Describing Syntax and Semantics

of

Programming Languages

Part II

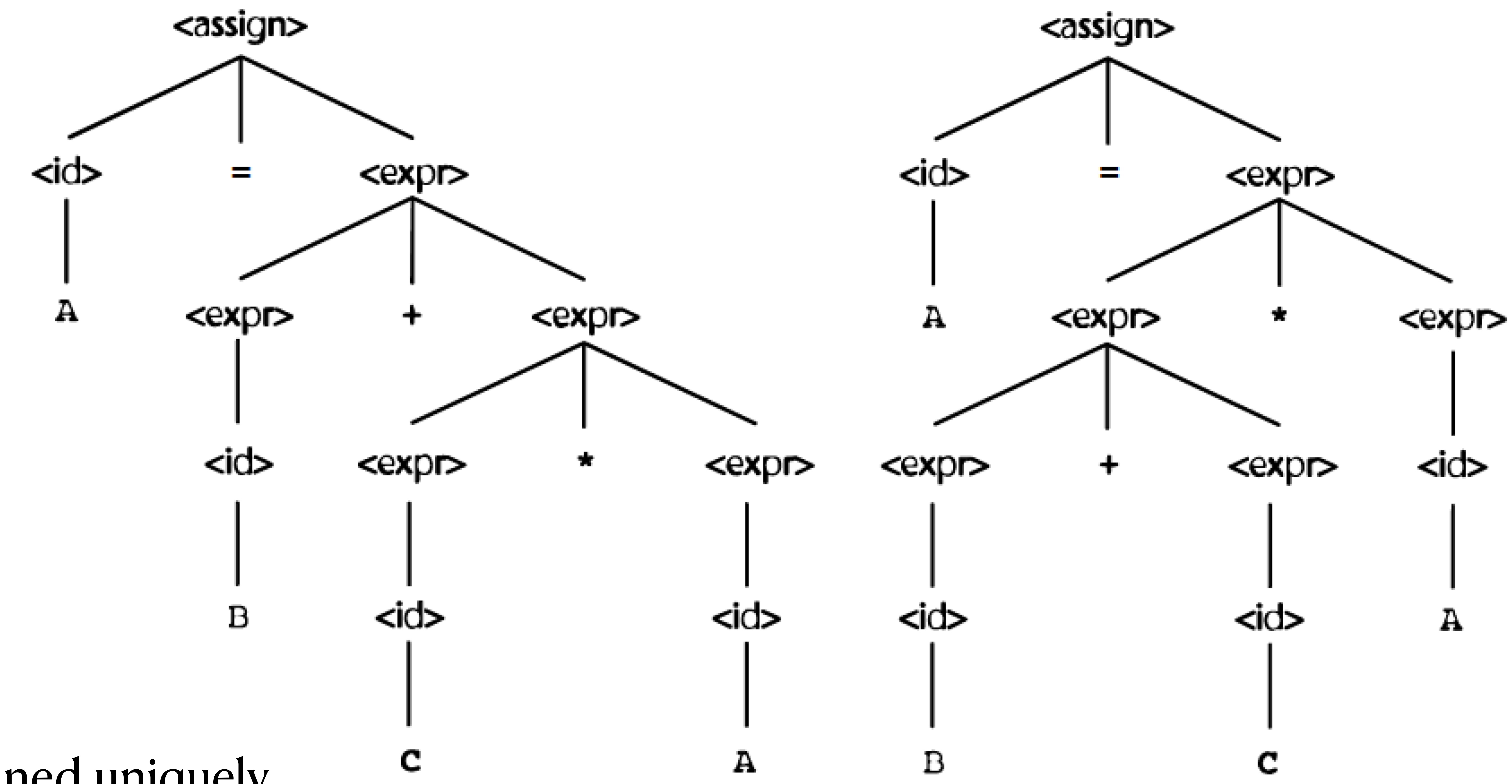
Ambiguity

A grammar that generates a sentence for which there are <u>two or more distinct parse trees</u> is said to be **ambiguous**.

Example: ambiguous grammar for simple assignment statements

Consider the string: A = B + C * A

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



ambiguous grammars are problematic

meaning of sentences cannot be determined uniquely

Operator Precedence

Ambiguity in an expression grammar can often be resolved by rewriting the grammar rules to reflect **operator precedence**. This rewrite will involve additional non-terminals and rules.

Modified Grammar

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

Leftmost Derivation for `A = B + C * A`

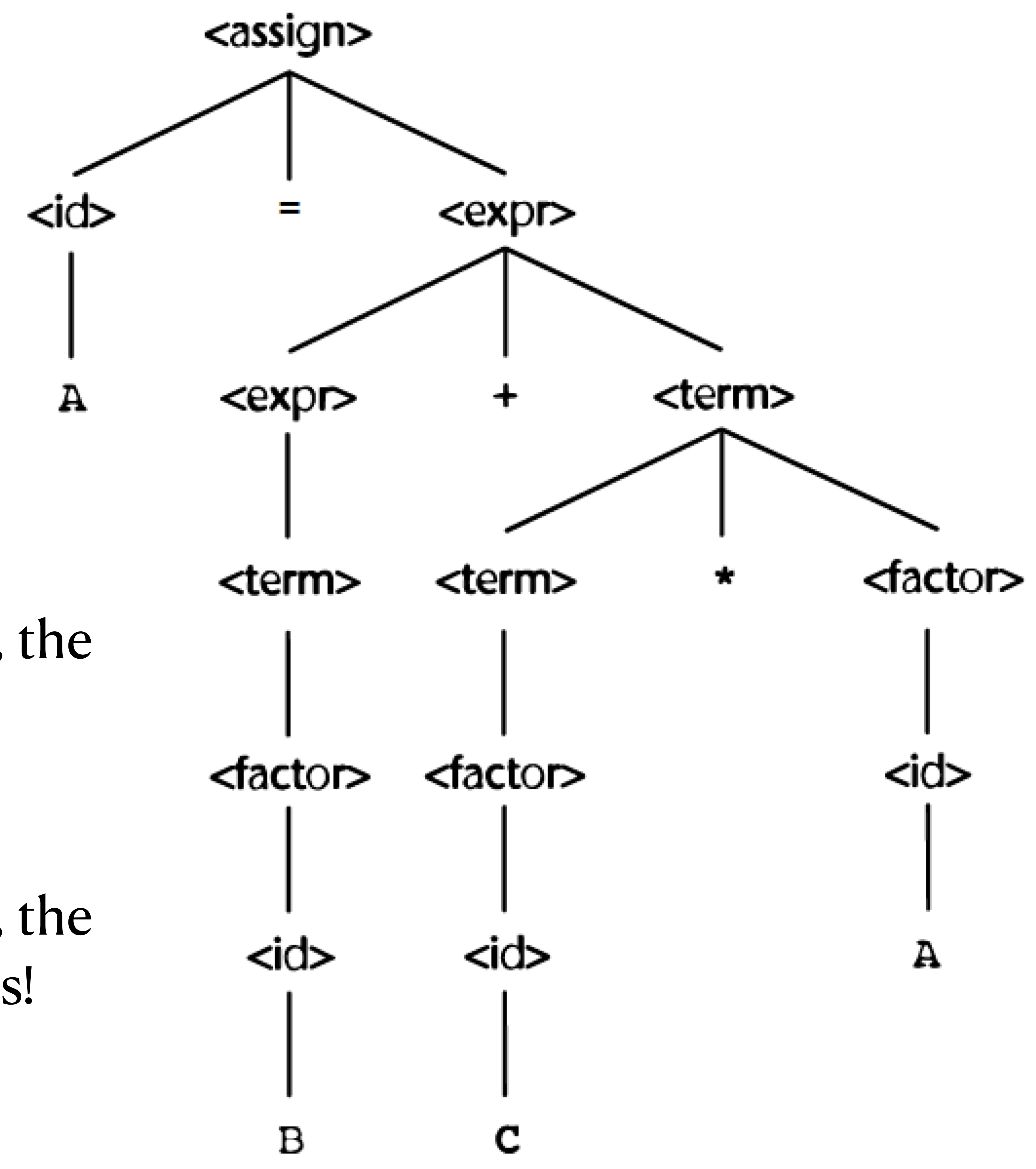
```
<assign>
⇒ <id> = <expr>
⇒ A = <expr>
⇒ A = <expr> + <term>
⇒ A = <term> + <term>
⇒ A = <factor> + <term>
⇒ A = <id> + <term>
⇒ A = B + <term>
⇒ A = B + <term> * <factor>
⇒ A = B + <factor> * <factor>
⇒ A = B + <id> * <factor>
⇒ A = B + C * <factor>
⇒ A = B + C * <id>
⇒ A = B + C * A
```

Unique Parse Tree

Modified Grammar

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

`Parse tree for A = B + C * A`



Higher the precedence of operator, the **lower** in the parse tree!

