Principles of Programming Languages
Syntax Analysis

Hien D. Nguyen, ph.D

University of Information Technology

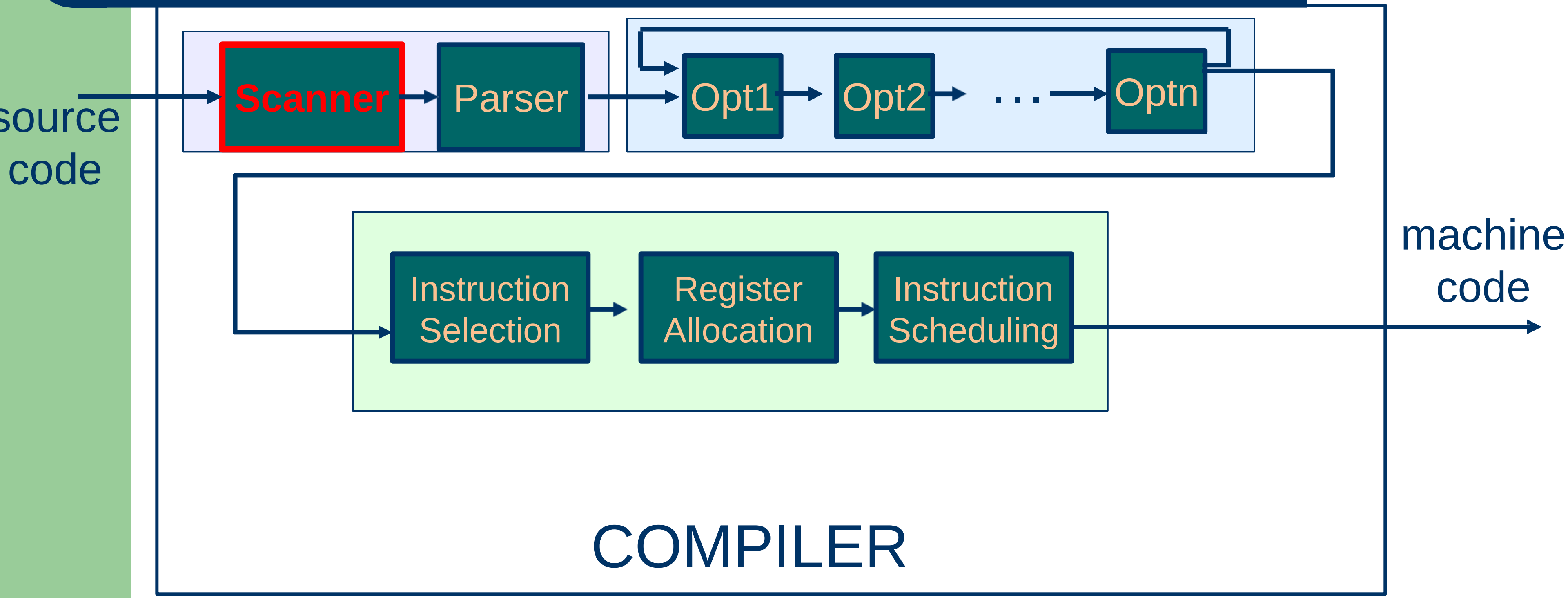
Lecture slides prepared by:

Quan Thanh Tho (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 = ({exp,op},{+,-,*,/,**id**},exp)
- exp → exp op exp
- exp → **id**
- op → +|-|*|/

Derivation

- α = uXv derives β = uγv if X-> γ is a production

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

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

  $\alpha \Rightarrow^+ \beta$

Derivations: $S \Rightarrow^+ \alpha$ where α 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

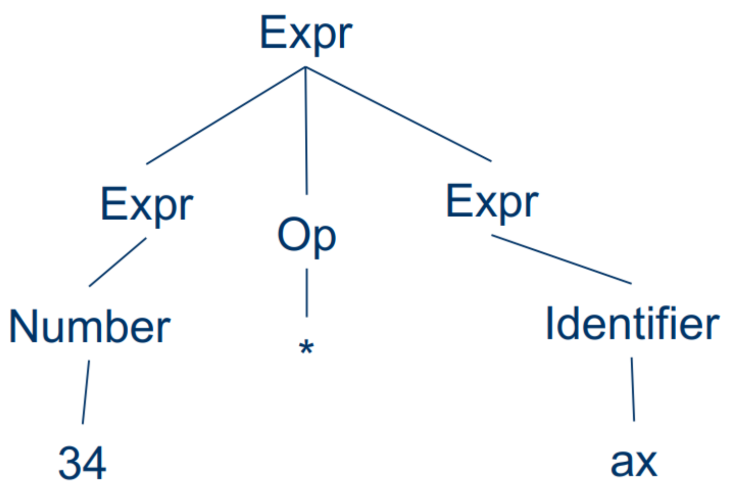
- exp $\Rightarrow$ exp op exp $\Rightarrow$ exp op **id** $\Rightarrow$ **id** op **id** $\Rightarrow$ **id** + **id**
- exp $\Rightarrow$ exp op exp $\Rightarrow$ **id** op exp $\Rightarrow$ **id** + exp $\Rightarrow$ **id** + **id**
- exp $\Rightarrow$ exp op exp $\Rightarrow$ exp op exp op exp $\Rightarrow$ **id** op exp op exp $\Rightarrow$ **id** + exp op exp $\Rightarrow$ **id** + exp * exp $\Rightarrow$ **id** + **id** * exp $\Rightarrow$ **id** + **id** * **id**

Example 3

- exp $\Rightarrow$ exp op exp $\Rightarrow$ **id** op exp $\Rightarrow$ **id** + exp $\Rightarrow$ **id** + **id**
- exp $\Rightarrow$ exp op exp $\Rightarrow$ exp op **id** $\Rightarrow$ exp + **id** $\Rightarrow$ **id** + **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:



