CHAPTER 2(1)

Describing Syntax and Semantics

TOPICS

- Introduction
- The General Problem of Describing Syntax
- Formal Methods of Describing Syntax
- Attribute Grammars
- Describing the Meanings of Programs: Dynamic Semantics

INTRODUCTION

- **Syntax:** the form or structure of the expressions, statements, and program units
- **Semantics:** the meaning of the expressions, statements, and program units
- Eg.

```
if (condition)
  statement1
else
  statement 2
```

The General Problem of Describing Syntax: Terminology

- A *sentence* is a string of characters over some alphabet
- A *language* is a set of sentences
- A *lexeme* is the lowest level syntactic unit of a language, including
  - Numeric literals
  - Operators
  - Special words
- A *token* is a category of lexemes, such as
  - Identifier
  - Arithmetic plus operator et cetera.

The General Problem of Describing Syntax: Terminology

- Consider the following statement:

```
index = 2 * count + 17;
```

Lexemes	Token
index	identifier
=	equal_sign
2	int_literal
*	mult_op
count	identifier
+	plus_op
17	int_literal
;	semicolon

The General Problem of Describing Syntax: Terminology

- **2 distinct ways of defining a language:**
  - **Language recognizers**
    - A recognition device reads input strings of the language and decides whether the input strings belong to the language
    - R(∑)= L?
    - Example: syntax analysis part of a compiler
  - **Language generators**
    - A device that generates sentences of a language
    - One can determine if the syntax of a particular sentence is correct by comparing it to the structure of the generator

Formal Methods of Describing Syntax

- Context-Free Grammars (CFG)
- Backus-Naur Form (BNF) and
- Extended BNF

Formal Methods of Describing Syntax: CFG

- Context-Free Grammars
  - Developed by Noam Chomsky, a noted linguist, in the mid-1950s
  - Two of the four generative devices (grammars), meant to describe the four classes of (natural) languages are found to be useful for describing the syntax of programming languages.
  - These are
    - Context-free grammars: describe the syntax of whole programming languages
    - Regular grammars: describe the forms of the tokens of programming languages

Formal Methods of Describing Syntax: BNF

- Backus-Naur Form
  - A formal notation for specifying programming language syntax initially presented by John Backus to describe ALGOL 58 in 1959.
  - It was later modified slightly by Peter Naur to describe ALGOL 60, hence the Backus-Naur Form (BNF).
  - Most popular method for concisely describing programming language syntax

Formal Methods of Describing Syntax: BNF

- BNF is a *metalanguage* for programming languages
- It uses abstractions to represent classes of syntactic structures
- An abstraction of a JAVA assignment statement:

<assign> → <var> = <expression>

A rule

The left-hand side (LHS) i.e. the abstraction being defined

The right-hand side (RHS) i.e. the definition of the LHS

Formal Methods of Describing Syntax: BNF

- A possible instantiation of the previous rule is:

  total = subtotal1 + subtotal2

- The RHS, i.e. the definition can be a mixture of
  - tokens
  - lexemes and
  - references to other abstractions

Formal Methods of Describing Syntax: BNF

- BNF abstractions are often called the *non-terminal symbols* or in short the *nonterminals*.
- Likewise, the lexemes and tokens are called the *terminal symbols* or the *terminals*.
- BNF grammar is therefore a collection of rules.

Formal Methods of Describing Syntax: BNF

- Nonterminals can have two or more distinct definitons.
- Consider another example of BNF rules:

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

which, can also be written as a single rule separated by the | symbol to mean logical OR.

