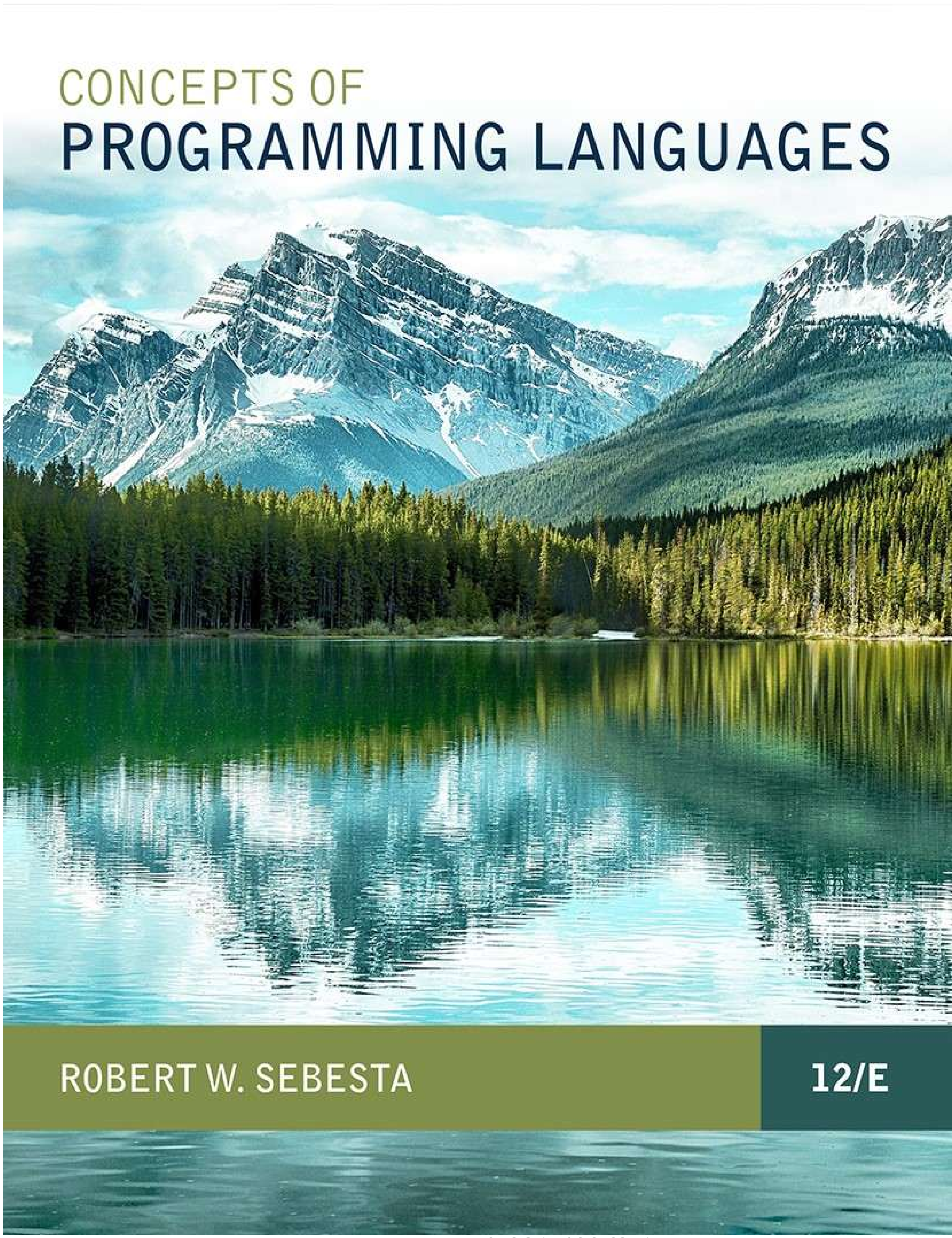
# Lecture 2

# Describing Syntax

Chapter 3 – Part 1

First Semester
1447 - 2025




ISBN 0-321-49362-1

# Lecture 2 Topics:

- Introduction
- The General Problem of Describing Syntax:
  - Language Recognizers
  - Language Generators
- Formal Methods of Describing Syntax:
  - Backus-Naur Form (BNF):
    - Extended BNF (EBNF)
  - Context-Free Grammars (CFG):
    - Grammars
    - Derivations
    - Parse Trees
    - Ambiguity
- Attribute Grammars:
  - Static Semantics

# Introduction

- The study of programming languages, like the study of natural languages, can be divided into:
  - Examinations of syntax.
  - Examinations of semantics.
- Syntax: the form or structure of the expressions, statements, and program units.
- Semantics: the meaning of the expressions, statements, and program units.
- Syntax and semantics provide a language's definition.

# Introduction: Example

- Example:
  - The syntax of a Java “while” statement is:
    - while (boolean_expr) statement
  - The semantics of the same statement is:
    - When the current value of the Boolean expression is true, the embedded statement is executed.
    - Then control implicitly returns to the Boolean expression to repeat the process.
    - If the Boolean expression is false, control transfers to the statement following the while construct.

# Introduction: Language Users

- Users of a language definition:
  - Other language designers (evaluators).
  - Implementers.
  - Programmers (the users of the language).

