Syntax and Semantics

Lecture 05

Instructor: C. Pu (Ph.D., Assistant Professor)

puc@marshall.edu

Parse Tree

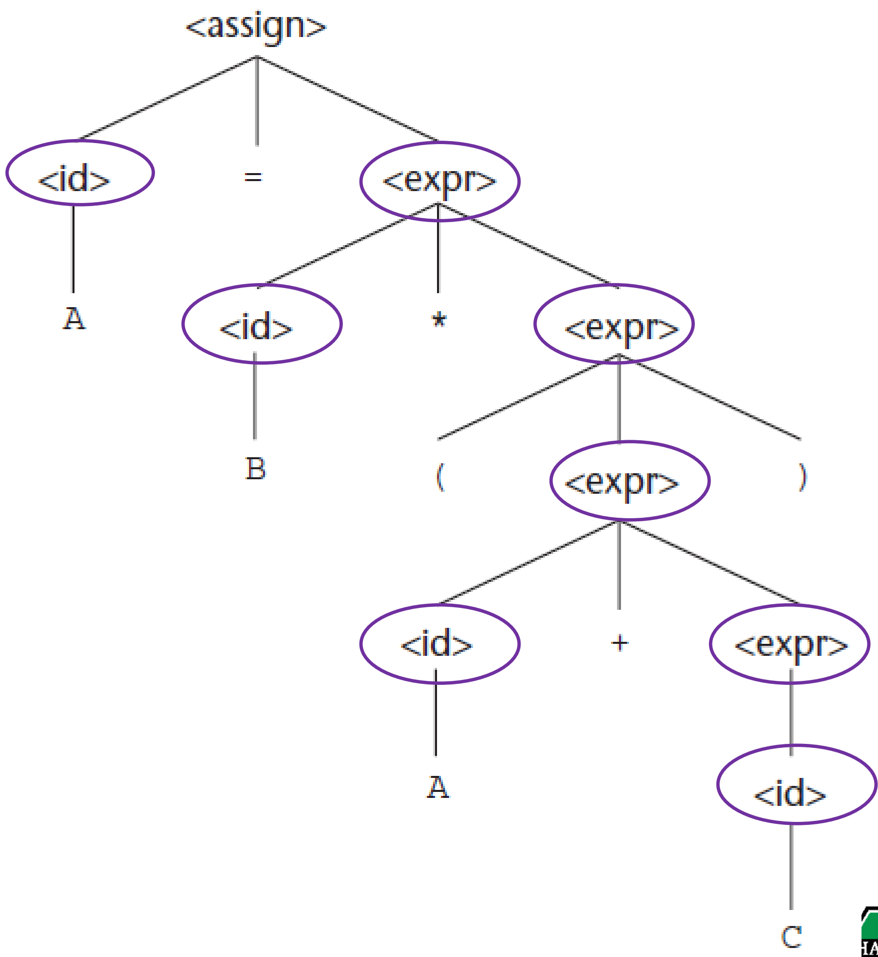
- One of the most attractive features of grammars is that they naturally describe the *hierarchical syntactic structure* of the sentences of the languages they define.
- These hierarchical structures are called ***parse trees***.
- For example, `A = B * (A + C)`

```
<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 )
```

