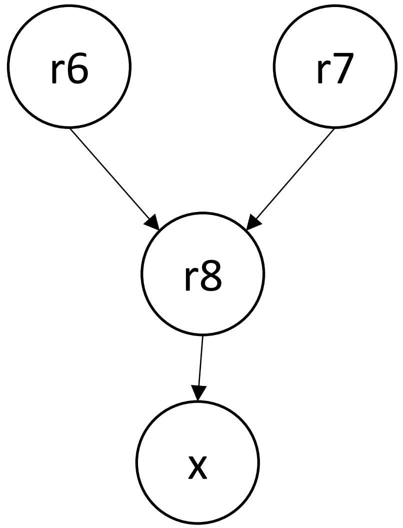
```
r0 = neg(b);
r1 = b * b;
r2 = 4 * a;
r3 = r2 * c;
r4 = r1 – r3;
r5 = sqrt(r4);
r6 = r0 – r5;
r7 = 2 * a;
r8 = r6 / r7;
x  = r8;
```



What now?

We can make a data-dependency graph (DDG)

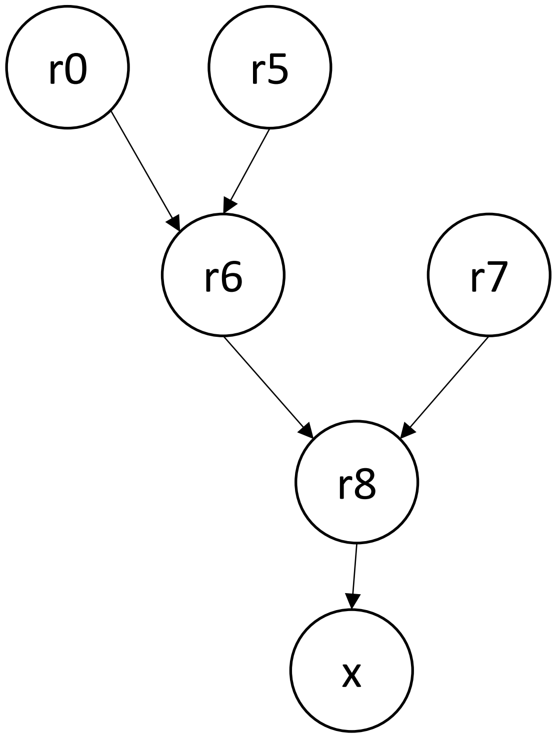
```
r0 = neg(b);
r1 = b * b;
r2 = 4 * a;
r3 = r2 * c;
r4 = r1 – r3;
r5 = sqrt(r4);
r6 = r0 – r5;
r7 = 2 * a;
r8 = r6 / r7;
x  = r8;
```



What now?

We can make a data-dependency graph (DDG)

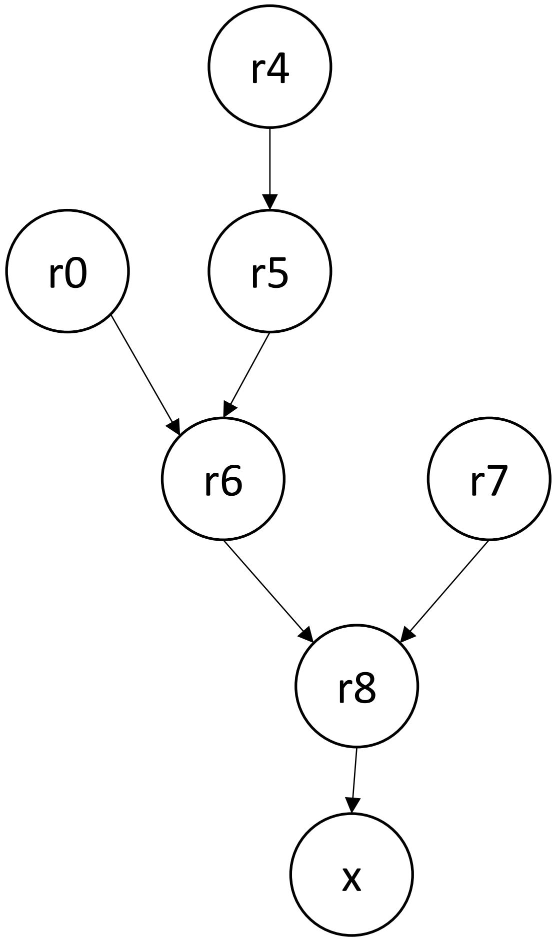
```
r0 = neg(b);
r1 = b * b;
r2 = 4 * a;
r3 = r2 * c;
r4 = r1 – r3;
r5 = sqrt(r4);
r6 = r0 – r5;
r7 = 2 * a;
r8 = r6 / r7;
x  = r8;
```



