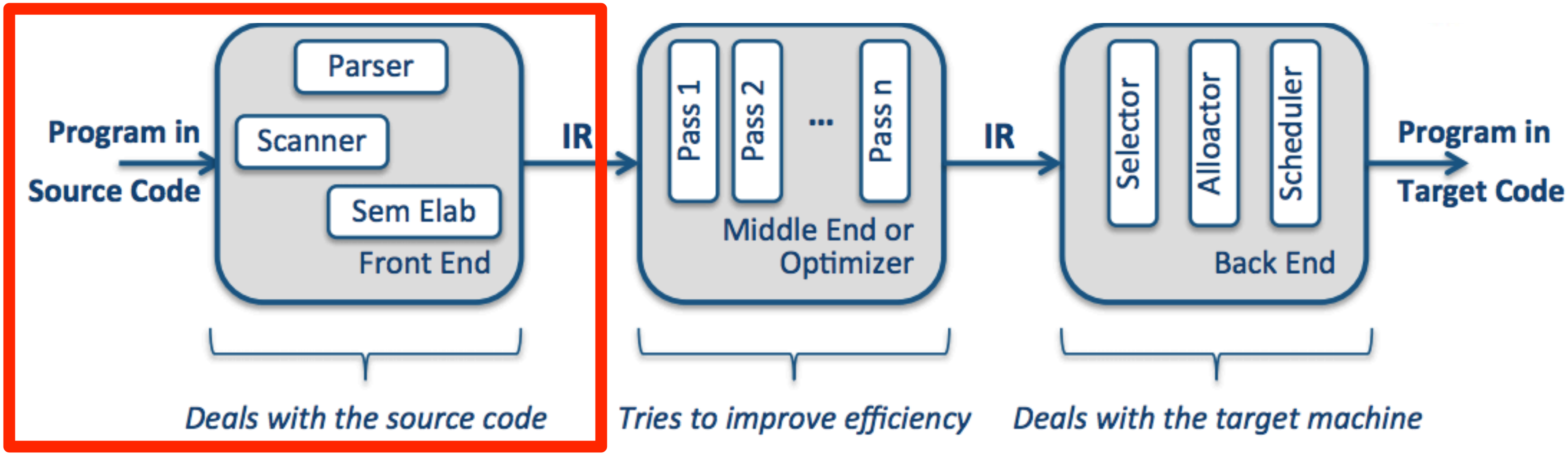
ABSTRACT SYNTAX

Compiler Design: Syntactic and Semantic Analysis

§4.1

Slides by Rohit Chadha, Bill Harrison, Chris Hathhorn.

Inside a Compiler

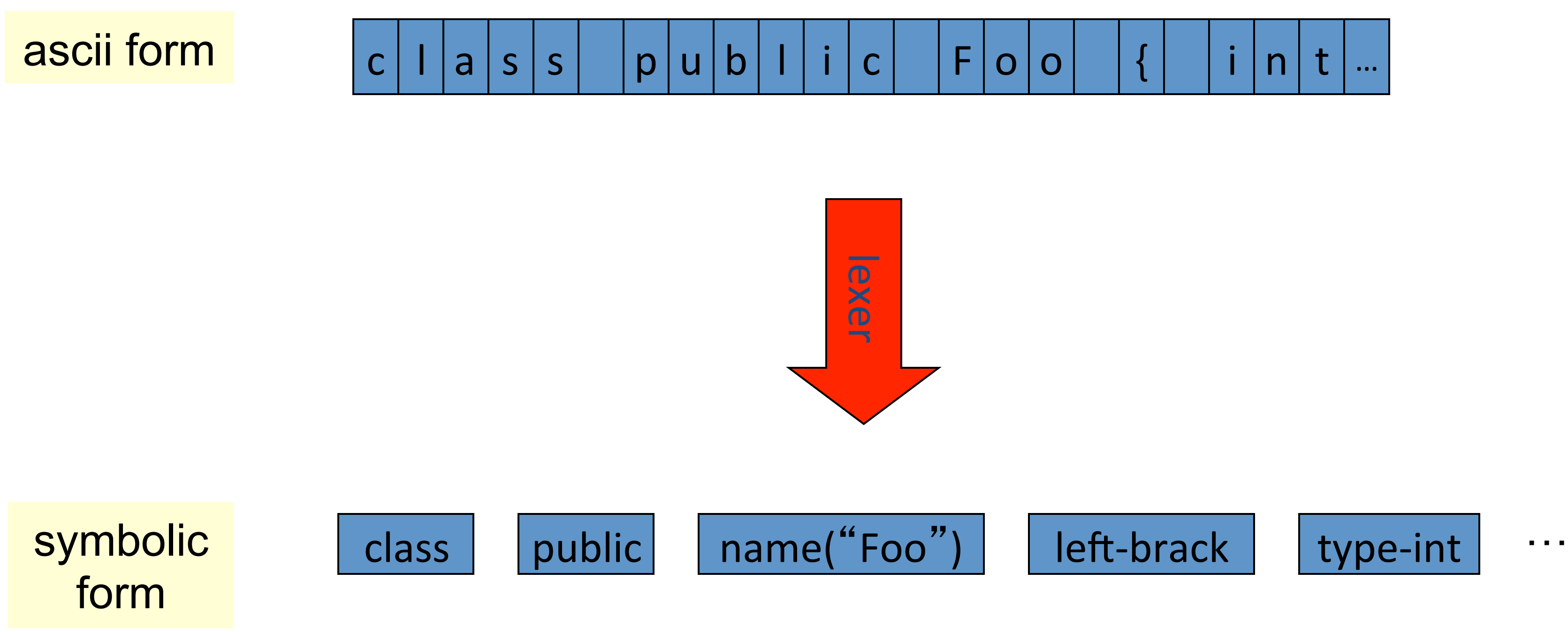


Front end

- Lexical analysis (the lexer or scanner)
- Syntax analysis (parsing)
- Semantic analysis

