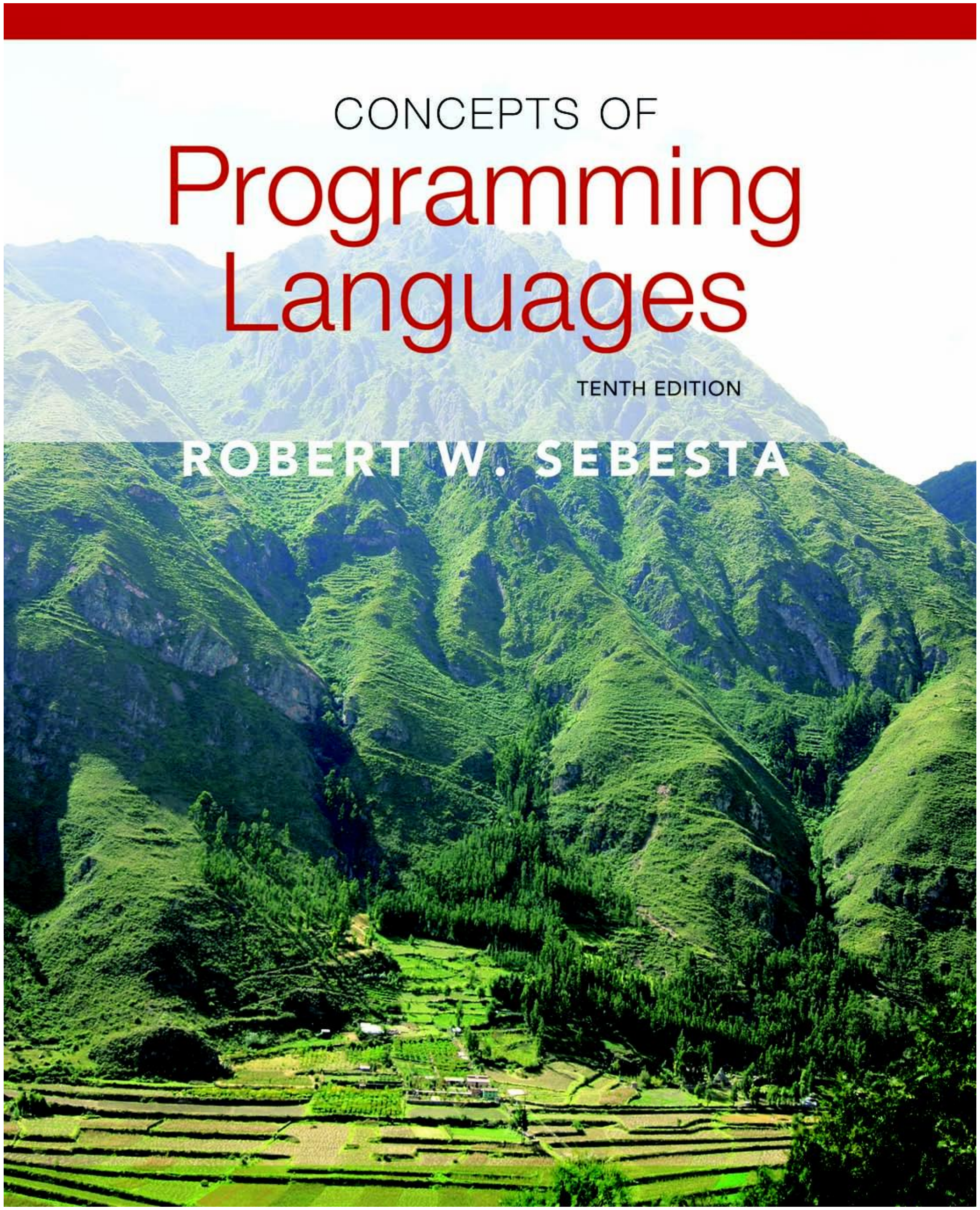
Chapter 3

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

Introduction

- **Syntax**: the form or structure of the expressions, statements, and program units
- **Semantics**: the meaning of the expressions, statements, and program units
- Syntax and semantics provide a language's definition
  - Users of a language definition
    - Other language designers
    - Implementers (interpret, compile)
    - Programmers (the users of the language)

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 (e.g., `*`, `sum`, `begin`)
- A *token* is a category of lexemes (e.g., identifier)

The General Problem of Describing Syntax: Terminology

- index = 3 * count + 5;

