Faculty of Information Technology

Spring 2025

Concepts of Programming Languages

CS 211

Lecture (3)

Outline

- Parse Trees
- Ambiguity
- Unambiguous Grammar

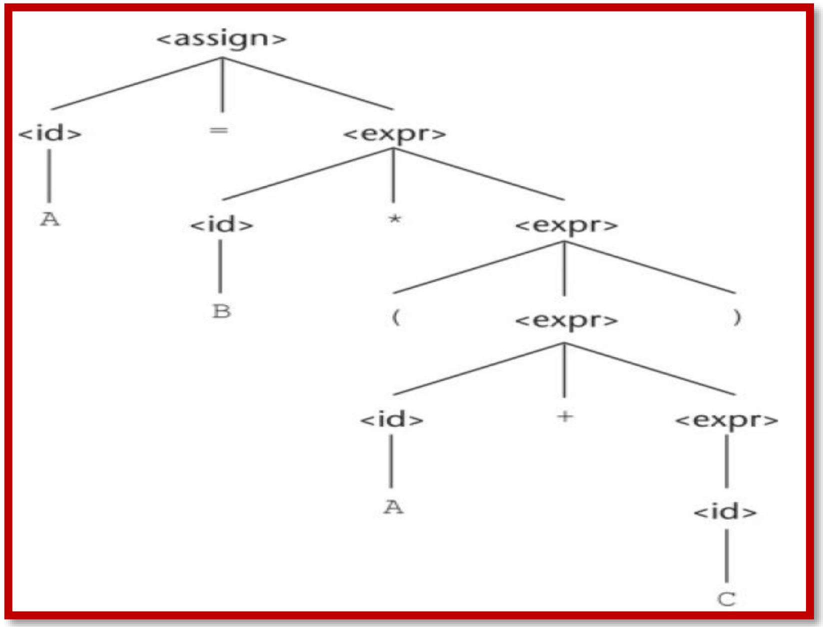
Outline

- Parse Trees
- Ambiguity
- Unambiguous Grammar

Parse Trees

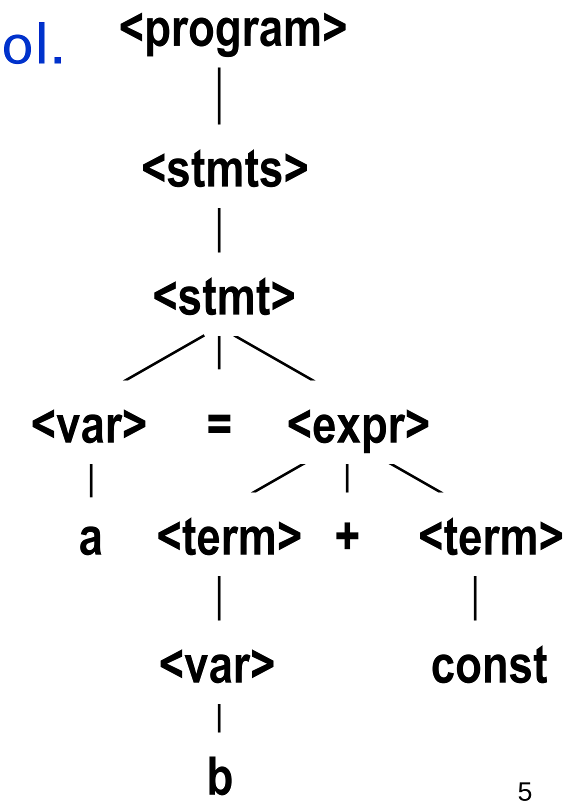
- One of the most attractive features of grammars is that they naturally describe the hierarchical syntactic structure of the sentences of the languages they define.
- For example, the following parse tree shows the structure of the assignment statement derived previously.

A Parse Tree for the Simple Statement
A = B * (A + C)



Parse Trees (Cont.)

- A hierarchical representation of a derivation.
- Every internal node of a parse tree is labelled with a non-terminal symbol.
- Every leaf is labelled with a terminal symbol.



Example (1)

begin B = C ; A = B + C end

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>

Parse Tree use
Leftmost

Example (1) (Cont.)

<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 B = C ; A = B + C end

<program>

Example (1) (Cont.)

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

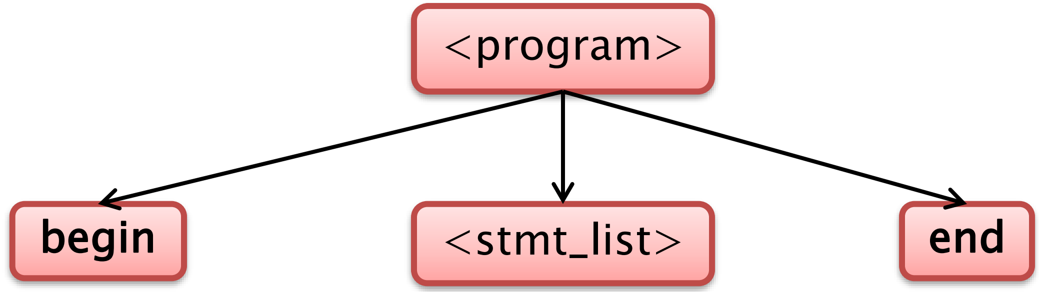
<stmt_list> → <stmt>
  | <stmt> ; <stmt_list>

<stmt> → <var> = <expression>

<var> → A | B | C

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

begin B = C ; A = B + C end



Example (1) (Cont.)

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

<stmt_list> → <stmt>

| <stmt> ; <stmt_list>

<stmt> → <var> = <expression>

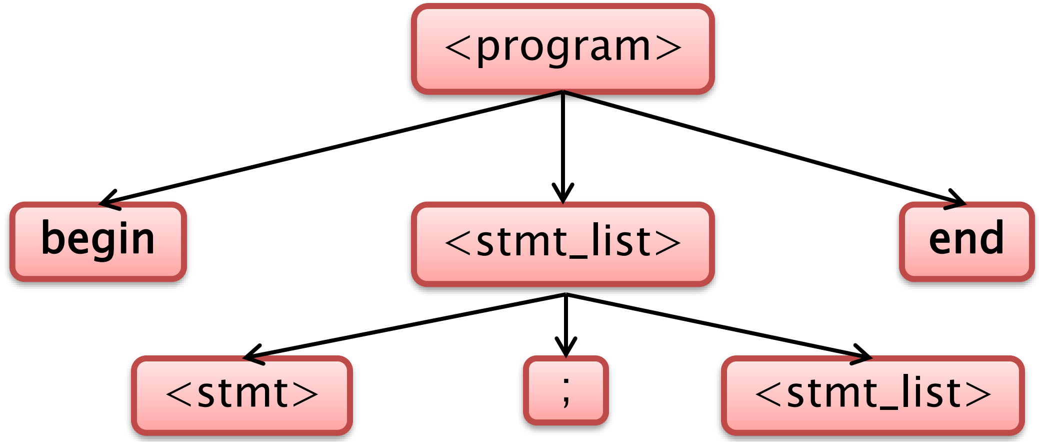
<var> → A | B | C

<expression> → <var> + <var>

| <var> – <var>

| <var>

begin B = C ; A = B + C end



Example (1) (Cont.)

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

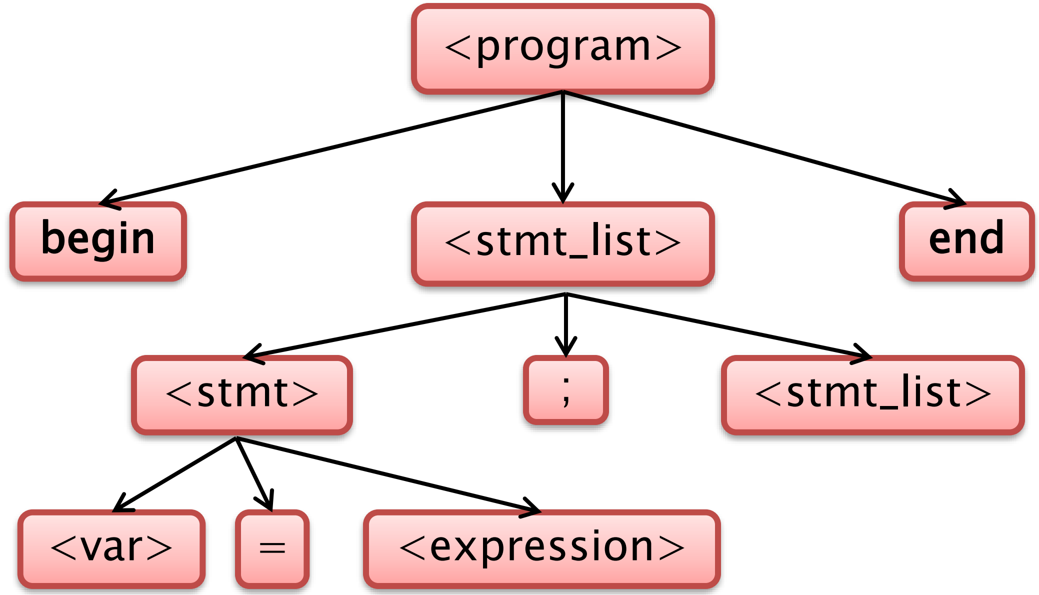
<stmt_list> → <stmt>
| <stmt> ; <stmt_list>

<stmt> → <var> = <expression>

<var> → A | B | C

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

begin B = C ; A = B + C end



Example (1) (Cont.)

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

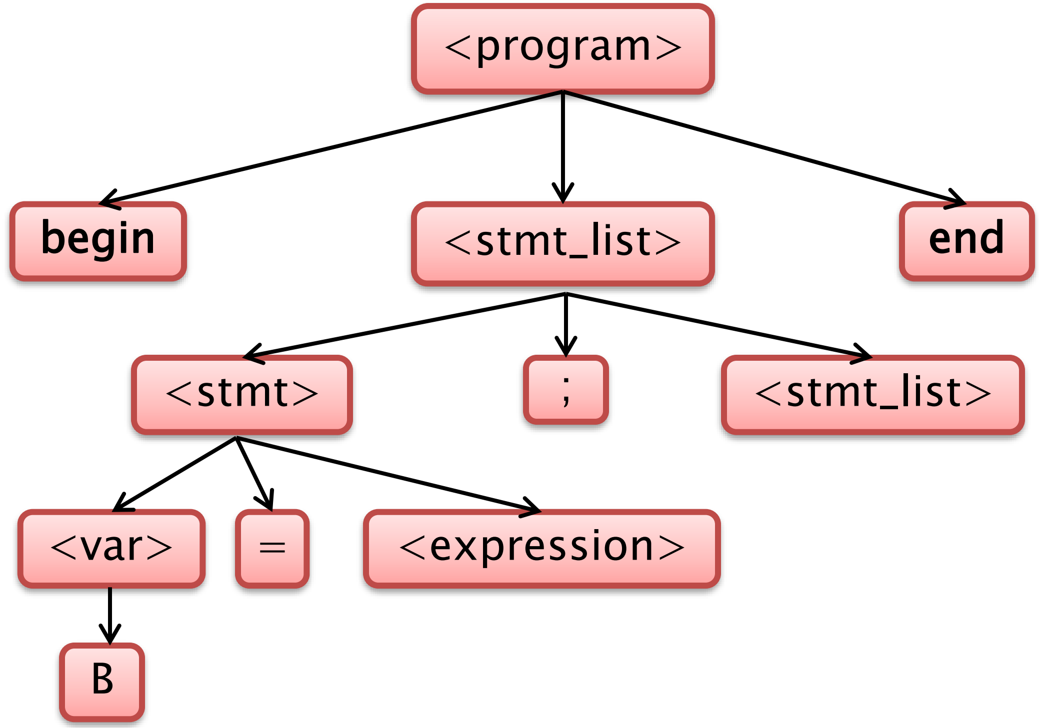
<stmt_list> → <stmt>
| <stmt> ; <stmt_list>

<stmt> → <var> = <expression>

<var> → A | B | C

<expression> → <var> + <var>
| <var> – <var>
| <var>

begin B = C ; A = B + C end



Example (1) (Cont.)