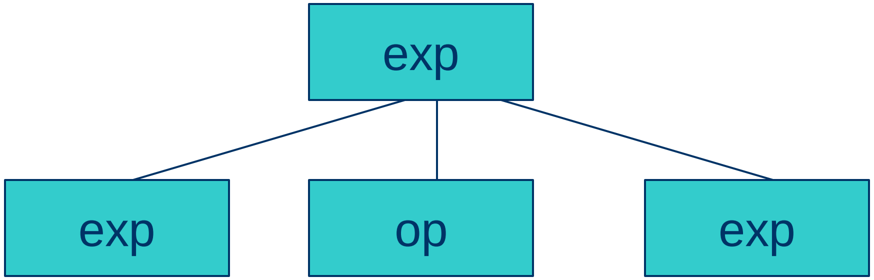
Example 4

- exp

exp

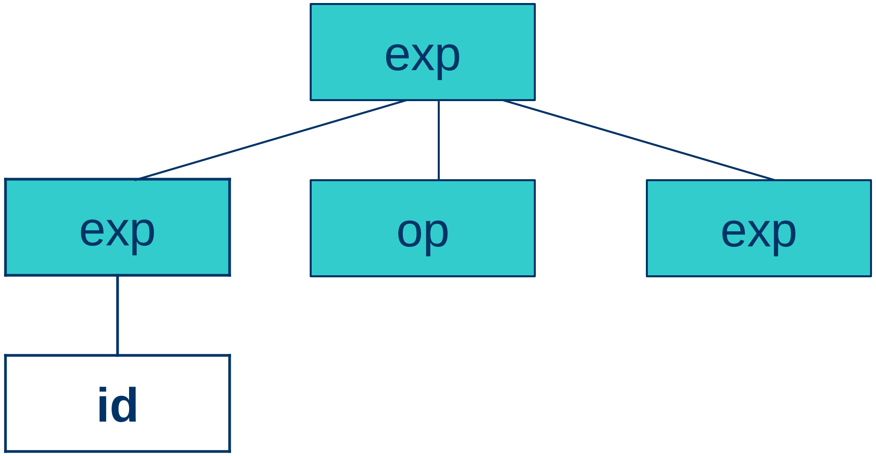
Example 4

- exp $\Rightarrow$ exp op exp



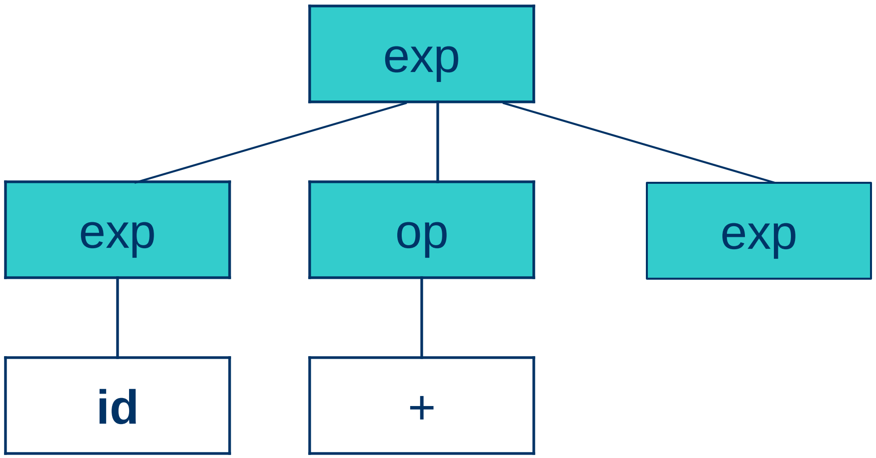
Example 4

exp $\Rightarrow$ exp op exp $\Rightarrow$ **id** op exp



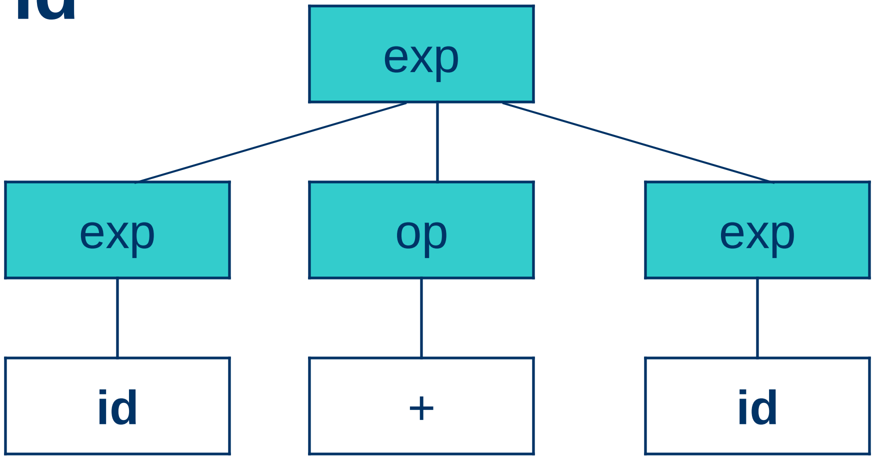
Example 4

- exp $\Rightarrow$ exp op exp $\Rightarrow$ **id** op exp $\Rightarrow$ **id** + exp



Example 4

- exp $\Rightarrow$ exp op exp $\Rightarrow$ **id** op exp $\Rightarrow$ **id** + exp $\Rightarrow$ **id** + **id**