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

Formal Methods of Describing Syntax: BNF

- BNF uses **recursion** to describe lists of syntactic elements in programming languages.
- A rule is recursive if its LHS appears in its RHS. For example:

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

Formal Methods of Describing Syntax: BNF

- **Derivation** is a process of generating sentences through repeated application of rules, starting with the *start symbol*.
- Consider the following BNF grammar:

```
<program> → begin <stmt_list> end
<stmt_list> → <stmt>
                | <stmt>; <stmt_list>
<stmt> → <var> = <expression>
<var> → A | B | C
<expression> → <var> + <var>
                | <var> - <var>
                | <var>
```

Formal Methods of Describing Syntax: BNF

- A derivation of the previous grammar is:

```
<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 + C; <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
```

Formal Methods of Describing Syntax: BNF

- Every string in the derivation is called a *sentential form*
- A *sentence* is a sentential form that has only terminal symbols
- Previous derivation is called *leftmost derivation*, where the leftmost nonterminal in each sentential form is expanded.
- A *rightmost* derivation is also possible, neither leftmost nor rightmost derivation is also possible.
- It is not possible to exhaustively generate all possible sentences in finite time.

Formal Methods of Describing Syntax: BNF

- Exercise: Create a derivation from this grammar

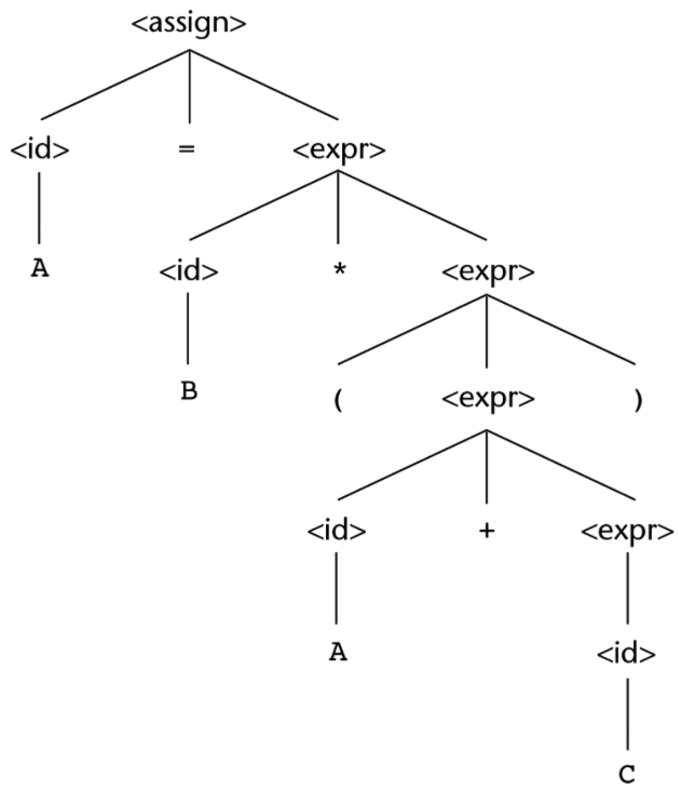
```
<assign> → <id> = <expr>
<id> → A | B | C
<expr> →  <id> + <expr>
         |<id> * <expr>
         | ( <expr> )
         |<id>
```

FORMAL METHODS OF DESCRIBING SYNTAX: BNF

- *Parse tree* is a hierarchical syntactic structure of a derived sentence.