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

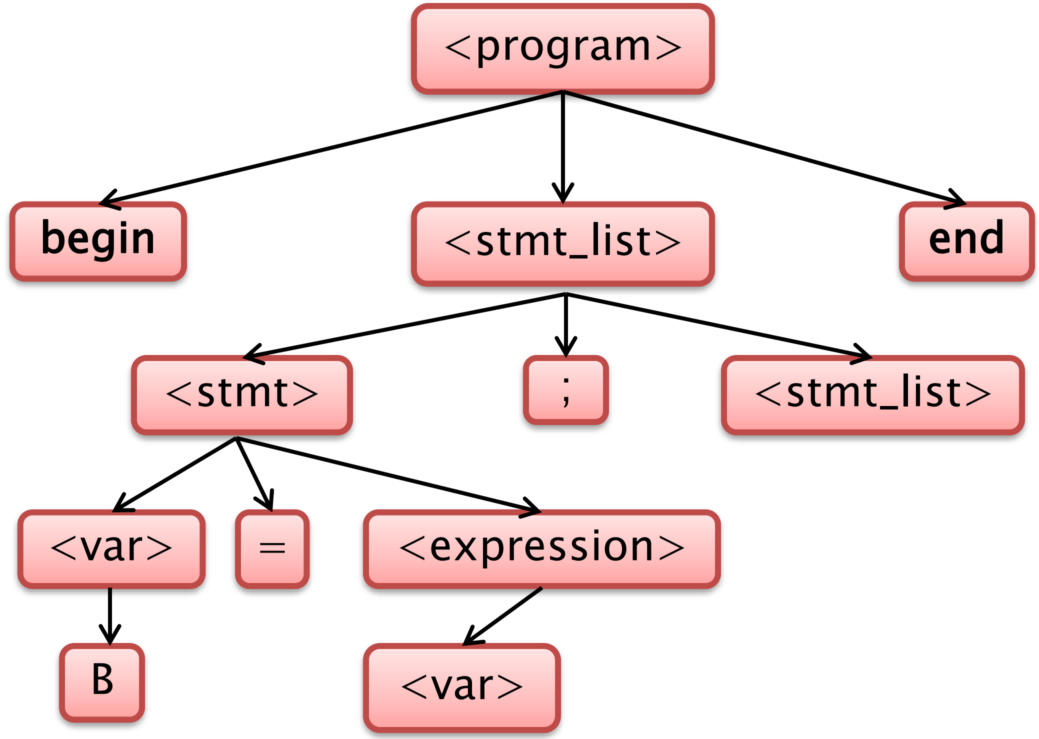
<stmt_list> → <stmt>
| <stmt> ; <stmt_list>

<stmt> → <var> = <expression>

<var> → A | B | C

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

begin B = C ; A = B + C end



Example (1) (Cont.)

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

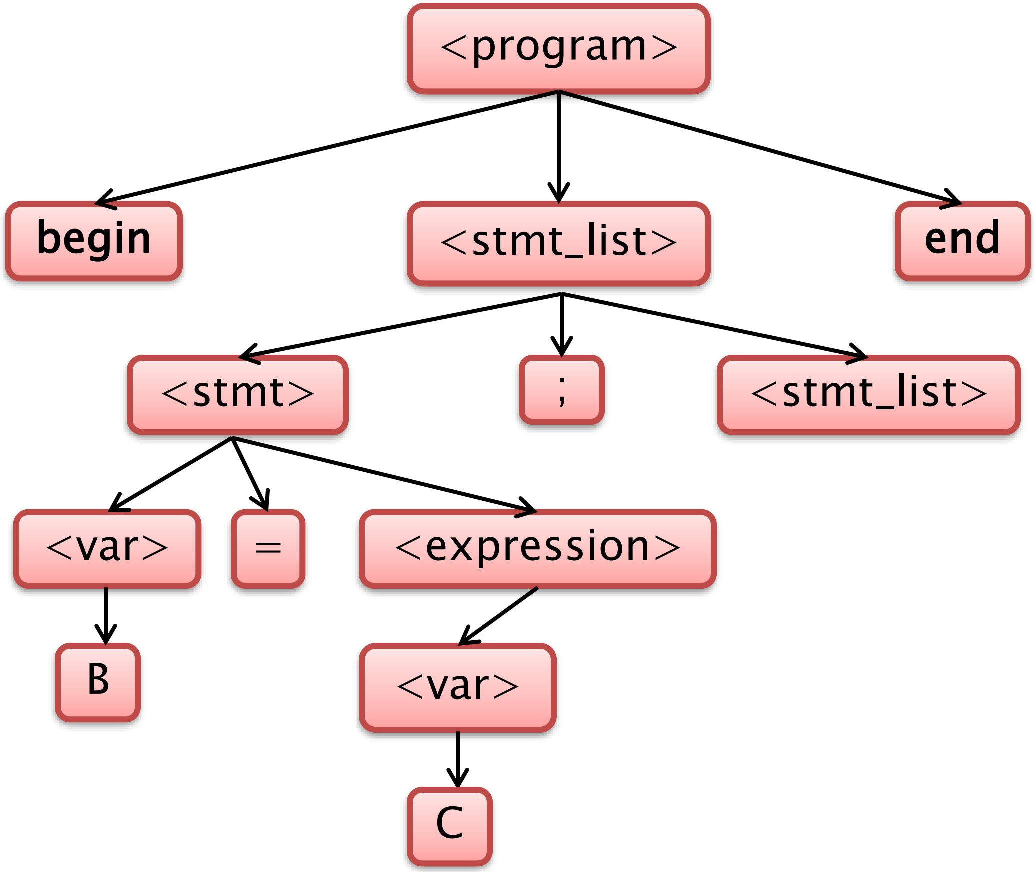
<stmt_list> → <stmt>
| <stmt> ; <stmt_list>

<stmt> → <var> = <expression>

<var> → A | B | C

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

begin B = C ; A = B + C end



Example (3) (Cont.)

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

<stmt_list> → <stmt>

| <stmt> ; <stmt_list>

<stmt> → <var> = <expression>

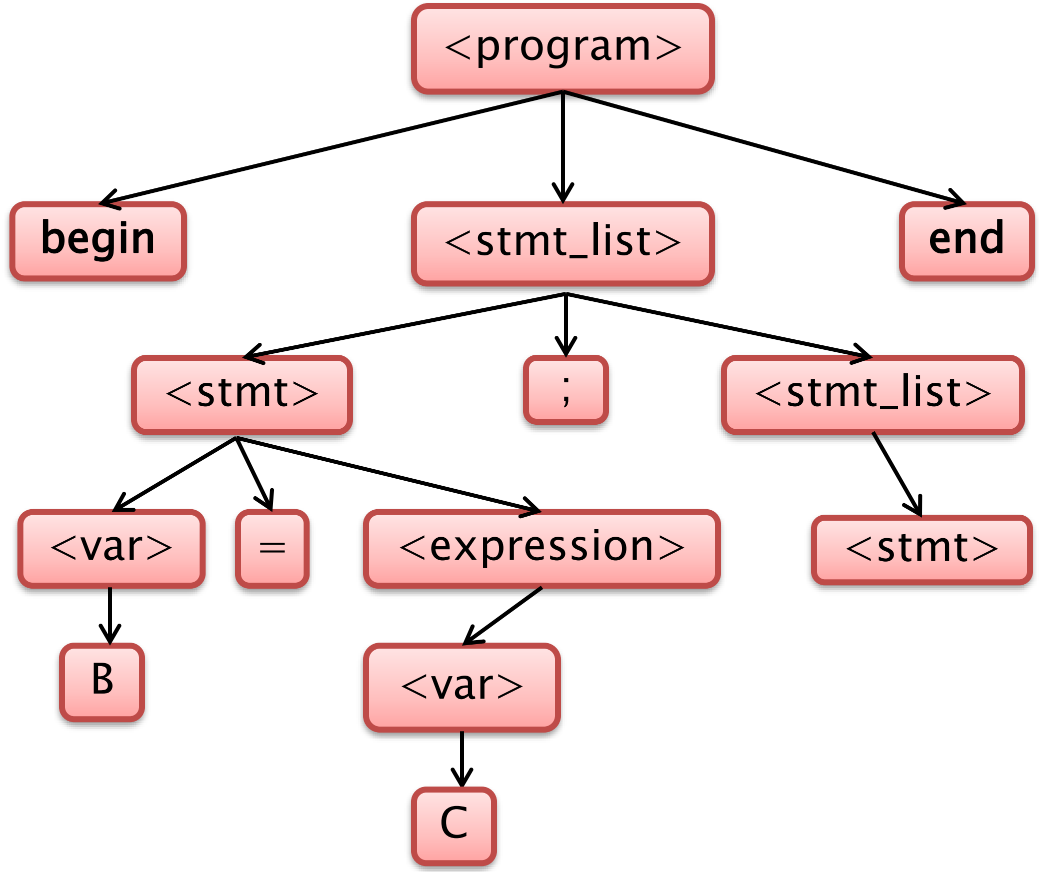
<var> → A | B | C

<expression> → <var> + <var>

| <var> - <var>

| <var>

begin B = C ; A = B + C end



Example (1) (Cont.)

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

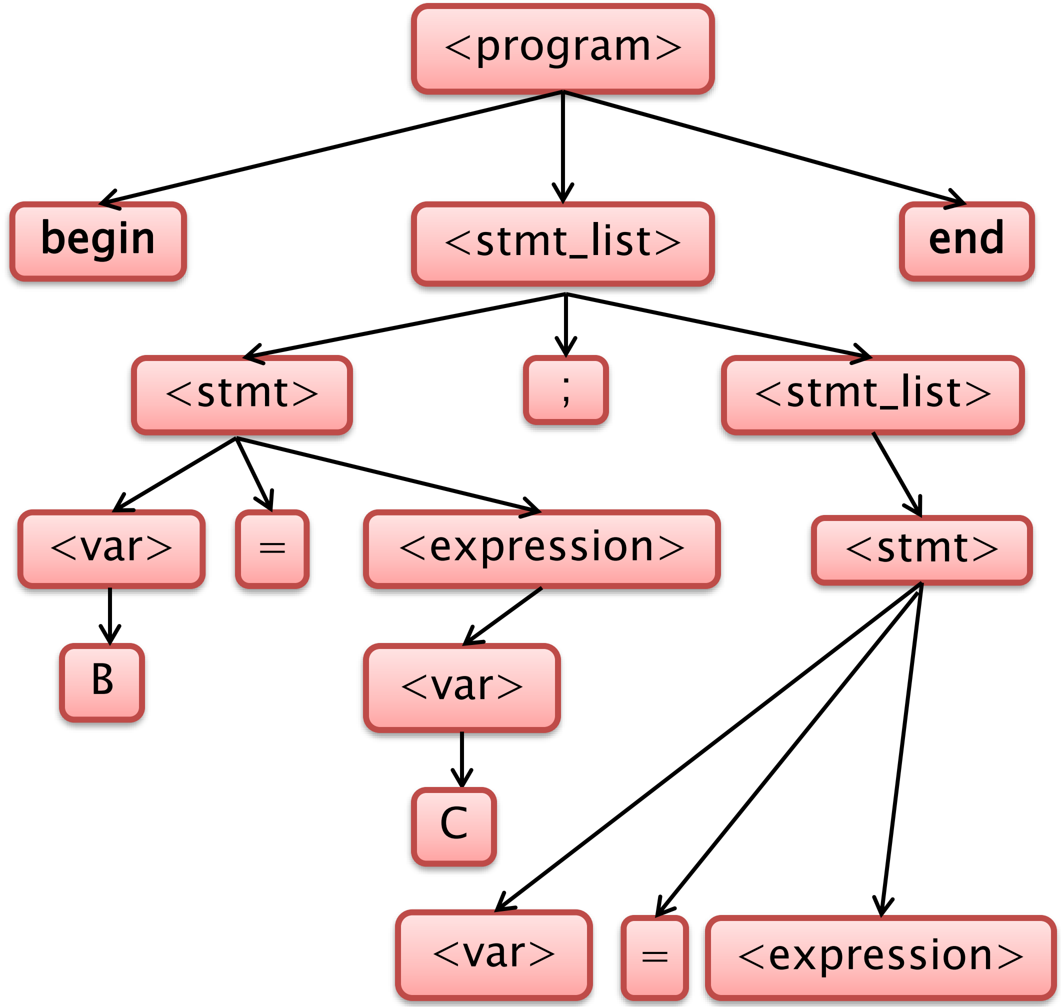
<stmt_list> → <stmt>
| <stmt> ; <stmt_list>

<stmt> → <var> = <expression>

<var> → A | B | C

<expression> → <var> + <var>
| <var> – <var>
| <var>

begin B = C ; A = B + C end



Example (1) (Cont.)