OR

Higher the precedence of operator, the **later** in the grammar rules it appears!

Operator Precedence continued

The connection between parse trees and derivations is very close; either can easily be constructed from the other.

Every derivation with an unambiguous grammar has a unique parse tree, although that tree can be represented by different derivations.

For example, the derivation of the sentence A = B + C * A (shown to the right) is different from the derivation of the same sentence given previously. But, since the grammar we are using is unambiguous, the parse tree (shown in previous slide) is the same for both derivations.

Rightmost Derivation for 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

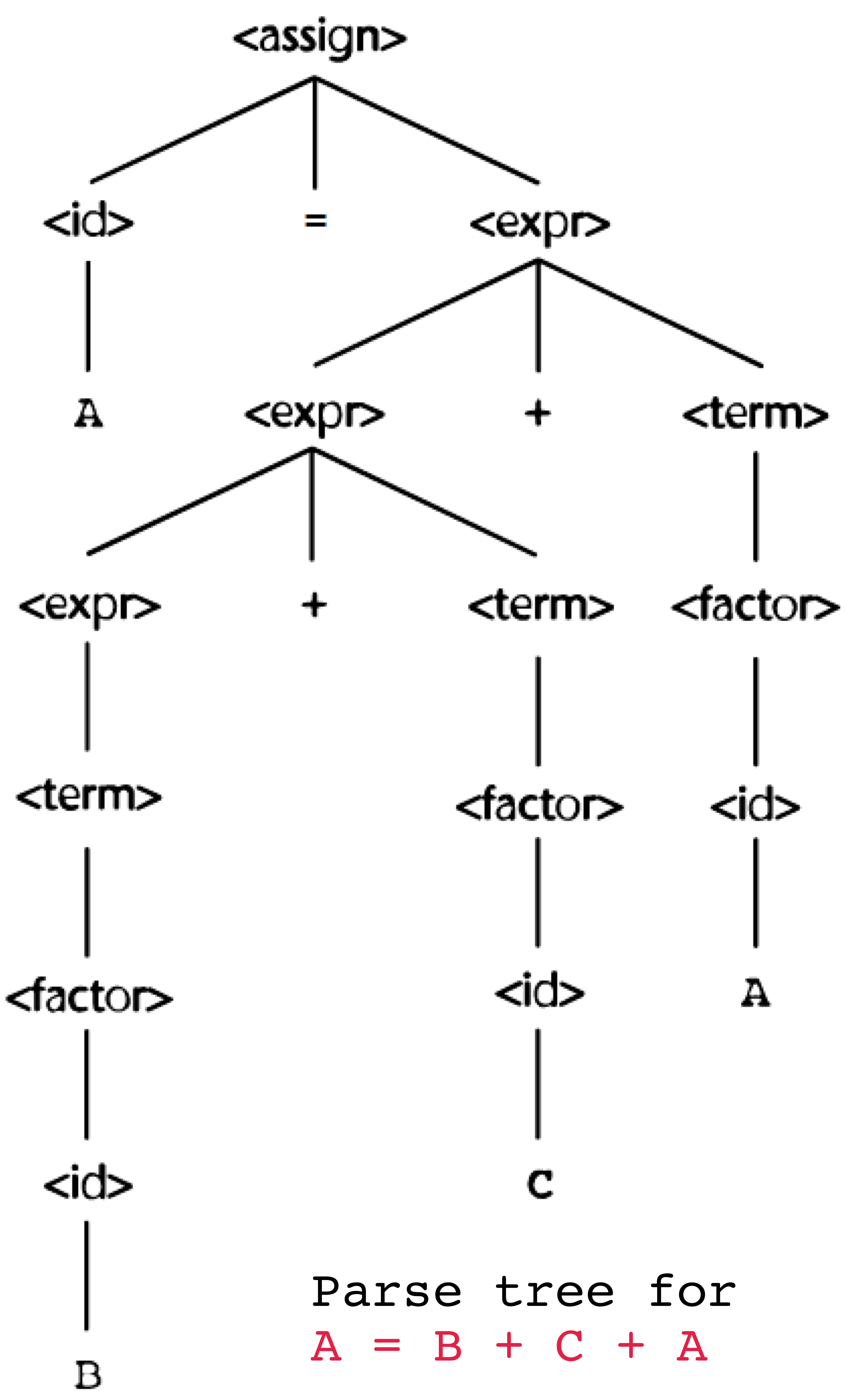
A grammar that describes expressions must handle associativity properly.

The parse tree to the right shows the left addition lower than right addition, indicating left-associativity.

The left-associativity is because of the left-recursion in the first rule for `<expr>`:

```
<expr> : <expr> + <term>
<expr> : <term>
```

To express right-associativity, we can use right-recursive rules.



Parse tree for
A = B + C + A

Right-Associativity (exponent operator)

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

Left-recursive; left-associative

Left-recursive; left-associative

Right-recursive; right-associative

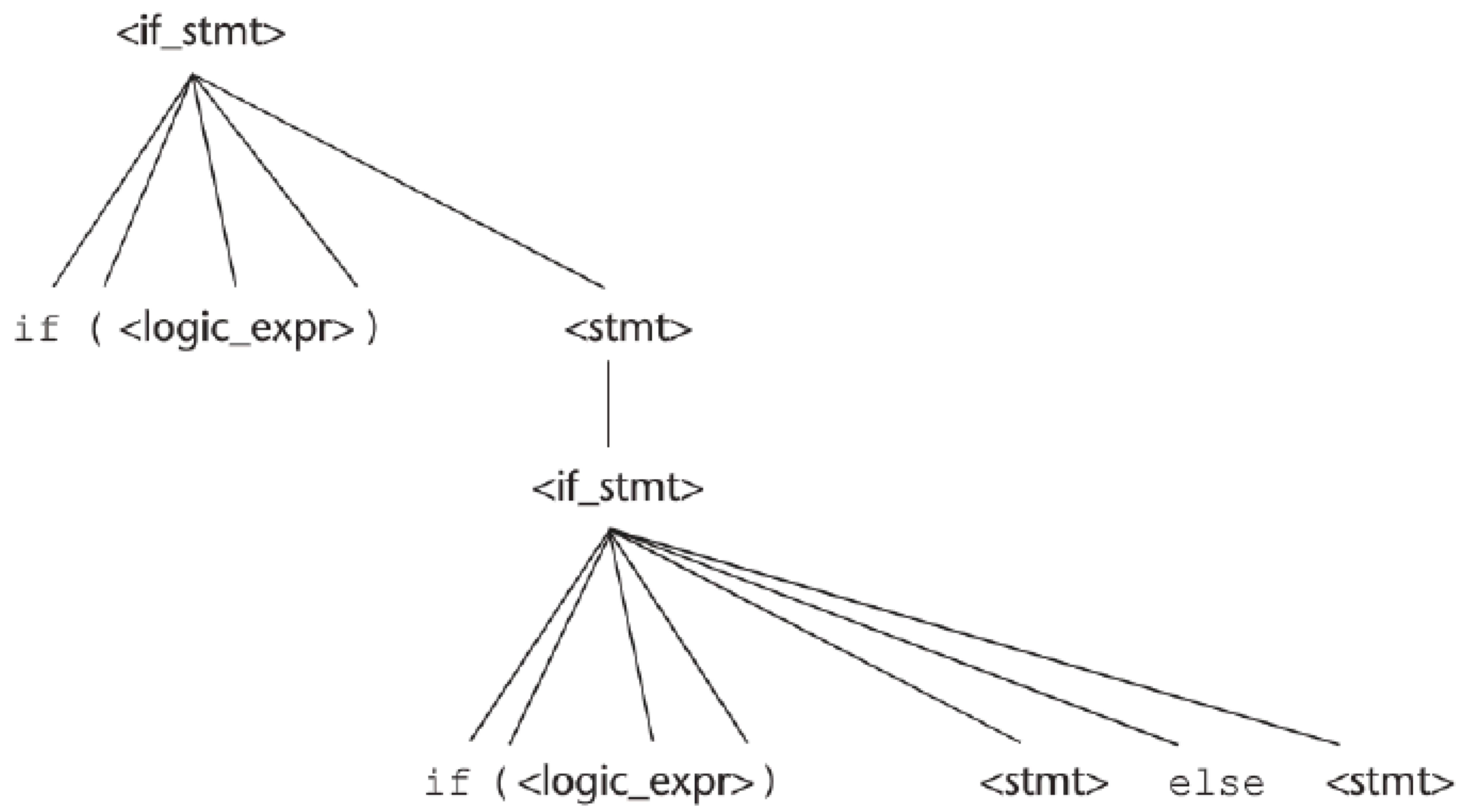
precedence(**) > precedence(*) > precedence(+)

