

Chapter 3

Describing Syntax and Semantics

Chapter 3 Topics

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

3.1 Introduction

- **Syntax:** The syntax of a programming language is the form or structure of the expressions, statements, and program units.
- **Semantics:** The semantics is the meaning of the expressions, statements, and program units.
e.g., **while** statement in Java
while (boolean_expr) statement
- Syntax and semantics provide a language's definition
 - Users of a language definition
 - Other language designers
 - Implementers
 - Programmers (the users of the language)

3.2 The General Problem of Describing Syntax: Terminology

- A *sentence* is a string of characters over some alphabet
- A *language* is a set of strings of characters from some alphabet.
- Formal description of the syntax of programming languages often do not include the description of lexemes.
- A *lexeme* is the lowest level syntactic unit of a language (e.g., `*`, `sum`, `begin`). The lexemes of a programming language include its numeric literals, operators, and special words, among others. Each lexeme group is represented by a name or token.
- A *token* of a language is a category of lexemes.

e.g. `index = 2 * count + 17`

Lexemes

`index`

`=`

`2`

`*`

`count`

`+`

`17`

`;`

Tokens

`identifier`

`equal_sign`

`int_literal`

`mult_op`

`identifier`

`plus_op`

`int_literal`

`semicolon`

In general, languages can be formally defined in two distinct ways: by recognition and by generation.

- **Language Recognizers** accept a Language. Recognizers are Machines.
M: Machine; **L**: Language; **S**: String;
Machines take a string as input. The Machine will accept the input when it runs, the Machine stops at an accept state.
If M recognizes all S in L, **M is said to accept S**.
Otherwise **M is said to reject S**. S is in L if and only if M accepts S.
Example: syntax analysis part of a compiler (Details -> Chapter 4)
- **Language Generators** create the strings of a Language.
Generators are string constructors. A generator provides a construction description. If a generator is able to construct all strings in a Language L, and every string S that can be constructed by that generator is in L, we can say that the generator is a generator for the language L. If there is no way to construct a string S from the generator, S is not in L.

Context Free Grammars (CFGs) are a well known type of language generators.

Push Down Automatas (PDAs) are a well known form of language recognizers.

CFGs generate the same class of languages that are recognized by PDAs.

3.3 Formal Methods of Describing Syntax

- **Context-Free Grammars (CFGs)**
 - Developed by Noam Chomsky in the mid-1950s
 - Language generators, meant to describe the syntax of natural languages
 - Define a class of languages called context-free languages
- **Backus-Naur Form (1959) or Backus Normal Form (BNF)**
 - Invented by John Backus to describe Algol 58
 - **BNF is equivalent to context-free grammars**

BNF Fundamentals

A *metalanguage* is a language used to define other languages.
BNF is a metalanguage for programming languages.

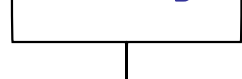
A *grammar* is a metalanguage used to define the syntax of a language.

Our interest : using grammars to define the syntax of a programming language.

- BNF uses abstractions for syntactic structures, which act like syntactic variables (also called *nonterminal symbols*, or just *nonterminal*)

e.g., Java assignment statement can be given by

`<assign> → <var> = <expression>`


abstraction to
be defined


definition of LHS

- **BNF rule** (or **production**): It is a definition of an abstraction. Its rule has a left-hand side (LHS), which is a nonterminal, and a right-hand side (RHS), which is a string of terminals and/or nonterminals
- **Terminals** are lexemes or tokens
- **Nonterminals** are often enclosed in angle brackets

e.g., BNF rules:

`<if_stmt> → if(<logic_expr>) <stmt>`

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

or with the rule

`<if_stmt> → if(<logic_expr>) <stmt>`

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

Describing lists:

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

Grammars and Derivations

- **Grammar**: a finite non-empty set of rules
- A **derivation** is a repeated application of rules, starting with the start symbol and ending with a sentence (all terminal symbols)

Example 3.1 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>

```

An Example Derivation

