

THEORY OF COMPILATION

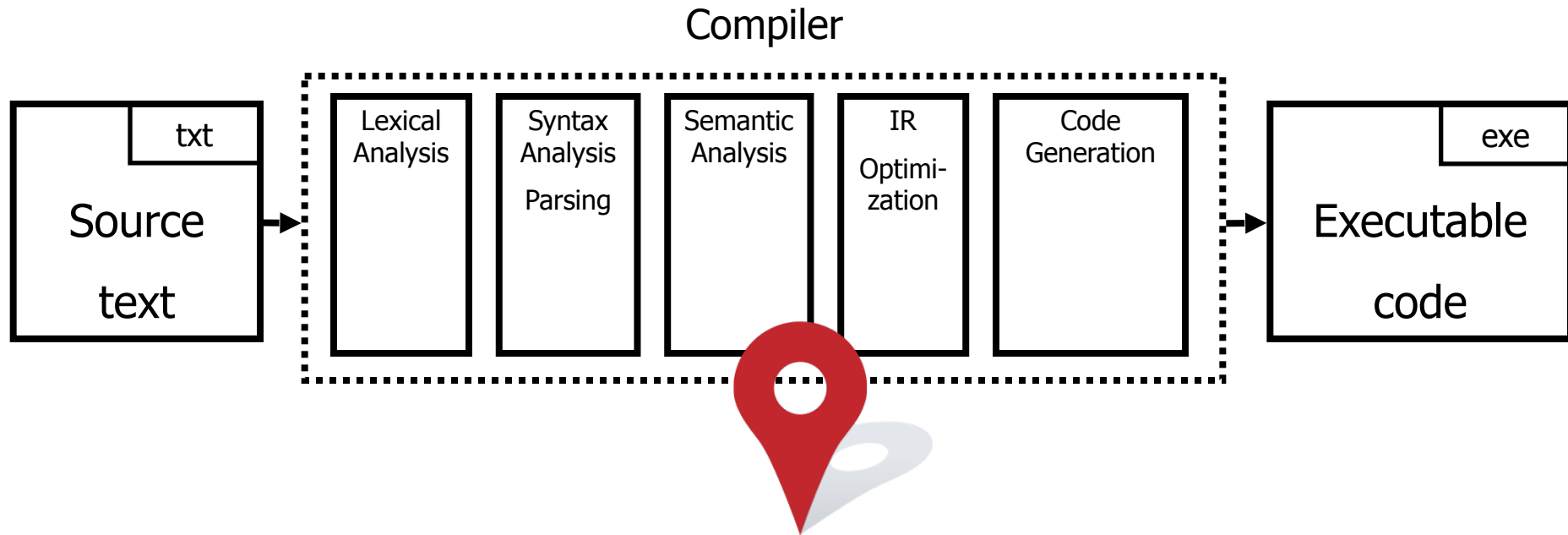
LECTURE 11

~~COMPLICATED~~
STATIC
ANALYSIS

Totally Awesome



You are here



Plan for Today

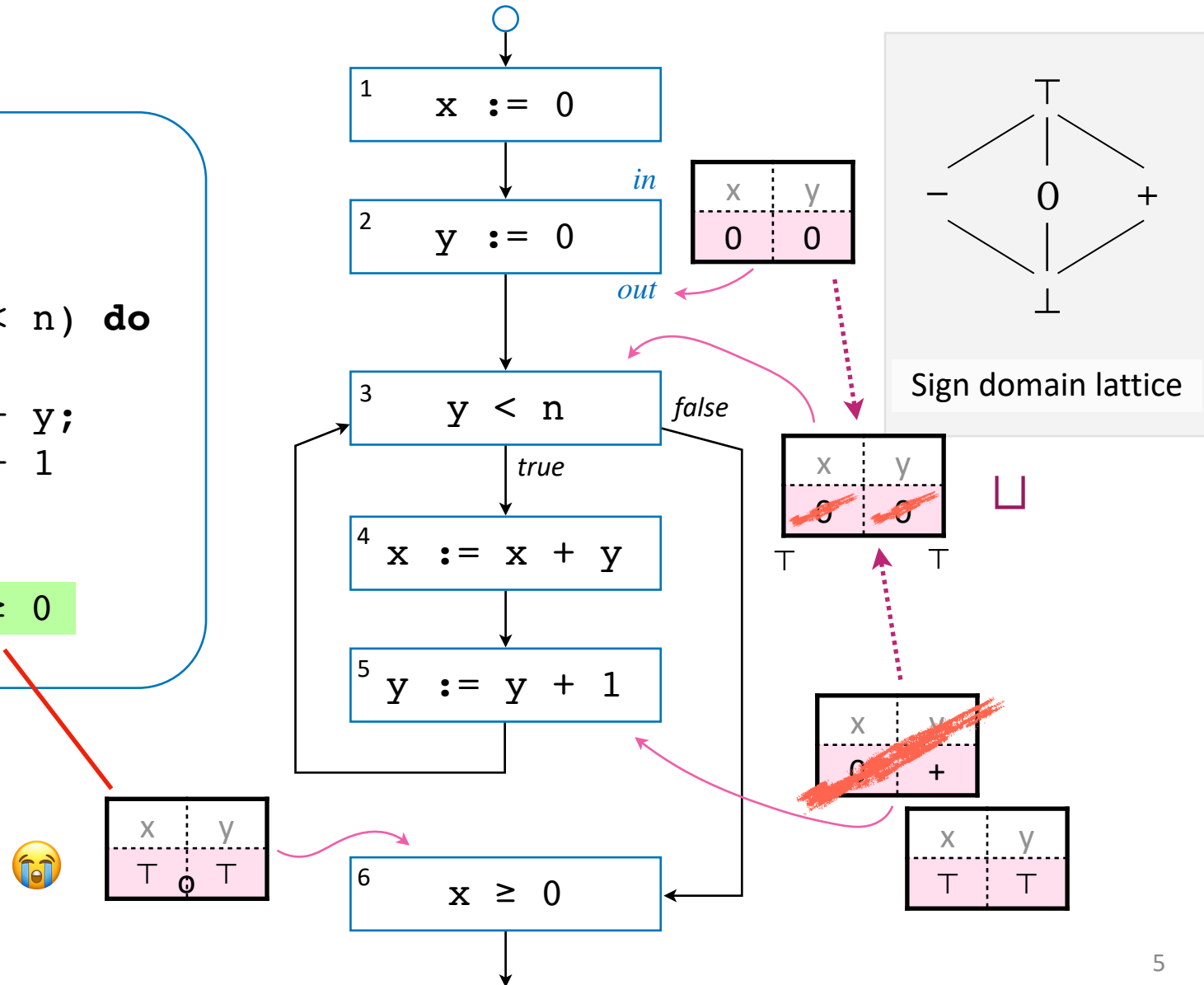
- Refresh our memory of Data Flow Analysis
- Another example: pointer analysis
- Combining domains

Data Flow Analysis (ugh, again?!)

```
1: x := 0;  
2: y := 0;  
  
3: while (y < n) do  
  (  
4:   x := x + y;  
5:   y := y + 1  
  )  
  
6: z := sqrt(x)
```