Figure 3.1

A parse tree for the simple statement
`A = B * (A + C)`



Formal Methods of Describing Syntax: BNF

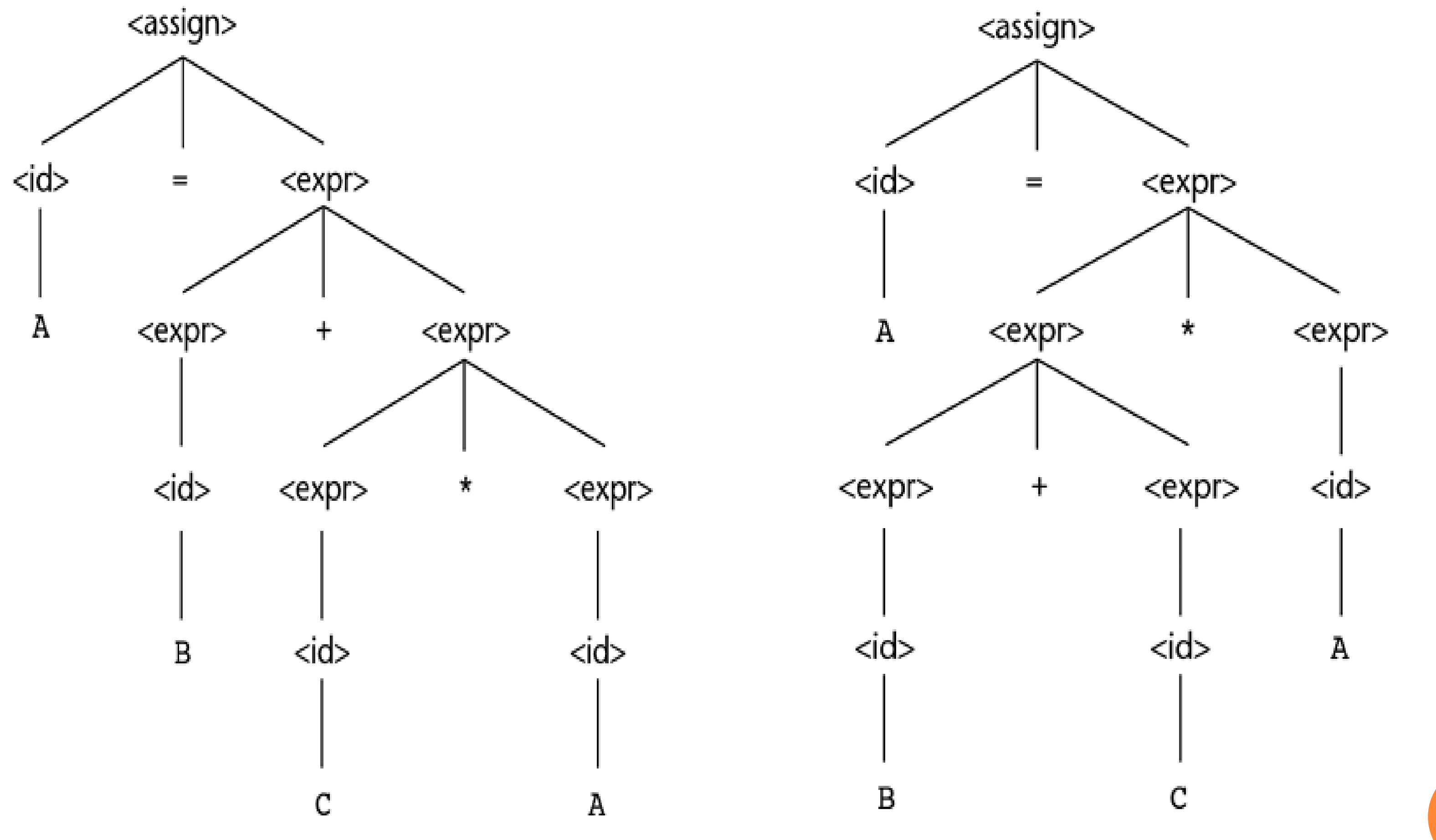
- Every internal node of a parse tree is labeled with a nonterminal symbol.
- Every leaf is labeled with a terminal symbol.
- Every subtree of a parse tree describes one instance of an abstraction in the sentence.

Formal Methods of Describing Syntax: BNF

- A grammar is *ambiguous* if it generates a sentential form that has two or more distinct parse trees.
- Consider the following BNF grammar:

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

Formal Methods of Describing Syntax: BNF



Formal Methods of Describing Syntax: BNF

- Solutions to ambiguity:
  - Operator precedence
    - Assigning different precedence levels to operators
  - Associativity of operators
    - Specifies ‘precedence’ of two operators that have the same precedence level.

