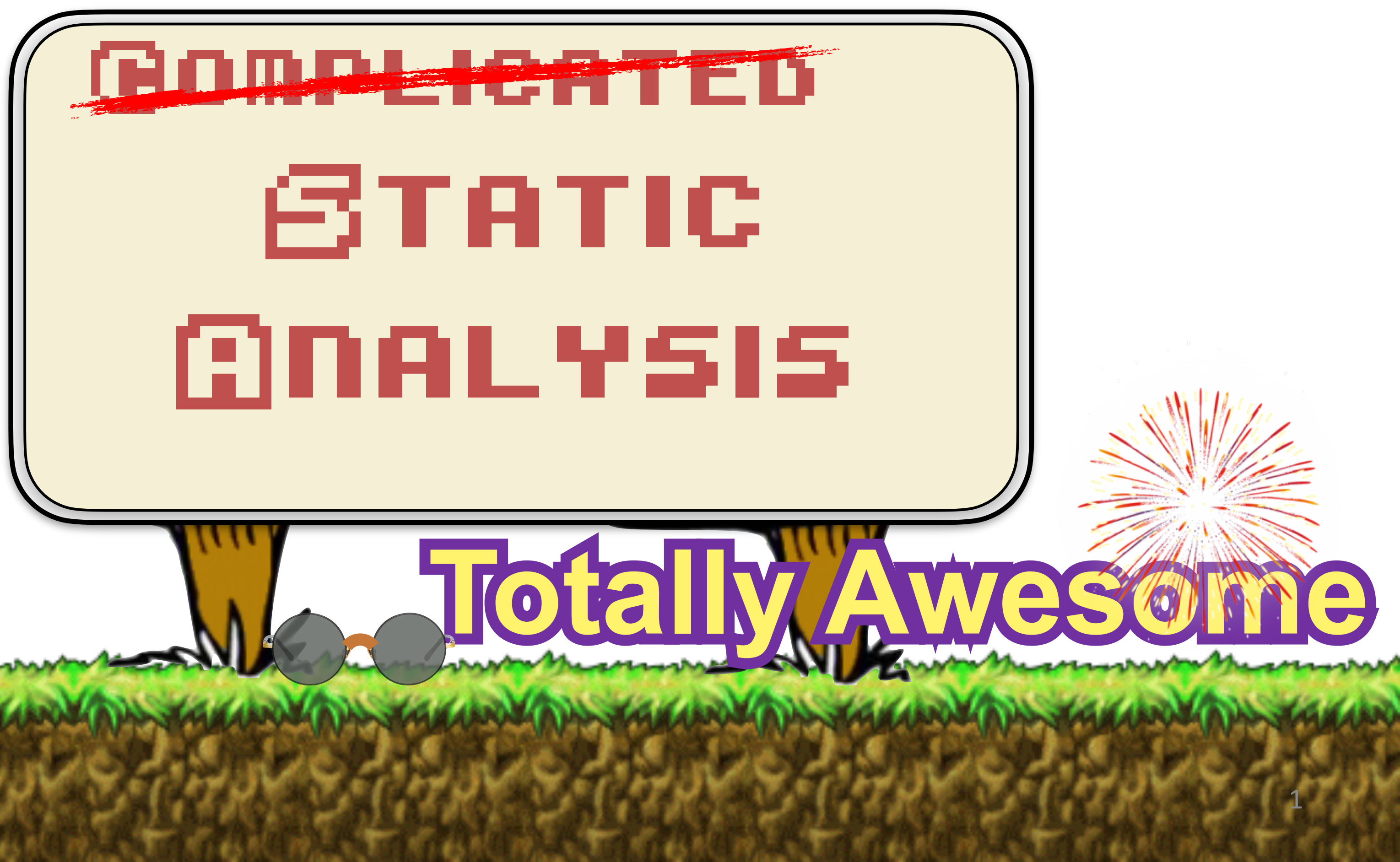
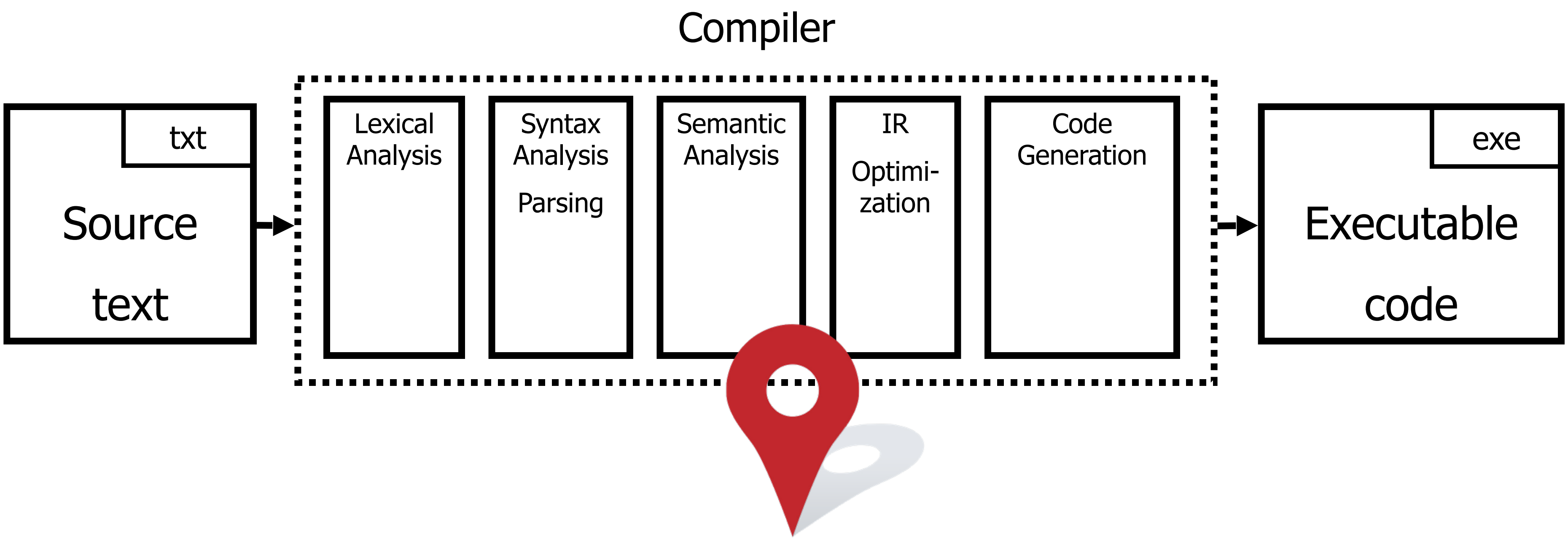
THEORY OF COMPILATION

LECTURE 11



You are here


Compiler
txt
Source
text
Lexical
Analysis
Syntax
Analysis
Parsing
Semantic
Analysis
IR
Optimi-
zation
Code
Generation
exe
Executable
code