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

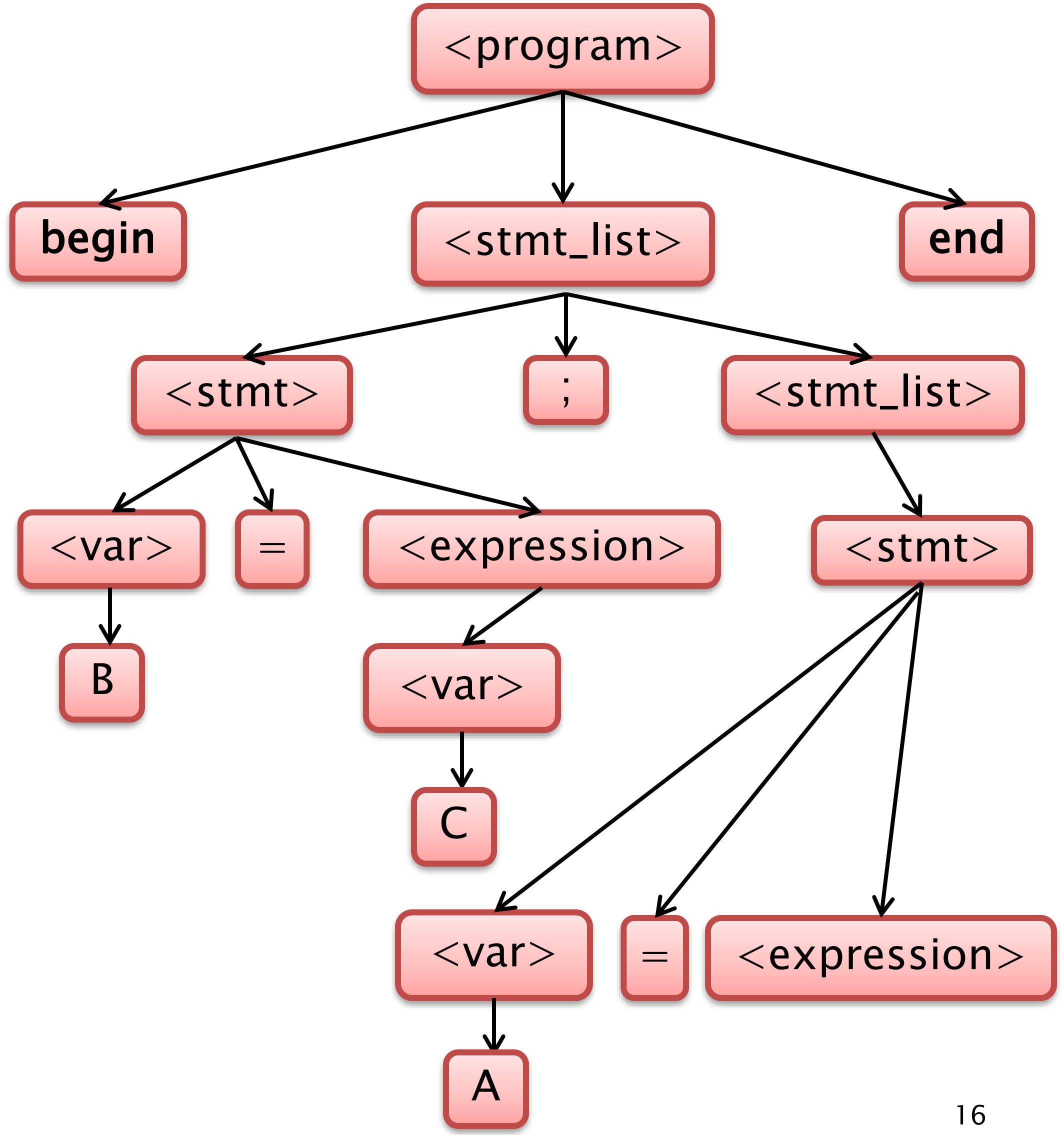
<stmt_list> → <stmt>
| <stmt> ; <stmt_list>

<stmt> → <var> = <expression>

<var> → A | B | C

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

begin B = C ; A = B + C end



Example (1) (Cont.)

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

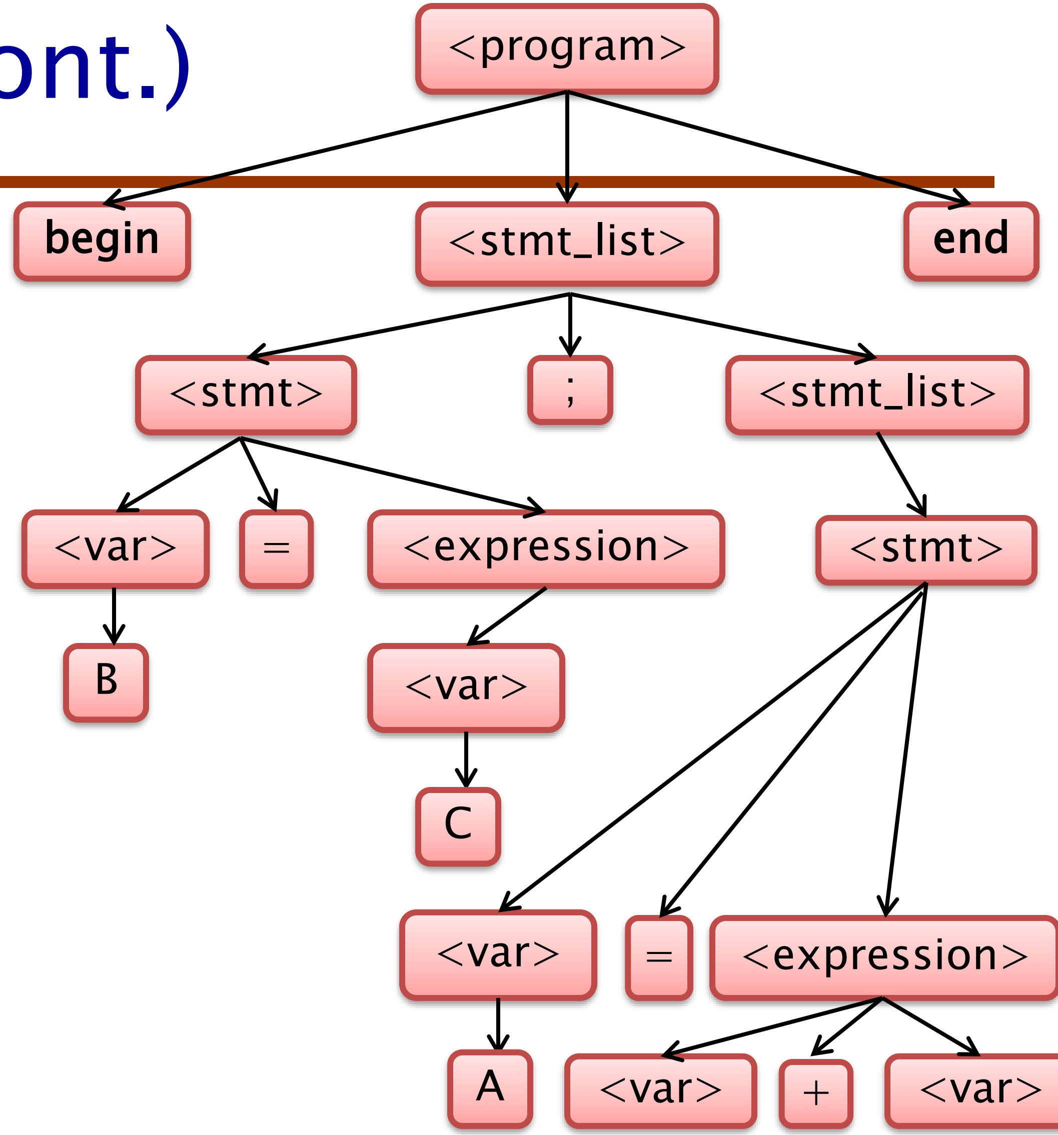
<stmt_list> → <stmt>
| <stmt> ; <stmt_list>

<stmt> → <var> = <expression>

<var> → A | B | C

<expression> → <var> + <var>
| <var> – <var>
| <var>

begin B = C ; A = B + C end



Example (1) (Cont.)

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

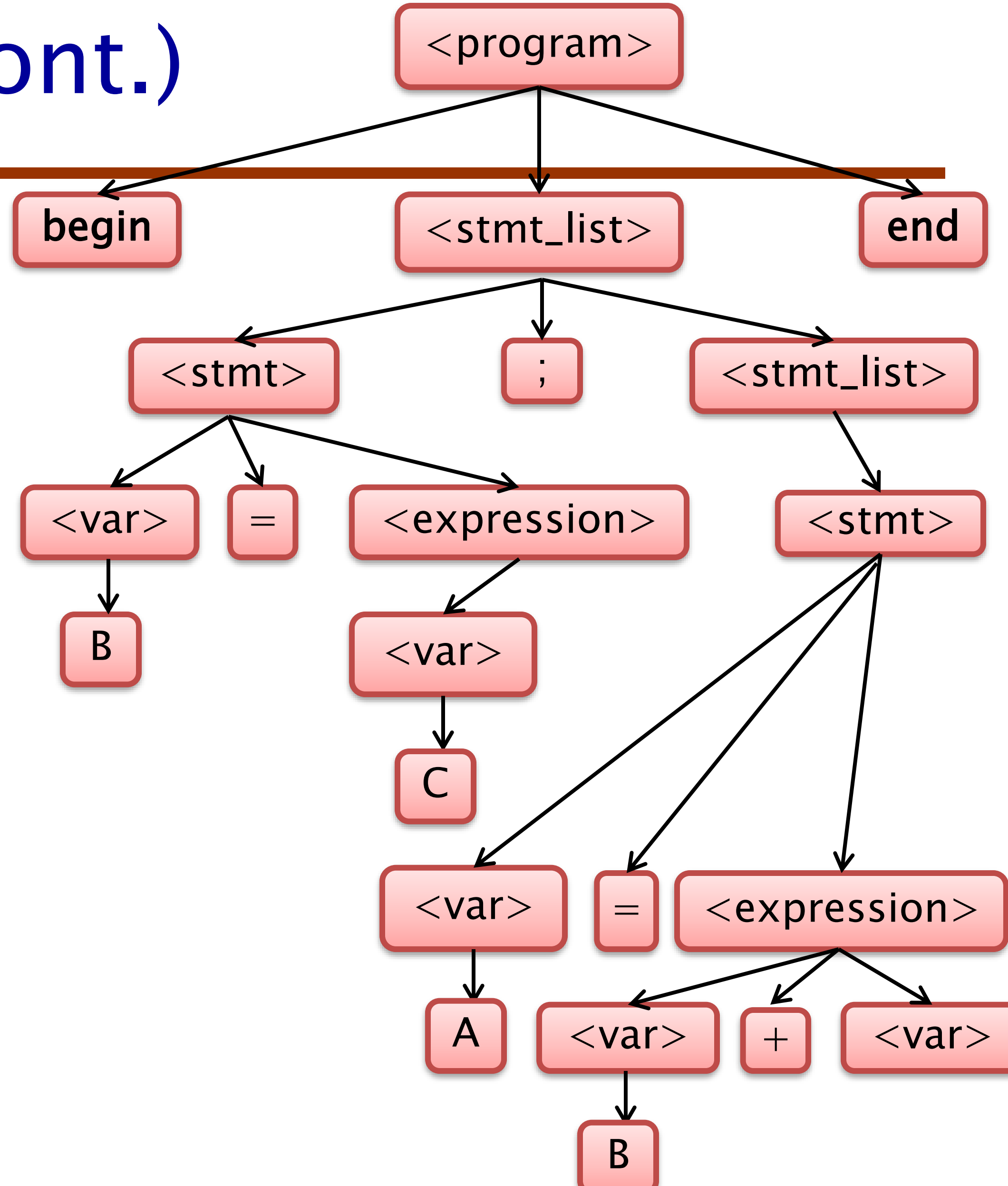
<stmt_list> → <stmt>
| <stmt> ; <stmt_list>

<stmt> → <var> = <expression>

<var> → A | B | C

<expression> → <var> + <var>
| <var> – <var>
| <var>

begin B = C ; A = B + C end



Example (1) (Cont.)