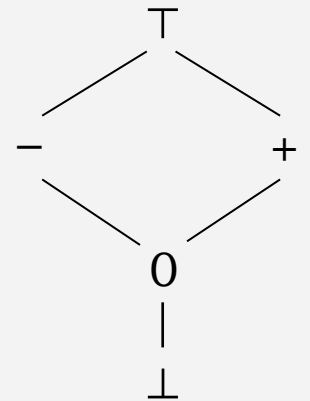
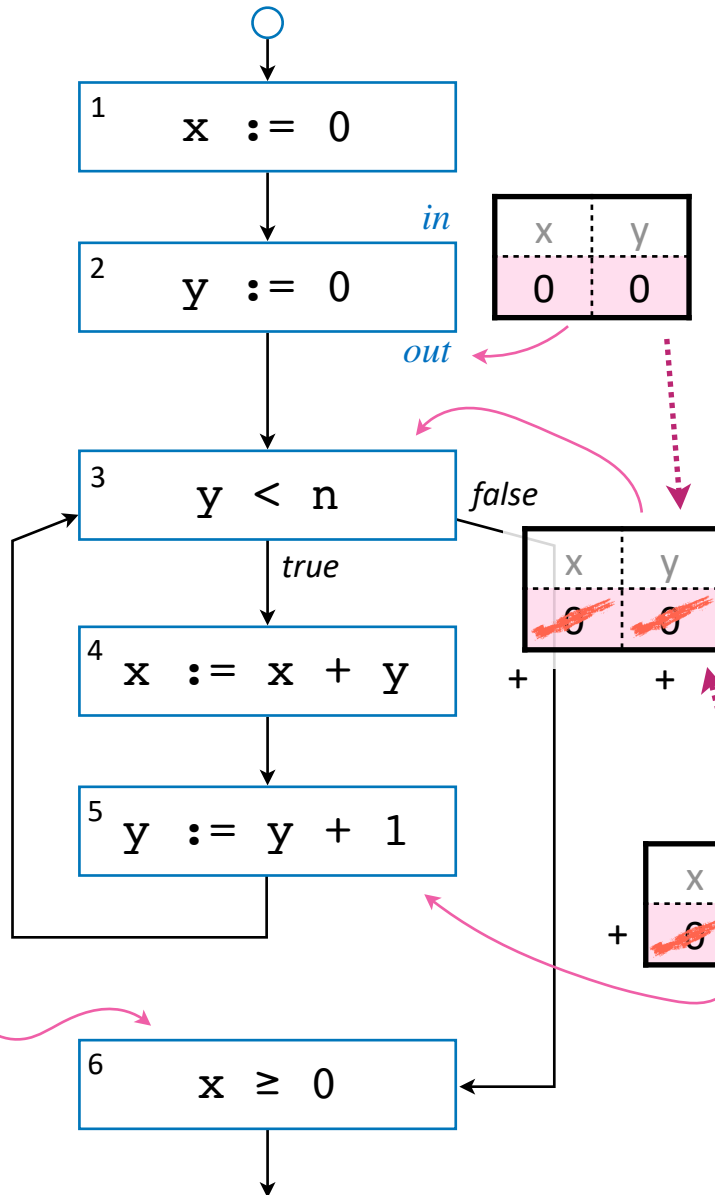
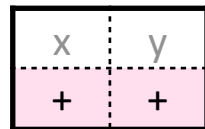
Data Flow Analysis (ugh, again?!)

```
1: x := 0;
2: y := 0;
3: while (y < n) do
4:     x := x + y;
5:     y := y + 1
6: assert x ≥ 0
```



Data Flow Analysis (ugh, again?!)

```
1: x := 0;  
2: y := 0;  
  
3: while (y < n) do  
  (  
4:   x := x + y;  
5:   y := y + 1  
  )  
  
6: assert x ≥ 0
```

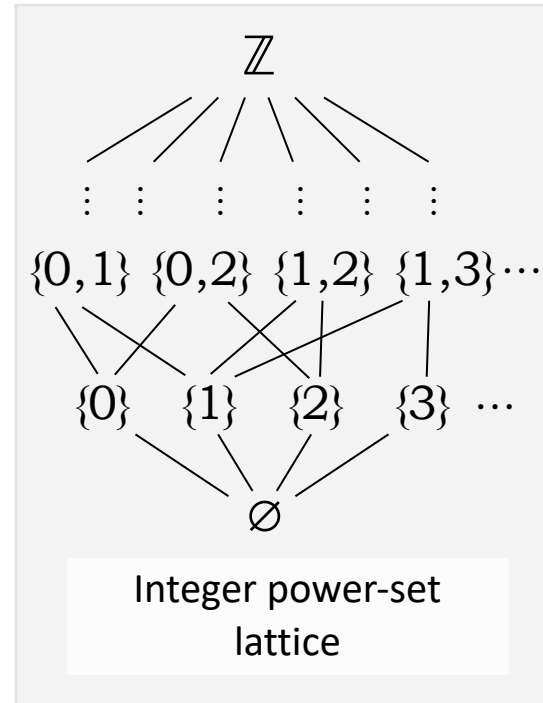
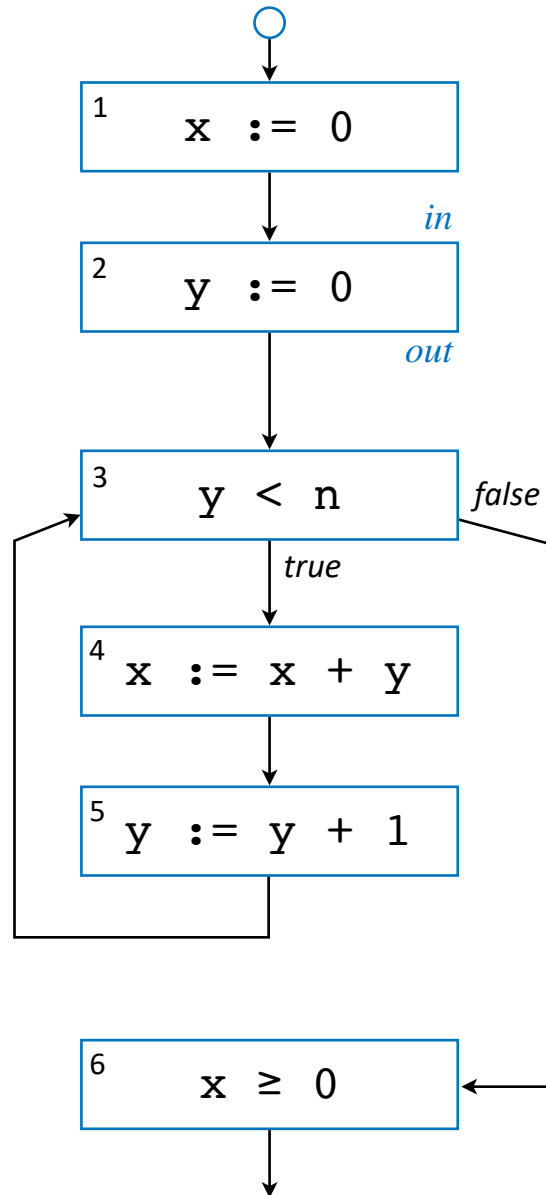


Sign domain lattice

+ means ≥ 0
- means ≤ 0

Data Flow Analysis (ugh, again?!)