Plan for Today

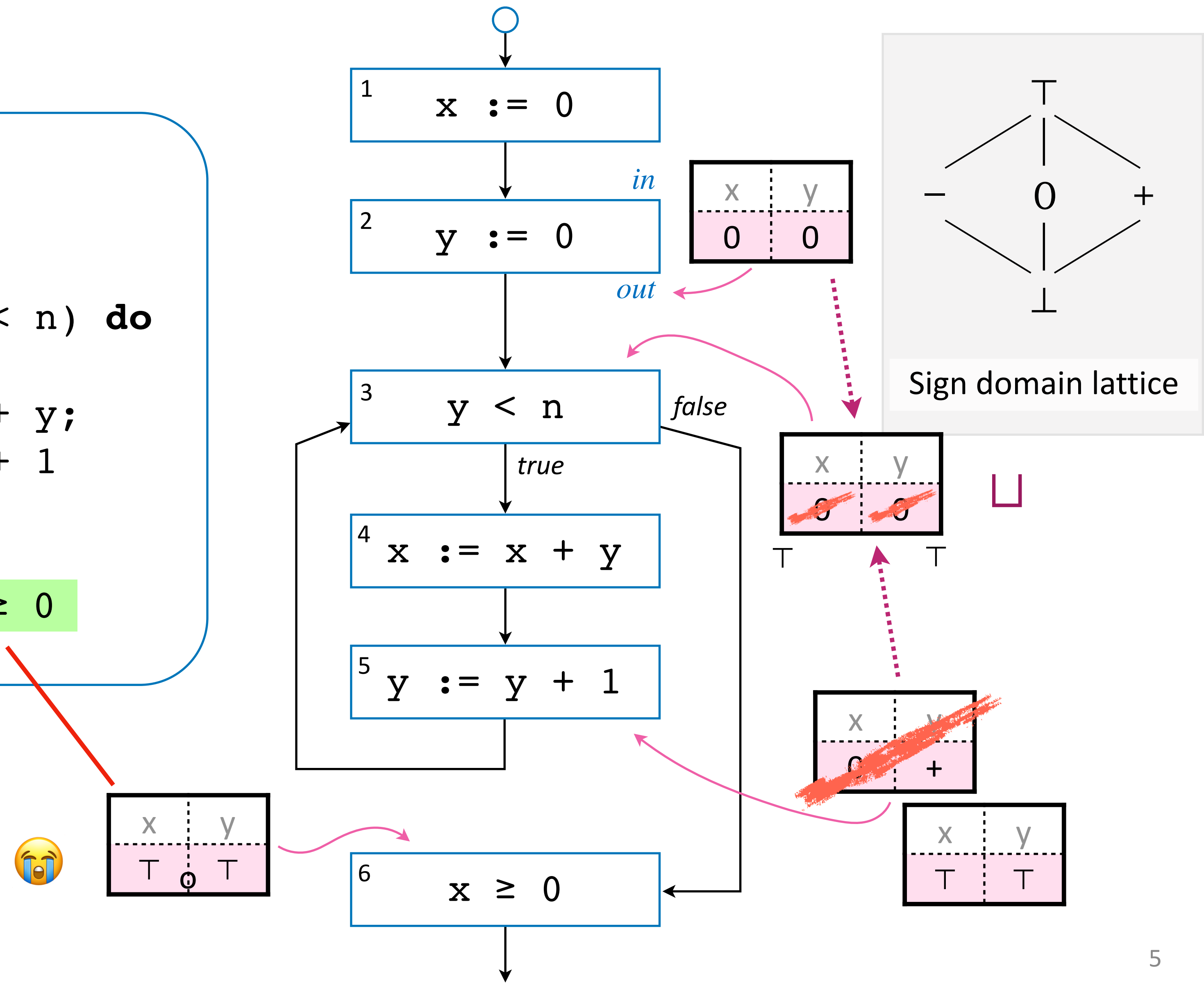
- Refresh our memory of Data Flow Analysis
- Another example: pointer analysis
- Combining domains

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)
```

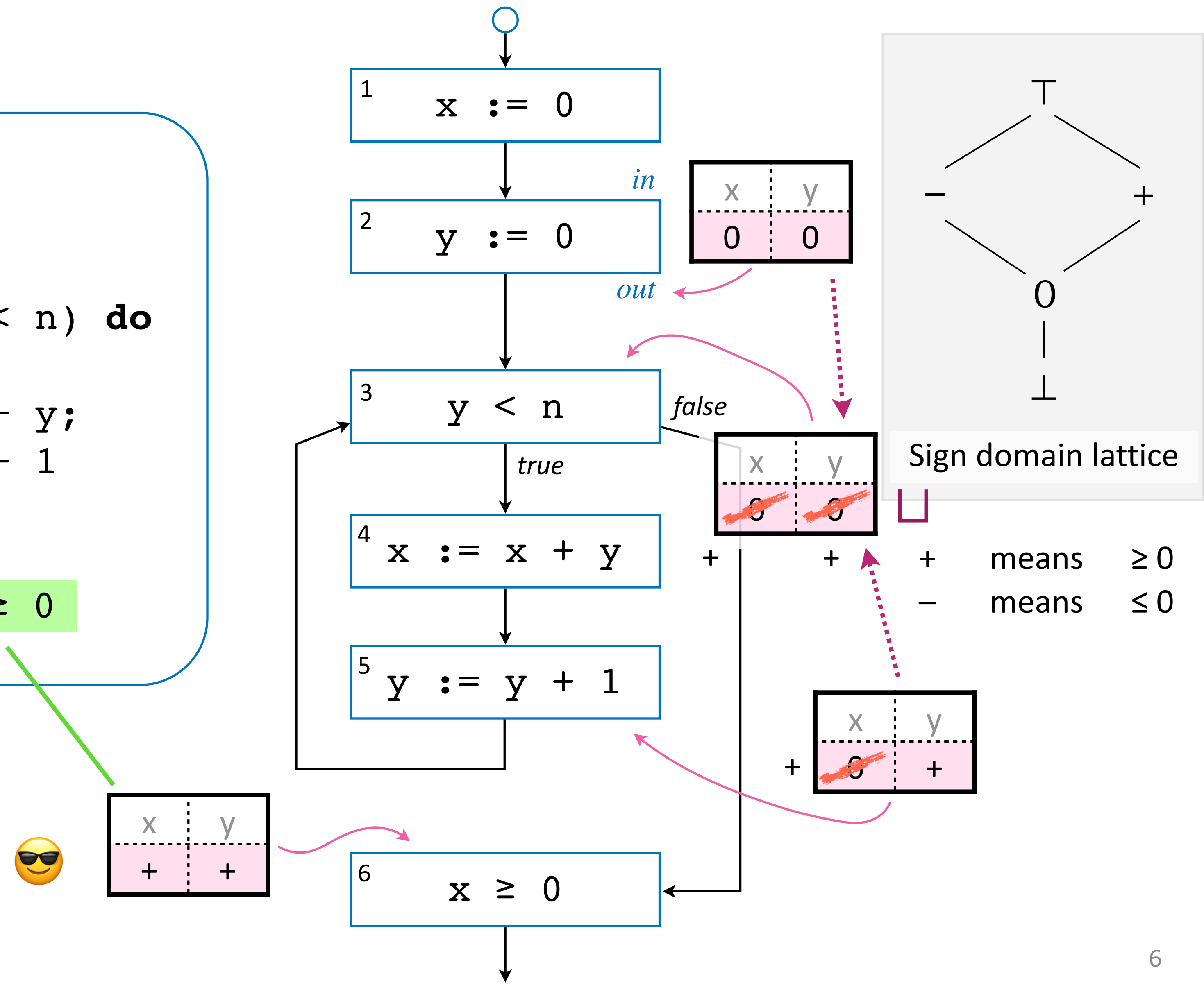
Data Flow Analysis (ugh, again?!)