```
<program> => begin <stmt_list> end  
=> begin <stmt> ; <stmt_list> end  
=> begin <var> = <exprssion>; <stmt_list> end  
=> begin A = <exprssion>; <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 ; <var> = <exprssion>; end  
=> begin A = B + C ; B = <exprssion>; end  
=> begin A = B + C ; B = <var>; end  
=> begin A = B + C ; B = C; end
```

Derivations

- *Sentential form*: Every string of symbols in a derivation, including <program> is a *sentential form*.
- **Generated sentence**: A sentential form that has only terminal symbols.
- *Leftmost derivation*: A derivation in which the leftmost nonterminal in each sentential form is expanded
- A derivation may be neither leftmost nor rightmost

Example 3.2 A Grammar for Simple Assignments

$\langle \text{assign} \rangle \rightarrow \langle \text{id} \rangle = \langle \text{expr} \rangle$

$\langle \text{id} \rangle \rightarrow A \mid B \mid C$

$\langle \text{expr} \rangle \rightarrow \langle \text{id} \rangle + \langle \text{expr} \rangle$
 $\mid \langle \text{id} \rangle * \langle \text{expr} \rangle$
 $\mid (\langle \text{expr} \rangle)$
 $\mid \langle \text{id} \rangle$

$A = B * (A + c)$

is generated by leftmost derivation:

$\langle \text{assign} \rangle \Rightarrow \langle \text{id} \rangle = \langle \text{expr} \rangle$

$\Rightarrow A = \langle \text{expr} \rangle$

$\Rightarrow A = \langle \text{id} \rangle * \langle \text{expr} \rangle$