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

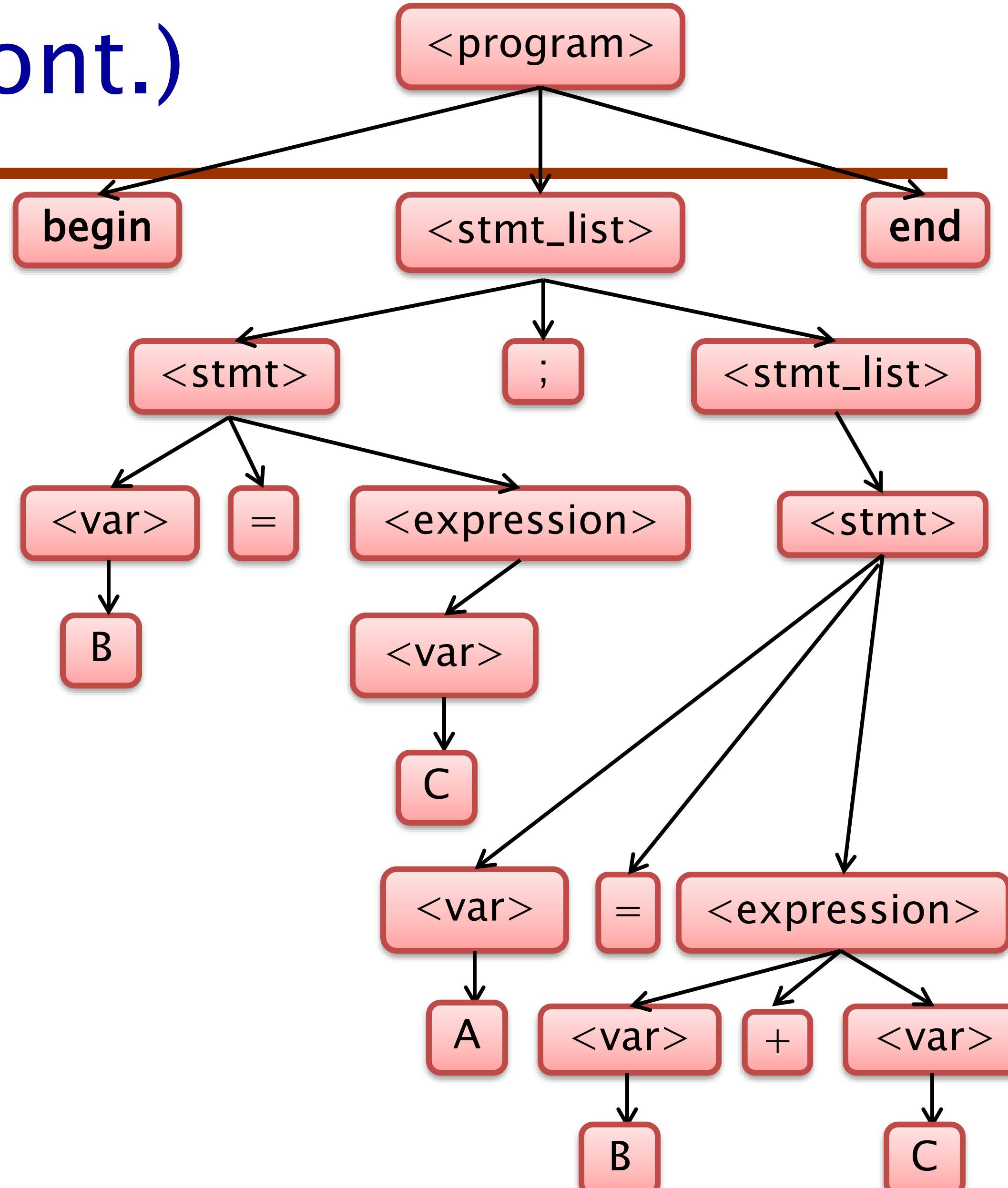
<stmt_list> → <stmt>
| <stmt> ; <stmt_list>

<stmt> → <var> = <expression>

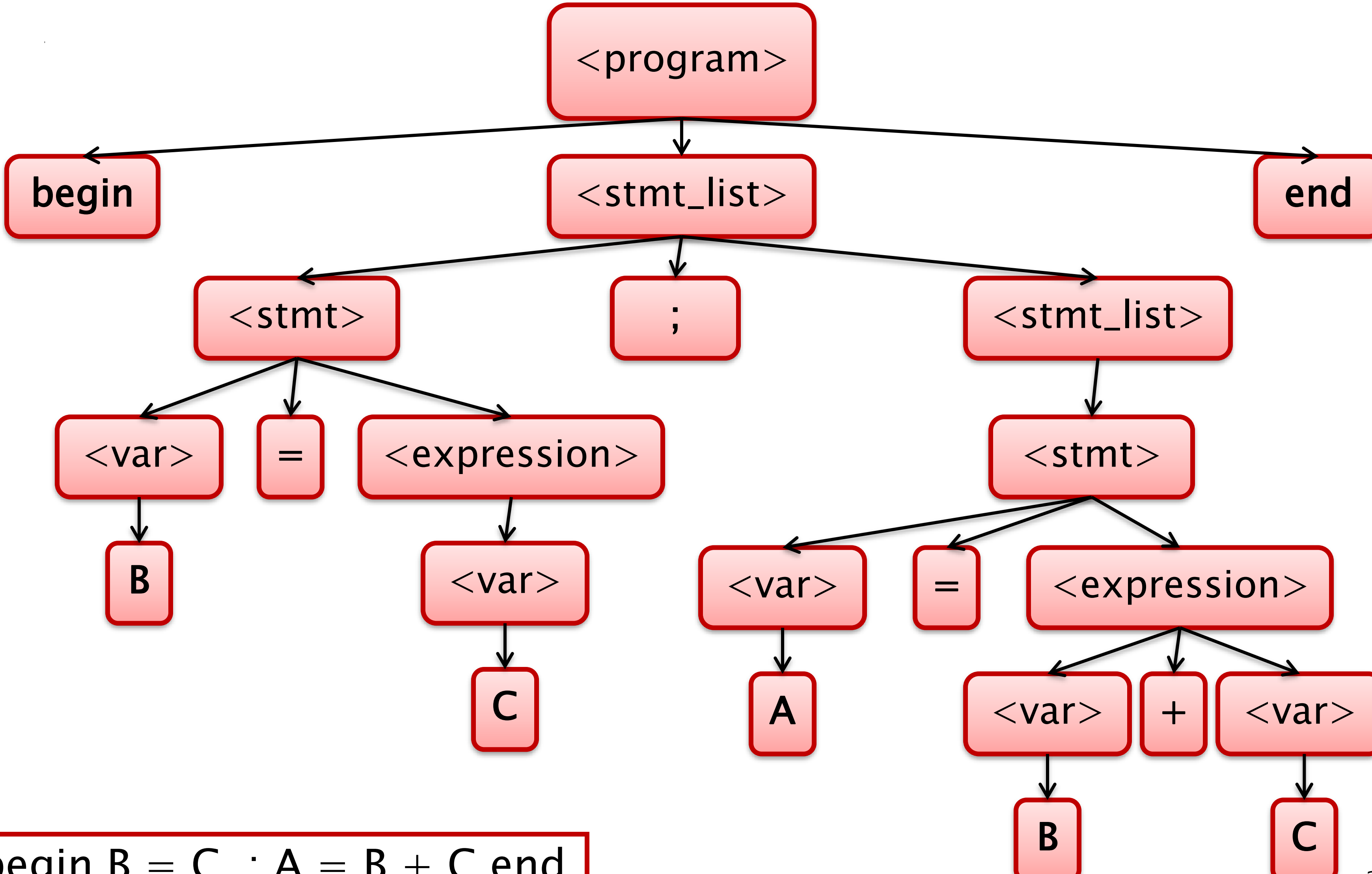
<var> → A | B | C

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

begin B = C ; A = B + C end

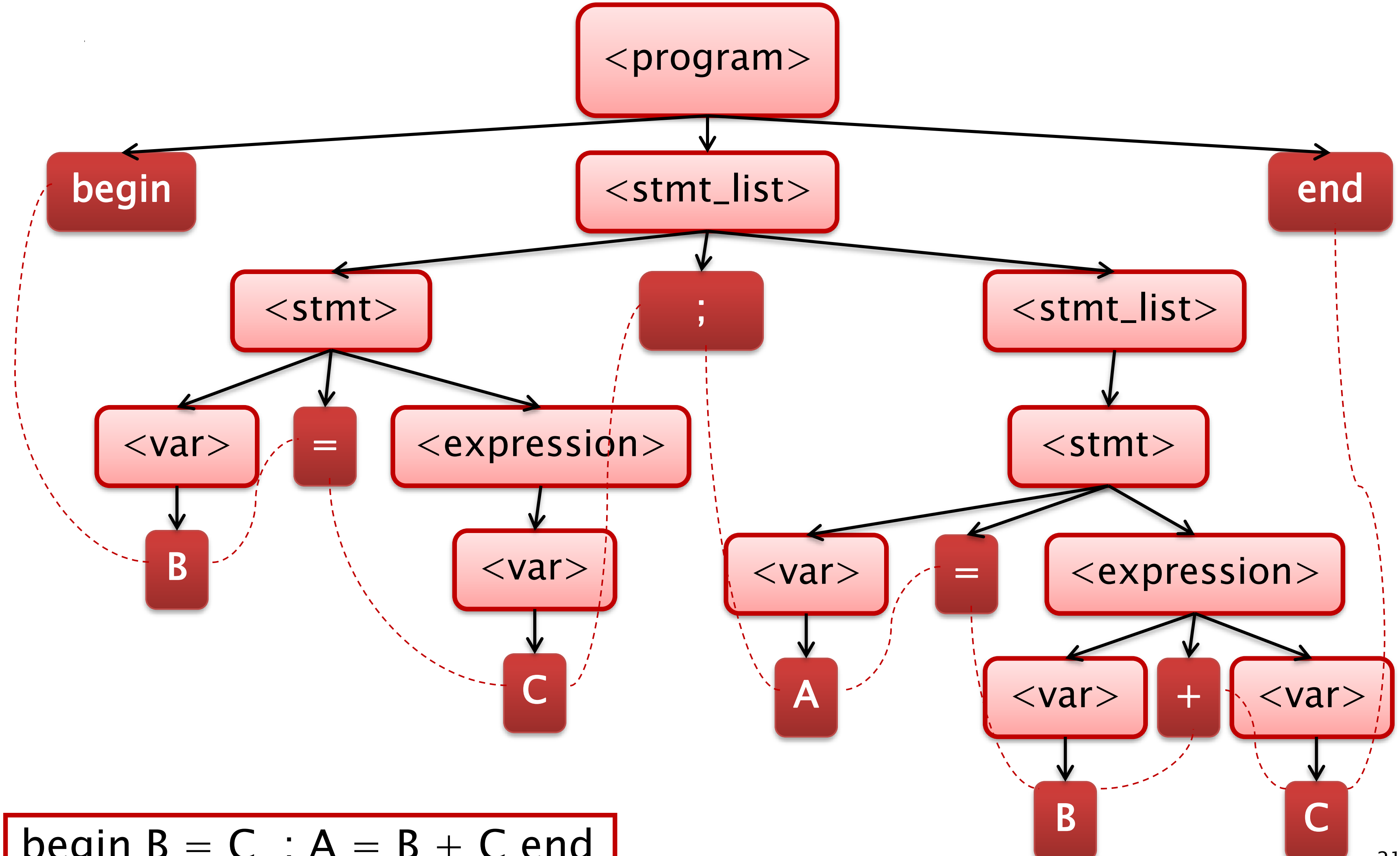


Example (1) (Cont.)



begin B = C ; A = B + C end

Example (1) (Cont.)



begin B = C ; A = B + C end

Example (1) (Cont.)

