


# Principles of Programming Languages

## Syntax Analysis

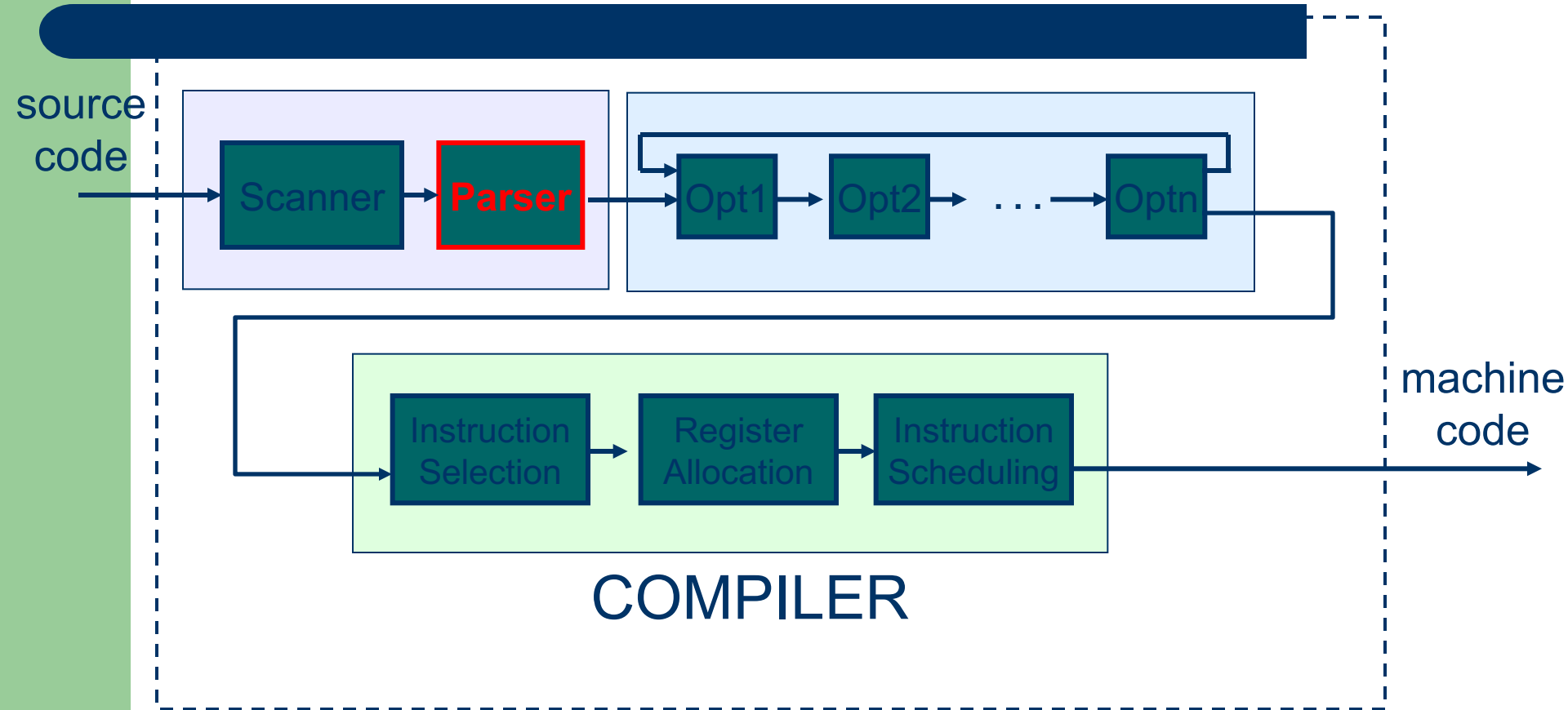
Quan Thanh Tho, Ph.D.  
CSE – HCMUT  
qttho@hcmut.edu.vn



# Outline

- Grammar
  - Context-free grammar
  - Derivation and Derivation Tree
- Grammar for Arithmetic Expression
  - Operation precedence and associativity
- Syntax Analysis
- Ambiguity in Grammar
- Parser Construction

# The Big Picture Again



# Syntax and Grammar

- Syntax (programming language sense):
  - Define **structure** of a program
  - **Not reflect** the meaning (semantic) of the program
- Grammar:
  - **Rule-based formalism to specify a language syntax**

# Context-Free Grammar (CFG)

- A kind of grammar
- **Not as complex as** context-sensitive and phase-structure grammar
- **More powerful** than regular grammar

# Formal Definition of CFG

- $G = (V_N, V_T, S, P)$
- $V_N$ : finite set of **nonterminal symbols**
- $V_T$ : finite set of **tokens** ( $V_T \cap V_N = \emptyset$ )
- $S \in V_N$ : **start symbol**
- $P$ : finite set of **rules** (or **productions**) of BNF (Backus – Naur Form) form  **$A \rightarrow (a)^*$**  where  $A \in V_N, a \in (V_T \cup V_N)$

# Example 1

- $G = (\{\text{exp}, \text{op}\}, \{+, -, *, /, \text{id}\}, \text{exp})$
- $\text{exp} \rightarrow \text{exp op exp}$
- $\text{exp} \rightarrow \text{id}$
- $\text{op} \rightarrow +|-|*|/$

# Derivation

- $\alpha = uXv$  derives  $\beta = u\gamma v$  if  $X \rightarrow \gamma$  is a **production**

Notation:  $\alpha \Rightarrow \beta$  (directly derive)

$$\alpha \Rightarrow^* \beta \quad (\alpha \Rightarrow \dots \Rightarrow \beta \mid \alpha = \beta)$$

$$\alpha \Rightarrow^+ \beta$$

**Derivations:**  $S \Rightarrow^+ \alpha$  where  $\alpha$  consists of tokens only.

**Sentential form:**  $S \Rightarrow^+ \alpha \Leftrightarrow \alpha$  is a sentential form

**Sentence:**  $S \Rightarrow^* \alpha$  is a derivation  $\Leftrightarrow \alpha$  is a sentence

**Language:** set of all sentences possibly derived



# Example 1

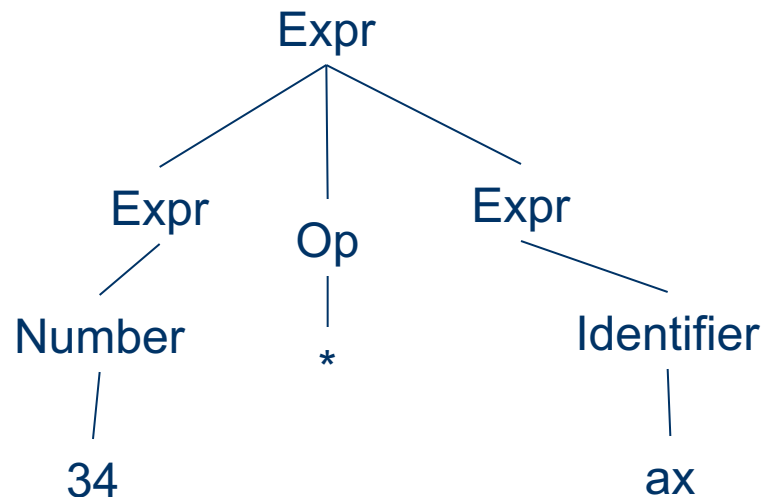
- $\text{exp} \Rightarrow \text{exp op exp} \Rightarrow \text{exp op id} \Rightarrow \text{id op id} \Rightarrow \text{id} + \text{id}$
- $\text{exp} \Rightarrow \text{exp op exp} \Rightarrow \text{id op exp} \Rightarrow \text{id} + \text{exp} \Rightarrow \text{id} + \text{id}$
- $\text{exp} \Rightarrow \text{exp op exp} \Rightarrow \text{exp op exp op exp} \Rightarrow \text{id op exp op exp} \Rightarrow \text{id} + \text{exp op exp} \Rightarrow \text{id} + \text{exp} * \text{exp} \Rightarrow \text{id} + \text{id} * \text{exp} \Rightarrow \text{id} + \text{id} * \text{id}$