```
1: x := 0;
2: y := 0;

3: while (y < n) do
     (
4:     x := x + y;
5:     y := y + 1
     )

6: assert x ≥ 0
```



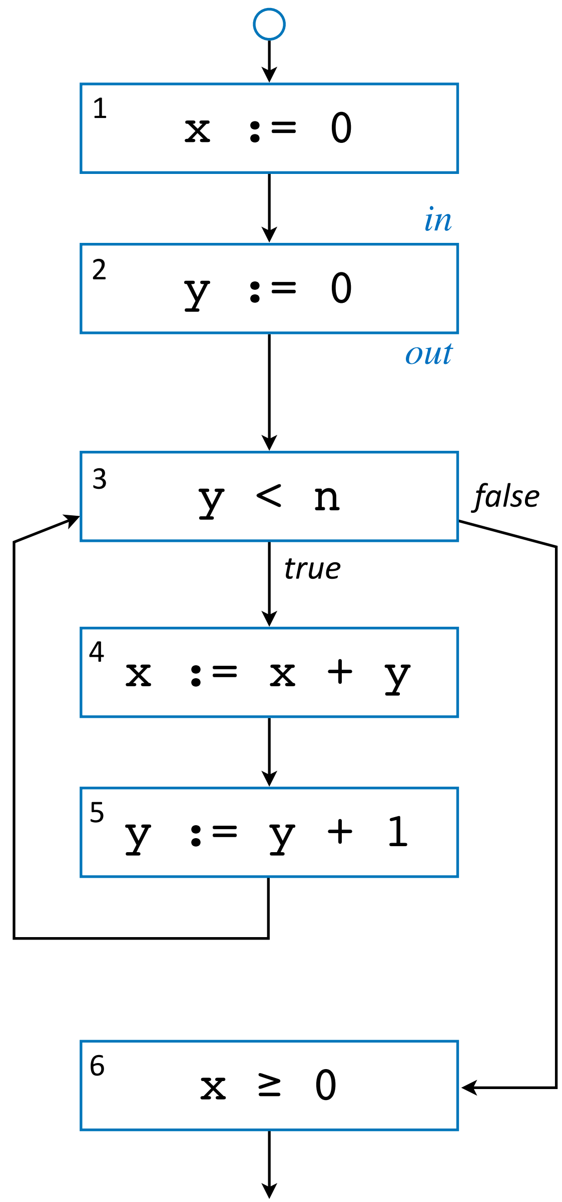
Data Flow Analysis (ugh, again?!)

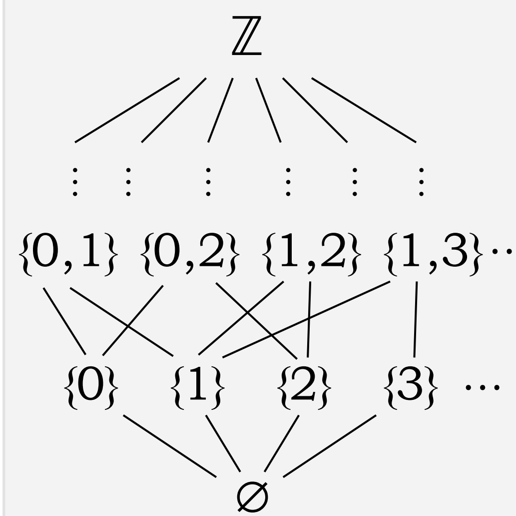
```
1: x := 0;
2: y := 0;

3: while (y < n) do
   (
4:    x := x + y;
5:    y := y + 1
   )

6: assert x ≥ 0
```



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
```



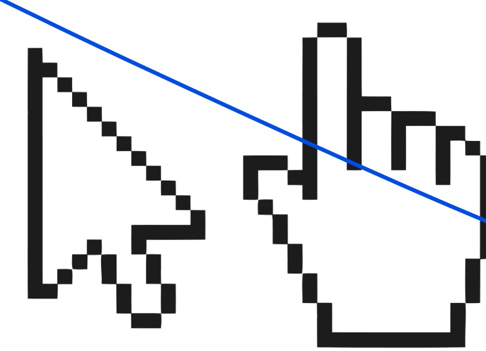


Integer power-set lattice

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

$a \sqcup b = a \cup b$

Pointer Analysis



Simple Example

program

```
x := 5          S1
ptr := &x       S2
*ptr := 9       S3
y := x          S4
```

data flow

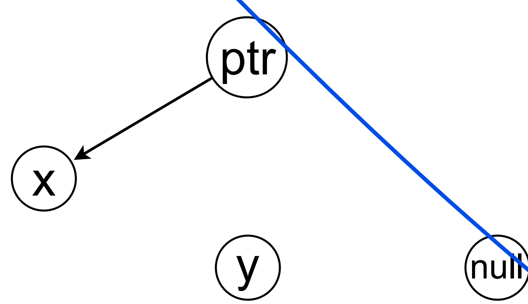
- What are the data dependencies in this program?
- <u>Problem</u>: just looking at variable names will not give you the correct information
  - After statement S2, program names "x" and "*ptr" are both expressions that refer to the same memory location.
  - We say that ptr *points-to* x after statement 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

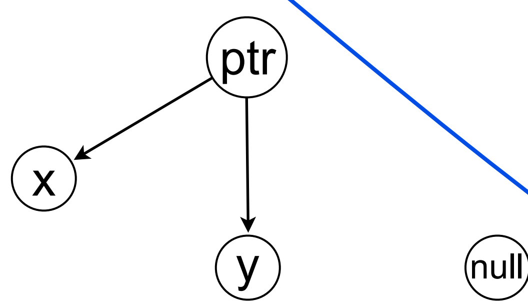
- Directed graph:
  - Nodes are program variables (+ special node for null)
  - Edge (a,b) — variable a points-to variable b

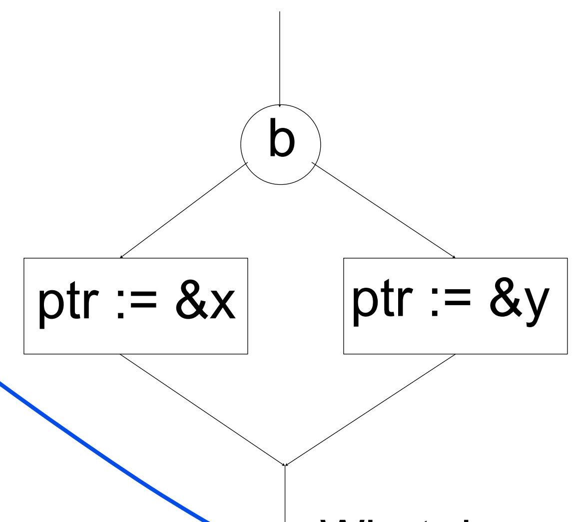


- Of course, points-to is different at different program locations

Points-to Relation

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



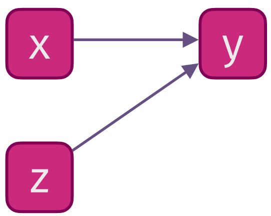


What does x point to here?