# The General Problem of Describing Syntax: Terminology

- A *sentence* (statement) is a string of characters over some alphabet.
- A *language* is a set of sentences (statements).
- A *lexeme* is the lowest level syntactic unit of a language (e.g., `*`, `sum`, `begin`).
  - The language operators.
  - The language special words.
  - The language numerical literals.
  - Etc.
- A *token* is a category of lexemes (e.g., identifier).

# Example

- Consider the following Java statement:
  - `index = 2 * count + 17;`

| Lexemes | Tokens |
| --- | --- |
| `index` | identifier |
| `=` | equal_sign |
| `2` | int_literal |
| `*` | mult_op |
| `count` | identifier |
| `+` | plus_op |
| `17` | int_literal |
| `;` | semicolon |

# Formal Definition of Languages

- Recognizers:
  - A recognition device reads input strings over the alphabet of the language and decides whether the input strings belong to the language (accept or reject the given input strings).
  - Example: syntax analysis (parsing) part of a compiler.
    - The syntax analyzer determines whether the given programs are syntactically correct.
  - Detailed discussion of syntax analysis appears in Chapter 4.
- Generators:
  - A device that generates sentences of a language.
  - One can determine if the syntax of a particular sentence is syntactically correct by comparing it to the structure of the generator.

# BNF and Context-Free Grammars

- Context-Free Grammars:
  - Developed by Noam Chomsky in the mid-1950s.
  - Language generators, meant to describe the syntax of natural languages.
  - Define a class of languages called context-free languages.
- Backus-Naur Form (1959):
  - Invented by John Backus to describe the syntax of Algol 58.
  - BNF is equivalent to context-free grammars.

# BNF Fundamentals

- BNF is a natural notation for describing syntax.
- In BNF, abstractions are used to represent classes of syntactic structures.
  - They act like syntactic variables (also called *nonterminal symbols*, or just *terminals*).
- Example:
  - A simple Java assignment statement might be represented by the abstraction <assign>
  - Pointed brackets are often used to delimit names of abstractions.
  - The actual definition of <assign> can be given by:
  - `<assign> → <var> = <expression>`

# BNF Fundamentals (continued)

- Terminals are lexemes or tokens.
- A rule has a left-hand side (LHS), which is a nonterminal, and a right-hand side (RHS), which is a string of terminals and/or nonterminals.
- Example:
  - See the next slide!

$$\texttt{<assign>} \rightarrow \texttt{<var>} = \texttt{<expression>}$$

- The text on the left side of the arrow, which is aptly called the left-hand side (LHS), is the abstraction being defined.
- The text to the right of the arrow is the definition of the LHS.
- It is called the right-hand side (RHS) and consists of some mixture of tokens, lexemes, and references to other abstractions. (Actually, tokens are also abstractions.)
- Altogether, the definition is called a rule, or production.
- In the example rule just given, the abstractions <var> and <expression> obviously must be defined for the <assign> definition to be useful.

# How a rule can be read?

**<assign>** → **<var>** = **<expression>**

- This particular rule specifies that the abstraction <assign> is defined as an <u>instance</u> of the abstraction <var>, <u>followed</u> by the lexeme =, <u>followed</u> by an <u>instance</u> of the abstraction <expression>.
- One example sentence whose <u>syntactic structure</u> is described by the rule above is:
  - `total = subtotal1 + subtotal2`

# BNF Fundamentals (continued)

- Nonterminals are often enclosed in "*angle brackets*" (abstraction).
  - Examples of BNF rules:

```
<ident_list> → identifier | identifier, <ident_list>

<if_stmt> → if <logic_expr> then <stmt>
```

- Grammar: a finite non-empty set of rules.
- A *start symbol* is a special element of the nonterminals of a grammar.

# BNF Rules

- An abstraction (or nonterminal symbol) can have more than one RHS:

<stmt> → <single_stmt>

<stmt> → **begin** <stmt_list> **end**

- These two rules can be written as:

<stmt> → <single_stmt> | **begin** <stmt_list> **end**

# BNF Rules: More Examples

- a Java if statement can be described with the rules:

<if_stmt> → **if** ( <logic_expr> ) <stmt>

<if_stmt> → **if** ( <logic_expr> ) <stmt> **else** <stmt>

or with the rule

<if_stmt> → **if** ( <logic_expr> ) <stmt>
| **if** ( <logic_expr> ) <stmt> **else** <stmt>

# Describing Lists

- Example of a list:
  - a list of identifiers appearing on a data declaration statement.
- Syntactic lists are described using recursion:

```
<ident_list> → ident
             | ident, <ident_list>
```

- A rule is recursive if its LHS appears in its RHS.

# Derivation

- A grammar is a generative device for defining languages.
- A derivation is a repeated application of rules (grammars), starting with the start symbol and ending with a sentence (all terminal symbols).
- The start symbol represents a complete program and is often named <program>.

# An Example Grammar

```
<program> → <stmts>

<stmts> → <stmt> | <stmt> ; <stmts>

<stmt> → <var> = <expr>

<var>  → a | b | c | d

<expr> → <term> + <term> | <term> - <term>

<term> → <var> | const
```

- The **language** described by the **grammar** of has only <u>one</u> statement form:
  - assignment.