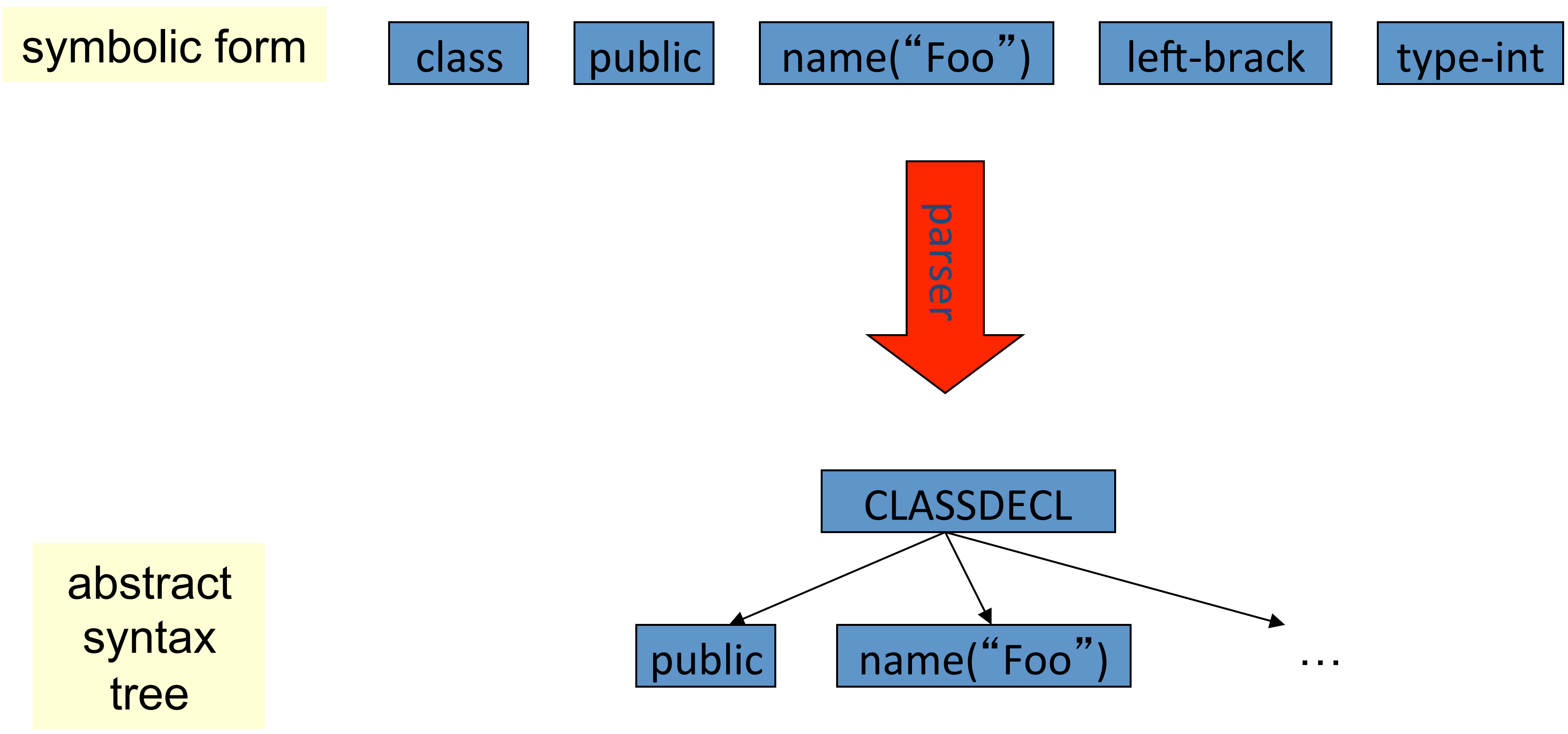
*graphic © Keith Cooper & Linda Torczon.

Front End: Lexical Analysis



Key Concept: regular expressions

Front End: Parsing



Key Concept: "Backus-Naur form" grammars (BNF)

Front End: Semantic Analysis

For example, type checking and violations of scope rules:

```
string x = “abc”;
int y = 2;
int z = x + y;
```

```
int x = 1;
{
    int y = 2;
}
int z = x + y;
```

Front End: Semantic Analysis

For example, type checking and violations of scope rules:

```
string x = “abc”;
int y = 2;
int z = x + y;
```

Error! Attempting to add an int to a string.

```
int x = 1;
{
    int y = 2;
}
int z = x + y;
```

Error! “y” is not in scope.

Concrete vs. Abstract Syntax

- A *concrete parse tree* (or just *parse tree*) represents the *concrete syntax* of our language.
  - In general, it would contain a node for every non-terminal in the grammar and a leaf node for every terminal.
- Much of the information in a concrete parse tree is irrelevant for the later phases in the compiler.
- For our purposes, the parser will pass on an *abstract syntax tree* to the semantic analysis phase. An abstract syntax tree is the parse tree pruned of irrelevant details.

Parse Tree → AST

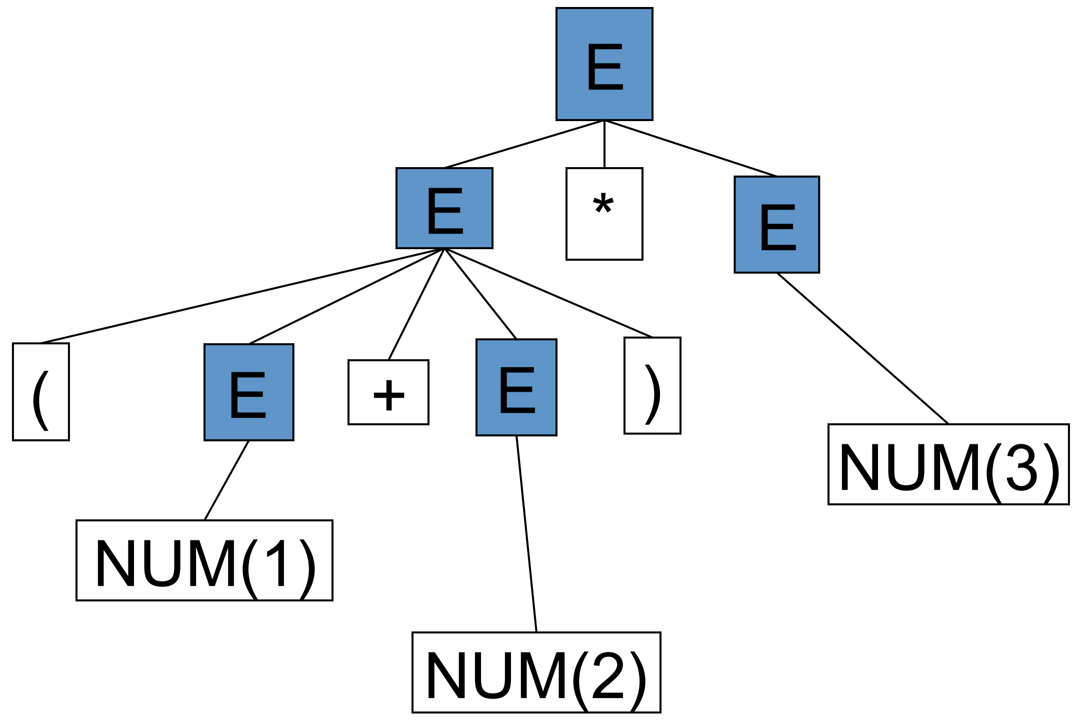
***E* → NUM**

E* → *E* * *E

E* → *E* + *E

***E* → (*E*)**

Parse (1 + 2) * 3.



But do we need to remember the brackets, the production symbol E etc?

Parse Tree → AST

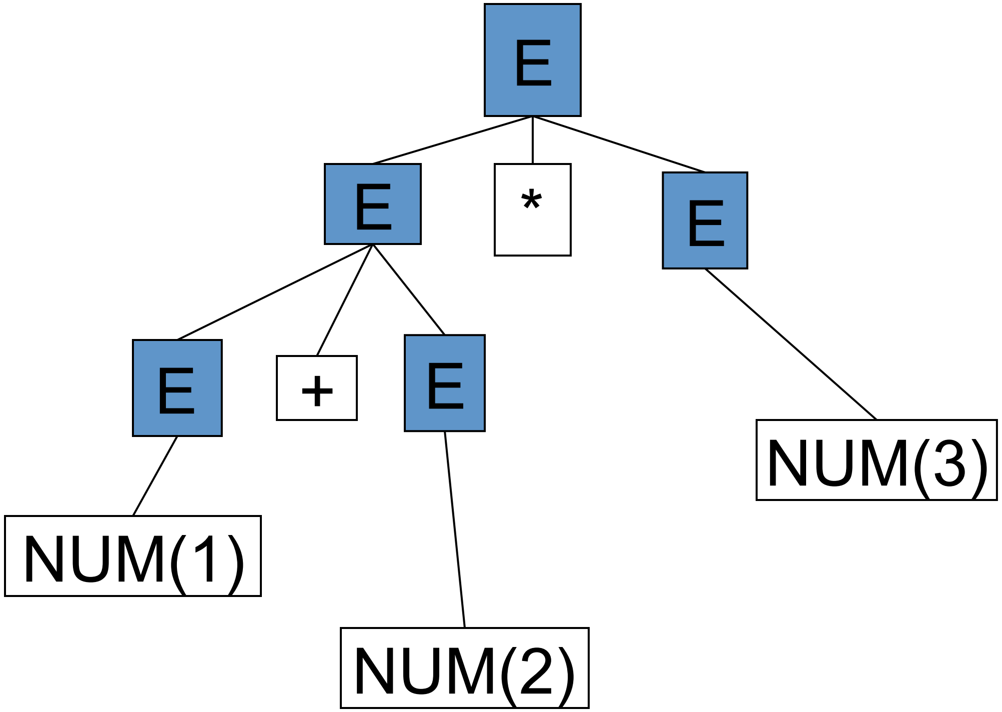
***E* → NUM**

E* → *E* * *E

E* → *E* + *E

***E* → (*E*)**

Parse (1 + 2) * 3.