$\Rightarrow A = B * \langle \text{expr} \rangle$

$\Rightarrow A = B * (\langle \text{expr} \rangle)$

$\Rightarrow A = B * (\langle \text{id} \rangle + \langle \text{expr} \rangle)$

$\Rightarrow A = B * (A + \langle \text{expr} \rangle)$

$\Rightarrow A = B * (A + \langle \text{id} \rangle)$

$\Rightarrow A = B * (A + C)$

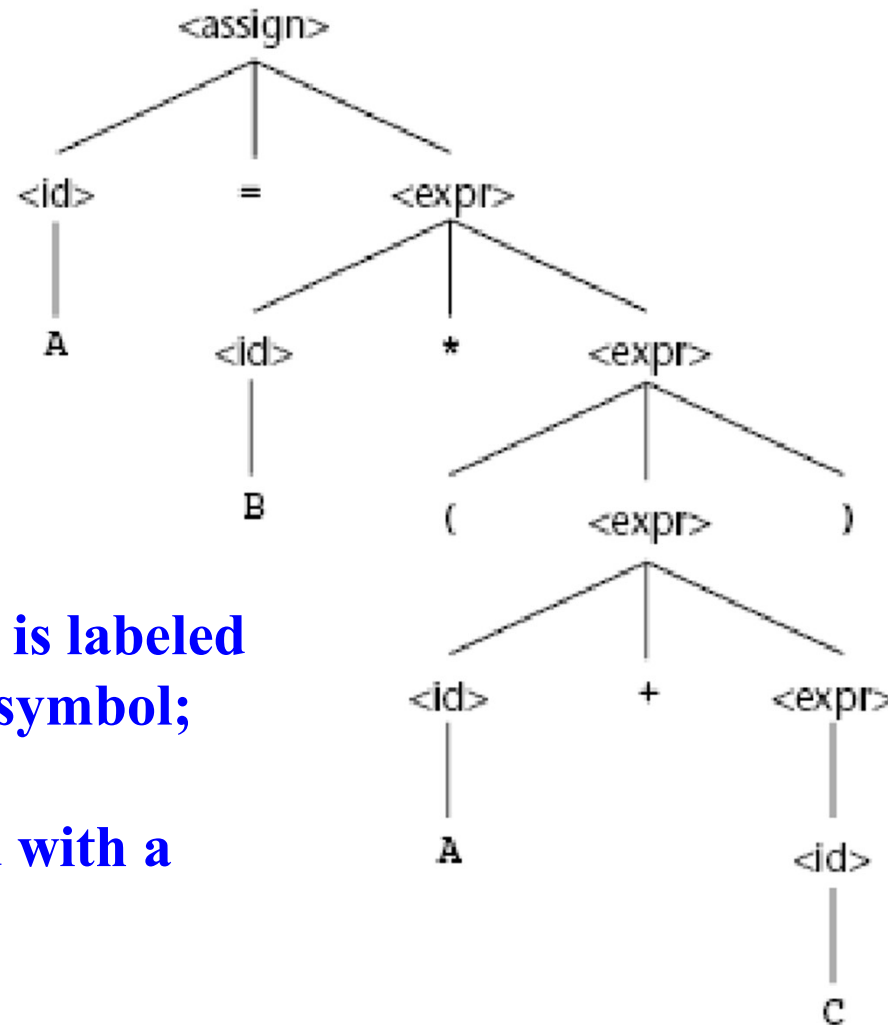
Parse Tree

- A hierarchical representation of a derivation

Figure 3.1

A parse tree for the
simple statement

$A = B * (A + C)$



Every internal node is labeled with a nonterminal symbol;

Every leaf is labeled with a terminal symbol.

Ambiguity in Grammars

- A grammar is *ambiguous* if and only if it generates a sentential form that has two or more distinct parse trees

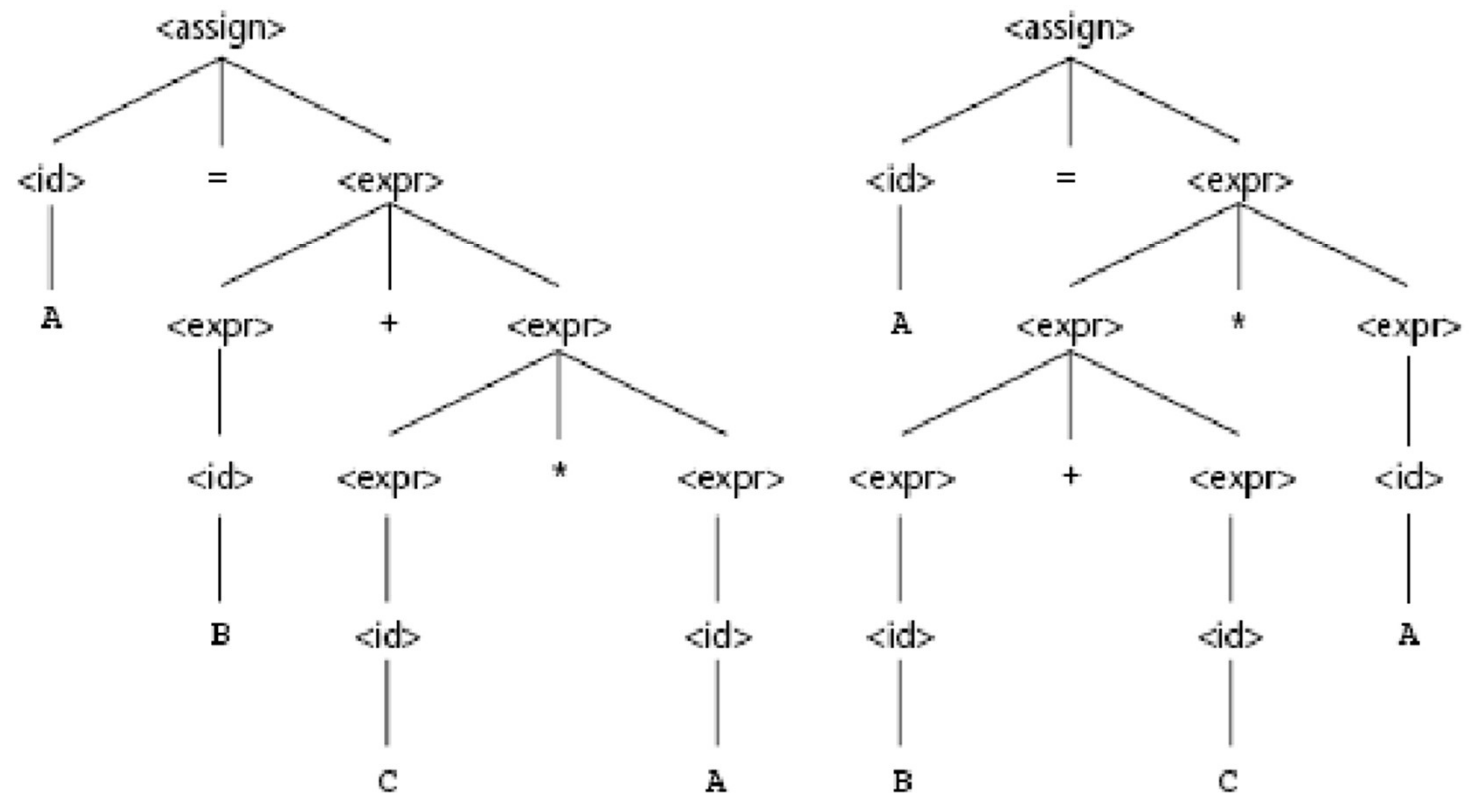
EXAMPLE 3.3

An Ambiguous Grammar for Simple Assignment Statements

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


Figure 3.2

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



Operator which is generated lower in parse tree, has precedence over operators which are generated higher in the parse tree.

An Unambiguous Expression Grammar

EXAMPLE 3.4

An Unambiguous Grammar for Expressions

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

Using the grammar in Example 3.4,

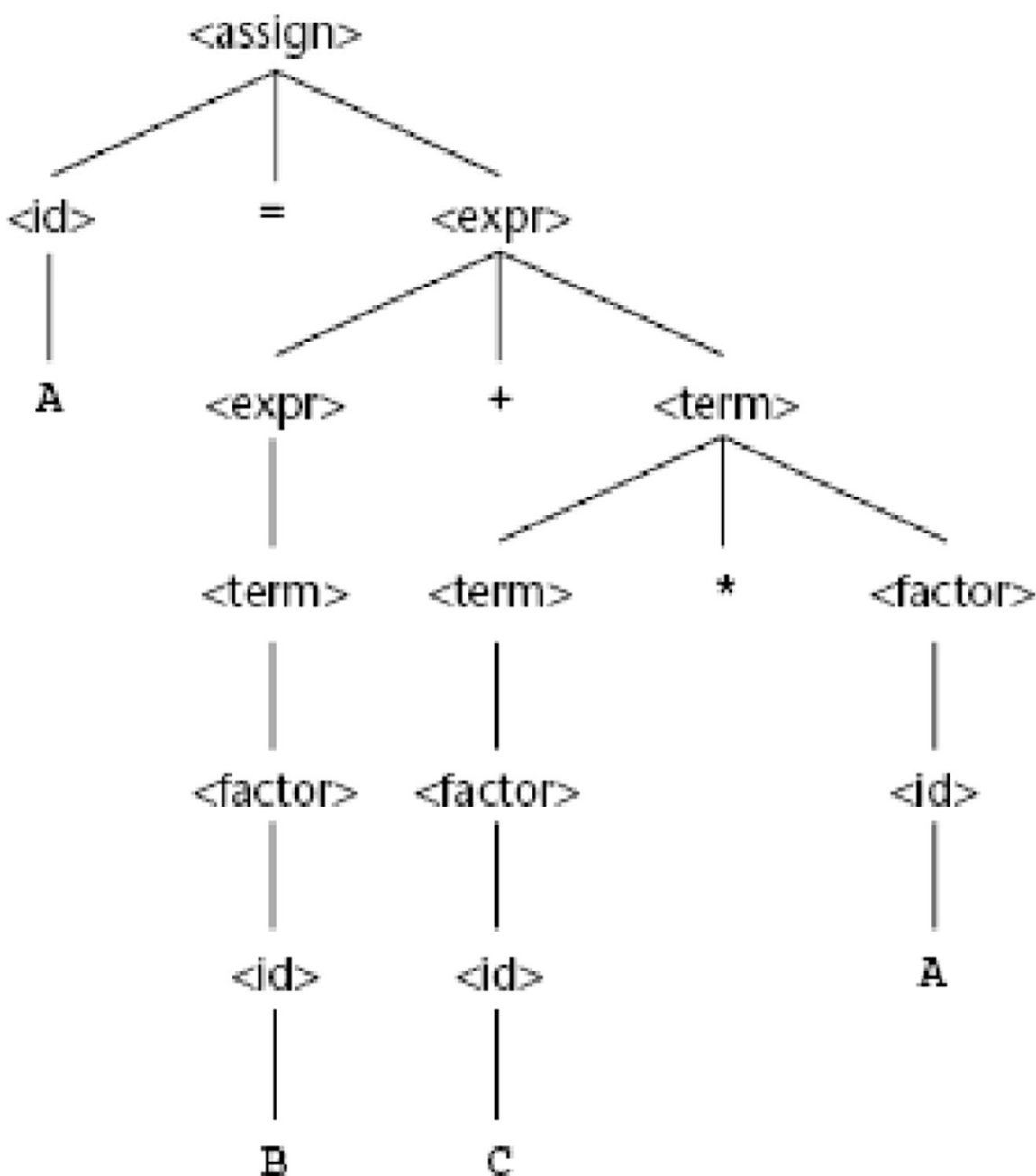
$$A = B + C * A$$

is derived as:

```
<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
```

Figure 3.3

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



Associativity of Operators

Associativity: A semantic rule to specify which operator should have precedence.

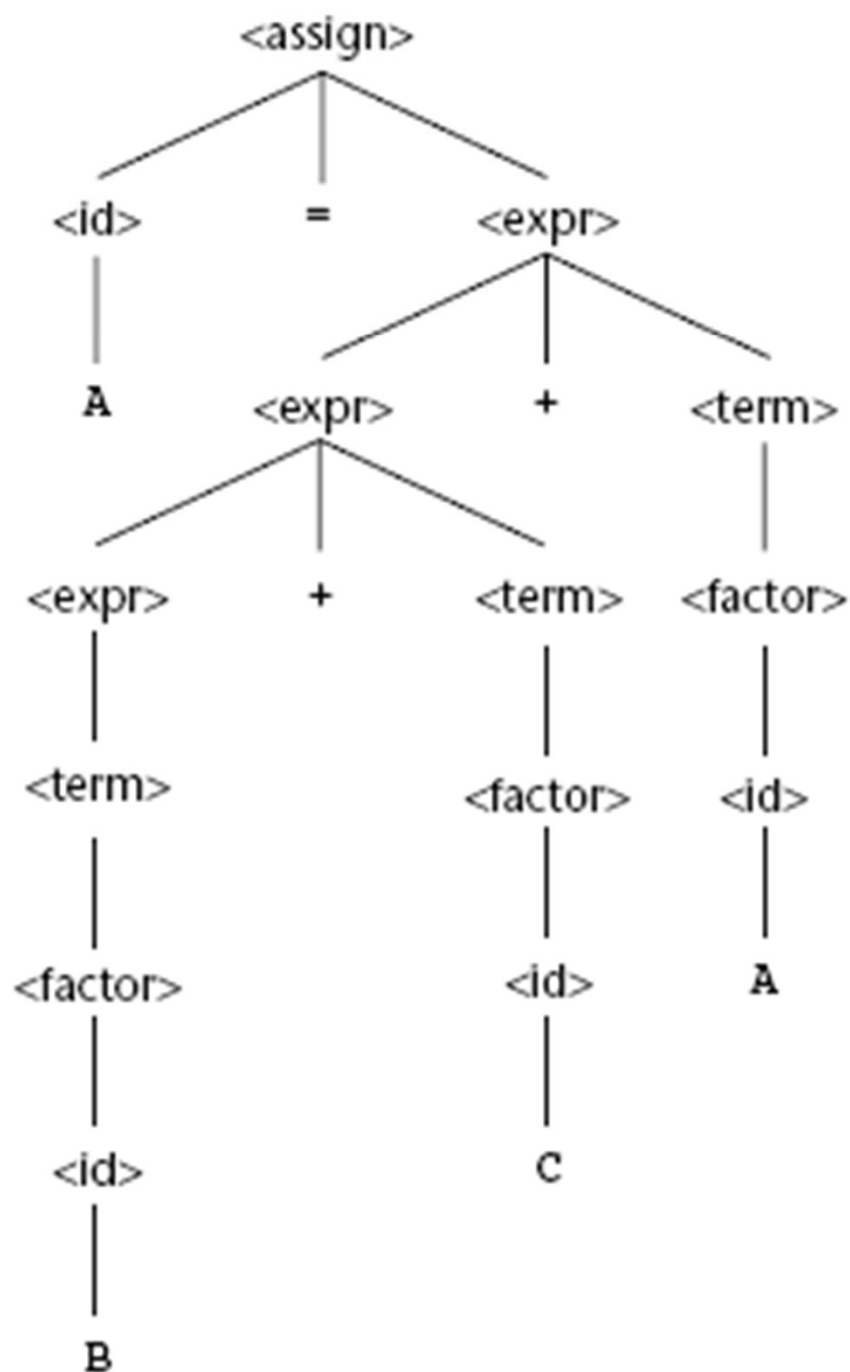
Example: Parse tree for

$$A = B + C + A$$

is shown in Figure 3.4, by using grammar in Example 3.4.

Figure 3.4

A parse tree for $A = B + C + A$
illustrating the
associativity of
addition



-
- Integer addition is associative, i.e.,
 $(A + B) + C = A + (B + C)$
 - Floating-point addition is not associative.
e.g., $1.0 + 1.0 + \dots + 1.0 + 1.0 \times 10^7 = 1.0000001 \times 10^7$

$$\underbrace{\hspace{10em}}_{11}$$

$$1.0 \times 10^7 + \underbrace{1.0 + 1.0 + \dots + 1.0}_{11} = 1.0000000 \times 10^7$$

- Subtraction and division are not associate.

Left Recursive vs Right Recursive

- *Left recursive*: When a grammar rule has its LHS also appearing at the beginning of its RHS, the rule is said to be left recursive. e.g., $N = N a$
 - Left recursive specifies left associativity. Rules in Example 3.4 make both addition and multiplication left recursive.
 - Left recursive disallows the use of some important syntax analysis algorithms.
- *Right recursive*: If LHS appears at the right end of RHS, the rule is said to be right recursive. Ex.1, $N = a N$

Ex.2 BNF to describe exponentiation

$\langle \text{factor} \rangle \rightarrow \langle \text{exp} \rangle ** \langle \text{factor} \rangle \mid \langle \text{exp} \rangle$

$\langle \text{exp} \rangle \rightarrow (\langle \text{expr} \rangle) \mid \langle \text{id} \rangle$

3.3.2 Extended BNF (EBNF)

- BNF:
 - recursion for iteration
 - nonterminals for grouping
- EBNF: additional metacharacters
 - { } for a series of zero or more
 - () for a list, must pick one
 - [] for an optional list; pick none or one

Three extensions in EBNF

- First, optional parts of an RHS are placed in brackets [].
e.g., `<if_stmt> → if (<expr>) <stmt> [else <stme>]`
- Second, repetitions (0 or more) are placed inside braces { }
e.g., `<ident_list> → <identifier> {, <identifier> }`
- Third, alternative parts of RHSs are placed inside parentheses and separated via vertical bars
e.g., `<term> → <term> (* | / | %) <factor>`

BNF and EBNF

- **BNF**
 $\langle \text{expr} \rangle \rightarrow \langle \text{expr} \rangle + \langle \text{term} \rangle$