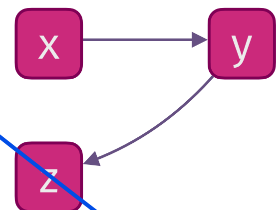
- Out-degree may be > 1
  if there there are multiple paths

Points-to Relation

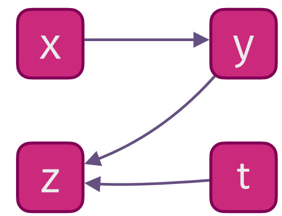
```
x := &y
z := &y
```



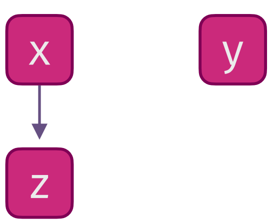

```
x := &y
y := &z
```



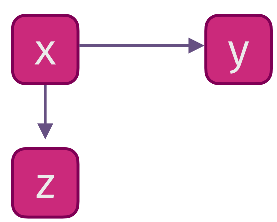

```
x := &y
t := &z
*x := t
```




```
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

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

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'

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

G

x := &y

G' = G with $pt'(x) \mapsto \{y\}$

G

x := *y

G' = G with

$pt'(x) \mapsto \bigcup\{pt(a) \mid a \in pt(y)\}$

G

x := y

G' = G with $pt'(x) \mapsto pt(y)$

G

*x := y

G' = G with

$pt'(a) \mapsto pt(a) \cup pt(y)$

for all $a \in pt(x)$

strong updates

weak update (why?)

Dynamic Allocation

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

also called *summary objects*

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

Type-based

AbsObj = $\{ \mathrm{H}_{\mathrm{T}} \mid$ T is a type in the program $\}$

Allocation-site based

AbsObj = $\{ \mathrm{H}_{\ell} \mid \ell \in$ Lab *s.t.* ℓ: `p := new T` $\}$

Dynamic Allocation: Semantics

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

```
p := new Z[5];
q := new Z[5];
if (p = q) then
      z := p
else
      z := q
```

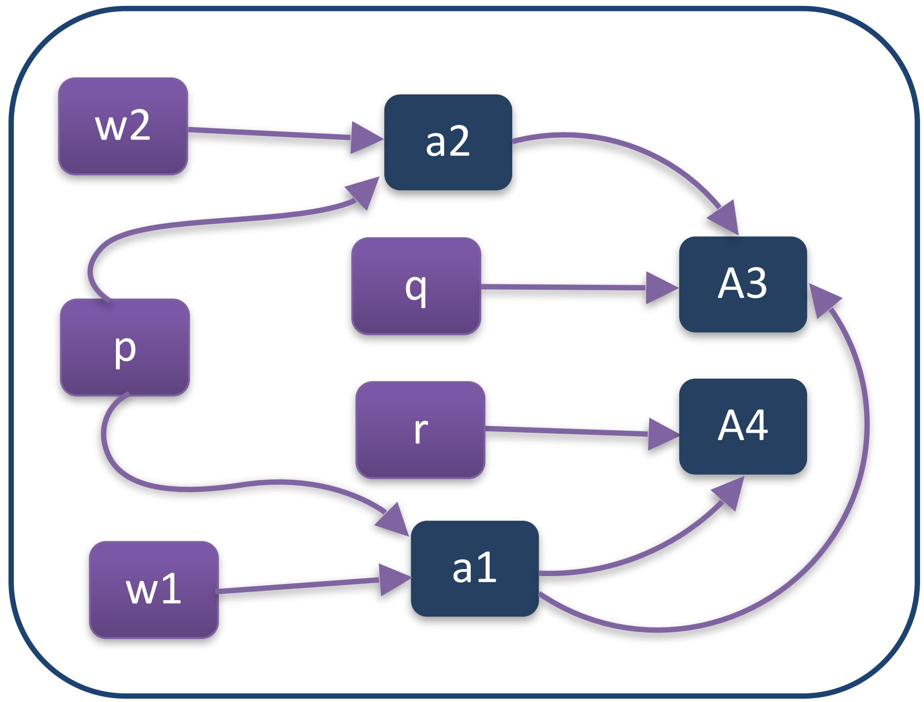
```
p := &A1;
q := &A2;
if (p = q) then
      z := p
else
      z := q
```

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

Points-to Analysis: Example

```
w1 := &a1;
w2 := &a2;
q  := new Z[5];
r  := new Z[5];
*w1  := r;
if (…) then
    p := w1
else
    p := w2;
*p := q
```

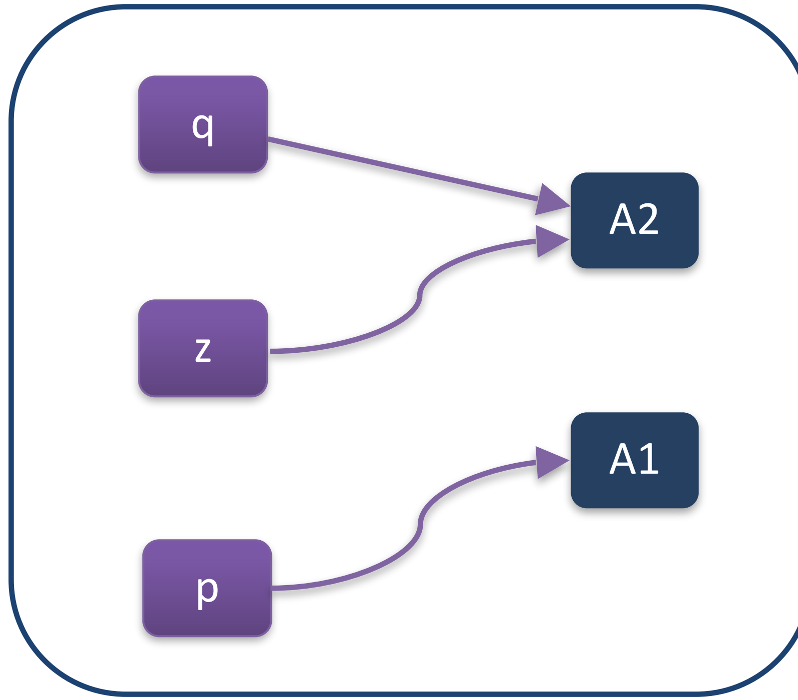
```
a1, a2, q, r : Z*
w1, w2, p    : Z**
```



Aliasing Analysis

Derived from result of point-to analysis