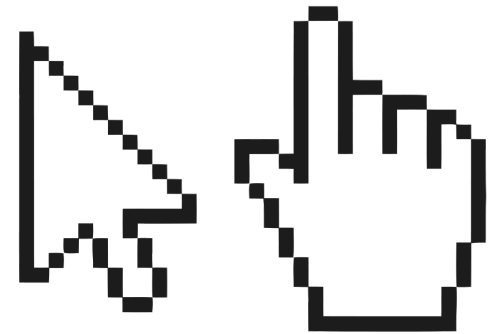
```
1: x := 0;  
2: y := 0;  
  
3: while (y < n) do  
  (  
4:   x := x + y;  
5:   y := y + 1  
  )  
  
6: assert x ≥ 0
```



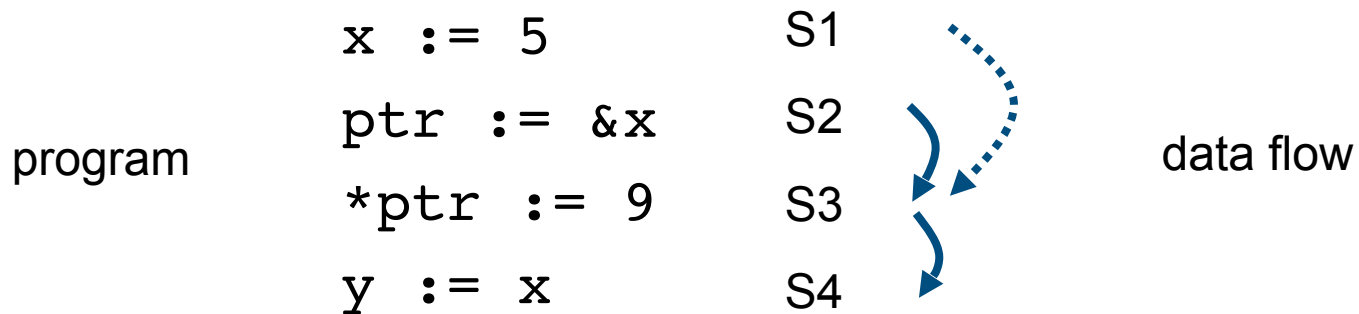
$$L = \langle \mathcal{P}(\mathbb{Z}), \subseteq \rangle$$

$$a \sqcup b = \hat{a} \cup b$$

Pointer Analysis



Simple Example



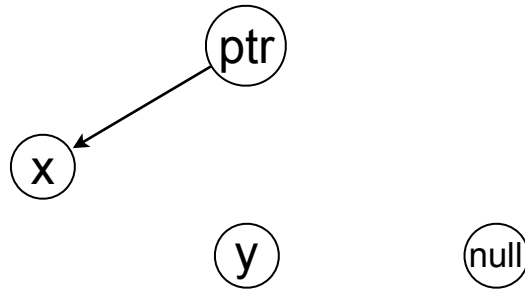
- What are the data dependencies in this program?
- Problem: just looking at variable names will not give you the correct information
 - ▶ After statement S2, program names “x” and “*ptr” are both expressions that refer to the same memory location.
 - ▶ We say that *ptr points-to x* after statement S2.
- In a C-like language that has pointers, we must know the *points-to relation* to be able to determine dependencies correctly

Program Model

- The programming language/IR has instructions that deal with pointers:
 - ▶ address: $x := \&y$
 - ▶ copy: $x := y$ (regular assignment)
 - ▶ load: $x := *y$
 - ▶ store: $*x := y$
- For now: no heap, no function calls. Allowed types are \mathbb{Z} , \mathbb{Z}^* (pointer to number), \mathbb{Z}^{**} , ...

Points-to Relation

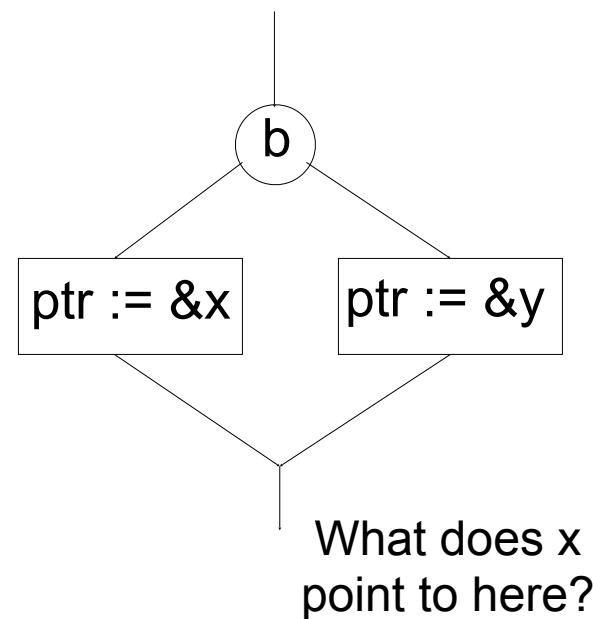
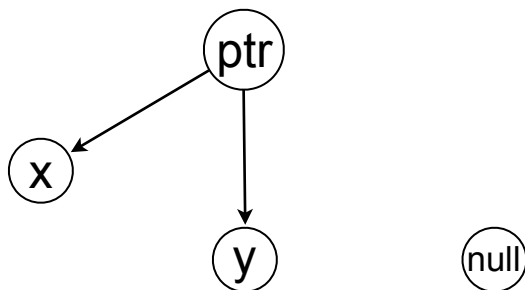
- Directed graph:
 - ▶ Nodes are program variables (+ special node for null)
 - ▶ Edge (a,b) — variable a points-to variable b



- Of course, points-to is different at different program locations

Points-to Relation

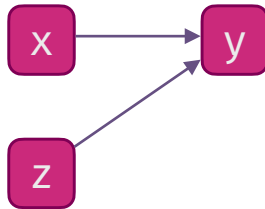
- Directed graph:
 - ▶ Nodes are program variables (+ special node for null)
 - ▶ Edge (a,b) — variable a points-to variable b



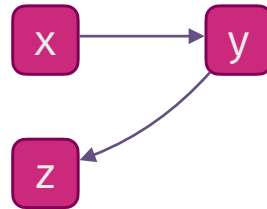
- Out-degree may be > 1
if there are multiple paths

Points-to Relation

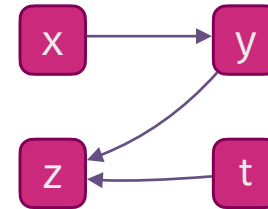
$x := \&y$
 $z := \&y$



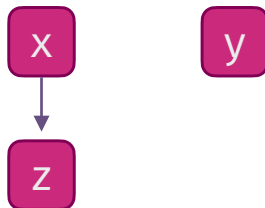
$x := \&y$
 $y := \&z$



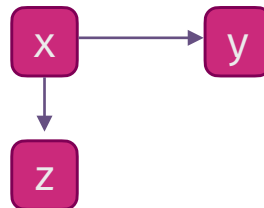
$x := \&y$
 $t := \&z$
 $*x := t$



$x := \&y$
 $x := \&z$



if (...)
 $x := \&y$
else
 $x := \&z$



Points-to Relation

- As an abstract domain (a lattice):
 - ▶ Nodes are fixed per program
 - ◇ can think of it as a power-set domain of possible edges
 - ▶ \perp is a graph with no edges
 - ▶ \sqsubseteq is the subgraph relation (edge subset)
 - ▶ \sqcup is obtain by union of edges