# An Example Derivation

- How can we **derive** the following **assignment** statement using the previous **grammars** (rules)?
  - `a = b + const`

The symbol => is read "derives."

```
<program> => <stmts>
          => <stmt>
          => <var> = <expr>
          => a = <expr>
          => a = <term> + <term>
          => a = <var> + <term>
          => a = b + <term>
          => a = b + const
```

# Another Example Grammar

**A Grammar for a Small Language**

<program> → **begin** <stmt_list> **end**

<stmt_list> → <stmt>
  | <stmt> ; <stmt_list>

<stmt> → <var> = <expression>

<var> → A | B | C

<expression> → <var> + <var>
  | <var> - <var>
  | <var>

```
begin
A = B + C ;
B = C
end
```

How can we **derive** the following small program?

# Another Example Derivation

<program> => **begin** <stmt_list> **end**

=> **begin** <stmt> ; <stmt_list> **end**

=> **begin** <var> = <expression> ; <stmt_list> **end**

=> **begin** A = <expression> ; <stmt_list> **end**

=> **begin** A = <var> + <var> ; <stmt_list> **end**

=> **begin** A = B + <var> ; <stmt_list> **end**

=> **begin** A = B + C ; <stmt_list> **end**

=> **begin** A = B + ; <stmt> **end**

=> **begin** A = B + C ; <var> = <expression> **end**

=> **begin** A = B + C ; B = <expression> **end**

=> **begin** A = B + C ; B = <var> **end**

=> **begin** A = B + C ; B = C **end**

# Derivations

- Every string of symbols in a derivation is a *sentential form*.
- A *sentence* (statement) is a sentential form that has only terminal symbols.
- A *leftmost derivation* is one in which the leftmost nonterminal in each sentential form <u>is the one that is expanded</u>.
- A derivation may be <u>neither</u> leftmost <u>nor</u> rightmost.

# Yet Another Example Grammar

**A Grammar for Simple Assignment Statements**

```
<assign> → <id> = <expr>
<id> → A | B | C
<expr> → <id> + <expr>
       | <id> * <expr>
       | ( <expr>)
       | <id>
```

How can we **derive** the following **assignment** statement?

```
A = B * ( A + C )
```

# Yet Another Example Derivation

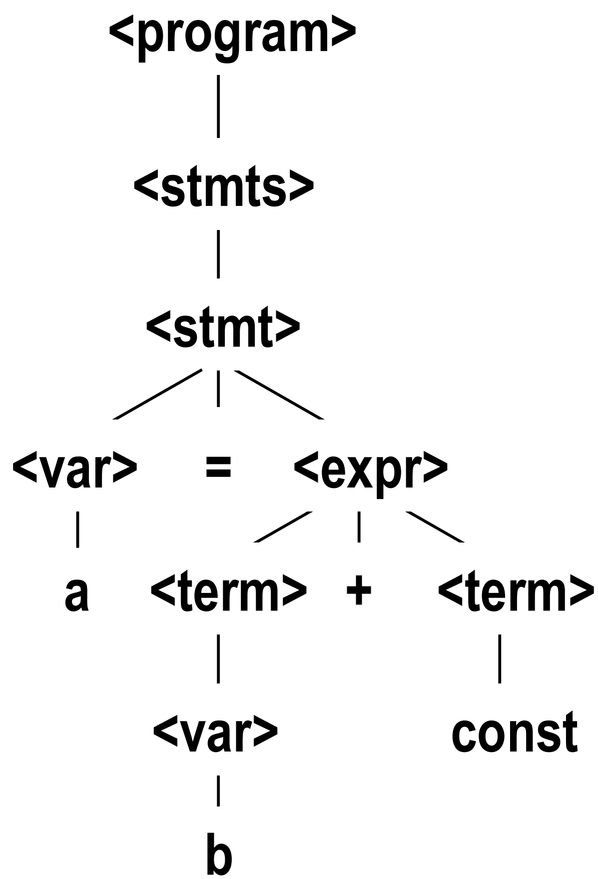
- The statement A = B * ( A + C ) is generated by the following leftmost derivation:

```
<assign> => <id> = <expr>
         => A = <expr>
         => A = <id> * <expr>
         => A = B * <expr>
         => A = B * ( <expr>)
         => A = B * ( <id> + <expr>)
         => A = B * ( A + <expr>)
         => A = B * ( A + <id>)
         => A = B * ( A + C )
```

