Course Code 007313 (Spring 2019)

프로그래밍언어의 개념

Concepts of Programming Language

(Lecture 06 : Chapter 3 - Describing Syntax and Semantics)

Prof. A. P. Shrestha, Ph.D.

Dept. of Computer Science and Engineering, Sejong University

Last Class

Chapter 3 - Describing Syntax and Semantics

- Introduction
- The General Problem of Describing Syntax
- Formal Methods of Describing Syntax

Today

Chapter 3 - Describing Syntax and Semantics

- Parse Tree
- Ambiguity
- EBNF
- Syntax Graph

Next class

Chapter 3- Describing Syntax and Semantics

6.1.1 An Example Grammar and Derivation

a = b + const

Example : Grammar

```
<program> → <stmts>
 <stmts> → <stmt> | <stmt> ; <stmts>
 <stmt> → <var> = <expr>
 <var> → a | b | c | d
 <expr> → <term> + <term> | <term> - <term>
 <term> → <var> | const
```

Example :Derivation

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

6.1.2 Leftmost and Rightmost Derivations

- Leftmost Derivations

  - at each step the leftmost nonterminal is replaced.

  - often indicated as $\overset{L}{\Rightarrow}$

- Rightmost Derivations

  - at each step the rightmost nonterminal is replaced.

  - often indicated as $\overset{R}{\Rightarrow}$

6.1.3 An Example Derivation and Parse Tree

- Parse tree is a hierarchical representation of a derivation

a = b + const

Derivation

<program> => <stmts> => <stmt>

$\overset{L}{\Rightarrow}$ <var> = <expr>

$\overset{L}{\Rightarrow}$ a = <expr>

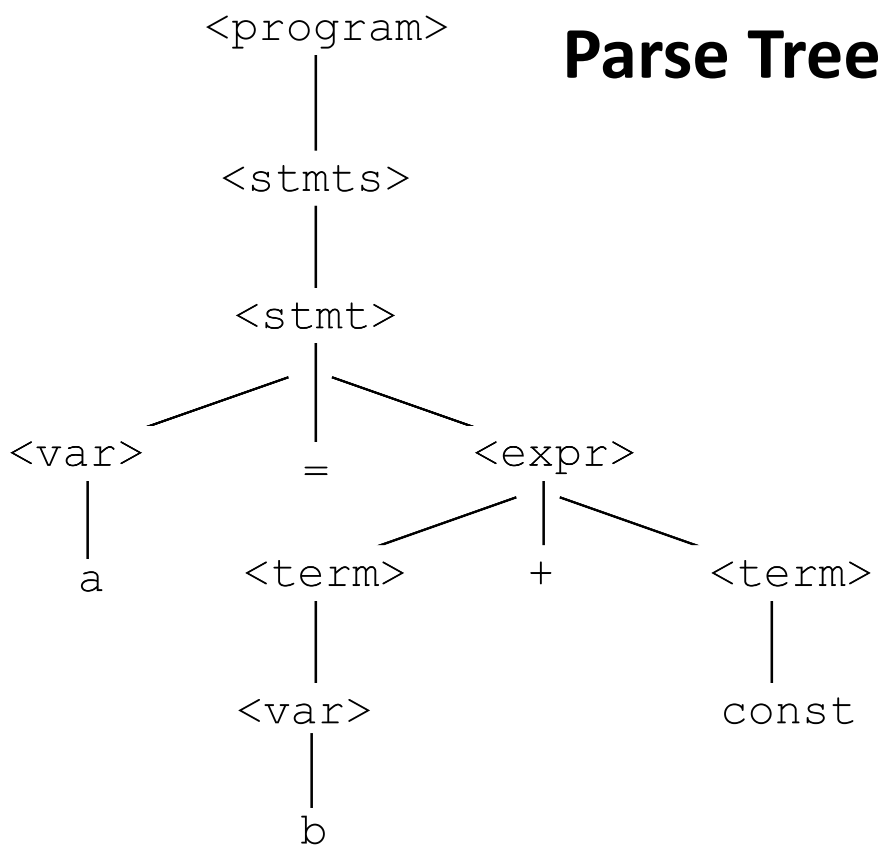
$\overset{L}{\Rightarrow}$ a = <term> + <term>