begin B = C ; A = B + C end

A Grammar for a Small Language

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

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

<stmt> → <var> = <expression>

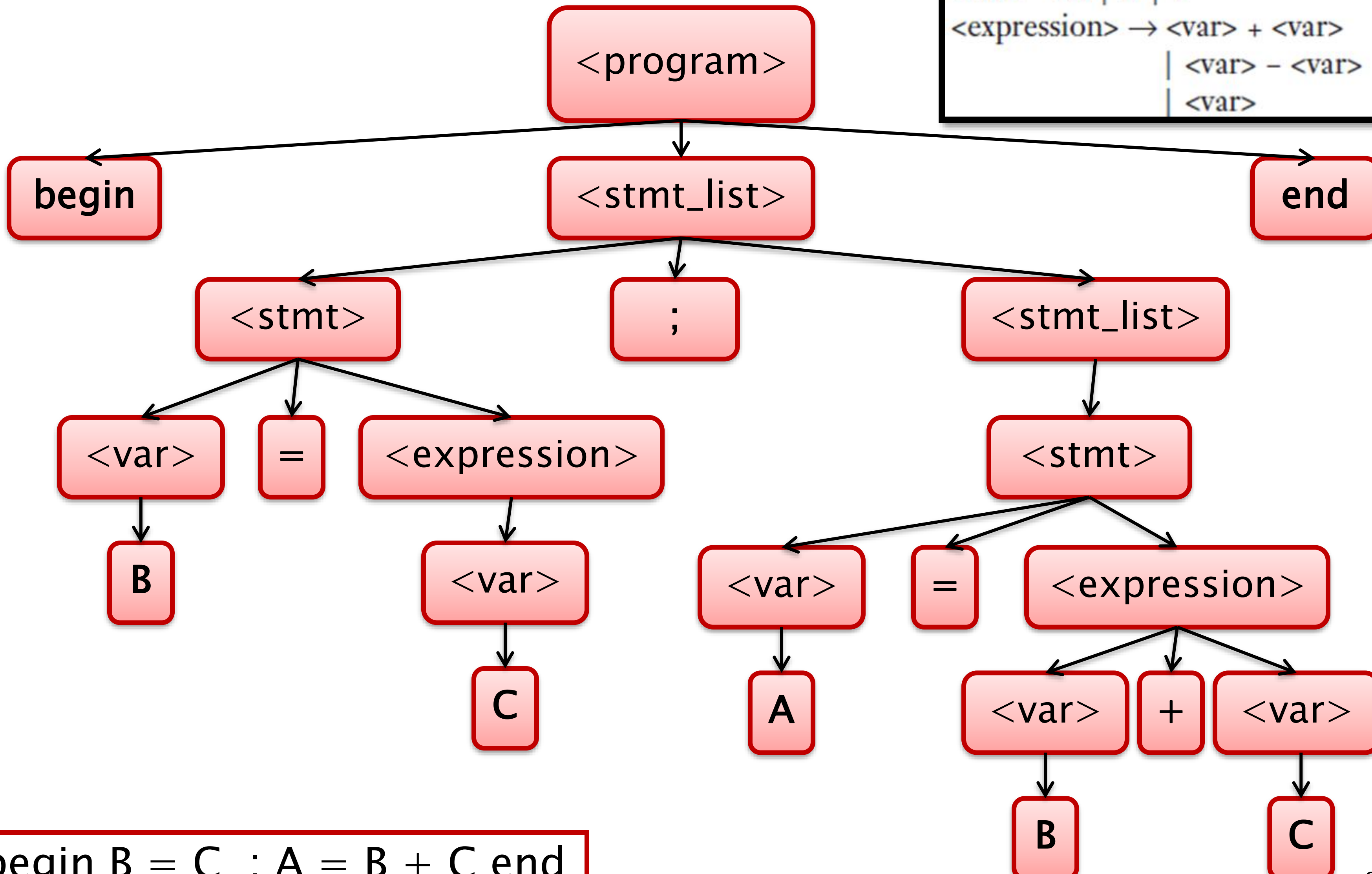
<var> → A | B | C

<expression> → <var> + <var>
| <var> – <var>
| <var>

Parse Tree use
Rightmost

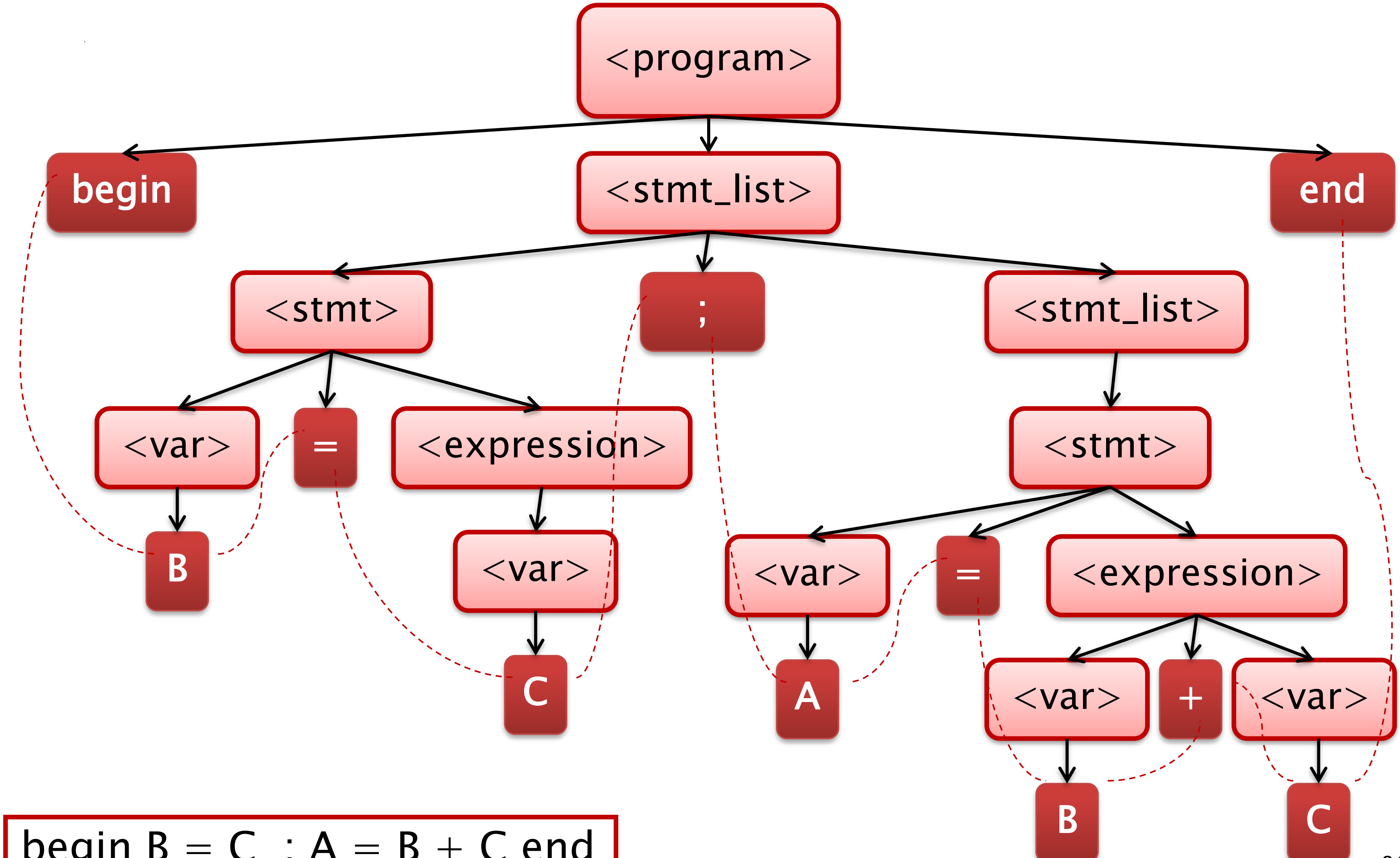
Example (1) (Cont.)

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

Example (1) (Cont.)



begin B = C ; A = B + C end

Example (2)

A = B * (A + C)

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

Example (2) (Cont.)

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

`A = B * ( A + C )`

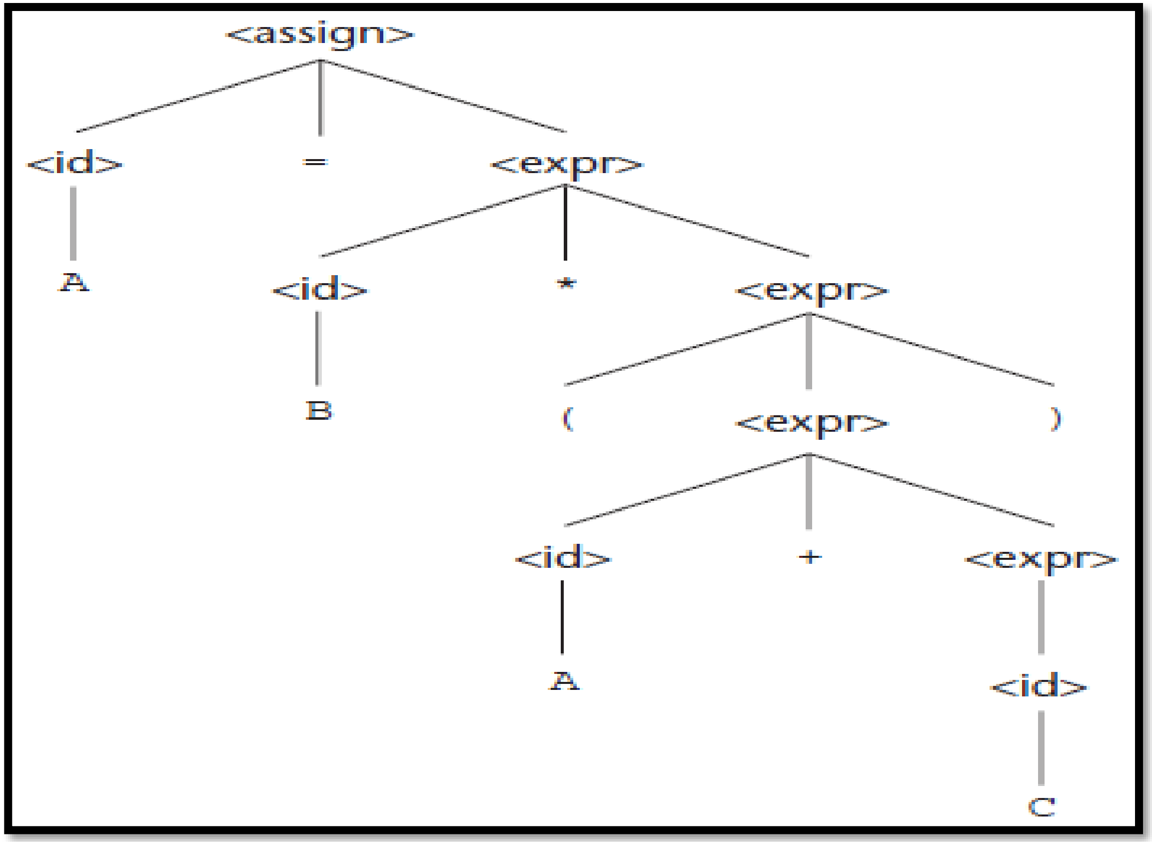
```
<assign> => <id> = <expr>
         => A = <expr>
         => A = <id> * <expr>
         => A = B * <expr>
         => A = B * ( <expr> )
         => A = B * ( <id> + <expr> )
         => A = B * ( A + <expr> )
         => A = B * ( A + <id> )
         => A = B * ( A + C )
```

Accepted

Example (2) (Cont.)

A = B * (A + C)

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



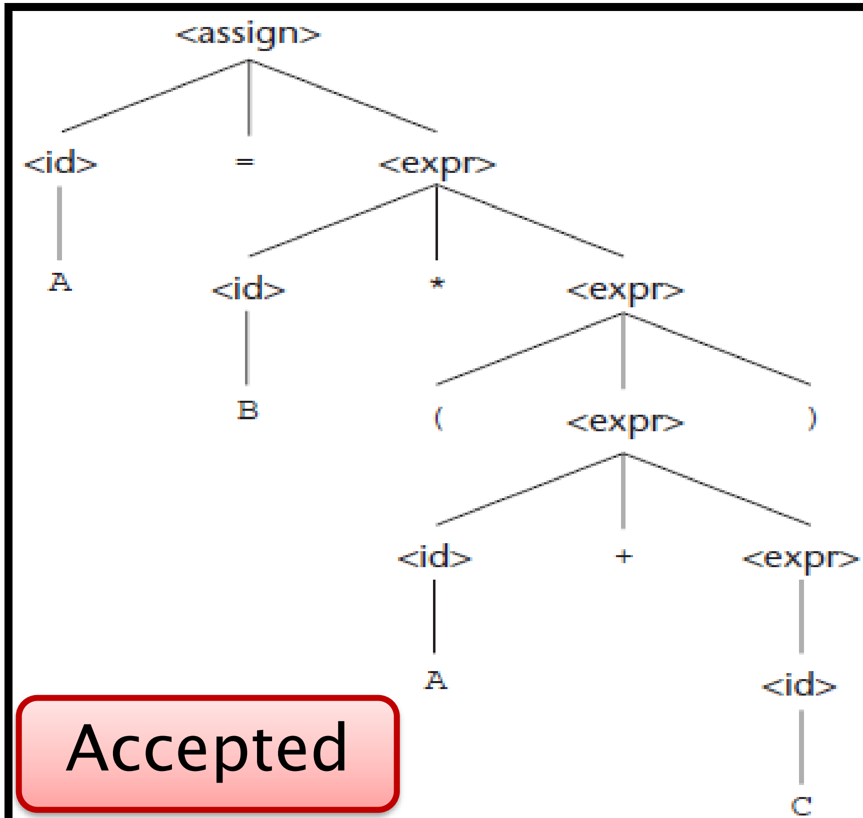
Accepted

Example (2) (Cont.)

A = B * (A + C)

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

<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)



Example (3)

A = B + C * A

$$
\begin{aligned}
\langle assign \rangle &\rightarrow \langle id \rangle = \langle expr \rangle \\
\langle id \rangle &\rightarrow A \mid B \mid C \\
\langle expr \rangle &\rightarrow \langle expr \rangle + \langle expr \rangle \\
&\quad\mid \langle expr \rangle * \langle expr \rangle \\
&\quad\mid (\langle expr \rangle) \\
&\quad\mid \langle id \rangle
\end{aligned}
$$

Example (3) (Cont.)