## Example 3

- $\text{exp} \Rightarrow \text{exp op exp} \Rightarrow \text{id op exp} \Rightarrow \text{id} + \text{exp}$   
 $\Rightarrow \text{id} + \text{id}$
- $\text{exp} \Rightarrow \text{exp op exp} \Rightarrow \text{exp op id} \Rightarrow \text{exp} + \text{id}$   
 $\Rightarrow \text{id} + \text{id}$

# Derivations as Trees

- Internally, in the parser, derivations are implemented as trees
- A convenient and natural way to represent a sequence of derivations is a **syntactic tree** or **parse tree**
- Example:



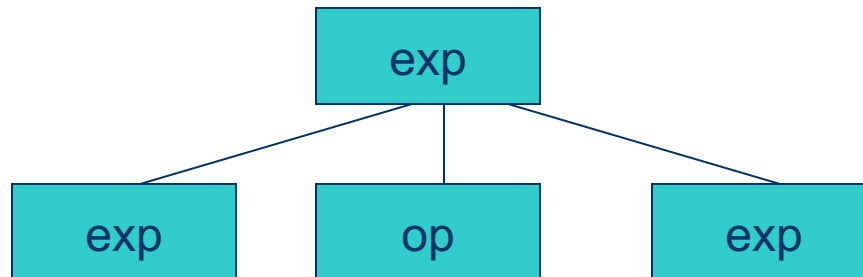
## Example 4

- exp

exp

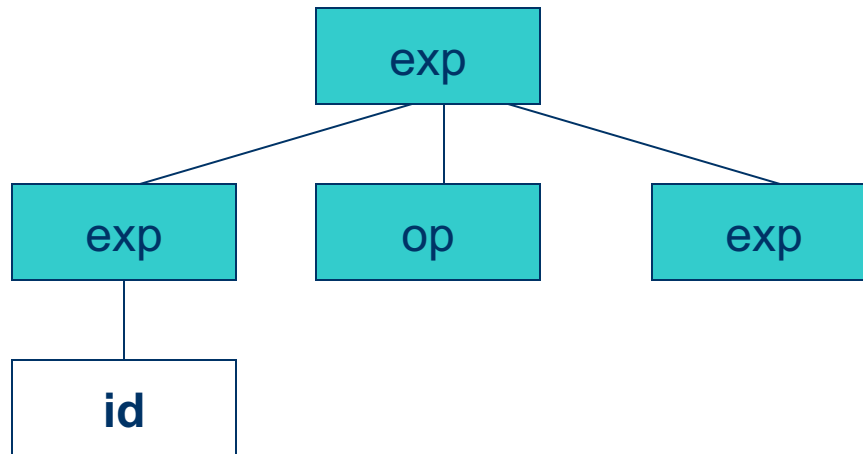
## Example 4

- $\text{exp} \Rightarrow \text{exp op exp}$



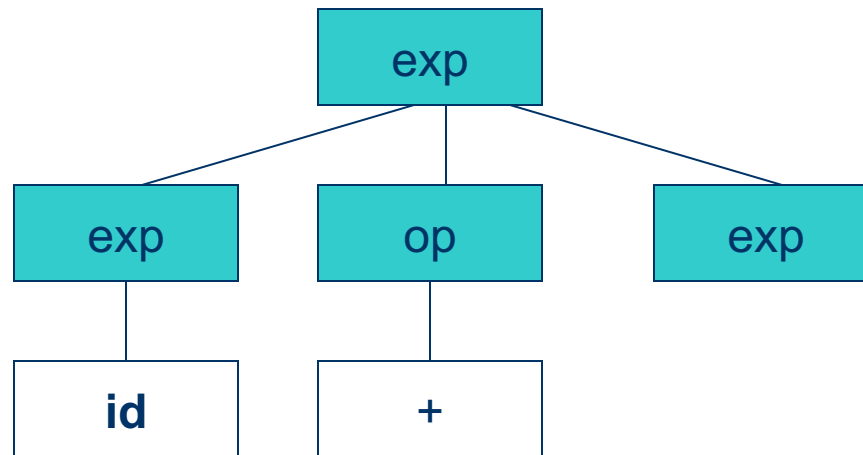
## Example 4

$\text{exp} \Rightarrow \text{exp op exp} \Rightarrow \mathbf{id} \text{ op exp}$



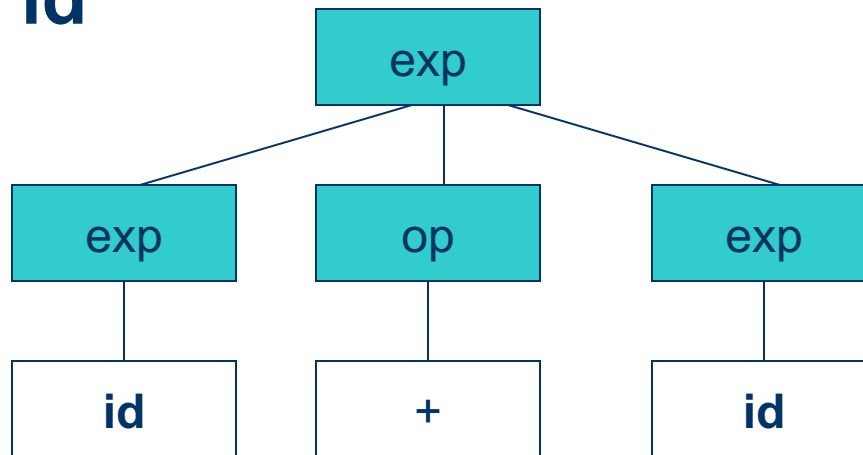
## Example 4

- $\text{exp} \Rightarrow \text{exp op exp} \Rightarrow \mathbf{id} \text{ op exp} \Rightarrow \mathbf{id} + \text{exp}$



## Example 4

- $\text{exp} \Rightarrow \text{exp op exp} \Rightarrow \mathbf{id} \text{ op exp} \Rightarrow \mathbf{id} + \text{exp}$   
 $\Rightarrow \mathbf{id} + \mathbf{id}$