Classic Expression Grammar

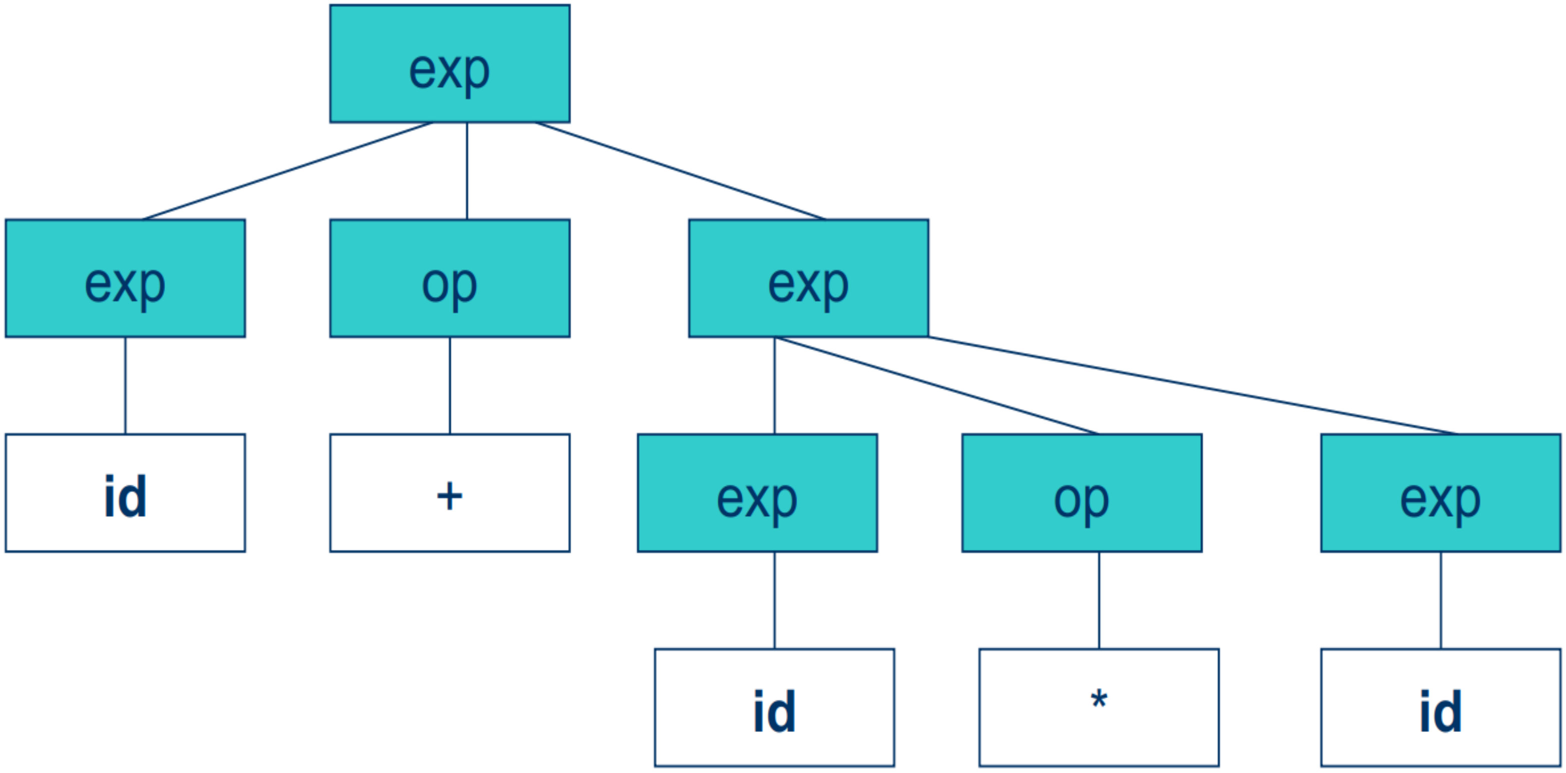
exp → exp + term | exp – term | term

term → term * factor | term /factor |factor

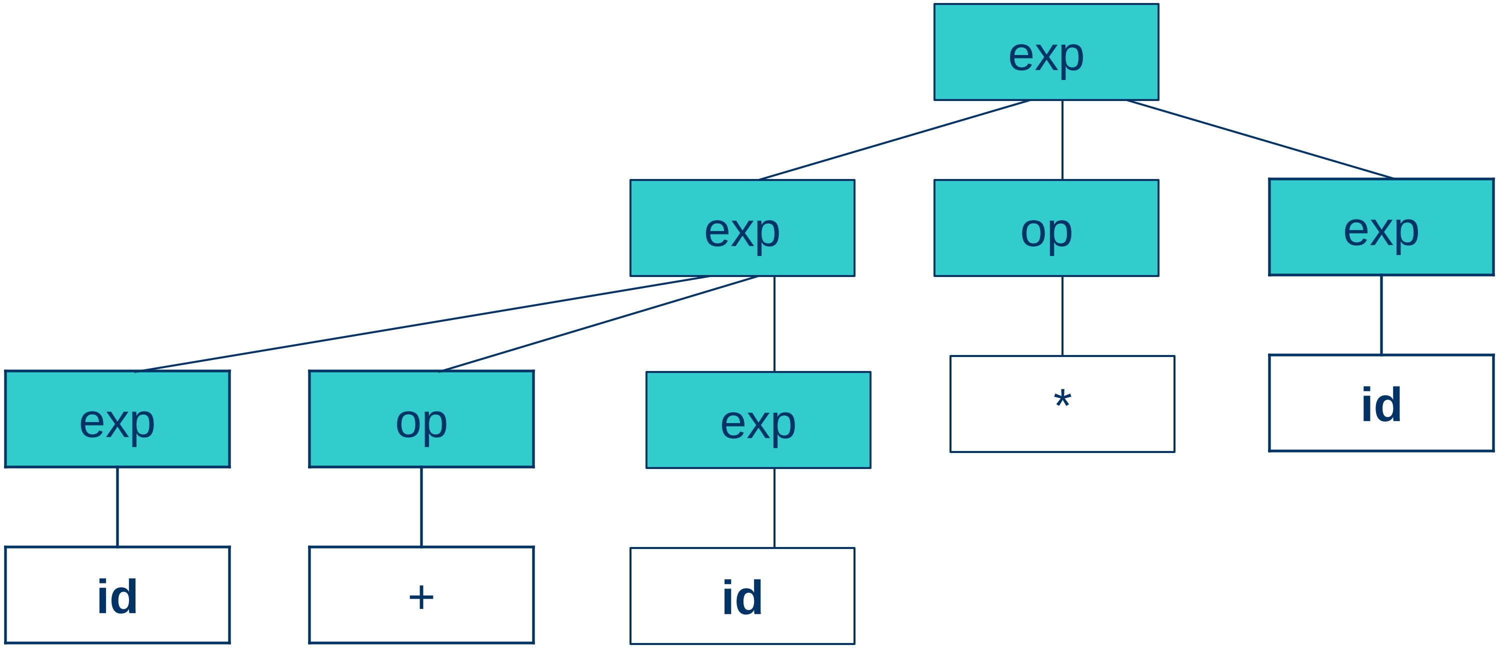
factor → (exp) | **ID** | **INT**

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

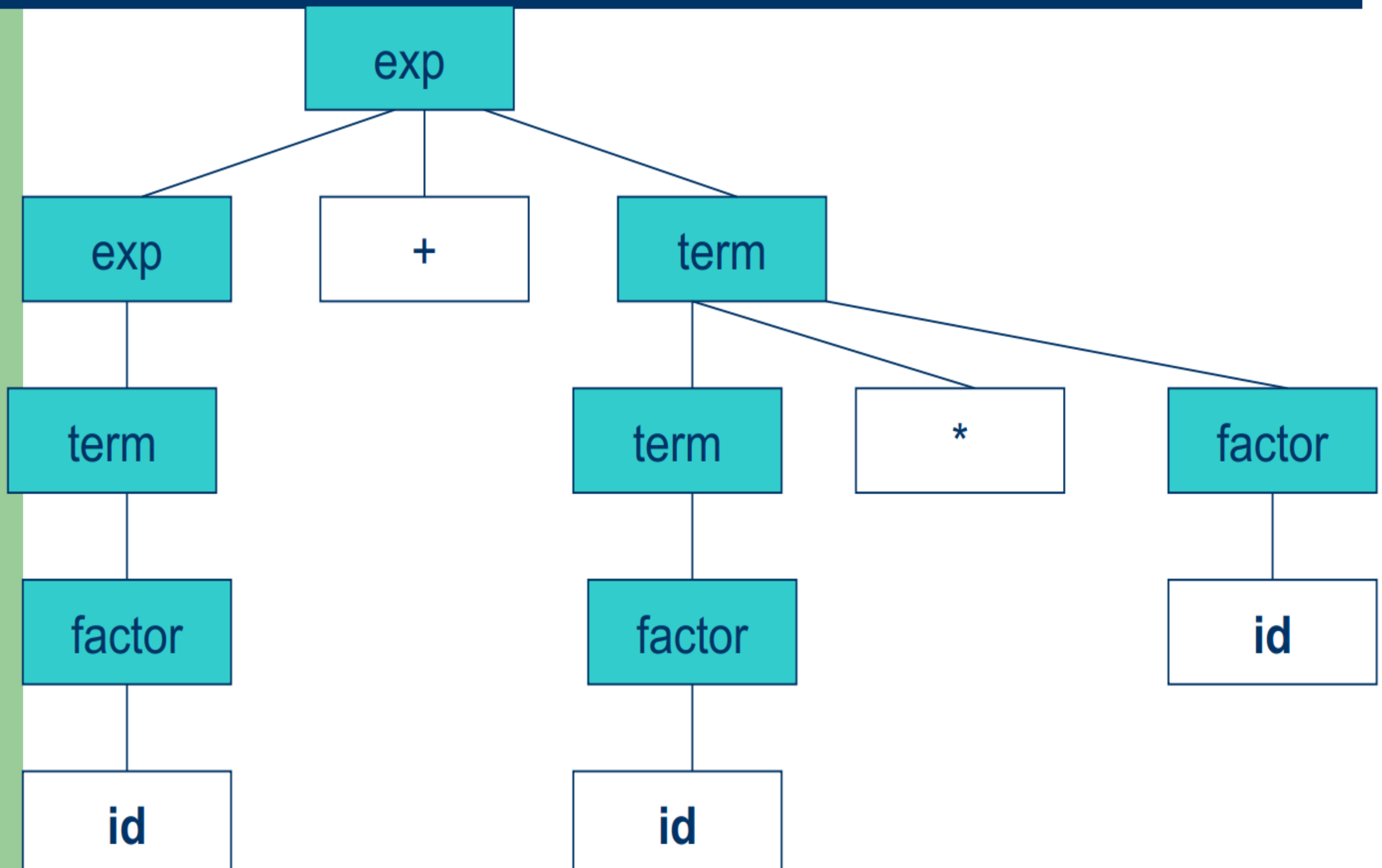