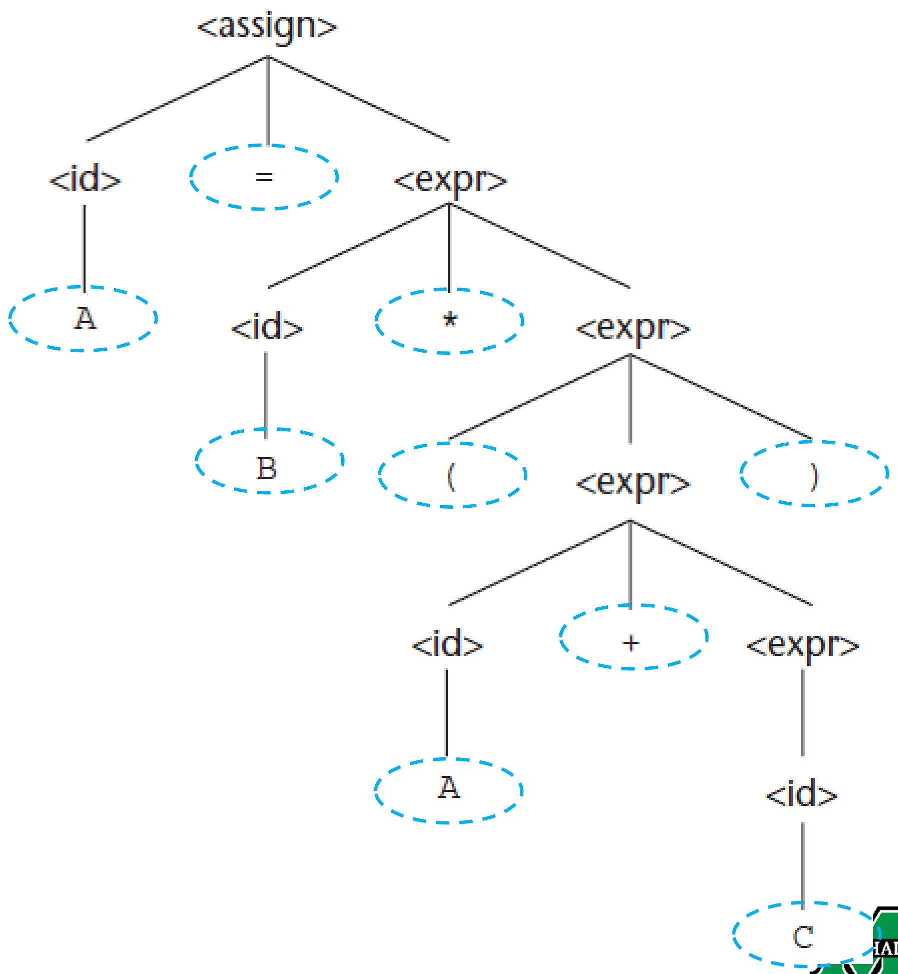
Parse Tree (cont.)

- The ***parse tree*** shows the structure of the assignment statement derived.
  - Every ***internal node*** of a parse tree is labeled with a ***nonterminal*** symbol



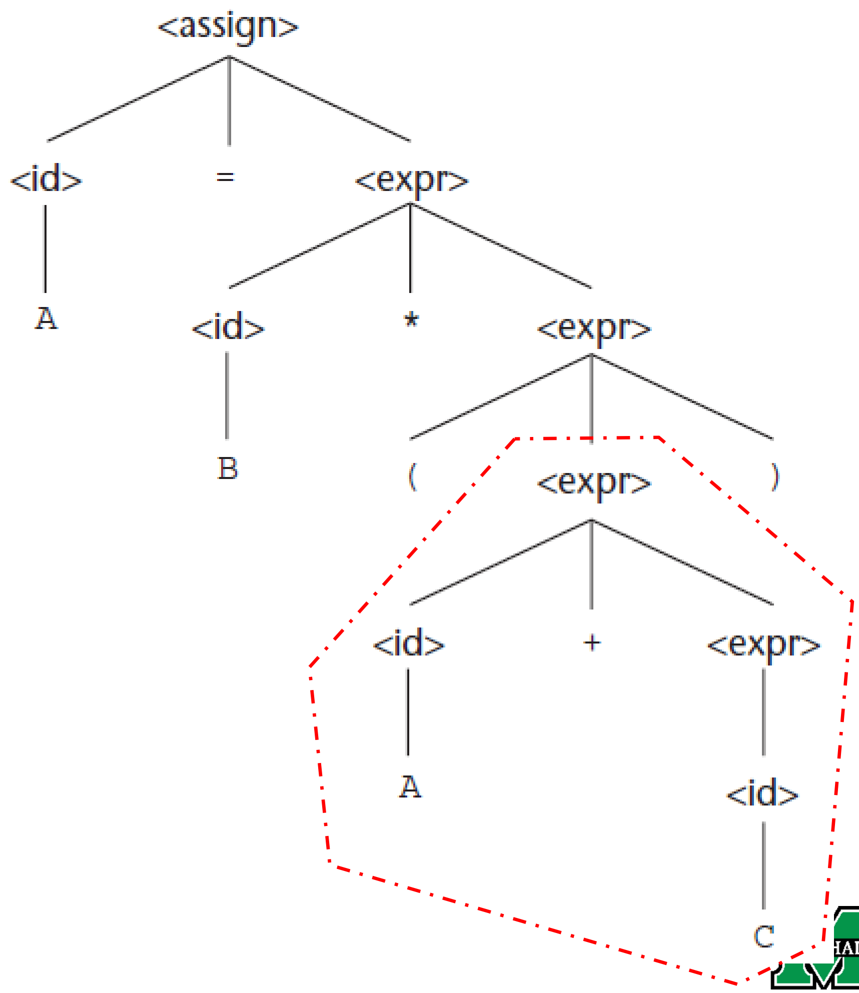
Parse Tree (cont.)

- The ***parse tree*** shows the structure of the assignment statement derived.
  - Every ***leaf*** is labeled with a ***terminal*** symbol



Parse Tree (cont.)