$\overset{L}{\Rightarrow}$ a = <var> + <term>

$\overset{L}{\Rightarrow}$ a = b + <term>

$\overset{L}{\Rightarrow}$ a = b + const

Parse Tree



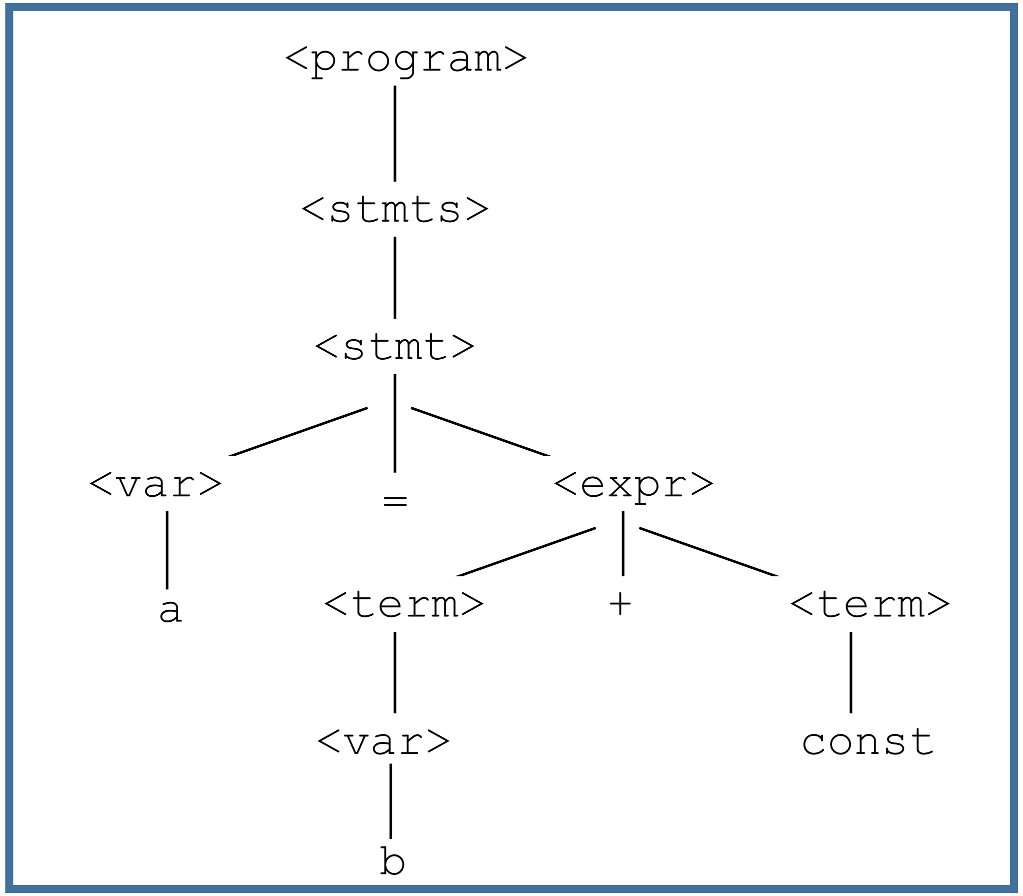
6.1.4 Grammar and Parse Tree

- Compiler tries to build a parse tree for every program we want to compile, using the grammar of the programming language
- The grammar can be viewed as a set of rules that say how to build a parse tree
- We put start symbol at the root of the tree
- Add children to every non-terminal, following any one of the rules for that non-terminal
- Done when all the leaves are terminals.
- Read off leaves from left to right—that is the string derived by the tree

6.1.5 Parse Tree

- Root of the tree is start symbol
- Every internal node of a parse tree is labeled with a nonterminal symbol
- Every leaf is labeled with a terminal symbol.

Parse Tree



6.1.6 Parse Tree – Practice Question

For a language L(G), with a CFG G as shown below

```
S → aAs
  | a,
A → SbA,
  | SS
  | ba
S is the start symbol
```

```
Uppercase letters are Nonterminal
Lowercase letters are terminal
```

determine whether x∈L(G) if x is a string as shown below

```
aabbaa
```

- Try with leftmost derivation and rightmost derivation then draw corresponding parse trees

6.1.6 Parse Tree – Practice Question-Solution

G

```
S → aAs      (1)
  | a,       (2)
A → SbA,     (3)
  | SS       (4)
  | ba       (5)
S is the start symbol
```

Leftmost Derivation

aabbaa

$S \overset{L}{\Rightarrow}$	aAS	using (1)
$\overset{L}{\Rightarrow}$	aSbAS	using (3)
$\overset{L}{\Rightarrow}$	aabAS	using (2)
$\overset{L}{\Rightarrow}$	aabbaS	using (5)
$\overset{L}{\Rightarrow}$	aabbaa	using (2)

Rightmost Derivation

aabbaa

$S \overset{R}{\Rightarrow}$	aAS	using (1)
$\overset{R}{\Rightarrow}$	aAa	using (2)
$\overset{R}{\Rightarrow}$	aSbAa	using (3)
$\overset{R}{\Rightarrow}$	aSbbaa	using (5)
$\overset{R}{\Rightarrow}$	aabbaa	using (2)

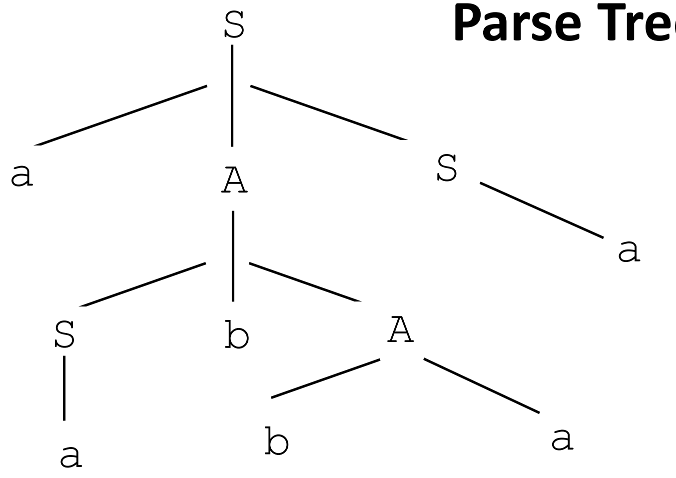
6.1.6 Parse Tree – Practice Question-Solution