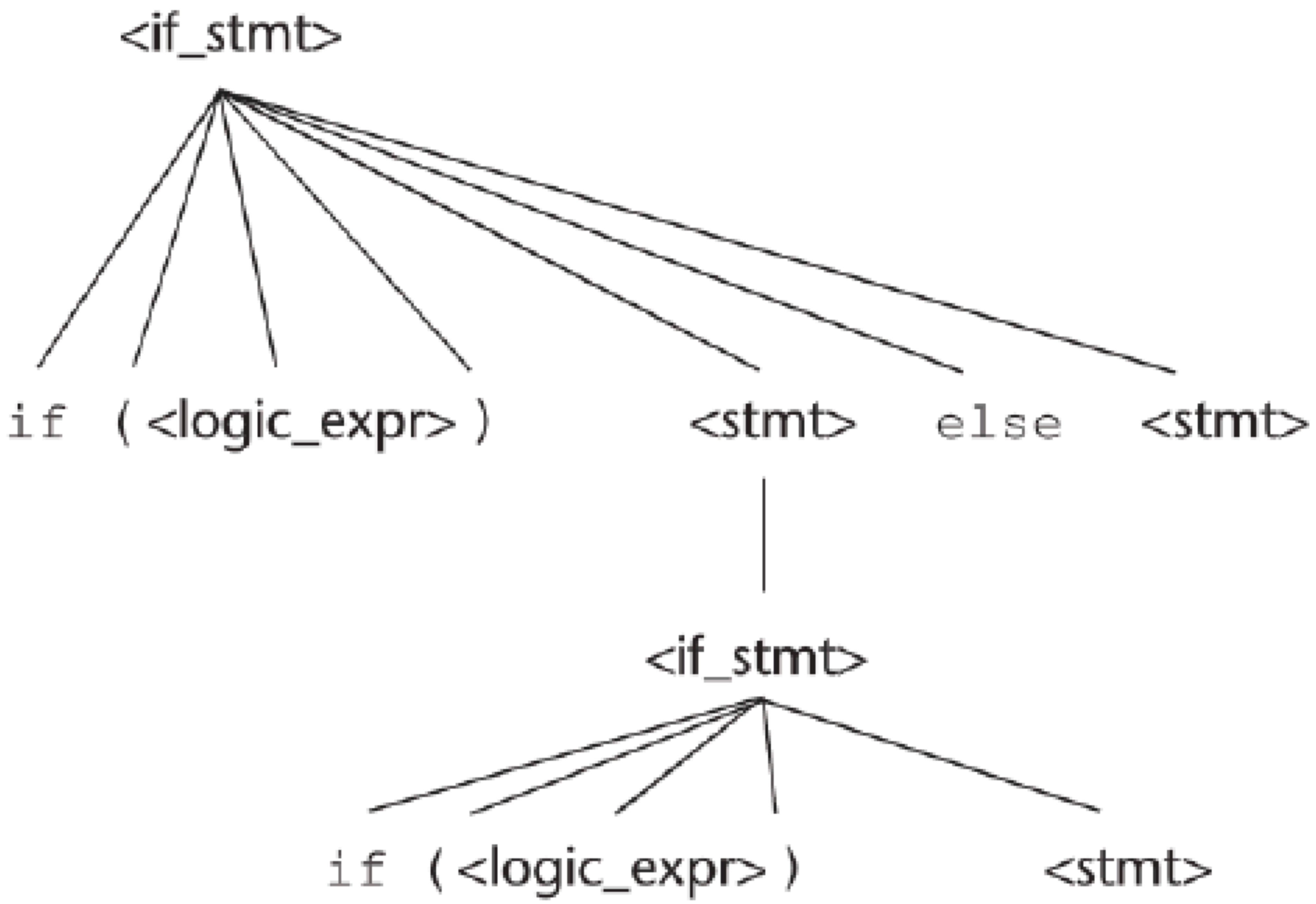
because + is earlier than * which is earlier than ** in the grammar.

if-else Grammar Rules

```
<if_stmt> : IF ( <logic_expr> ) <stmt>
<if_stmt> : IF ( <logic_expr> ) <stmt> ELSE <stmt>
<stmt> : <if_stmt>
```

This is an ambiguous grammar!






```
if ( <logic_expr> ) if ( <logic_expr> ) <stmt> else <stmt>
```

An unambiguous Grammar for if-else

The rule for if-else statements in most languages is that an else is matched with the nearest previous unmatched if.

Therefore, between an `if` and it's matching `else`, there cannot be an `if` statement without an `else` (an “unmatched” statement).

To make the grammar unambiguous, two new nonterminals are added, representing matched statements and unmatched statements:

```
<stmt> : <matched>
<stmt> : <unmatched>

<matched> : if ( <logic_expr> ) <matched> else <matched>
<matched> : any non-if statement

<unmatched> : if ( <logic_expr> ) <stmt>
<unmatched> : if ( <logic_expr> ) <matched> else <unmatched>
```

Attribute Grammars

- An **attribute grammar** can be used to describe more of the structure of a programming language than is possible with a context-free grammar.
- Attribute grammars are useful because some language rules (such as **type compatibility**) are difficult to specify with CFGs.
- Other language rules cannot be specified in CFGs at all, such as the rule that **all variables must be declared before they are referenced**.
- Rules such as these are considered to be part of the **static semantics** of a language, not part of the language's syntax. The term "static" indicates that these rules can be checked at compile time.
- Attribute grammars, designed by **Donald Knuth**, can describe both syntax and static semantics.

Attribute Grammars: continued

- An **attribute grammar** may be informally defined as a context-free grammar that has been **extended** to provide context sensitivity using a set of attributes, assignment of attribute values, evaluation rules, and conditions.
- A finite, possibly empty set of attributes is associated with each distinct symbol in the grammar.
- Each attribute has an associated domain of values, such as integers, character and string values, or more complex structures.
- *Viewing the input sentence (or program) as a parse tree, attribute grammars can pass values from a node to its parent, using a* ***synthesized*** *attribute, or from the current node to a child, using an* ***inherited*** *attribute.*
- In addition to passing attribute values up or down the parse tree, the attribute values may be assigned, modified, and checked at any node in the derivation tree.
- In particular, attribute grammars add the following to context-free grammars:
  - **Attributes** or properties that can have values assigned to them.
  - **Attribute computation functions** (semantic functions) that specify how attribute values are computed
  - **Predicate functions** that state the semantic rules of the language.

