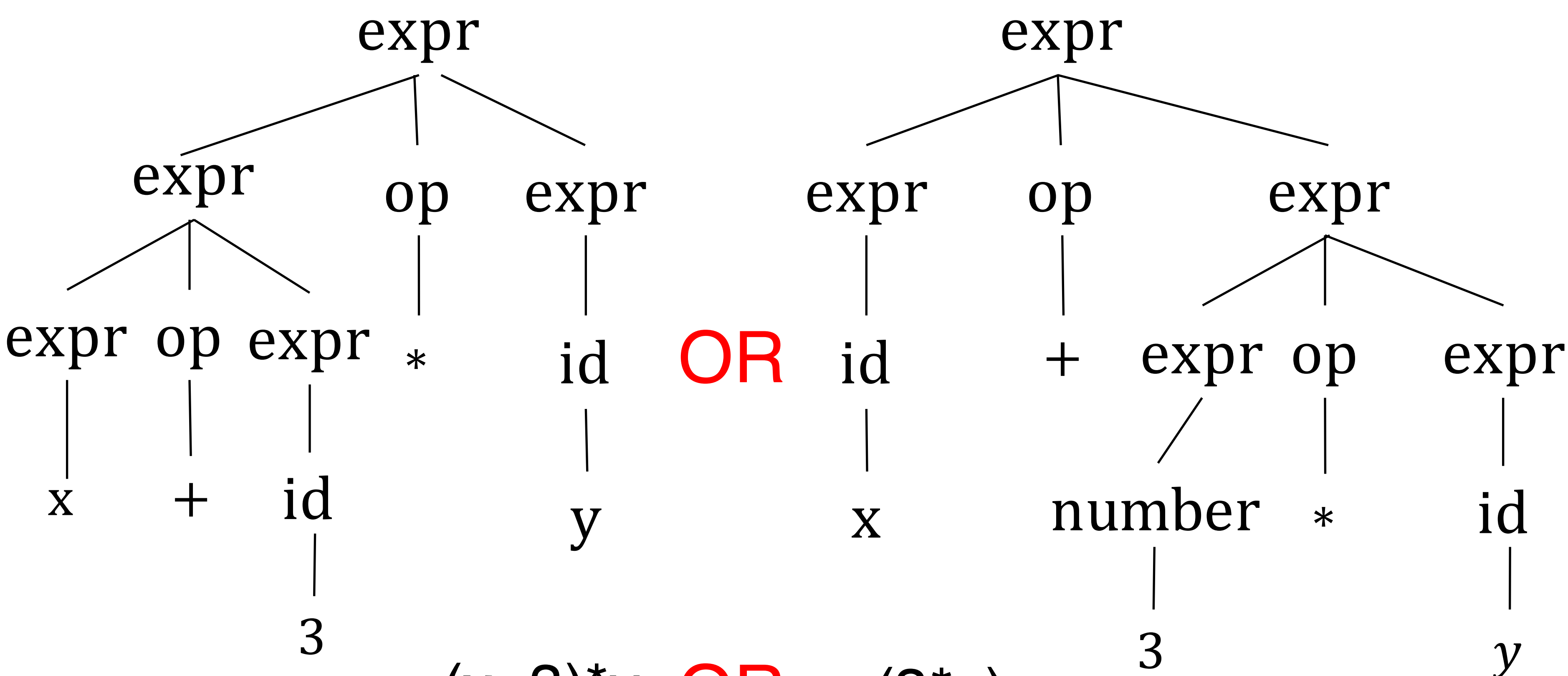
Syntax

CMPSC 461
Programming Language Concepts
Penn State University
Fall 2016

Ambiguity



(x+3)*y OR x+(3*y)