Leftmost Derivation

aabbaa

$$
\begin{aligned}
S &\overset{L}{\Rightarrow} aAS && \text{using (1)}\\
&\overset{L}{\Rightarrow} aSbAS && \text{using (3)}\\
&\overset{L}{\Rightarrow} aabAS && \text{using (2)}\\
&\overset{L}{\Rightarrow} aabbaS && \text{using (5)}\\
&\overset{L}{\Rightarrow} aabbaa && \text{using (2)}
\end{aligned}
$$

Parse Tree

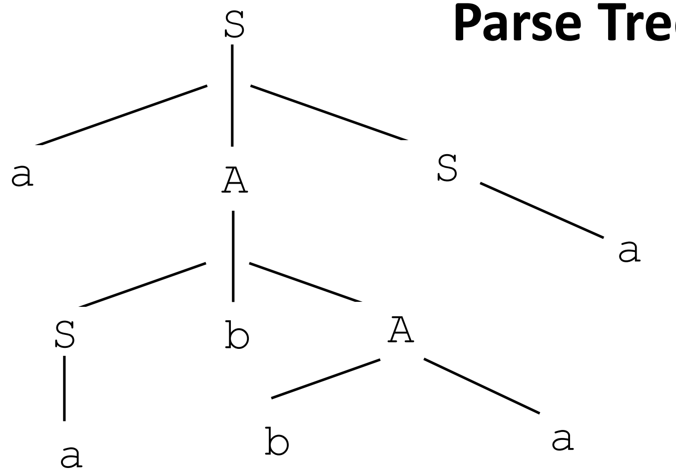


Rightmost Derivation

aabbaa

$$
\begin{aligned}
S &\overset{R}{\Rightarrow} aAS && \text{using (1)}\\
&\overset{R}{\Rightarrow} aAa && \text{using (2)}\\
&\overset{R}{\Rightarrow} aSbAa && \text{using (3)}\\
&\overset{R}{\Rightarrow} aSbbaa && \text{using (5)}\\
&\overset{R}{\Rightarrow} aabbaa && \text{using (2)}
\end{aligned}
$$

Parse Tree



6.2.1 Ambiguity in Grammars

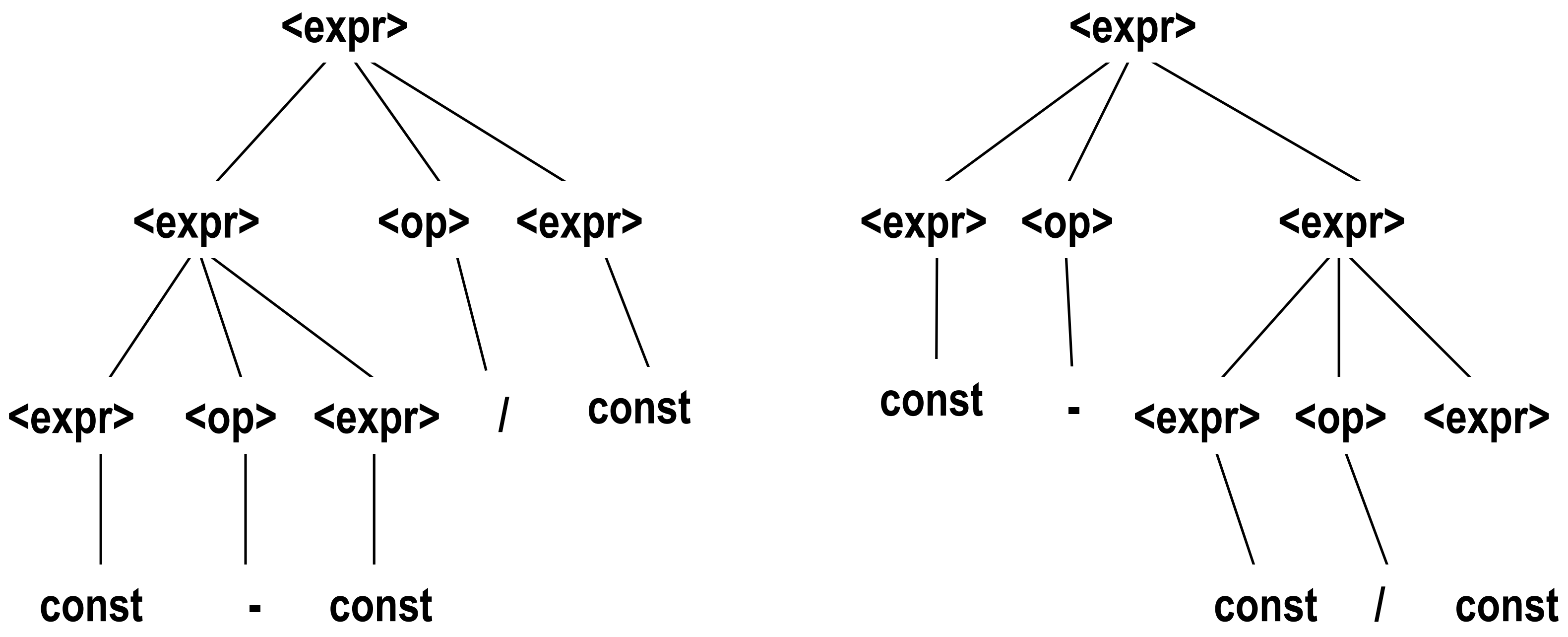
- A grammar is *ambiguous* if there is a string for which there are two different parse trees.

```
<expr> → <expr> <op> <expr> | const
<op> → /  |  -
```

Grammar

```
const-const/const
```

String



Two different parse trees

Is it
(const-const)/const
or
const-(const/const)?