# Classic Expression Grammar

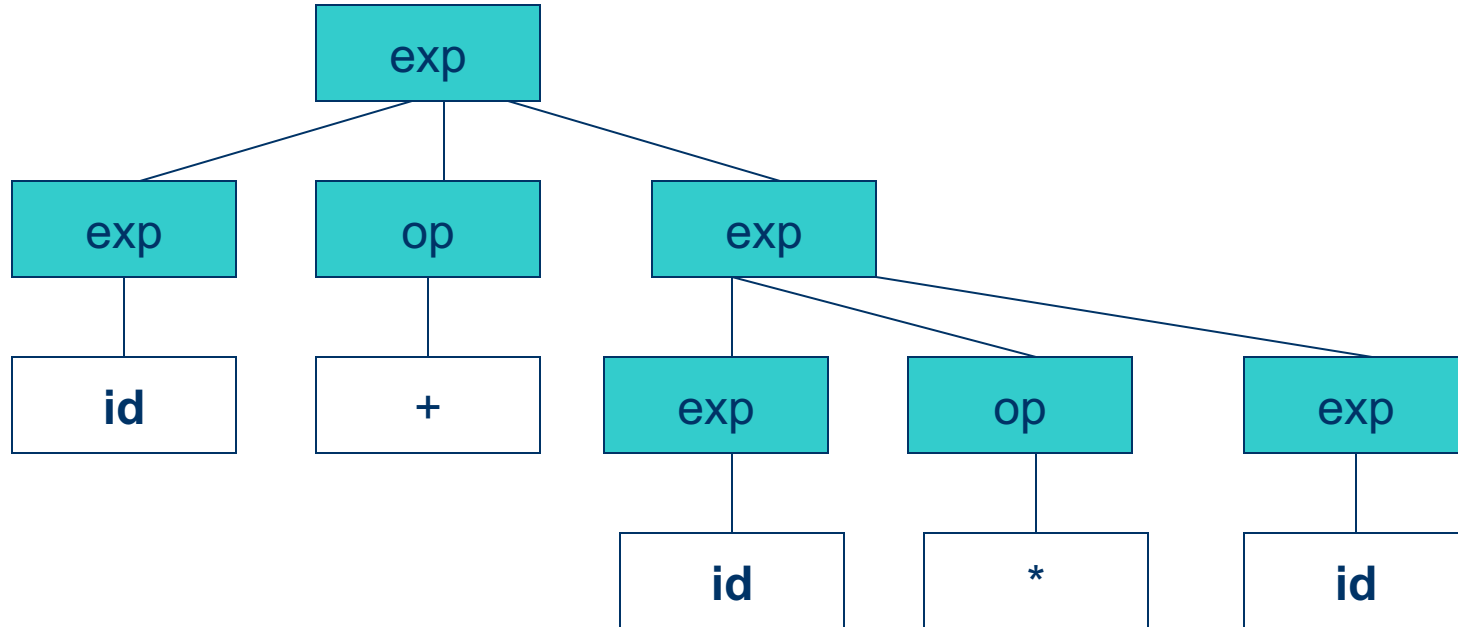
$\text{exp} \rightarrow \text{exp} + \text{term} \mid \text{exp} - \text{term} \mid \text{term}$

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

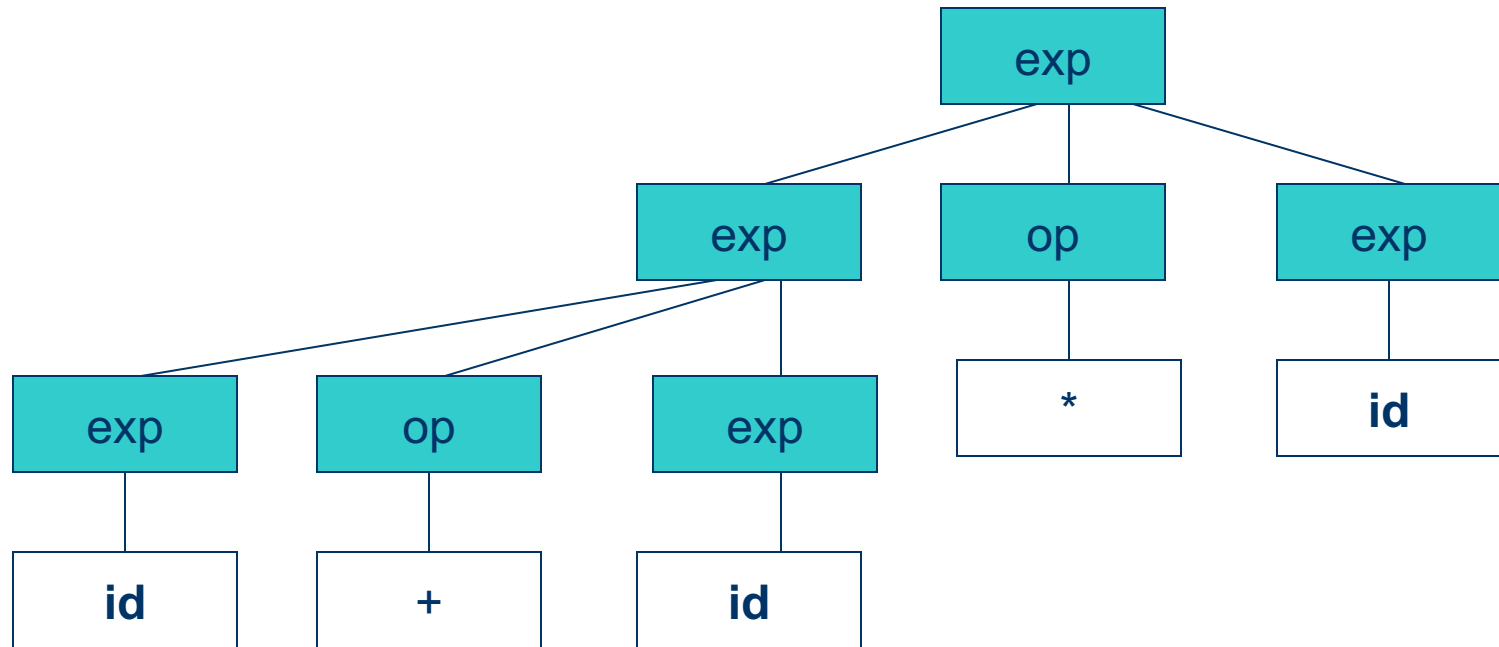
$\text{factor} \rightarrow ( \text{exp} ) \mid \mathbf{ID} \mid \mathbf{INT}$

why is this classic expression grammar better than the previously used one?