But do we need to remember the brackets, the production symbol E etc?

Parse Tree → AST

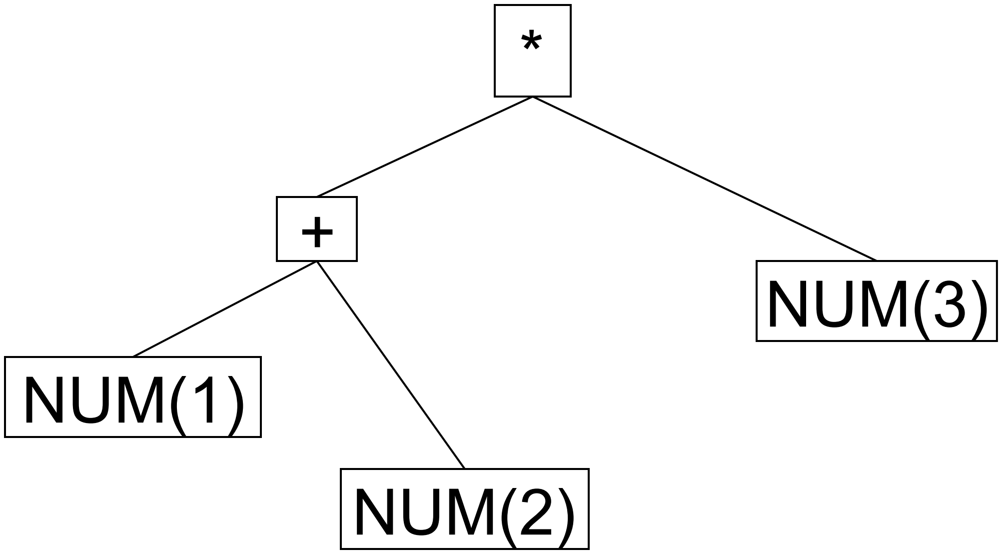
***E* → NUM**

E* → *E* * *E

E* → E + *E

***E* → (*E*)**

Parse (1 + 2) * 3.



We remember only the important details.

Representing AST in Haskell

$E \rightarrow \mathbf{NUM}$

$E \rightarrow E * E$

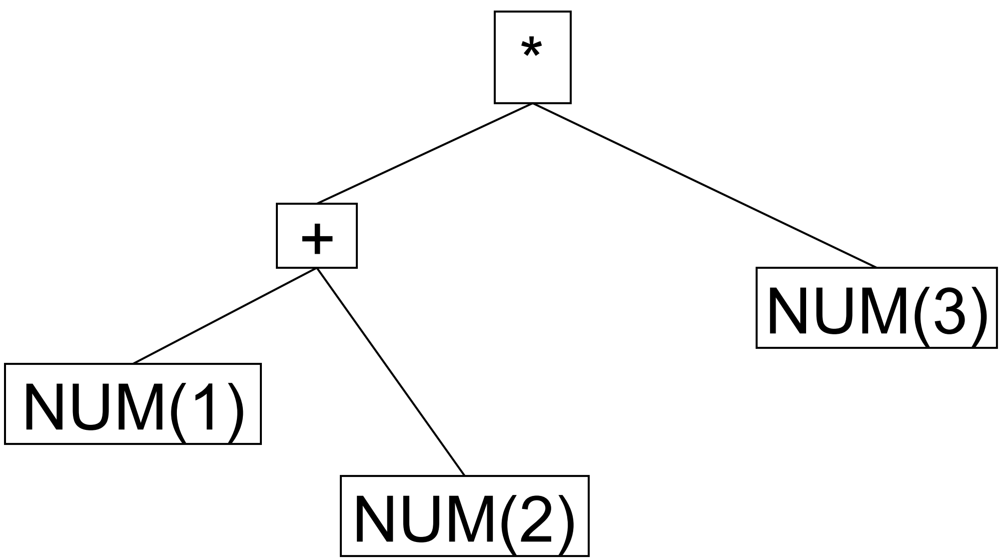
$E \rightarrow E + E$

$E \rightarrow (E)$

```
data E = NUM Int   |
         MULT E E  |
         PLUS E E
```

Parse (1 + 2) * 3.

```
MULT (PLUS (NUM 1) (NUM 2)) (NUM 3)
```



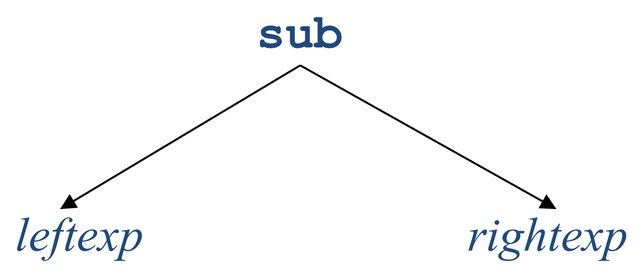
We remember only the important details.