- The ***parse tree*** shows the structure of the assignment statement derived.
  - Every ***subtree*** of a parse tree describes one ***instance*** of an abstraction in the sentence



Ambiguity

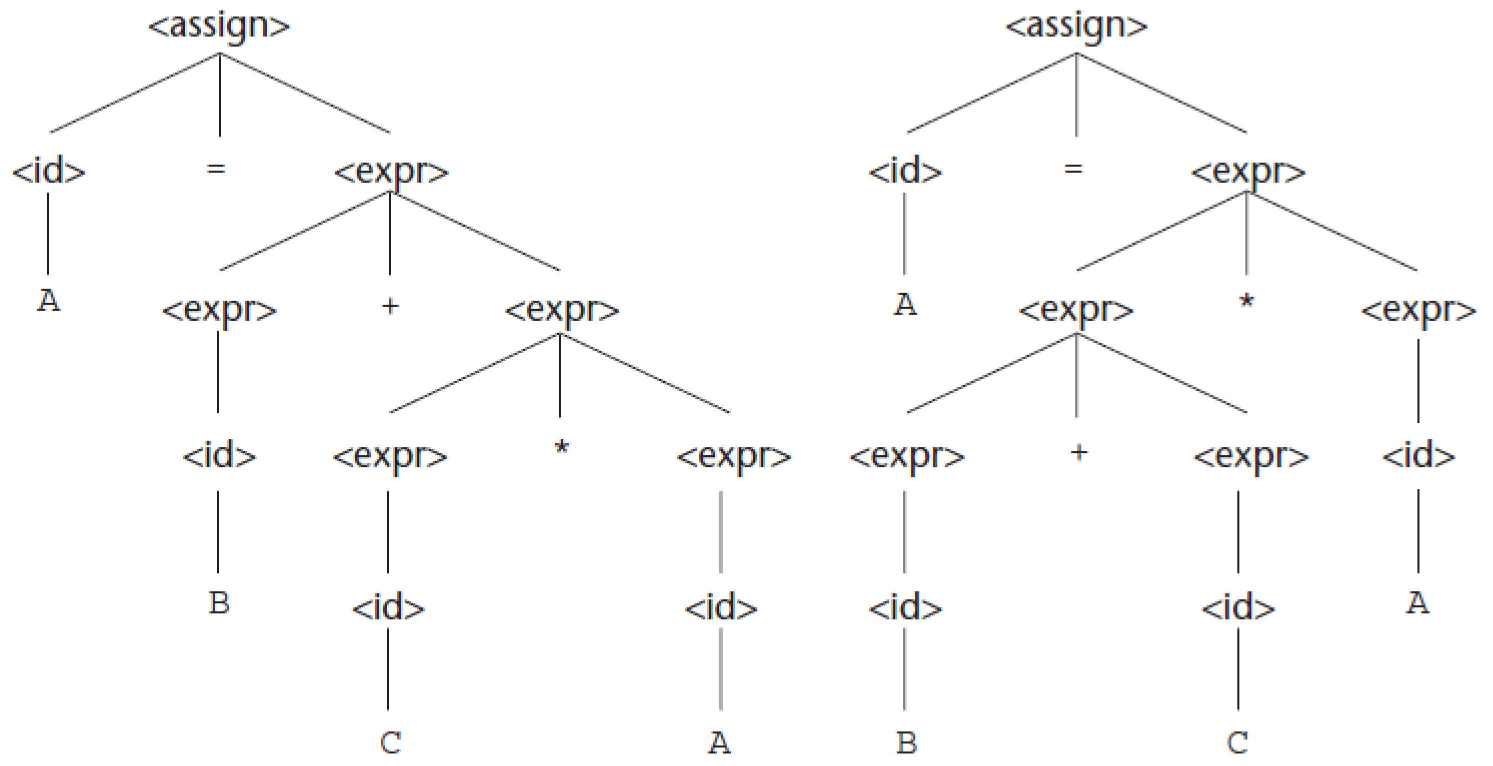
- A grammar that generates a *sentential form* for which there are *two or more* ***distinct*** *parse trees* is said to be ***ambiguous***.
- Consider the grammar for simple assignment statements

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

- The grammar is ***ambiguous*** because the sentence `A = B + C * A` has ***two distinct*** parse trees