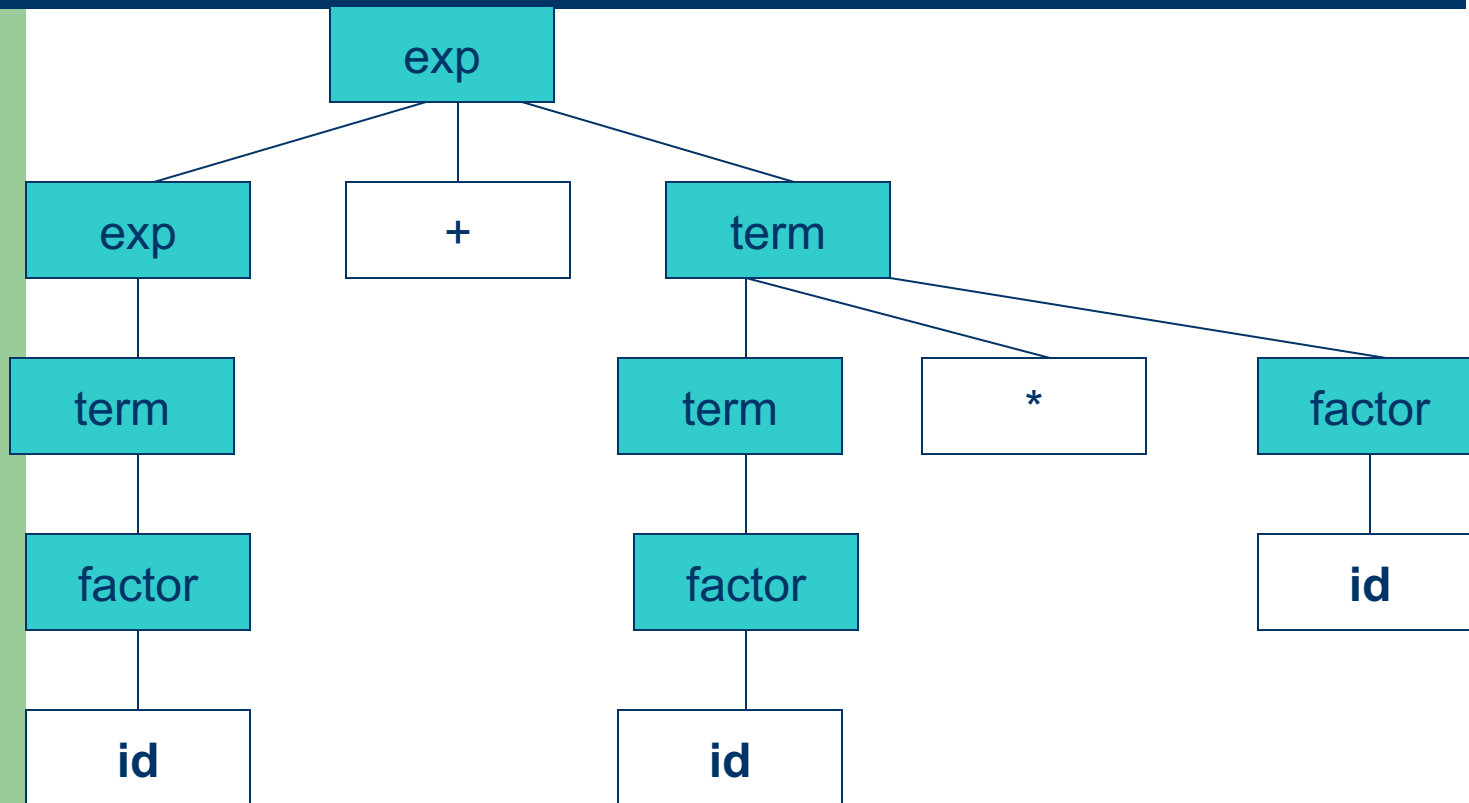
# Operation Precedence



# Operation Precedence

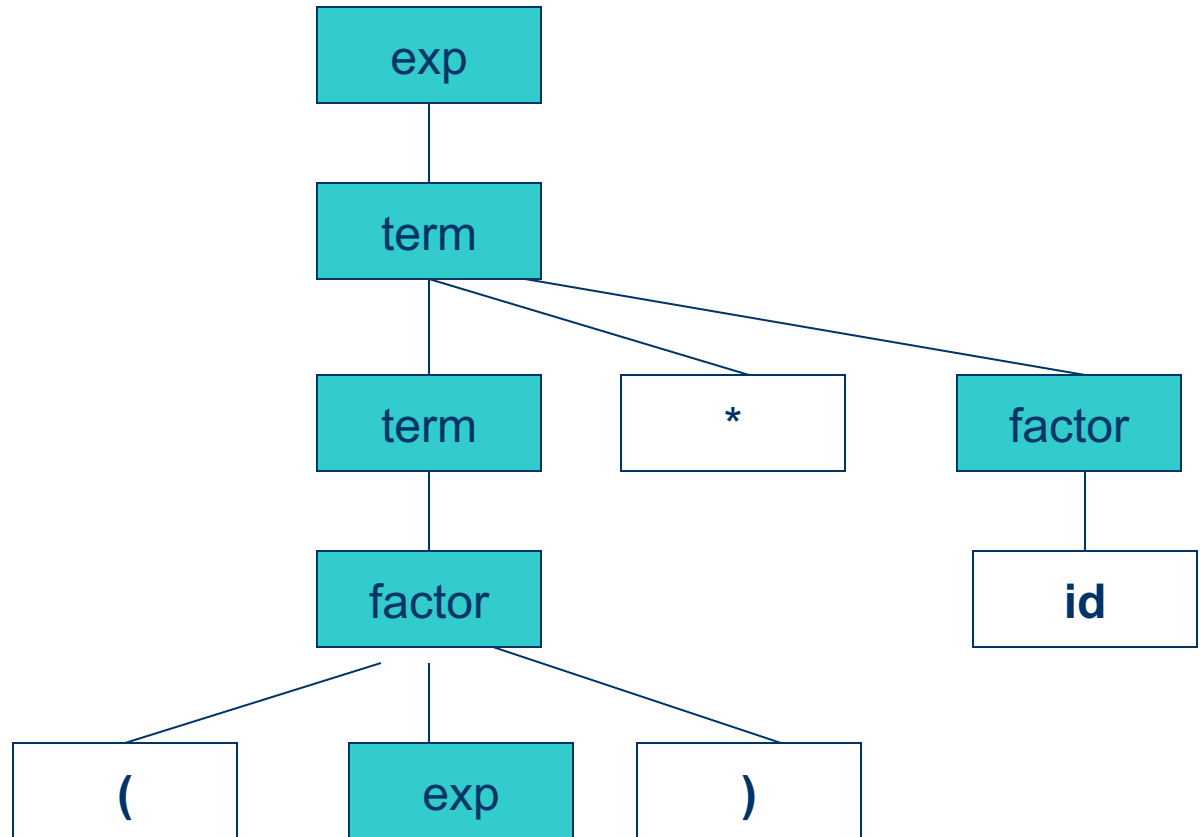


# Operation Precedence

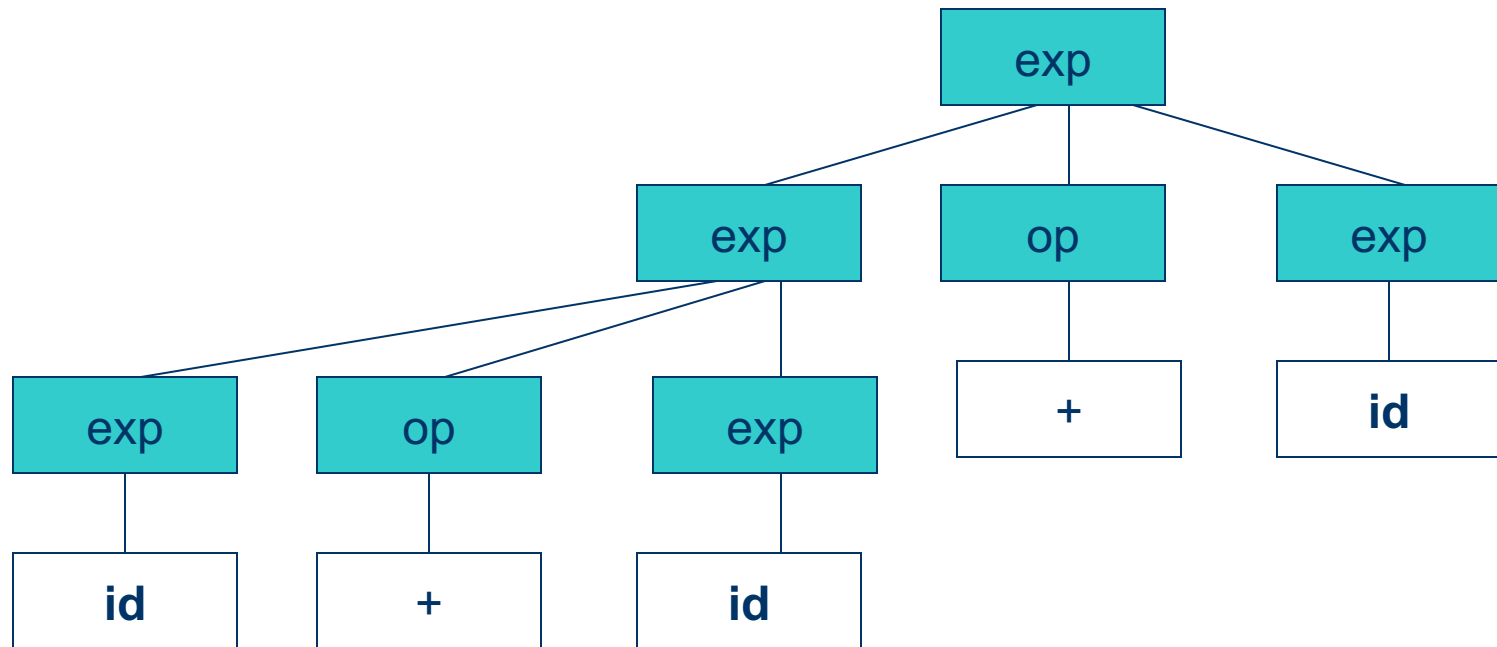


# Operation Precedence

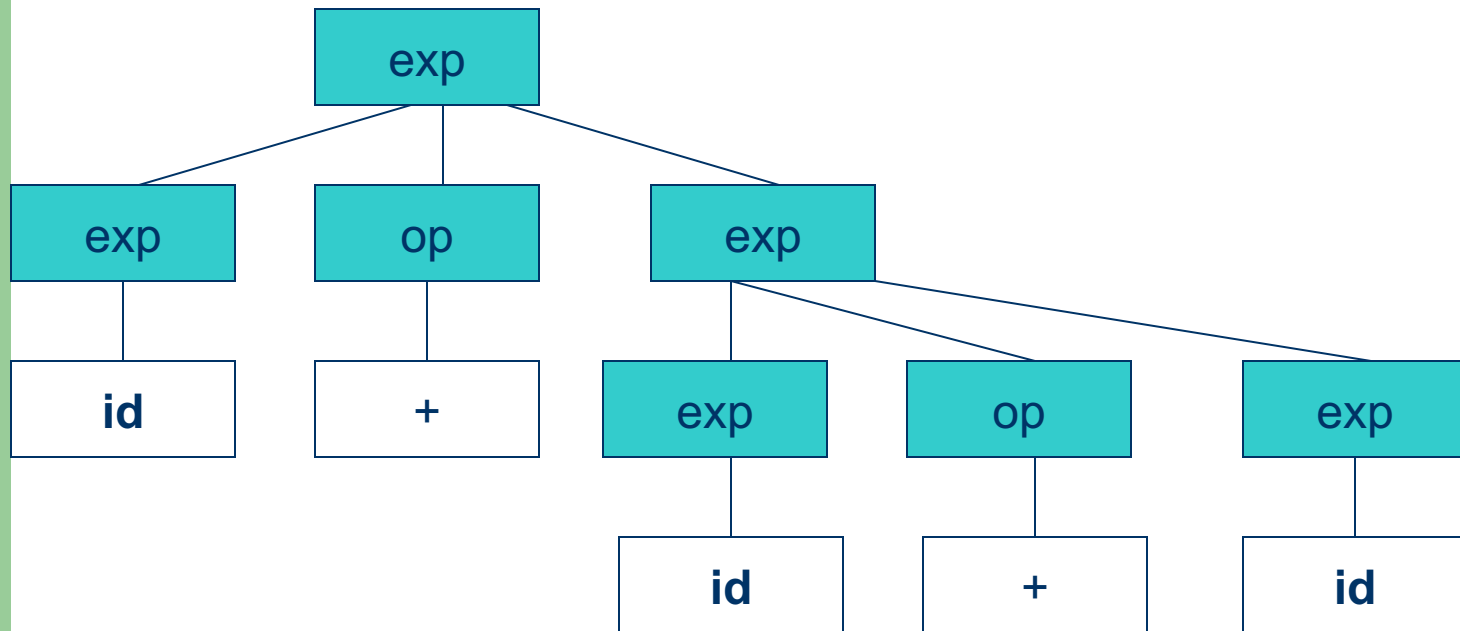