String with different parse tree
might have different meanings

Defining Precedence in Grammar

Define operations at different "levels"

$$\begin{aligned} \text{expr} &\rightarrow \text{id} \mid \text{number} \mid - \text{ expr} \\ &\quad \mid (\text{expr}) \mid \text{expr op expr} \\ \text{op} &\rightarrow + \mid - \mid * \mid / \end{aligned}$$

$$\begin{aligned} \text{expr} &\rightarrow \text{expr} + \text{expr} \mid \text{expr} - \text{expr} \mid \text{term} \\ \text{term} &\rightarrow \text{term} * \text{term} \mid \text{term} / \text{term} \mid \text{factor} \\ \text{factor} &\rightarrow \text{id} \mid \text{number} \mid (\text{expr}) \mid - \text{factor} \end{aligned}$$

Level1

Level2

Level3

The farther from start symbol, the higher precedence

Associativity of Operators

… + a + ...: sign “+” is left-associative since
a is associated with the left “+”

An operator with left (right) associativity
is evaluated left-to-right (right-to-left)

Left: +,-,*,/
Right: = in C (a=b=c same as a=(b=c))

Defining Associativity in Grammar

Left-recursive: LHS is the start of RHS in a production

$$E_1 \rightarrow E_1 \ \dots \ E_n,$$

we say this rule is left-recursive

Right-recursive: LHS is the end of RHS in a production

$$E_n \rightarrow E_1 \ \dots E_n$$

we say this rule is right-recursive

$$\text{expr} \rightarrow \text{expr} + \text{expr} \mid \text{expr} - \text{expr} \mid \text{term}$$

$$\text{term} \rightarrow \text{term} * \text{term} \mid \text{term} / \text{term} \mid \text{factor}$$

$$\text{factor} \rightarrow \text{id} \mid \text{number} \mid (\text{expr}) \mid - \text{factor}$$

The production rule of + is both left- and right-recursive.
So the grammar is ambiguous.

Defining Associativity in Grammar

$$\text{expr} \rightarrow \text{expr} + \text{expr} \mid \text{expr} - \text{expr} \mid \text{term}$$
$$\text{term} \rightarrow \text{term} * \text{term} \mid \text{term} / \text{term} \mid \text{factor}$$
$$\text{factor} \rightarrow \text{id} \mid \text{number} \mid (\text{expr}) \mid - \text{expr}$$

Remove right-recursion

$$\text{expr} \rightarrow \text{expr} + \text{term} \mid \text{expr} - \text{term} \mid \text{term}$$
$$\text{term} \rightarrow \text{term} * \text{factor} \mid \text{term} / \text{factor} \mid \text{factor}$$
$$\text{factor} \rightarrow \text{id} \mid \text{number} \mid (\text{expr}) \mid - \text{factor}$$

Indirect recursion: $\text{factor} \rightarrow - \text{factor}$ → negation is right-associative