What now?

We can make a data-dependency graph (DDG)

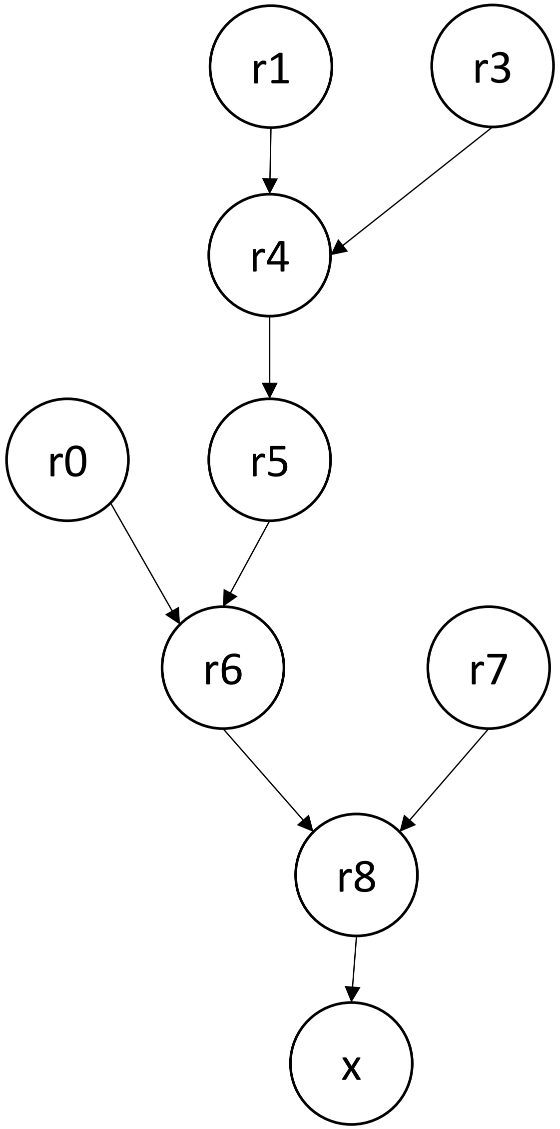
```
r0 = neg(b);
r1 = b * b;
r2 = 4 * a;
r3 = r2 * c;
r4 = r1 – r3;
r5 = sqrt(r4);
r6 = r0 – r5;
r7 = 2 * a;
r8 = r6 / r7;
x  = r8;
```



What now?

We can make a data-dependency graph (DDG)