Translating **simple** expressions

Sample in Concrete Syntax

```
c - 1
```

Corresponding Abstract Syntax

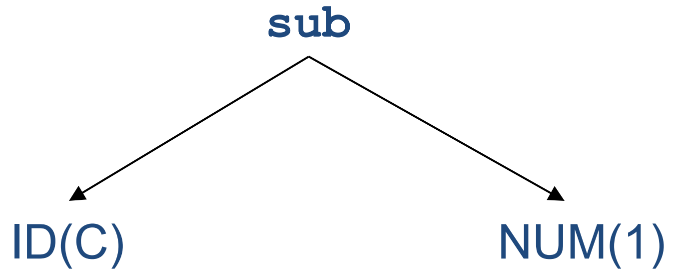


Translating **simple** expressions

Sample in Concrete Syntax

```
C - 1
```

Corresponding Abstract Syntax

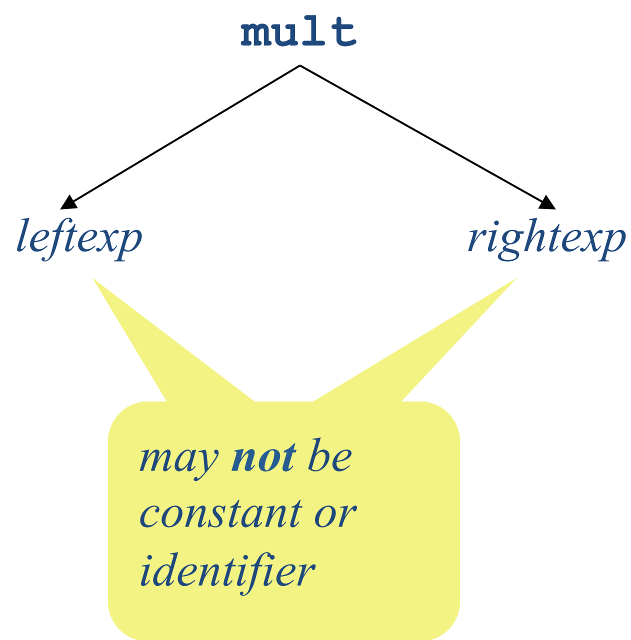


Translating composite expressions

Sample in Concrete Syntax

```
2 * (C – 1)
```

Corresponding Abstract Syntax

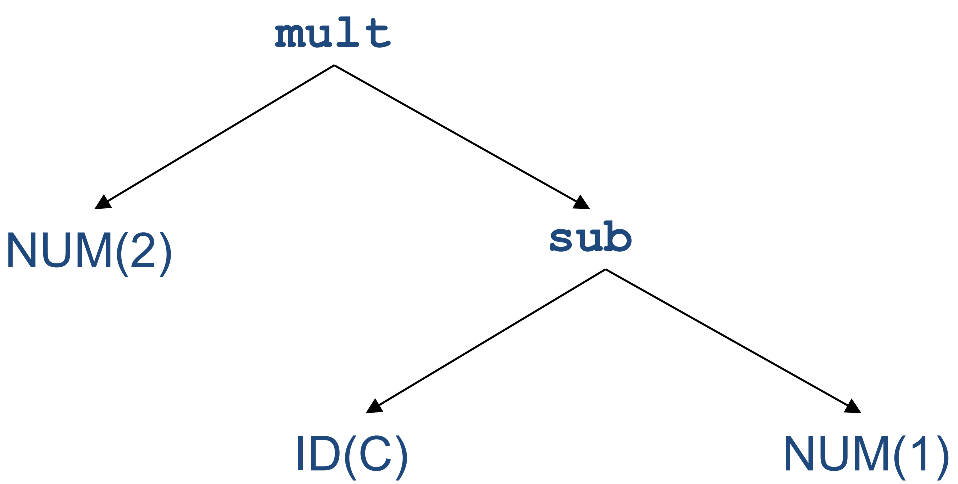


Translating composite expressions

Sample in Concrete Syntax

```
2 * (C – 1)
```

Corresponding Abstract Syntax

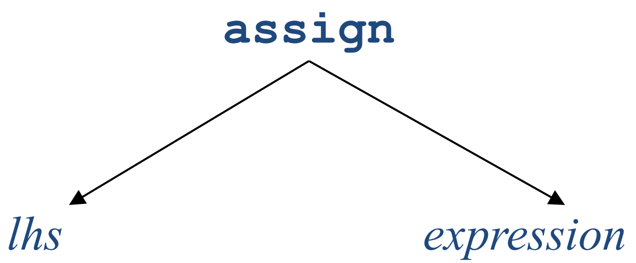


Translating assignment "… = …"

Sample in Concrete Syntax

```
C = A + 5
```

Corresponding Abstract Syntax

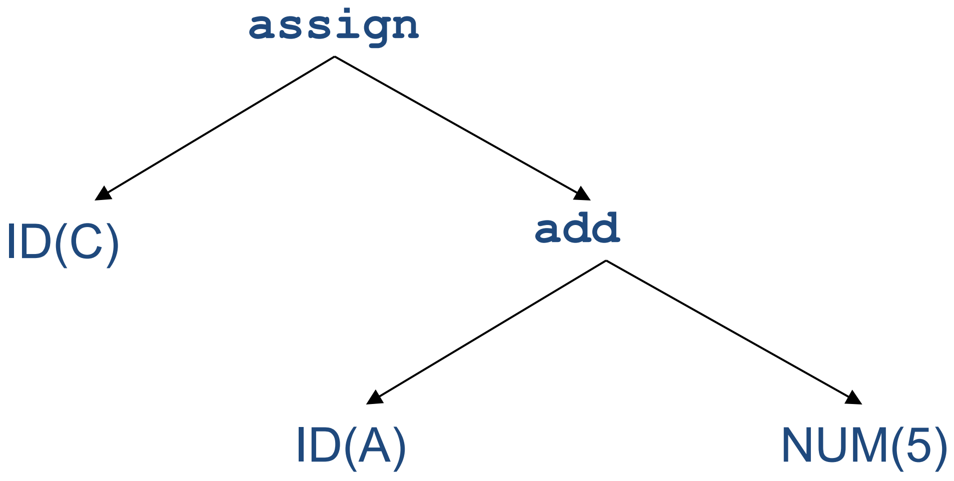


Translating assignment "… = …"

Sample in Concrete Syntax

```
C = A + 5
```

Corresponding Abstract Syntax

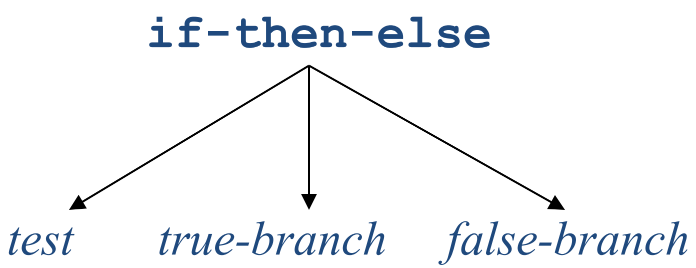


Translating "if ... else ..."

Sample in Concrete Syntax

```
if A > B {
   C = A + 5;
} else {
   C = B + 5;
}
```

Corresponding Abstract Syntax



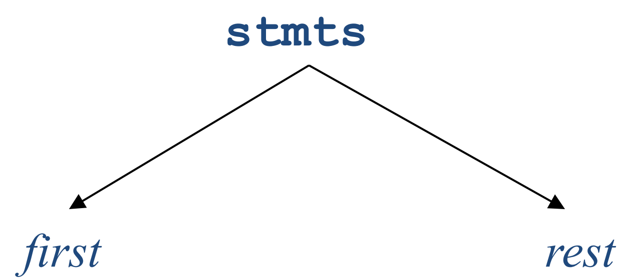
Question: what is the AST for the code on the left?

Translating “$\texttt{stmt}_1$; $\texttt{stmt}_2$”

Sample in Concrete Syntax

```
C = A + 5;
C = B + 5
```

Corresponding Abstract Syntax



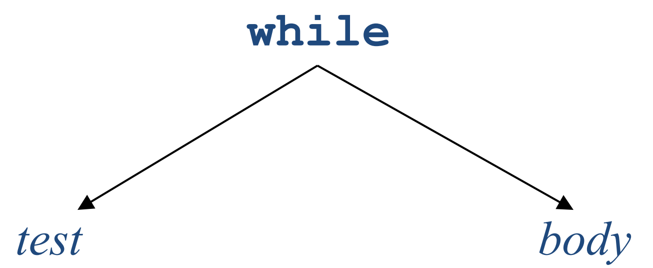
Question: what is the AST for the code on the left?

Translating while loops

Sample in Concrete Syntax

```
while A > B {
     C = A + 5;
}
```

Corresponding Abstract Syntax



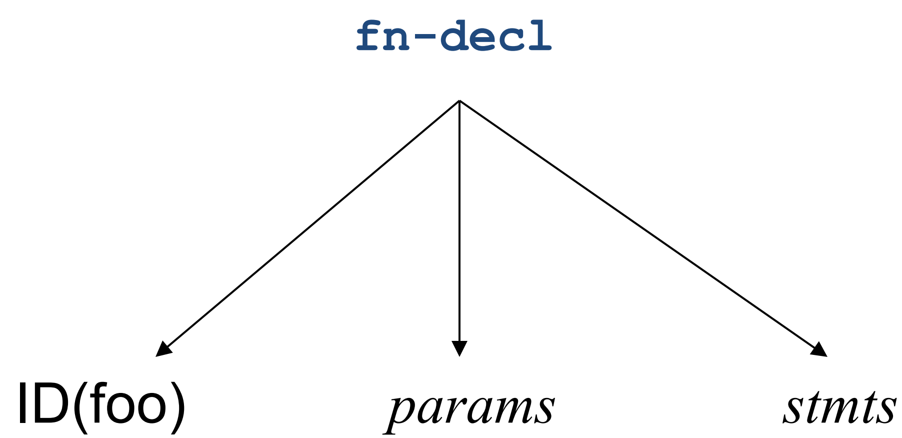
Question: what is the AST for the code on the left?