FORMAL METHODS OF DESCRIBING SYNTAX: BNF

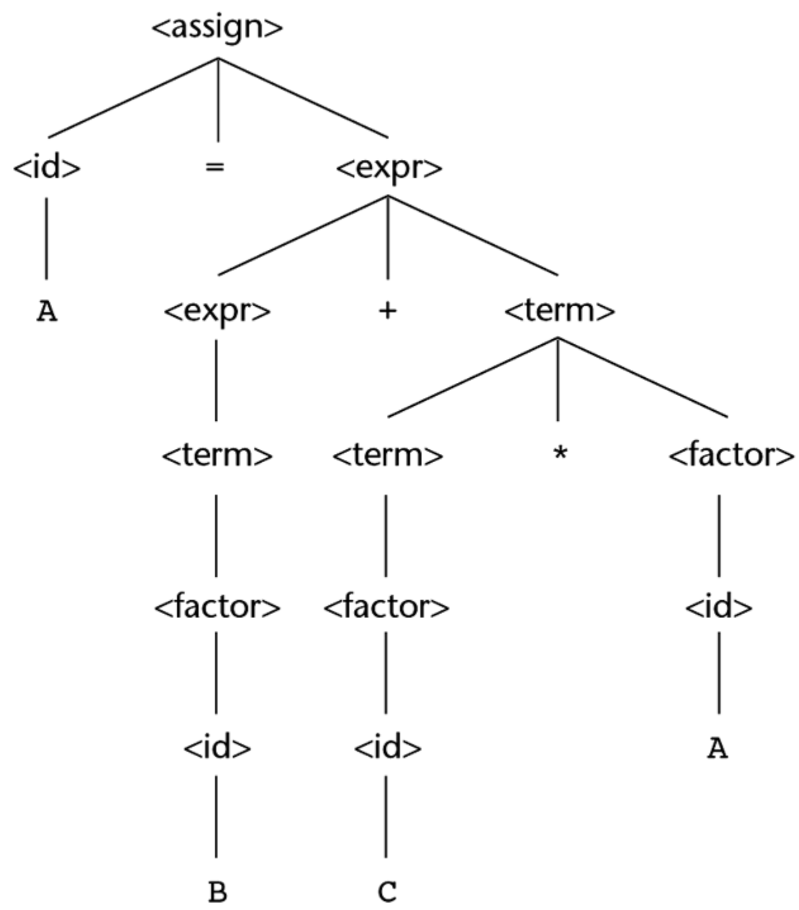
- An example of an unambiguous grammar that defines operator precedence:

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

FORMAL METHODS OF DESCRIBING SYNTAX: BNF

Figure 3.3

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



Formal Methods of Describing Syntax: BNF

- A rule is said to be **left recursive** if its LHS is also appearing at the beginning of its RHS.
- Likewise, a grammar rule is **right recursive** if the LHS appears at the right end of the RHS.

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

Formal Methods of Describing Syntax: Extended BNF

- Improves readability and writability of BNF
- Three common extensions are:
  1. Optional parts of an RHS, delimited by square brackets. E.g.

EBNF Format

```
<if_stmt> → if (<logic_expr>) <stmt> [else<stmt>]
```

BNF Format

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

Formal Methods of Describing Syntax: Extended BNF

2. The use of curly braces in an RHS to indicate that the enclosed part can be repeated indefinitely or left out altogether.

EBNF Format

```
<ident_list> → <identifier> {, <identifier>}
```

BNF Format

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

Formal Methods of Describing Syntax: Extended BNF

3. For multiple choice options, the options are placed inside parentheses and separated by the OR operator.

EBNF Format

```
<term> → <term> (*|/|%) <factor>
```

BNF Format

```
<term> → <term> * <factor>
       | <term> / <factor>
       | <term> % <factor>
```

Formal Methods of Describing Syntax: Extended BNF

- BNF

```
<expr> → <expr> + <term>
         | <expr> - <term>
         | <term>
<term> → <term> * <factor>
         | <term> / <factor>
         | <factor>
<factor> → <exp> ** <factor>
         | <exp>
<exp> → ( <expr> )
         | id
```

Formal Methods of Describing Syntax: Extended BNF

- EBNF

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

Formal Methods of Describing Syntax: Extended BNF

- Other variations of EBNF:
  - Numeric superscript attached to the right curly brace to indicate repetition upper limit.
  - A plus (+) superscript to indicate one or more repetition.
  - A colon used in place of the arrow and the RHS is moved to the next line.
  - Alternative RHSs are separated by new line rather than vertical bar.
  - Subscript opt is used to indicate something being optional rather than square brackets.
  - Et cetera …