Ambiguity

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



Ambiguity

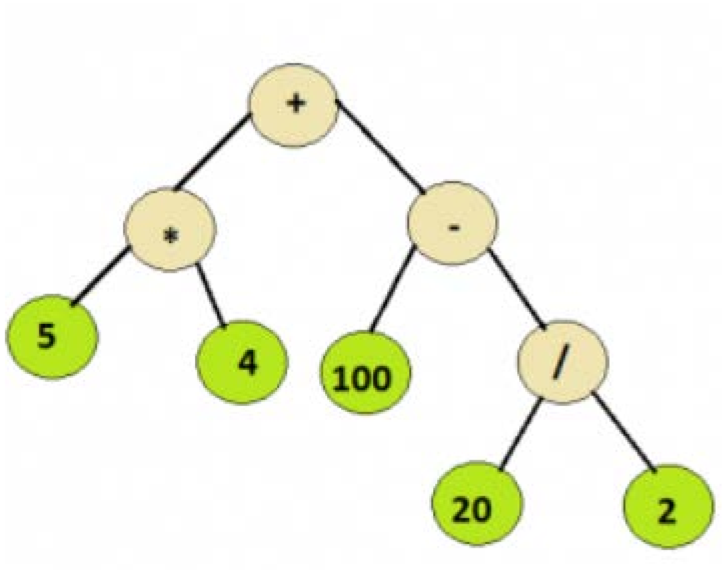
- ***Syntactic ambiguity*** of language structures is a ***problem*** because compilers often base the semantics of those structures on their syntactic form.
- Specifically, the compiler chooses the code to be generated for a statement by examining its parse tree.
- If a language structure has more than 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
  - (1) if the grammar generates a sentence with more than one leftmost derivation
  - (2) if the grammar generates a sentence with more than one rightmost derivation

Ambiguity

- Some parsing algorithms can be based on ambiguous grammars.
- When such a parser encounters an ambiguous construct, it uses nongrammatical information provided by the designer to construct the correct parse tree.
- In many cases, an ambiguous grammar can be rewritten to be unambiguous but still generate the desired language.

Evaluation of Parse Tree

- Example (solve.py)



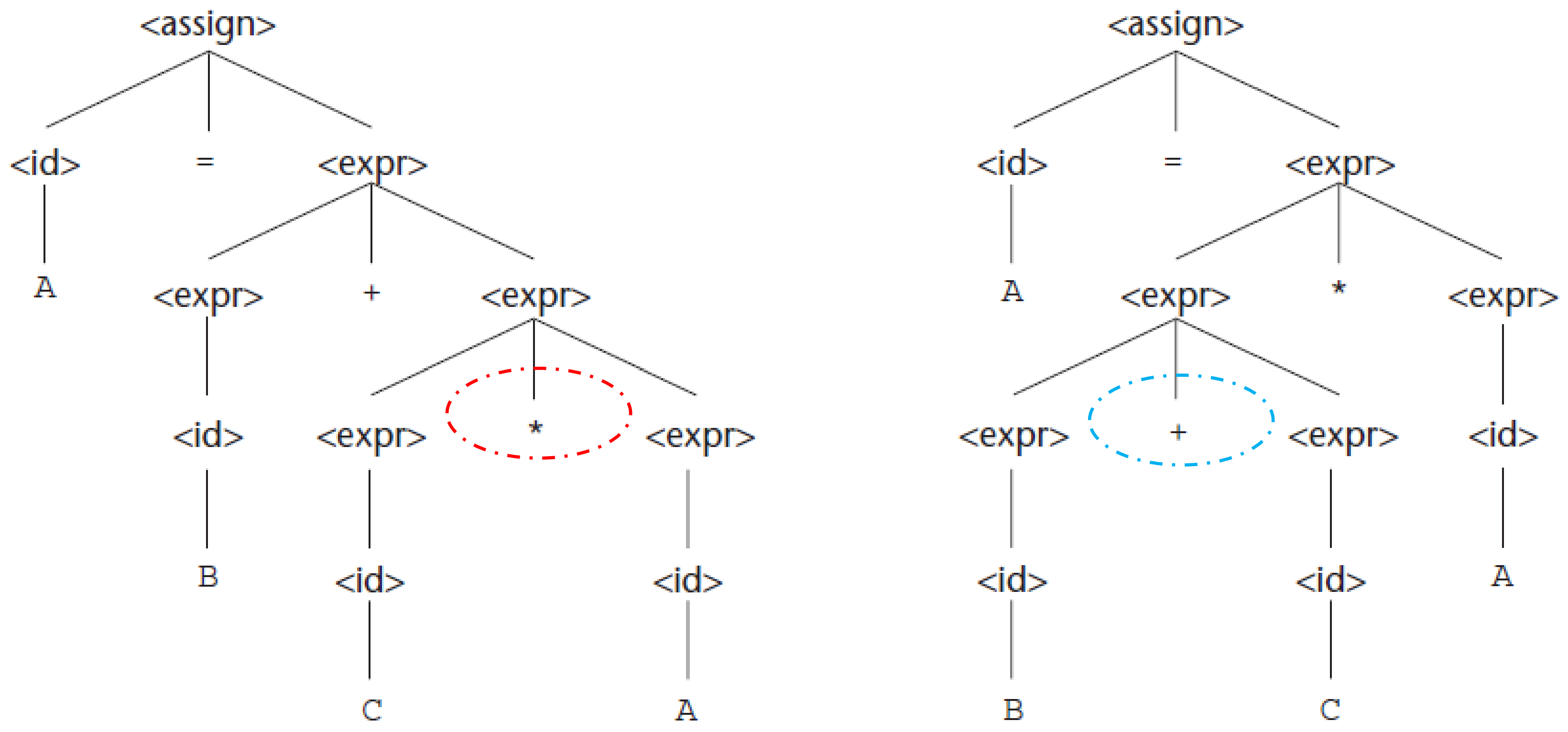
```
solve(tree t):
  let t be the parse tree
  if t is not null then
      if t.info is operand then
         return t.info
      else
         A = solve(t.left)
         B = solve(t.right)
         return A operator B
         where operator is the info contained in t
```