6.2.2 Ambiguity in Grammars – Practice Question

- Draw two different parse trees for `a+b*c` with given grammar

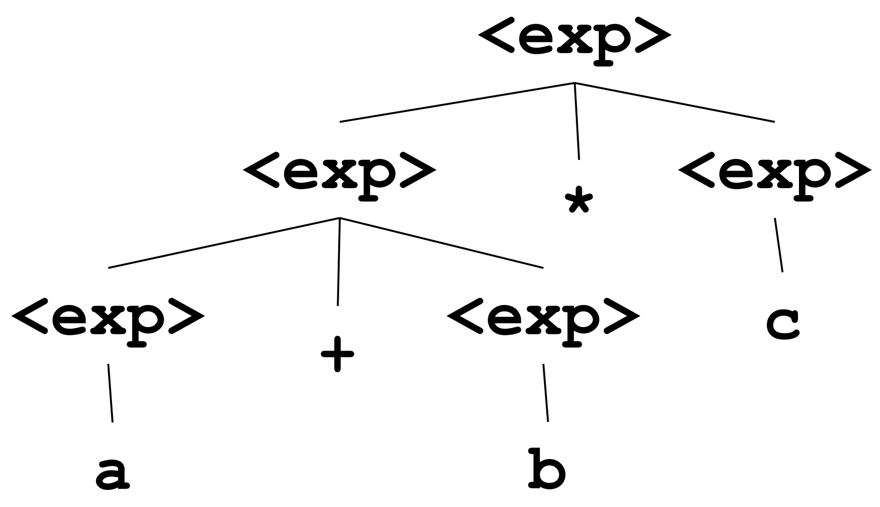
Grammar

```
<exp> → <exp> + <exp> | <exp> * <exp>
      | (<exp>)| a | b | c
```

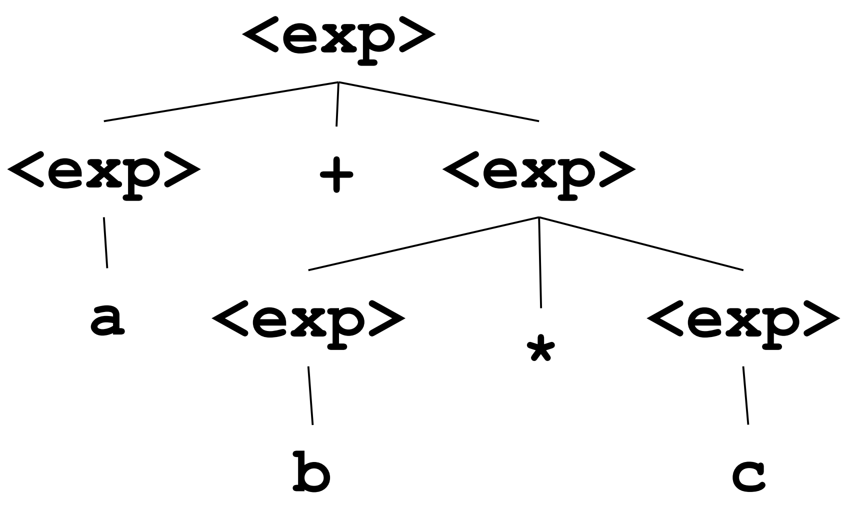
6.2.2 Ambiguity in Grammars – Practice Question Solution

- Two different parse trees for `a+b*c`

```
<exp> → <exp> + <exp> | <exp> * <exp>
      | (<exp>) | a | b | c
```



Meaning: (a+b)*c



Meaning: a+(b*c)

6.2.3 Consequences of Ambiguity

- The compiler will generate different codes, depending on which parse tree it builds
  - In previous example, according to convention, we would like to use the parse tree at the right, i.e., performing a+(b*c)

- Cause of the problem:
  Grammar lacks semantic of *operator precedence*
  - Applies when the order of evaluation is not completely decided by parentheses
  - Each operator has a precedence level, and those with higher precedence are performed before those with lower precedence, as if parenthesized

6.2.4 Putting Semantics into Grammar

```
<exp> → <exp> + <exp> | <exp> * <exp>
      | (<exp>) | a | b | c
```

- To fix the precedence problem, we modify the grammar so that it is forced to put * below + in the parse tree

```
   <exp> → <exp> + <exp> | <mulexp>
<mulexp> → <mulexp> * <mulexp>
         | (<exp>) | a | b | c
```