Lexeme	Token
index, count	identifier
=, *, +	Operator
3, 5	int_literal

Formal Definition of Languages

- How can you define a PL?
- **Recognizers**
  - A **recognition device** reads input strings over the alphabet of the language and decides whether the input strings **belong** to the **language**
  - Example: syntax analysis part of a compiler
    - Detailed discussion of syntax analysis (**parser**) appears in Chapter 4
  - Limited usage because trial and error mode; generators preferred
- **Generators**
  - A device that **generates sentences** of a **language**
  - One can determine if the syntax of a particular sentence is syntactically **correct** by **comparing** it to the **structure of the generator**

BNF and Context-Free Grammars

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

Quiz

- What is the difference between lexeme and token?

BNF Fundamentals

- What is a **metalanguage**?
  - A language that is used to describe other languages.

- **BNF** is a metalanguage for **programming languages**.

BNF Fundamentals

- In **BNF**, **abstractions** are used to represent classes of syntactic structures--they act like syntactic variables (also called *nonterminal symbols*, or just *terminals*)
- *Terminals* are
  - lexemes
  - or tokens
- A 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**

BNF Fundamentals (continued)

- Nonterminals are often enclosed in angle brackets
  - Examples of BNF rules:

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

    <ident_list> → identifier | identifier, **<ident_list>**

- Grammar:
  - a finite <u>non-empty</u> set of rules

- A *start symbol* is a special element of the nonterminals of a grammar

BNF Rules

- An abstraction (or nonterminal symbol) can have more than one RHS

```
<stmt> → <single_stmt>
       | begin <stmt_list> end
```

Describing Lists

- **Syntactic lists** are described using **recursion**

```
<ident_list> → ident
             | ident, <ident_list>
```

- A **derivation** is a repeated application of rules,
  - starting with the **start symbol** and ending with a **sentence** (<u>all terminal symbols</u>)

A Grammar for a Small Language

```
<program> → begin <stmts> end
<stmts> → <stmt> | <stmt> ; <stmts>
<stmt> → <var> = <expr>
<var> → a | b | c | d
<expr> → <term> + <term> | <term> - <term>
<term> → <var> | const
```

An Example Derivation

```
<program> => <stmts>
          => <stmt>
          => <var> = <expr>
          => a = <expr>
          => a = <term> + <term>
          => a = <var> + <term>
          => a = b + <term>
          => a = b + const
```

```
<program> → <stmts>
   <stmts> → <stmt> | <stmt> ; <stmts>
   <stmt> → <var> = <expr>
   <var> → a | b | c | d
   <expr> → <term> + <term> | <term> - <term>
   <term> → <var> | const
```

Derivations

- Every string of symbols in a derivation is a ***sentential form***
- A ***sentence*** is a **sentential form** that has <u>only terminal symbols</u>
- ***Leftmost derivation***
  - leftmost nonterminal in each sentential form is the one that is expanded

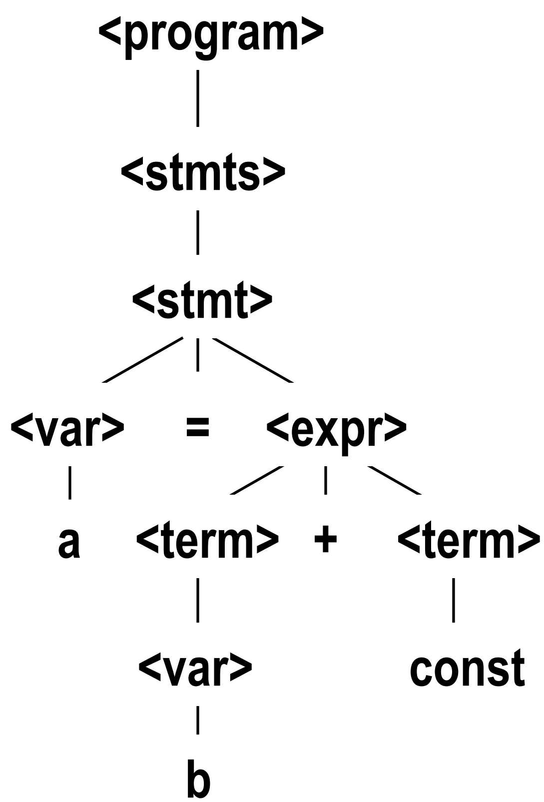
An Example Derivation

```
<program> => <stmts>
          => <stmt>
          => <var> = <expr>
          => a = <expr>
          => a = <term> + <term>
          => a = <var> + <term>
          => a = b + <term>
          => a = b + const
```

```
<program> → <stmts>
   <stmts> → <stmt> | <stmt> ; <stmts>
   <stmt> → <var> = <expr>
   <var> → a | b | c | d
   <expr> → <term> + <term> | <term> - <term>
   <term> → <var> | const
```