```
p := new Z[5];
q := new Z[5];
if (p = q) then
     z := p
else
     z := q
```



z and p **may not** alias

q and p **may not** alias

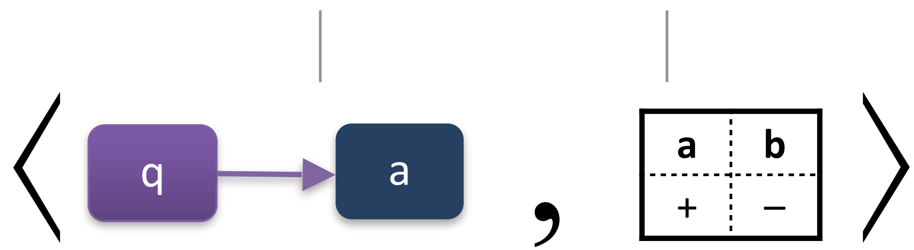
z and q **may** alias

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

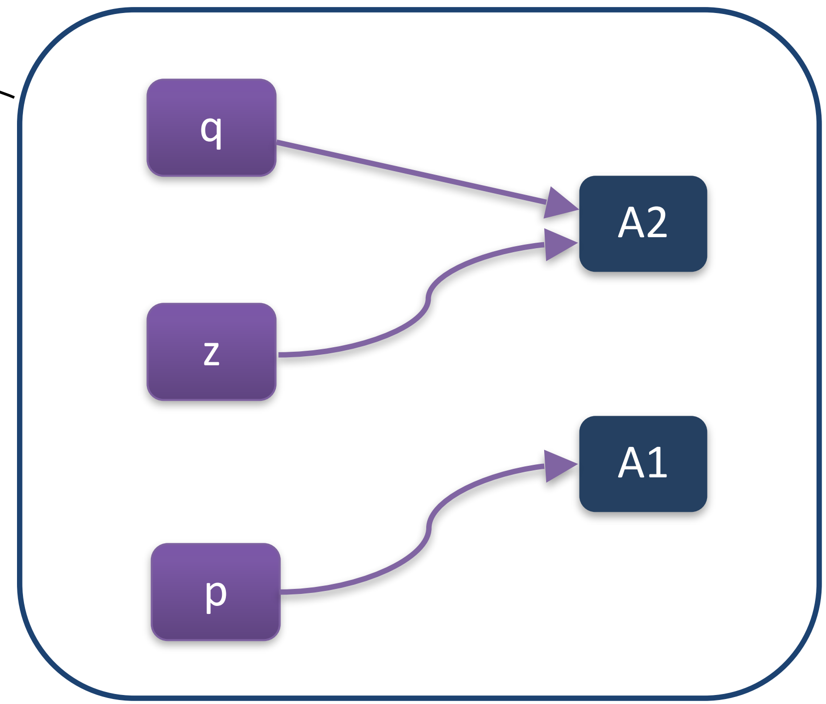
1: a := *q + 4 * c;

2: b := 0;

3: *p := 1;

4: b := ~~*q + 4 * c~~

a



Optimization is valid:

p and q are **not** aliased

What we learned about

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, …

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

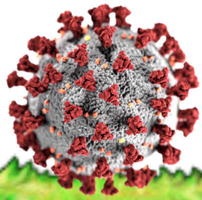