# Parse Tree

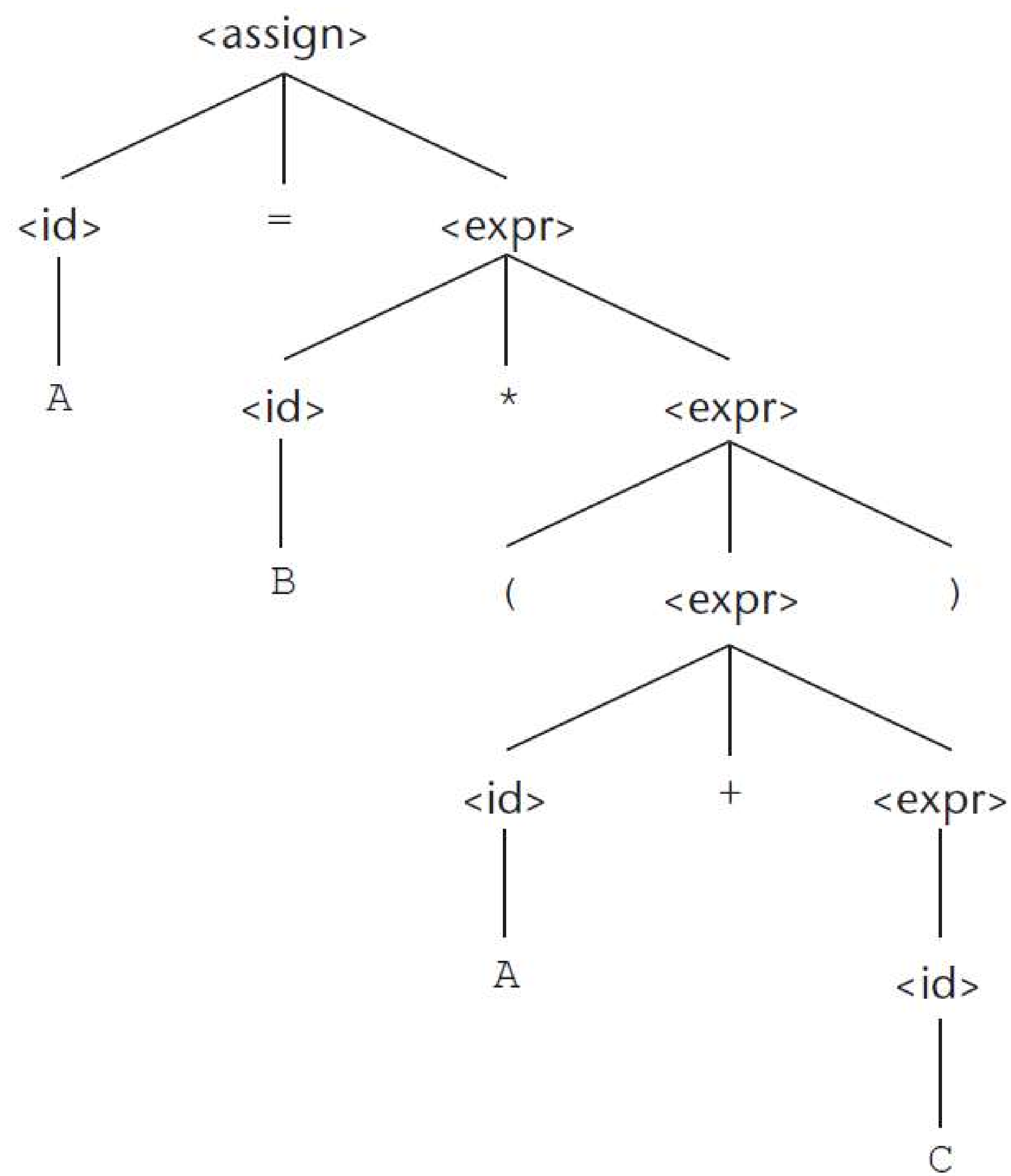
- A hierarchical representation of a derivation.