Operation Precedence



Operation Precedence

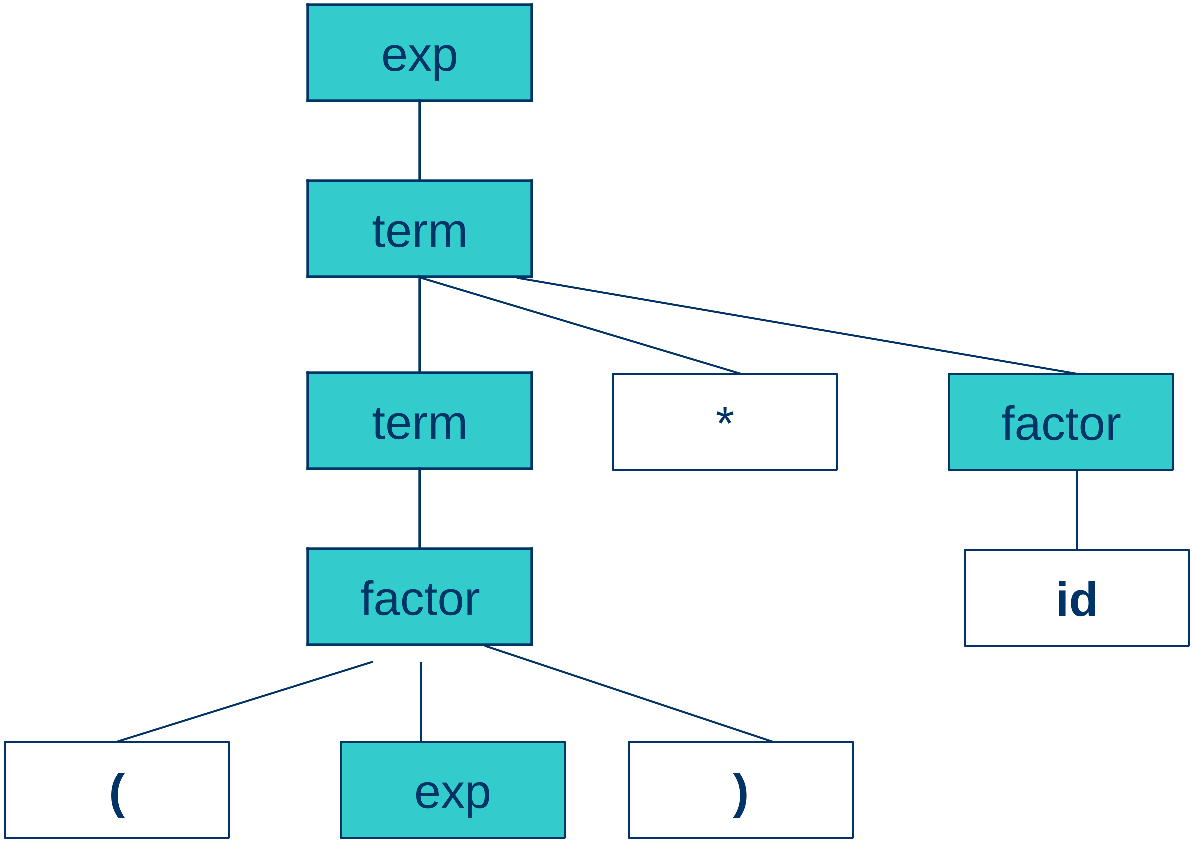


Operation Precedence

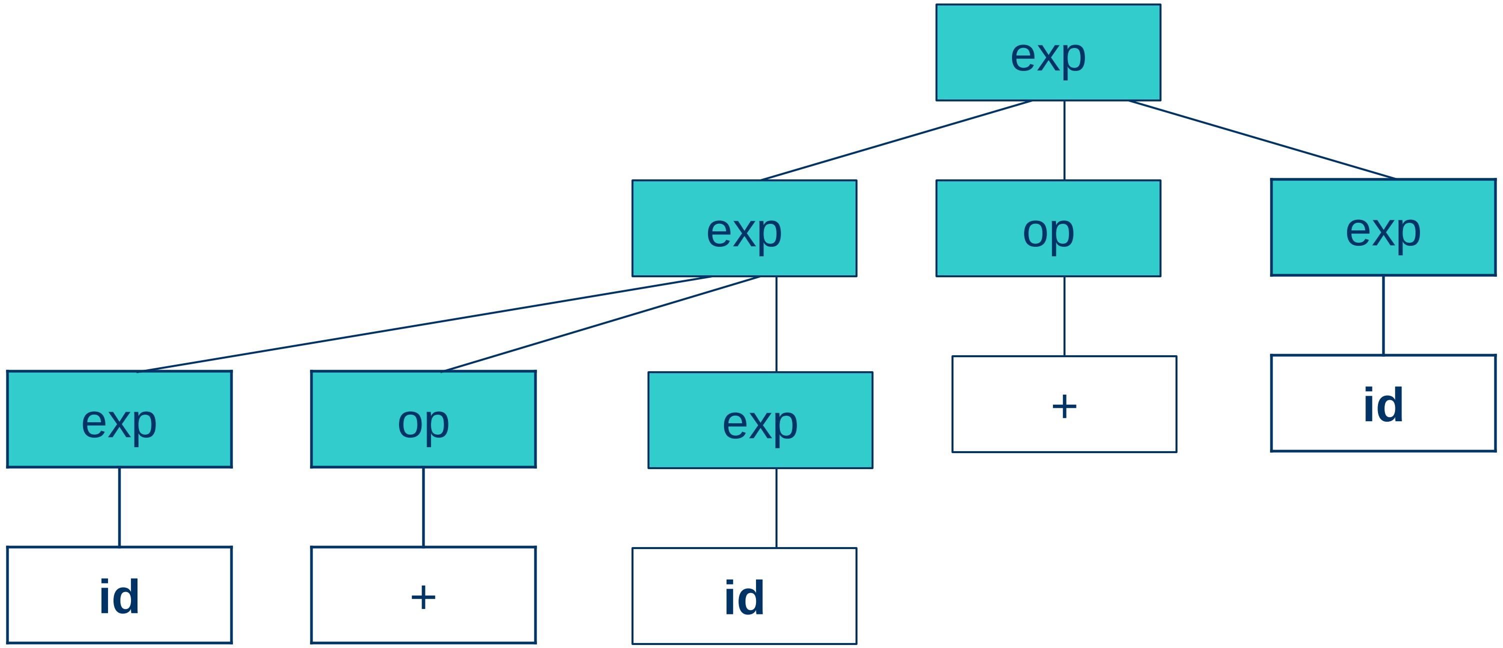


Operation Precedence

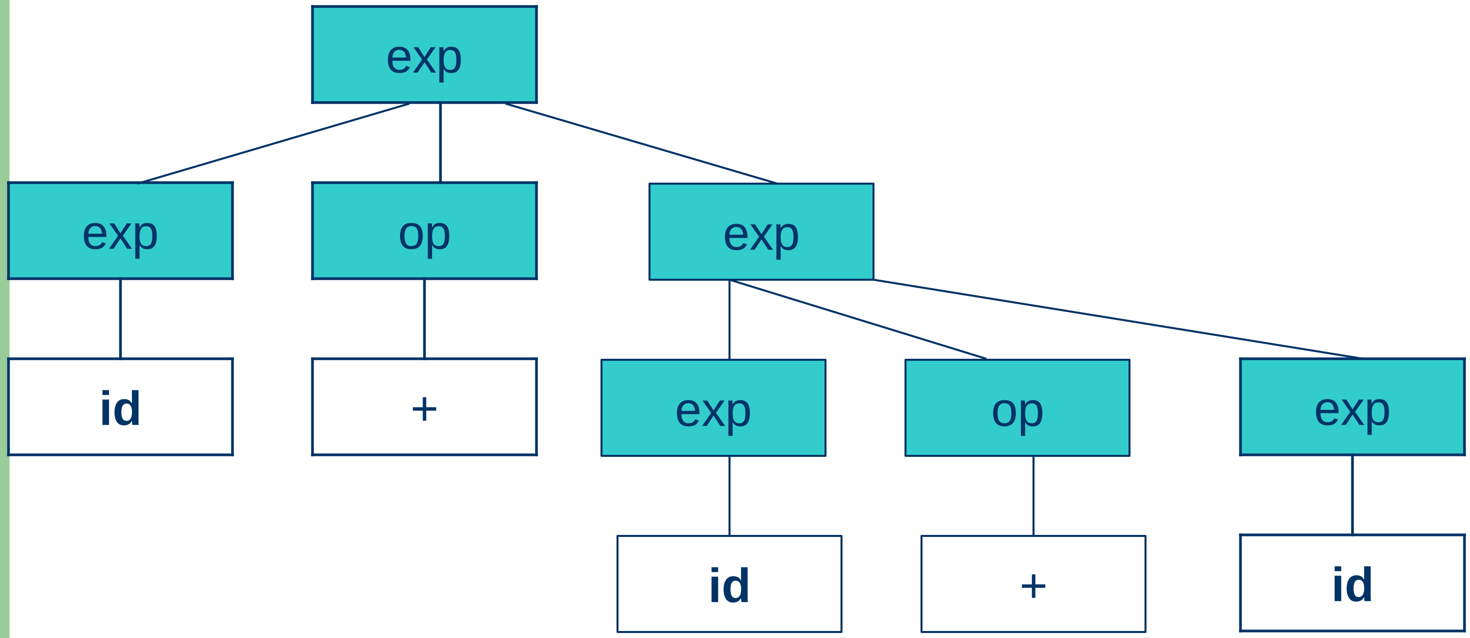
- (id+id) *id



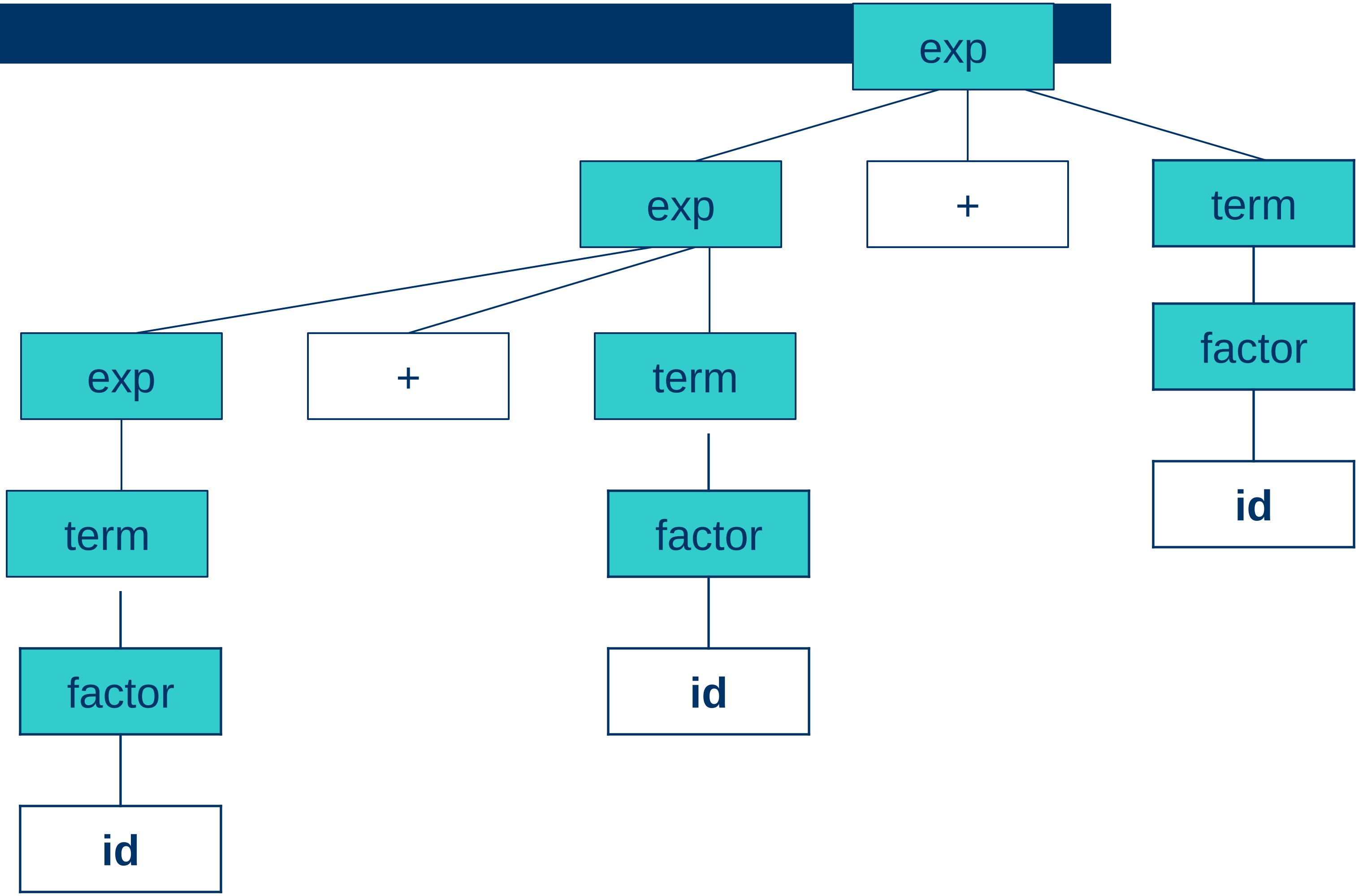
Operator Associativity



Operator Associativity



Operator Associativity