ATTRIBUTE GRAMMARS

- An extension to CFG that allows some characteristics of the structure of programming languages that are either difficult or impossible to be described using BNF.
  - Difficult e.g. type compatibility
  - Impossible e.g. all variable must be declared before they are referenced.
- Therefore, the need for *static semantic* rules e.g. attribute grammars.
- The additions are:
  - attributes
  - attribute computation functions
  - predicate functions

Attribute Grammars: Definition

- Associated with each grammar symbol X is a set of attributes A(X).
- The set A(X) consists of two disjoint sets,
  - S(X), **synthesised attributes**, used to pass semantic information up a parse tree and
  - I(X), **inherited attributes**, used to pass semantic information down and across a tree.

Attribute Grammars: Definition

- Associated with each grammar rule is a set of semantic functions and a possibly empty set of predicate functions over the attributes of the symbols in the grammar rule.
  - For a rule $X_0 \rightarrow X_1 \ldots X_n$, the synthesised attributes of X_0 are computed with semantic functions of the form:

$$S(X_0) = f(A(X_1), \ldots , A(X_n))$$

  - Likewise, inherited attributes of symbols X_j, $1 \leq j \leq n$ are computed with a semantic function of the form:

$$I(X_j) = f(A(X_0), \ldots , A(X_n))$$

Attribute Grammars: Definition

- A predicate function has the form of a Boolean expression on the union of the attribute set $\{A(X_0), \ldots, A(X_n)\}$
  - The only derivations allowed with an attribute grammar are those in which every predicate associated with every nonterminal is true.

- Intrinsic attributes are synthesised attributes of leaf nodes whose values are determined outside the parse tree

Attribute Grammars: Example

- Syntax rule:

```
<proc_def> → procedure <proc_name>[1]
             <proc_body> end <proc_name>[2];
```

- Predicate:

```
<proc_name>[1].string == <proc_name>[2].string
```

- I.e. the predicate rule states that the name string attribute of the <proc_name> nonterminal in the subprogram header must match the name string attribute of the <proc_name> nonterminal following the end of the subprogram.

Attribute Grammars: Example

- Consider the following grammar

```
<assign> → <var> = <expr>
<expr> → <var> + <var>
        | <var>
<var> → A | B | C
```

Attribute Grammars: Example

- And the following requirements …
  - The variables can be one of two types, int or real
  - When there are two variables on the right side of an assignment, they need not be the same type
  - The type of the expression when the operand types are not the same is always real
  - When they are the same, the expression type is that of the operands.
  - The type of the left side of the assignment must match the type of the right side
  - So, the types of operands in the right side can be mixed, but the assignment is valid only if the target and the value resulting from evaluating the right side have the same type.