Attribute Grammars: Formal Definition

- An **attribute grammar** is a context-free grammar with the following additional features:
  - A set of attributes $A(X)$ for each grammar symbol X
  - A set of semantic functions and possibly an empty set of predicate functions for each grammar rule
- $A(X)$ consists of two disjoint sets $S(X)$ and $I(X)$, called **synthesized** and **inherited** attributes, respectively

  Synthesized attributes are used to pass semantic information <u>up</u> the parse tree

  Inherited attributes are used to pass semantic information <u>down</u> and <u>across</u> the tree
- For rule $X_0 : X_1, \ldots, X_n$ the **synthesized** attributes for X_0 are computed with a **semantic function** of the form $S(X_0) = f(A(X_1),\ldots,A(X_n))$.
- **Inherited** attributes for the symbol X_j , $1 <= j <= n$ (in the rule $X_0 : X_1, \ldots, X_n$) are computed with a **semantic function** of the form $I(X_j) = f(A(X_0),\ldots,A(X_n))$.

  To avoid circularity, inherited attributes are often restricted to functions of the form $I(X_j) = f(A(X_0),\ldots,A(X_{j-1}))$.
- A **predicate** function is a Boolean function on the union of attribute sets $A(X_0) \cup \ldots \cup A(X_n)$ and a set of literal attribute values. A derivation is allowed to proceed only if all predicates on the rule evaluate to true.

Attribute Grammars: continued

- A parse tree of an attribute grammar is the parse tree based on its underlying CFG, with a possibly empty set of attribute values attached to each node
- If all the attribute values in a parse tree have been computed, the tree is said to be **fully attributed**
- **Intrinsic attributes**: are synthesized attributes of leaf nodes whose values are determined outside the parse tree (coming from the Lexer)
- Initially, the only attributes with values are the intrinsic attributes of the leaf nodes. The semantic functions can then be used to compute the remaining attribute values.

Attribute Grammars: Example 1

The following fragment of an attribute grammar describes the rule that the name on the end of an **Ada**[1] procedure must match the procedure's name:

Syntax Rule:
```
<proc_def> : PROCEDURE PROCNAME[2] <proc_body> END PROCNAME[5]  SEMI
```

Predicate:
```
PROCNAME[2].value == PROCNAME[5].value
```

Here, we have introduced attribute, called `value`, which is associated with the terminal, PROCNAME

Nonterminals/Terminals that appear more than once in a rule are subscripted to distinguish them.

[1] Ada is the name of a famous programming language from the 70s/80s; used by DOD!

Attribute Grammars: Example 2

Type checking using Attribute Grammars.

Consider the following CFG:

```
assign : var = expr

expr : var + var

expr : var

var : A

var : B

var : C
```

with the following conditions:

- Variable types are either **int_type** or **real_type**
- Variables that are added need not be both of the same type. If the types are **different** then the resulting type is **real_type**
- The variable on the LHS of the assignment must have the same type as the expression on the RHS

Attribute Grammars: Example 2 continued

Attributes:

actual_type: A **synthesized** attribute associated with the nonterminals `<var>` and `<expr>`. Stores the actual type (int or real) of a variable or expression. In the case of a variable, the actual type is intrinsic.

expected_type: An **inherited** attribute associated with the nonterminal `<expr>`. Stores the type expected for the expression.

Complete Attribute Grammar

Syntax rule 1: <assign> : <var> = <expr>
Semantic rule: <expr>.expected_type = <var>.actual_type

Syntax rule 2: <expr> : <var>[2] + <var>[3]
Semantic rule:
<expr>.actual_type = **if** (<var>[2].actual_type == int_type) and
(<var>[3].actual_type == int_type)
then int_type
else real_type
Predicate: <expr>.actual_type == <expr>.expected_type

Syntax rule 3: <expr> : <var>
Semantic rule: <expr>.actual_type = <var>.actual_type
Predicate: <expr>.actual_type == <expr>.expected_type

Syntax rule 4 <var> : `A`
Semantic rule: <var>.actual_type = look-up(A.value)
Syntax rule 5 <var> : `B`
Semantic rule: <var>.actual_type = look-up(B.value)
Syntax rule 6 <var> : `C`
Semantic rule: <var>.actual_type = look-up(C.value)

The look-up function looks up a variable name in the **symbol table** and returns the variable's type.

Attribute Grammars: Example 2 continued

The process of **decorating** the parse tree with attributes could proceed in a completely top-down order if all attributes were inherited.

Alternatively, it could proceed in a completely bottom-up order if all the attributes were synthesized.

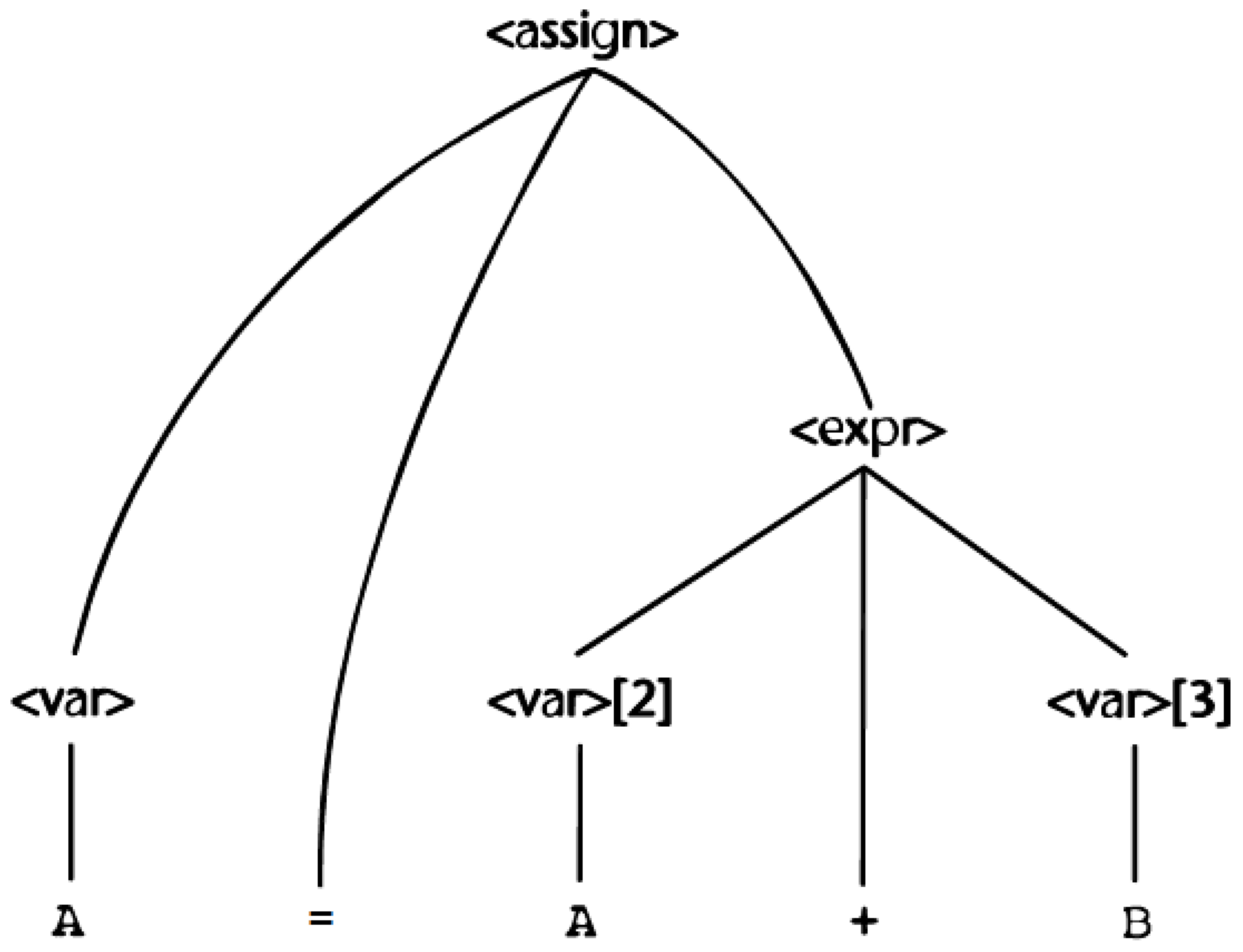
Because our grammar has both synthesized and inherited attributes, the evaluation process cannot be in any single direction. One possible order for attribute evaluation is:

(1) <var>.actual_type = look-up(A) (Rule 4)
(2) <expr>.expected_type = <var>.actual_type (Rule 1)
(3) <var>[2].actual_type = look-up(A) (Rule 4)
<var>[3].actual_type = look-up(B) (Rule 4)
(4) <expr>.actual_type = either int_type or real_type (Rule 2)
(5) <expr>.expected_type == <expr>.actual_type
is either true or false (Rule 2)

Determining attribute evaluation order is a complex problem, requiring the construction of a graph that shows all attribute dependencies.

Consider the assignment

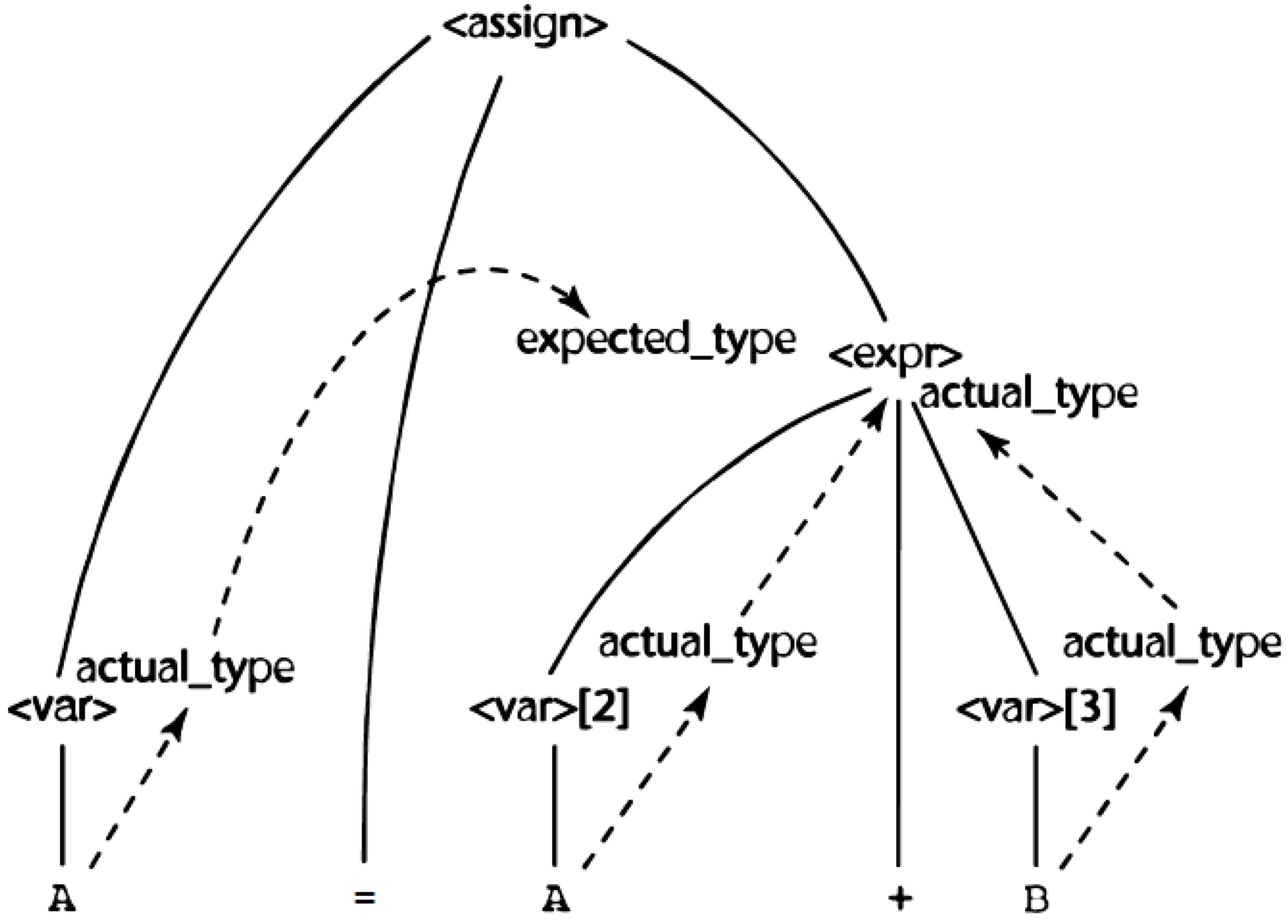
```
A = A + B
```



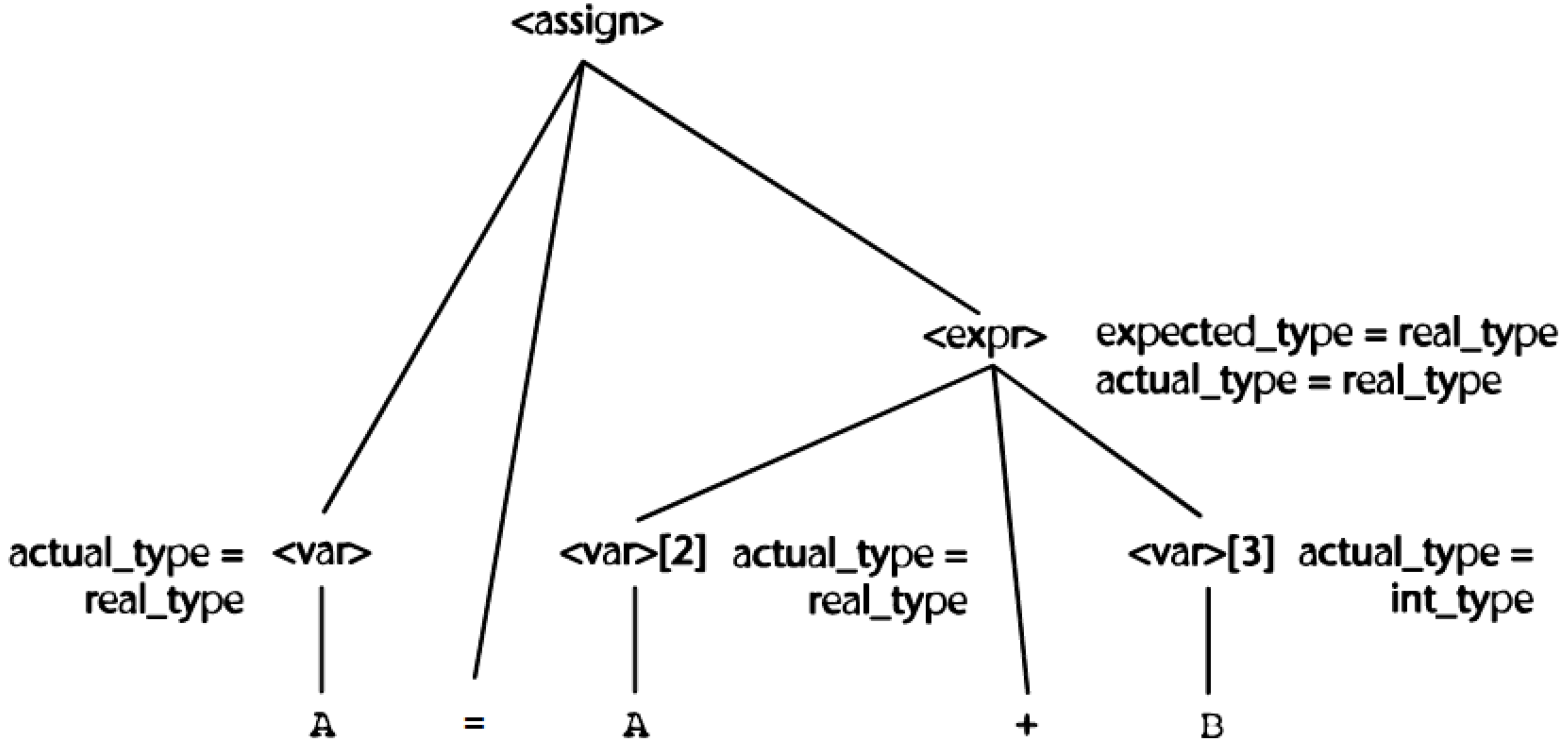
Parse Tree

Attribute Grammars: Example 2 continued

The following figure shows the flow of attribute values. Solid lines are used for the parse tree; dashed lines show attribute flow.