Defining Associativity in Grammar

Left-recursive: LHS is the start of RHS in a derivation

$$\mathrm{E}_1 \to^* \mathrm{E}_1 \ \mathrm{op} \ \mathrm{E}_2$$

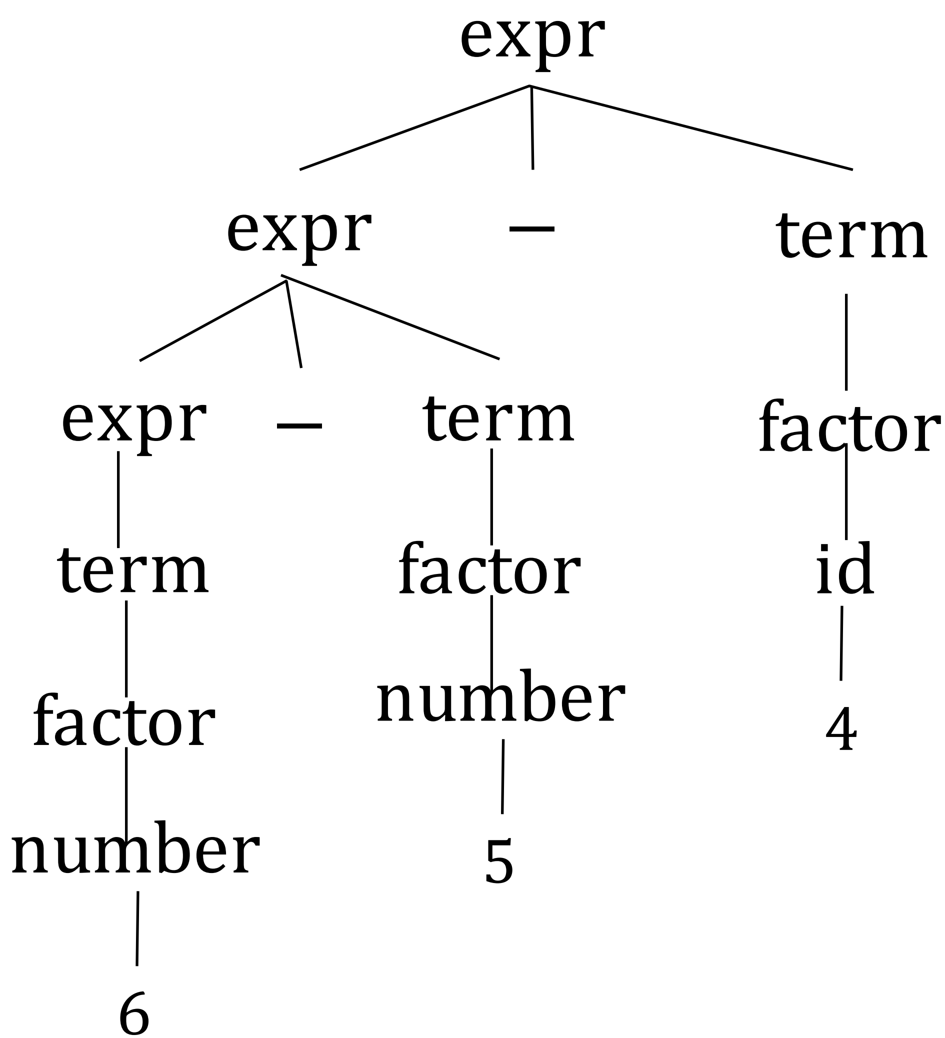
Defines left-associativity

Right-recursive: LHS is the end of RHS in a derivation

$$\mathrm{E}_2 \to^* \mathrm{E}_1 \ \mathrm{op} \ \mathrm{E}_2$$

Defines right-associativity

Recursion can be direct and indirect



(6-5)-4

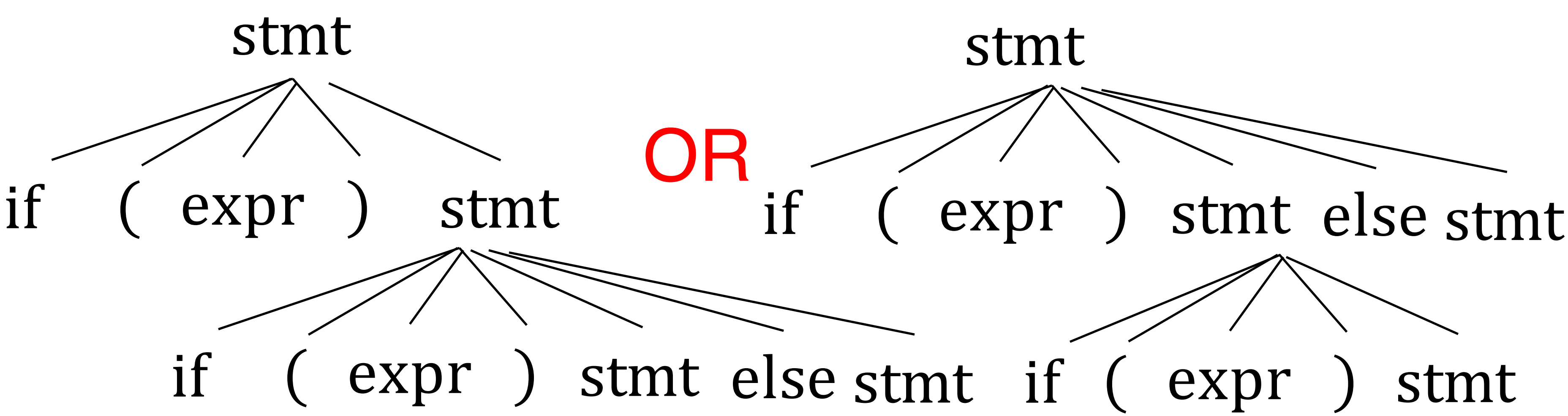
Only one parse tree for 6-5-4

Dangling Else Ambiguity

```
if (b1)
   if (b2) c1
   else c2
```

OR

```
if (b1)
   if (b2) c1
else c2
```