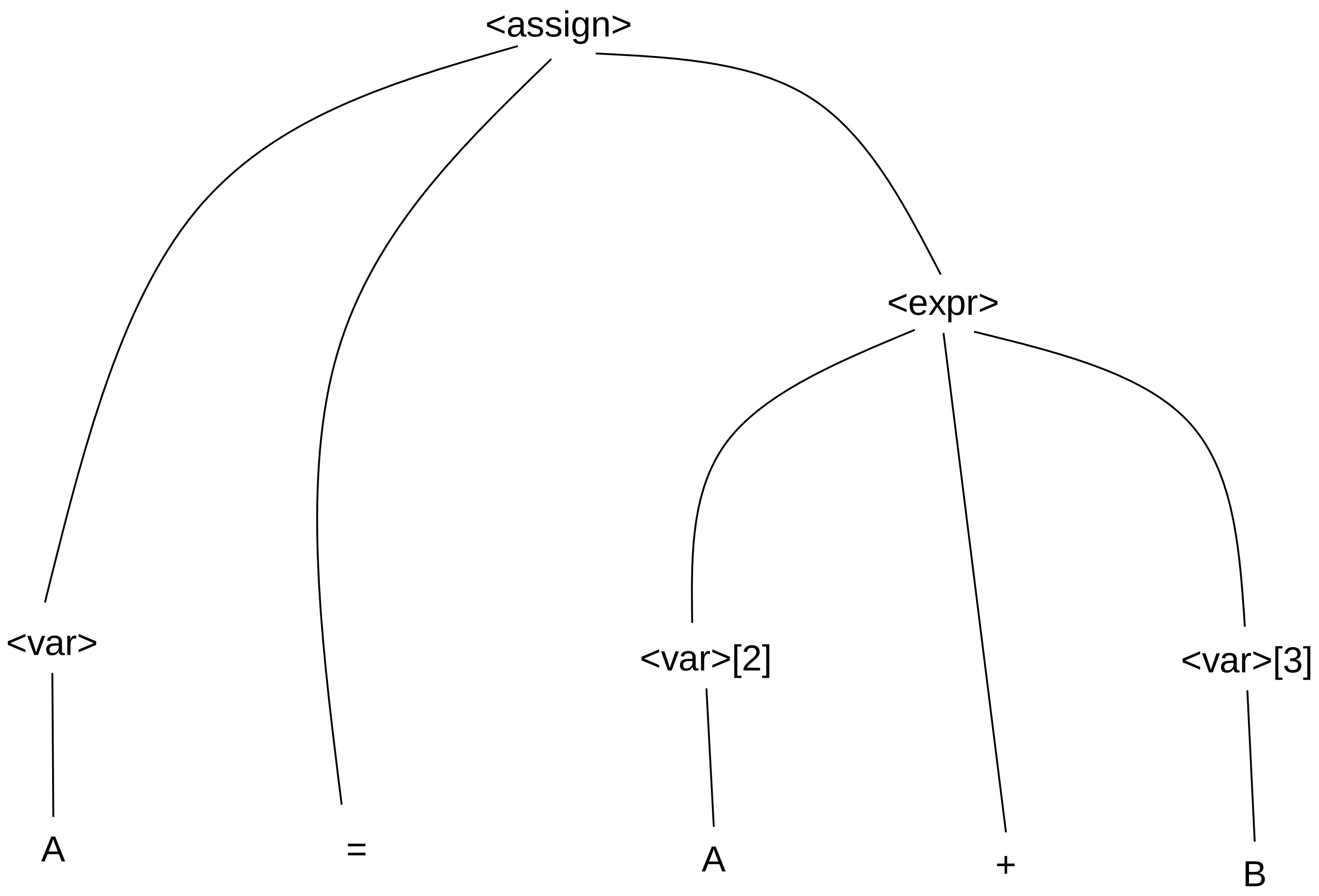
Attribute Grammars: Example

EXAMPLE 3.6	An Attribute Grammar for Simple Assignment Statements

1. Syntax rule: <assign> → <var> = <expr>
   Semantic rule: <expr>.expected_type ← <var>.actual_type
2. Syntax rule: <expr> → <var>[2] + <var>[3]
   Semantic rule: <expr>.actual_type ←
   if (<var>[2].actual_type = int) and
   (<var>[3].actual_type = int)
   then int
   else real
   end if
   Predicate: <expr>.actual_type == <expr>.expected_type
3. Syntax rule: <expr> → <var>
   Semantic rule: <expr>.actual_type ← <var>.actual_type
   Predicate: <expr>.actual_type == <expr>.expected_type
4. Syntax rule: <var> → A | B | C
   Semantic rule: <var>.actual_type ← look-up(<var>.string)