$$\text{pt}(u) = \{v \mid (u,v) \in E\}$$

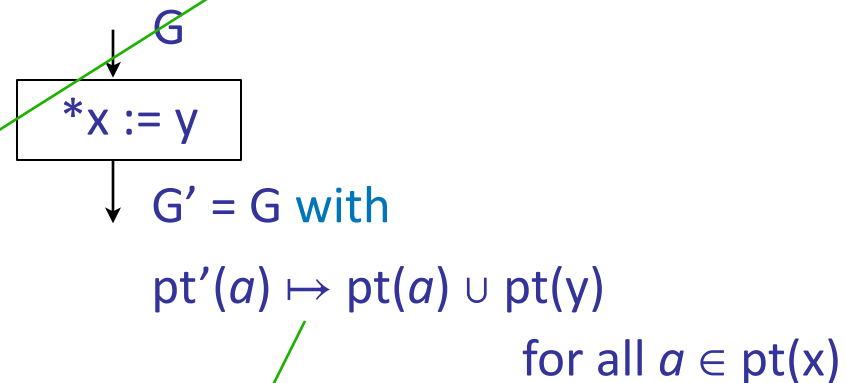
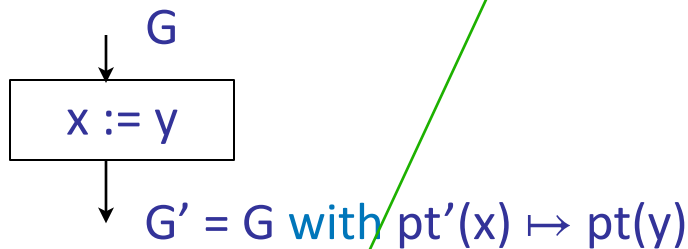
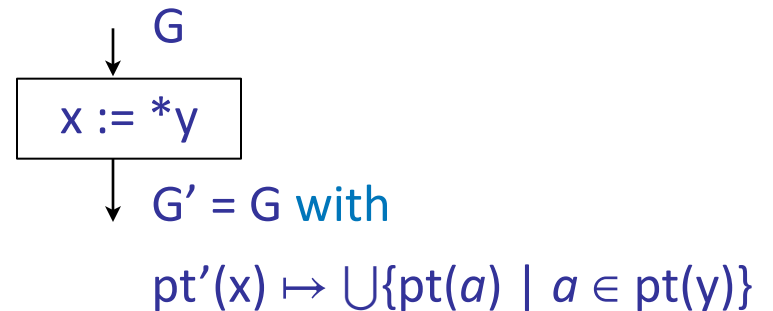
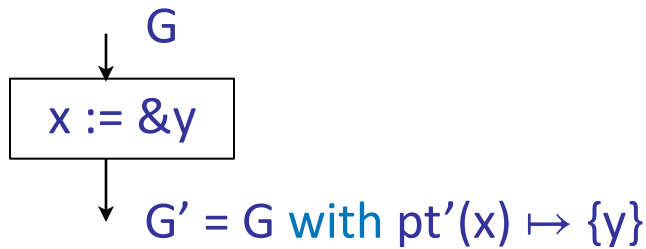
Points-to Analysis: Two Flavors

- Flow Sensitive (we'll be doing this)
 - ▶ Based on abstract interpretation / dataflow
 - ▶ Can examine behavior at different locations
- Flow Insensitive (we won't be doing this)
 - ▶ Computes a single points-to relation for the entire program
 - ▶ Works by generating constraints and solving them
 - ▶ (Andersen's algorithm / Steengards algorithm)

Points-to: Abstract Semantics

in = G , out = G'

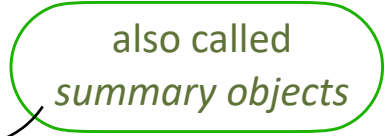
$pt(u) = \{v \mid (u,v) \in E\}$



strong updates

weak update (why?)

Dynamic Allocation

- What to do with $x := \text{new } Z[\dots]$?
 - Program can create an unbounded number of objects
 - Need some static naming scheme for dynamically allocated objects
- AbsObj — set of *abstract objects* 

Single name for the entire heap $\text{AbsObj} = \{ H \}$

Type-based

$$\text{AbsObj} = \{ H_T \mid T \text{ is a type in the program} \}$$

Allocation-site based

$$\text{AbsObj} = \{ H_\ell \mid \ell \in \text{Lab s.t. } \ell : p := \text{new } T \}$$

Dynamic Allocation: Semantics

- Basically: model every “new” as “address of”

```
1: p := new Z[5];  
2: q := new Z[5];  
3: if (p = q) then  
4:     z := p  
5: else  
6:     z := q
```



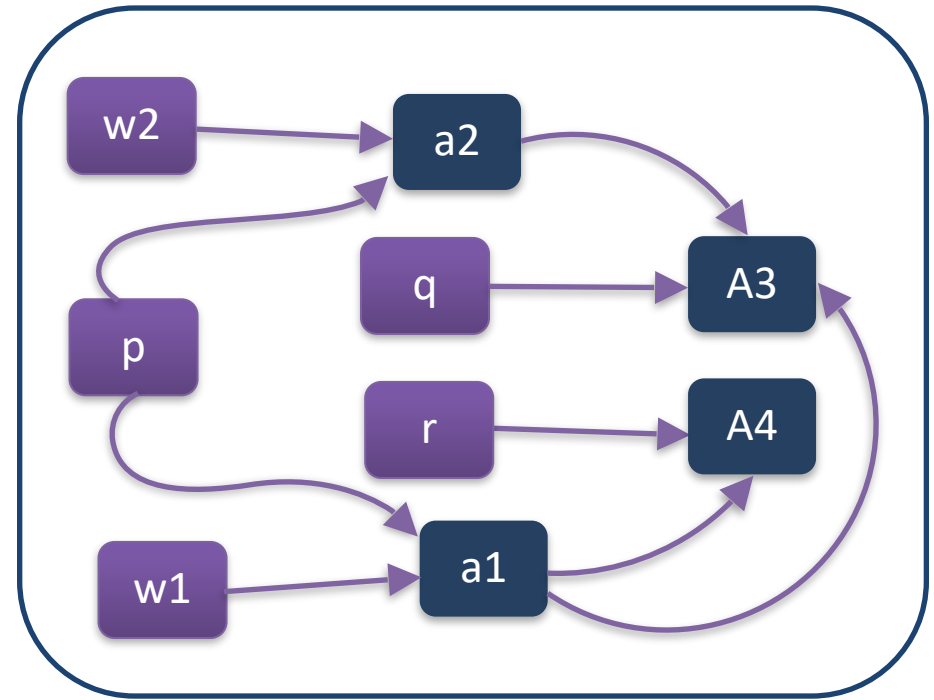
```
1: p := &A1;  
2: q := &A2;  
3: if (p = q) then  
4:     z := p  
5: else  
6:     z := q
```

- *Conservative*: may result in spurious “may point to” entries; but “must not point to” results are always sound.

Points-to Analysis: Example

```
1: w1 := &a1;  
2: w2 := &a2;  
3: q := new Z[5];  
4: r := new Z[5];  
5: *w1 := r;  
6: if (...) then  
7:   p := w1  
8: else  
9:   p := w2;  
10: *p := q
```

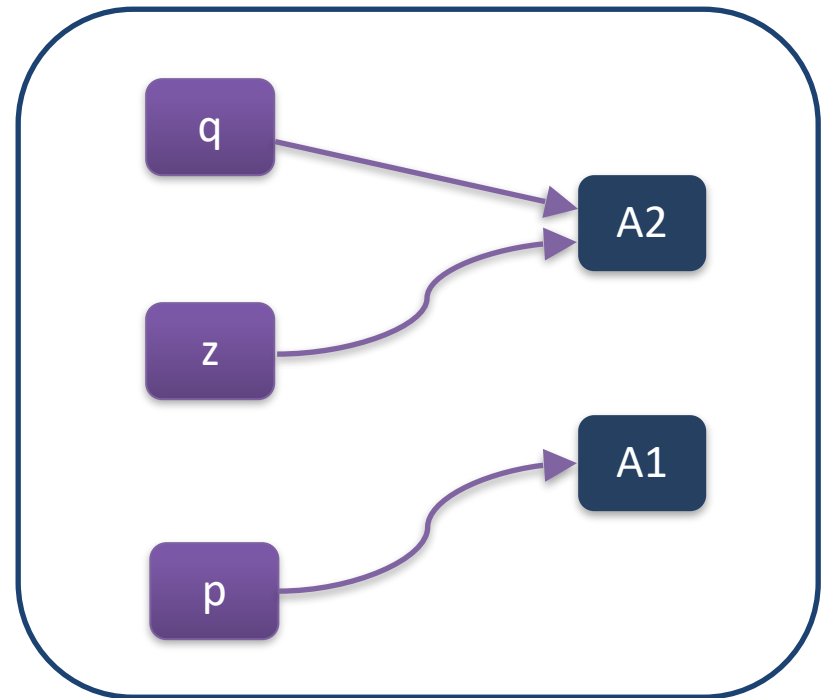
$a1, a2, q, r : Z^*$
 $w1, w2, p : Z^{**}$