The following tree shows the final attribute values on the nodes.



Attribute Grammars: Example 3

Consider the language

$\{ A^n B^n C^n \mid n > 0 \} = \{ ABC, AABBCC, AAABBBCCC, \ldots \}$

It so happens that there is no CFG for this language!

But, we can devise an attribute grammar for this language.

Step 1: Devise a CFG for the language $\{ A^m B^n C^p \mid m > 0, n > 0, p > 0 \} = \{ABBBC, ABC, AAABC, \ldots\}$

<s> : <a> <b> <c>

<a> : A | <a> A

<b> : B | <b> B

<c> : C | <c> C

Step 2: Extend the CFG by introducing attributes. Introduce synthesized attribute **`count`** for non-terminals <a>, <b>, and <c>. The semantic functions and the predicate functions are shown in the next slide.

Attribute Grammars: Example 3 continued

Syntax rule 1: <s> : <a> <b> <c>
Predicate: <a>.count == <b.count> and <b>.count == <c>.count

Syntax rule 2: <a> : A
Semantic rule: <a>.count = 1

Syntax rule 3: <a> : <a> A
Semantic rule: <a>[0].count = <a>[1].count + 1

Syntax rule 4: <b> : B
Semantic rule: <b>.count = 1

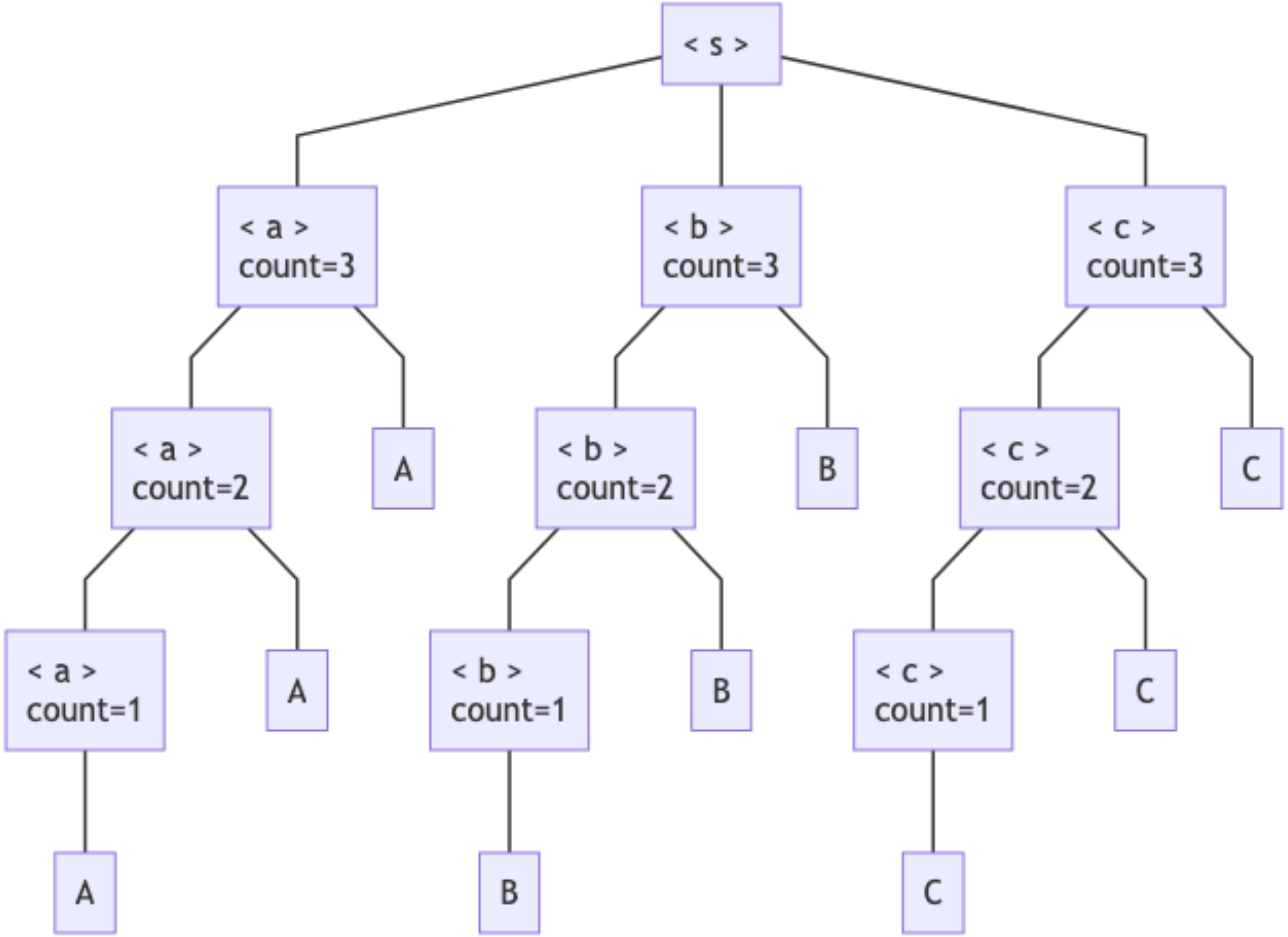
Syntax rule 5: <b> : <b> B
Semantic rule: <b>[0].count = <b>[1].count + 1

Syntax rule 6: <c> : C
Semantic rule: <c>.count = 1

Syntax rule 7: <c> : <c> C
Semantic rule: <c>[0].count = <c>[1].count + 1

Attribute Grammars: Example 3 continued

Parse tree for `AAABBBCCC` `<a>.count == <b>.count and <b>.count == <c>.count`



Describing the meaning of programs: Dynamic Semantics

Reasons for creating a formal semantic definition of a language:

- Programmers need to understand the meaning of language constructs in order to use them effectively.
- Compiler writers need to know what language constructs mean to correctly implement them.
- Programs could potentially be proven correct without testing.
- The correctness of compilers could be verified.
- Could be used to automatically generate a compiler.
- Would help language designers discover ambiguities and inconsistencies.

Semantics are typically described in English. Such descriptions are often imprecise and incomplete.

Operational Semantics

Operational semantics describes the meaning of programs by specifying the effects of running it on a machine.

- Using an actual machine language for this purpose is not feasible.
  - The individual steps and the resulting state are too small and too numerous.
  - The storage of a real computer is too large and complex, with several levels of memory (registers, cache, main memory etc)
- **Intermediate-level languages** and interpreters for virtualized computers are used instead.
- Each construct in the intermediate language must have an obvious and unambiguous meaning.
- Operational semantics is the method used in textbooks etc. to describe meaning of programming language constructs!

C for-loop

```
for (expr1; expr2; expr3) {
  stmts;
}
```

Meaning

```
expr1;
loop: if (expr2 == 0) goto out
  stmts;
  expr3;
  goto loop
out:
```

The human is the virtual computer who is assumed to execute these instructions correctly!