- $(id+id) * id$



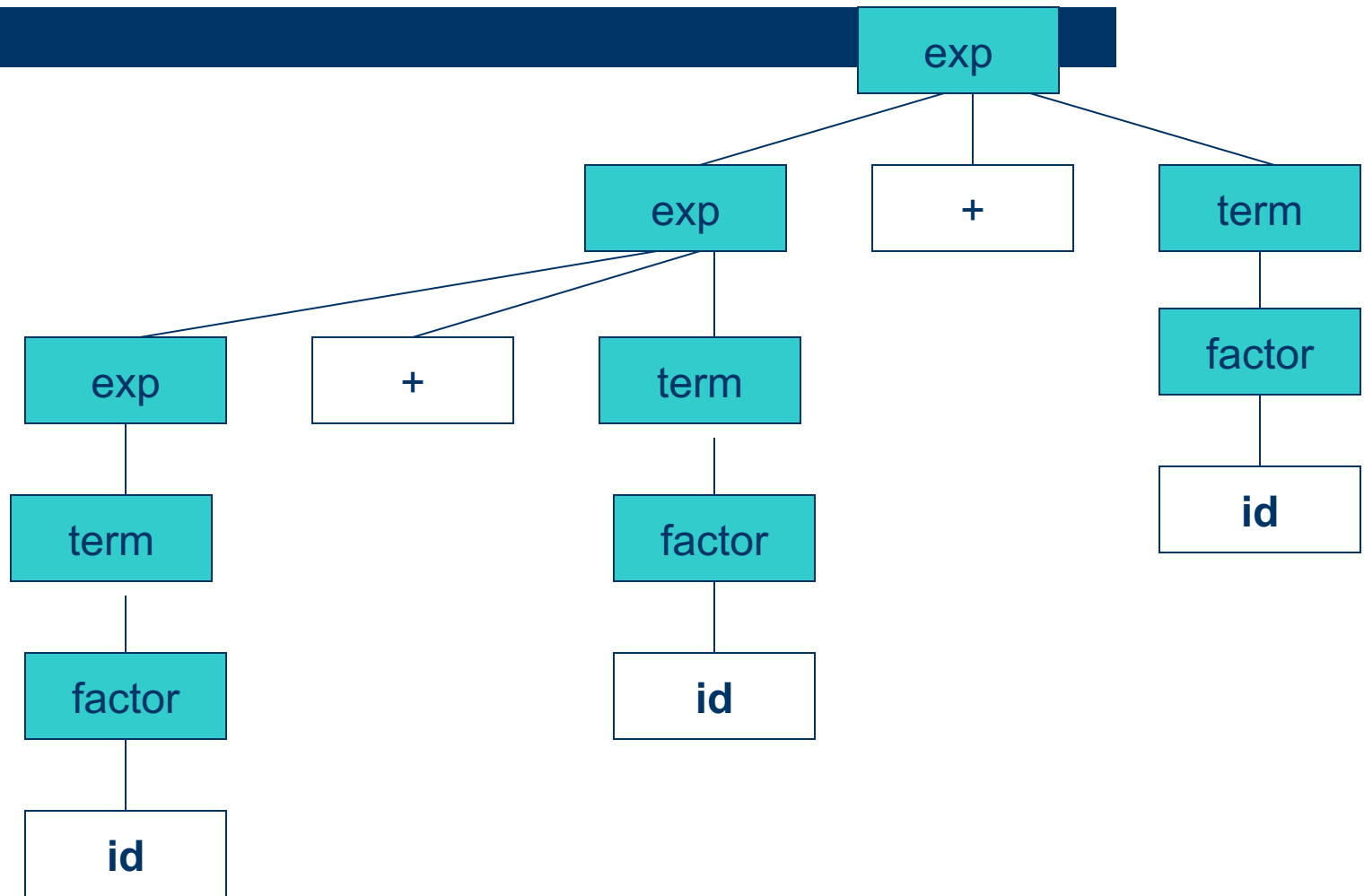
# Operator Associativity



# Operator Associativity



# Operator Associativity





# Precedence and Associativity

- When properly written, a grammar can enforce operator precedence and associativity as desired