Note the hierarchical structure of the production rules

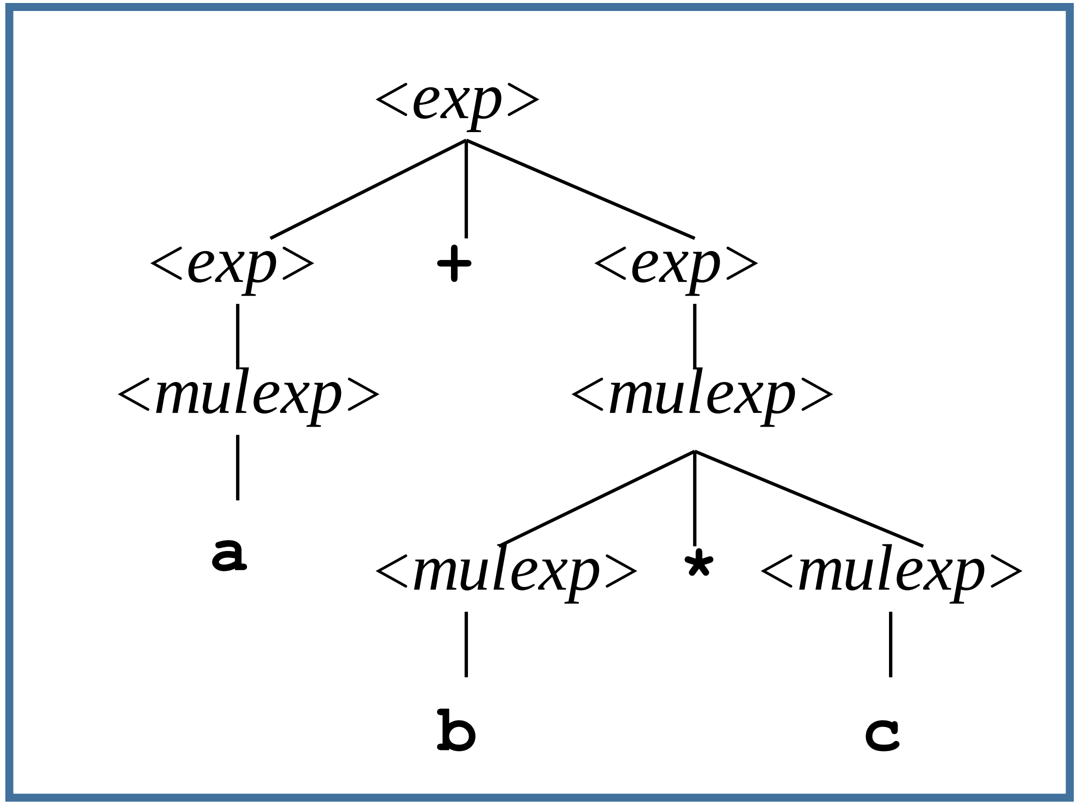
6.2.5 Putting Semantics into Grammar – Practice Question

- Is it possible to have 2 different parse trees for **`a+b*c`** with following modified grammar?

```
    <exp> → <exp> + <exp> | <mulexp>
 <mulexp> → <mulexp> * <mulexp>
            | (<exp>)| a | b | c
```

<u>6.2.5 Putting Semantics into Grammar</u> – Practice Question Solution

a+b*c



Correct Precedence

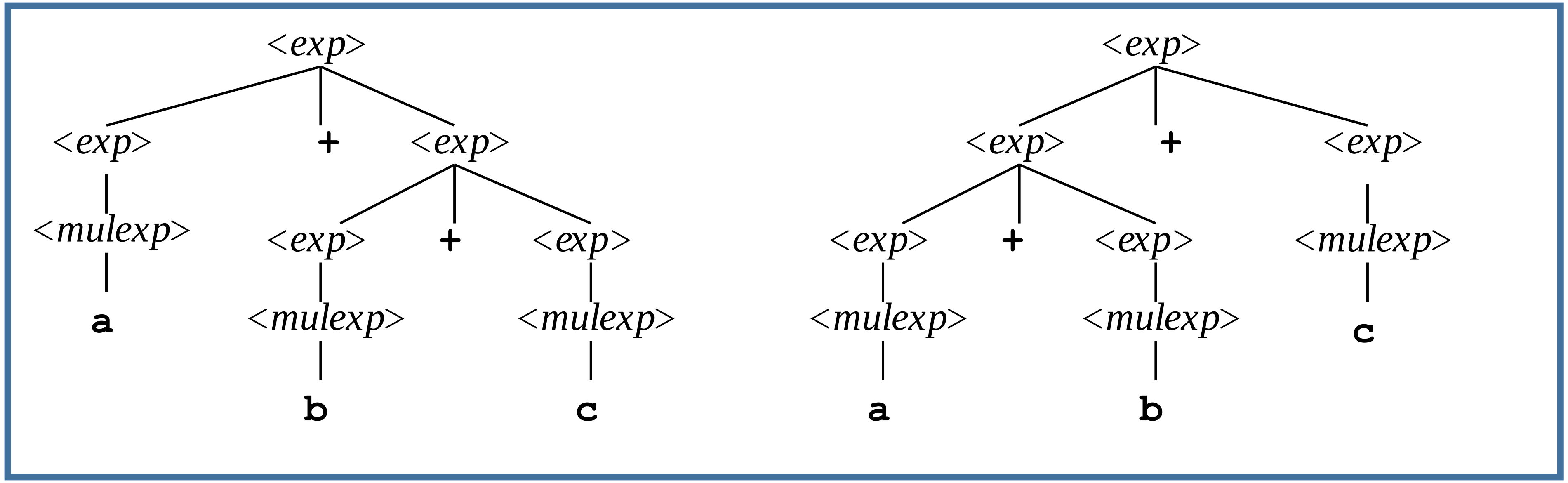
- Our new grammar generates same language as before, but no longer generates parse trees with incorrect precedence.