Operational Semantics (continued)

The following statements would be adequate for describing the semantics of the simple control statements of a typical programming language:

```
ident = var
ident = ident + 1
ident = ident - 1
ident = un_op var
ident = var bin_op var
goto label
if (var relop var) goto label
```

where `relop` is a relational operator, `ident` is an identifier, and `var` is either an identifier or a constant.

Adding a few more instructions would allow the semantics of arrays, records, pointers, and subprograms to be described.

Denotational Semantics

- **Denotational semantics**, which is based on recursive function theory, is the most rigorous and most widely known formal method for describing the meaning of programs.
- A denotational description of a language entity is a **function** that maps instances of that entity onto mathematical objects (e.g. numbers, sets of numbers, etc.)
- The term denotational comes from the fact that mathematical objects "denote" the meaning of syntactic entities.
- Each mapping function has a **domain** and a **range**:
  - The syntactic domain specifies which syntactic structures are to be mapped.
  - The range (a set of mathematical objects) is called the semantic domain.

Two Simple Examples

Example 1: Binary numbers

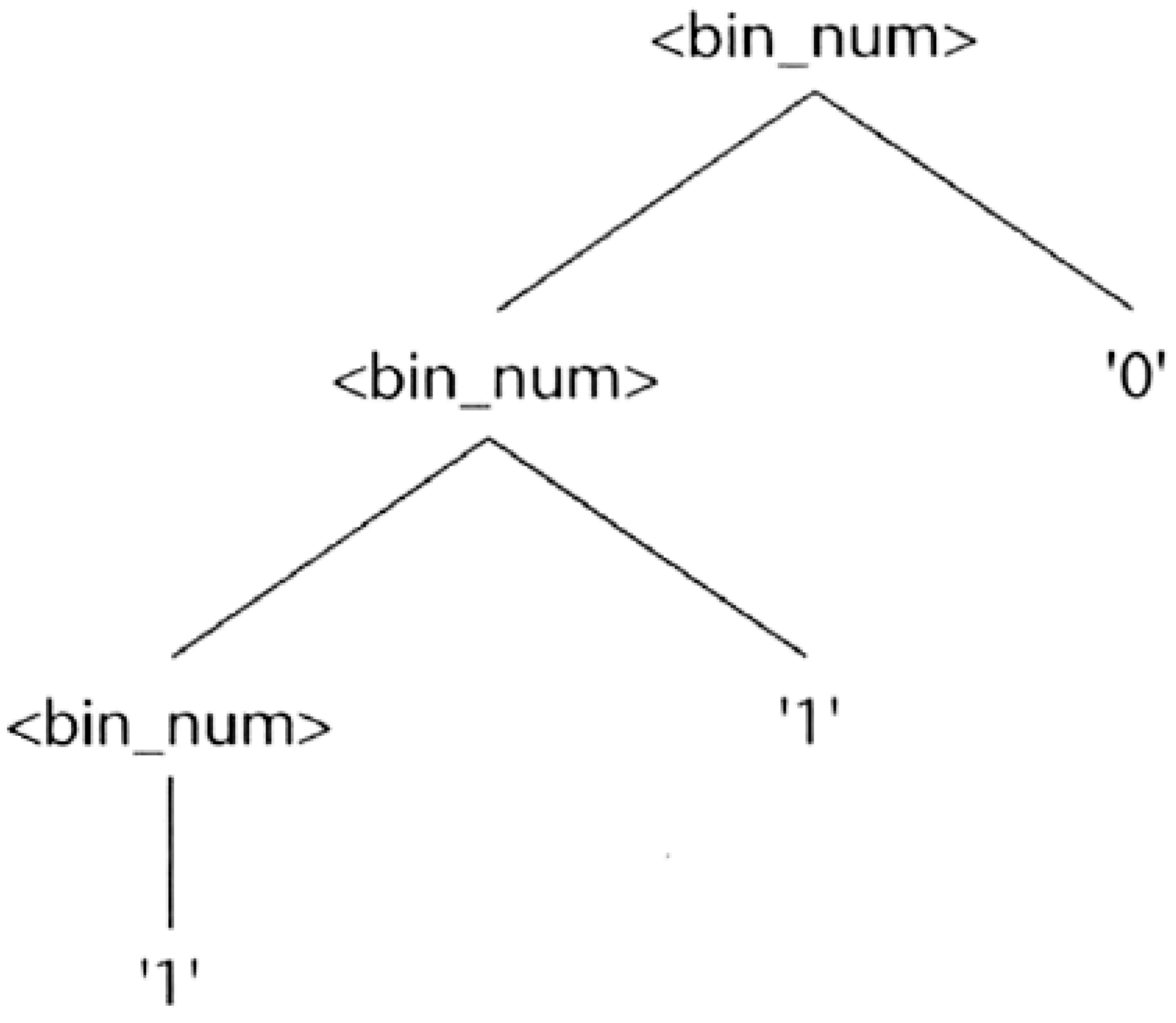
Consider the following grammar to specify string representation of binary numbers:

<bin_num> : ‘0’

<bin_num> : ‘1’

<bin_num> : <bin_num> ’0’

<bin_num> : <bin_num> ’1’