Operator Precedence

- When an expression includes two different operators, for example, x + y * z, one obvious semantic issue is *the order of evaluation of the two operators.*
  - This semantic question can be answered by assigning different ***precedence levels*** to operators.
- For example, if * has been assigned ***higher precedence*** than +
  - multiplication will be ***done first***, regardless of the order of appearance of the two operators in the expression.
- A grammar can describe a certain syntactic structure so that part of the meaning of the structure can be determined from its parse tree.
  - *An operator in an arithmetic expression is generated lower in the parse tree can be used to indicate that it has precedence over an operator produced higher up in the tree.*

Operator Precedence

- Two different parse trees for `A = B + C * A`



Operator Precedence

- The grammar is not ambiguous, the precedence order of its operators is not the usual one.
  - Regardless of the particular operators involved, a parse tree of sentence has the rightmost operator in the expression at the lowest point in the parse tree, with the other operators in the tree moving progressively higher as one moves to the left in the expression.
    - The expression A + B * C, * is the lowest in the tree, indicating it is to be done first.
    - The expression A * B + C, + is the lowest, indicating it is to be done first.

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

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

Operator Precedence

- A grammar can be written for the simple expression regardless of the order in which the operators appear in an expression.
- The correct ordering is specified by *using separate nonterminal symbols to represent the operands of the operators that have different precedence.*
  - Instead of using <expr> for both operands of both + and *, we could use three nonterminals 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 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.

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

Operator Precedence

- An unambiguous grammar for expressions

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

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

Operator Precedence

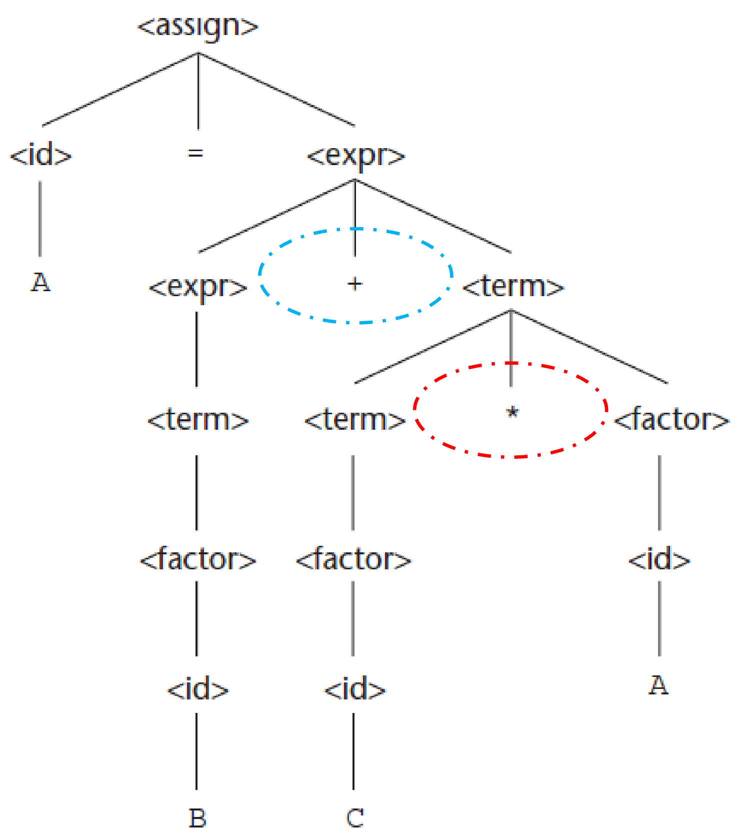
- The following derivation of the sentence A = B + C * A

```
<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
```

```
<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
```