A **parse tree** for the simple statement
**`a = b + const`**




See slide 20

# Parse Tree




A **parse tree** for the simple statement

`A = B * (A + C)`

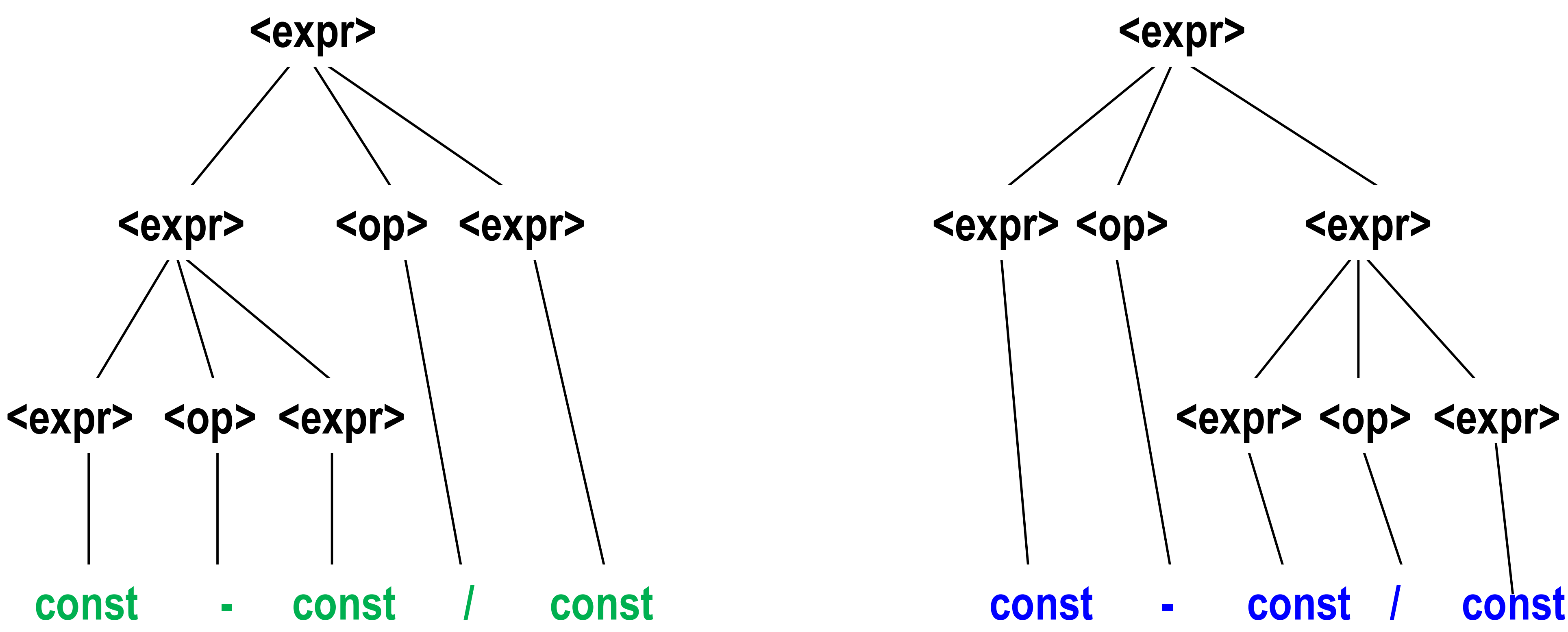
# Ambiguity in Grammars

- A grammar is *ambiguous* if and only if it generates a sentential form that has two or more distinct parse trees.
- This type of grammar allows the parse tree of an expression to grow on both left and right.
  - It should allow the tree to grow on the right only in such cases.
- How can this be a problem?
  - It confuses compilers during syntax analysis as compilers use the parse tree to generate code.
  - So, the meaning of the structure cannot be determined uniquely.

# An Ambiguous Expression Grammar

<expr> → <expr> <op> <expr> | const

<op> → / | -

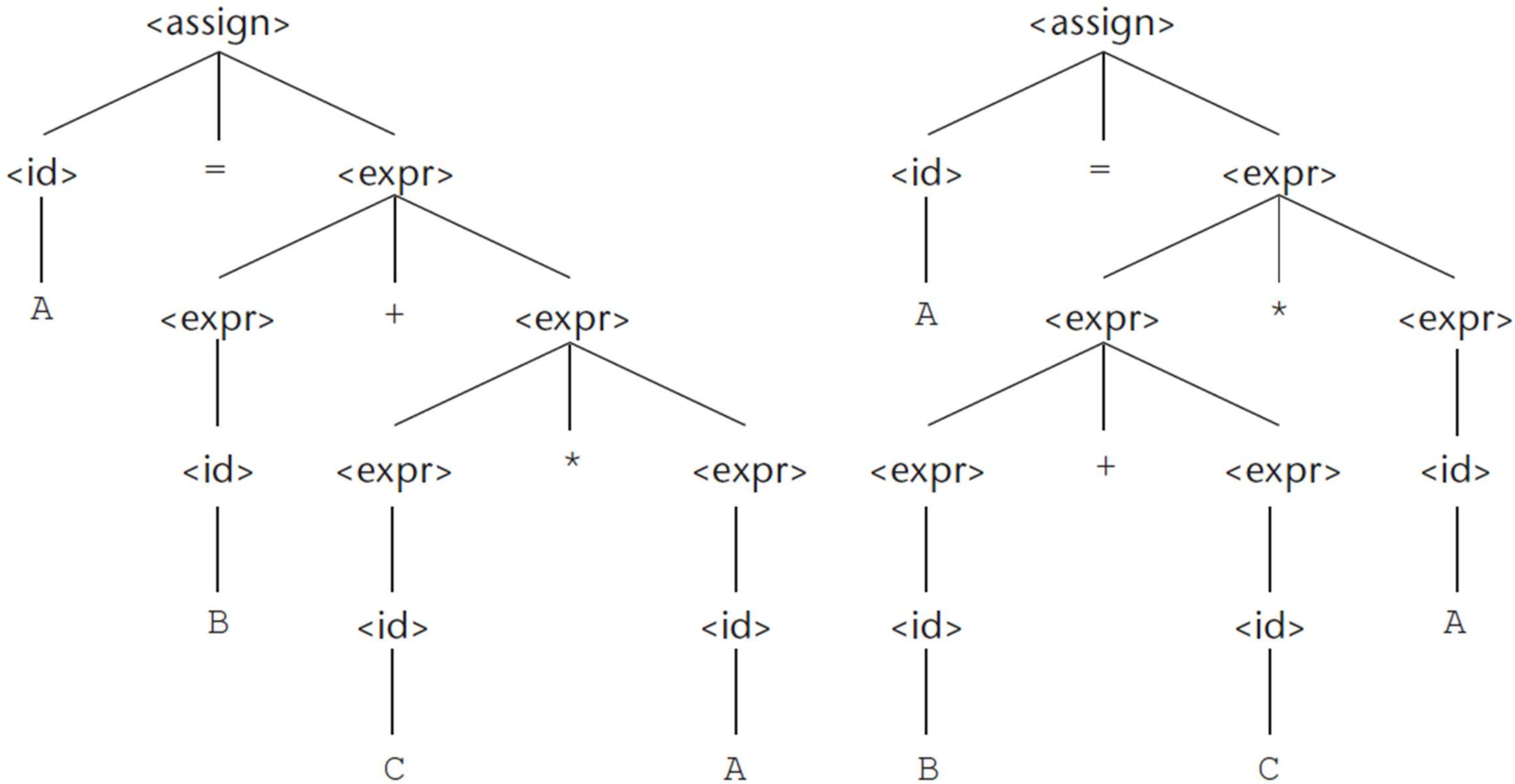


# Another Ambiguous Grammar

**An Ambiguous Grammar for Simple Assignment Statements**

<assign> → <id> = <expr>
<id> → A | B | C
<expr> → <expr> + <expr>
            | <expr> * <expr>
            | ( <expr> )
            | <id>

- This grammar is ambiguous because the sentence A = B + C * A has <u>two</u> *distinct* **parse trees**.
  - See the figure in the next slide.