# Hands-on Exercise

- Rewrite the grammar to fulfill the following requirements:
  - operator “\*” takes lower precedence than “+”
  - operator “-” is right-associativity

# Syntactic Analysis

- Lexical Analysis was about ensuring that we extract a set of valid **words** (i.e., tokens/lexemes) from the source code
- But nothing says that the words make a coherent **sentence** (i.e., program)

# Syntactic Analysis

- Example:
  - “for while i == == == 12 + for ( abcd)”
  - Lexer will produce a stream of tokens:  
<TOKEN\_FOR> <TOKEN\_WHILE> <TOKEN\_IDENT, “i”>  
<TOKEN\_COMPARE> <TOKEN\_COMPARE> <TOKEN\_COMPARE>  
<TOKEN\_NUMBER, “12”> <TOKEN\_OP, “+”> <TOKEN\_FOR>  
<TOKEN\_OPAREN> <TOKEN\_ID, “abcd”> <TOKEN\_CPAREN>
  - But clearly we do not have a valid program
  - This program is lexically correct, but syntactically incorrect

# A Grammar for Expressions

Expr	→ Expr Op Expr
Expr	→ Number   Identifier
Identifier	→ Letter   Letter Identifier
Letter	→ a-z
Op	→ "+"   "-"   "*"   "/"
Number	→ Digit Number   Digit
Digit	→ 0   1   2   3   4   5   6   7   8   9

# What is Parsing?

- What we just saw is the process of, starting with the start symbol and, through a sequence of rule derivation obtain a string of terminal symbols
  - We could generate all correct programs (infinite set though)
- **Parsing**: the other way around
  - Give a string of non-terminals, the process of discovering a sequence of rule derivations that produce this particular string

# What is parsing

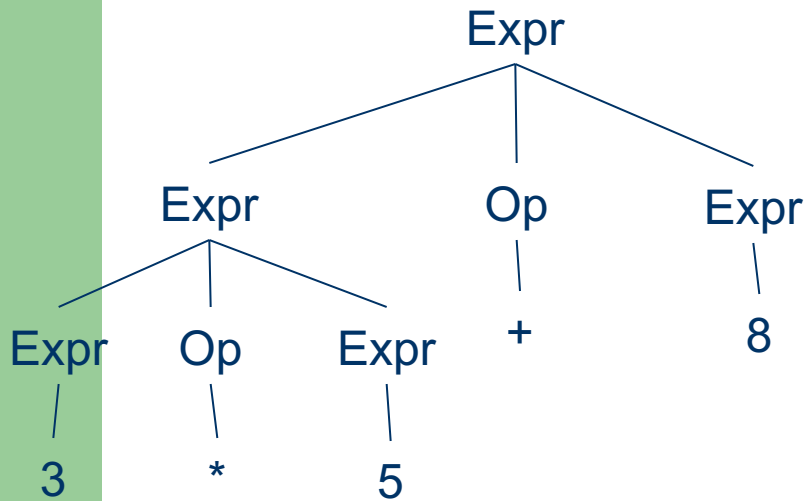
- When we say we can't parse a string, we mean that we can't find any legal way in which the string can be obtained from the start symbol through derivations
- What we want to build is a **parser**: a program that takes in a string of tokens (terminal symbols) and discovers a derivation sequence, thus validating that the input is a syntactically correct program

# Ambiguity

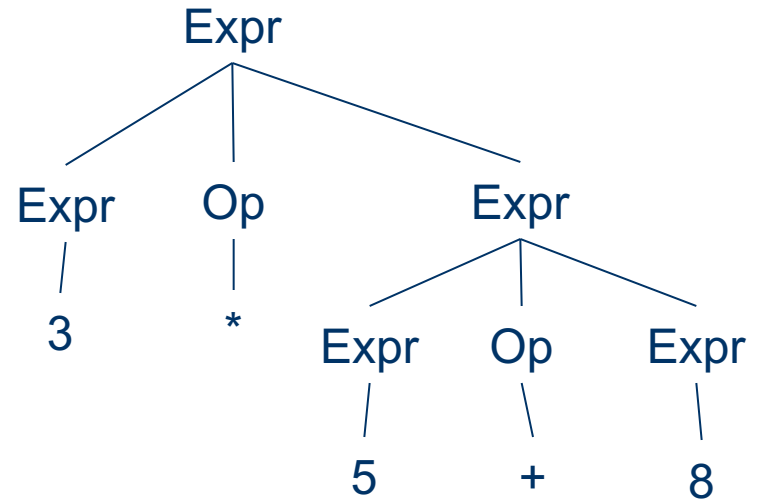
- We call a grammar **ambiguous** if a string of terminal symbols can be reached by two different derivation sequences
- In other terms, a string can have more than one parse tree
- It turns out that our expression grammar is ambiguous!
- Let's show that string  $3*5+8$  has two parse trees



# Ambiguity



“left parse-tree”



“right parse-tree”

# Problems with Ambiguity

- The problem is that the syntax impacts meaning (for the later stages of the compiler)
- For our example string, we'd like to see the left tree because we most likely want \* to have a higher precedence than +
- We don't like ambiguity because it makes the parsers difficult to design because we don't know which parse tree will be discovered when there are multiple possibilities
- So we often want to disambiguate grammars

# Problems with Ambiguity

- It turns out that it is possible to modify grammars to make them non-ambiguous
  - by adding non-terminals
  - by adding/rewriting production rules
- In the case of our expression grammar, we can rewrite the grammar to remove ambiguity and to ensure that parse trees match our notion of operator precedence
  - We get two benefits for the price of one
  - Would work for many operators and many precedence relations

# Non-Ambiguous Grammar

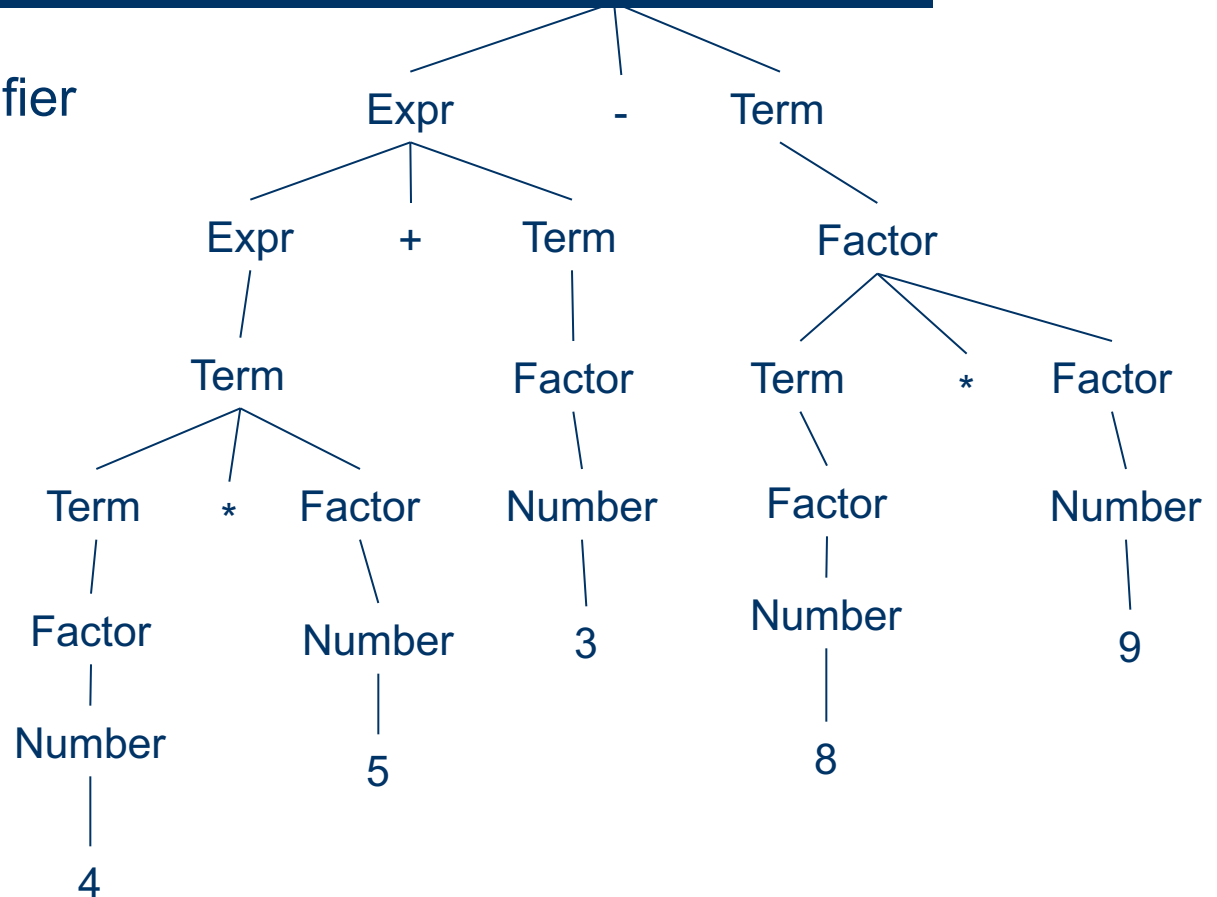
Expr  $\rightarrow$  Term | Expr + Term | Expr - Term