Precedence and Associativity

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

Hands-on Excersice

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

Expr -> Term | Expr + Term | Expr - Term

Term -> Factor | Term * Factor

Factor -> (Expr) | number

number -> digit {digit}

digit -> 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

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

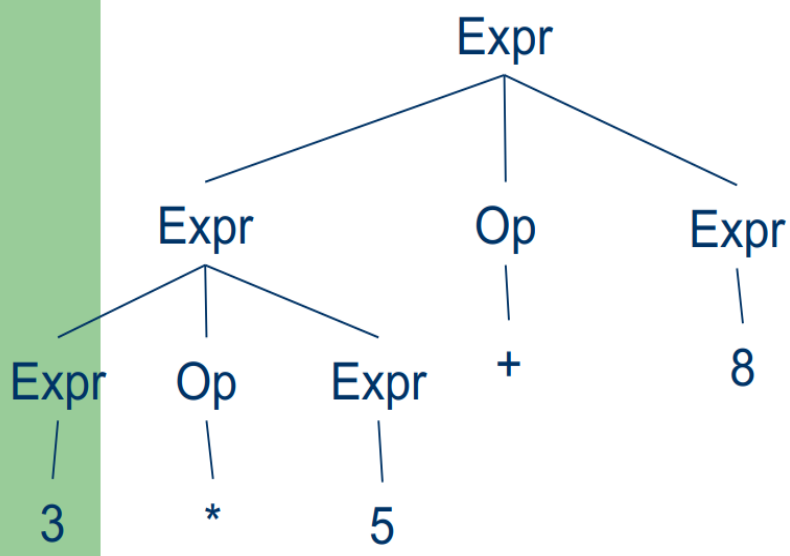
What is parsing

- 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

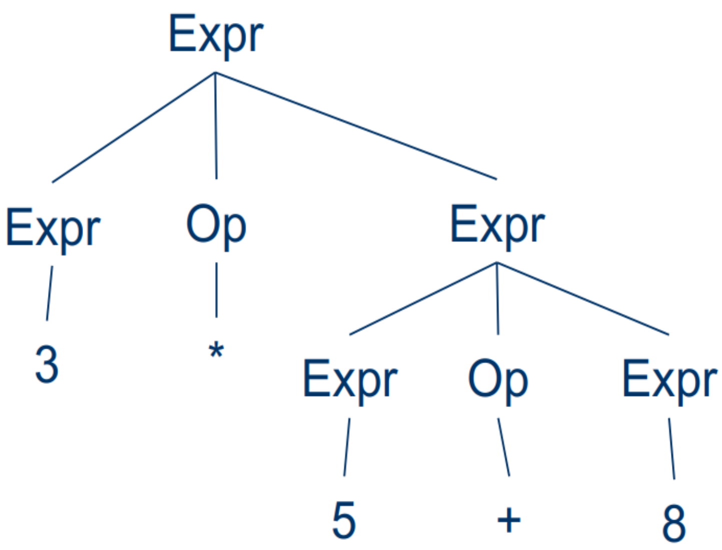
Ambiguity

- 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 → Term | Expr + Term | Expr - Term