A = B + C * A

<assign> → <id> = <expr>

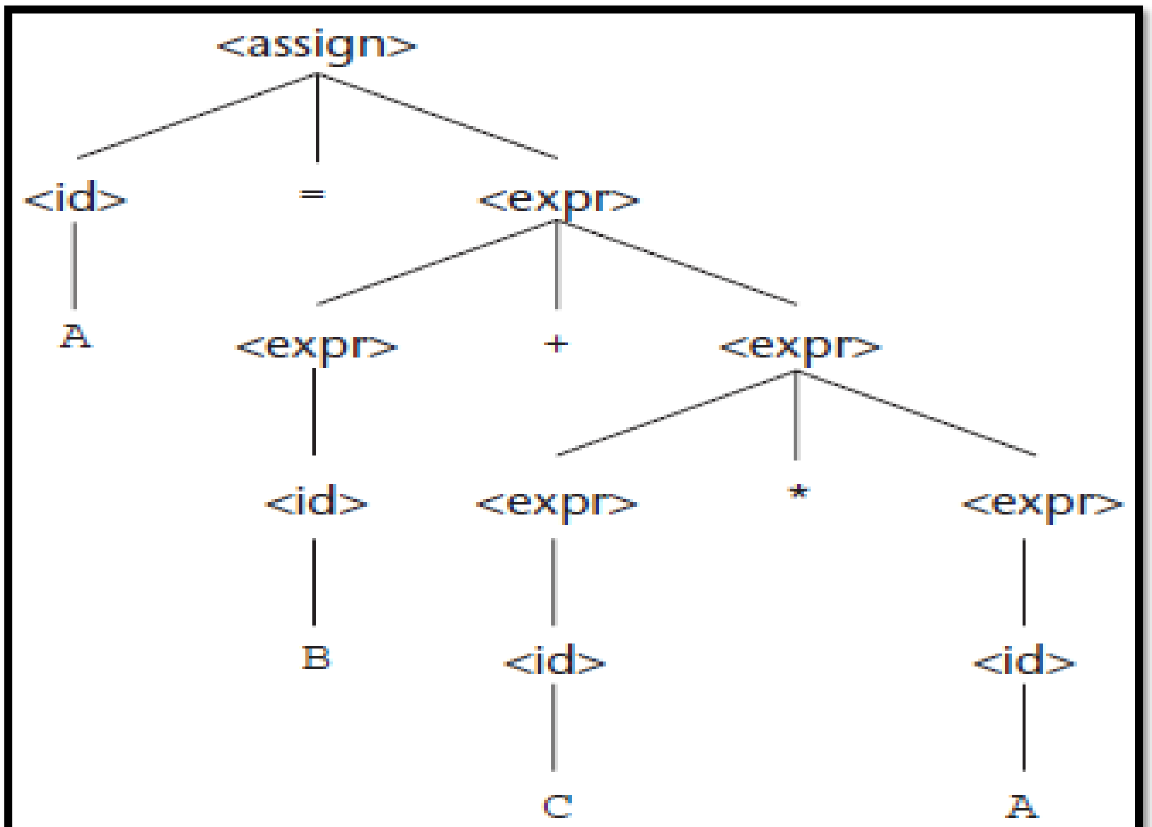
<id> → A | B | C

<expr> → <expr> + <expr>

| <expr> * <expr>

| (<expr>)

| <id>

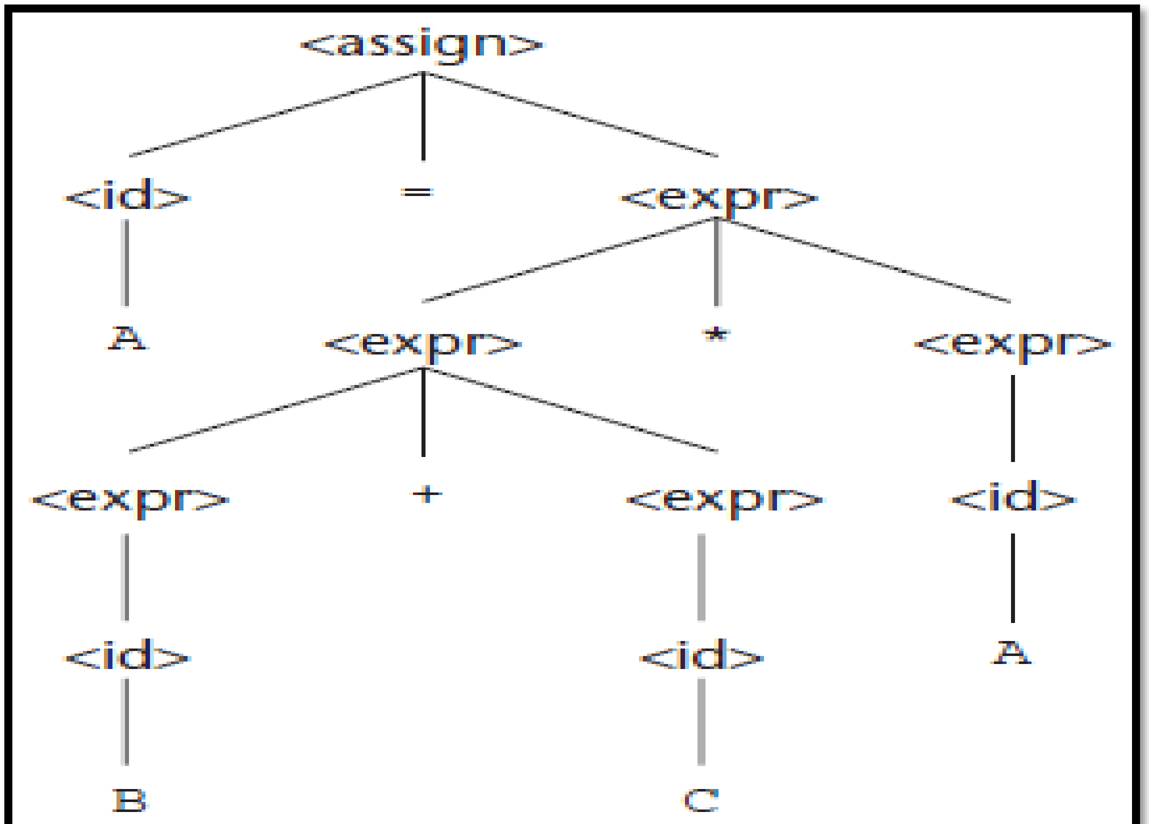


Accepted

Example (3) (Cont.)

A = B + C * A

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



Accepted

Example (4)

A = B + C * A

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

Example (4) (Cont.)

A = B + C * A

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

<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

Accepted

Example (4) (Cont.)