Grammar: stmt → if (expr) stmt | if (expr) stmt else stmt

Avoid Dangling Else

Approach 1: Change syntax (ALGOL, Ada)

Grammar: $\text{stmt} \rightarrow \text{if}\ (\text{expr})\ \text{stmt fi} \mid \text{if}\ (\text{expr})\ \text{stmt else stmt fi}$

```
if (b1)
   if (b2) c1
   else c2
   fi
fi
```

```
if (b1)
   if (b2) c1
   fi
else c2
fi
```

Avoid Dangling Else

Approach 2: Keep syntax, change grammar

Grammar: stmt → matched | unmatched

matched → if (expr) matched else matched

unmatched → if (expr) stmt

| if (expr) matched else unmatched

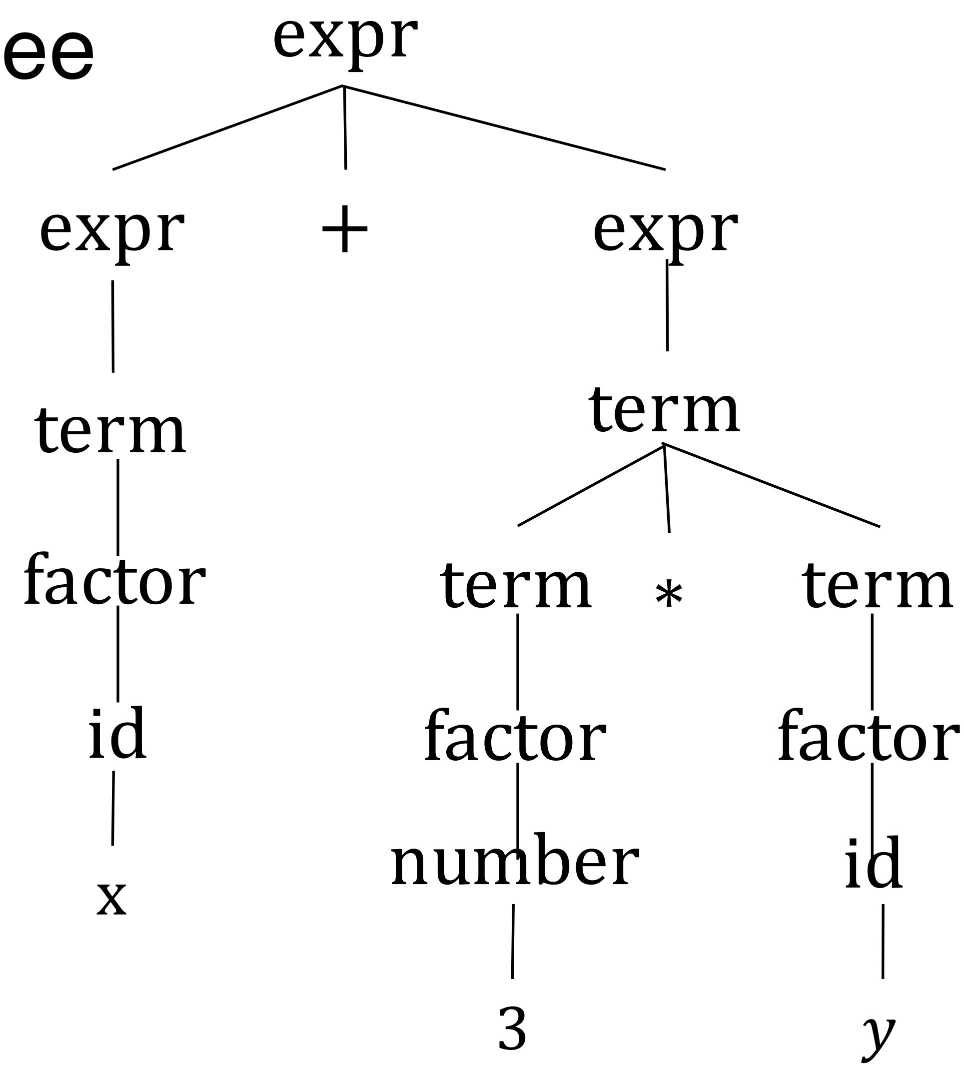
```
if (b1)
   if (b2) c1
   else c2
```