6.2.6 Semantics of Associativity

- Grammar can also handle the semantics of operator associativity

```
<exp> → <exp> + <exp> | <mulexp>
<mulexp> → <mulexp> * <mulexp>
           | (<exp>) | a | b | c
```

a+b+c



Associativity: how operators of the same precedence are grouped in the absence of parentheses

6.2.7 Operator Associativity

- Applies when the order of evaluation is not decided by parentheses or by precedence
- *Left-associative operators* group operands left to right: a+b+c+d = ((a+b)+c)+d
- *Right-associative operators* group operands right to left: a+b+c+d = a+(b+(c+d))
- Most operators in most languages are left-associative, but there are exceptions, e.g., C

a=b=0	— right-associative (assignment)

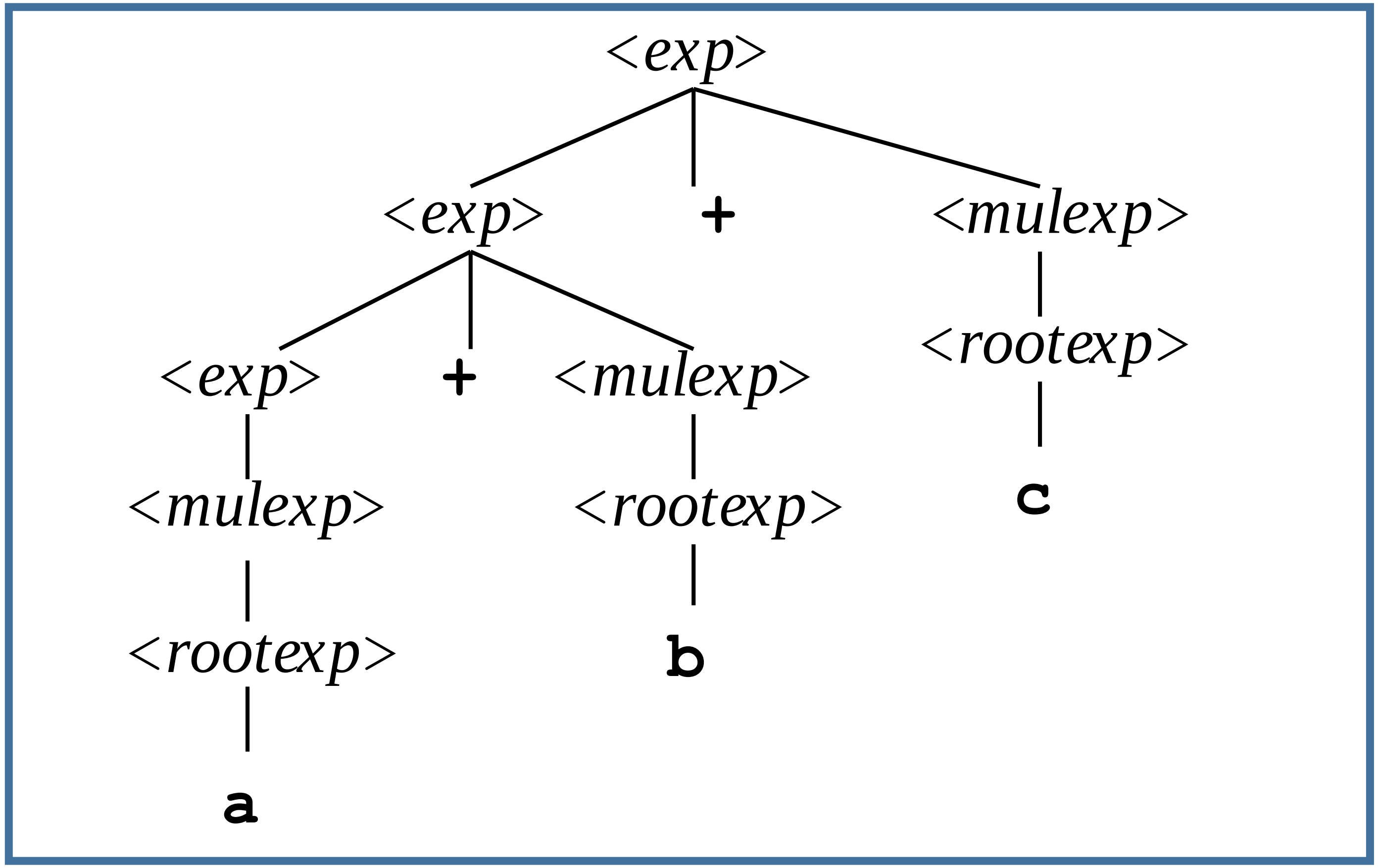
6.2.8 Associativity in the Grammar

```
   <exp> → <exp> + <exp> | <mulexp>
<mulexp> → <mulexp> * <mulexp>
         | (<exp>)| a | b | c
```

- To fix the associativity problem, we modify the grammar to make trees of +s grow down to the left (and likewise for *s)

```
    <exp> → <exp> + <mulexp> | <mulexp>
 <mulexp> → <mulexp> * <rootexp> | <rootexp>
<rootexp> →(<exp>)| a | b | c
```