13:00 – 16:00

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

Hybrid Exam!

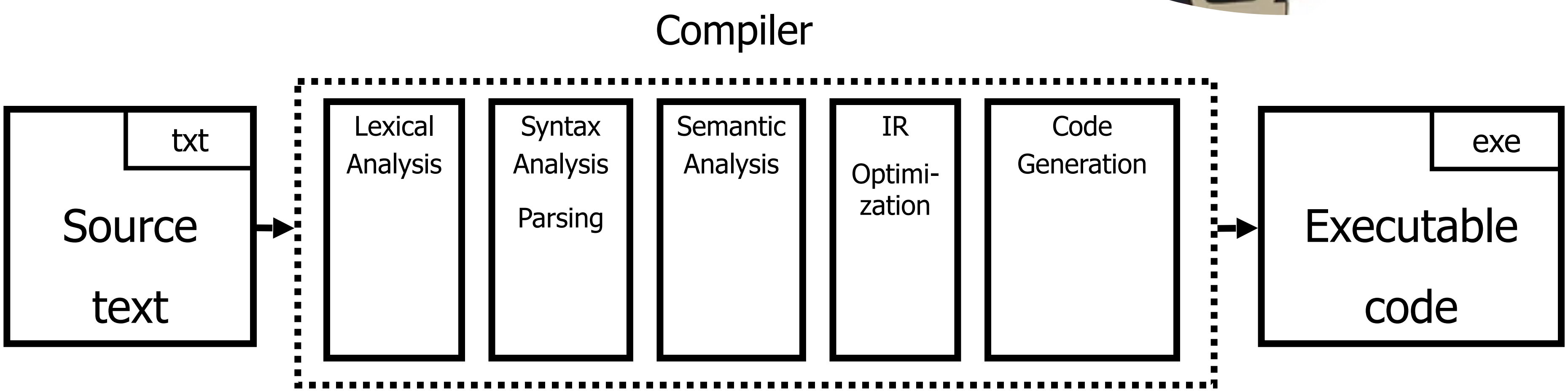
45% multiple choice questions

55% open questions



Recap


total
RE

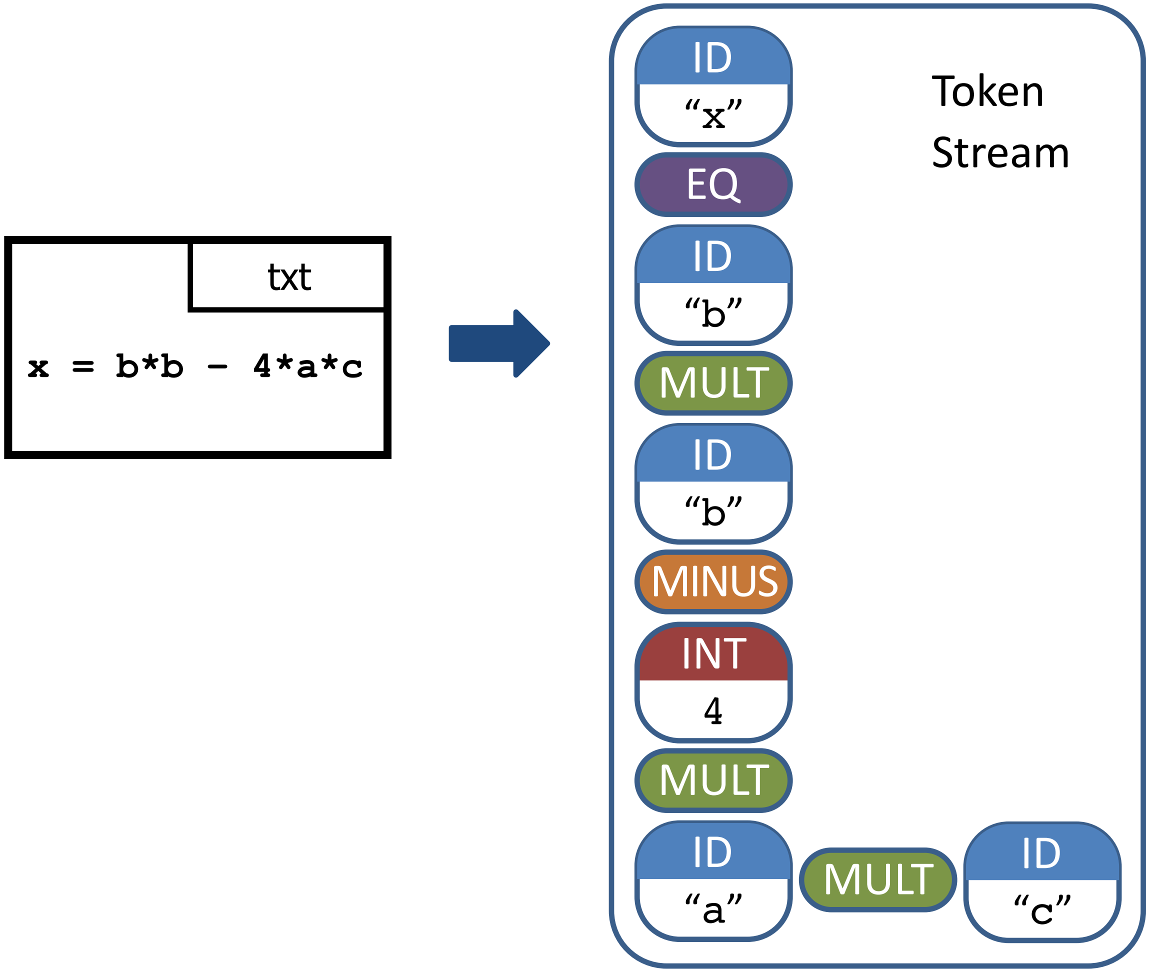

Compiler
txt
Source text
Lexical Analysis
Syntax Analysis
Parsing
Semantic Analysis
IR
Optimi-zation
Code Generation
exe
Executable code

Recap

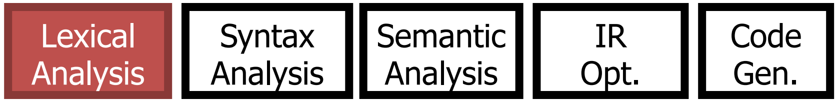
txt

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

Recap

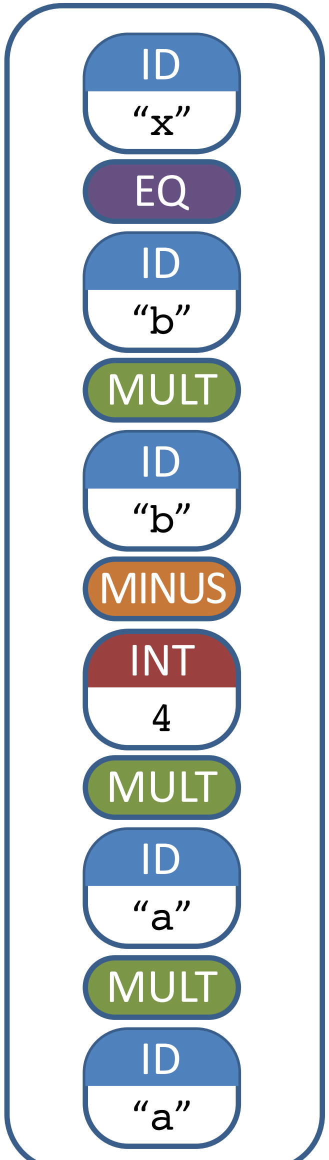


txt
x = b*b – 4*a*c
ID
"x"
EQ
ID
"b"
MULT