Translating function declarations

Sample in Concrete Syntax

```
fn foo(a: i64) {
   let mut w = 0;
   let z = 2;
   w = z+4;
}
```

Corresponding Abstract Syntax



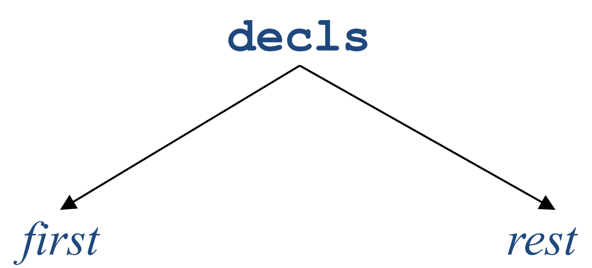
Question: what is the AST for the code on the left?

Translating declarations

Sample in Concrete Syntax

```
fn a(…) { … }
fn b(…) { … }
…
```

Corresponding Abstract Syntax



Question: what is the AST for the code on the left?

Create ASTs with YACC/Happy specification

Combine code generation with a YACC style specification?

…

<statement> → ID := <expression> ; { }

<statement> → read (<id list>) ; { }

<statement> → write (<expr list>) ; { }

…

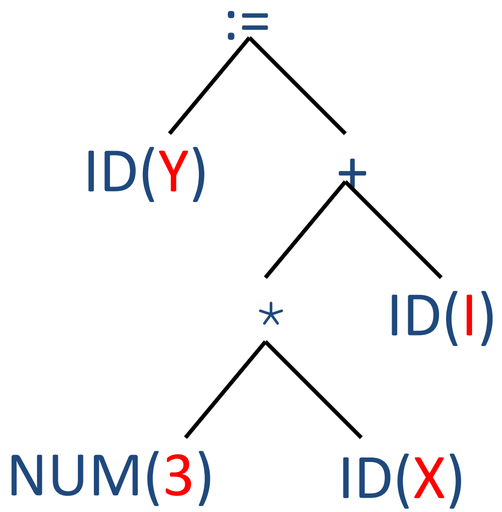
create AST in C or Haskell

Static vs. Dynamic program properties

- **Static** properties
  - any property that may be determined through analysis of <u>program text</u>
    - e.g., for some languages, the type of a program may be determined entirely through analysis of program source
      - e.g., ML, Java, & Pascal have "static type inference"
- **Dynamic** properties
  - any property that may only be discovered through execution of the program
    - e.g., "the final result of program p is 42" – can't be discovered in general without some form of execution
- Compilation involves many forms of "static analysis"
  - e.g., type checking, the definition and use of variables, information of data and control flow, …

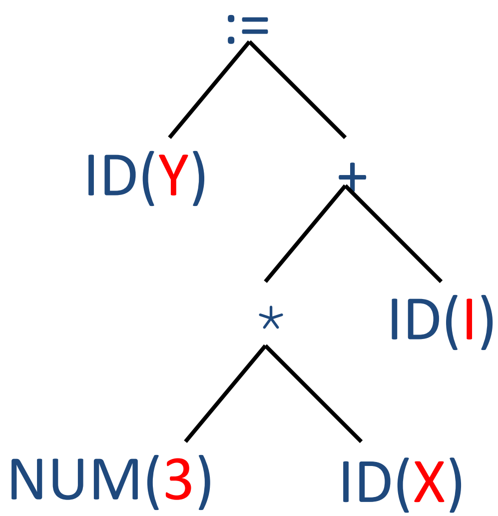
Attribute ASTs and static checking

Assume: we know Y, I, and X are variables of type float
Question: is the following a legal program?



Attribute ASTs and static checking

Assume: we know Y, I, and X are variables of type float
Question: is the following a legal program?