 $\quad \quad \quad | \langle \text{expr} \rangle - \langle \text{term} \rangle \quad | \langle \text{term} \rangle$
 $\langle \text{term} \rangle \rightarrow \langle \text{term} \rangle * \langle \text{factor} \rangle$
 $\quad \quad \quad | \langle \text{term} \rangle / \langle \text{factor} \rangle \quad | \langle \text{factor} \rangle$
 $\langle \text{factor} \rangle \rightarrow \langle \text{exp} \rangle ** \langle \text{factor} \rangle \quad | \langle \text{exp} \rangle$
 $\langle \text{exp} \rangle \rightarrow (\langle \text{expr} \rangle) \quad | \langle \text{id} \rangle$

- **EBNF**

$\langle \text{expr} \rangle \rightarrow \langle \text{term} \rangle \{ (+ \mid -) \langle \text{term} \rangle \}$
 $\langle \text{term} \rangle \rightarrow \langle \text{factor} \rangle \{ (* \mid /) \langle \text{factor} \rangle \}$
 $\langle \text{factor} \rangle \rightarrow \langle \text{exp} \rangle \{ ** \langle \text{exp} \rangle \}$
 $\langle \text{exp} \rangle \rightarrow (\langle \text{expr} \rangle) \quad | \langle \text{id} \rangle$

Recent Variations in EBNF

- Alternative RHSs are put on separate lines
- Use of a colon ($::=$) instead of \Rightarrow
- Use of $_{\text{opt}}$ for optional parts
- Use of $_{\text{oneof}}$ for choices

e.g., AssignmentOperator \rightarrow one of $=$ $*=$
 $/=$ $\%=$ $+=$ $-=$ $<<=$ $>>=$ $\&=$ $|=$ $\hat{=}$

3.4 Attribute Grammars (AGs)

- An **attribute grammar** is a device used to describe more of the structure of a programming language than can be described with a CFG. It is an extension to a CFG.
- Attribute grammars (AGs) have additions to CFGs to carry some semantic info on parse tree nodes

3.4.1 Static Semantics

- The **static semantics** defines restrictions on the structure of valid texts that are hard or impossible to express in standard syntactic formalisms. For compiled languages, static semantics essentially include those semantic rules that can be checked at compile time.

Examples include checking that every identifier is declared before it is used (in languages that require such declarations) or that the labels on the arms of a case statement are distinct.

Context-free grammars (equivalent to BNF) cannot describe all of the syntax of programming languages

For example, in Java, a floating-point value cannot be assigned to an integer type variable, although the opposite is legal.