6.2.9 Correct Associativity



- Our new grammar generates same language as before with intended associativity

6.3.1 Origin of EBNF

- Stands for “Extended Backus-Naur Form”

- Increases readability and writability

6.3.2 Extended BNF

- Optional parts are placed in brackets [ ]

```
<if_stmt>->if(<expression>)<statement>[else(statement)]
```

Otherwise, it would have been

```
<if_stmt>->if(<expression>)<statement>
          |if(<expression>)<statement>else(statement)
```

6.3.3 Extended BNF

- Repetitions (0 or more) are placed inside braces { }

```
<stmts> → <stmt>{;<stmt>}
```

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

```
value → (+|-)integer
```

Otherwise, it would have been

```
value →  +integer|-integer
```

6.3.4 BNF and EBNF

- BNF

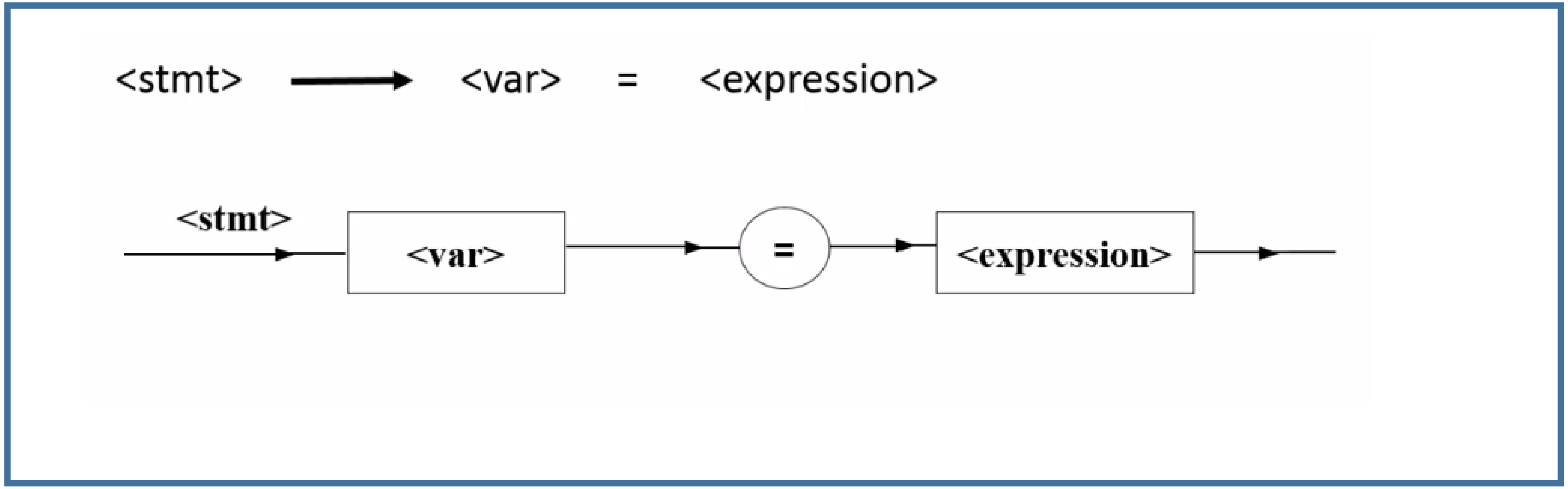
```
<expr> → <expr> + <term>
       | <expr> - <term>
       | <term>
<term> → <term> * <factor>
       | <term> / <factor>
       | <factor>
```

- EBNF

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

6.4.1 Syntax Graph/ Syntax Diagram

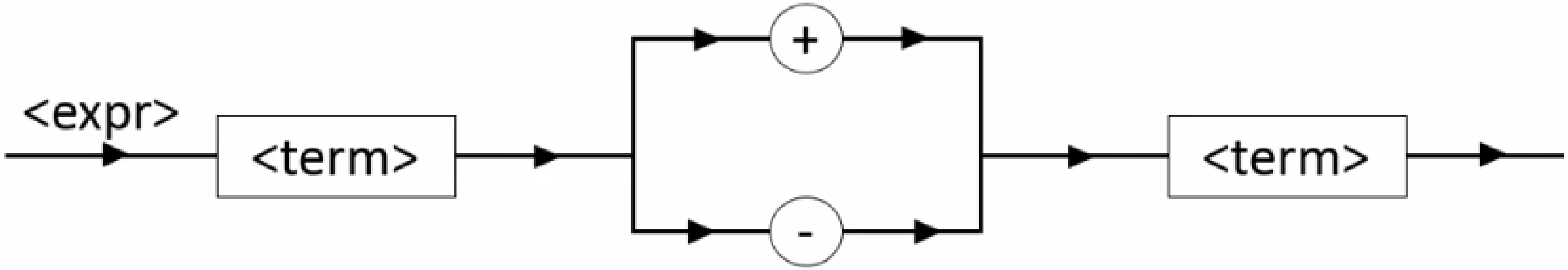
- They represent a graphical alternative to BNF or EBNF