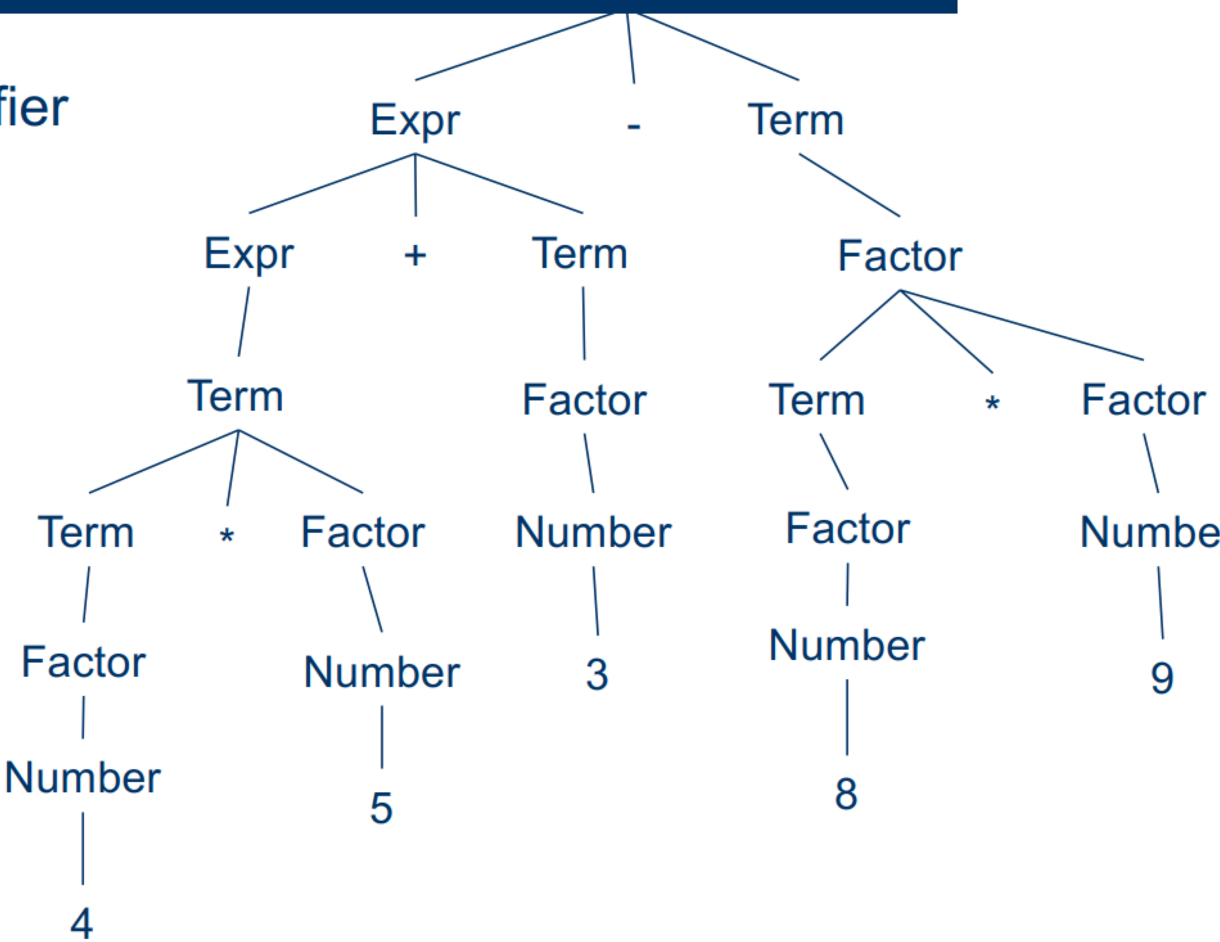
Term → Term * Factor

[redacted]

| Factor

Factor → Number | Identifier

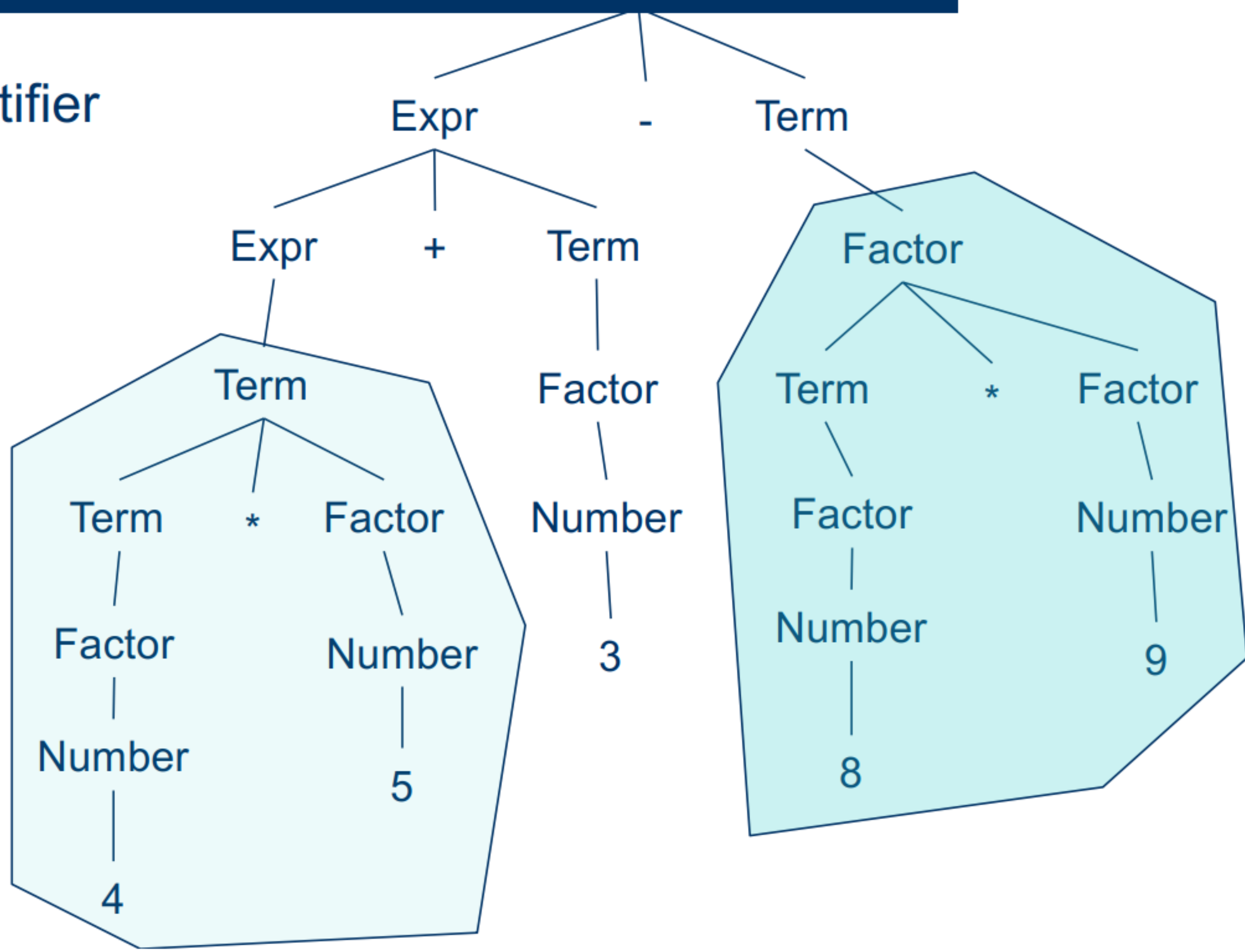
Example: 4*5+3-8*9



Non-Ambiguous Grammar

Expr → Term | Expr + Term | Expr - Term
Term → Term * Factor
[redacted]
| Factor
Factor → 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