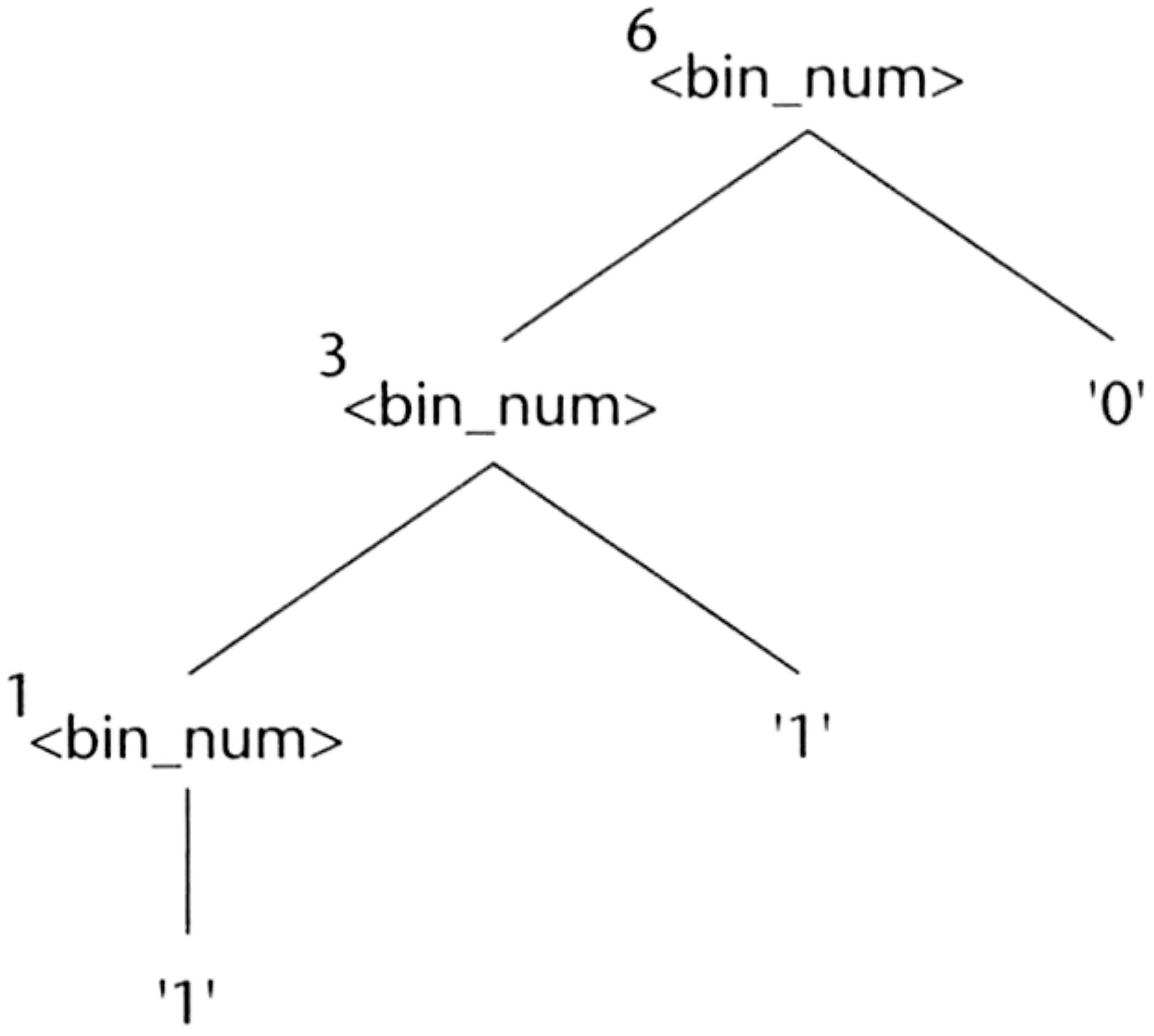
Parse tree for binary string 110

The denotational semantic function $\mathbf{M_{bin}}$ maps syntactic objects to nonnegative integers:

```
Mbin('0') = 0
Mbin('1') = 1
Mbin(<bin_num> '0') = 2 * Mbin(<bin_num>)
Mbin(<bin_num> '1') = 2 * Mbin(<bin_num>) + 1
```

Binary Numbers (continued)

The meanings, or denoted objects (integers in this case), can be attached to the nodes of the parse tree:



M_{bin}('110')
= 2 * M_{bin}('11') + 0
= 2 * (2 * M_{bin}('1') + 1) + 0
= 2 * (2 * 1 + 1) + 0
= 2 * 3 + 0
= 6

Decimal Numbers

Example 2: Decimal numbers

Grammar rules:

```
<dec_num> : '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | ‘9’

<dec_num> : <dec_num> ('0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | ‘9')
```

Denotational semantic mappings for these grammar rules:

$M_{dec}(\text{'0'}) = 0$
$M_{dec}(\text{'1'}) = 1$
…
$M_{dec}(\text{‘9’}) = 9$
$M_{dec}(\text{<dec\_num> '0'}) = 10 * M_{dec}(\text{<dec\_num>})$
$M_{dec}(\text{<dec\_num> '1'}) = 10 * M_{dec}(\text{<dec\_num>}) + 1$
…
$M_{dec}(\text{<dec\_num> '9'}) = 10 * M_{dec}(\text{<dec\_num>}) + 9$

$M_{dec}(\text{’736’})$
$= 10 * M_{dec}(\text{’73’}) + 6$
$= 10 * (10 * M_{dec}(\text{’7’}) + 3) + 6$
$= 10 * (10 * 7 + 3) + 6$
$= 10 * 73 + 6$
$= 736$

State of a Program

- The **state** of a program in denotational semantics consists of the values of the program's variables.
- Formally, the state s of a program can be represented as a set of ordered pairs:

$$s = \{\langle i_1, v_1\rangle, \langle i_2, v_2\rangle, \ldots, \langle i_n, v_n\rangle\}$$

where, $i_1, \ldots, i_n$ are names of variables and the $v_1, \ldots, v_n$ their associated current values.

- Any of the `v`'s can have the special value **`undef`**, which indicates that its associated variable is currently undefined.
- Let `VARMAP` be a function of two parameters, a variable name and the program state. The value of $\texttt{VARMAP}(i_j, s)$ is v_j.
- Most semantics mapping functions for language constructs map <u>states to states</u>. These state changes are used to define the meanings of the constructs.
- Some constructs, such as expressions, are mapped to values, not states.

Expressions

- In order to develop a concise denotational definition of the semantics of expressions, the following simplifications will be assumed:
  - No side effects.
  - Operators are `+` and `*`.
  - At most one operator.
  - Operands are scalar integer variables and integer literals.
  - No parentheses.
  - Value of an expression is an integer.
  - Errors never occur during evaluation; however, the value of a variable may be undefined.
- Here is a CFG for such expressions:

```
<expr> : <dec_num> | <var> | <binary_expr>
<binary_expr> : <left_expr> <operator> <right_expr>
<left_expr> : <dec_num> | <var>
<right_expr> : <dec_num> | <var>
<operator> : + | *
```

Expressions: M_E

```
<expr> : <dec_num> | <var> | <binary_expr>
<binary_expr> : <left_expr> <operator> <right_expr>
<left_expr> : <dec_num> | <var>
<right_expr> : <dec_num> | <var>
<operator> : + | *
```

```
ME(<expr>, s) =
  case <expr> of
    <dec_num> => Mdec(<dec_num>)
    <var> => if (VARMAP(<var>,s) == undef) then error else VARMAP(<var>,s)
    <binary_expr> =>
      if (ME(<binary_expr>.<left_expr>,s)  == error OR
          ME(<binary_expr>.<right_expr>,s) == error)
      then error
      elif (<binary_expr>.<operator> == '+')
      then ME(<binary_expr>.<left_expr>,s) + ME(<binary_expr>.<right_expr>,s)
      else ME(<binary_expr>.<left_expr>,s) * ME(<binary_expr>.<right_expr>,s)
```

Expressions: M_E Example

Given state s `= { (x,36), (y,20), (z,15) }`

```
ME(x + 24,s)
= ME(x,s) + ME(24,s)
= VARMAP(x,s) + Mdec(24)
= 36 + 10*Mdec(2) + 4
= 36 + 10*2 + 4
= 60
```

```
ME(x + u,s)
= ME(x,s) + ME(u,s)
= VARMAP(x,s) + ME(u,s) + 4
= 36 + ME(u,s) + 4
= error
```

```
ME(15 + 24,s)
= ME(15) + ME(24)
= Mdec(15) + Mdec(24)
= 10*Mdec(1)+5 + 10*Mdec(2)+4
= 10*1+5 + 10*2+4
= 39
```

Assignment Statement: M_A

```
MA(x = E, s) =
  if ME(E, s) == error
  then error
  else s' = {<i1, v1'>, ..., <in, vn'>},
       where for j = 1, …, n
             if (ij == x) // note that we are comparing names of variables here
             then vj' = ME(E, s)
             else vj' = VARMAP(ij, s)
```

Note: This mapping for assignment statement does not return any mathematical object, but simply returns a new state s′ of the program.

Assignment Statement: M_A Example

Given state s `= { (x,36), (y,20), (z,15) }`

```
To compute MA(x = x + 24, s)

first calculate meaning of RHS of assignment statement
ME(x + 24,s)
= ME(x,s) + ME(24,s)
= VARMAP(x,s) + Mdec(24)
= VARMAP(x,s) + 10*Mdec(2) + 4
= 36 + 10*2 + 4
= 60

Then, return the state s' = { (x,60), (y,20), (z,15) }
```

While Loop: M_L

```
ML(while B do L, s) =
  if MB(B, s) == error then error
  elif MB(B, s) == false then s
  elif MSL(L, s) == error then error
  else ML(while B do L, MSL(L, s))
```

Note:

M_{SL} maps statement lists to states.

M_B maps Boolean expressions to Boolean values (or error).

We assume that these two mappings are defined elsewhere.

While Loop: M_L Example

Consider the statement:

`while (count > 0) {x = x + 24; count = count - 1}`

and the initial state **`s1 = { (x,36), (count,1), (z,15) }`**

```
MB((count > 0),s1) = TRUE

ML(while (count > 0) {x = x + 24; count = count - 1},s1)
= ML(while (count > 0) {x = x + 24; count = count - 1},s2)

where, s2 = MSL({x = x + 24; count = count - 1},s1)
          = { (x,60), (count,0), (z,15) }

MB((count > 0),s2) = FALSE

ML(while (count > 0) {x = x + 24; count = count - 1},s2)
= s2
```

Final answer: `s2 = { (x,60), (count,0), (z,15) }`