✓

```
if (b1)
   if (b2) c1
else c2
```

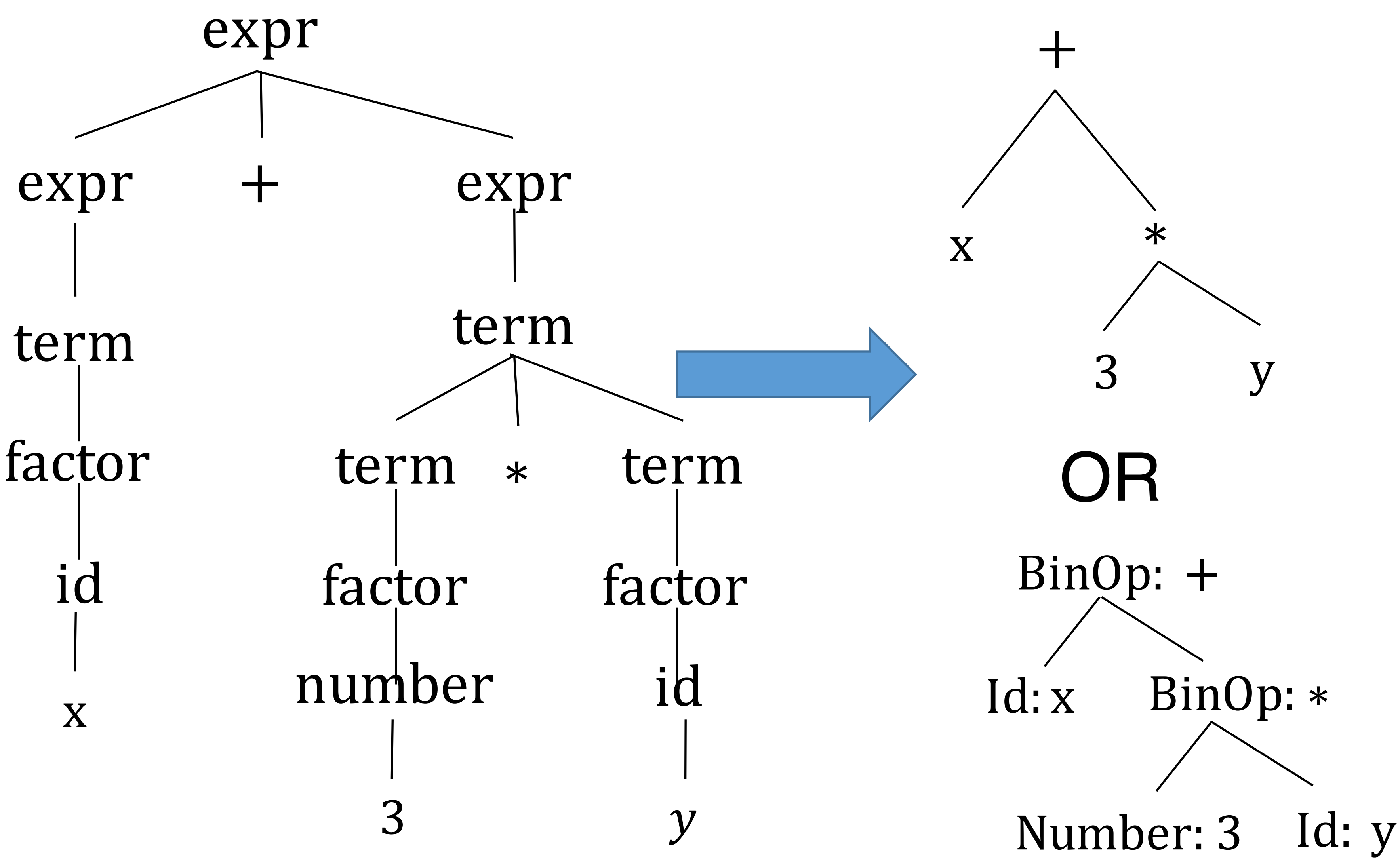
✗

Concrete Syntax Tree

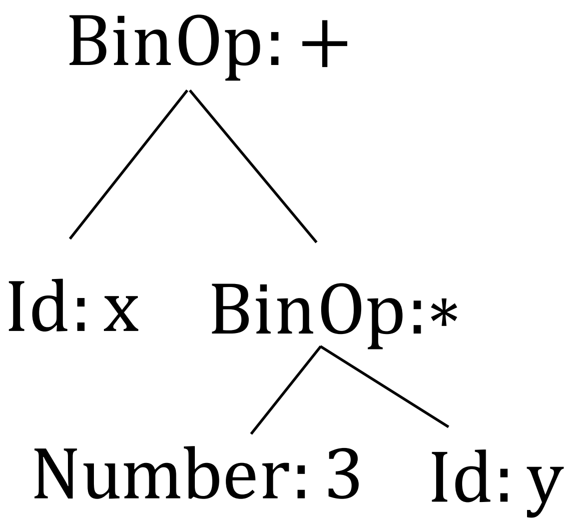


The parse tree is very verbose. It tracks information only useful for the tree construction.

Abstract Syntax Tree (AST)



Abstract Syntax Tree (AST)



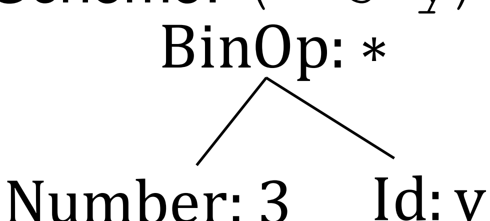
AST doesn’t show the whole syntactic clutter,
e.g., parentheses, nonterminals added for unambiguity

AST keeps the same structure as parse tree

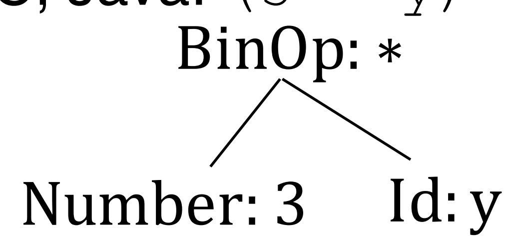
Each node usually corresponds to an object in impl.
(more manageable for later compiler stages)

AST hides syntactical language differences

Scheme: `(* 3 y)`



C, Java: `(3 * y)`



Avoid ambiguity

- Define precedence (grammar with different levels of operators)
- Define associativity (left-recursive vs. right-recursive derivations)

Abstract Syntax Tree (AST)