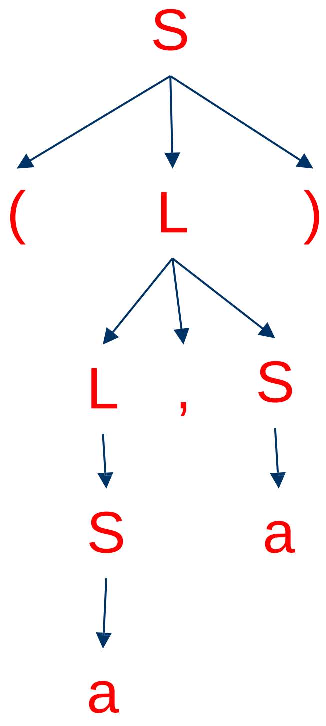
- Consider the CFG:

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

Draw parse trees for:

(a, a)

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



In-class Exercise

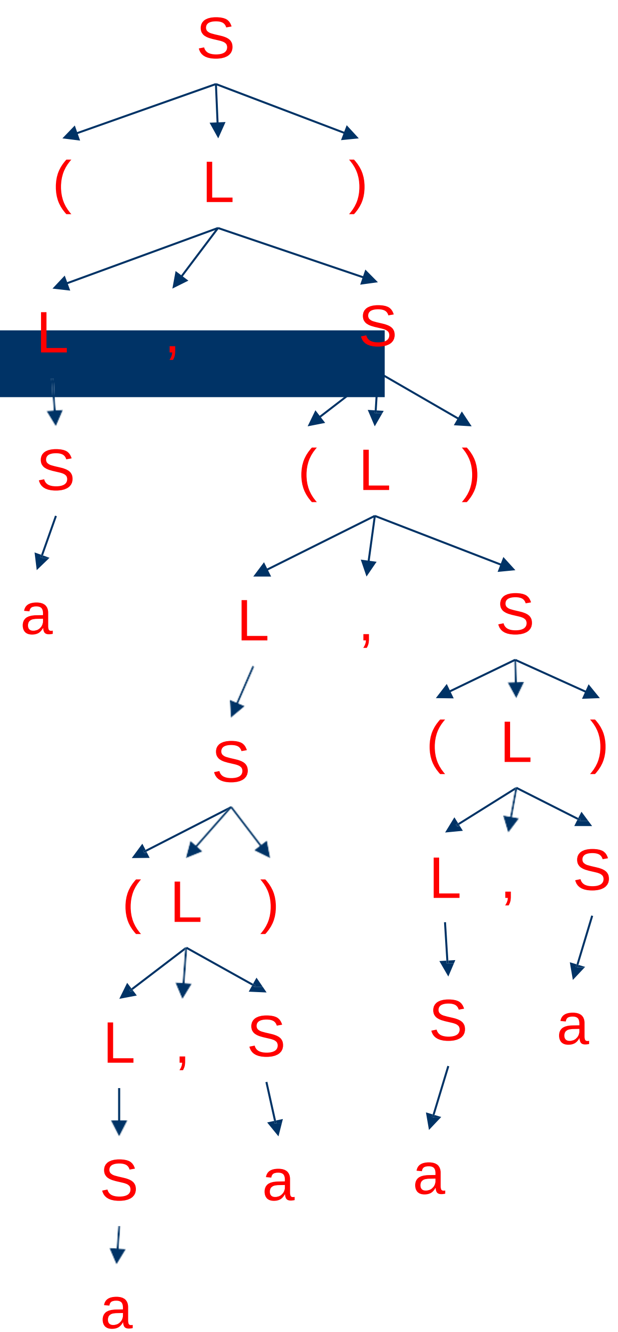
- Consider the CFG:

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

Draw parse trees for:

(a, a)

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



In-class Exercise

- 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 $\rightarrow$ () | PP | (P)

In-class Exercise

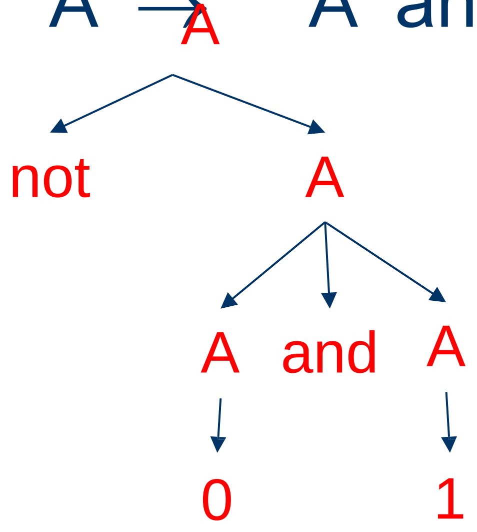
- Is the following grammar ambiguous?

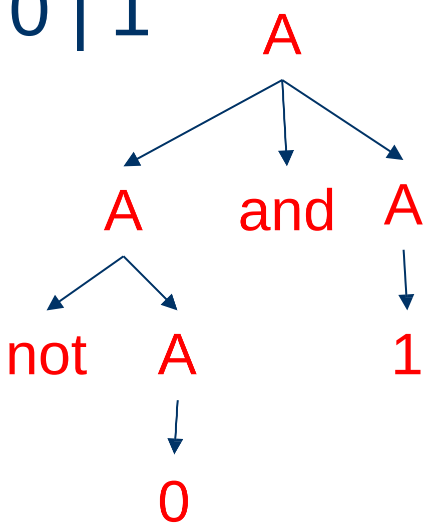
$A \rightarrow$ A "and" A | "not" A | "0" | "1"

In-class Exercise

- Is the following grammar ambiguous?

A $\rightarrow$ A “and” A | not A | 0 | 1





Another Example Grammar

ForStatement → for “(“ StmtCommaList “;” ExprCommaList “;” StmtCommaList ”)” “{“ StmtSemicList “}”

StmtCommaList → ε | Stmt | Stmt “,” StmtCommaList

ExprCommaList → ε | Expr | Expr “,” ExprCommaList

StmtSemicList → ε | Stmt | Stmt “;” StmtSemicList

Expr → . . .

Stmt → . . .

Full Language Grammar Sketch

Program ➞ VarDeclList FuncDeclList

VarDeclList ➞ ε | VarDecl | VarDecl VarDeclList

VarDecl ➞ Type IdentCommaList “;”

IdentCommaList ➞ Ident | Ident “,” IdentCommaList

Type ➞ int | char | float

FuncDeclList ➞ ε | FuncDecl | FuncDecl FuncDeclList

FuncDecl ➞ Type Ident “(“ ArgList “)” “{“ VarDeclList StmtList “}”

StmtList ➞ ε | Stmt | Stmt StmtList

Stmt ➞ Ident “=“ Expr “;” | ForStatement | ...

Expr ➞ ...

Ident ➞ ...

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 much 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)