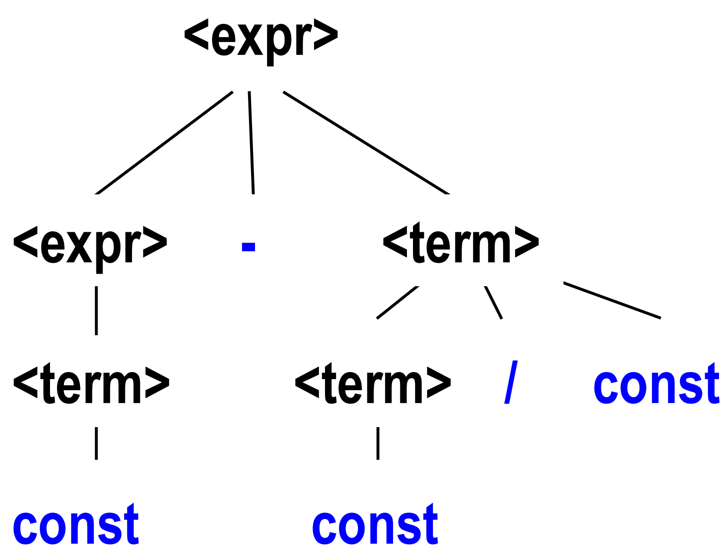



**Two** *distinct* **parse trees** for the same sentence, `A = B + C * A`

# An Unambiguous Expression Grammar

- If we use the parse tree to indicate precedence levels of the operators, we cannot have ambiguity.

<expr> → <expr> - <term> | <term>
<term> → <term> / const | const



# Operators Precedence

- Given the expression `x + y * z`, one obvious semantic_issue is the order of evaluation of the two operators.
  - Is it add and then multiply, or vice versa?
- This semantic question can be answered by <u>assigning different precedence levels to operators</u>.
- As grammar can describe a <u>certain syntactic structure</u> so that part of the meaning of the structure can be <u>determined from</u> its parse tree.

# Operators Precedence

- It is a fact that an operator in an arithmetic expression is generated lower in the parse tree must be evaluated first.
- This fact can be used to indicate that a lower operator in a parse tree has precedence over an operator produced higher up in the tree.
- However, *is this fact sufficient to solve the operator precedence problem?*
  - Not always! See the unambiguous grammar in slide 24.

# Operators Precedence

- For example, using the grammar in slide 24, try to sketch the parse trees for these two expressions:
  - `A + B * C`
  - `A * B + C`
- What have you noticed?
- You will see that:
  - For `A + B * C`, the (*) operator is the lowest in the tree, which will lead to a correct evaluation.
  - However, for `A * B + C` instead, the (+) operator is the lowest (indicating it is to be done first), which will lead to an incorrect evaluation.
- So, the grammar (slide 24) is sensitive to the order of the operators in the expressions.

# Operators Precedence

- So, how this problem can be solved?
  - Simply, take the order into consideration when designing the grammar by:
    - Use separate nonterminal symbols to represent the operands of the operators that have different precedence.
    - This requires additional nonterminals and some new rules.
  - For example, to correct the grammar in slide 24, we could use three nonterminals to represent operands, which allows the grammar to force different operators to different levels in the parse tree.
  - But, how? See the next slide!

# Operators Precedence

- If <expr> is the root symbol for expressions, + can be forced to the top of the parse tree by having <expr> directly generate only + operators, using the new nonterminal, <term>, as the right operand of +.
- Next, we can define <term> to generate * operators, using <term> as the left operand and a new nonterminal, <factor>, as its right operand.
- Now, * will always be lower in the parse tree, simply because it is farther from the start symbol than + in every derivation.

## A Grammar for Simple Assignment Statements

```
<assign> → <id> = <expr>
<id> → A | B | C
<expr> → <id> + <expr>
       | <id> * <expr>
       | ( <expr>)
       | <id>
```