This restriction can be specified in BNF, but it requires additional nonterminals and rules.

If all of the typing rules of Java were specified in BNF, the grammar would be too large to be useful because the grammar determines the syntax analyzer.

3.4.2 Basic Concepts

- **Attribute grammars** are context-free grammars to which have been added attributes, attribute computation functions and predicate functions.
- **Attributes** are associated with grammar symbols (the terminal and nonterminal symbols), are similar to variables that they can have values assigned to them.
- **Attribute computation functions** (also called semantic functions) are associated with grammar rules. They are used to specify how attribute values are computed.
- **Predicate functions**, which state the static semantic rules of languages, are associated with grammar rules.

3.4.3 Attribute Grammars Defined

- X : grammar symbol

$A(X)$: a set of attributes associated to X . $A(X)$ consists of two disjoint sets $S(X)$ and $I(X)$.

$S(X)$: **Synthesized attributes**, are used to pass semantic information up a parse tree.

$I(X)$: **Inherited attributes**, are used to pass semantic information down and across a parse tree.

- Let $X_0 \rightarrow X_1 \dots X_n$ be a rule.

Functions of the form $S(X_0) = f(A(X_1), \dots, A(X_n))$ define *synthesized attributes* of X_0 . The value of a synthesized attribute on a parse tree node depends only on that node's children node.

Functions of the form $I(X_j) = f(A(X_0), \dots, A(X_n))$, for $1 \leq j \leq n$, define *inherited attributes*. The value of a inherited attribute on a parse tree node depends only on that node's parent node, and those of its sibling nodes.

Attribute Grammars Defined (cont'd)

- A **Predicate function** has the form of a Boolean expression on the union of the attribute set $\{A(X_0), \dots, A(X_n)\}$ and a set of literal attribute values.
- **Intrinsic attributes**: synthesised attributes of leaf nodes whose values are determined outside the parse tree.

If all the attribute values in a parse tree have been computed, the tree is said to be **fully attributed**. Although in practice it is not always done this way, it is convenient to think of attribute values as being computed after the complete unattributed parse tree has been constructed by the compiler.