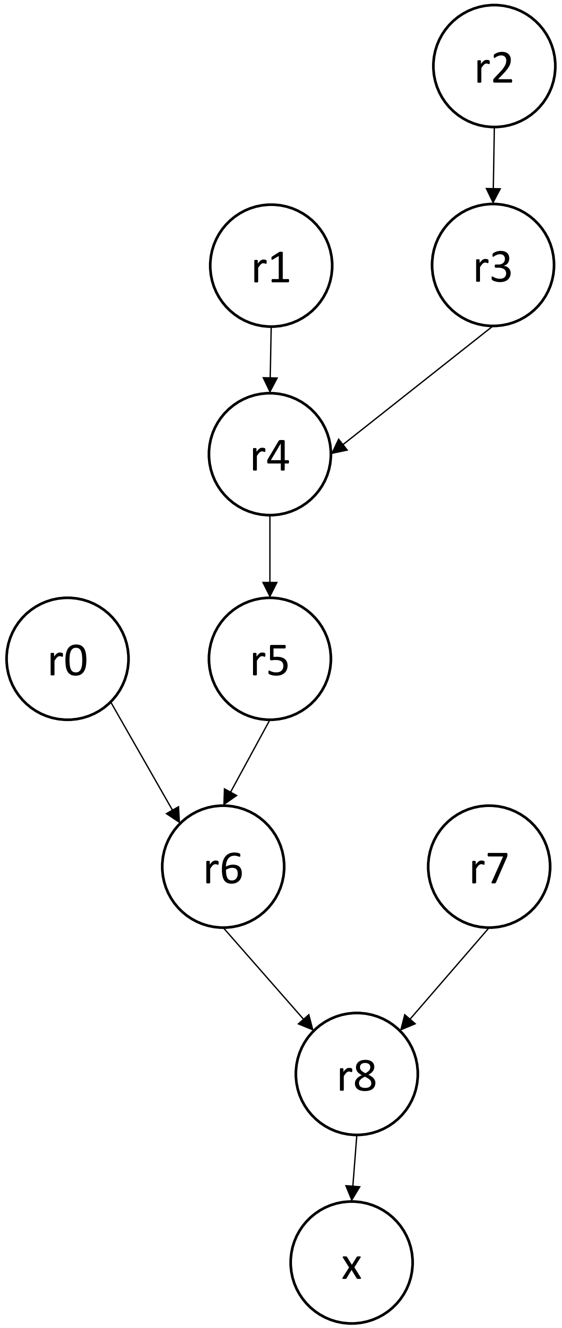
```
r0 = neg(b);
r1 = b * b;
r2 = 4 * a;
r3 = r2 * c;
r4 = r1 – r3;
r5 = sqrt(r4);
r6 = r0 – r5;
r7 = 2 * a;
r8 = r6 / r7;
x  = r8;
```