## An Unambiguous Grammar for Expressions

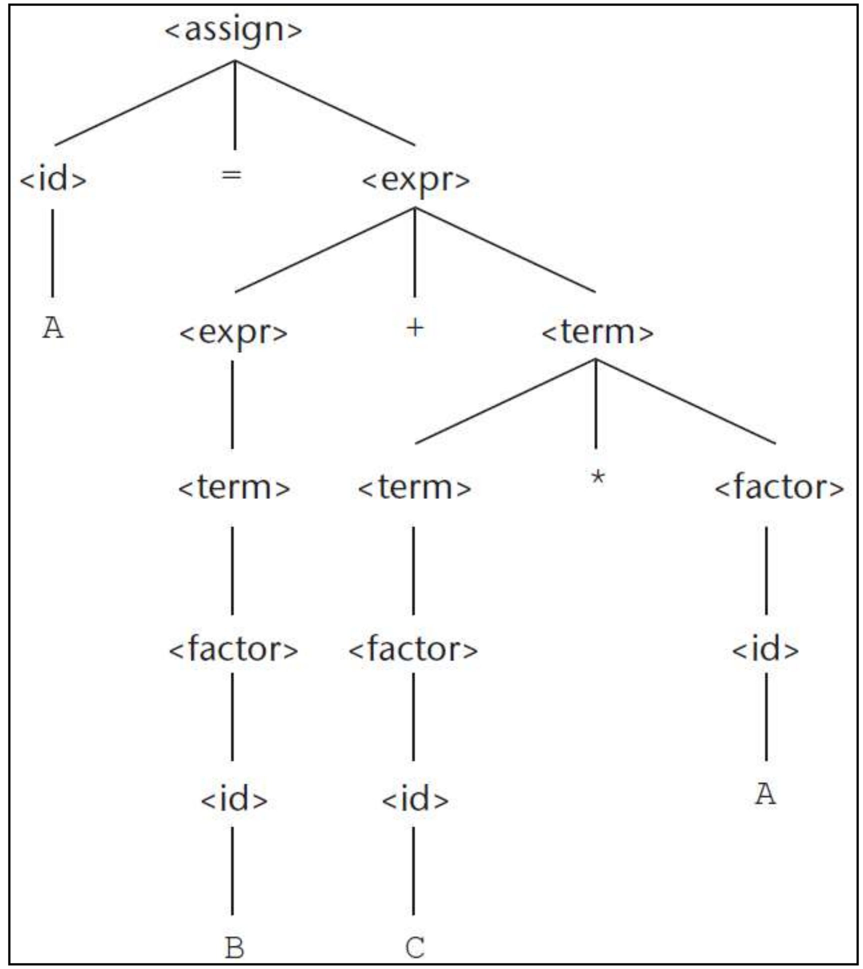
```
<assign> → <id> = <expr>
<id> → A | B | C
<expr> → <expr> + <term>
       | <term>
<term> → <term> * <factor>
       | <factor>
<factor> → ( <expr> )
         | <id>
```

This grammar generates the same language as the above grammar. It is unambiguous and it specifies the usual precedence order of multiplication and addition operators.

# Operators Precedence: Example (Leftmost Derivation)

`A = B + C * A`

<assign> => <id> = <expr>
=> A = <expr>
=> A = <expr> + <term>
=> A = <term> + <term>
=> A = <factor> + <term>
=> A = <id> + <term>
=> A = B + <term>
=> A = B + <term> * <factor>
=> A = B + <factor> * <factor>
=> A = B + <id> * <factor>
=> A = B + C * <factor>
=> A = B + C * <id>
=> A = B + C * A




The unique **parse tree** for `A = B + C * A` using an unambiguous grammar

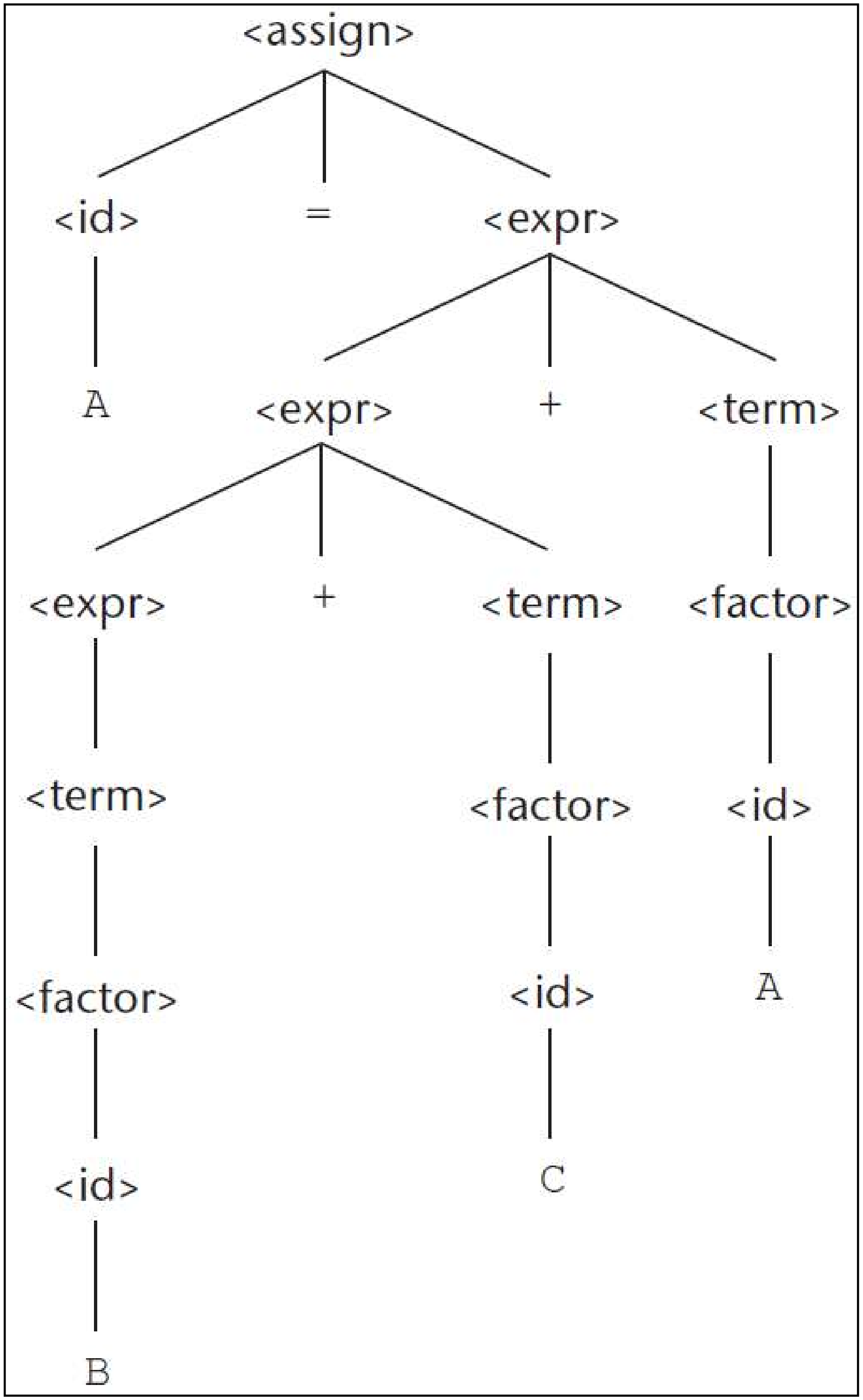
# Operators Precedence: Example (Rightmost Derivation)

```
A = B + C * A
```

```
<assign> => <id> = <expr>
         => <id> = <expr> + <term>
         => <id> = <expr> + <term> * <factor>
         => <id> = <expr> + <term> * <id>
         => <id> = <expr> + <term> * A
         => <id> = <expr> + <factor> * A
         => <id> = <expr> + <id> * A
         => <id> = <expr> + C * A
         => <id> = <term> + C * A
         => <id> = <factor> + C * A
         => <id> = <id> + C * A
         => <id> = B + C * A
         => A = B + C * A
```