Aliasing Analysis

Derived from result of point-to analysis

```
1: p := new Z[5];  
2: q := new Z[5];  
3: if (p = q) then  
4:     z := p  
5: else  
6:     z := q
```



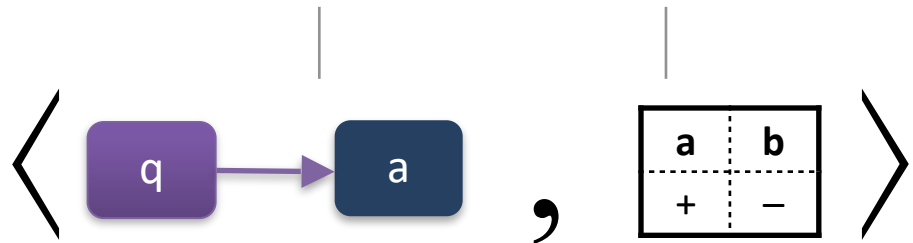
z and p **may not** alias
q and p **may not** alias
z and q **may** alias

Example: Pointers + Sign

```
1: a := 3;  
2: b := -3;  
3: q := &a;  
4: *q := *q + 1  
5: assert a + b > 0  
6: assert b < 0
```

Abstract Domain:

Points-to × Signs



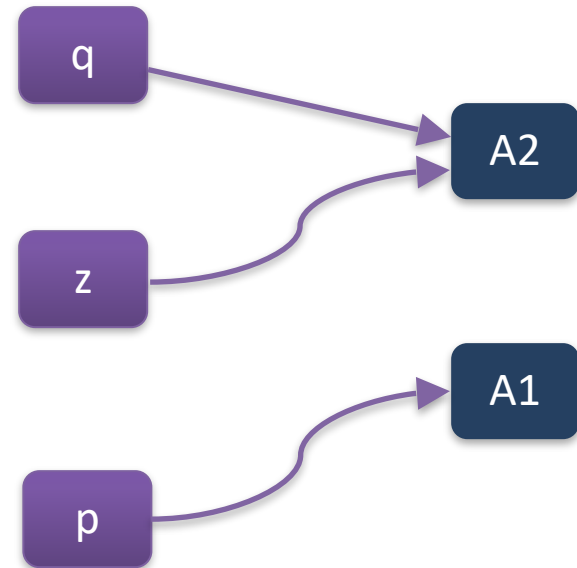
Example: Aliasing + Available Expressions

```
1: a := *q + 4 * c;
```

```
2: b := 0;
```

```
3: *p := 1;
```

```
4: b := *q + 4 * c
      a
```



Optimization is valid:
`p` and `q` are **not** aliased

Static Program Analysis

- Can automatically **prove** interesting properties
 - ▶ absence of null pointer dereferences, numerical assertions, termination, absence of data races, information flow, ...
- Nice combination of **math** and **system** building
 - ▶ combines program semantics, data structures, discrete math, logic, parallelism, decision procedures, ...

What we learned about

Static Program Analysis

- No need to run the program!
 - ▶ No concrete input needed!
 - ▶ Properties are guaranteed to hold for *any* input and *any* execution!!

Exam

11/4/2024

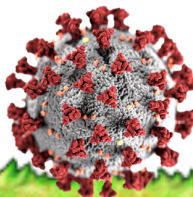
13:00 – 16:00

- ★ 20% Compiler Phases
- ★ 30% Syntax, Semantics, Code generation
- ★ 20% Optimizations
- ★ 30% Static Analysis

Hybrid Exam!