ID
"b"
MINUS
INT
4
MULT
ID
"a"
MULT
ID
"c"
Token Stream



Lexical Analysis
Syntax Analysis
Semantic Analysis
IR Opt.
Code Gen.

Recap



Token Stream

Grammar

E ⟶ E (PLUS) T

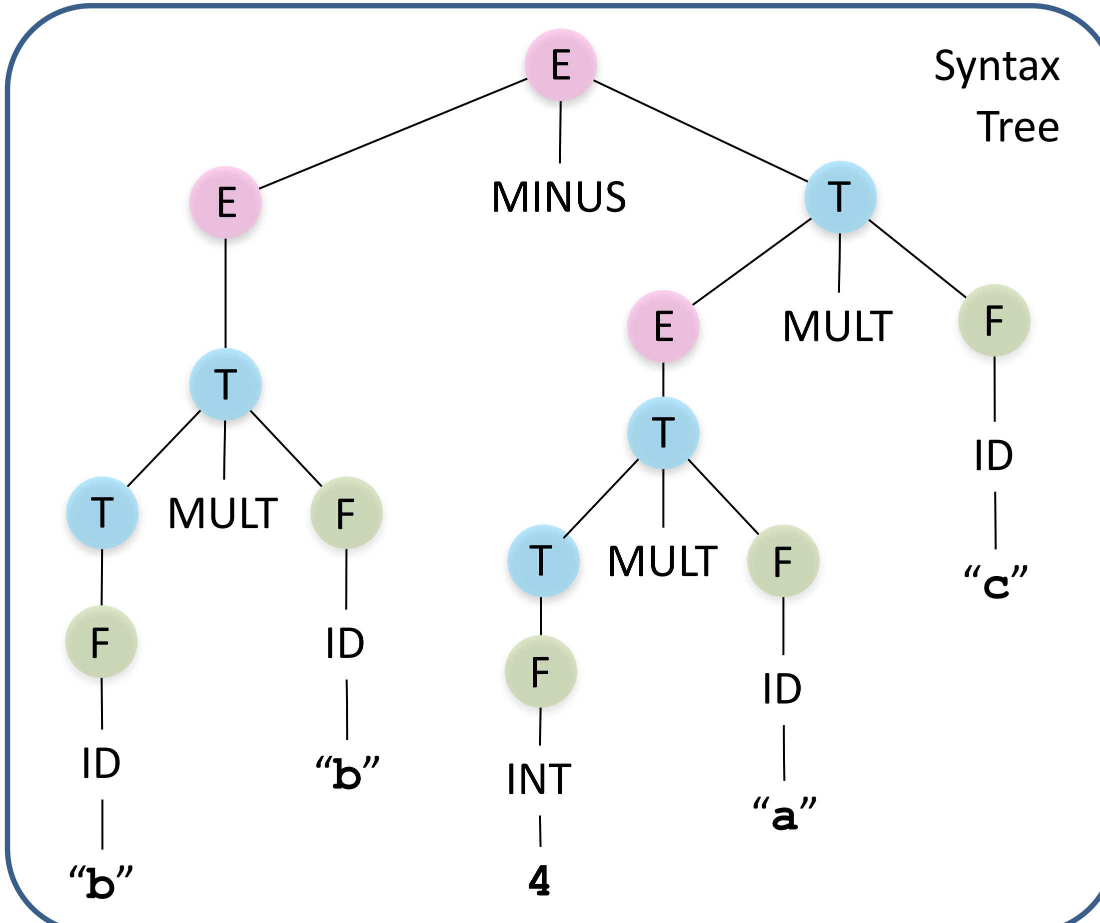
E ⟶ E (MINUS) T

T ⟶ T (MULT) F

T ⟶ T (DIV) F

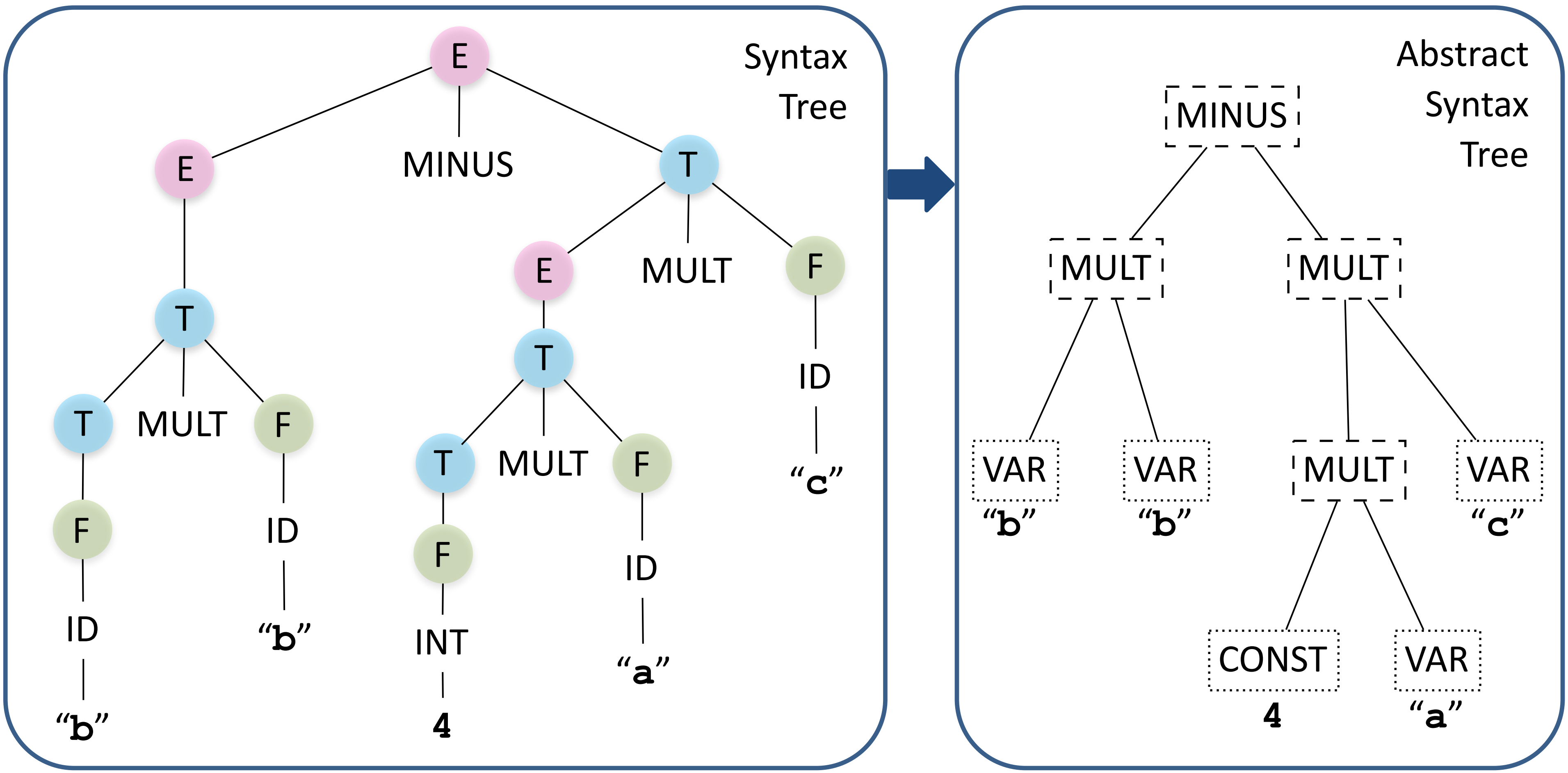
F ⟶ (ID)

F ⟶ (INT)



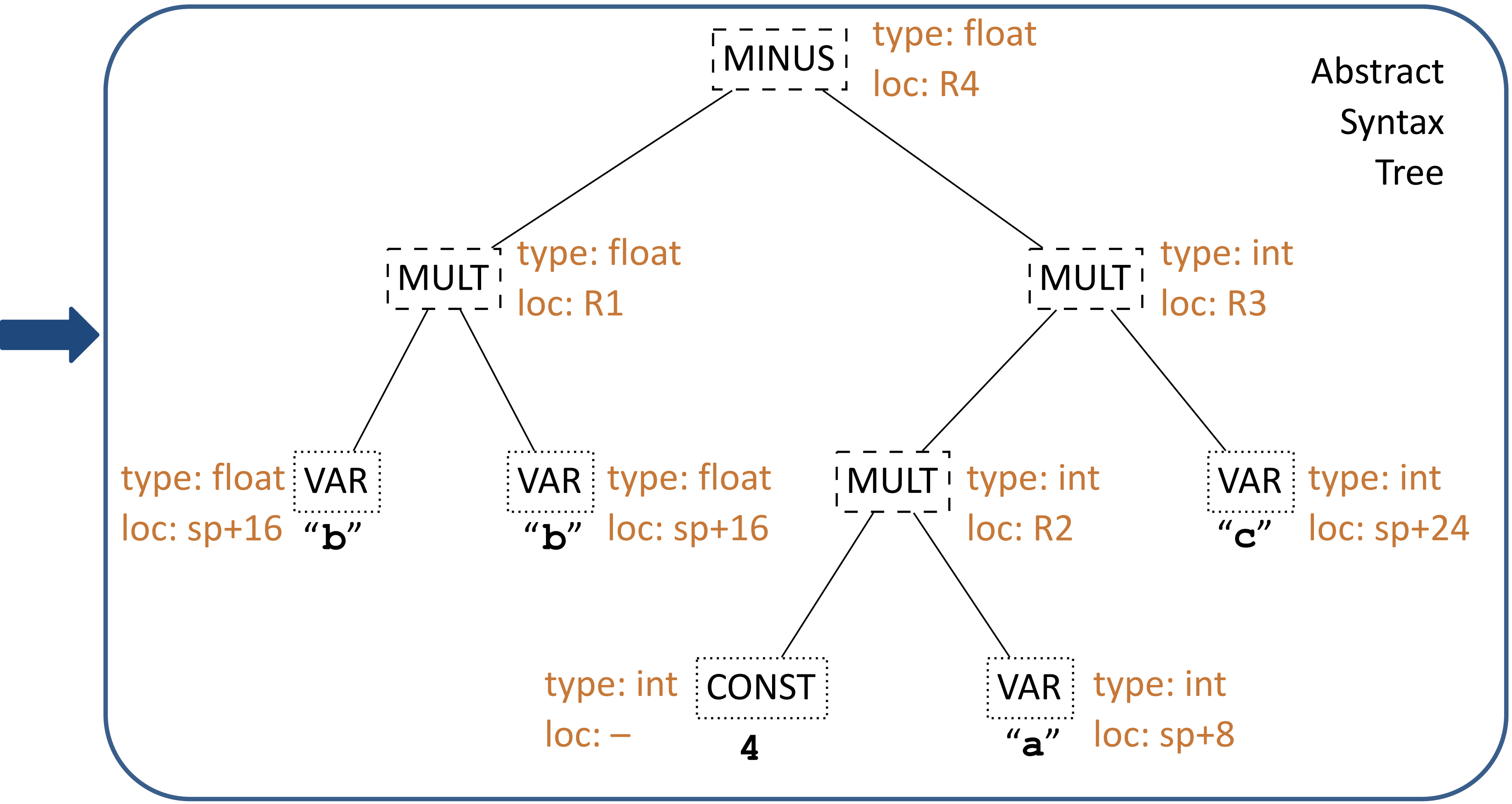
Lexical Analysis | Syntax Analysis | Semantic Analysis | IR Opt. | Code Gen.

Recap



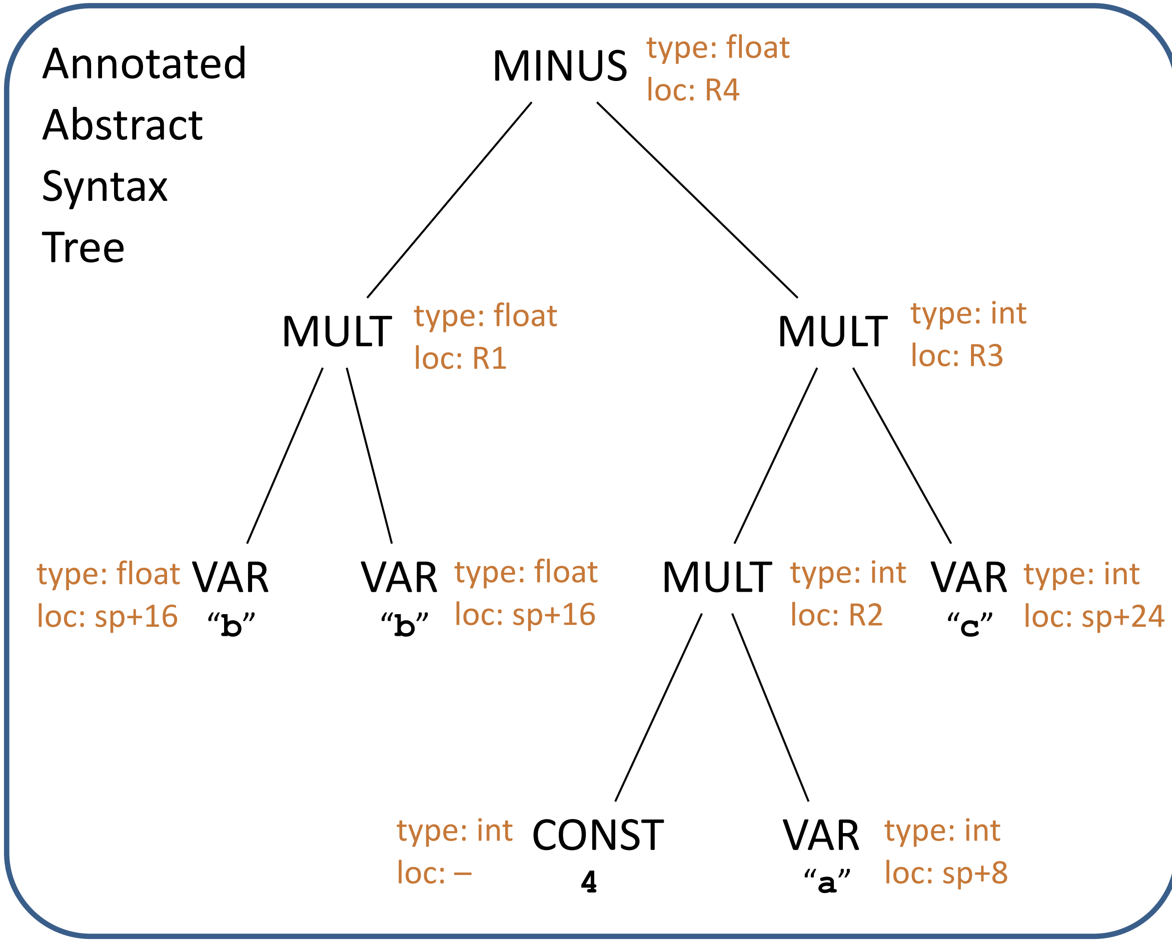
Lexical Analysis | Syntax Analysis | Semantic Analysis | IR Opt. | Code Gen.

Recap


MINUS