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

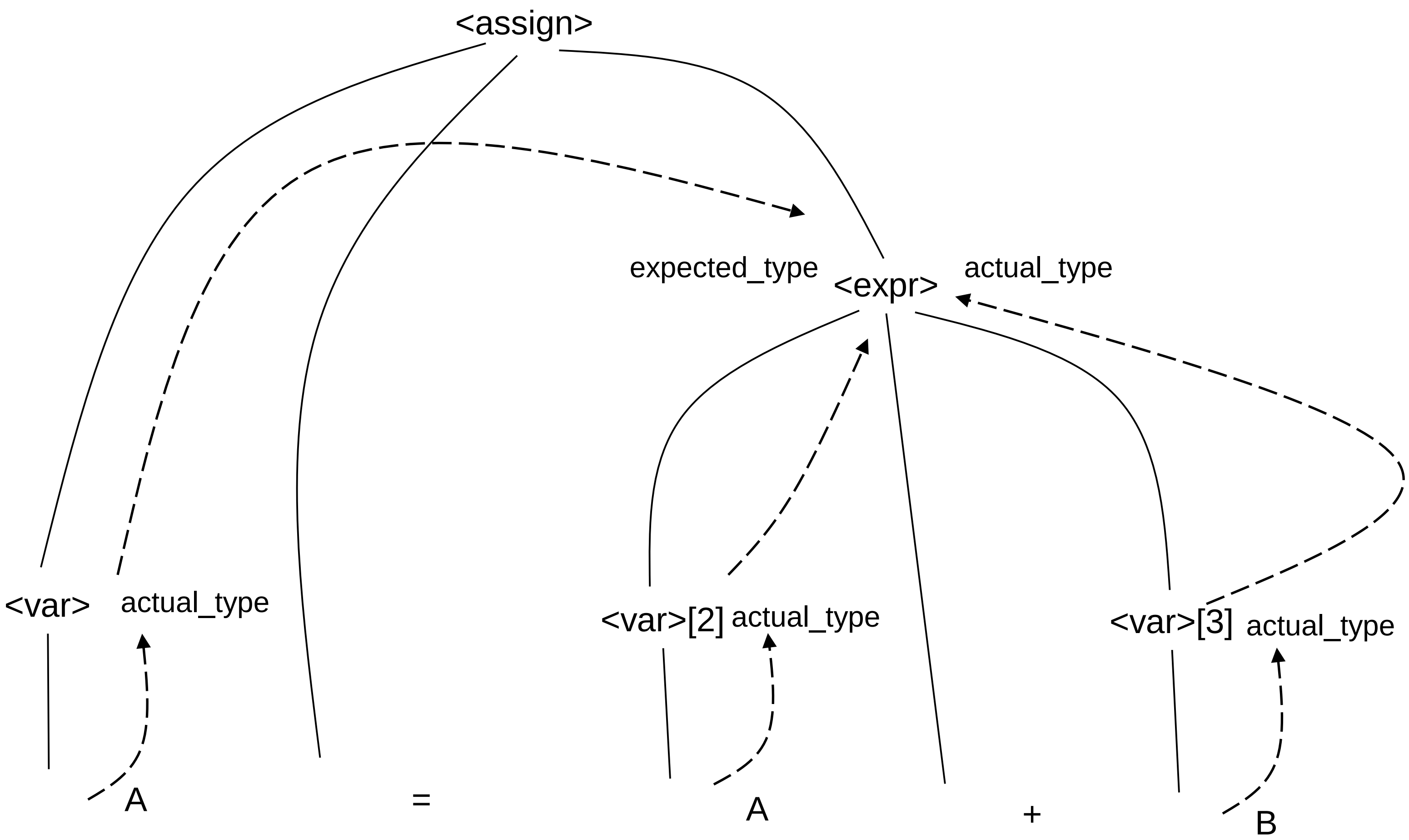
ATTRIBUTE GRAMMARS: EXAMPLE



Attribute Grammars: Computing Attribute Values

- If all attributes were inherited, the tree could be decorated in top-down order.
- If all attributes were synthesized, the tree could be decorated in bottom-up order.
- In many cases, both kinds of attributes are used, and it is some combination of top-down and bottom-up orders. E.g.:
  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 or real (Rule 2)
  5. <expr>.expected_type == <expr>.actual_type is either TRUE or FALSE (Rule 2)

Attribute Grammars: Computing Attribute Values



ATTRIBUTE GRAMMARS: COMPUTING ATTRIBUTE VALUES