3.4.5 Attribute Grammars: An Example

This example illustrates how an AG can be used to check the type rules of a simple assignment statement.

- A, B, C : variable name, they can be **int** or **real**
- RHS of the assignment can be a variable or an expression.
- When there are two variables on RHS, they need not be same type.
- When operand types are not the same, the type of the expression is always **real**.
- When operand types are the same, the type of the expression is that of operand.
- The type of LHS must match the type of RHS.
- The types of operands in RHS can be mixed, but the assignment is valid only if the target and the value resulting from RHS have the same type.

Attribute Grammars: An Example (cont'd)

- Syntax portion of this example AG is:

$\langle \text{assign} \rangle \rightarrow \langle \text{var} \rangle = \langle \text{expr} \rangle$

$\langle \text{expr} \rangle \rightarrow \langle \text{var} \rangle + \langle \text{var} \rangle \mid \langle \text{var} \rangle$

$\langle \text{var} \rangle \rightarrow A \mid B \mid C$

- Attributes for the nonterminals in this example AG are:

actual_type: synthesized attribute for nonterminals $\langle \text{var} \rangle$ and $\langle \text{expr} \rangle$. It is used to store actual type, **int** or **real**. In the case of a variable, the actual type is intrinsic. In the case of an expression, it is determined from the actual types of the child node or children nodes of the $\langle \text{expr} \rangle$ nonterminal.

expected_type: inherited attribute for nonterminal $\langle \text{expr} \rangle$. It is used to store the type, **int** or **real**, for the expression, as determined by the type of the variable on LHS.

Attribute Grammars: An Example (cont'd)

EXAMPLE 3.6

An Attribute Grammar for Simple Assignment Statements

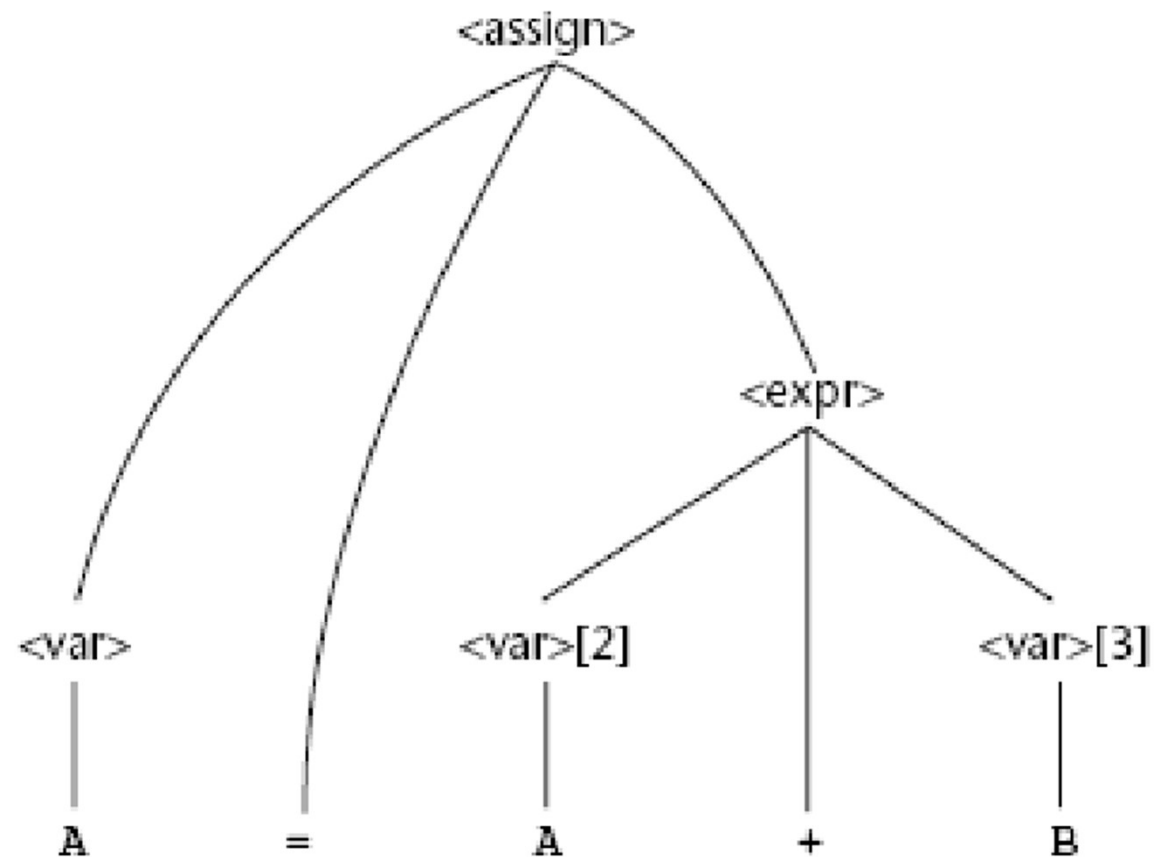
1. Syntax rule: $\langle \text{assign} \rangle \rightarrow \langle \text{var} \rangle = \langle \text{expr} \rangle$
Semantic rule: $\langle \text{expr} \rangle.\text{expected_type} \leftarrow \langle \text{var} \rangle.\text{actual_type}$
2. Syntax rule: $\langle \text{expr} \rangle \rightarrow \langle \text{var} \rangle[2] + \langle \text{var} \rangle[3]$
Semantic rule: $\langle \text{expr} \rangle.\text{actual_type} \leftarrow$
if $(\langle \text{var} \rangle[2].\text{actual_type} = \text{int})$ and