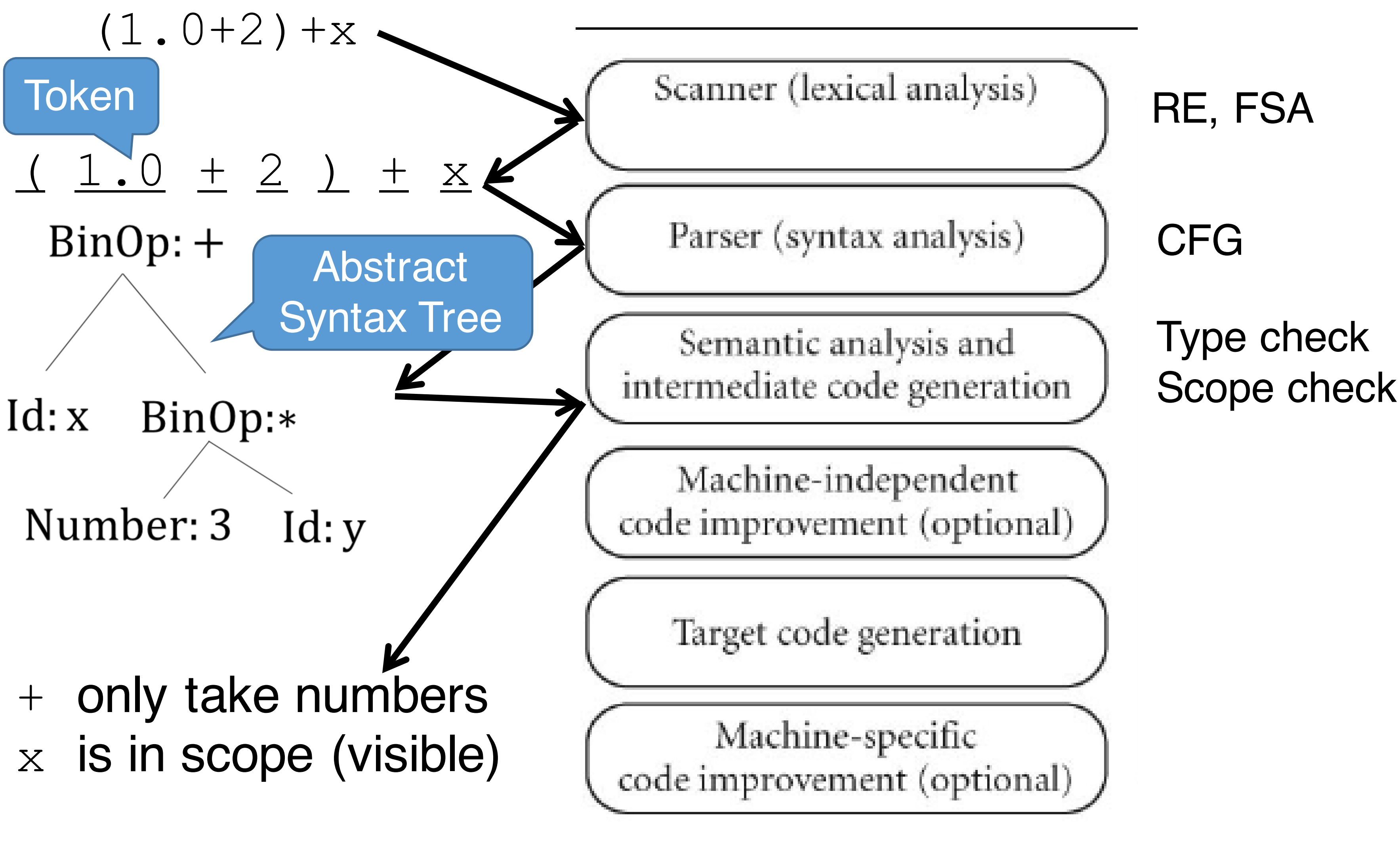
Parsing

Assemble tokens (from scanner) to a syntax tree, according to a CFG

For any CFG, a parser runs in $O(n^3)$ time exists, where n is the length of program

In practice, most programming languages falls in CFG with linear time parser (e.g., LL, LR)

Parser generator (e.g., Yacc, Bison) automatically generates parser from CFG


(1.0+2)+x
Token
(1.0 + 2) + x
BinOp: +
Abstract Syntax Tree
Id: x
BinOp:*
Number: 3
Id: y
+ only take numbers
x is in scope (visible)
Scanner (lexical analysis)
Parser (syntax analysis)
Semantic analysis and intermediate code generation
Machine-independent code improvement (optional)
Target code generation
Machine-specific code improvement (optional)
RE, FSA
CFG
Type check
Scope check