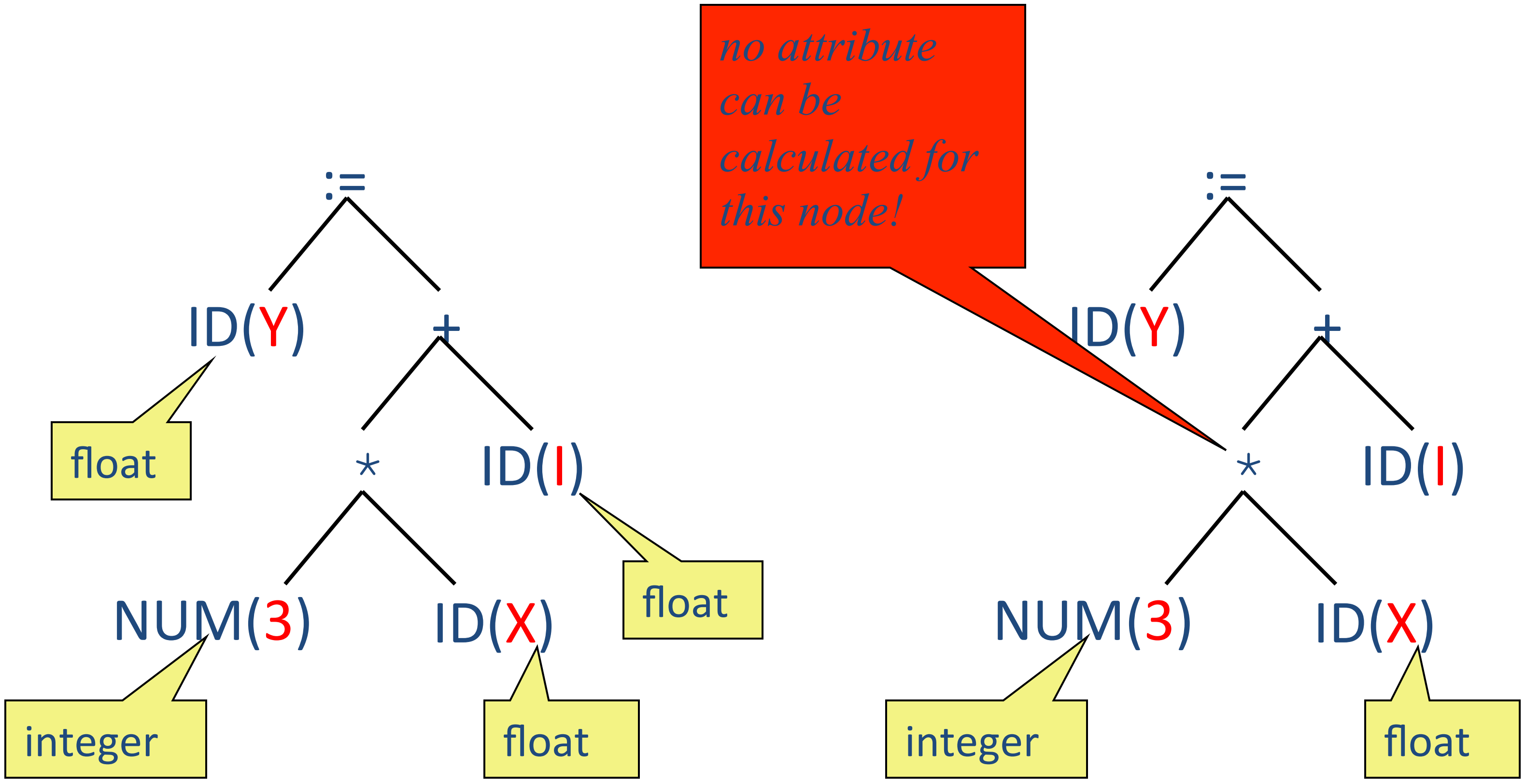


Answer: it depends on the language definition

- ML, Java, etc: no implicit coercion
- C, Basic, Scheme would allow

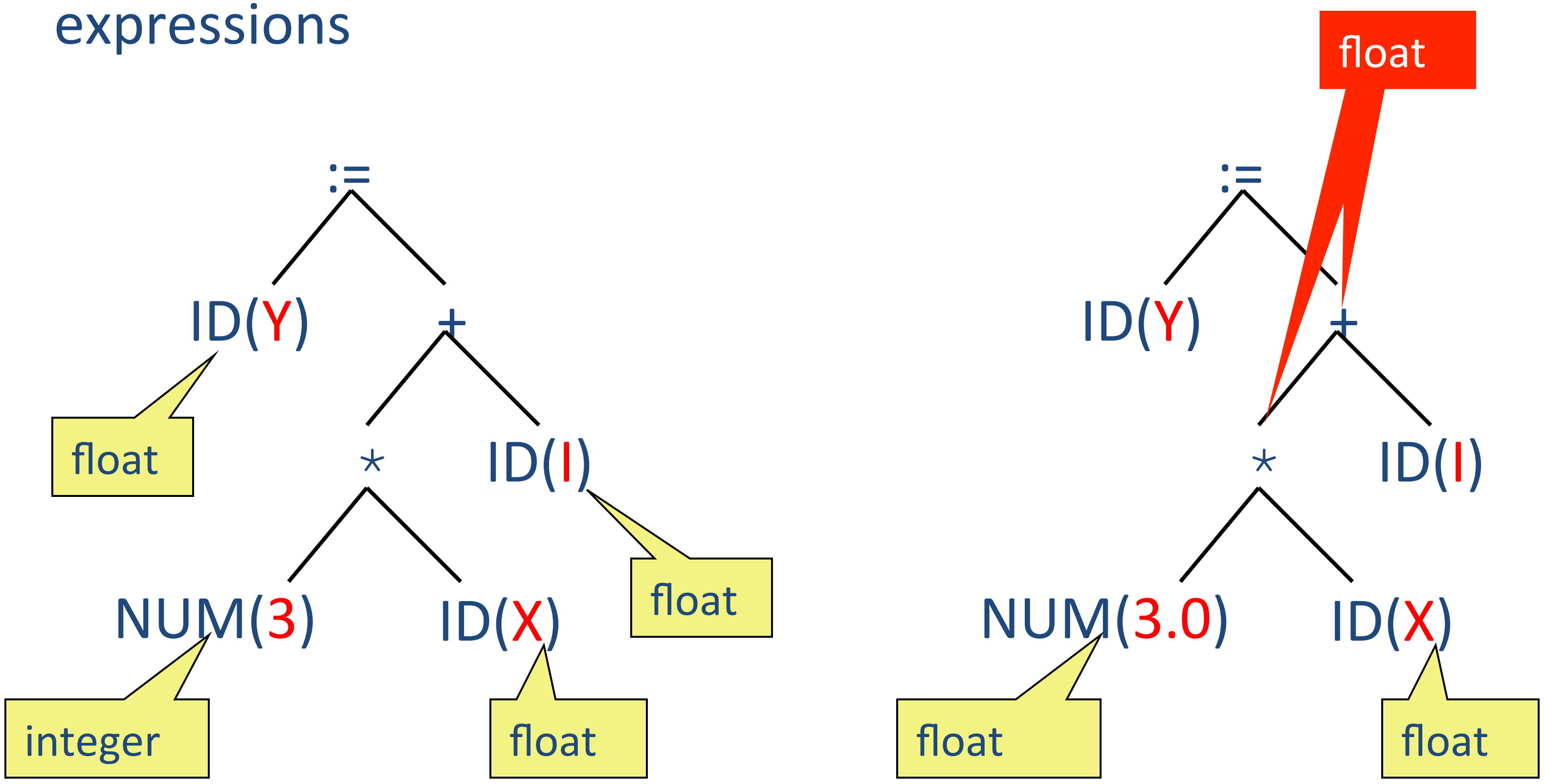
Attribute ASTs and static checking

first case: it's illegal