 $(\langle \text{var} \rangle[3].\text{actual_type} = \text{int})$
 then int
 else real
 end if
Predicate: $\langle \text{expr} \rangle.\text{actual_type} == \langle \text{expr} \rangle.\text{expected_type}$
3. Syntax rule: $\langle \text{expr} \rangle \rightarrow \langle \text{var} \rangle$
Semantic rule: $\langle \text{expr} \rangle.\text{actual_type} \leftarrow \langle \text{var} \rangle.\text{actual_type}$
Predicate: $\langle \text{expr} \rangle.\text{actual_type} == \langle \text{expr} \rangle.\text{expected_type}$
4. Syntax rule: $\langle \text{var} \rangle \rightarrow A \mid B \mid C$
Semantic rule: $\langle \text{var} \rangle.\text{actual_type} \leftarrow \text{look-up}(\langle \text{var} \rangle.\text{string})$

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

Attribute Grammars: An Example (cont'd)

Figure 3.6

A parse tree for
 $A = A + B$



Attribute Grammars (continued)

- Computing the attribute values of a parse tree is sometimes called **decorating the parse tree**.
- How are attribute values computed?
 - 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 that must be used.

Attribute Grammars (continued)

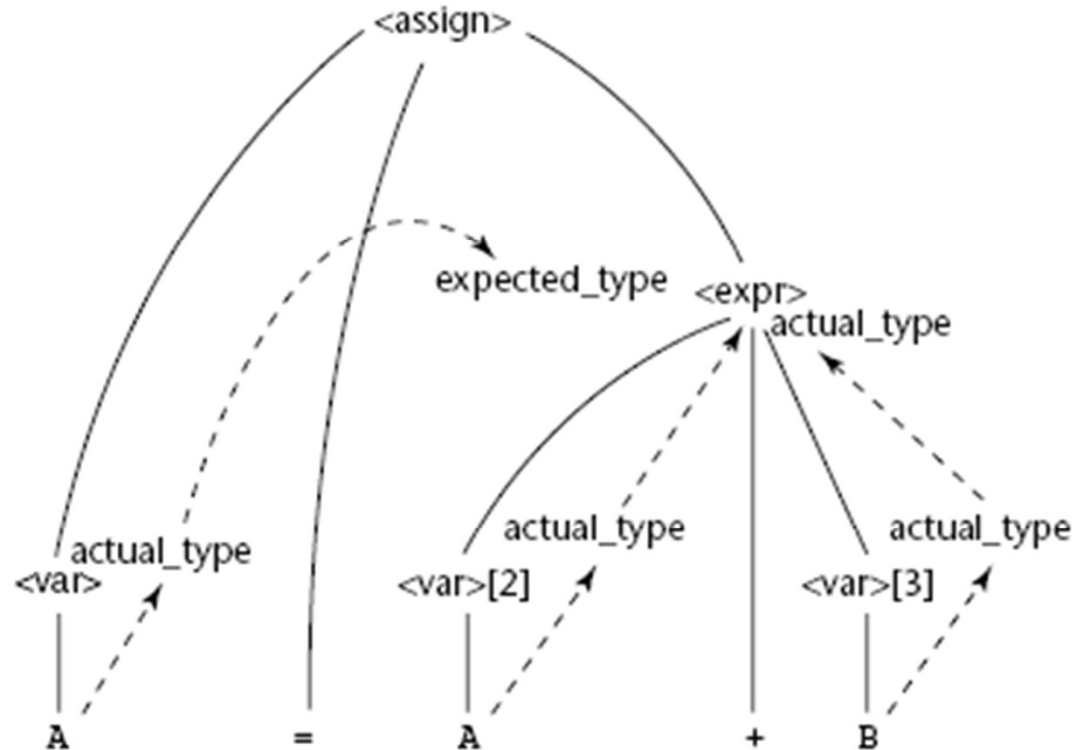
Process to calculate attributes:

1. $\langle \text{var} \rangle.\text{actual_type} \leftarrow \text{look-up}(A)$ (Rule 4)
2. $\langle \text{expr} \rangle.\text{expected_type} \leftarrow \langle \text{var} \rangle.\text{actual_type}$
(Rule 1)
3. $\langle \text{var} \rangle[2].\text{actual_type} \leftarrow \text{lookup}(A)$ (Rule 4)
 $\langle \text{var} \rangle[3].\text{actual_type} \leftarrow \text{lookup}(B)$ (Rule 4)
4. $\langle \text{expr} \rangle.\text{actual_type} \leftarrow \text{either int or real}$
(Rule 2)
5. $\langle \text{expr} \rangle.\text{expected_type} == \langle \text{expr} \rangle.\text{actual}$ is
either TRUE or FALSE (Rule 2)

Attribute Grammars (continued)

Figure 3.7

The flow of attributes
in the tree

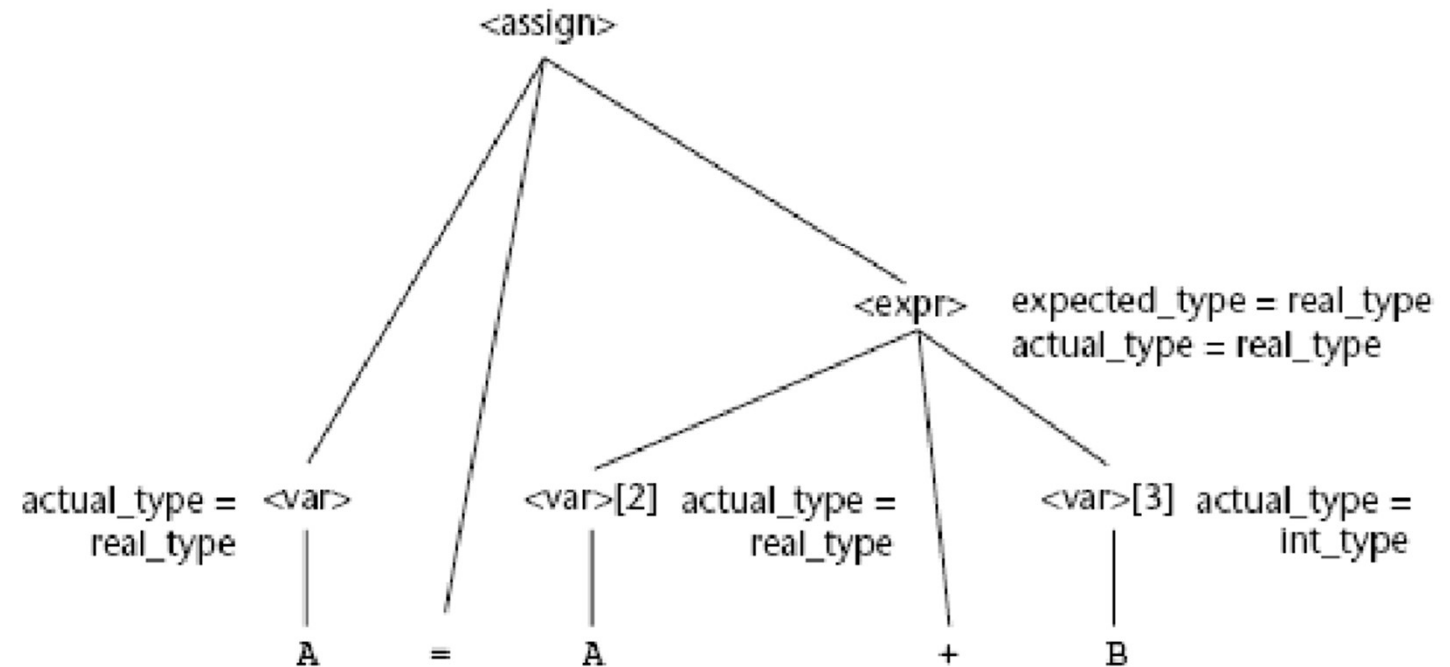


- Solid lines: parse tree
- Dashed lines: attribute flow in tree

Attribute Grammars (continued)

Figure 3.8

A fully attributed
parse tree



Summary

- BNF and context-free grammars are equivalent meta-languages
 - Well-suited for describing the syntax of programming languages
- An attribute grammar is a descriptive formalism that can describe both the syntax and the semantics of a language