type: float
loc: R4
Abstract Syntax Tree
MULT
type: float
loc: R1
MULT
type: int
loc: R3
type: float
loc: sp+16
VAR
"b"
VAR
"b"
type: float
loc: sp+16
MULT
type: int
loc: R2
VAR
"c"
type: int
loc: sp+24
type: int
loc: –
CONST
4
VAR
"a"
type: int
loc: sp+8


Lexical Analysis
Syntax Analysis
Semantic Analysis
IR Opt.
Code Gen.

Recap



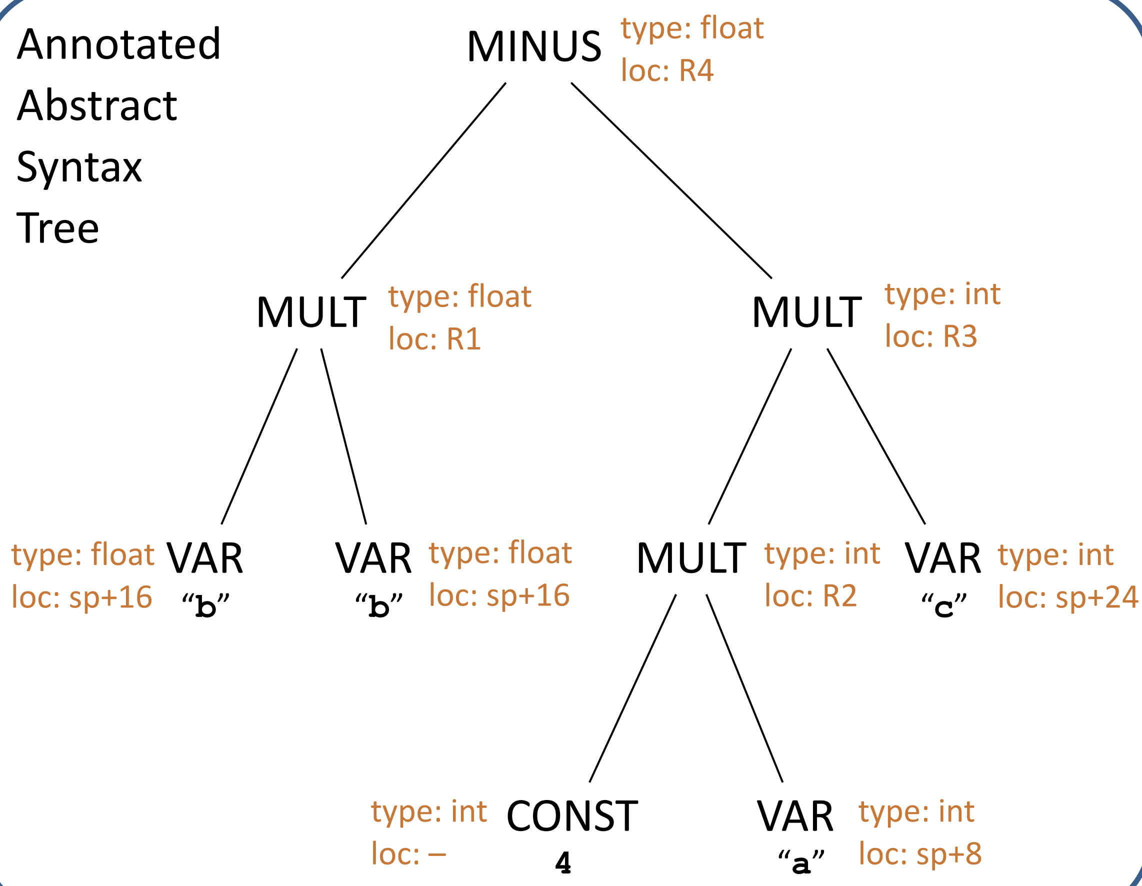
Intermediate Representation

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

Lexical Analysis | Syntax Analysis | Semantic Analysis | IR Opt. | Code Gen.

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