Operator Precedence

- The unique parse tree of the sentence A = B + C * A



An Unambiguous Grammar for if-then-else

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

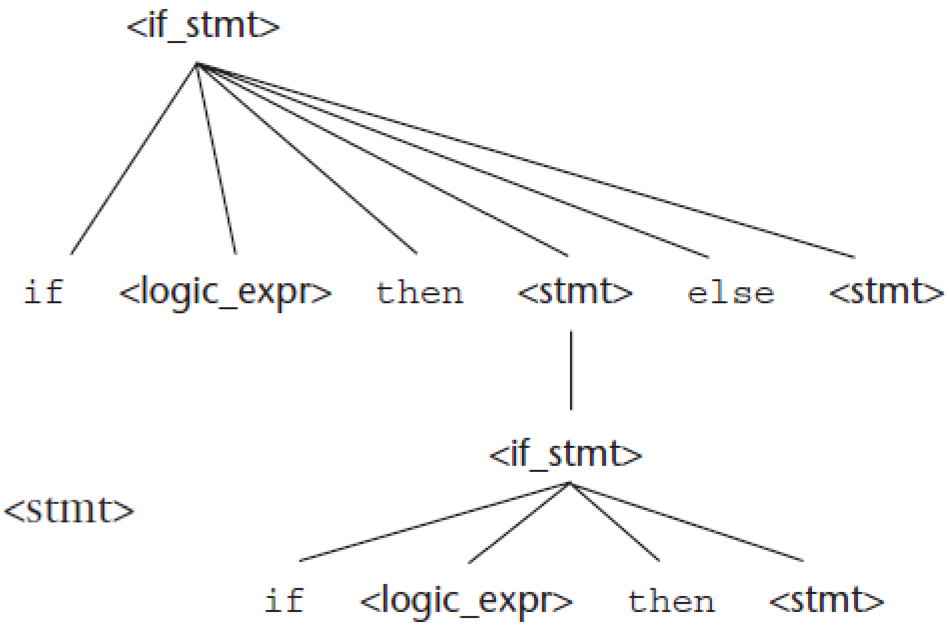
```
<if_stmt> → if <logic_expr> then <stmt>
            if <logic_expr> then <stmt> else <stmt>
```

- If we also have <stmt> → <if_stmt>, this grammar is ambiguous.

An Unambiguous Grammar for if-then-else

- The simplest sentential form that illustrates this ambiguity is

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



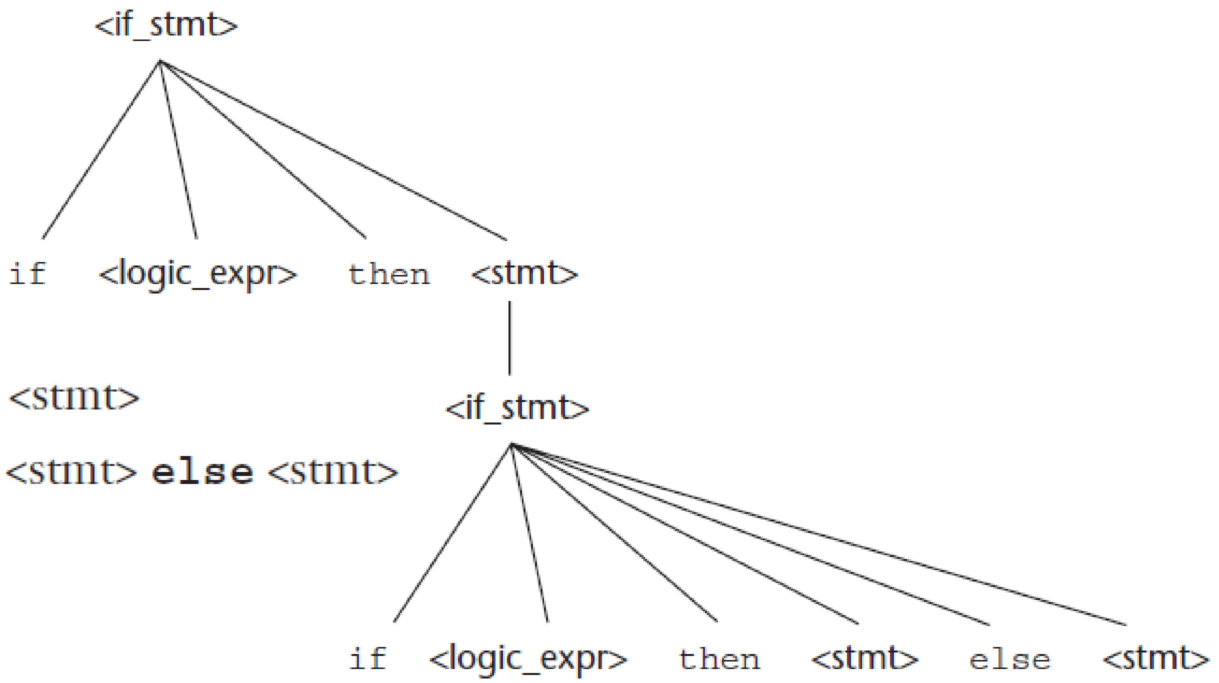
<if_stmt> → **if** <logic_expr> **then** <stmt>
if <logic_expr> **then** <stmt> **else** <stmt>