45% multiple choice
questions

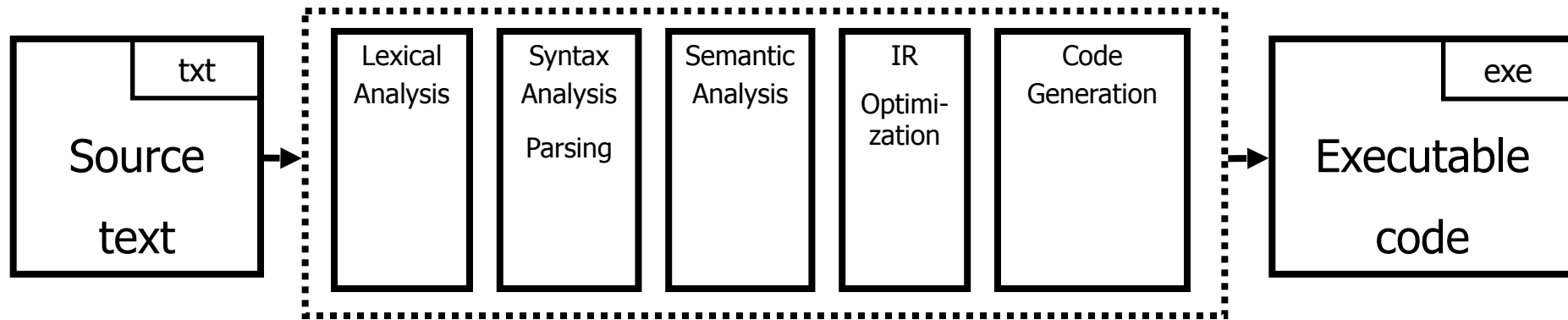
55% open questions



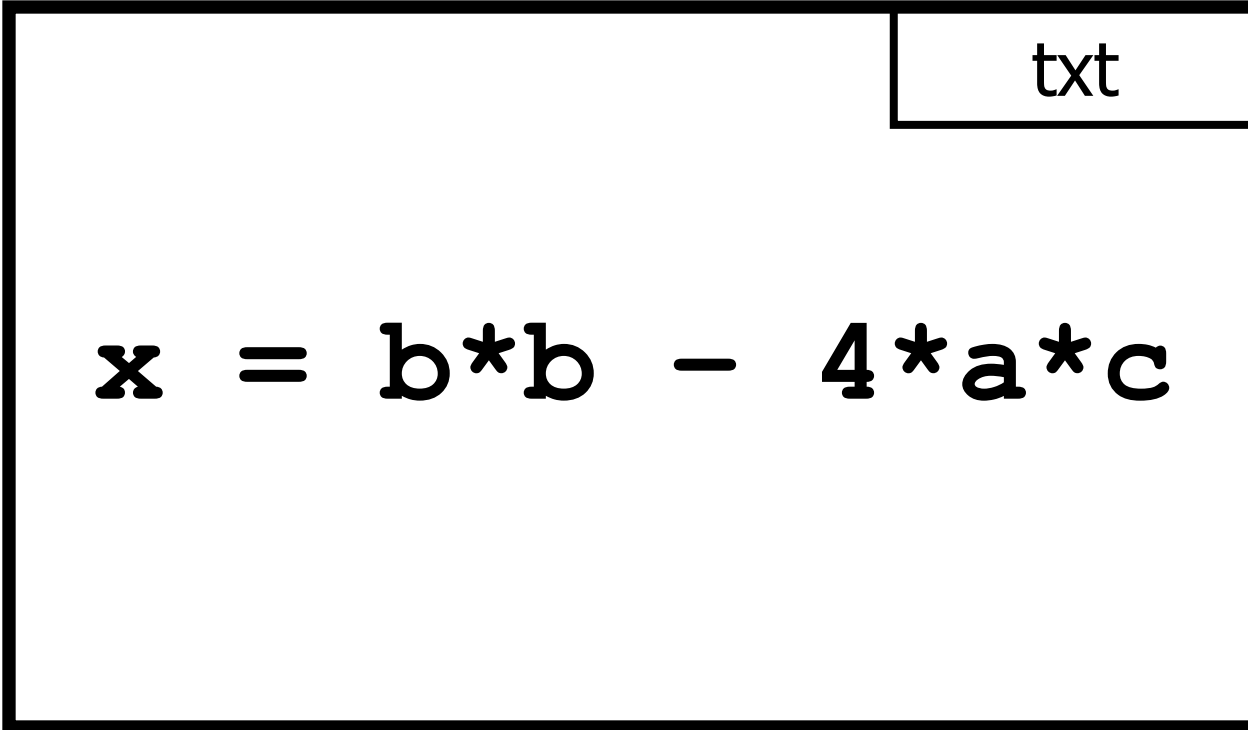
Recap



Compiler

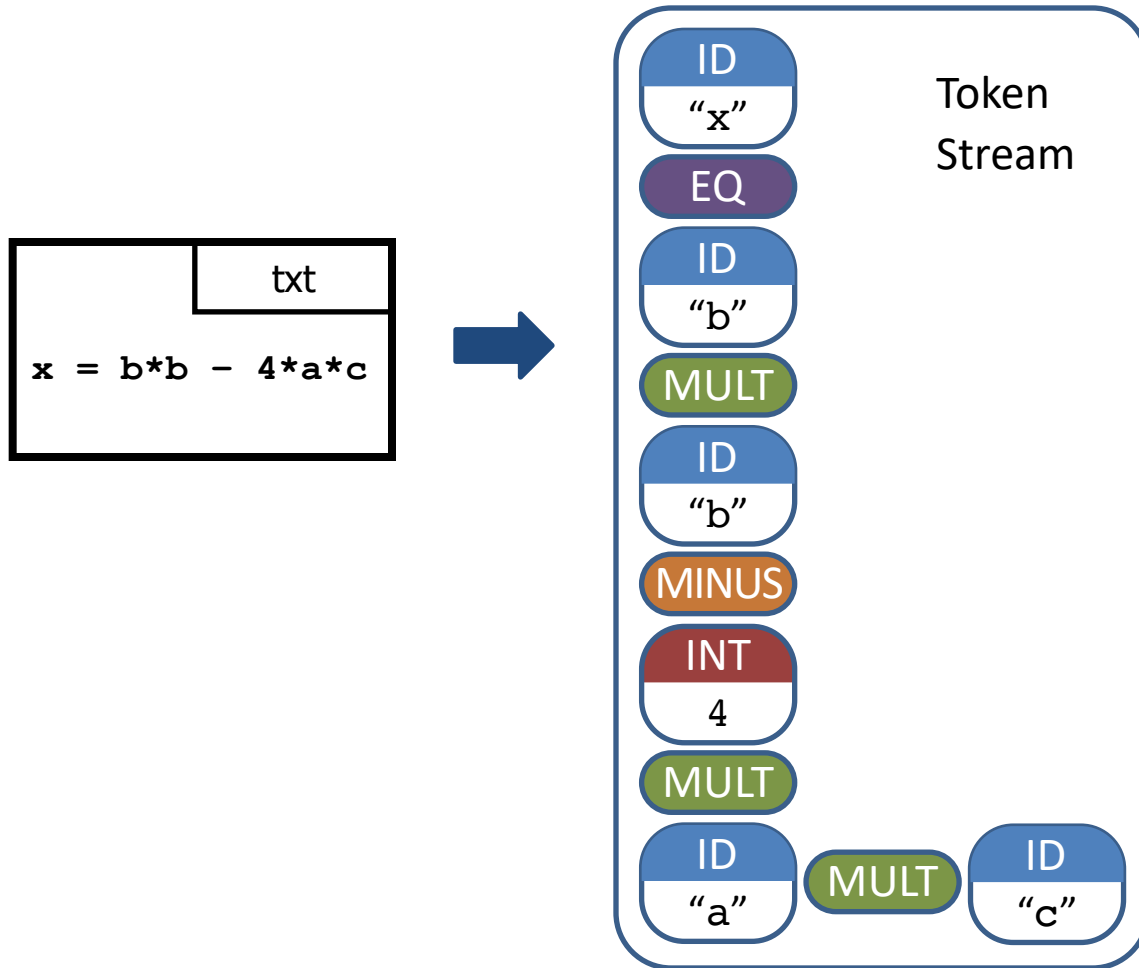


Recap



```
txt  
  
x = b*b - 4*a*c
```

Recap



```
graph TD; E1((E)) --- E2((E)); E1 --- MINUS; E1 --- T1((T)); E2 --- T2((T)); T2 --- T3((T)); T2 --- MULT1[MULT]; T2 --- F1((F)); T3 --- T4((T)); T3 --- F2((F)); T4 --- F3((F)); F3 --- ID1[ID]; ID1 --- b1["b"]; F2 --- ID2[ID]; ID2 --- b2["b"]; F1 --- ID3[ID]; ID3 --- c["c"]; MINUS --- T5((T)); T5 --- E3((E)); T5 --- MULT2[MULT]; T5 --- F4((F)); E3 --- T6((T)); T6 --- T7((T)); T6 --- MULT3[MULT]; T6 --- F5((F)); T7 --- T8((T)); T7 --- F6((F)); T8 --- F7((F)); F7 --- ID4[ID]; ID4 --- 4["4"]; F6 --- ID5[ID]; ID5 --- a["a"]; F5 --- ID6[ID]; ID6 --- c2["c"];
```

$$\begin{aligned} E &\rightarrow E \text{ (PLUS) } T \\ E &\rightarrow E \text{ (MINUS) } T \\ T &\rightarrow T \text{ (MULT) } F \\ T &\rightarrow T \text{ (DIV) } F \\ F &\rightarrow (\text{ID}) \\ F &\rightarrow (\text{INT}) \end{aligned}$$