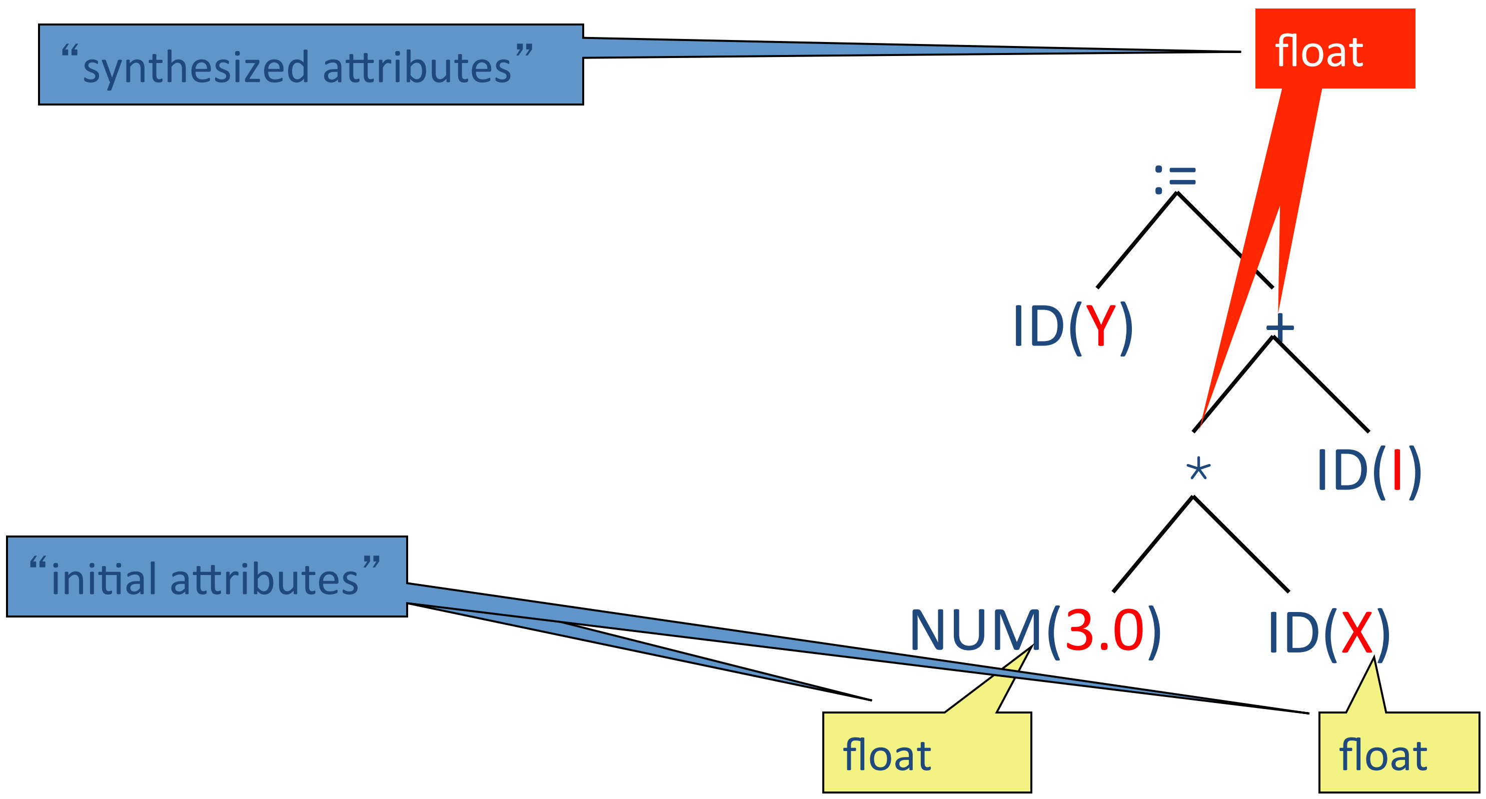


Attribute ASTs and static checking

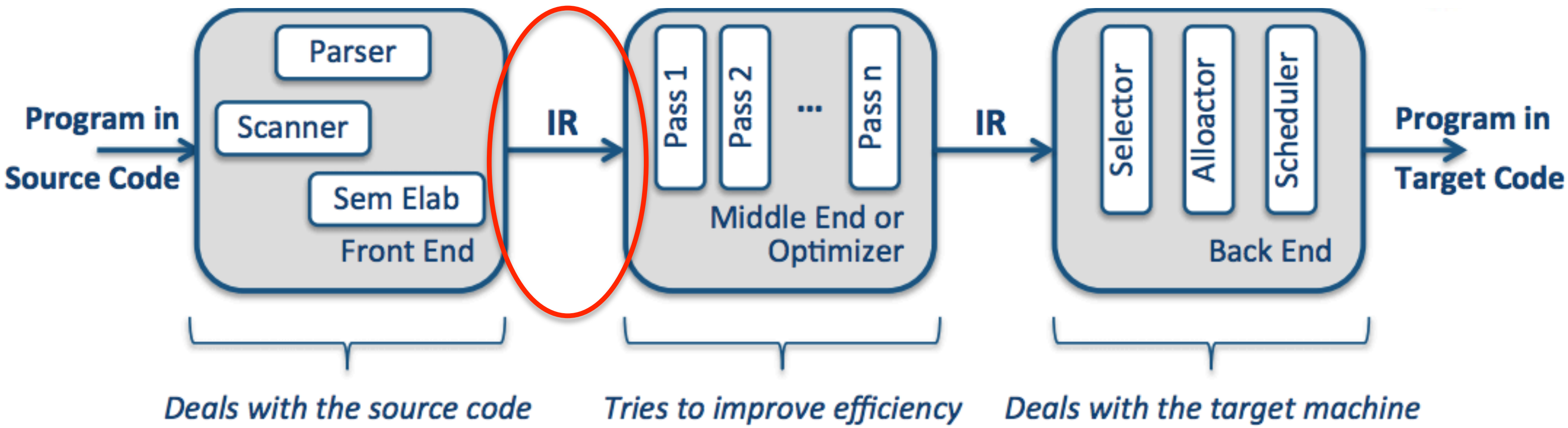
second case: implicitly coerce the constant so that it makes sense; calculate the types of the intermediate expressions