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;
```

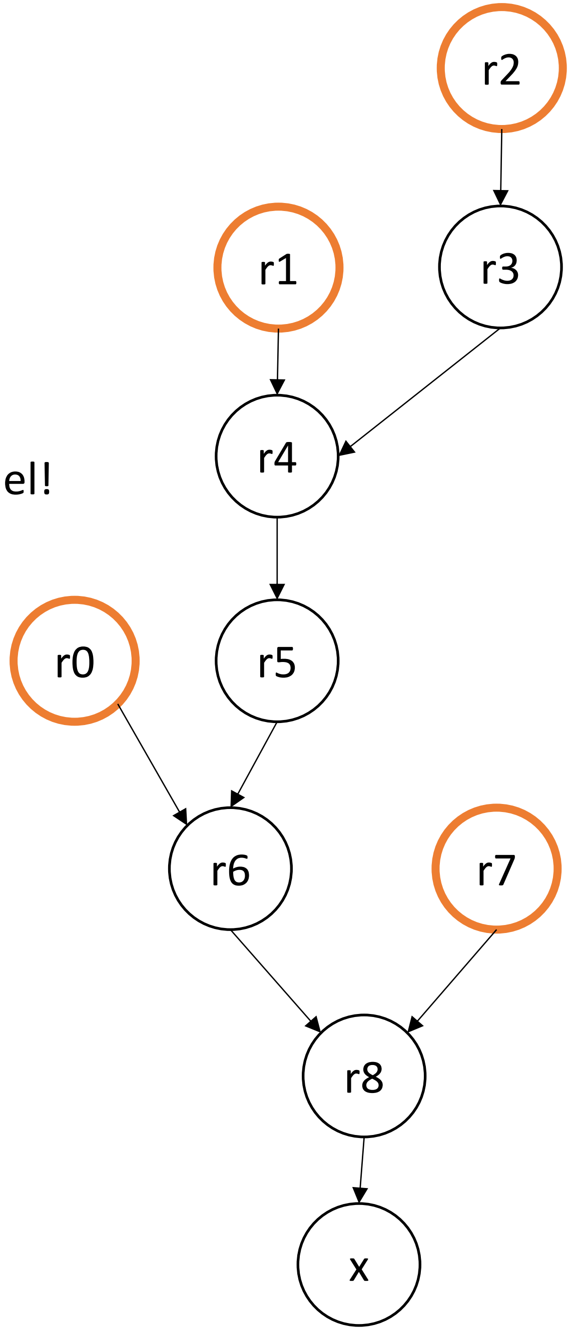


What now?

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!



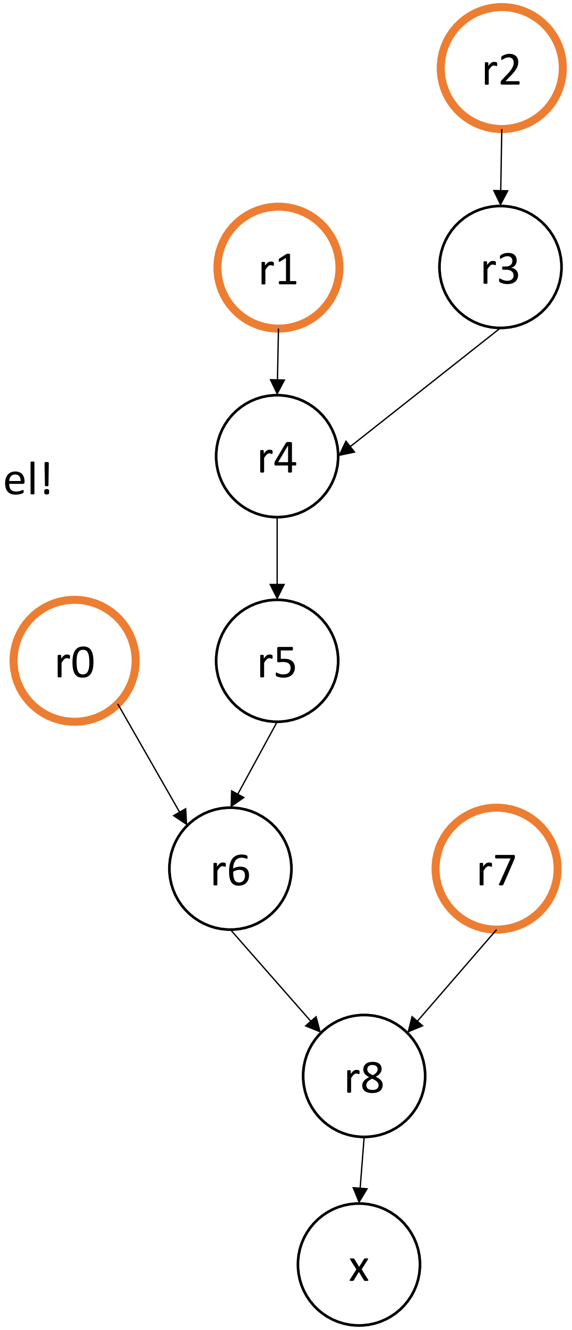
What now?

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!

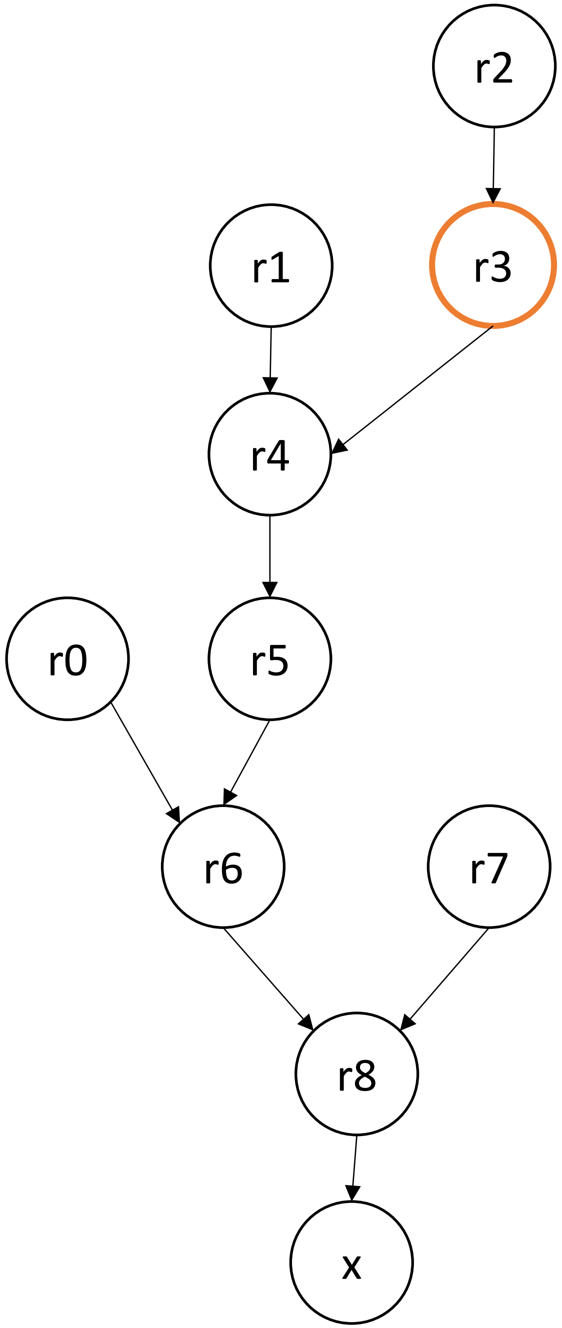


What now?

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?



Power of IRs

- We’ve shown 3 different IRs:
  - AST
  - 3 address code
  - DDG

- Converting between them allowed different types of code reasoning

Next lecture

- More optimizations for each IR

- AST:
  - Tree balancing for more instruction-level scheduling

- Three address code:
  - Local value numbering for redundant expression pruning

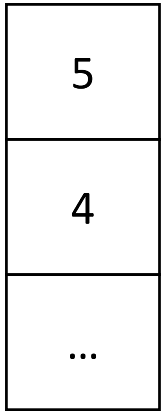
- Control flow graphs:
  - to expand the range of analysis

Bonus: From AST to a stack virtual machine:

A common IR for java bytecode and web assembly.

Easy to implement (can be done completely at the parser)

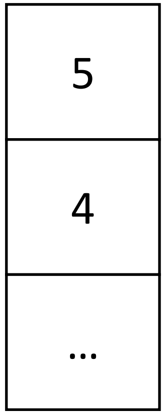
hard to analyze…



unlimited virtual stack

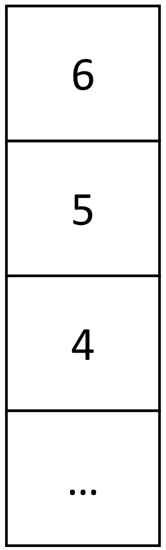
Bonus: From AST to a stack virtual machine:

push 6



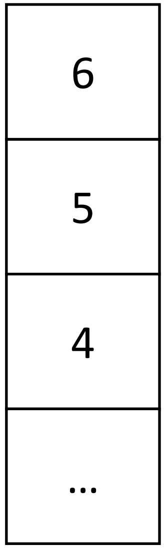
Bonus: From AST to a stack virtual machine:

push 6



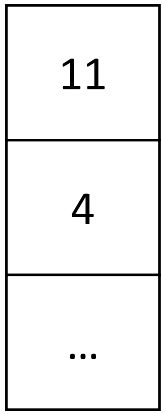
Bonus: From AST to a stack virtual machine:

add



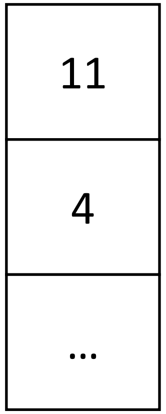
Bonus: From AST to a stack virtual machine:

add



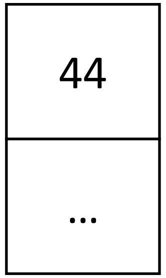
Bonus: From AST to a stack virtual machine:

mult

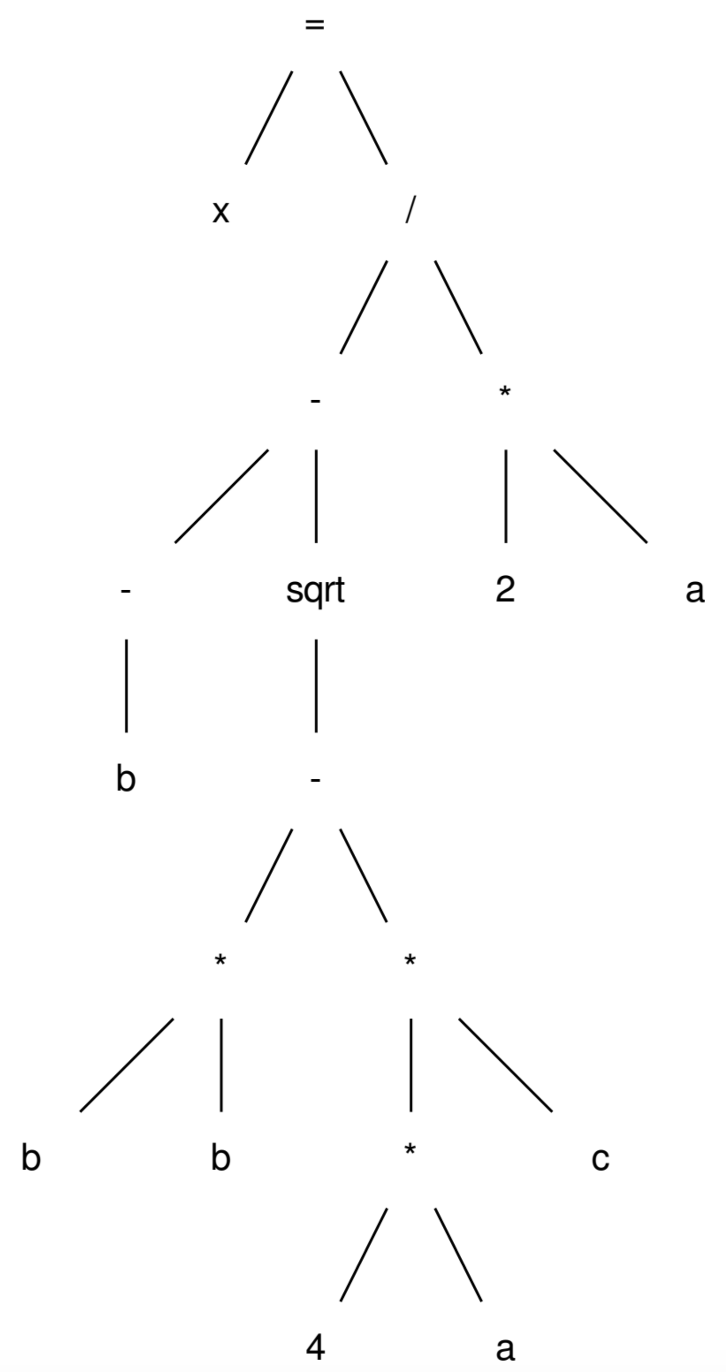


Bonus: From AST to a stack virtual machine:

mult



Bonus: From AST to a stack virtual machine:



Bonus: From AST to a stack virtual machine:

```
push b;
negate;
push b;
push b;
mult;
push 4;
push a;
mult;
push c
mult;
minus;
sqrt;
push 2;
push a;
mult;
divide;
assign x;
```