Parse Tree

- A hierarchical representation of a derivation

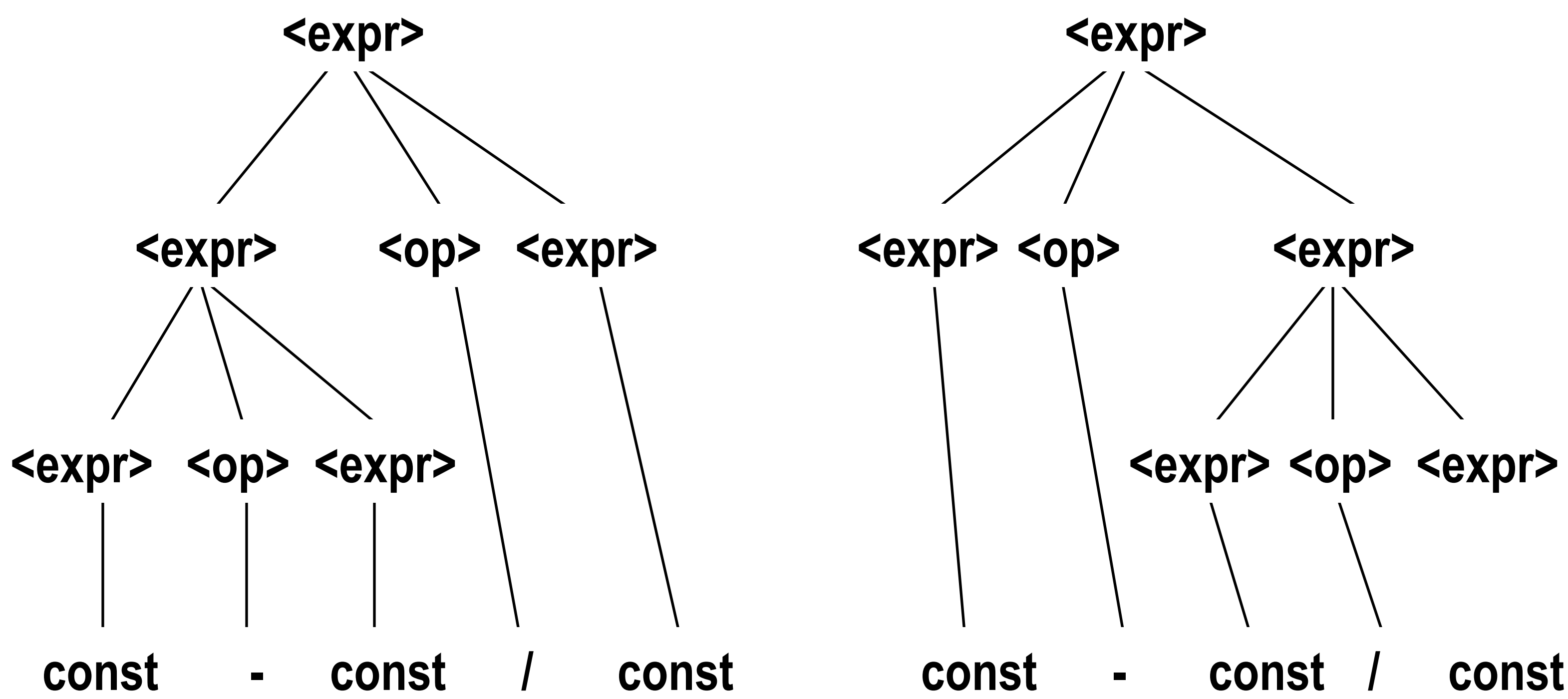


Ambiguity in Grammars

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

An Ambiguous Expression Grammar

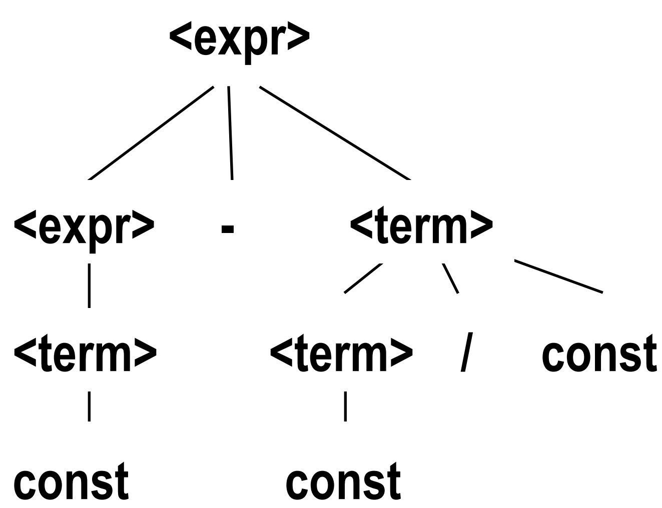
```
<expr> → <expr> <op> <expr>  |  const
<op> → /  |  -
```



An Unambiguous Expression Grammar

- If parse tree to **indicate** <u>precedence levels</u> of the operators

```
<expr> → <expr> - <term>  |  <term>
<term> → <term> / const| const
```



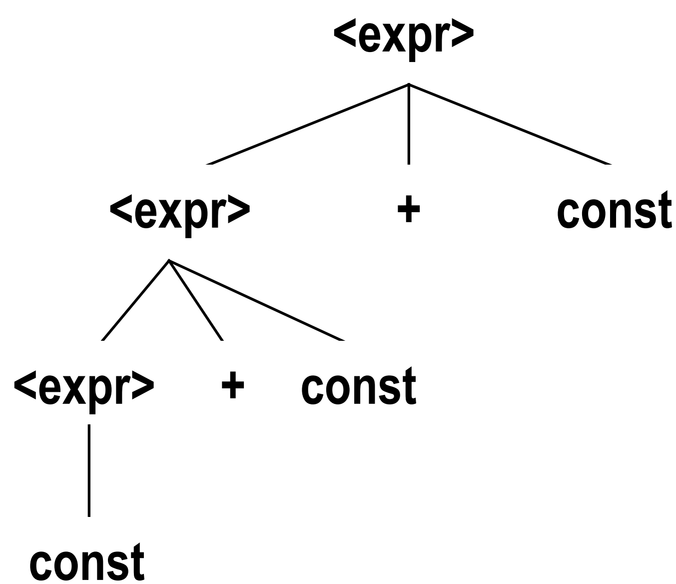
Quiz

- Prove that the following grammar is ambiguous:
  <A> → <C>
- <C> → <C> <op> <C> | <id>
- <op> → * | / | + | -
- <id> → a | b | c

Associativity of Operators

- Operator associativity can also be indicated by a grammar

```
<expr> -> <expr> + <expr> |  const  (ambiguous)
<expr> -> <expr> + const  |  const  (unambiguous)
```



Associativity of Operators

- Integer addition
  (3+4) +5 = 3 + (4+5)

- Floating point addition
  (10^7 + 1) + … + 1 =? 10^7 + (1+ ... + 1)
  - 1.000000x10^7 =? 1.000001 x 10^7

Quiz

- Rewrite the following BNF to give / precedence over * .

<assign> → <id> = <expr>

<id> → A | B | C

<expr> → <expr> / <term> | <term>

<term> → <term> * <factor>| <factor>

<factor> → (<expr>) | <id>

B = A / (A / B) * C

Quiz

- Using the grammar above, show a parse tree and a leftmost derivation for the following statement:

  B = A / (A / B) * C

Quiz

- Describe, in English, the language defined by the following grammar:

$$S \rightarrow a\,S \mid S\,b \mid a \mid b$$

Extended BNF

- **Optional parts** are placed in brackets [ ]

  `<proc_call> -> ident [(<expr_list>)]`

- **Alternative parts:**
  - parentheses

  `<term> → <term> (+|-) const`

- Repetitions
  - (0 or more) are placed inside braces { }

  `<ident> → letter {letter|digit}`

BNF and EBNF

- BNF

```
<expr> → <expr> + <term>
              | <expr> - <term>
              | <term>
 <term> → <term> * <factor>
              | <term> / <factor>
         | <factor>
```

- EBNF

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

Static Semantics

- Nothing to do with the meaning
- Context-free grammars (CFGs) cannot describe all of the syntax of programming languages
  - (e.g., variables must be declared before they are used)