Term  $\rightarrow$  Term \* Factor

Factor

Factor  $\rightarrow$  Number | Identifier

Example:  $4*5+3-8*9$



# Non-Ambiguous Grammar

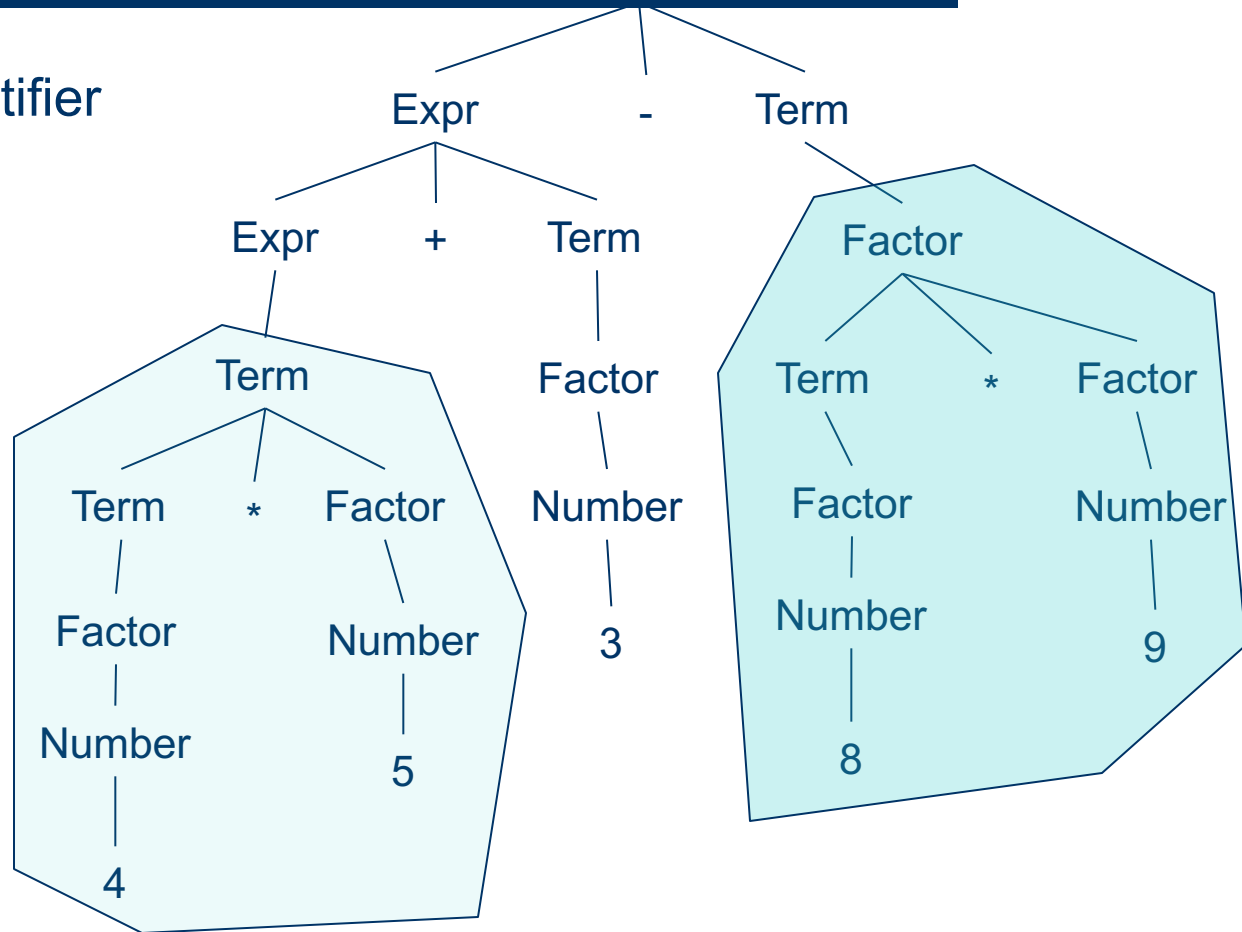
Expr  $\rightarrow$  Term | Expr + Term | Expr - Term

Term  $\rightarrow$  Term \* Factor

Factor

Factor  $\rightarrow$  Number | Identifier

Example:  $4*5+3-8*9$



# In-class Exercise

- Consider the CFG:

$$S \rightarrow (L) \mid a$$
$$L \rightarrow L, S \mid S$$

Draw parse trees for:

(a, a)

(a, ((a, a), (a, a)))

# In-class Exercise

- Consider the CFG:

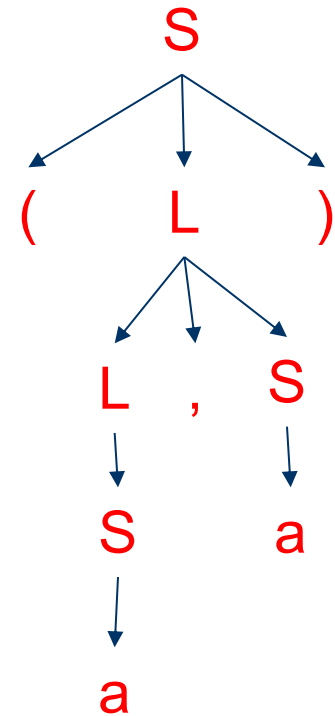
$S \rightarrow (L) \mid a$

$L \rightarrow L, S \mid S$

Draw parse trees for:

$(a, a)$

$(a, ((a, a), (a, a)))$



100%

- Consider the CFG:

