Chapter 10: Compilers and Language Translation

Invitation to Computer Science, C++ Version, Third Edition
Objectives

In this chapter, you will learn about:

- The compilation process
  - Phase I: Lexical analysis
  - Phase II: Parsing
  - Phase III: Semantics and code generation
  - Phase IV: Code optimization
Introduction

- High-level language instructions must be translated into machine language prior to execution

- Compiler
  
  - A piece of system software that translates high-level languages into machine language
Goals of a compiler when performing a translation

- Correctness

- Producing a reasonably efficient and concise machine language code
Figure 10.1
General Structure of a Compiler
The Compilation Process

- **Phase I: Lexical analysis**
  - Compiler examines the individual characters in the source program and groups them into syntactical units called tokens

- **Phase II: Parsing**
  - The sequence of tokens formed by the scanner is checked to see whether it is syntactically correct
The Compilation Process (continued)

- **Phase III: Semantic analysis and code generation**
  - The compiler analyzes the meaning of the high-level language statement and generates the machine language instructions to carry out these actions

- **Phase IV: Code optimization**
  - The compiler takes the generated code and sees whether it can be made more efficient
Figure 10.2
Overall Execution Sequence on a High-level Language Program
The Compilation Process (continued)

- Final step
  - Object program is written to an object file

- Source program
  - Original high-level language program

- Object program
  - Machine language translation of the source program
Phase I: Lexical Analysis

- **Lexical analyzer**
  - The program that performs lexical analysis
  - More commonly called a scanner

- **Job of lexical analyzer**
  - Group input characters into tokens
    - **Tokens**: syntactical units that are treated as single, indivisible entities for the purposes of translation
  - Classify tokens according to their type
<table>
<thead>
<tr>
<th>Token Type</th>
<th>Classification Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>symbol</td>
<td>1</td>
</tr>
<tr>
<td>number</td>
<td>2</td>
</tr>
<tr>
<td>=</td>
<td>3</td>
</tr>
<tr>
<td>+</td>
<td>4</td>
</tr>
<tr>
<td>-</td>
<td>5</td>
</tr>
<tr>
<td>;</td>
<td>6</td>
</tr>
<tr>
<td>==</td>
<td>7</td>
</tr>
<tr>
<td>if</td>
<td>8</td>
</tr>
<tr>
<td>else</td>
<td>9</td>
</tr>
<tr>
<td>(</td>
<td>10</td>
</tr>
<tr>
<td>)</td>
<td>11</td>
</tr>
</tbody>
</table>

Figure 10.3
Typical Token Classifications
Phase I: Lexical Analysis (continued)

- Input to a scanner
  - A high-level language statement from the source program

- Scanner’s output
  - A list of all the tokens in that statement
  - The classification number of each token found
Phase II: Parsing
Introduction

- Parsing phase

- A compiler determines whether the tokens recognized by the scanner are a syntactically legal statement

- Performed by a parser
Phase II: Parsing
Introduction (continued)

- Output of a parser
  - A parse tree, if such a tree exists
  - An error message, if a parse tree cannot be constructed

- Successful construction of a parse tree is proof that the statement is correctly formed
Example

- High-level language statement: \( a = b + c \)
Grammars, Languages, and BNF

- Syntax
  - The grammatical structure of the language
- The parser must be given the syntax of the language
- BNF (Backus-Naur Form)
  - Most widely used notation for representing the syntax of a programming language
Grammars, Languages, and BNF (continued)

- In BNF
  - The syntax of a language is specified as a set of rules (also called productions)
  - A grammar
    - The entire collection of rules for a language
  - Structure of an individual BNF rule
    left-hand side ::= “definition”
Grammars, Languages, and BNF (continued)

- BNF rules use two types of objects on the right-hand side of a production
  - Terminals
    - The actual tokens of the language
    - Never appear on the left-hand side of a BNF rule
  - Nonterminals
    - Intermediate grammatical categories used to help explain and organize the language
    - Must appear on the left-hand side of one or more rules
Grammars, Languages, and BNF (continued)

- Goal symbol
  - The highest-level nonterminal
  - The nonterminal object that the parser is trying to produce as it builds the parse tree

- All nonterminals are written inside angle brackets
Fundamental rule of parsing

- By repeated applications of the rules of the grammar
  - If a parser can convert the sequence of input tokens into the goal symbol, then that sequence of tokens is a syntactically valid statement of the language
  - If the parser cannot convert the input tokens into the goal symbol, then this is not a syntactically valid statement of the language
Parsing Concepts and Techniques (continued)

- One of the biggest problems in building a compiler is designing a grammar that:
  - Includes every valid statement that we want to be in the language
  - Excludes every invalid statement that we do not want to be in the language
Parsing Concepts and Techniques (continued)

- Another problem in constructing a compiler: designing a grammar that is not ambiguous

  - An ambiguous grammar allows the construction of two or more distinct parse trees for the same statement
Phase III: Semantics and Code Generation

- Semantic analysis
  - The compiler makes first pass over parse tree to determine whether all branches of the tree are semantically valid
    - If they are valid, the compiler can generate machine language instructions
    - If not, there is a semantic error; machine language instructions are not generated
Phase III: Semantics and Code Generation (continued)

- Code generation
  - Compiler makes the second pass over the parse tree to produce the translated code
Phase IV: Code Optimization

- Two types of optimization
  - Local
  - Global
- Local optimization
  - The compiler looks at a very small block of instructions and tries to determine how it can improve the efficiency of this local code block
  - Relatively easy; included as part of most compilers
Phase IV: Code Optimization (continued)

- Examples of possible local optimizations
  - Constant evaluation
  - Strength reduction
  - Eliminating unnecessary operations
Phase IV: Code Optimization (continued)

- Global optimization
  - The compiler looks at large segments of the program to decide how to improve performance
  - Much more difficult; usually omitted from all but the most sophisticated and expensive production-level “optimizing compilers”

- Optimization cannot make an inefficient algorithm efficient
Summary

- A compiler is a piece of system software that translates high-level languages into machine language.

- Goals of a compiler: correctness, and producing efficient and concise code.

- **Source program**: high-level language program.
Summary

- Object program: the machine language translation of the source program

- Phases of the compilation process
  - Phase I: Lexical analysis
  - Phase II: Parsing
  - Phase III: Semantic analysis and code generation
  - Phase IV: Code optimization