Token Stream

Lexical Analysis

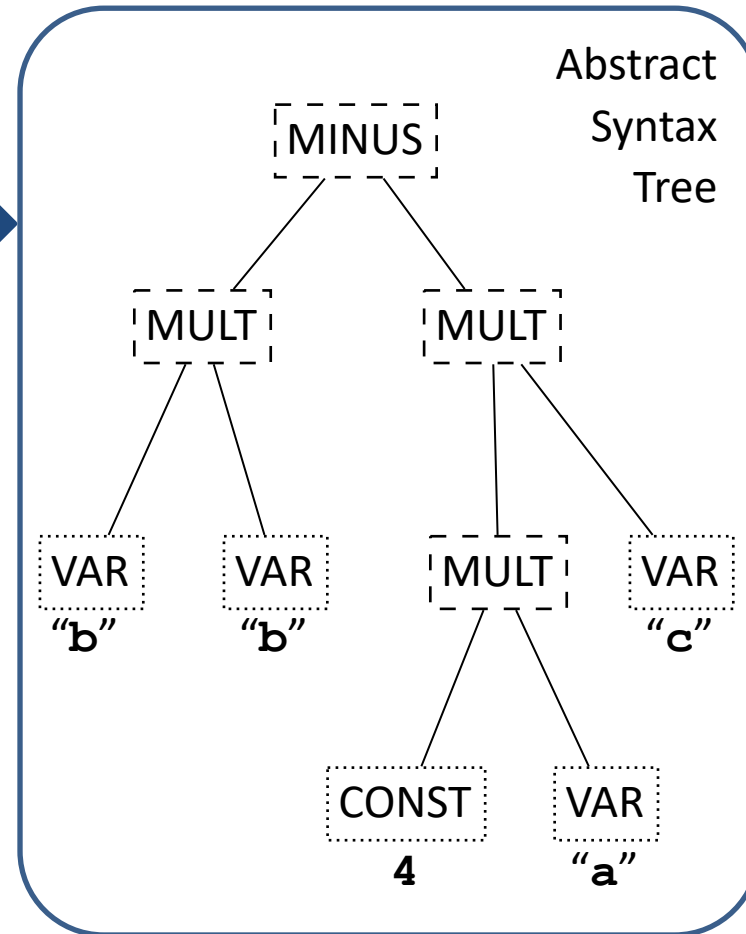
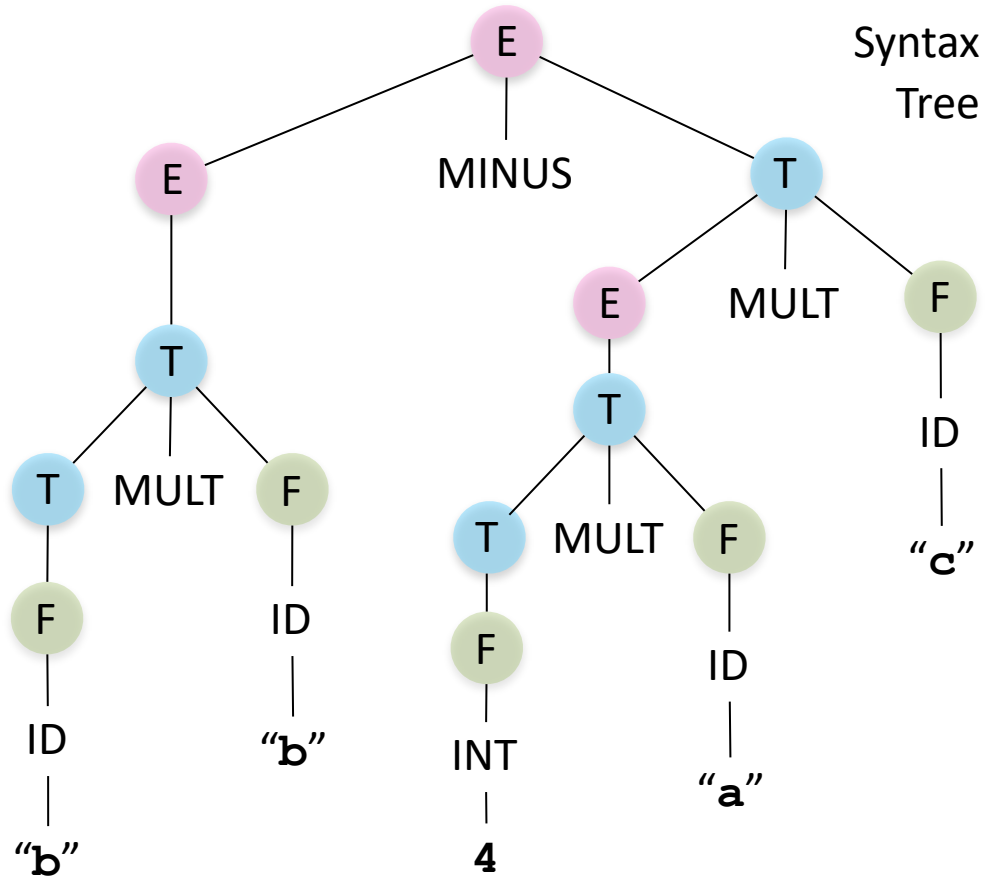
Syntax Analysis

Semantic Analysis

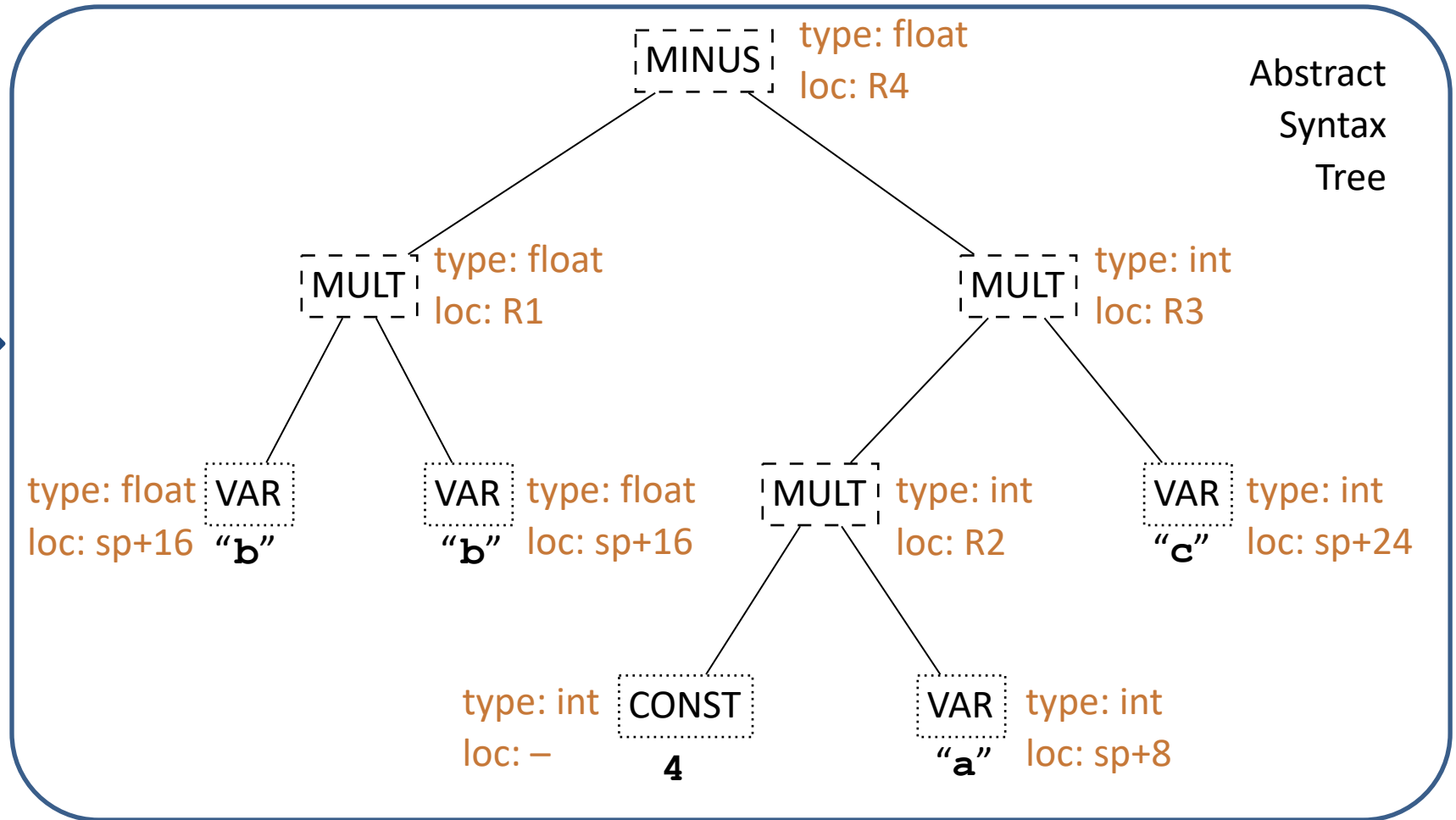
IR
Opt.