# Associativity of Operators

- When an expression includes two operators that have the same precedence (as `*` and `/` usually have)—for example, "`A / B * C`", then a semantic rule is required to specify which should have precedence.
- This rule is named associativity.
- A grammar for expressions may correctly imply operator associativity.
- Consider the following example of an assignment statement:
  - `A = B + C + A`
  - After using the grammar in the next slide for the derivation of this statement, then its parse tree will look like:




## An Unambiguous Grammar for Expressions

<assign> → <id> = <expr>
<id> → A | B | C
<expr> → <expr> + <term>
| <term>
<term> → <term> * <factor>
| <factor>
<factor> → ( <expr> )
| <id>

A **parse tree** for
A = B + C + A
illustrating the
associativity of addition

# Associativity of Operators

- The previous parse tree shows the left addition operator lower than the right addition operator.
  - This is the correct order if addition is meant to be left associative, which is typical.
  - In most cases, the associativity of addition in a computer is irrelevant.
- In mathematics, addition is associative, which means that left and right associative orders of evaluation mean the same thing.
  - That is, `(A + B) + C` = `A + (B + C)`
- Subtraction and division are not associative, whether in mathematics or in a computer.
  - Therefore, correct associativity may be essential for an expression that contains either of them.

# Associativity of Operators

- When a grammar rule has its LHS also appearing at the beginning of its RHS, the rule is said to be left recursive.
  - This left recursion specifies left associativity.
- For example, the left recursion of the rules of the grammar below causes it to make both addition and multiplication left associative.

```
<assign> → <id> = <expr>
<id> → A | B | C
<expr> → <expr> + <term>
       | <term>
<term> → <term> * <factor>
       | <factor>
<factor> → ( <expr> )
         | <id>
```

# Associativity of Operators

- The exponentiation operator is right associative in most languages that provide it.
- To indicate right associativity, right recursion can be used.
- A grammar rule is right recursive if the LHS appears at the right end of the RHS.
- Rules such as:

```
<factor> → <exp> ** <factor>
         | <exp>
<exp> → ( <expr> )
      | id
```

could be used to describe exponentiation as a right-associative operator.

# Extended BNF

- Optional parts are placed in brackets [ ]

```
<proc_call> → ident [(<expr_list>)]
```

- Alternative parts of RHSs are placed inside parentheses and separated via vertical bars.

```
<term> → <term> (+|-) const
```

- Repetitions (zero or more) are placed inside braces { }

```
<ident> → letter {letter|digit}
```

# BNF and EBNF

- BNF:

```
<expr> → <expr> + <term>
       | <expr> - <term>
       | <term>
<term> → <term> * <factor>
       | <term> / <factor>
       | <factor>
```

- EBNF:

```
<expr> → <term> {(+ | -) <term>}
<term> → <factor> {(* | /) <factor>}
```

# Recent Variations in EBNF

- Alternative RHSs are put on separate lines.
- Use of a colon instead of `=>`
- Use of $_{\texttt{opt}}$ for optional parts.
- Use of `oneof` for choices.

# Any Questions?

- Please, read chapter 3 (*first 2 sections*)
- I hope you were taking some notes!
- To test your understanding of this lecture, have a go with the "Review Questions" in page 156 of the textbook.
- We will do more exercises later on.
- Please, keep reviewing this lecture regularly.
- We may have a quiz (lecture 1) next week!
- Please, start doing your assignment.