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
    - 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)
Formal Definition of Languages

- **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 appears in Chapter 4

- **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 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)
  - Invented by John Backus to describe the syntax of Algol 58
  - BNF is equivalent to context-free grammars
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
Nonterminals are often enclosed in angle brackets

Examples of BNF rules:

- `<ident_list> → identifier | identifier, <ident_list>`
- `<if_stmt> → if <logic_expr> then <stmt>`

Grammar: a finite non-empty 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

\[ <\text{stmt}> \rightarrow <\text{single\_stmt}> \]

\[ \quad \mid \text{begin} <\text{stmt\_list}> \text{end} \]
Describing Lists

- Syntactic lists are described using recursion

\[<\text{ident\_list}> \rightarrow \text{ident} \]

\[\quad | \text{ident, } <\text{ident\_list}>\]

- A derivation is a repeated application of rules, starting with the start symbol and ending with a sentence (all terminal symbols)
An Example Grammar

\[
\begin{align*}
\langle \text{program} \rangle & \rightarrow \langle \text{stmts} \rangle \\
\langle \text{stmts} \rangle & \rightarrow \langle \text{stmt} \rangle \mid \langle \text{stmt} \rangle, \langle \text{stmts} \rangle \\
\langle \text{stmt} \rangle & \rightarrow \langle \text{var} \rangle = \langle \text{expr} \rangle \\
\langle \text{var} \rangle & \rightarrow a \mid b \mid c \mid d \\
\langle \text{expr} \rangle & \rightarrow \langle \text{term} \rangle + \langle \text{term} \rangle \mid \langle \text{term} \rangle - \langle \text{term} \rangle \\
\langle \text{term} \rangle & \rightarrow \langle \text{var} \rangle \mid \text{const}
\end{align*}
\]
An Example Grammar

\[
\begin{align*}
\text{<program> } &\rightarrow \text{ <stmts>} \\
\text{<stmts> } &\rightarrow \text{ <stmt> } \mid \text{ <stmt> ; <stmts>} \\
\text{<stmt> } &\rightarrow \text{ <var> = <expr>} \\
\text{<var> } &\rightarrow \text{ a \mid b \mid c \mid d} \\
\text{<expr> } &\rightarrow \text{ <term> + <term> } \mid \text{ <term> - <term>} \\
\text{<term> } &\rightarrow \text{ <var> } \mid \text{ const}
\end{align*}
\]

Write a dummy program based on the grammar
An Example Grammar

\[
\text{a} = \text{b} + \text{const}
\]
An Example Derivation

\[
\begin{align*}
\langle \text{program} \rangle & \Rightarrow \langle \text{stmts} \rangle \Rightarrow \langle \text{stmt} \rangle \\
& \Rightarrow \langle \text{var} \rangle = \langle \text{expr} \rangle \\
& \Rightarrow a = \langle \text{expr} \rangle \\
& \Rightarrow a = \langle \text{term} \rangle + \langle \text{term} \rangle \\
& \Rightarrow a = \langle \text{var} \rangle + \langle \text{term} \rangle \\
& \Rightarrow a = b + \langle \text{term} \rangle \\
& \Rightarrow a = b + \text{const}
\end{align*}
\]
Derivations

- Every string of symbols in a derivation is a sentential form
- A sentence is a sentential form that has only terminal symbols
- A leftmost derivation is one in which the leftmost nonterminal in each sentential form is the one that is expanded
- A derivation may be neither leftmost nor rightmost
Parse Tree

- A hierarchical representation of a derivation

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

\[
\begin{align*}
<\text{expr}> & \rightarrow <\text{expr}> \ <\text{op}> \ <\text{expr}> \mid \text{const} \\
<\text{op}> & \rightarrow / \mid - \\
\end{align*}
\]
An Ambiguous Expression Grammar

\[<\text{expr}> \rightarrow <\text{expr}> \ <\text{op}> \ <\text{expr}> \mid \text{const} \]

\[<\text{op}> \rightarrow / \mid - \]
An Unambiguous Expression Grammar

- If we use the parse tree to indicate precedence levels of the operators, we cannot have ambiguity

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

<expr>
If we use the parse tree to indicate precedence levels of the operators, we cannot have ambiguity

\[<\text{expr}> \rightarrow <\text{expr}> - <\text{term}> \mid <\text{term}>\]
\[<\text{term}> \rightarrow <\text{term}> / \text{const} \mid \text{const}\]
Operator associativity can also be indicated by a grammar

\[ <\text{expr}> -> <\text{expr}> + <\text{expr}> \mid \text{const} \quad \text{(ambiguous)} \]
\[ <\text{expr}> -> <\text{expr}> + \text{const} \mid \text{const} \quad \text{(unambiguous)} \]
Unambiguous Grammar for Selector

- **Java if-then-else grammar**

  \[
  \begin{align*}
  <\text{if_stmt}> & \rightarrow \text{if} (<\text{logic_expr}>) <\text{stmt}> \\
  & \quad \mid \text{if} (<\text{logic_expr}>) <\text{stmt}> \text{ else } <\text{stmt}>
  \end{align*}
  \]

  Ambiguous!

  - An unambiguous grammar for if-then-else

  \[
  \begin{align*}
  <\text{stmt}> & \rightarrow <\text{matched}> \mid <\text{unmatched}>
  \\
  <\text{matched}> & \rightarrow \text{if} (<\text{logic_expr}>) <\text{stmt}>
  \\
  & \quad \mid \text{a non-if statement}
  \\
  <\text{unmatched}> & \rightarrow \text{if} (<\text{logic_expr}>) <\text{stmt}>
  \\
  & \quad \mid \text{if} (<\text{logic_expr}>) <\text{matched}> \text{ else } <\text{unmatched}>
  \end{align*}
  \]