<stmt> → <if_stmt>

An Unambiguous Grammar for if-then-else

- The simplest sentential form that illustrates this ambiguity is

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



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

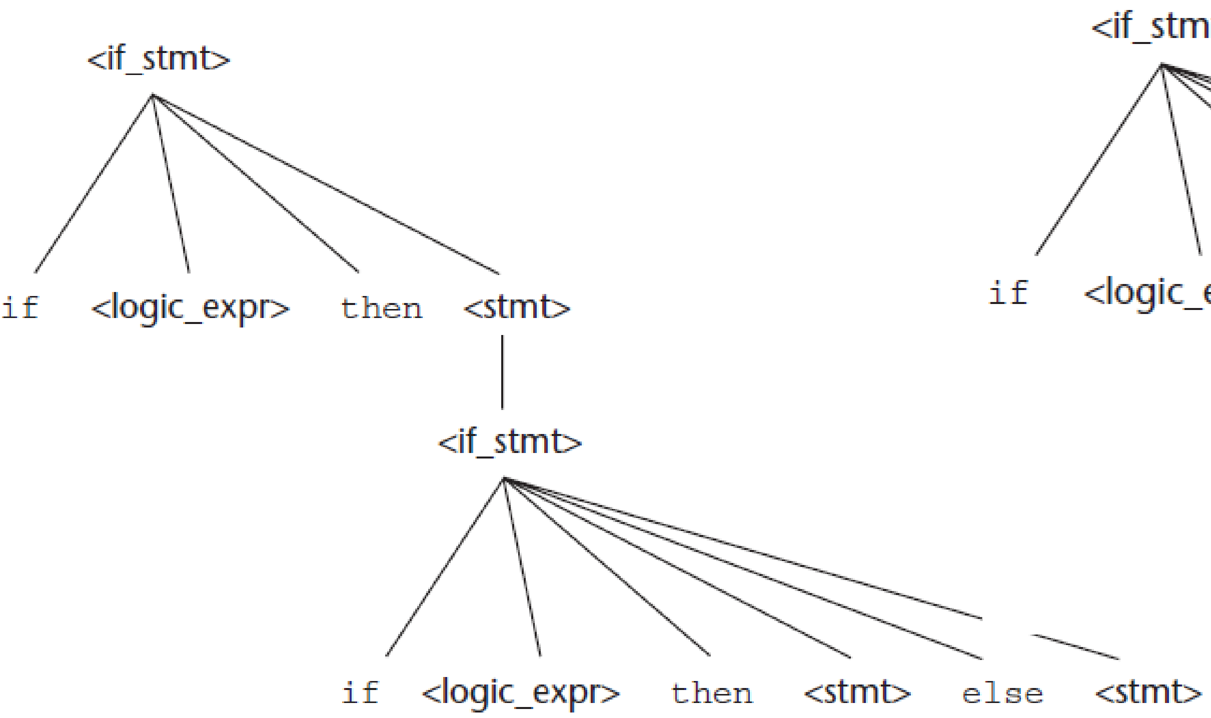
if <logic_expr> **then** <stmt> **else** <stmt>