Code
Gen.

Recap

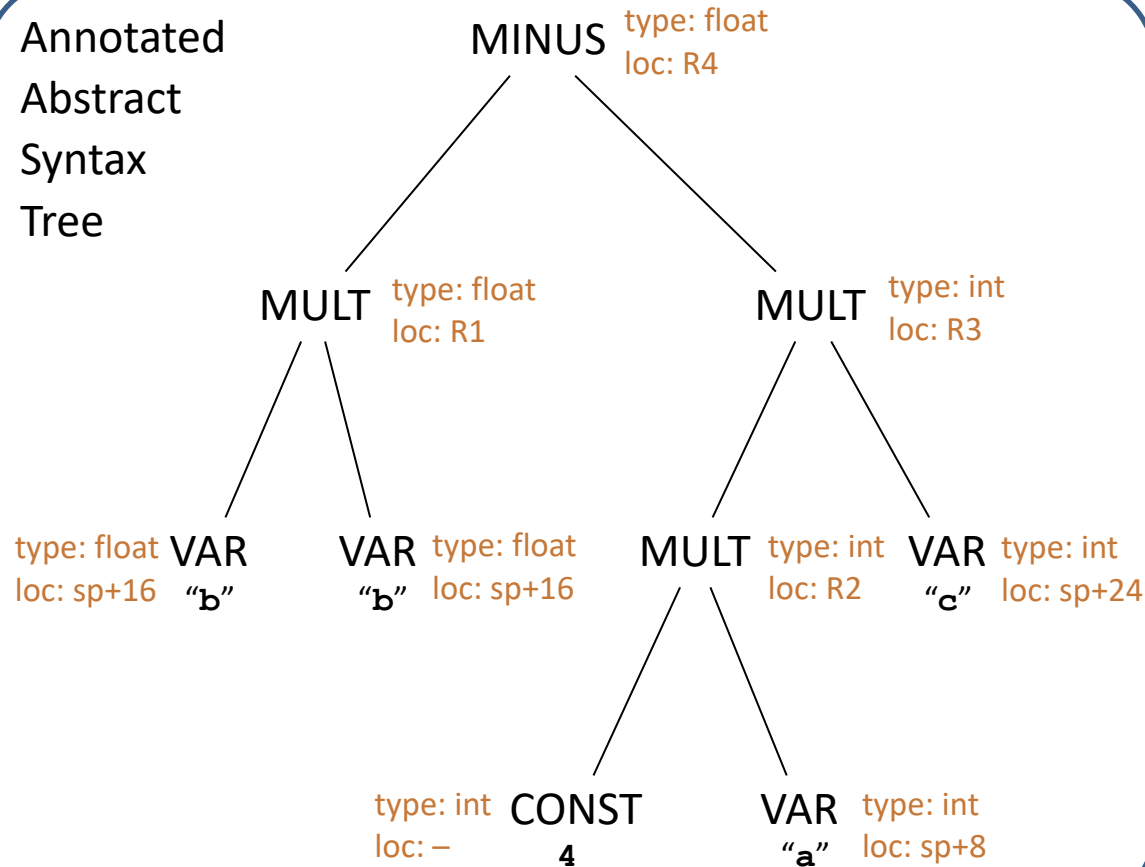


Recap



Recap

Annotated
Abstract
Syntax
Tree

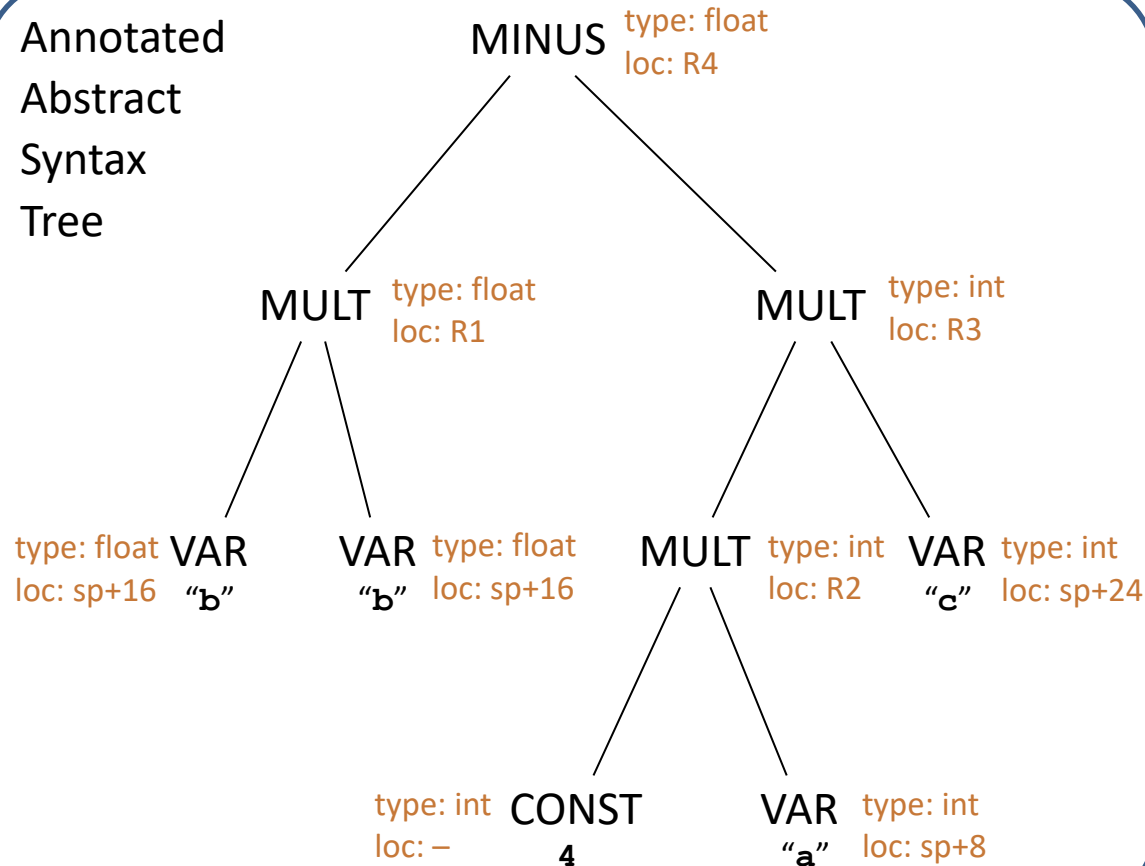


Intermediate
Representation

$R1 = b * b$
 $R2 = 4 * a$
 $R3 = R2 * c$
 $R5 = (\text{float})R3$
 $R4 = R1 - R5$

Recap

Annotated
Abstract
Syntax
Tree



Intermediate
Representation

```
R1 = b * b
R2 = 4 * a
R3 = R2 * c
R5 = (float)R3
R4 = R1 - R5
```

Assembly
Code

```
fild    [esp+16]
fmul    st0, st0
mov     ebx, [esp+8]
sal     ebx, 2
mul     ebx, [sp+24]
fild    ebx
fsub    st0, st1
```

Lexical
Analysis

Syntax
Analysis

Semantic
Analysis

IR
Opt.

Code
Gen.

Recap

- Runtime considerations
 - activation records (for managing function calls)
 - dynamic memory management, garbage collection
 - object oriented aspects: inheritance & dynamic dispatch

Make sure you understand:

- What happens at **compile time**
- What happens at **runtime**
- How the first affects the second!!



You Have Reached
Your Destination