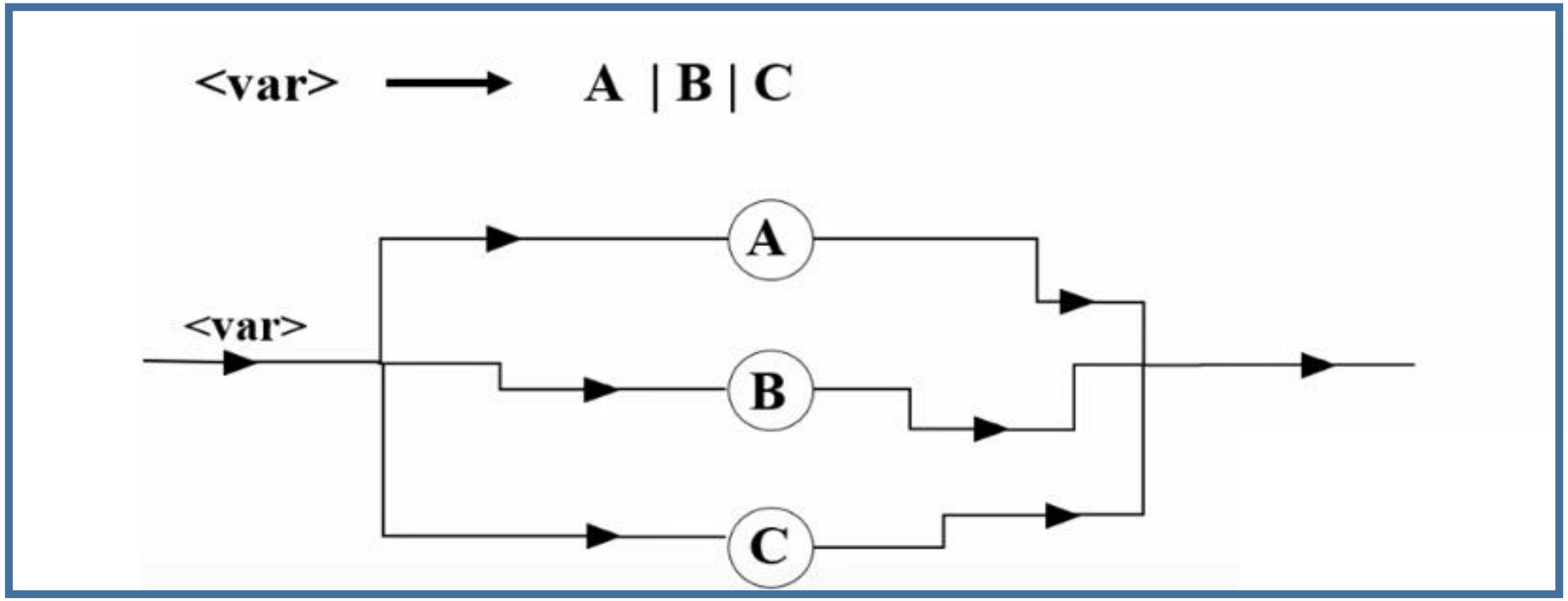
Note!! Terminals in circle and nonterminals in rectangle

6.4.2 Syntax Graph/ Syntax Diagram

<expr> ⟶ <term> + <term> | <term> - <term>



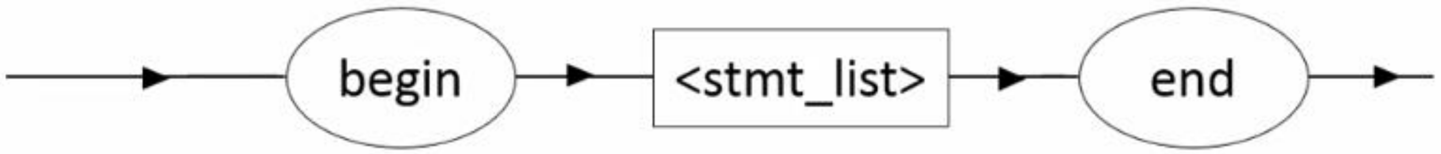
6.4.3 Syntax Graph/ Syntax Diagram



Note!! Terminals in circle and nonterminals in rectangle

6.4.4 Syntax Graph/ Syntax Diagram

<program> ⟶ begin <stmt_list> end



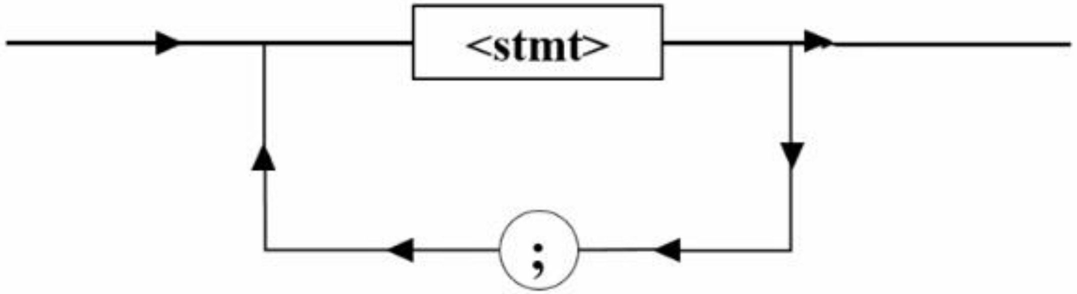
6.4.5 Syntax Graph/Syntax Diagram – Practice Question

Q) Draw syntax diagram for

<stmt> → <stmt> | ; <stmt>

6.4.5 Syntax Graphs – Practice Question - Solution

<stmt> ⟶ <stmt> | ; <stmt>



Q & A