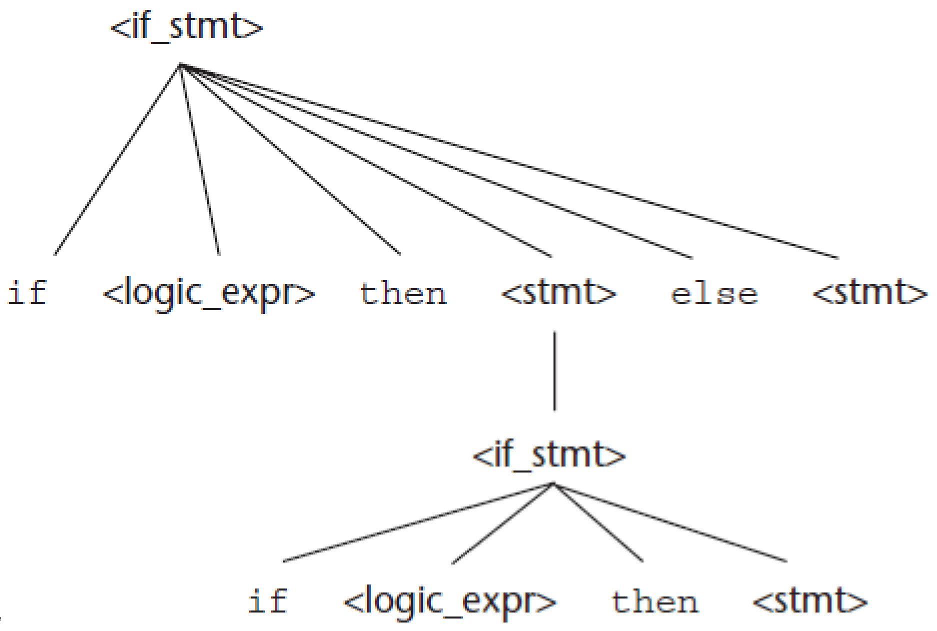
<stmt> → <if_stmt>

An Unambiguous Grammar for if-then-else

- Consider the following example of this construct:

```
if done == true
  then if denom == 0
    then quotient = 0;
    else quotient = num / denom;
```



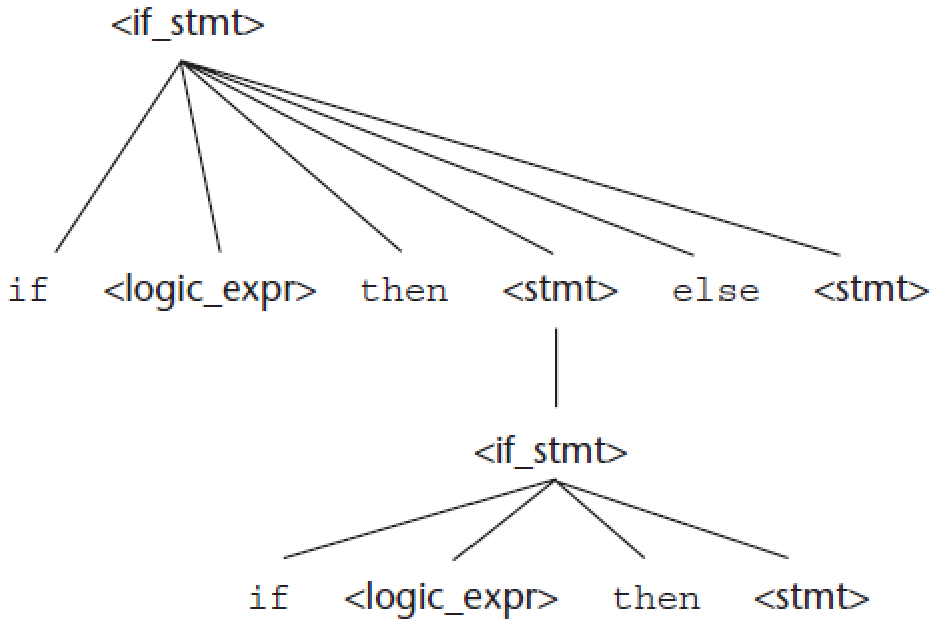


An Unambiguous Grammar for if-then-else

- Consider the following example of this construct:

```
if done == true
  then if denom == 0
    then quotient = 0;
    else quotient = num / denom;
```

- The problem is that if the following tree is used as the basis for translation, the else clause would be executed when *done* is not true



An Unambiguous Grammar for if-then-else

- The rule for ***if*** constructs in many languages is that an ***else*** clause, when present, is matched with the nearest previous unmatched ***if***.

- Therefore, there cannot be an ***if*** statement without an ***else*** between a ***if*** 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.

An Unambiguous Grammar for if-then-else

- The unambiguous grammar based on these ideas follows:

```
<stmt> → <matched> | <unmatched>
<matched> → if <logic_expr> then <matched> else <matched>
             |any non-if statement
<unmatched> → if <logic_expr> then <stmt>
               |if <logic_expr> then <matched> else <unmatched>
```

An Unambiguous Grammar for if-then-else

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

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

```
<stmt> → <matched> | <unmatched>
<matched> → if <logic_expr> then <matched> else <matched>
              |any non-if statement
<unmatched> → if <logic_expr> then <stmt>
               |if <logic_expr> then <matched> else <unmatched>
```