A = B + C * A

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

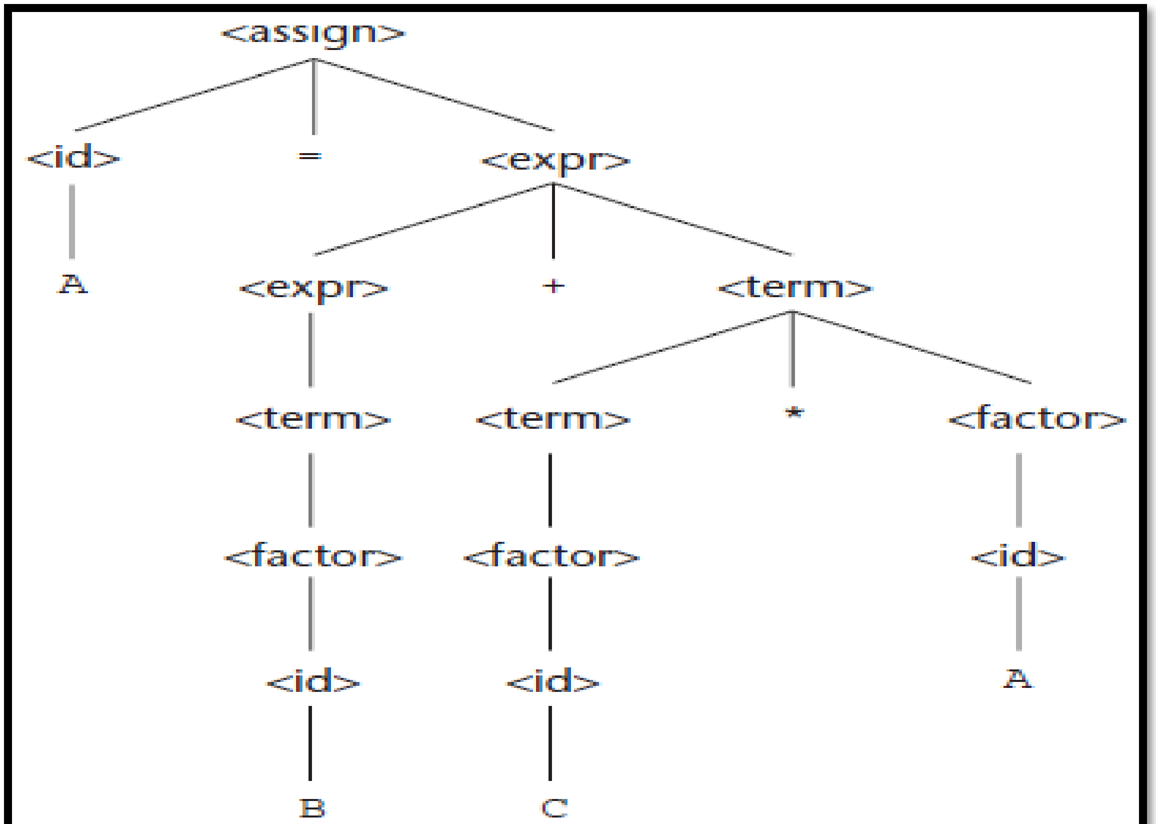
```
<assign> => <id> = <expr>
         => <id> = <expr> + <term>
         => <id> = <expr> + <term> * <factor>
         => <id> = <expr> + <term> * <id>
         => <id> = <expr> + <term> * A
         => <id> = <expr> + <factor> * A
         => <id> = <expr> + <id> * A
         => <id> = <expr> + C *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

Accepted

Example (4) (Cont.)

A = B + C * A

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



Accepted

Outline

- Parse Trees
- Ambiguity
- Unambiguous Grammar

Ambiguity

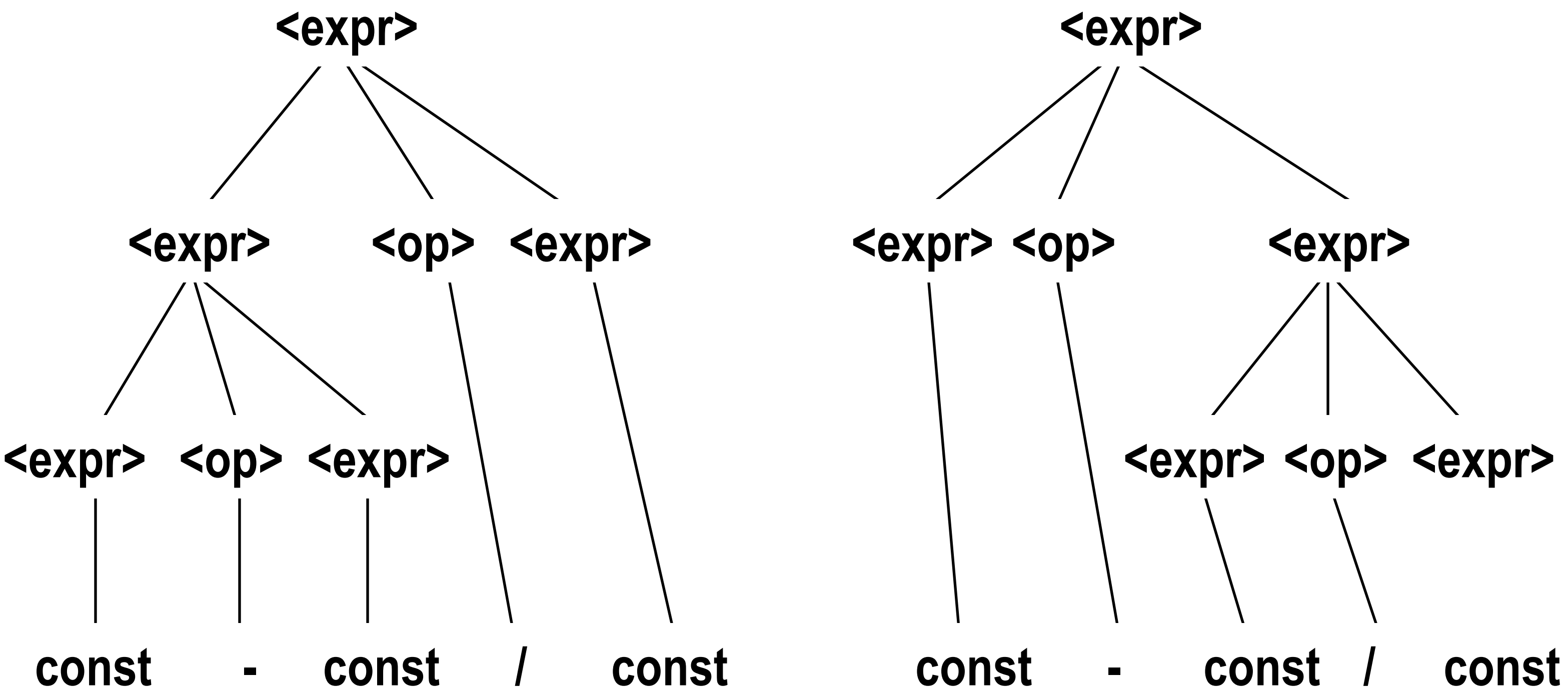
- A grammar is ambiguous if and only if it generates a sentential form that has *two* or *more* distinct parse trees.
- Example: An Ambiguous Grammar for Simple Assignment Statements.

$$
\begin{aligned}
\langle assign \rangle &\rightarrow \langle id \rangle = \langle expr \rangle \\
\langle id \rangle &\rightarrow A \mid B \mid C \\
\langle expr \rangle &\rightarrow \langle expr \rangle + \langle expr \rangle \\
&\mid \langle expr \rangle * \langle expr \rangle \\
&\mid (\langle expr \rangle) \\
&\mid \langle id \rangle
\end{aligned}
$$

Ambiguity (Cont.)

<expr> → <expr> <op> <expr> | const

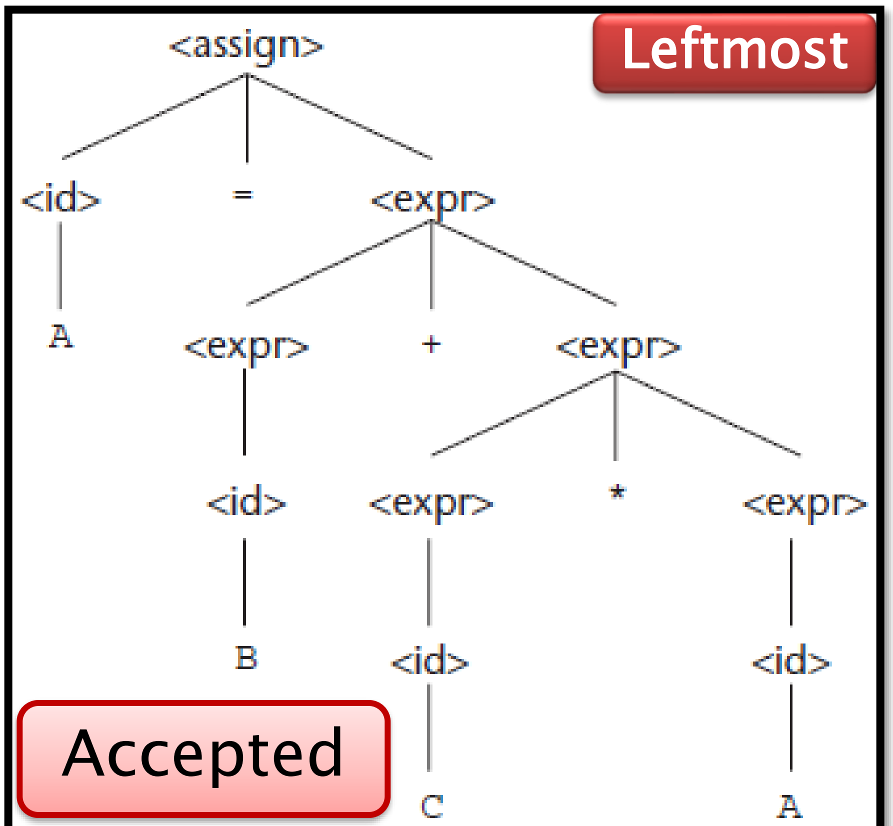
<op> → / | -

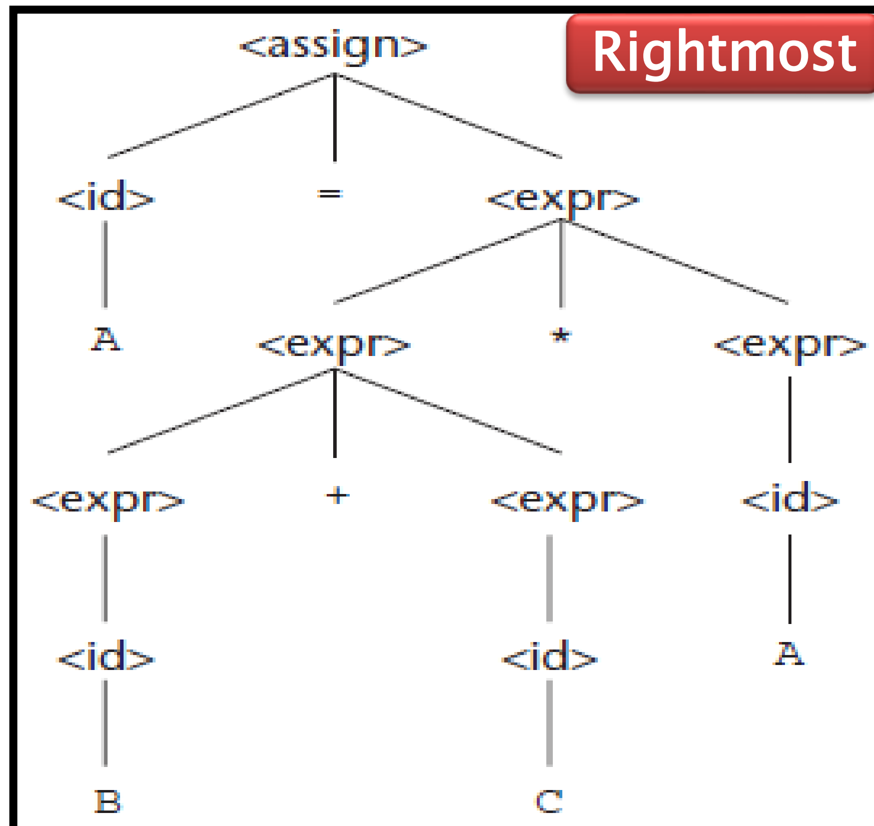


Ambiguity (Cont.)

A = B + C * A

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

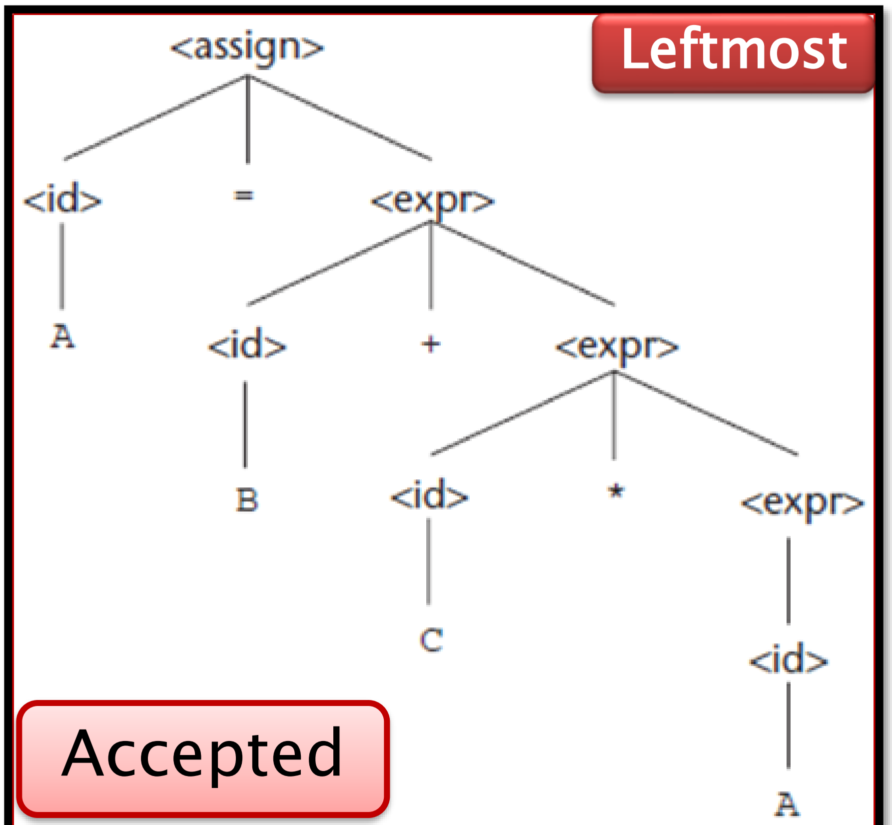


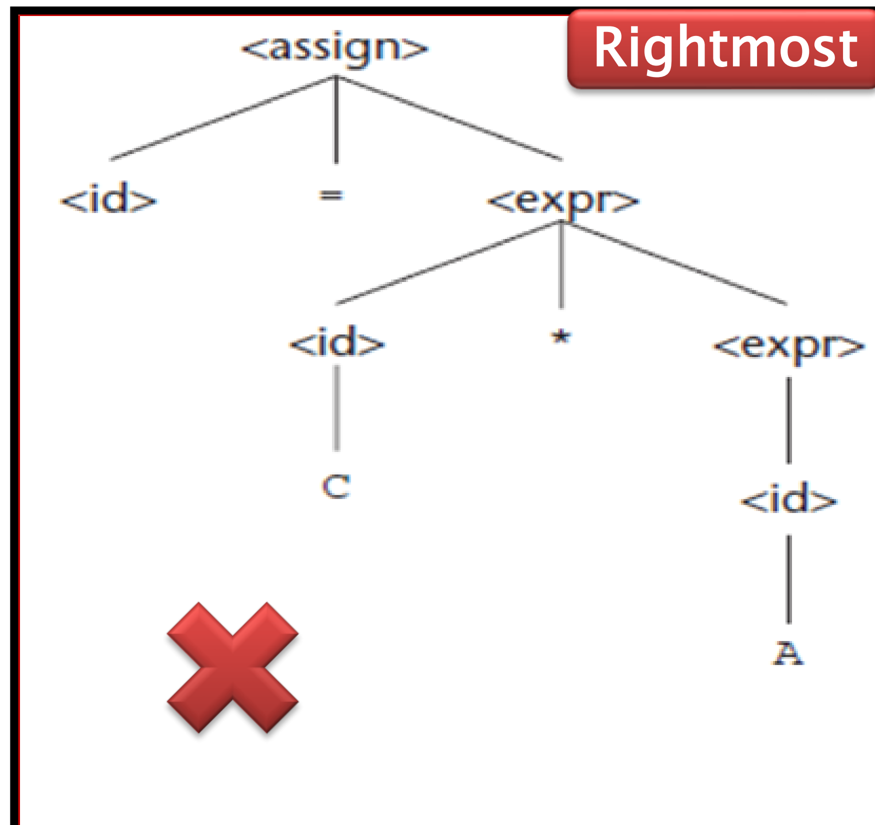


Ambiguity (Cont.)

A = B + C * A

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





Ambiguity (Cont.)

A = B + C * A

An Ambiguous Grammar

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

Leftmost

Rightmost

An Unambiguous Grammar

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

Leftmost

Ambiguity (Cont.)

- The grammar of the former Example is ambiguous because the sentence:-

  $$A = B + C * A$$

  has *two* distinct parse trees. Rather than allowing the parse tree of an expression to grow only on the right, this grammar allows growth on both the *left* and the *right*.

- Syntactic ambiguity of language structures is a problem because compilers often base the semantics of those structures on their syntactic form.

- Specifically, the compiler chooses the code to be generated for a statement by examining its parse tree.

Ambiguity (Cont.)

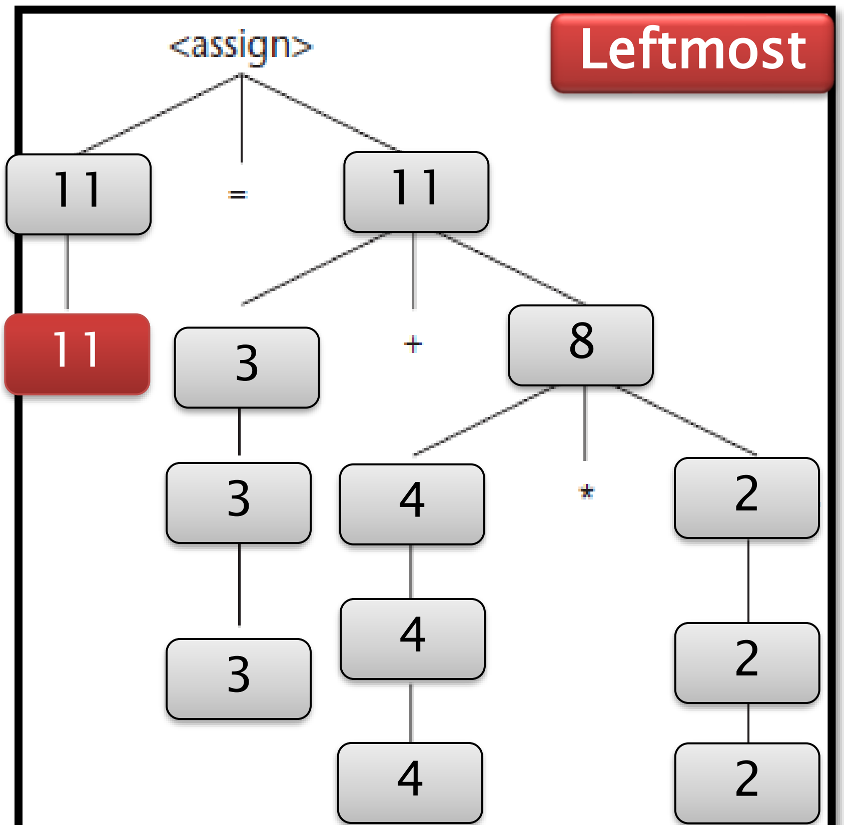
- If a language structure has *more* than one parse tree, then the meaning of the structure *cannot* be determined uniquely.
- There are several other characteristics of a grammar that are sometimes useful in determining whether a grammar is ambiguous.
- They include the following:-
  - If the grammar generates a sentence with more than one leftmost derivation, and
  - If the grammar generates a sentence with more than one rightmost derivation.

Ambiguity (Cont.)

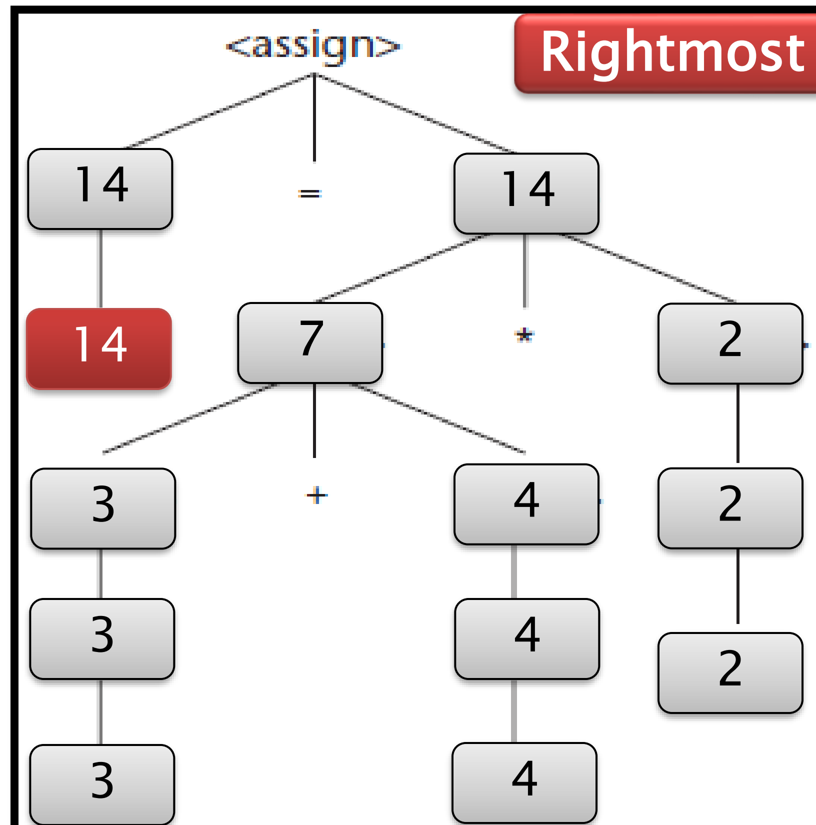
A = B + C * A

A = 2 , B = 3 , C = 4

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



≠



Outline

- Parse Trees
- Ambiguity
- Unambiguous Grammar

Unambiguous Grammar

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

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

Unambiguous Grammar (Cont.)

- Instead of using <expr> for both operands of both + and *, we could use three non-terminals to represent operands, which allows the grammar to force different operators to different levels in the parse tree.
  - 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 non-terminal, <term>, as the *right* operand of +.
  - Next, we can define <term> to generate * operators, using <term> as the left operand and a new non-terminal, <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.

Unambiguous Grammar (Cont.)

- The unambiguous grammar in *generates* the same language as the grammars of the ambiguous.

- Example, but it is unambiguous, and it specifies the usual precedence order of multiplication and addition operators.

- The following derivation of the sentence A = B + C * A uses the grammar of the unambiguous grammar:-

Unambiguous Grammar (Cont.)

A = B + C * A

Accepted

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