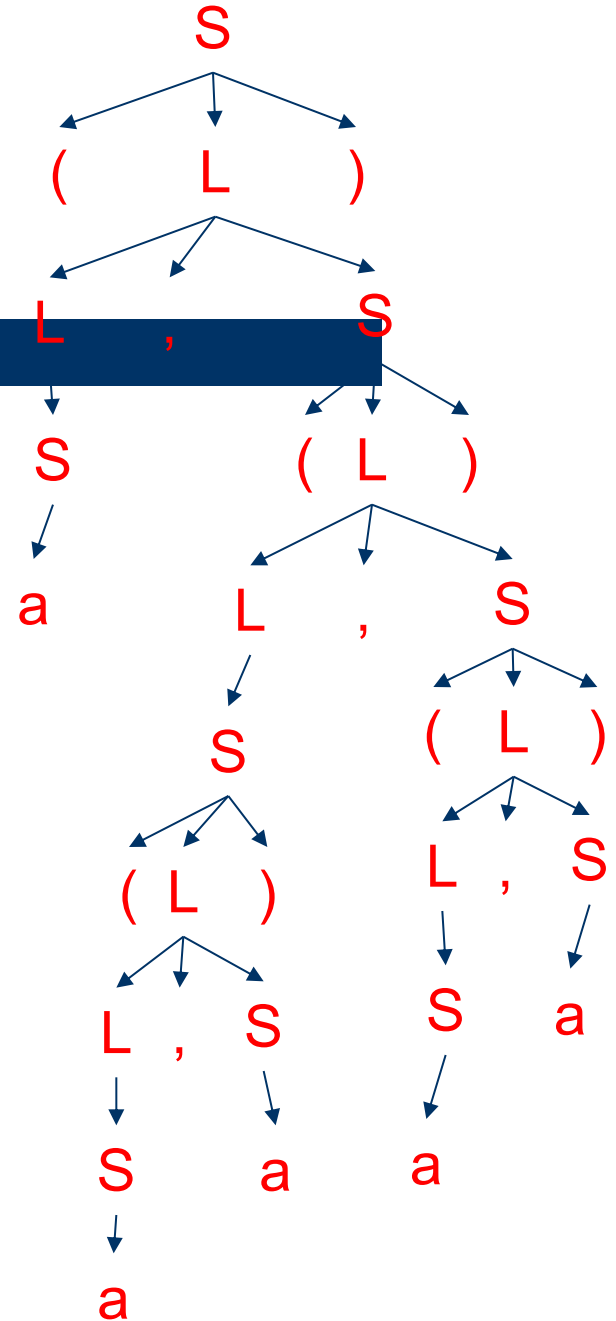
S → ( L ) |

a

$$L \rightarrow L, S \mid S$$

## Draw parse trees for:

$(a, a)$

$$(a, ((a, a), (a, a)))$$




# In-class Exercise

- Write a CFG grammar for the language of well-formed parenthesized expressions
  - $()$ ,  $(( ))$ ,  $()()$ ,  $(( ))()$ , etc.: OK
  - $()$ ),  $)()$ ,  $(( ($ ),  $(( ($ , etc.: not OK

# In-class Exercise

- Write a CFG grammar for the language of well-formed parenthesized expressions
  - $()$ ,  $(( ))$ ,  $()()$ ,  $(( ))()$ , etc.: OK
  - $()$ ),  $)()$ ,  $((()$ ),  $((()$ , etc.: not OK

$P \rightarrow () \mid PP \mid (P)$

## In-class Exercise

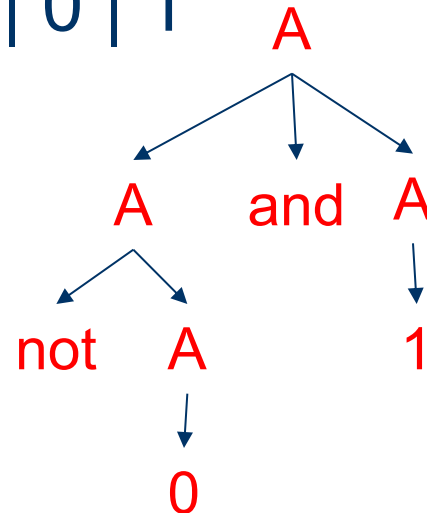
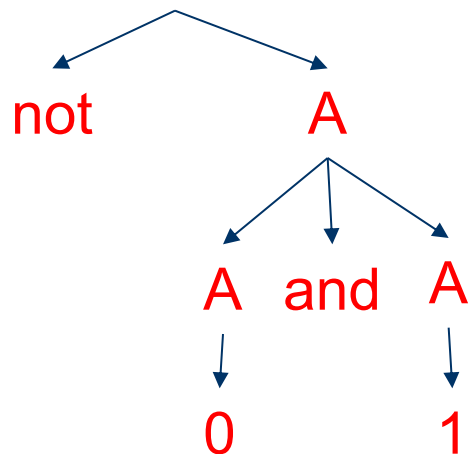
- Is the following grammar ambiguous?

$$A \rightarrow A \text{ "and" } A \mid \text{ "not" } A \mid \text{ "0" } \mid \text{ "1" }$$

# In-class Exercise

- Is the following grammar ambiguous?

$A \rightarrow A \text{ "and" } A \mid \text{not } A \mid 0 \mid 1$



# Another Example Grammar

ForStatement  $\rightarrow$  for "(" StmtCommaList ";"  
ExprCommaList ";" StmtCommaList ")" "{"  
StmtSemicList "}"

StmtCommaList  $\rightarrow$   $\varepsilon$  | Stmt | Stmt "," StmtCommaList

ExprCommaList  $\rightarrow$   $\varepsilon$  | Expr | Expr "," ExprCommaList

StmtSemicList  $\rightarrow$   $\varepsilon$  | Stmt | Stmt ";" StmtSemicList

Expr  $\rightarrow$  . . .

Stmt  $\rightarrow$  . . .

# Full Language Grammar Sketch

**Program**  $\rightarrow$  VarDeclList FuncDeclList

VarDeclList  $\rightarrow$   $\varepsilon$  | VarDecl | VarDecl VarDeclList

VarDecl  $\rightarrow$  Type IdentCommaList “;”

IdentCommaList  $\rightarrow$  Ident | Ident “,” IdentCommaList

Type  $\rightarrow$  int | char | float

FuncDeclList  $\rightarrow$   $\varepsilon$  | FuncDecl | FuncDecl FuncDeclList

FuncDecl  $\rightarrow$  Type Ident “(“ ArgList “)” “{“ VarDeclList StmtList “}”

StmtList  $\rightarrow$   $\varepsilon$  | Stmt | Stmt StmtList

Stmt  $\rightarrow$  Ident “=” Expr “;” | ForStatement | ...

Expr  $\rightarrow$  ...

Ident  $\rightarrow$  ...

# Real-world CFGs

- Some sample grammars found on the Web
  - LISP: 7 rules
  - PROLOG: 19 rules
  - Java: 30 rules
  - C: 60 rules
  - Ada: 280 rules

# So What Now?

- We want to write a compiler for a given language
- We come up with a definition of the tokens embodied in regular expressions
- We build a lexer (see previous lecture)
- We come up with a definition of the syntax embodied in a context-free grammar
  - not ambiguous
  - enforces relevant operator precedences and associativity
- Question: How do we build a parser?



# How do we build a Parser?

- This question could keep us busy for 1/2 semester in a full-fledge compiler course
- So we're just going to see a very high-level view of parsing
  - If you go to graduate school you'll most likely have an in-depth compiler course with all the details

# How do we build a Parser?

- There are two approaches for parsing:
  - **Top-Down**: Start with the start symbol and try to expand it using derivation rules until you get the input source code
  - **Bottom-Up**: Start with the input source code, consume symbols, and infer which rules could be used
- Note: this does not work for all CFGs
  - CFGs must have some properties to be parsable with our beloved parsing algorithms

# Writing Parsers?

- Nowadays one doesn't really write parsers from scratch, but one uses a parser generator (Yacc is a famous one)