Attribute ASTs and static checking

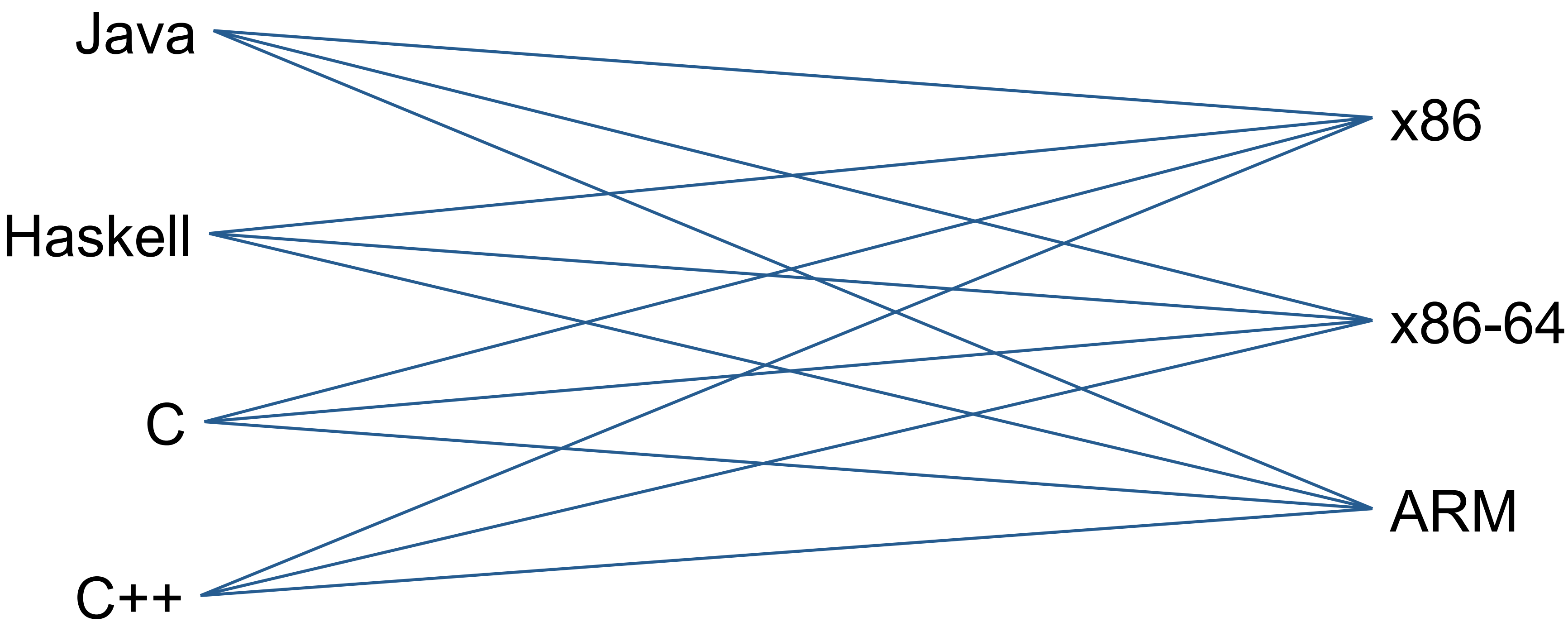


Intermediate Representations

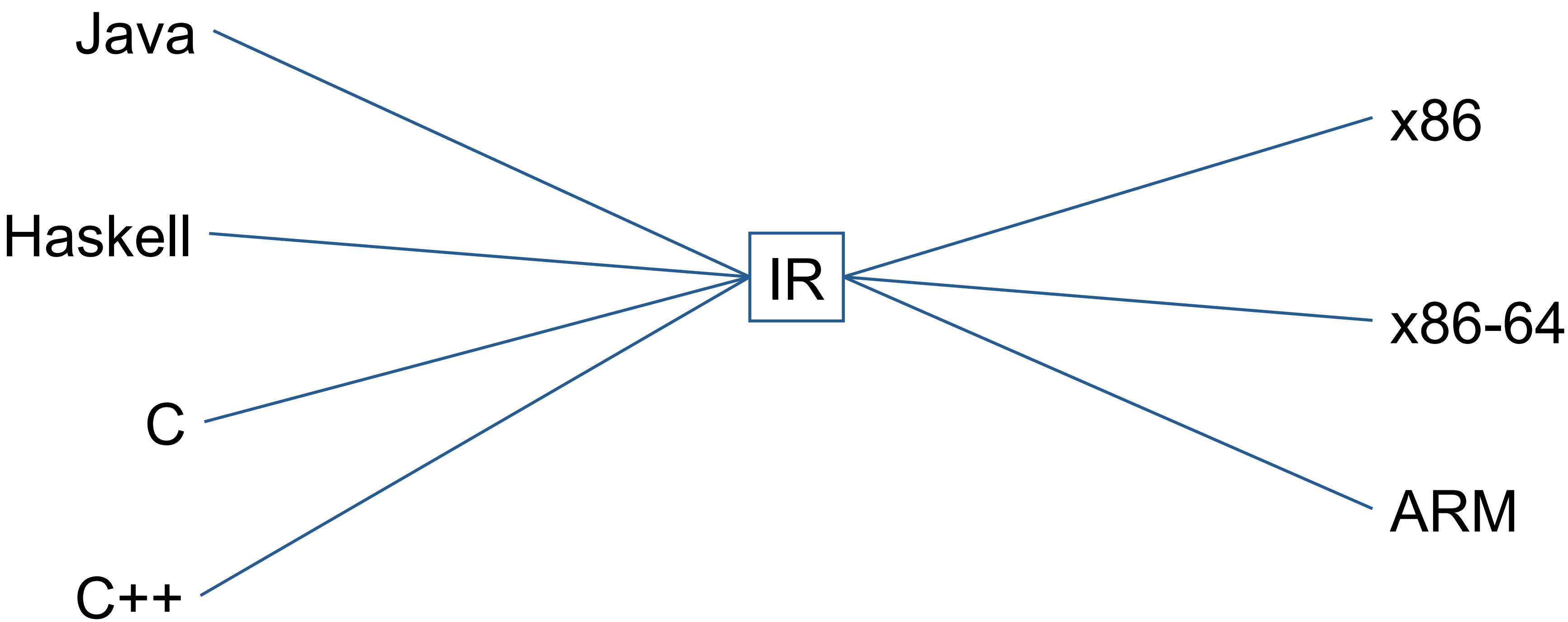


*graphic © Keith Cooper & Linda Torczon.

Intermediate Representations



Intermediate Representations



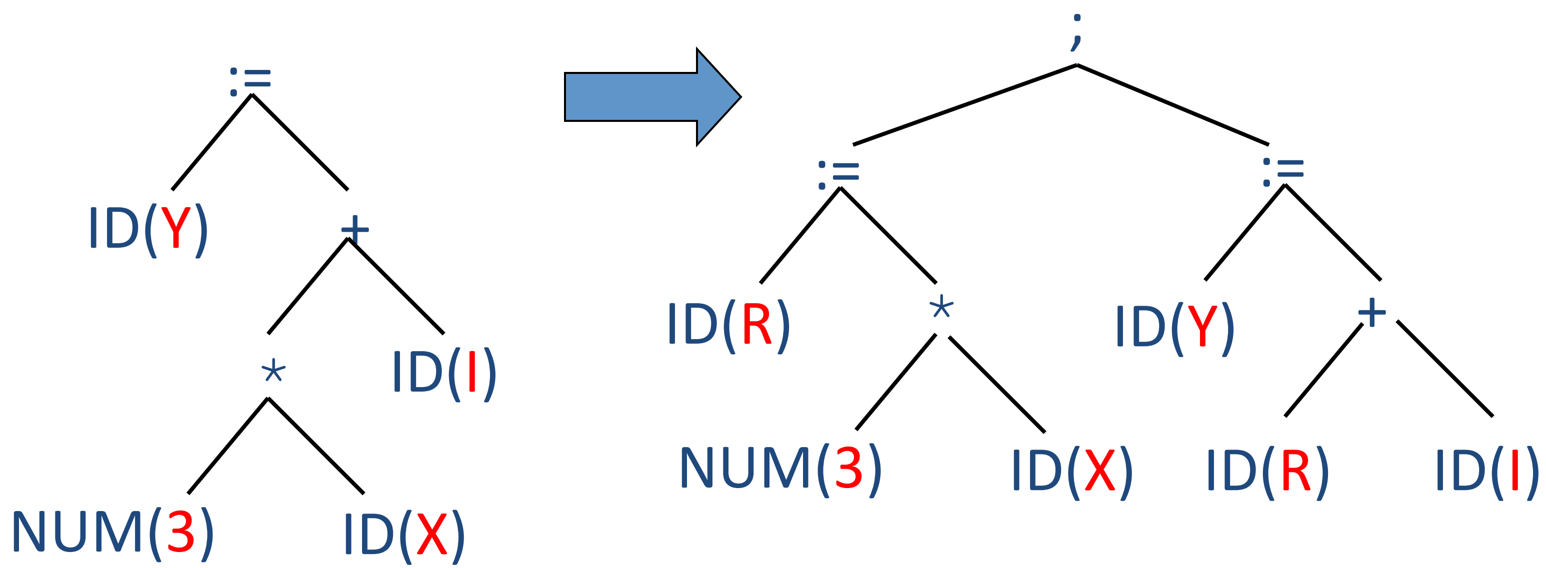
E.g., LLVM

Intermediate Representations

- A.k.a., "IR" or "Intermediate Code"
- Varieties of IR
  - abstract syntax trees
    - written in a particular style to resemble target code
  - three-address code
    - a.k.a. register transfer language (RTL)
  - "Enriched" forms: IR annotated with useful information
    - def-use, use-def chains: connects definition and use of variables
    - Single Static Assignment form (SSA)
- Many compilers use multiple forms

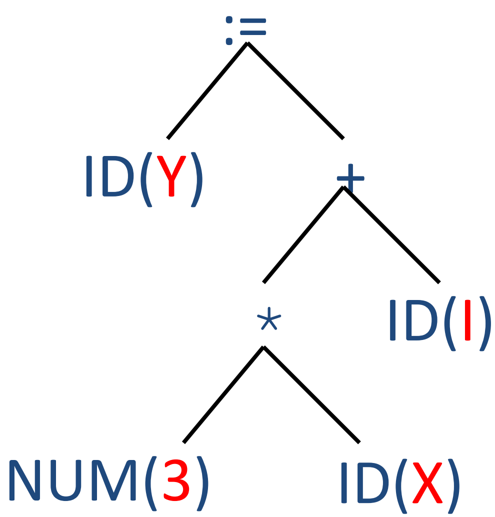
ASTs as IR

Problem: ASTs are generally too high-level and source-language-specific.
Idea: use a lower-level IR.



** May involve introduction of new temporaries like* R

three-address code/RTL



3-address/RTL representation

R ← 3 * X;
Y ← R + X;

Advantage: RTL enforces simplicity of expressions
Disadvantage: not as flexible

Static Single-Assignment (SSA)

Invariant on IR

- Every virtual register has **one** (static) definition site
- **Never** re-assign a virtual register.

This is straightforward for straight-line code.

$$\begin{array}{l} a \leftarrow x * y \\ b \leftarrow a - 1 \\ a \leftarrow y * b \\ b \leftarrow x * 4 \\ a \leftarrow a + b \end{array}$$

$$\begin{array}{l} a_1 \leftarrow x * y \\ b_1 \leftarrow a_1 - 1 \\ a_2 \leftarrow y * b_1 \\ b_2 \leftarrow x * 4 \\ a_3 \leftarrow a_2 + b_2 \end{array}$$