```
<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
```

Unambiguous Grammar (Cont.)

A = B + C * A

Accepted

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

```
<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
```

Unambiguous Grammar (Cont.)

A = B + C * A

Accepted

<assign> → <id> = <expr>

<id> → A | B | C

<expr> → <expr> + <term>

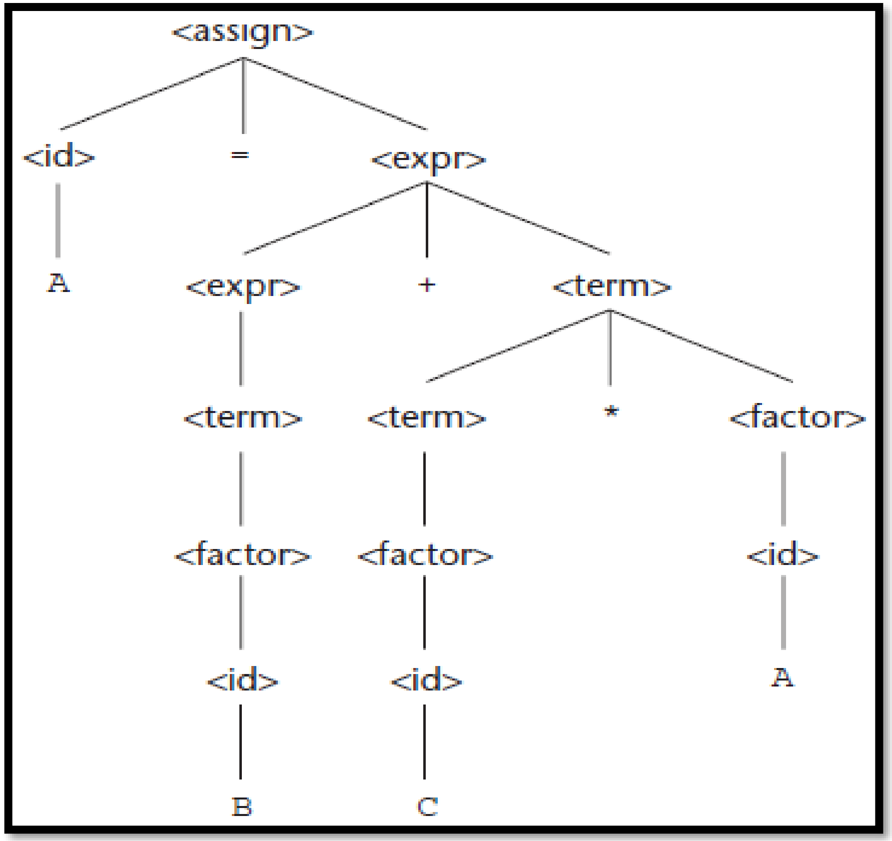
| <term>

<term> → <term> * <factor>

| <factor>

<factor> → (<expr>)

| <id>



Unambiguous Grammar (Cont.)

An Unambiguous Grammar for `if-then-else`

The BNF rules for an Ada **if-then-else** statement are as follows:

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

if <logic_expr> **then** <stmt> **else** <stmt>

Unambiguous Grammar (Cont.)

An Unambiguous Grammar for `if-then-else`

The BNF rules for an Ada **`if-then-else`** statement are as follows:

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

`if` <logic_expr> **`then`** <stmt> **`else`** <stmt>

If we also have <stmt> → <if_stmt>, this grammar is ambiguous. The simplest sentential form that illustrates this ambiguity is

`if` <logic_expr> **`then`** **`if`** <logic_expr> **`then`** <stmt> **`else`** <stmt>

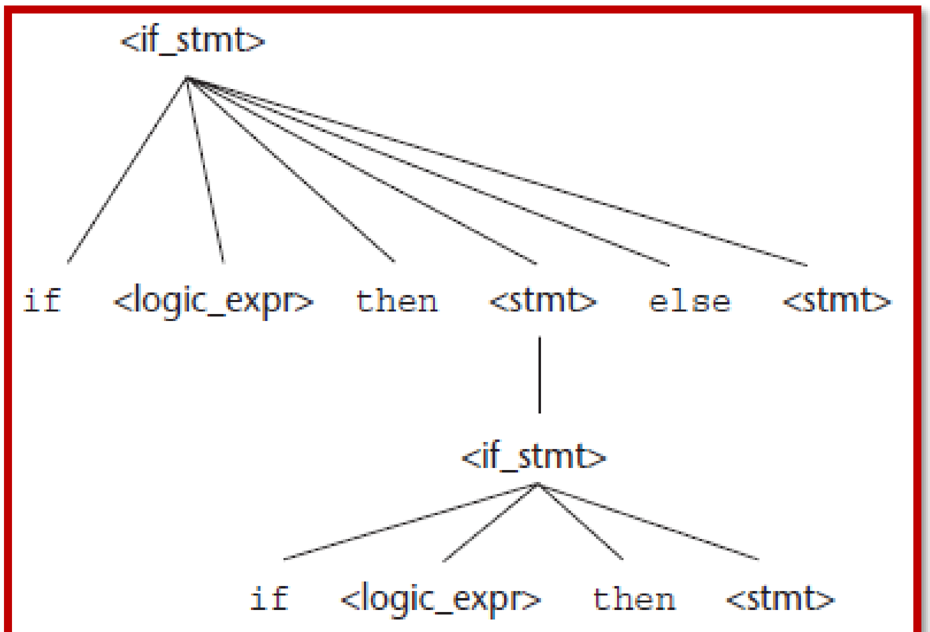
Unambiguous Grammar (Cont.)

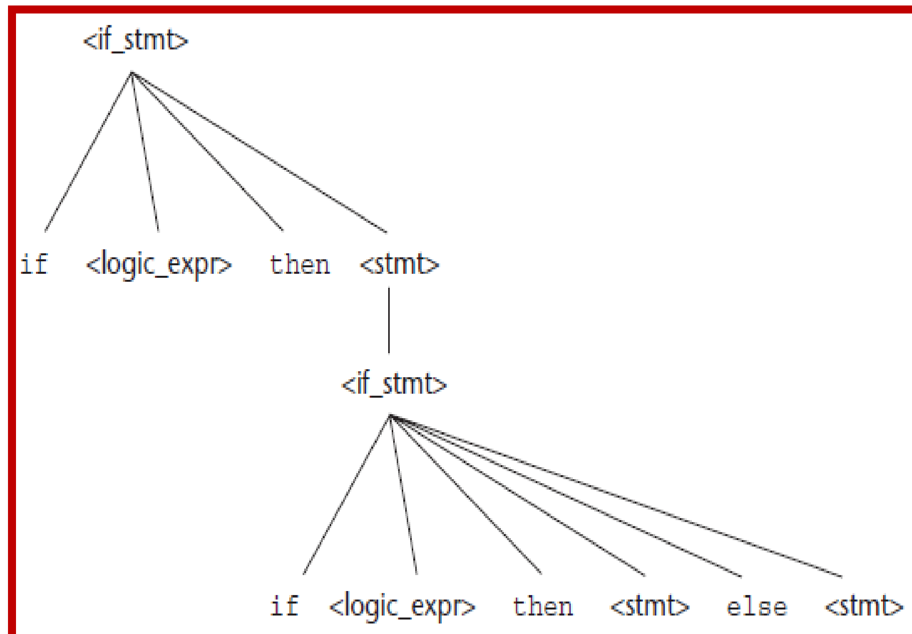
The BNF rules for an Ada **if-then-else** statement are as follows:

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

if <logic_expr> **then** <stmt> **else** <stmt>

if <logic_expr> **then** **if** <logic_expr> **then** <stmt> **else** <stmt>





Unambiguous Grammar (Cont.)

- We will now develop an unambiguous grammar that describes this if statement. The rule for if constructs in many languages is that an else clause, when present, is matched with the nearest previous unmatched then clause.

- Therefore, there cannot be an if statement without an else between a then clause and its matching else.

- So, for this situation, statements must be *distinguished* between those that are matched and those that are unmatched, where unmatched statements are else-less ifs and all other statements are matched.

Unambiguous Grammar (Cont.)

- To reflect the different categories of statements, different abstractions, or non-terminals, must be used. The unambiguous grammar based on these ideas follows:-

<stmt> → <matched> | <unmatched>

<matched> → **if** (<logic_expr>) <matched> **else** <matched>

|any non-if statement

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

|**if** (<logic_expr>) <matched> **else** <unmatched>

There is just one possible parse tree, using this grammar, for the following sentential form:

- **if** (<logic_expr>) **if** (<logic_expr>) <stmt> **else** <stmt>

ANY QUESTIONS

Thank You!